S.M. GAYNAZAROV
D.Y. SAIDOV

ALGORITMLAR VA BERILGANLAR TUZILMASI

O‘ZBEKISTON RESPUBLIKASI

OLIY TA’LIM, FAN VA INNOVATSIYALAR VAZIRLIGI

MIRZO ULUG‘BEK NOMIDAGI O‘ZBEKISTON MILLIY UNIVERSITETI

S.M.GAYNAZAROV, D.Y.SAIDOV

ALGORITMLAR VA BERILGANLAR TUZILMASI

1-qism

O‘QUV QO‘LLANMA

Toshkent

“Ma’rifat”

2023

UO‘K: 004.021(075.8)
KBK: 32.973.202ya73
G 15

Gaynazarov S.M., Saidov D.Y. Algoritmlar va berilganlar tuzilmasi. O‘quv qo‘llanma. –T.: “Ma’rifat”, 2023. 254 bet.

O‘quv qo‘llanma 5330100 – Kompyuter ilmlari va dasturlash texnologiyalari, 5330200 – Axborot tizimlari va texnologiyalari, 5330300 – Axborot xavfsizligi bakalavr ta‘lim yo‘nalishi talabalariga mo‘ljallangan. Ushbu o‘quv qo‘llanma algoritmlar va berilganlarning asosiy tushunchalari, algoritmlar tahlili va ularni baholash usullari, berilganlarning abstrakt turlari, saralash algoritmlari, qidirish algoritmlari bo‘yicha ma’lumotlarni o‘z ichiga qamrab olgan. Bundan tashqari ushbu o‘quv qo‘llanmadan ushbu fanni mustaqil o‘rganuvchilar ham foydalanishlari mumkin.

O‘quv qo‘llanmada mavzular bo‘yicha ma’ruza matnlari, har bir bobdan keyin nazorat savollari keltirilgan.

UO‘K: 004.021(075.8)
KBK: 32.973.202ya73
G 15

Taqrizchilar:

J.U.Sevinov – Toshkent davlat texnika universiteti “Axborotlarga ishlov berish va boshqarish tizimlari” kafedrasi mudiri, professor, texnika fanlari doktori

U.E.Adambayev – O‘zbekiston Milliy Universiteti matematika fakulteti “Algoritmlar va dasturlash texnologiyalari” kafedrasi dotsenti, fizika-matematika fanlari nomzodi

ISBN: 978-9943-9313-0-5

MUNDARIJA

Kirish

Ushbu o‘quv qo‘llanma dasturlashda berilganlar tuzilmalari va algoritmlardan foydalanishga bag‘ishlangan.

Berilganlar tuzilmalari berilganlarning kompyuter xotirasida (yoki diskda) qanday tashkil etilishini ifodalaydi. Algoritmlar esa ushbu tuzilmalar bilan turli amallarni bajarishni ta'minlaydi.

Berilganlar tuzilmalari va algoritmlar deyarli barcha kompyuter dasturlarida, shu jumladan eng oddiy dasturlarda ham qo‘llaniladi.

Kichik hajmdagi berilganlarga ega oddiy dasturlar uchun berilganlarni ifodalashning oddiy yechimi ham yetarli bo‘lishi mumkin. Ammo hech bo‘lmaganda o‘rtacha hajmdagi berilganlarni qayta ishlaydigan yoki noodatiy muammolarni hal qiladigan dasturlar uchun yanada murakkab usullar talab qilinadi. Bunday hollarda C# yoki C++ kabi dasturlash tilining sintaksisini bilishning o‘zi yetarli bo‘lmaydi.

Ushbu o‘quv qo‘llanma dasturlash tilini o‘rganganingizdan keyin bilish kerak bo‘lgan muhim tushunchalarni o‘z ichiga oladi.

Taqdim etilayotgan material oliy ta’lim muassasalarida informatika va axborot texnologiyalari va unga turdosh boshqa yo‘nalishlarda o‘qitishning ikkinchi yilida, talaba dasturlash asoslarini o‘zlashtirgandan so‘ng o‘qitiladi.

Berilganlarning abstrakt turi (BAT) tushunchasi berilganlar tuzilmasini tavsiflash uchun asos sifatida ishlatilgan. Berilganlarni tavsiflash va ular ustida bajariladigan amallar kodi C# dasturlash tilining obyektga yo‘naltirilgan dasturlash tamoyillarini qo‘llagan holda amalga oshirilgan. Asosiy algoritmlar mos keladigan berilganlar tuzilmasini tavsiflovchi sinf usullari sifatida taqdim etiladi.

Qo‘llanma ustida ishlashda biz birinchi navbatda ushbu fan bo‘yicha o‘quv dasturi doirasida materialning aniq va tushunarli tarzda taqdim etilishini ta'minlashga harakat qildik.

Ma’ruzada berilgan har bir mavzu bo‘yicha to‘liq dastur ilovalarda keltirilgan.

Shu bilan birga, ma'ruzalarda algoritmlarni bosqichma-bosqich amalga oshirish, berilganlar tuzilmalari va algoritmlarning qanday ishlashini ko‘rsatish uchun muayyan muhim usullar tushuntirilgan.

Ushbu qo‘llanmada asosiy e'tibor amaliy muammolarni hal qilish usullarini oddiy tushuntirishga qaratilgan. Ushbu materialda murakkab matematik hisob-kitoblar keltirilmasdan, iloji boricha o‘quvchilar uchun sodda shaklda ko‘plab rasmlar bilan tushuntirilgan.

Qo‘llanma olti qismga bo‘lingan:

Birinchi qismda algoritm tushunchasi, uning ta’rifi va maqsadi, algoritmning murakkabligi, algoritmlarning asimptotik baholari keltirilgan.

Ikkinchi qismda berilganlarning chiziqli tuzilmalari, massivlar, bir bog‘lamli (yo‘nalishli) ro‘yxatlar, ikki bog‘lamli (yo‘nalishli) ro‘yxatlar, steklar, navbatlar, deklar, halqasimon ro‘yxatlar, ustuvor navbatlar tavsiflanadi.

Uchinchi qismda rekursiya usuli ko‘rib o‘tiladi. Algoritmlarning rekursiv bajarilishi C# tilida rekursiv usullar orqali aniqlanadi.

To‘rtinchi qismda daraxt ko‘rinishidagi nochiziqli berilganlar tuzilmalari tavsiflanadi. Daraxt bo‘ylab o‘tish turlari ko‘rsatilgan, qidirish algoritmlarida daraxtdan foydalanish, piramidali daraxt va piramidalar orqali ustuvor navbatni amalga oshirish tasvirlangan.

Beshinchi qismda tartiblash algoritmlari tasvirlangan. Bu qismda tartiblash algoritmlarini qanday tasniflash ko‘rsatilgan. Kvadrat murakkablikdan chiziqli murakkablikgacha bo‘lgan algoritmlar keltirilgan.

Oltinchi qismda qidirish algoritmlari keltirilgan. Bu qismda turli xil qidirish algoritmlari taqdim etilib, ularning murakkabligi va samaradorligi ko‘rsatilgan.

Ushbu fan bo‘yicha ma’lumotlarning ko‘pligini hisobga olgan holda, graflar, to‘plamlarni kesib o‘tmaydigan graflar, satrlarni qayta ishlash algoritmlari, berilganlarni xeshlash masalalari va boshqalar kabi shu fanga tegishli bo‘lgan bo‘limlar qo‘llanmaning 2-qismida taqdim etiladi.

Ushbu qo‘llanma dasturlash, berilganlar tuzilmalari va algoritmlar bilan bog‘liq mutaxasisliklar bo‘yicha tahsil olayotgan bakalavriat va magistratura talabalari hamda algoritm va dasturlashga qiziquvchi barcha uchun foydali bo‘ladi.

Mualliflar, prof. S.Gaynazarov va dots. D.Saidovlar ushbu qo‘llanma bo‘yicha qimmatli mulohazalar va takliflar uchun oldindan minnatdorchilik bildiradilar.

1. Asosiy tushunchalar

Ushbu ma'ruzaning maqsadi algoritmlarni tahlil qilish, ularning belgilanishlari, munosabatlari va iloji boricha koʻproq masalalarni yechishning ahamiyatini tushuntirishdan iborat.

Avvalo algoritmlarning asosiy elementlarini tushunishga, algoritmlarni tahlil qilishning ahamiyatiga toʻxtalamiz. Ushbu ma'ruzani tugatgandan soʻng, har qanday berilgan algoritm (ayniqsa, rekursiv funksiyalar) ning murakkabligini aniqlay olishingiz mumkin.

Keling, algoritm ishlatadigan tushunchalarni koʻrib chiqaylik.

1.1. Oʻzgaruvchilar

Oʻzgaruvchilar- bu berilganlarni saqlash uchun ishlatilib, algoritm ishlashi davomida har xil berilganlarni oʻzgaruvchilarda saqlashi mumkin, shuning uchun berilganlarni saqlash uchun algoritmda oʻzgaruvchilar kerak boʻladi.

1.2. Berilganlar turlari

Oʻzgaruvchilar har qanday qiymatlarni saqlashi mumkin, masalan, butun (10, 20), haqiqiy sonlar (0.23,.5.5) yoki shunchaki 0 va 1.

Oʻzgaruvchilardan foydalanish uchun ularni qabul qilishi mumkin boʻlgan qiymatlar turiga bogʻlashimiz kerak. Berilganlar turi - bu dasturlash tilida oldindan aniqlangan qiymatlarga ega boʻlgan ma'lumotlar toʻplamiga murojaat qilish uchun ishlatiladigan nom. Berilganlar turlariga misollar: butun son, suzuvchi nuqtali haqiqiy son, belgi, satr va boshqalar.

Kompyuter xotirasida faqat nol va bir saqlanadi. Faraz qilamiz bizda qandaydir berilganlar mavjud boʻlib, biz ularni kodlashni xohlasak, unda nol va bir shaklida yechim berish juda qiyin. Foydalanuvchilarga yordam berish uchun dasturlash tillari va kompilyatorlar bizga berilganlar turlarini taqdim etadi. Masalan, butun son *int* 2 baytni egallaydi (aslida qiymat kompilyatorga bogʻliq), *float* 4 baytni egallaydi va hokazo.

Bu shuni anglatadiki, biz xotirada 2 baytni (16 bit) birlashtiramiz va uni butun son deb ataymiz. Xuddi shunday, 4 baytni (32 bit) birlashtirib, uni haqiqiy son deb ataymiz. Berilganlar turi kodlash mashaqqatini kamaytiradi.

Yuqori darajada berilganlarning ikkita turi mavjud:

- Tizim berilganlar turlari (tayanch berilganlar turlari deb ham ataladi);
- Foydalanuvchi tomonidan aniqlangan berilganlar turlari.

1.3. Tayanch berilganlar turlari

Tizim tomonidan aniqlanadigan berilganlar turlari tayanch turlar deyiladi.

Ko‘pgina dasturlash tillaridagi berilganlarning tayanch turlariga quyidagi turlar: *int, float, char, double, bool* va boshqalar kiradi.

Har bir tayanch tur uchun kompyuter xotirasidan ajratiladigan baytlar soni dasturlash tillariga, kompilyatorga va operatsion tizimga bog‘liq.

Turli tillar bir xil tayanch berilganlar turi uchun turli o‘lchamlardan foydalanishi mumkin. Berilganlar turlarining hajmiga qarab, ular qabul qilishi mumkin bo‘lgan qiymatlar diapozoni ham turlicha bo‘ladi.

Masalan, *int* 2 yoki 4 bayt egallashi mumkin. Agar u 2 baytli (16 bit) bo‘lsa, unda umumiy mumkin bo‘lgan qiymatlar diapozoni -32768 dan +32767 oraliqgacha ya’ni -2^{15} dan $2^{15}-1$ oralig‘idagi butun sonlarni qabul qilishi mumkin. Agar 4 baytli (32 bit) bo‘lsa, unda mumkin bo‘lgan qiymatlar [-2147483648, +2147483647] oqaliqdagi butun sonlarni qabul qiladi ya’ni -2^{31} dan $2^{31}-1$ gacha. Boshqa berilganlar turlari ham xuddi shu tarzda bo‘ladi.

1.4. Foydalanuvchi tomonidan aniqlangan berilganlar turlari

Agar qo‘yilgan masalani yechishda tizimda aniqlangan tayanch turlar yetarli bo‘lmasa, aksariyat dasturlash tillari foydalanuvchi tomonidan aniqlangan berilganlar turlari deb nomlangan yangi berilganlar turlarini aniqlashga imkon beradi.

C/C++ dagi strukturalar va Java dagi sinflar maxsus berilganlar turlariga yaxshi misol bo‘la oladi. Masalan, quyidagi dastur matnida bir nechta tizim berilganlar turlarini birlashtirish orqali foydalanuvchi tomonidan aniqlangan *Circle* nomli berilganlar turi yaratilgan. Bu kompyuter xotirasi bilan ishlashda ko‘proq moslashuvchanlik va qulaylik beradi.

struct Point {double x; double y;};

struct RGBColor { char red; char green; char blue;};

struct Circle { Point center; double radius; int thickness; RGBColor color; };

1.5. Berilganlar tuzilmalari

Yuqoridagi mulohazalarga asoslanib, bizda o‘zgaruvchilarda berilganlar mavjud bo‘lganda, muammoni hal qilish uchun ushbu berilganlarni manipulyatsiya qilish uchun qandaydir mexanizm kerak bo‘ladi.

Berilganlar tuzilmasi – bu berilganlarni samarali ishlatish uchun kompyuterda saqlash va tartibga solishning maxsus usuli.

Berilganlar tuzilmasi – bu berilganlarni tartibga solish va saqlash uchun maxsus format.

Berilganlar tuzilmalarining keng tarqalgan turlariga massivlar, fayllar, bog‘langan ro‘yxatlar, steklar, navbatlar, daraxtlar, graflar va boshqalar kiradi.

Elementlarning tashkil qilinishiga qarab berilganlar tuzilmalari ikki turga bo‘linadi:

1) Berilganlarning chiziqli tuzilmalari: elementlarga murojaat ketma – ket tartibda amalga oshiriladi, ammo barcha elementlarni ketma – ket saqlash shart emas. Misollar: bog‘langan ro‘yxatlar, steklar va navbatlar.

2) Berilganlarning nochiziqli tuzilmalari: bu berilganlar tuzilmasining elementlari chiziqli bo‘lmagan tartibda saqlanadi. Misollar: daraxtlar va graflar.

1.6. Berilganlarning abstrakt turi (BAT)

Berilganlarning abstrakt turlarini aniqlashdan oldin, tizim tomonidan aniqlangan berilganlar turlarini boshqa nuqtai nazar bilan ko‘rib chiqamiz. Barchamizga ma’lumki, barcha tayanch berilganlar turlari (*int, float* va boshqalar) kelishuv bo‘yicha qo‘shish va ayirish kabi asosiy amallarni qo‘llab-quvvatlaydi. Tizim ushbu amallarni berilganlarning tayanch turlari uchun amalga oshirilishini ta'minlaydi va har xil turlar uchun ular mashina buyruqlarining turli guruhlari tomonidan amalga oshiriladi.

Foydalanuvchilar tomonidan yaratiladigan berilganlar turlari uchun biz amallarni ham aniqlashimiz kerak bo‘lib, bu amallar hatto butun tuzilma

ustida ham amal qilishi mumkin, odatda foydalanuvchi tomonidan aniqlangan berilganlar turlari ularning amallari bilan bir qatorda aniqlanadi.

Masalani hal qilish jarayonini soddalashtirish uchun biz berilganlar tuzilmalarini ulardagi amallar bilan birlashtiramiz. Biz bu birlashishni berilganlarning abstrakt turlari (BAT) deb ataymiz.

BAT ikki qismdan iborat:

1. Berilganlarni e'lon qilish;
2. Amallarni aniqlash.

Odatda ishlatiladigan BAT lariga quyidagilar kiradi: bog‘langan ro‘yxatlar, steklar, navbatlar, ustuvor navbatlar, ikkilik daraxtlar, lug‘atlar, to‘plamlar (birlashma va izlash), xesh jadvallar, graflar va boshqa ko‘plab berilganlar tuzilmalari.

Masalan, stekda berilganlarni saqlash uchun LIFO (Last-In First-Out) mexanizmidan foydalaniladi. Stekka oxirgi kirgan element birinchi bo‘lib olib tashlanadi. Umumiy amallar quyidagilardir: stekni yaratish, stekga elementni qo‘shish, stekdan elementni olib tashlash, stekning joriy uchini izlash, stekdagi elementlar sonini aniqlash va h.k.

BAT ni aniqlashda, tafsilotlari haqida qayg‘urmasligingiz kerak. Ular biz foydalanmoqchi bo‘lgan paytdagina amalga oshiriladi.

Har xil turdagi BATlar har xil turdagi ilovalarga mos keladi, ba'zilari esa aniq vazifalar uchun ixtisoslashgan.

1.7. Algoritm tushunchasi

“Algoritm” termini eramizning 780 – 850 yillarida yashagan olim Muhammad Al Xorazmiy nomi bilan bog‘liq. Oshkor ravishda algoritm tushunchasi XX asr boshlarida ijod qilgan D. Gilbert, K. Gyodel, S. Klini, A. Chyorch, E. Post, A. Tyuring, N. Viner va A. Markovlarning ilmiy ishlarida shakllangan.

Eng birinchi sonli algoritmlardan biri bizning eramizdan avvalgi III asrda paydo bo‘lgan Evklid algoritmi, yani ikkita natural sonning EKUB ini topish algoritmi hisoblanadi.

• “Algoritm – bu chekli sondagi qoidalar to‘plami bo‘lib, u aniq bir to‘plamga tegishli masalalarni yechadigan amallar ketma-ketligi aniqlaydi va beshta muhim xysusiyatlarga ega: cheklilik, aniqlilik, kirish, chiqish va samaradorlilik” (D. E. Knut).

• “Algoritm – bu qat’iy aniqlangan qoidalarni bajaruvchi qandaydir hisoblash sistemasi bo‘lib, bir qator qadamlardan so‘ng qo‘yilgan masalani yechishga olib keladi”(A. Kolmogorov).

• “Algoritm – bu o‘zgarib turadigan kirish berilganlaridan kerakli natijaga olib boradigan hisoblash jarayonini aniqlaydigan aniq ko‘rsatmadir”(A. Markov).

• “Algoritm – bir turdagi barcha masalalarni yechishga olib keladigan amallar tizimining ma’lum bir tartibda bajarilishi uchun aniq ko‘rsatmadir”(M. M. Rozental tahriri ostidagi filosofik lug‘atidan).

Algoritmlar nazariyasida yechiladigan masalalar va maqsadlar

Algoritm tushunchasini formallashtirish va formal algoritmik tizimlarni tadqiq qilish;

• Masalaning algoritmik yechilmasligini formal isboti;

• Masalani klassifikatsiyalash (tasniflash), murakkab sinflarni aniqlash va tadqiq qilish;

• Algoritmning murakkabligini asimptotik tahlil qilish;

• Rekursiv algoritmlarni tadqiq qilish va tahlil qilish;

• Algoritmlarni qiyosiy tahlil qilish maqsadida mehnat hajmining oshkor funksiyalarini olish;

• Algoritmlar sifatini qiyosiy baholash mezonlarini ishlab chiqish.

1.8. Algoritm tushunchasini formallashtirish

Ta’rif 1. Algoritm - bu mumkin bo‘lgan dastlabki berilganlar sinfi uchun umumiy bo‘lgan muammoni hal qilish uchun bajariladigan elementar amallarning chekli ketma-ketligini belgilaydigan ma'lum bir tilda berilgan chekli ko‘rsatmadir.

Masalaning boshlang‘ich berilganlar sohasi (to‘plami) D va mumkin bo‘lgan natijalar to‘plami R bo‘lsin, u holda algoritm D ni R ga akslantiradi, ya’ni D→R. Bu akslantirish to‘liq bo‘lmasligi mumkin.

Agar natija faqat ba’zi bir d∈D uchun olingan bo‘lsa, algoritm qisman algoritm deb ataladi, agar algoritm barcha d∈D uchun to‘g‘ri natijaga erishsa, u holda to‘liq algoritm deyiladi.

Ta’rif 2. Algoritm – bu cheklangan vaqt ichida masalani yechishga erishish uchun ijrochining harakatlar tartibini tavsiflovchi aniq ko‘rsatmalar to‘plamidir.

Algoritmning oshkor yoki oshkormas shaklda aniqlanishi bir qator talablarni keltirib chiqaradi:

• Algoritm cheklangan miqdordagi elementar bajariluvchi ko‘rsatmalardan iborat bo‘lishi kerak, ya’ni yozuvning cheklilik talabini qanoatlantirishi kerak;

• Algoritm masalani yechishda chekli sondagi qadamlarni bajarishi kerak, ya’ni bajarilishning cheklilik talabini qondirishi kerak;

• Kiruvchi berilganlarning mumkin bo‘lgan barcha qiymatlari uchun algoritm yagona bo‘lishi kerak, ya’ni ommaviylik talabini bajarishi kerak;

• Algoritm qo‘yilgan masalaga nisbatan to‘g‘ri yechimga olib kelishi kerak, ya’ni to‘g‘rilik talabini bajarishi kerak.

1.9. Algoritmlarning rasmiy xususiyatlari

Qo‘yilgan biror masalani kompyuterda yechish uchun, avval uning matematik modelini, keyin algoritmini va dasturini tuzish kerak bo‘ladi. Bu uchlikda algoritm bloki muhim ahamiyatga ega. Endi algoritm tushunchasining ta’rifi va xossalarini bayon qilamiz.

Algoritm bu oldimizga qo‘yilgan masalani yechish uchun zarur bo‘lgan amallar ketma-ketligidir.

Masalan kvadrat tenglamani yechish uchun quyidagi amallar ketma-ketligi zarur bo‘ladi: *a,b,c* - koeffitsientlar berilgan bo‘lsin

1. $D=b^2-4ac$ hisoblanadi;
2. $D>0$ bo‘lsa, yechim $x_{1,2} = -b \pm \sqrt{D}/(2a)$, agar $D<0$ bo‘lsa, haqiqiy yechimi yo‘q.

Misol sifatida yana berilgan a, *b, c* tomonlari bo‘yicha uchburchakning yuzasini Geron formulasi bo‘yicha hisoblash masalasini ko‘rib o‘taylik.

a, b, c –uchburchakning tomonlari uzunliklari bo‘lsin:

1. $p=(a+b+c)/2$ – perimetrning yarmi hisoblanadi;
2. $T=p(p-a)(p-b)(p-c)$ hisoblanadi;
3. $s = \sqrt{T}$ hisoblandi.

Yuqoridagi misollardan ko‘rinib turibdiki, algoritmning har bir qadamda bajariladigan amallar tushunarli va aniq tarzda ifodalangan, hamda chekli sondagi amallardan keyin aniq natija olinishi kerak.

Yuqorida aytib o‘tilgan, tushunarlilik, aniqlilik, cheklilik va natijaviylik tushunchalari algoritmning asosiy xossalarini tashkil etadi.

Algoritmning 5-ta asosiy xossasi mavjud:

Diskretlilik (Cheklilik). Bu xossaning mazmuni algoritmlarni doimo chekli qadamlardan iborat qilib bo‘laklash imkoniyatini beradi. Ya’ni uni chekli sondagi oddiy ko‘rsatmalar ketma-ketligi shaklida ifodalash mumkin. Agar kuzatilayotgan jarayonni chekli qadamlardan iborat qilib qo‘llay olmasak, uni algoritm deb bo‘lmaydi.

Tushunarlilik. Biz kundalik hayotimizda berilgan algoritmlar bilan ishlayotgan elektron soatlar, mashinalar, dastgohlar, kompyuterlar, turli avtomatik va mexanik qurilmalarni kuzatamiz. Ijrochiga tavsiya etilayotgan ko‘rsatmalar, uning uchun tushunarli mazmunda bo‘lishi shart, aks holda ijrochi oddiygina amalni ham bajara olmaydi. Undan tashqari, ijrochi har qanday amalni bajara olmasligi ham mumkin.

Har bir ijrochining bajarishi mumkin bo‘lgan ko‘rsatmalar yoki buyruqlar majmuasi mavjud, u ijrochining ko‘rsatmalar tizimi (sistemasi) deyiladi. Demak, ijrochi uchun berilayotgan har bir ko‘rsatma ijrochining ko‘rsatmalar tizimiga mansub bo‘lishi lozim.

Ko‘rsatmalarni ijrochining ko‘rsatmalar tizimiga tegishli bo‘ladigan qilib ifodalay bilishimiz muhim ahamiyatga ega. Masalan, quyi sinfning a’lochi o‘quvchisi "son kvadratga oshirilsin" degan ko‘rsatmani tushunmasligi natijasida bajara olmaydi, lekin "son o‘zini o‘ziga ko‘paytirilsin" shaklidagi ko‘rsatmani bemalol bajaradi, chunki u ko‘rsatma mazmunidan ko‘paytirish amalini bajarish kerakligini anglaydi.

Aniqlilik. Ijrochiga berilayotgan ko‘rsatmalar aniq mazmunda bo‘lishi zarur. Chunki ko‘rsatmadagi noaniqliklar mo‘ljaldagi maqsadga erishishga olib kelmaydi. Odam uchun tushunarli bo‘lgan "3-4 marta silkitilsin", "5-10 daqiqa qizdirilsin", "1-2 qoshiq solinsin", "tenglamalardan biri yechilsin" kabi noaniq ko‘rsatmalar robot yoki kompyuterni qiyin ahvolga solib qo‘yadi.

Bundan tashqari, ko‘rsatmalarning qaysi ketma-ketlikda bajarilishi ham muhim ahamiyatga ega. Demak, ko‘rsatmalar aniq berilishi va faqat algoritmda ko‘rsatilgan tartibda bajarilishi shart ekan.

Ommaviylik. Har bir algoritm mazmuniga ko‘ra bir turdagi masalalarning barchasi uchun ham o‘rinli bo‘lishi kerak. ya’ni masaladagi

boshlang‘ich ma’lumotlar qanday bo‘lishidan qat’iy nazar algoritm shu xildagi har qanday masalani yechishga yaroqli bo‘lishi kerak. Masalan, ikki oddiy kasrning umumiy maxrajini topish algoritmi, kasrlarni turlicha o‘zgartirib bersangiz ham ularning umumiy maxrajlarini aniqlab beraveradi. Yoki uchburchakning yuzini topish algoritmi, uchburchakning qanday bo‘lishidan qat’iy nazar, uning yuzini hisoblab beraveradi.

Natijaviylik. Har bir algoritm chekli sondagi qadamlardan so‘ng albatta natija berishi shart. Bajariladigan amallar ko‘p bo‘lsa ham baribir natijaga olib kelishi kerak. Chekli qadamdan so‘ng qo‘yilgan masala yechimga ega emasligini aniqlash ham natija hisoblanadi. Agar ko‘rilayotgan jarayon cheksiz davom etib natija bermasa, uni algoritm deb atay olmaymiz.

Algoritmda xatolar mavjud bo‘lsa u noto‘g‘ri natijaga olib keladi yoki umuman natija bermaydi.

Algoritmda xatolar mavjud bo‘lmasa, har qanday mumkin bo‘lgan berilganlar uchun to‘g‘ri natija beradi.

Algoritmlarni an'anaviy ravishda o‘rganishda algoritmning afzalliklarini baholashning ikkita asosiy mezonlari mavjud:

to‘g‘riligi - algoritm masalani chekli bosqichda hal qila olishi;

samaradorlilik - vazifani bajarish uchun qancha resurs (xotira va vaqt jihatidan) talab qilinadi.

1 - bob bo‘yicha nazorat savollari

1. O‘zgaruvchilarning ishlatilishi.
2. Berilganlar turlari.
3. Tayanch berilganlar turlari.
4. Foydalanuvchi tomonidan aniqlangan berilganlar turlari.
5. Berilganlar tuzilmalari.
6. Berilganlarning abstrakt turi (BAT).
7. Algoritm tushunchasi.
8. Algoritmlar nazariyasida yechiladigan masalalar va maqsadlar.
9. Algoritm tushunchasini formallashtirish.
10. Algoritmlarning rasmiy xususiyatlari.

2. Algoritmlar tahlili

"A" shahardan "B" shaharga borishni ko‘p usullar bilan amalga oshirish mumkin: samolyotda, avtobusda, poyezdda, shuningdek, velosipedda. Mavjud usullar va ularning qulayligiga qarab, biz o‘zimizga mos keladigan usulni tanlaymiz.

Xuddi shu singari, informatika fanida biror bir masalani yechish uchun bir nechta algoritmlar mavjud (masalan, tartiblash masalasida qo‘shish orqali tartiblash, tanlash orqali tartiblash, tezkor tartiblash va boshqalar kabi ko‘plab algoritmlar mavjud).

Algoritm tahlili turli algoritmlarning hisoblash vaqti va kompyuter xotirasidan egallaydigan joyi nuqtai-nazaridan qanchalik samarali ekanligini aniqlashga yordam beradi.

Algoritm tahlilining maqsadi algoritmlarni (yoki yechimlarni) asosan bajarilish vaqti va boshqa omillar (masalan, xotira hajmi, algoritm murakkabligi va boshqalar) bo‘yicha solishtirishdir.

2.1. Bajarilish vaqti tahlili

Bu masala hajmi ortishi (kiruvchi berilganlarning hajmi oshganda) bilan bajarilish vaqti qanday o‘zgarishini aniqlash jarayonidir. Kiruvchi berilganlarning o‘lchami - kirishdagi elementlarning soni bo‘lib, u masalani va berilganlarning turlariga bog‘liq.

Kiruvchi berilganlar har xil turda tegishli bo‘lishi mumkin.

Quyida keng tarqalgan kiruvchi berilganlarning turlari keltirilgan:

- Massiv hajmi;
- Polinom darajasi;
- Matritsadagi elementlar soni;
- Berilganlarning ikkilik ko‘rinishidagi bitlar soni;
- Grafning uchlari va qirralari soni.

2.2. Algoritmlarni solishtirish

Algoritmlarni solishtirish uchun bir nechta ob'ektiv ko‘rsatkichlardan foydalaniladi:

Bajarilish vaqti. Bu eng yaxshi ko‘rsatkich emas, chunki bajarish vaqti aniq bir kompyuter arxitekturasiga bog‘liq.

Bajarilgan amallar soni. Bu ham yaxshi o‘lchov emas, chunki bajarilgan amallar soni ba'zi shartlarga qarab farq qilishi mumkin.

Dasturlash tilini tanlash, shuningdek, individual dasturchi tomonidan dastur yozish uslubi.

Umumiy qabul qilingan eng yaxshi yechim quyidagi usul hisoblanadi:

Aytaylik, ma'lum bir algoritmning bajarilish vaqti masalaning kiruvchi berilganlari hajmi - n (ya'ni f (n)) funksiyasi sifatida aniqlangan va turli bajarilish usullariga mos keladigan ushbu turli funksiyalarni taqqoslash kerak bo‘ladi.

Bunday taqqoslash usuli mashina vaqtiga, dasturlash uslubiga va boshqa parametrlarga bog‘liq emas.

O‘sish tezligi

Kiruvchi berilganlarning hajmiga qarab algoritmning ishlash vaqti ortib borish tezligi **o‘sish tezligi** deb ataladi.

Aytaylik, siz do‘konga mashina va velosiped sotib olish uchun borasiz. Agar siz u yerda do‘stingizni uchratsangiz va u nima sotib olayotganingizni so‘rasa, umuman olganda siz unga mashina sotib olayotganingizni aytasiz. Buning sababi, avtomobilning narxi velosiped narxidan ancha yuqori.

Yuqoridagi keltirilgan misolda avtomobil va velosiped narxini funksiya nuqtai nazaridan qaraylik. Ushbu funksiya uchun siz nisbatan kichik tartibli hadlariga e’tibor qaratmaysiz chunki ularning qiymatlari nisbatan kichik bo‘lib, yetarlicha katta kiruvchi berilganlar qiymati *n* uchun ahamiyatsiz bo‘lib qoladi.

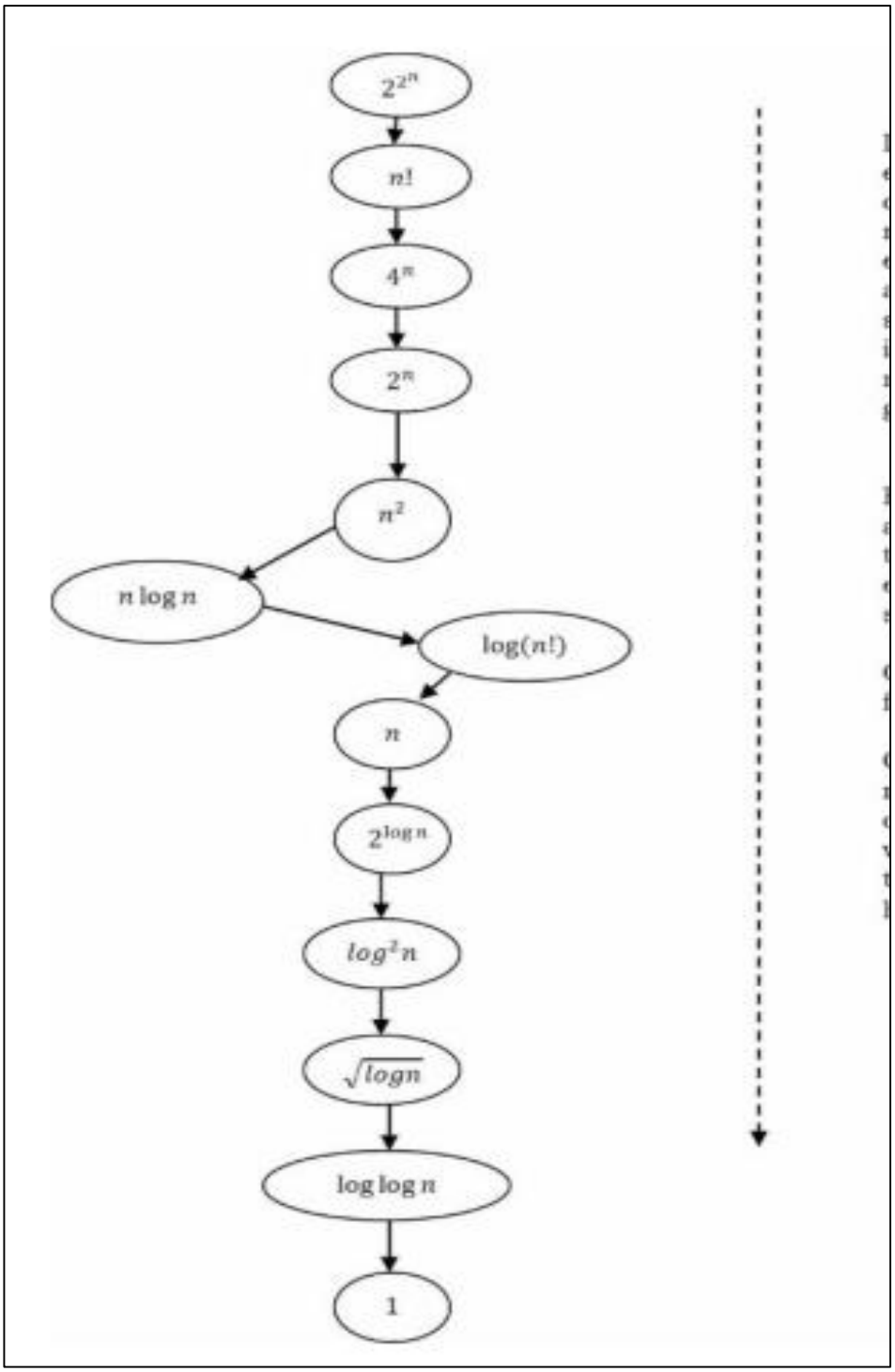
Misol tariqasida, n^4, $2n^2$, $100n$ va 500 ba'zi bir funksiyaning individual xarajatlari bo‘lsin:

$$f(n) = n^4 + 2n^2 + 100n + 500$$

u holda yetarlicha katta *n* uchun umumiy xarajatlar n^4 tartibida bo‘ladi, chunki n^4 had funksiyadagi o‘sish tezligi eng kattasi hisoblanadi.

O‘sish tezliklari diagrammasi

Quyidagi diagrammada turli o‘sish tezliklari o‘rtasidagi bog‘liqlik ko‘rsatilgan.



Quyida keyingi ma'ruzalarda duch keladigan o‘sish tezligining jadvali keltirilgan.

Jadval №1. Ko‘p uchraydigan o‘sish tezligi funksiyalari.

Mehnat hajmi	**Nomi**	**Misol**
1	O‘zgarmas	Bog‘langan ro‘yxatning oxiriga yoki boshiga element qo‘shish

log*n*	Logarifmik	Saralangan massivda elementni qidirish
n	Chiziqli	Saralanmagan massivda elementni qidirish
*n**log*n*	Chiziqli-logarifmik	"Bo‘l va hukmronlik qil" tamoyili orqali *n* elementlarni saralash. Birlashtirish orqali saralash
n^2	Kvadratik	Grafning ikki tuguni orasidagi eng qisqa yo‘lni topish
n^3	Kubik	Matritsalarni ko‘paytirish
2^n	Eksponensial	"Xanoy minorasi" masalasi

2.3. Tahlil turlari

Berilgan algoritmni tahlil qilish uchun algoritm qaysi kiruvchi berilganlarga kamroq vaqt va qaysi kiruvchi berilganlarga ko‘proq vaqt talab qilishini bilishimiz kerak.

Biz algoritmni ifoda ko‘rinishida berilishi mumkinligini ko‘rib chiqdik. Bu shuni anglatadiki, biz bir nechta ifodalar bilan algoritmni taqdim etamiz: biri kamroq vaqt talab qiladigan holat uchun bo‘lsa, boshqasi ko‘proq vaqt talab qiladigan holat uchun.

Odatda, birinchisi eng yaxshi holat, ikkinchisi esa algoritmning eng yomon holati deb ataladi.

Algoritmni tahlil qilish uchun bizga asimptotik tahlil uchun asos bo‘ladigan qandaydir qoida - simvolik / belgilashlar / baholash kerak.

Tahlilning uchta turi mavjud:

• *Eng yomon holat.*

○ Kiruvchi berilganlarning algoritm bajarilishi uchun eng ko‘p vaqt (tugatish uchun eng uzoq vaqt) talab qilinadigan to‘plamini aniqlaydi.

• *Eng yaxshi holat.*

○ Kiruvchi berilganlarning algoritm bajarilishi uchun eng kam vaqt (tugatish uchun eng qisqa vaqt) talab qilinadigan to‘plamini aniqlaydi.

• *O‘rtacha holat.*

○ Algoritm bajarilish vaqtini bashorat qiladi;

○ Algoritm turli xil kiruvchi berilganlar to‘plamlari uchun mos ravishda ko‘p marta bajariladi;

○Ba'zi umumiy berilganlar guruhidan kirish berilganlarining sinov to‘plami tanlanadi va ularda algoritmning umumiy ishlash vaqti hisoblab chiqiladi, so‘ngra sinovlar soniga bo‘linadi;

○ Kirish berilganlari tasodifiy tanlanadi.

Pastki chegara <= O‘rtacha vaqt <= Yuqori chegara

Berilgan algoritm uchun biz eng yaxshi, eng yomon va o‘rtacha holatlarni ifoda ko‘rinishida ko‘rsatishimiz mumkin. Misol tariqasida berilgan algoritmni ifodalovchi *f(n)* funksiya qaraylik:

$$f(n) = n^2 + 500 \;\; Eng\ yomon holat$$
$$f(n) = n + 100n + 500 \;\; Eng\ yaxshi holat$$
$$f(n) = n^2/2 + 50n + 500 \;\; O'rta holat$$

2.4. Asimptotik baholar

Eng yaxshi, o‘rtacha va eng yomon holatlar uchun ifodalarga ega bo‘lgan holda, barcha uchta holat uchun biz murakkablikning yuqori va pastki baholarini aniqlashimiz lozim. Ushbu yuqori va pastki chegaralarni tasvirlash uchun bizga ba'zi belgilashlar kerak bo‘ladi.

Faraz qilaylik, ushbu algoritmning murakkabligi *f(n)* funksiya bilan ifodalangan bo‘lsin.

O-yuqori bahosi (yuqori chegara funksiyasi)

Bu baholar berilgan funksiya uchun aniq yuqori chegarani beradi. Ular odatda *f(n)* = **O***(g(n))* shaklida yoziladi. Bu *n* ning yetarlicha katta qiymatlari uchun *f(n)* ning yuqori chegarasi *g(n)* ga teng ekanligini anglatadi.

O yuqori baho *f(n)* funksiyaning qandaydir $n > n_0$ dan boshlab, o‘zgarmas ko‘paytuvchigacha bo‘lgan aniqlikda *g(n)* dan oshmasligini talab qiladi.

Masalan, $f(n) = n^4 + 100n^2 + 10n + 50$ – algoritmning berilgan murakkabligi bo‘lsa, $g(n) = n^4$ ga teng bo‘ladi. Bu shuni anglatadiki, *g(n) n* ning yetarlicha katta qiymatlari uchun *f(n)* ning maksimal o‘sish tezligini beradi.

Keling, **O** yuqori bahoning matematik ta'rifini beraylik.

O(g(n)) = {f(n): ∃ c va n_0 musbat o‘zgarmaslar mavjudki, barcha n > n_0 uchun $0 \leq f(n) \leq cg(n)$ bo‘ladi }. *g(n) – f(n)* uchun asimptotik aniq yuqori baho bo‘ladi.

O(*g(n)*) yozuvi o‘zgarmas ko‘paytuvchigacha aniqlikda $g(n)$ funksiyasidan tez o‘smaydigan funksiyalar sinfini bildiradi, shuning uchun ba’zan $f(n)$ funksiyani $g(n)$ kattalashtiradi, deyishadi.

Maqsadimiz berilgan *f(n)* algoritmining o‘sish tezligidan kam bo‘lmagan eng kichik o‘sish tezligi *g(n)* ni olishdir.

Biz odatda *n* ning kichik qiymatlarini rad etamiz. Bu shuni anglatadiki, *n* ning kichik qiymatlarida o‘sish tezligi unchalik muhim emas.

Ifodadagi n_0 nuqta algoritmning ushbu nuqtadan boshlab o‘sish tezligini qarashimiz kerak bo‘lgan nuqta. O‘sish tezligi n_0 dan kichik bo‘lganda boshqacha bo‘lishi mumkin. n_0 – ushbu funksiyaning chegarasi deyiladi.

O(*g(n)*) - o‘sish tartibi *g(n)* dan kichik yoki bir xil bo‘lgan funksiyalar to‘plami. Masalan:

O(n^2) tarkibiga O(*1*), O(*n*), O(*nlogn*) va boshqalar kiradi.

Algoritmlarni faqat *n* ning katta qiymatlari uchun tahlil qilish kerak. Bu shuni anglatadiki, n_0 dan kichik bo‘lgan holatlarni hisobga olmasligimiz mumkin.

Big-O uchun misollar:

1-misol. $f(n) = 3n + 8$ uchun yuqori chegarani toping.

Yechish: hamma $n \geq 8$ uchun $3n + 8 \leq 4n$;

$\therefore 3n + 8 = O(n)$ bo‘lib, $c = 4$ va $n_0 = 8$ ga teng bo‘ladi;

2-misol. $f(n) = n^2 + 1$ uchun yuqori chegarani toping.

Yechish: hamma $n \geq 1$ uchun $n^2 + 1 \leq 2n^2$;

$\therefore n^2 + 1 = O(n^2)$ bo‘lib, $c = 2$ va $n_0 = 1$ ga teng bo‘ladi;

3-misol. $f(n) = n^4 + 100n^2 + 50$ uchun yuqori chegarani toping.

Yechish: hamma $n \geq 11$ uchun $n^4 + 100n^2 + 50 \leq 2n^4$;

$\therefore n^4 + 100n^2 + 50 = O(n^4)$ bo‘lib, $c = 2$ va $n_0 = 11$ teng bo‘ladi.

Asimptotik baholarni isbotlashda n_0 va *c* uchun yagona qiymatlar to‘plami mavjud emas.

Masalan, $100n + 5 = O(n)$ ifodani ko‘rib chiqaylik.

Bu funksiya uchun n_0 va *c* ning bir nechta qiymatlari mavjud.

1-yechim: $100n + 5 \leq 100n + n = 101n \leq 101n$, bu holda barcha $n \geq 5$ uchun $n_0 = 5$ va $c = 101$ bo‘lgan qiymatlar yechim hisoblanadi.

2-yechim: $100n + 5 \leq 100n + 5n = 105n \leq 105n$, bu holda barcha $n > 1$ uchun $n_0 = 1$ va $c = 105$ bo‘lgan qiymatlar yechim hisoblanadi.

Ω – *quyi bahosi [pastki chegara funksiyasi]*

O – yuqori baho kabi, ushbu baho berilgan algoritm bo‘yicha aniqroq pastki chegarani beradi va $f(n) = \Omega(g(n))$ shaklida yoziladi. Bu n ning yetarlicha katta qiymatlari uchun $g(n)$ funksiya $f(n)$ ning pastki chegarasi ekanligini anglatadi.

Masalan, agar $f(n) = 100n^2 + 10n + 50$ bo‘lsa, g(n) – Ω (n^2) ga teng.

Ω *quyi baho*ning matematik ta'rifini beraylik.

$\Omega(g(n)) = \{f(n)$: $\exists\, c$ va n_0 musbat o‘zgarmaslar mavjudki, barcha $n > n_0$ uchun $0 \leq cg(n) \leq f(n)$ bo‘ladi$\}$. $g(n) – f(n)$ uchun asimptotik aniq quyi chegara (baho) bo‘ladi. Bizning maqsadimiz berilgan algoritm tezligidan kichik yoki teng bo‘lgan $g(n)$ ning eng yuqori o‘sish tezligini olishdir.

Ω –quyi baholashga misollar:

1-misol. $f(n) = 5n^2$ uchun quyi chegarani toping.

Yechish: $\exists\, c$, n_0 mavjudki: $0 \leq cn^2 \leq 5n^2 \Rightarrow cn^2 \leq 5n^2 \Rightarrow c = 5$ va $n_0=1$.

$\therefore 5n^2 = \mathbf{\Omega}(n^2)$ bo‘lib, $c = 5$ va $n_0 = 1$ ga teng bo‘ladi.

2-misol. $f(n) = 100n + 5 \neq \mathbf{\Omega}(n^2)$ ekanligini isbotlang.

Yechish: $\exists c$, n_0 mavjudki: $0 \leq cn^2 \leq 100n + 5$.

$100n + 5 \leq 100n + 5n(\forall n \geq 1) = 105n$.

$cn^2 \leq 105n \Rightarrow n(cn - 105) \leq 0$.

n musbat bo‘lgani uchun $\Rightarrow cn - 105 \leq 0 \Rightarrow n \leq 105/c$

$\Rightarrow$ Qarama-qarshilik: n o‘zgarmasdan kichik bo‘lishi mumkin emas.

Θ - o‘rta baho (o‘rtacha asimptotik funksiya)

Bu baho berilgan funksiya (algoritm) ning yuqori va pastki chegaralari bir xil yoki yo‘qligini aniqlaydi. Algoritmning o‘rtacha ishlash vaqti har doim pastki va yuqori chegaralar orasida bo‘ladi.

Agar yuqori chegara (**O**) va quyi chegara (**Ω**) bir xil natijani bersa, u holda Θ ham bir xil o‘sish tezligiga ega bo‘ladi.

Misol tariqasida $f(n) = 10n + n$ ifodani qaraylik.

U holda uning aniq yuqori chegarasi $g(n)$ – $\mathbf{O}(n)$ bo‘ladi. Eng yaxshi o‘sish tezligi g(n) = $\mathbf{O}(n)$.

Uning aniq quyi chegarasi $g(n) - \mathbf{\Omega}(n)$ bo‘ladi. Eng yomon o‘sish tezligi $g(n) = \mathbf{\Omega}(n)$.

Ushbu misolda eng yaxshi va eng yomon o‘sish tezliklari bir xil. Natijada, o‘rtacha holat ham bir xil bo‘ladi.

Berilgan funksiya (algoritm) uchun, agar **O** va **Ω** baholar mos kelmasa, u holda o‘rtacha o‘sish tezligi mos kelmasligi mumkin. Bunday holda, biz barcha mumkin bo‘lgan vaqt baholarini qarab chiqishimiz va ularning o‘rtacha qiymatini hisoblashimiz kerak bo‘ladi.

Endi **Θ** – ***o‘rtacha baho*** ta'rifini beraylik.

$\Theta(g(n)) = \{f(n)$: $\exists\, c_1, c_2$ va n_0 musbat o‘zgarmaslar mavjudki, barcha $n > n_0$ uchun $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ bo‘ladi$\}$. $g(n) - f(n)$ uchun asimptotik aniq chegara (baho) bo‘ladi. Θ(g(n)) – g(n) kabi o‘sish tartibga ega funksiyalar sinfini ifodalaydi.

Quyida Θ – o‘rtacha baho uchun misollarni ko‘rib chiqamiz:

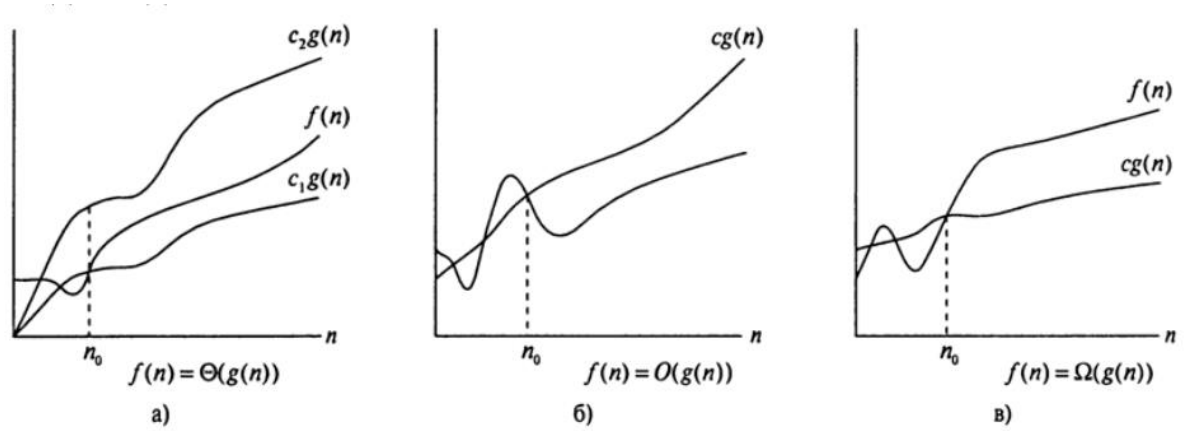
1-misol. $f(n)=n^2/2-n/2$ uchun Θ -bahoni toping.

Yechish: barcha $n \ge 2$ uchun $n^2/5 \le n^2/2-n/2 \le n^2$, demak $f(n) = \Theta(n^2)$ bo‘lib, $c_1=1/5$, $c_2=1$, $n_0=2$ ga teng bo‘ladi.

2-misol: $n \ne \Theta(n^2)$ ekanligini isbotlang.

$c_1 n^2 \le n \le c_2 n^2$ munosabati faqat $n \le 1/c_1$ uchun to‘g‘ri, lekin n o‘zgarmasdan kichik bo‘lishi mumkin emas, shuning uchun $n \ne \Theta(n^2)$

Quyidagi rasmda baholarning grafik ko‘rinishi ko‘rsatilgan:



Rasm №1. Baholarning grafik tasviri.

Ushbu baholashlardan tashqari yana boshqa o($g(n)$) baholash ham mavjud bo‘lib, u quyidagicha aniqlanadi:

o($g(n)$) = {$f(n)$: har qanday c o‘zgarmas uchun shunday musbat n_0 son mavjud bo‘lib, barcha $n > n_0$ uchun $0 \leq f(n) \leq cg(n)$ o‘rinli bo‘ladi}. U holda $g(n) – f(n)$ uchun asimptotik aniq yuqori baho bo‘ladi.

Tahlil qilish uchun (eng yaxshi, eng yomon va o‘rtacha holat) biz yuqori chegara (O) va pastki chegara (Ω) va o‘rtacha ishlash vaqtini (Θ) berishga harakat qilamiz. Ba'zan berilgan funksiya (algoritm) uchun yuqori chegara (O) va pastki chegara (Ω) va o‘rtacha ishlash vaqtini (Θ) olishning har doim ham imkoni bo‘lmaydi.

Agar biz algoritmning eng yaxshi versiyasini muhokama qilsak, biz yuqori chegara (O) va pastki chegara (Ω) va o‘rtacha ishlash vaqtini (Θ) berishga harakat qilamiz.

Keyinchalik biz odatda yuqori chegaraga (O) e'tibor qaratamiz, chunki algoritmning pastki chegarasi (Ω) amaliy ahamiyatga ega emas va biz undan yuqori va pastki chegaralar mos bo‘lgan hollarda foydalanamiz.

Yuqoridagi muhokamadan shuni ko‘rish mumkinki, har bir holatda, berilgan $f(n)$ funksiya uchun n ning yetarlicha katta qiymatlarida yaqinlashadigan boshqa $g(n)$ funksiyani topishga harakat qilamiz. Demak, $g(n)$ ham $f(n)$ ga n ning katta qiymatlarida yaqinlashadigan egri chiziqdir.

Matematikada bunday egri chiziqni asimptotik egri chiziq deb ataymiz. Boshqacha qilib aytganda, $g(n) – f(n)$ uchun asimptotik egri chiziqdir. Shu sababga ko‘ra algoritmlarni tahlil qilishni asimptotik tahlil deb ataymiz.

Asimptotik baholarning asosiy xossalari:

• Tranzitivlik: $f(n) = \Theta(g(n))$ va $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$. Bu O va Ω baholar uchun ham amal qiladi;

• Refleksivlik: $f(n) = \Theta(f(n))$. Bu O va Ω baholar uchun ham amal qiladi;

• Simmetriyalik: $f(n) = \Theta(g(n))$ bo‘ladi, agar $g(n) = \Theta(f(n))$ bo‘lsa;

• Simmetriya transpozitsiyasi: $f(n) = O(g(n))$ agar $g(n) = \Omega(f(n))$ bo‘lsa;

• Agar $f(n)$ har qanday doimiy $k > 0$ uchun $O(kg(n))$ da bo‘lsa, u holda $f(n)$ $O(g(n))$ da bo‘ladi;

• Agar $f_1(n)$ $O(g_1(n))$ va $f_2(n)$ $O(g_2(n))$ da bo‘lsa, u holda $(f_1 + f_2)(n)$ $O(max(g_1(n), g_2(n)))$ da bo‘ladi;

• Agar $f_1(n)$ $O(g_1(n))$ va $f_2(n)$ $O(g_2(n))$ da bo‘lsa, u holda $f_1(n)$ $f_2(n)$ $O(g_1(n)g_2(n))$ da bo‘ladi.

2 - bob bo‘yicha nazorat savollari

1. Bajarilish vaqti tahlili.
2. Algoritmlarni solishtirish.
3. Algoritmlarni o‘sish tezligi.
4. Tahlil turlari.
5. Asimptotik baholar va baholash turlari.

3. Bog‘langan ro‘yxatlar

Bog‘langan ro‘yxat - bu berilganlar to‘plamini saqlash uchun ishlatiladigan berilganlar tuzilmasi hisoblanadi. Bog‘langan ro‘yxat quyidagi xususiyatlarga ega:

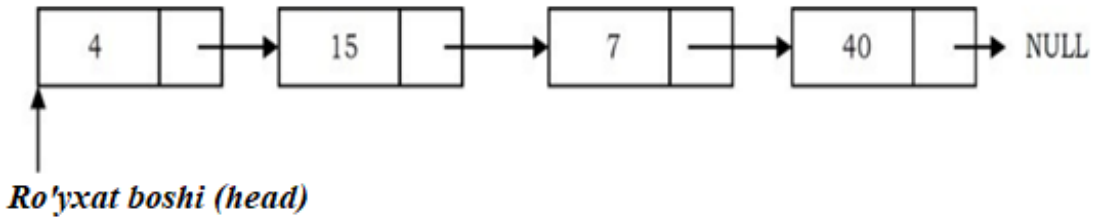
• Ketma-ket elementlar ko‘rsatkichlar orqali bog‘lanadi;

• Oxirgi element NULL ga ishora qiladi;

• Dasturni bajarish jarayonida hajmi kattalashishi yoki kichrayishi mumkin;

• Xoxlagancha kengaytirish mumkin (tizim xotirasi tugamaguncha);

• Xotirani behuda sarflamaydi (lekin ko‘rsatkichlar uchun qo‘shimcha xotira talab qiladi). Ro‘yxat o‘sishi bilan xotira ajratiladi.



Bog‘langan ro‘yxat BAT bo‘lib, quyidagi amallarga ega:

• Kiritish: elementni ro‘yxatga kiritadi (boshi, oxiri, o‘rtasiga);

• O‘chirish: belgilangan elementni ro‘yxatdan o‘chiradi;

Bog‘langan ro‘yxatlar bilan qo‘shimcha amallar:

• Ro‘yxatning barcha elementlarini o‘chirish (ro‘yxatni o‘chiradi);

• Ro‘yxatdagi elementlar sonini aniqlash;

• Ro‘yxatdagi n-tugunni topish;

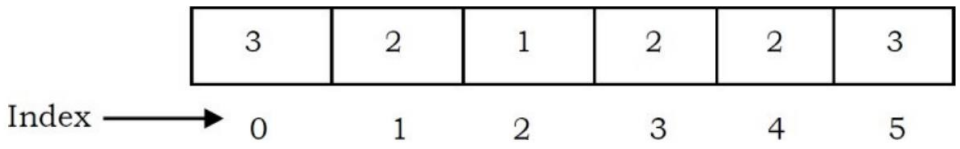
• Ro‘yxatda ko‘rsatilgan qiymatli berilganlar joylashgan tugunni topish.

Bog‘langan ro‘yxatlar va massivlar o‘rtasidagi farqni tushunish muhimdir.

Ushbu ikkita berilganlar tuzilmalari, ya’ni bog‘langan ro‘yxatlar va massivlar berilganlar to‘plamini saqlash uchun ishlatiladi va ikkalasi ham bir xil maqsadda foydalanilganligi sababli, qaysi hollarda massivlardan va qaysi hollarda ro‘yxatlardan foydalanish maqsadga muvofiqligini tushunish kerak.

3.1. Massiv

Massiv elementlarini saqlash uchun berilgan o‘lchamdagi bitta uzluksiz xotira bloki ajratiladi. Massiv elementlariga ma'lum bir element indeksi yordamida kirish mumkin. Kirish vaqti doimiy.



Massiv elementiga kirish uchun element manzilini bazaviyga nisbatan siljishni hisoblash kifoya, ya'ni

*element manzili = asosiy manzil + indeks * element hajmi.*

Bu jarayon bitta ko‘paytirish va bitta qo‘shishni talab qiladi. Ushbu ikki amalni bajarish doimiy vaqtni talab qilganligi sababli, massivga kirish doimiy vaqtda amalga oshirilishi mumkinligini aytishimiz mumkin.

Massivlarning afzalliklari:

• Oddiy va ishlatish uchun qulay;

• Elementlarga tezroq kirish (doimiy kirish).

Massivlarning kamchiliklari:

• Massiv bo‘sh bo‘lsa ham barcha kerakli xotirani oldindan ajratadi;

• Statik massivning fiksirlangan (o‘rnatilgan) o‘lchami (uni ishlatishdan oldin massiv hajmini ko‘rsatish kerak);

• Bitta uzlukzik xotira bloki ajratilishi, ya'ni belgilangan o‘lchamdagi uzluksiz xotira maydoni bo‘lmasligi mumkin (agar massiv hajmi yetarlicha katta bo‘lsa);

• Berilgan o‘ringa yangi qiymatni kiritishning murakkabligi, ya’ni elementni kiritish uchun biz mavjud elementlarni siljitishimiz kerak va shundan keyingina biz yangi qiymatni kiritishimiz mumkin va agar biz element qo‘shmoqchi bo‘lgan o‘rin massiv boshida bo‘lsa, u holda ko‘p amal bajarishga to‘g‘ri keladi.

3.2. Dinamik massivlar

Dinamik massiv o‘zgaruvchan o‘lchamdagi ixtiyoriy kirish mumkin bo‘lgan berilganlar tuzilmasidir. Dinamik massivlar dastur bajarilishi vaqtida yaratiladi va kerak bo‘lganda o‘lchamlarini o‘zgartirish mumkin.

Dastlab, ma'lum o‘lchamdagi massiv yaratiladi, bu massiv to‘lgandan so‘ng, dastlabki massivdan ikki baravar kattaroq yangi massiv yaratiladi. Xuddi shunday, agar massivdagi elementlar yarmidan kam bo‘lsa, siz massiv hajmini yarmiga qisqartirishingiz mumkin. Kamchiliklari va afzalliklari statik massivdagi kabi bo‘ladi.

3.3. Bog‘langan ro‘yxatlar

Bog‘langan ro‘yxatlarning afzalligi shundaki, ular doimiy ravishda kengayishi mumkin. Massiv yaratish uchun biz ma'lum miqdordagi elementlar uchun xotira ajratishimiz kerak. To‘lgan massivga qo‘shimcha elementlar qo‘shish uchun biz yangi massiv yaratishimiz va eski massivni yangi massivga nusxalashimiz kerak. Bu uzoq vaqt talab qilishi mumkin. Albatta, biz darhol juda ko‘p joy ajratishimiz mumkin, ammo bu xotirani behuda sarflashdir.

Bog‘langan ro‘yxatda biz yangi element uchun xotiradan joy ajratgan holda nusxa ko‘chirishsiz va elementlar joylashuvini qayta o‘zgartirmasdan osongina qo‘shishimiz mumkin.

Bog‘langan ro‘yxatlar bilan bog‘liq muammolar (kamchiliklari). Bog‘langan ro‘yxatlar bilan bog‘liq bir qator muammolar mavjud.

Bog‘langan ro‘yxatlarning asosiy kamchiliklari bitta alohida elementga kirish vaqtidir. Bog‘langan ro‘yxatlarda biror elementga kirish uchun eng yomon holatda O(*n*) vaqt talab qilinadi.

Massiv ixtiyoriy, ya’ni har qanday massiv elementiga kirish uchun O(1) vaqt ketadi. Massivlarning kirish vaqtidagi yana bir afzalligi xotiradagi fazoviy joylashuvidir. Massivlar uzluksiz xotira bloklari sifatida aniqlanadi, shuning uchun massivdagi har qanday element qo‘shnilariga jismonan qo‘shni bo‘ladi. Bu zamonaviy protsessor keshlash usullaridan foydalanganda sezilarli foyda keltiradi.

Ro‘yxatlarda xotiraning dinamik taqsimlanishi katta afzallik bo‘lsada, berilganlarni saqlash va olish uchun qo‘shimcha xarajatlar qaysidir ma'noda

ro‘yxat samaradorligini kamaytirishi mumkin. Agar oxirgi element olib tashlansa, oxirgidan oldingi elementga NULL qiymatni o‘z ichiga olgan ko‘rsatgichga o‘zgartirish kerak. Bu oxirgidan oldingi havolani va uning ko‘rsatgichini NULL ga o‘zgartirish uchun ro‘yxatni bosib o‘tishni talab qiladi. Nihoyat, bog‘langan ro‘yxatlar qo‘shimcha ko‘rsatgichlar uchun ortiqcha xotira sarflaydi.

Bog‘langan ro‘yxatlarni massivlar va dinamik massivlar bilan taqqoslash

Quyidagi jadvalda murakkablikning qiyosiy tahlili ko‘rsatilgan:

Jadval №2. Murakkablikning qiyosiy tahlili.

Parametrlar	**Bo‘glangan ro‘yxat**	**Massiv**	**Dinamik massiv**
Indekslash (elementga kirish)	O(*n*)	O(1)	O(1)
Boshidan elemetni joylash va olib tashlash	O(1)	O(*n*) (elementlarni siljitish uchun)	O(*n*) (elementlarni siljitish uchun)
Oxiriga elementni joylash	O(*n*)	O(1) massiv to‘liq bo‘lmasa	O(1) massiv to‘liq bo‘lmasa O(*n*) massiv to‘lgan bo‘lsa
Oxiridan elementni olib tashlash	O(*n*)	O(1)	O(*n*)
O‘rtaga element joylash	O(*n*)	O(*n*) (elementlarni siljitish uchun)	O(*n*) (elementlarni siljitish uchun)
O‘rtadan elementni olib tashlash	O(*n*)	O(*n*) (elementlarni siljitish uchun)	O(*n*) (har bir elementni siljitish uchun)
Qo‘shimcha xotira maydoni	O(*n*) ko‘rsatkichlar uchun	0	O(*n*)

3.4. Bir bog‘lamli ro‘yxatlar

Bir bog‘lamli ro‘yxatlar bir qator tugunlardan iborat. Har bir tugun keyingi elementga ko‘rsatgichga ega. Oxirgi element ko‘rsatkich qiymati *null* bo‘ladi.

Quyida bog‘langan ro‘yxat tugunining turi e’lon qilingan:

```
public class Node<T> where T : IComparable
{
  public Node(T data)
  {
    Data = data;
  }
  public T Data { get; set; }
  public Node<T> Next { get; set; }
  public override string ToString() => Data.ToString();
}
```

Node sinfi umumiydir, shuning uchun u har qanday turdagi ma'lumotlarni saqlashi mumkin. *Data* xususiyati berilganlarni saqlash uchun ishlatiladi. *Next* xususiyati keyingi tugunga havola uchun aniqlangan.

Keyinchalik, biz ro‘yxat sinfining o‘zini aniqlaymiz:

```
using System;
using System.Collections.Generic;
using System.Text;
namespace LList
{
  public class LList<T> where T : IComparable
  {
    public Node<T> head; // bosh/birinchi element
  }
}
```

Bir bog‘lamli ro‘yxatning asosiy amallari:

- Ro‘yxat bo‘ylab harakatlanish;
- Ro‘yxatga element qo‘shish;
- Ro‘yxatdan elementni olib tashlash.

3.4.1. Bir bog‘lamli ro‘yxat bo‘ylab o‘tish.

Ro‘yxat bo‘ylab o‘tish uchun quyidagi ketma-ketlikdagi amallar bajariladi:

- Joriy tugun sifatida ro‘yxatning bosh elementini tanlanadi;
- Joriy tugun ko‘rsatkich maydoniga qiymat sifatida joriy tugunning keyingi tugun ko‘rsatkich maydonini uzatish;
- keyingi ko‘rsatkich *null* ni ko‘rsatsa, to‘xtatish.

`Count()` usuli kiruvchi qiymat sifatida bog‘langan ro‘yxatni qabul qiladi va ro‘yxatdagi tugunlar sonini qaytaradi.

```
public int Count()
{
  int count = 0;
  Node<T> current = head;
  while (current != null)
  {
    count++;
    current = current.Next;
  }
  return count;
}
```

`Print()` usuli barcha tugunlar qiymatlarini konsol ekraniga chiqaradi:

```
public void Print()
{
  Node<T> current = head;
  while (current != null)
  {
    Console.WriteLine(current.Data);
    current = current.Next;
  }
}
```

Bu ikkala usulni vaqt bo‘yicha murakkabligi *n* o‘lchamli ro‘yxatni skanerlash (birma-bir o‘tib chiqish) uchun O(*n*) ga teng bo‘ladi.

Xotira sarfi bo‘yicha esa O(1) ga teng, faqat vaqtinchalik o‘zgaruvchilarni yaratish uchun resurs ketadi.

3.4.2. Bir bog‘lamli ro‘yxatga element qo‘shish

Bir bog‘lamli ro‘yxatga element qo‘shish uchta holatda bo‘ladi:

• Ro‘yxat boshiga yangi tugun qo‘shish;

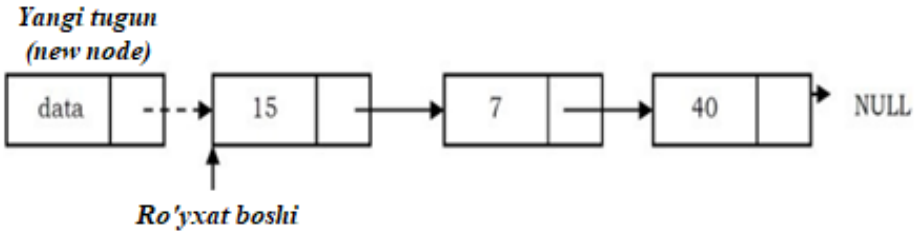
• Ro‘yxat oxiriga yangi tugun qo‘shish;

• Yangi tugunni ro‘yxatning o‘rtasiga (qandaydir *p*- tasodifiy o‘ringa) qo‘shish. Bu holda yangi element bog‘langan ro‘yxatdagi *p*-o‘rindagi elementdan oldin qo‘shiladi.

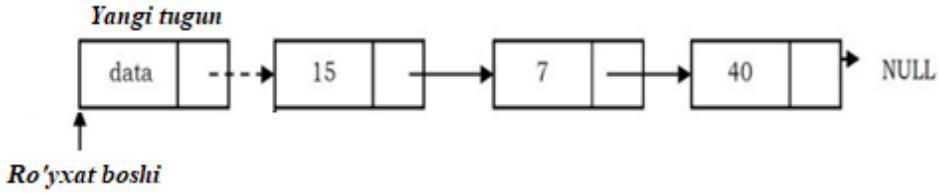
Yangi tugunni bir bo‘g‘lamli ro‘yxatning boshiga qo‘shish

Bu holda joriy bosh tugundan oldin yangi tugun qo‘shiladi. Qo‘shish ikki bosqichda amalga oshiriladi:

1. Dastlab yangi tugun yaratiladi va berilgan qiymat yoziladi va yangi tugunning keyingi tugunni bog‘lovchi maydoniga ko‘rsatgichda joriy bosh tugun yoziladi.



2. Yangi tugunning manzili bosh tugun ko‘rsatkichiga kiritiladi.



Ro‘yxat boshiga yangi tugun qo‘shish usuli:

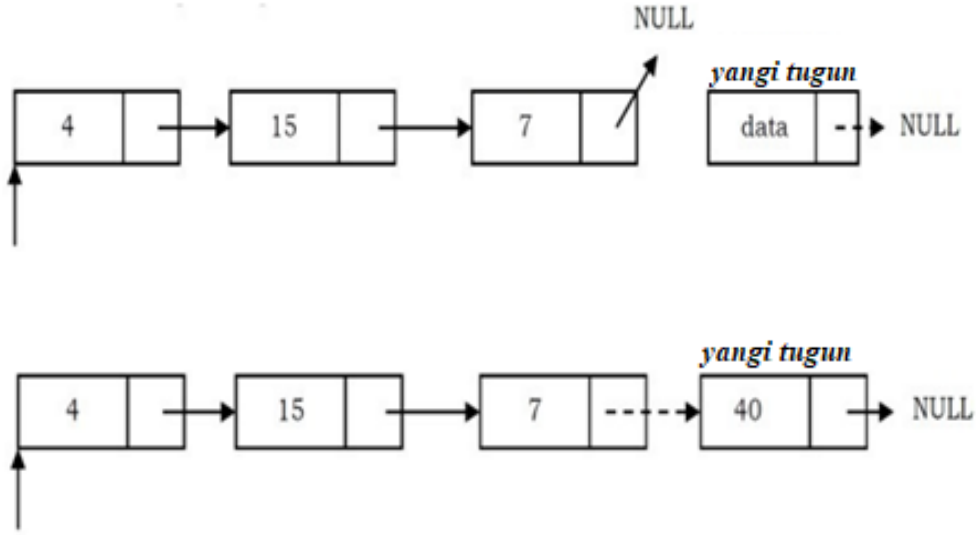
```
public void AppendFirst(T data)
{
  Node<T> node = new Node<T>(data);
  node.Next = head;
  head = node;  }
```

Bir bog‘lamli ro‘yxatning oxiriga yangi tugun qo‘shish

Bu holda biz ikkita ko‘rsatgichni keyingi elementga o‘zgartirishimiz kerak:

• oxirgi tugun ko‘rsatkichi yangi tugun manzilini o‘z ichiga olishi kerak;

• yangi tugun ko‘rsatkichi *null* ga teng bo‘lishi kerak.



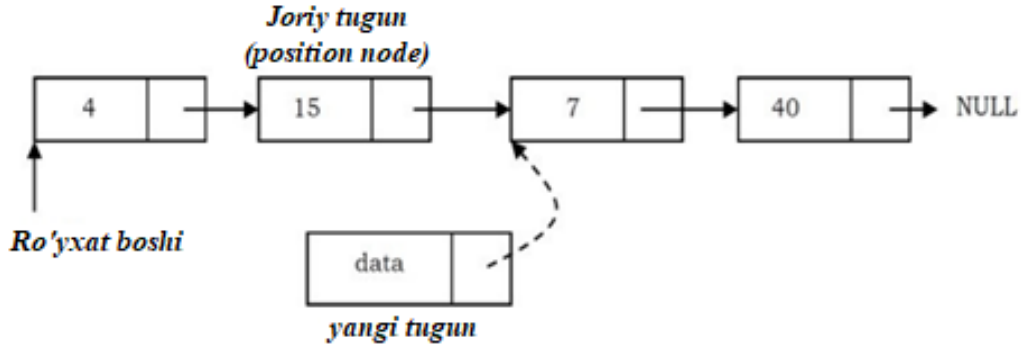

```
public void Add(T data)
{
  Node<T> node = new Node<T>(data);
  Node<T> current = head;
  if (current == null)
     head = node;
  else
  {
     while (current.Next != null)
     { current = current.Next; }
     current.Next = node;
  }
}
```

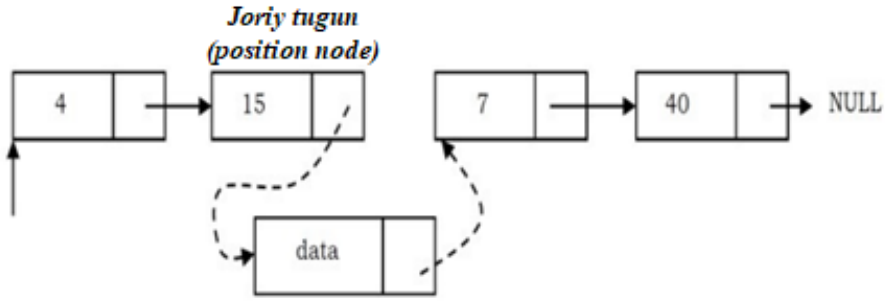
Bir bog‘lamli ro‘yxatning o‘rtasiga yangi tugunni qo‘shish

Faraz qilaylik, bizga yangi tugun kiritmoqchi bo‘lgan o‘rin berilgan.

• Agar biz 3-o‘ringa element qo‘shmoqchi bo‘lsak, biz 2-o‘rinda to‘xtaymiz. Bu biz ikkinchi tugundan keyin yangi tugunni joylashtiramiz degan ma'noni anglatadi. Yangi tugunning keyingi tugunni bog‘lovchi

ko‘rsatkich maydoniga qiymat sifatida 3-o‘rindagi tugunning manzilini beriladi. 2 - o‘rindagi tugunning keyingi tugunni bog‘lovchi ko‘rsatkich maydoniga qiymat sifatida yangi qo‘shilayotgan tugun manzili beriladi.





Yangi tugunni bir bog‘lamli ro‘yxatning biror bir berilgan *p*-o‘rniga qo‘shish usuli quyida keltirilgan:

```
public void Insert(T data, int pos)
{
  int k = 1;
  Node<T> node = new Node<T>(data);
  Node<T> current = head;
  while (current.Next != null && k < pos)
  { k++; current = current.Next; }
  node.Next = current.Next;
  current.Next = node;
}
```

Vaqt bo‘yicha murakkabligi: O(*n*), chunki eng yomon holatda biz ro‘yxat oxiriga tugun kiritishimiz kerak bo‘lishi mumkin.

Xotira bo‘yicha murakkabligi: O(1), bir nechta vaqtinchalik o‘zgaruvchilarni yaratish uchun.

3.4.3. Bir bog‘lamli ro‘yxatdan tugunni o‘chirish

Qo‘shish holatida bo‘lgani kabi, bu yerda ham uchta holat mavjud:

• Birinchi tugunni o‘chirish;

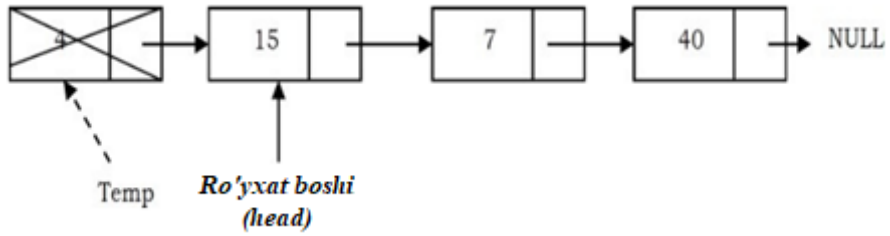
• Oxirgi tugunni o‘chirish;

• Oraliq tugunni o‘chirish.

Bir bog‘lamli ro‘yxat boshidagi tugunni o‘chirish

Birinchi tugunni (joriy bosh tugunni) o‘chirish quyidagicha amalga oshirilishi mumkin:

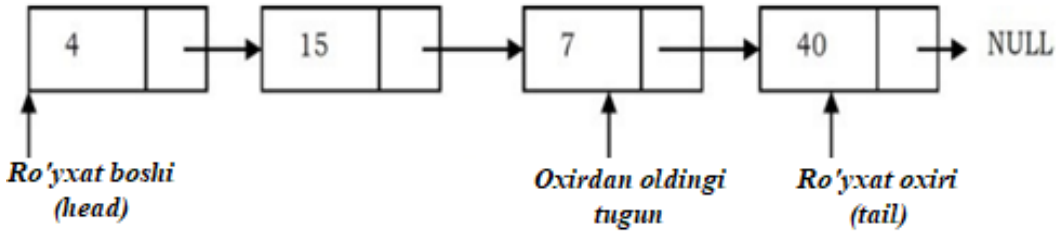
Vaqtinchalik o‘rgaruvchiga (tugunga) bosh tugun yuklanadi, Bosh tugun ko‘rsatkichi keyingi tugunga o‘tkazadi. Vaqtinchalik tugun o‘chiriladi.



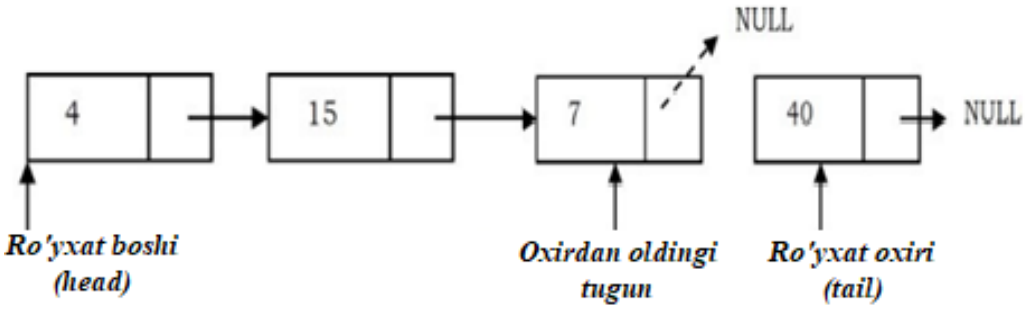
Bir bog‘lamli ro‘yxat oxiridagi tugunni o‘chirish

Bu holda oxirgi tugun ro‘yxatdan o‘chiriladi. Ushbu amal birinchi tugunni olib tashlashdan ko‘ra biroz murakkabroq, chunki algoritm oxirgidan oldingi tugunni topishi kerak. Bu amal uch bosqichda amalga oshiriladi:

• Ro‘yxat ko‘zdan kechiriladi va harakatlanayotganda oldingi tugun manzili ham saqlanadi. Ro‘yxatning oxiriga yetganda, biz ikkita ko‘rsatgichga ega bo‘lamiz, biri oxirgi tugunga, ikkinchisi esa oxirgi tugundan oldingi tugunga ishora qiladi.



• Oldingi tugunning keyingi tugun ko‘rsatkich maydoni NULL, chunki u oxirgi tuguniga aylanadi.

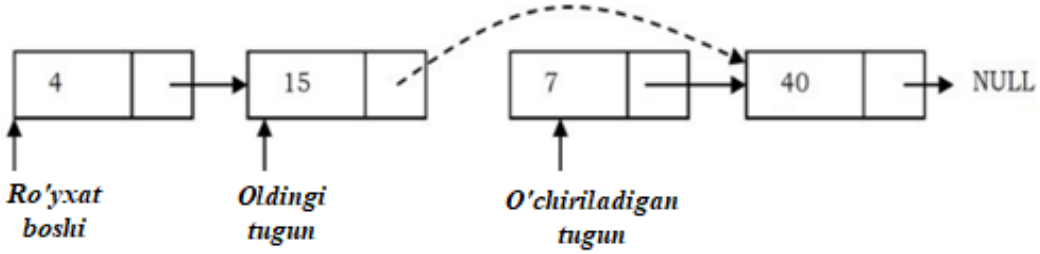


Bir bog‘lamli ro‘yxatdan oraliq tugunni o‘chirish.

Bu holda, o‘chiriladigan tugun ikki tugun orasida bo‘ladi. Bosh va oxirgi tugunlari yangilanmaydi.

O‘chirish ikki bosqichda amalga oshiriladi:

• Oldingi holatda bo‘lgani kabi, ro‘yxat bo‘ylab harakatlanayotganda oldingi va joriy tugun saqlanadi. O‘chirilishi kerak bo‘lgan tugunni topishimiz bilan oldingi tugunning ko‘rsatkich maydoni o‘chiriladigan tugunning ko‘rsatkich maydonining qiymatiga o‘zgartiriladi.



Keling, uchta holatni hisobga olgan holda, bog‘langan ro‘yxatdagi berilgan o‘rindagi tugunni o‘chirish usulini ko‘rib chiqamiz. Usul parametriga o‘chiriladigan tugun o‘rni uzatiladi:

```
public void Delete(int pos)
{
  int k = 1;
  Node<T> current = head;
  Node<T> previous = null;
  while (current.Next != null && k < pos)
  {
    k++; previous = current; current = current.Next;
  }
  if (pos <= 1)
```

```
      head = head.Next;
   else
      previous.Next = current.Next;
}
```

Vaqt bo‘yicha murakkabligi: O(n), chunki eng yomon holatda biz ro‘yxat oxirida joylashgan tugunni o‘chirishimiz kerak bo‘lishi mumkin.

Xotira sarfi bo‘yicha murakkabligi: O(1), bir nechta vaqtinchalik o‘zgaruvchilarni yaratish uchun.

Bog‘langan ro‘yxatdagi barcha tugunlarni o‘chirish

Ro‘yxatni tozalash uchun bosh tugunni tozalash kifoya.

```
public void Clear()
{
  head = null;
}
```

Bir bog‘lamli ro‘yxat uchun BAT ***1 - ilovada*** to‘liq keltirilgan.

3.5. Ikki bog‘lamli ro‘yxatlar

Ikki bog‘lamli (ikki yo‘nalishli) ro‘yxatning afzalligi shundaki, biz ro‘yxatdagi berilgan tugundan ikkala yo‘nalishda ham harakat qilishimiz mumkin.

Bir bog‘lamli ro‘yxatdagi tugunni o‘chirishda agar uning oldingi tuguniga ko‘rsatgich bo‘lmasa uni o‘chira olmaymiz. Ammo ikki bog‘lamli ro‘yxatda bizda oldingi tugunning manzili bo‘lmasa ham, tugunni o‘chirib tashlashimiz mumkin (chunki har bir tugun oldingi tugunga ishora qiluvchi chap ko‘rsatkichga ega va orqaga siljishi mumkin).

Ikki bog‘lamli ro‘yxatlarning asosiy kamchiliklari quyidagilardir:

• Har bir tugunga qo‘shimcha ko‘rsatgichlar uchun ortiqcha joy kerak bo‘ladi;

• Tugunni joylash yoki o‘chirish uchun biroz ko‘proq vaqt sarflanadi (ko‘rsatkichlar bilan ko‘proq amallar bajarilganligi sababli).

Misol sifatida, ikki bog‘lamli ro‘yxatni ko‘rib chiqamiz:

Quyida ikki bog‘lamli tugun uchun tur e’lon qilinishi berilgan:

```
using System;
using System.Collections.Generic;
using System.Text;
namespace DoubleList
{
  class DNode<T> where T : IComparable
  {
    public DNode(T data)
    {
      Data = data;
    }
    // Berilganlar
    public T Data { get; set; }
    // keyingi tugunga ko‘rsatkich
    public DNode<T> Next { get; set; }
    // oldingi tugunga ko‘rsatkich
    public DNode<T> Prev { get; set; }
    // Sinf ekzemplyarini satrga aylantirish
```

```
    public override string ToString() => Data.ToString();
  }
}
```

Ikki bog‘lamli ro‘yxat uchun BATni belgilashda ikkita ko‘rsatgichni saqlash qulay bo‘ladi, biri ro‘yxatning boshiga, ikkinchisi ro‘yxat oxiriga:

```
using System;
using System.Collections.Generic;
using System.Text;
namespace DoubleList
{
  class DList<T> where T: IComparable
  {
    // Bosh/birinchi element
    DNode<T> head;
    // Dum /oxirgi element
    DNode<T> tail;
    int count;  // Ro‘yxatdagi elementlar soni
    public int Count { get { return count; } }
    public bool IsEmpty { get { return count == 0; } }
  }
}
```

Ikki bog‘lamli ro‘yxatdagi asosiy amallar bir bog‘lamli ro‘yxatdagi kabi bo‘ladi:

- Ro‘yxat bo‘ylab harakatlanish;
- Ro‘yxatga element qo‘shish;
- Ro‘yxatdan elementni o‘chirish.

Ikki bog‘lamli ro‘yxatda elementni qo‘shish yoki o‘chirishda ro‘yxatning ikki tomonlamaligi e'tiborga olinishi kerak, chunki keyingi elementga havoladan tashqari, avvalgisiga ham havola o‘rnatish kerak bo‘ladi. Ammo shu bilan birga, biz ro‘yxatni birinchi elementdan oxirgi elementga va aksincha - oxirgi elementdan birinchi elementga o‘tish imkoniyatiga egamiz. Aks holda, ikki bog‘lamli ro‘yxat bir bog‘lamli ro‘yxatdan farq qilmaydi.

3.5.1. Ikki bog‘lamli ro‘yxat bo‘ylab o‘tish

Ro‘yxatni boshidan oxiriga va aksincha, oxiridan boshiga harakatlanishning ikkita usulini ko‘rib chiqamiz. Misol sifatida, berilgan qiymatga ega bo‘lgan tugunni qidirishni ko‘rib chiqamiz.

Ro‘yxatni ko‘rish uchun biz quyidagilarni bajaramiz:

• Joriy tugun sifatida ro‘yxatning bosh elementini o‘rnatish;

• Joriy tugundagi ko‘rsatgich maydoni qiymati bo‘yicha keyingi tugunni tanlash;

• Agar keyingi ko‘rsatkich *null*ni ko‘rsatsa yoki kerakli tugun topilsa, to‘xtash.

Print() usuli bir bog‘lamli ro‘yxatdagi kabi bo‘lib, ro‘yxat elementlarini ro‘yxat boshidan oxiriga qadar chiqaradi:

```
public void Print()
{
  if (!IsEmpty)
  {
    DNode<T> current = head;
    while (current != null)
    {
      Console.WriteLine(current.Data);
      current = current.Next;
    }
  }
  else
    Console.WriteLine("Ro‘yxat bo‘sh");
}
```

PrintBack() usuli ro‘yxat elementlarini ro‘yxat oxiridan boshiga qadar chiqaradi:

```
public void PrintBack()
{
  if (!IsEmpty)
  {
    DNode<T> current = tail;
    while (current != null)
```

```
    {
      Console.WriteLine(current.Data);
      current = current.Prev;
    }
  }
  else
    Console.WriteLine("Ro‘yxat bo‘sh");
}
```

ContainsH() usuli kiruvchi berilganlar sifatida ikki bog‘lamli ro‘yxatni oladi va birinchi elementdan oxirgisigacha tugunni qidiradi va agar element topilsa "true", aks holda "false" qiymatini qaytaradi.

```
public bool ContainsH(T data)
{
  DNode<T> current = head;
  while (current != null)
  {
    if (current.Data.Equals(data))
      return true;
    current = current.Next;
  }
  return false;
}
```

ContainsT() usuli ro‘yxatni oxiridan boshigacha aylanib o‘tadi:

```
public bool ContainsT(T data)
{
  DNode<T> current = tail;
  while (current != null)
  {
    if (current.Data.Equals(data))
      return true;
    current = current.Prev;
  }
  return false;
}
```

Vaqt bo‘yicha murakkabligi O(*n*) ga teng, *n* o‘lchamli ro‘yxat bo‘ylab o‘tish uchun sarflanadi.

Xotira sarfi bo‘yicha esa O(1) ga teng, faqat vaqtinchalik o‘zgaruvchilarni yaratish uchun resurs ketadi.

3.5.2. Ikki bog‘lamli ro‘yxatga yangi tugun qo‘shish

Ikki bog‘lamli ro‘yxatga yangi tugun qo‘shishda uchta holat bo‘ladi (xuddi bir bog‘lamli ro‘yxat kabi):

• Ro‘yxat boshiga yangi tugun qo‘shish;

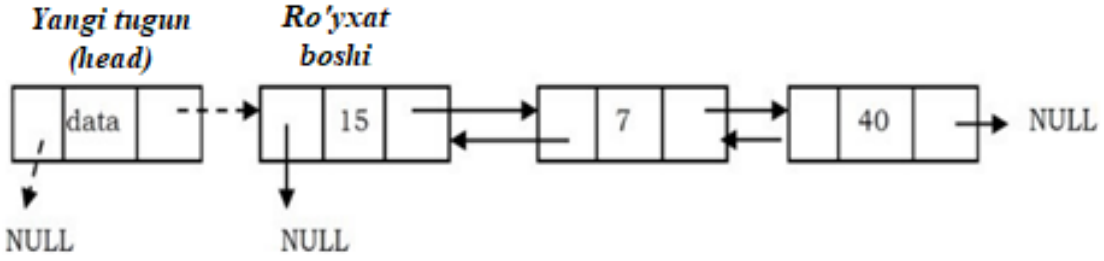
• Ro‘yxat oxiriga yangi tugun qo‘shish;

• Ro‘yxat o‘rtasiga yangi tugun qo‘shish.

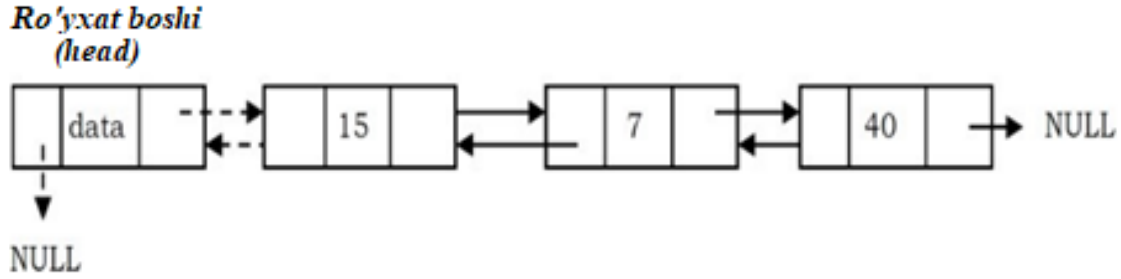
Ikki bog‘lamli ro‘yxat boshiga yangi tugunni qo‘shish

Bu holda, bosh tugundan oldin yangi tugun kiritiladi. Yangi va bosh tugun ko‘rsatkichlarini o‘zgartirish kerak va buni ikki bosqichda bajarish mumkin:

• Joriy bosh tugunni ko‘rsatish uchun yangi tugunning o‘ng ko‘rsatkichi yangilanadi (quyidagi rasmdagi nuqtali havola)



• Joriy bosh tugunning chap ko‘rsatkichi yangi tugunga ishora qiladigan tarzda yangilanadi va yangi tugun bosh tugunga aylanadi.




```
public void AddFirst(T data)
{
  DNode<T> node = new DNode<T>(data);
```

```
  if (IsEmpty)
    head = tail = node;
  else
  {
    head.Prev = node;
    node.Next = head;
    head = node;
  }
  count++;
}
```

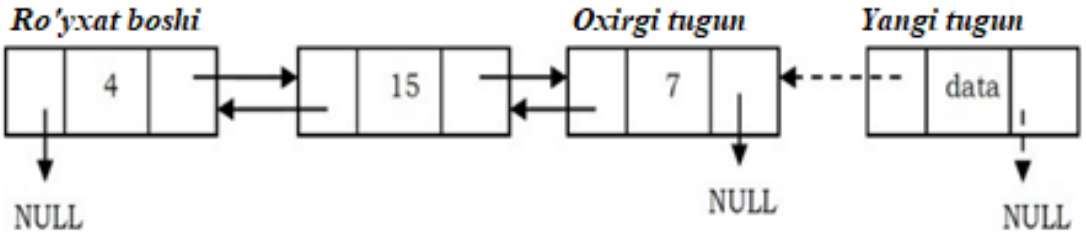
Vaqt bo‘yicha murakkabligi O(1) ga teng, chunki tugun ro‘yxatning boshiga kiritilgan.

Xotira sarfi bo‘yicha esa O(1) ga teng, faqat vaqtinchalik o‘zgaruvchilarni yaratish uchun resurs ketadi.

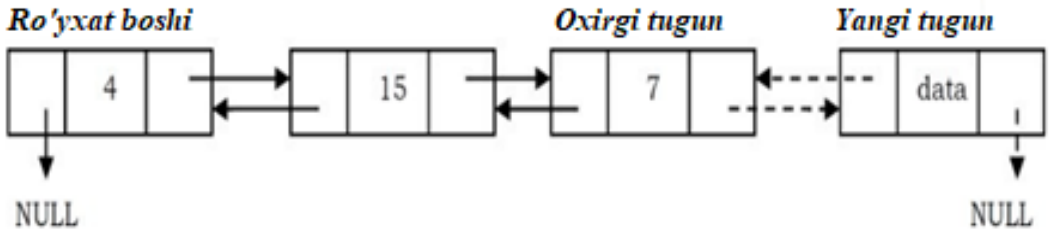
Ikki bog‘lamli ro‘yxatning oxiriga yangi tugun qo‘shish

Bu holda, dastlab yangi tugun yaratiladi.

• Yangi tugunning o‘ng ko‘rsatkichi *null*, chap ko‘rsatkich esa oxirgi tugun manziliga o‘rnatiladi.



• Oxirgi tugunning o‘ng ko‘rsatkichida yangi tugunning manzili yoziladi (yangi tugunga ko‘rsatgichning qiymati). Endi ro‘yxatning oxirgi tuguni yangi bo‘glangan tugundir.




```
public void AddLast(T data)
```

```
{
  DNode<T> node = new DNode<T>(data);
  if (IsEmpty)
    head = tail = node;
  else
  {
    tail.Next = node;
    node.Prev = tail;
    tail = node;
  }
  count++;
}
```

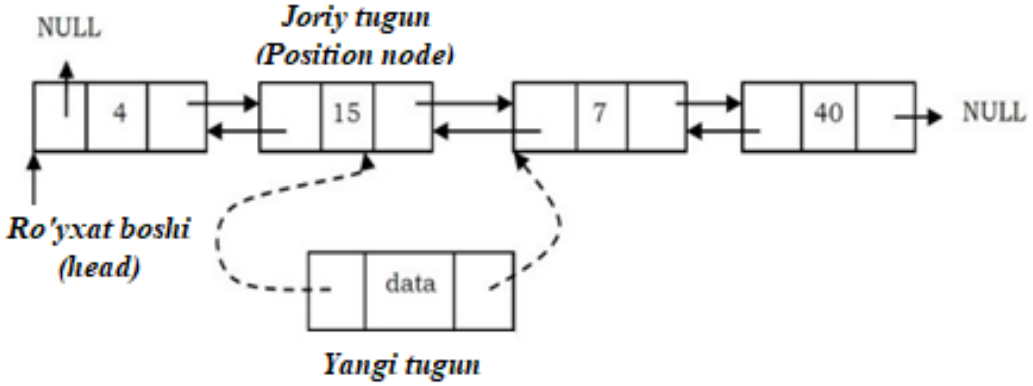
Vaqt bo‘yicha murakkabligi O(1) ga teng, chunki tugun ro‘yxatning oxiriga kiritilgan.

Xotira sarfi bo‘yicha esa O(1) ga teng, faqat vaqtinchalik o‘zgaruvchilarni yaratish uchun resurs ketadi.

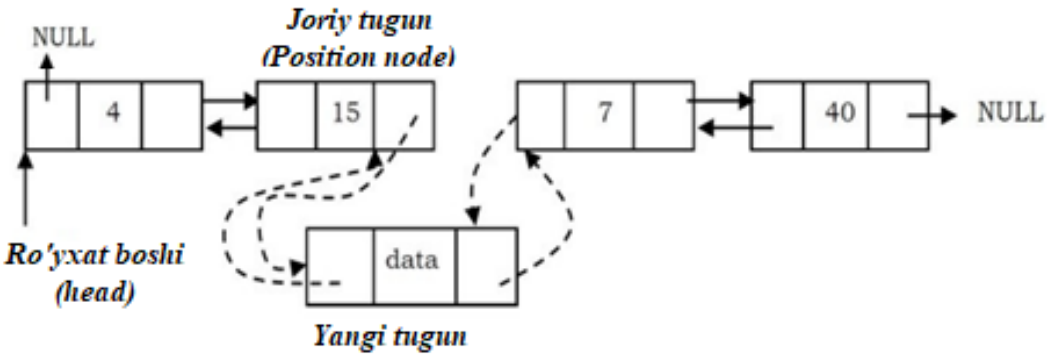
Ikki bog‘lamli ro‘yxat orasiga yangi tugun qo‘shish

Ro‘yxat ko‘rsatilgan o‘rindagi tugunga siljitiladi. Belgilangan o‘rindan so‘ng kiritiladigan yangi tugun yaratiladi.

• Yangi tugunning o‘ng ko‘rsatkichi keyingi tugunga ishora qiladi. Yangi tugunning chap ko‘rsatkichi oldingi o‘rindagi tuguniga ishora qiladi.



• Berilgan o‘rindagi tugunning o‘ng ko‘rsatkichi yangi tugunga, berilgan o‘rindagi tugundan keyingi tugunning chap ko‘rsatkichi esa yangi tugunga ishora qiladi.



Ro‘yxat boshiga nisbatan belgilangan o‘rindan keyin bog‘langan ro‘yxatga yangi tugun qo‘shadigan usulni yozamiz.

```
// Ko‘rsatilgan o‘ringa element qo‘shish
public void Insert(T data, int pos)
{
  int k = 1;
  DNode<T> node = new DNode<T>(data);
  DNode<T> current = head;
  while (current.Next != null && k < pos)
  { k++; current = current.Next; }
  node.Next = current.Next;
  node.Prev = current;
  current.Next.Prev = node;
  current.Next = node;
  count++;
}
```

Vaqt bo‘yicha murakkabligi O(n) ga teng, chunki eng yomon holatda biz ro‘yxat oxiriga tugun kiritishimiz kerak bo‘lishi mumkin.

Xotira sarfi bo‘yicha esa O(1) ga teng, faqat vaqtinchalik o‘zgaruvchilarni yaratish uchun resurs ketadi.

3.5.3. Ikki bog‘lamli ro‘yxatdan tugunni o‘chirish

Ro‘yxatga tugun qo‘shish kabi, ro‘yxatdan tugunni o‘chirishda ham uchta holat mavjud:

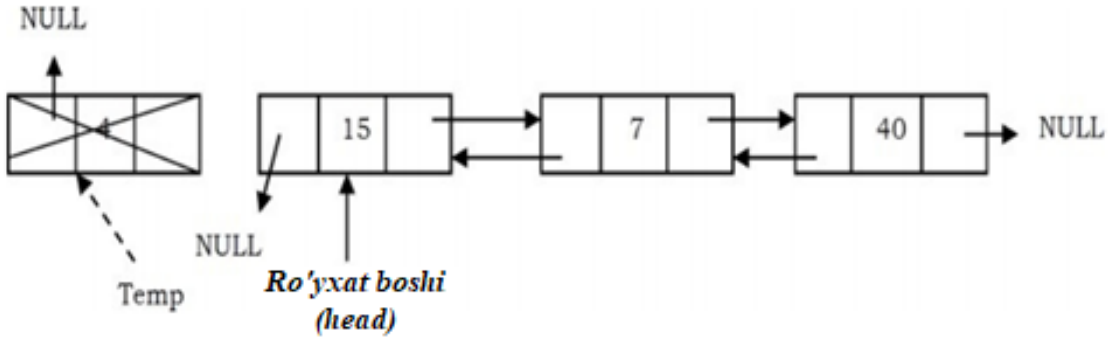
- Ro‘yxatning birinchi tugunini o‘chirish;
- Ro‘yxatning oxirgi tugunini o‘chirish;

• Ro‘yxatdan oraliq tugunni o‘chirish.

Ikki bog‘lamli ro‘yxatning birinchi tugunini o‘chirish

Birinchi tugunni (joriy bosh tugunni) o‘chirish quyidagicha amalga oshirilishi mumkin:

Bosh tugunga ko‘rsatkichni (*Head*ni qiymati) keyingi tugunga o‘tkaziladi. Yangi bosh tugunda chap ko‘rsatkich *null* ga o‘zgartiriladi.



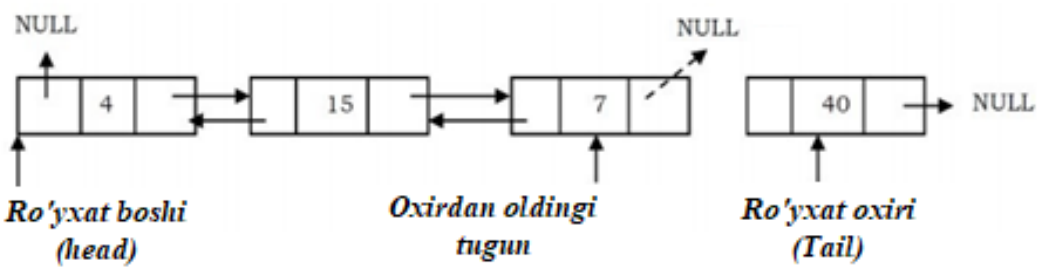

```
//Ikki bog‘lamli ro‘yxatning bosh tugunini o‘chirish usuli
public void DeleteFirst()
{
  if (!IsEmpty)
  {
    head = head.Next;
    head.Prev = null;
    count--;
  }
  else
    Console.WriteLine("Ro‘yxat bo‘sh");
}
```

Vaqt bo‘yicha murakkabligi O(1) ga teng, chunki faqat ro‘yxatning boshidan bitta tugun olib tashlanadi.

Ikki bog‘lamli ro‘yxatdagi oxirgi tugunni o‘chirish

Bu holda, oxirgi tugun ro‘yxatdan o‘chiriladi.

• Ro‘yxatning oxiridan oldingi tugunning keyingi tugun ko‘rsatkich maydoni *NULL* ga o‘zgartiriladi, chunki u endi oxirgi tuguniga aylanadi.



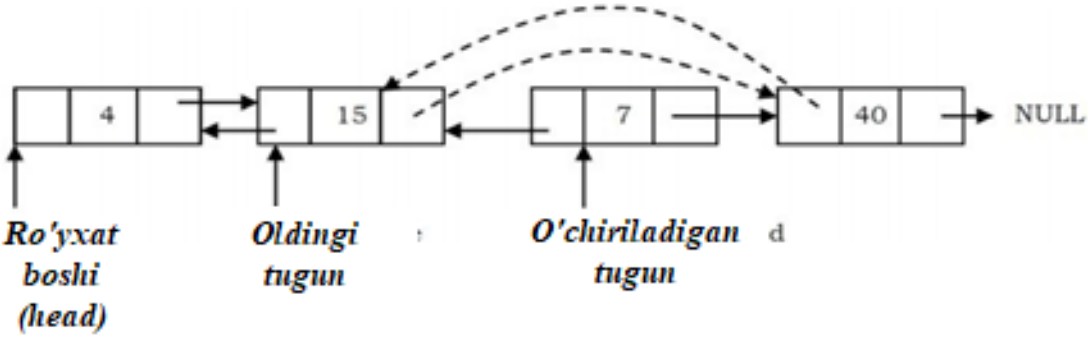

```
//Ikki bog‘lamli ro‘yxatning oxirgi tugunini o‘chirish usuli
public void DeleteLast()
{
  if (!IsEmpty)
  {
    tail = tail.Prev;
    tail.Next = null;
    count--;
  }
  else
    Console.WriteLine("Ro‘yxat bo‘sh");
  }
```

Vaqt bo‘yicha murakkabligi O(1) ga teng, chunki faqat ro‘yxatning boshidan bitta tugun olib tashlanadi.

Ikki bog‘lamli ro‘yxatdagi oraliq tugunni o‘chirish

Bu holda, o‘chiriladigan tugun ikki tugun orasida bo‘ladi. Bosh va oxirgi tugunlar yangilanmaydi.

• Bir bog‘lamli ro‘yxatda bo‘lgani kabi, ro‘yxat bo‘ylab harakatlanayotganda joriy tugunni aniqlaymiz. O‘chiriladigan tugunni topishimiz bilan biz oldingi tugunning o‘ng ko‘rsatkichini o‘chiriladigan tugunning o‘ng ko‘rsatkichi qiymatiga o‘zgartiramiz. O‘chiriladigan tugundan keyingi tugunning chap ko‘rsatkichi o‘chiriladigan tugunning chap ko‘rsatkichi qiymatini oladi.




```
// Ikki bog‘lamli ro‘yxatdagi berilgan o‘rindagi tugunni
//o‘chirish usuli
public void Delete(int pos)
{
  if (!IsEmpty)
  {
    int k = 1;
    DNode<T> current = head;
    while (current.Next != null && k < pos)
    {
      k++; current = current.Next;
    }
    if (k == 1)
    {
      head = head.Next;
      head.Prev = null;
    }
    else
    {
      current.Prev.Next = current.Next;
      current.Next.Prev = current.Prev;
    }
    count--;
  }
  else
    Console.WriteLine("Ro‘yxat bo‘sh"); }
```

Vaqt bo‘yicha murakkabligi O(n) ga teng, chunki eng yomon holatda biz ro‘yxat oxiridan oldingi tugunni olib tashlashimiz mumkin.

Xotira sarfi bo‘yicha esa O(1) ga teng, faqat vaqtinchalik o‘zgaruvchilarni yaratish uchun resurs ketadi.

Ikki bo‘glamli ro‘yxatni tozalash (barcha tugunlarini o‘chirish)

Ikki bo‘glamli ro‘yxatni tozalash uchun bosh tugunni tozalasak ya’ni unga *NULL* qiymat bersak yetarli.

```
// очистка списка
public void Clear()
{
  head = null; tail = null; count = 0;
}
```

Ikki bog‘lamli ro‘yxat uchun BAT ning to‘liq kodi ***2 - ilova***da keltirilgan.

3 – bob bo‘yicha nazorat savollari

1. Statik va dinamik massivlar farqi.
2. Massivlarning afzalliklari.
3. Massivlarning kamchiliklari.
4. Bog‘langan ro‘yxatlar.
5. Bog‘langan ro‘yxatlar bilan bog‘liq muammolar.
6. Bog‘langan ro‘yxatlarni massivlar va dinamik massivlar bilan taqqoslash.
7. Bir bog‘lamli ro‘yxatlar.
8. Bir bog‘lamli ro‘yxatning asosiy amallari.
9. Bir bo‘g‘lamli ro‘yxatga element qo‘shish.
10. Bir bog‘lamli ro‘yxatdan tugunni o‘chirish.
11. Ikki bog‘lamli ro‘yxatlar.
12. Ikki bog‘lamli ro‘yxat bo‘ylab o‘tish.
13. Ikki bog‘lamli ro‘yxatga yangi tugun qo‘shish.
14. Ikki bog‘lamli ro‘yxatda tugunni o‘chirish.

4. Stek

Stek - berilganlarni saqlash uchun ishlatiladigan oddiy berilganlar tuzilmasi bo‘lib, stekda berilganlarning kelish tartibi muhim ahamiyatga ega.

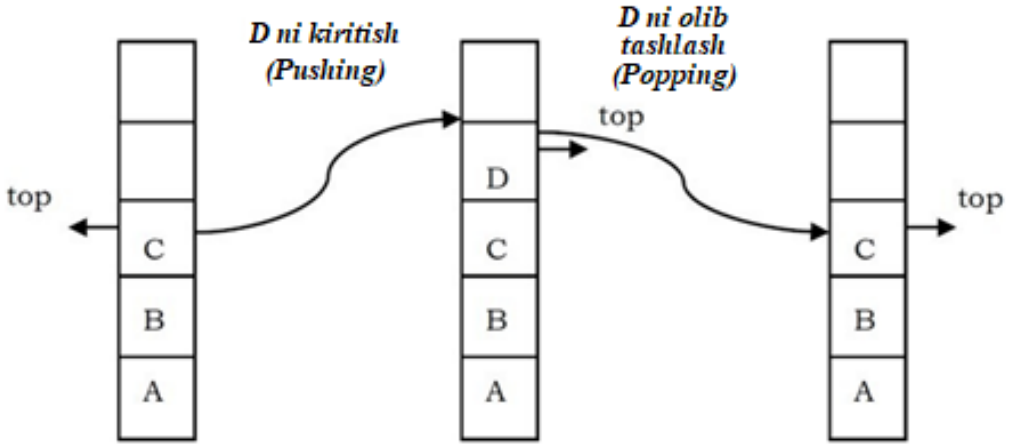
Kafeteriyadagi plastinkalar to‘plami bu stekga yaxshi misol bo‘ladi. Plastinkalar tozalangan va tepaga joylashtirilgani uchun stekga qo‘shiladi. Stekning tepasidan boshlab plastinka olinishi kerak. Stekga joylashtirilgan birinchi plastinka oxirgi ishlatiladigan plastinka hisoblanadi.

Ta'rif: **stek - qo‘shish va o‘chirish cho‘qqi deb ataluvchi bir uchida amalga oshiriladigan tartiblangan ro‘yxatdir.**

Oxirgi kiritilgan element birinchi bo‘lib olib tashlanadi. Stek "oxirgi kiruvchi, birinchi chiqadi" tamoyili asosida ishlaydi ya’ni LIFO (Last In First Out).

Stek holatini o‘zgartiruvchi ikkita jarayonga maxsus nomlar beriladi. Elementni stekga qo‘shish jarayoni **push**, elementni stekdan olib tashlash jarayoni **pop** deyiladi. Elementni bo‘sh stekdan olishga urinishni **stekni bo‘shatish harakati** deb ataladi, yangi elementni to‘lgan stekga qo‘shishga urinish esa **to‘lib ketish holati** deyiladi. Odatda bunday holatlar istisno sifatida qabul qilinadi.

Masalan quyidagi rasm orqali stekni tushunishga harakat qilamiz:



Stekni qo‘llanilishiga doir misol

Ofisdagi ish kunini ko‘rib chiqing. Aytaylik, dasturchi uzoq muddatli loyiha ustida ishlayapti. Keyin loyiha menejeri unga muhimroq bo‘lgan

yangi vazifani beradi. Dasturchi uzoq muddatli loyihani chetga surib, yangi vazifa ustida ishlay boshlaydi. Shu payt telefon qo‘ng‘irog‘i jiringlab qoldi va bu eng muhim vazifadir, chunki unga darhol javob berish kerak. Dasturchi joriy vazifani bajarishni vaqtincha to‘xtatadi va telefonga javob beradi. Qo‘ng‘iroq tugagach, vaqtincha chetga surib qo‘yilgan vazifa yana davom ettiriladi va shu tarzda ish davom etadi. Yana boshqa qo‘ng‘iroqqa javob berishga to‘g‘ri kelsa xuddi oxirgi marta bajargan amallar takrorlanishi mumkin, ammo oxir-oqibat yangi vazifa tugaydi va dasturchi bir chetga surib qo‘ygan uzoq muddatli loyihaga qaytishi mumkin.

4.1. Stek BAT

Stek quyidagi amallarni (usullarni) amalga oshiradi:

• **int Count**: stekda saqlangan elementlar sonini qaytaruvchi xususiyat;

•**bool IsEmpty**: stekning bo‘sh yoki bo‘sh emasligini aniqlaydigan xususiyat.

Stekning asosiy amallari:

• **Push(T data)**: data berilganini stekga qo‘shish.

• **T Pop()**: stekdan oxirgi kiritilgan elementni olib tashlaydi va qiymat sifatida qaytaradi.

Stekning yordamchi amallari va xususiyatlari:

• **T Peek()**: oxirgi kiritilgan elementni o‘chirib tashlamasdan qaytaradi;

• **Print()** stekning barcha elementlarini chop etadi.

Istisnolar

Stek ustida amallarni bajarishga urinish ba'zan istisno deb ataladigan xatoga olib kelishi mumkin. Ya’ni istisnolar stek ustida bajarib bo‘lmaydigan harakatlar amalga oshirilganda yuzaga keladi. Agar stek bo‘sh bo‘lsa, *Pop* va *Peek* amallarini bajarib bo‘lmaydi. Bo‘sh stek bilan *Pop* yoki *Peek* amalini bajarishga urinish istisnoga olib keladi. To‘liq stekga *Push* orqali yangi element qo‘shishga urinish istisnoga olib keladi.

Stek qo‘llaniladigan masalalar

Quyida steklar muhim rol o‘ynaydigan ba'zi masalalar mavjud.

To‘g‘ridan-to‘g‘ri stekdan foydalanadigan masalalar:

• Belgilarni muvozanatlash (masalan, qavslar);

• Infikslarni postfikslarga aylantirish;

• Postfiks ifodasini baholash;

• Funksiya chaqiruvlarini amalga oshirish (jumladan, rekursiya);

• Bo‘shliqlarni topish (birja bozorlarida bo‘shliqlarni qidirish);

• Web-brauzerdagi sahifalarga tashriflar tarixi (qaytish tugmalari);

• Matn muharriridagi bajarilgan amallar ketma-ketligini bekor qilish (Ctrl Z);

• HTML va XMLdagi teglarning qo‘llanilishi.

Stekdan bilvosita foydalaniladigan masalalar:

• Boshqa algoritmlar uchun yordamchi berilganlar tuzilmasi (masalan: daraxt bo‘ylab o‘tish algoritmlari);

• Boshqa berilganlar tuzilmalari komponenti (Misol: navbatlarni modellashtirish).

4.2. Stekni amalga oshirish

Stek BATni amalga oshirishning ko‘plab usullari mavjud. Quyida eng ko‘p qo‘llaniladigan usullar keltirilgan.

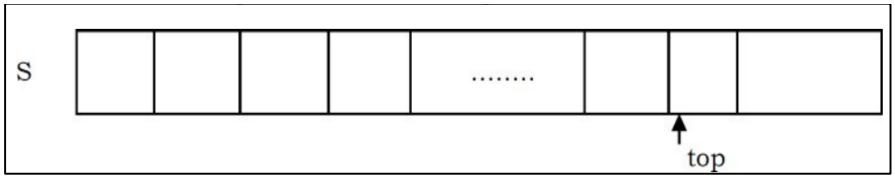
• Statik massiv asosida oddiy amalga oshirish;

• Dinamik massiv asosida amalga oshirish;

• Bog‘langan ro‘yxat asosida amalga oshirish.

4.2.1. Stekni statik massiv asosida amalga oshirish

Stek BATni amalga oshirishning ushbu turida oddiy statik massivdan foydalaniladi. Elementlar massivga chapdan o‘ngga ketma-ket qo‘shiladi va yuqori element indeksiga ishora qiluvchi o‘zgaruvchidan foydalaniladi.



Stek elementlari saqlanayotgan massiv to‘lishi mumkin va keyingi **Push** qo‘shish amali stek to‘lib ketishini anglatuvchi istisnoni yuzaga keltiradi.

Xuddi shunday, agar biz bo‘sh stekdan elementni **Pop** olib tashlashga harakat qilsak, u stek bo‘shaganligini bildiruvchi istisnoga olib keladi.

Push() va Pop() usullarni kodi quyidagicha bo‘ladi:

```
public void Push(T item)
{
// agar stek to‘la bo‘lsa istisno holatini yuzaga keltirish
  if (count == items.Length)
   throw new InvalidOperationException("Stek to‘lgan");
  items[count++] = item;
}
public T Pop()
{
// Agar stek bo‘sh bo‘lsa istisno holatini yuzaga keltirish
  if (IsEmpty)
    throw new InvalidOperationException("Stek bo‘sh");
  T item = items[--count];
  items[count] = default(T); //
  return item;
}
```

Stek BATni statik massiv orqali amalga oshirish misoli ***3.1 ilovada*** keltirilgan:

Samaradorligi:

Aytaylik *n* - stekdagi elementlar soni bo‘lsin. Ushbu o‘lchamdagi stek amallari uchun xotira va vaqt bahosi quyidagicha:

- Xotira sarfi (*n* ta kiritish amali uchun): O(*n*);
- Stekga element qo‘shish *Push*() amali uchun vaqt sarfi: O(1);
- Stekda elementni olib tashlash *Pop*() amali uchun vaqt sarfi : O(1);
- *Peek*() orqali elementni olish uchun vaqt sarfi: O(1);
- *IsEmpty* uchun vaqt sarfi: O(1);
- Stekdagi elementlar ro‘yxatini olish *Print*() amali uchun vaqt sarfi: O(*n*).

Cheklovlar:

Boshidanoq stek elementlarini saqlash uchun ishlatiladigan statik massivining maksimal hajmini aniqlash kerak va uni o‘zgartirib bo‘lmaydi. Yangi elementni to‘liq stekga qo‘shishga urinish amalga oshirishga bog‘liq istisnoni keltirib chiqaradi.

4.2.2. Stekni dinamik massiv asosida amalga oshirish

Stekni amalga oshirishning ushbu usulida stek elementlarini saqlash uchun dinamik massivga xotiradan ko‘rsatilgan o‘lchamda dinamik ravishda joy ajratiladi. Agar stek to‘la bo‘lsa, unda yangi elementni kiritishda stek hajmini oshirish kerak bo‘ladi, bu ikki usulda amalga oshirilishi mumkin:

Birinchi usul (inkrement usul): har safar yangi element kiritilganda massiv hajmini 1 taga oshirish. Bu usul juda qimmatga tushadi.

Misol uchun, $n = 1$ holat uchun stekka yangi elementni qo‘shish uchun o‘lchami 2 bo‘lgan yangi massiv yaratiladi va stekni eski massivdagi barcha elementlari yangi massivga ko‘chiriladi va yangi element massiv oxirida qo‘shiladi. $n = 2$ bo‘lgan holatda yangi elementni qo‘shish uchun 3 o‘lchamdagi yangi massiv yaratiladi va eski massivning barcha elementlarini yangi massivga nusxalash, yangi elementni oxiriga qo‘shish va hokazo. Xuddi shu tarzda, $n = n - 1$ uchun, agar yangi element kiritilsa, yangi n o‘lchamli massiv yaratiladi va eski massivning barcha elementlari yangi massivga ko‘chiriladi va yangi element oxirida qo‘shiladi. n ta kiritish amallaridan keyin umumiy vaqt samaradorligi T(n) (nusxalash amallari soni) $1 + 2 + ... + n \approx O(n^2)$ ga proportsionaldir;

Ikkinchi usul takroriy ikkilantirishdan foydalanish orqali amalga oshirishdir. Ushbu holatda massivni ikki barobarga oshirish texnikasidan foydalanib, bahoni (vaqt samaradorligini) yaxshilash mumkin. Agar massiv to‘la bo‘lsa, ikki barobar kattaroq yangi massiv yaratiladi va elementlar unga ko‘chiriladi. Ushbu yondashuv bilan n ta elementni ko‘chirish n ga proportsional vaqtni oladi (n^2 emas).

Misol uchun, biz dastlab $n = 1$ dan boshlab, $n = 32$ gacha o‘tdik deb faraz qilaylik. Bu shuni anglatadiki, biz 1,2,4,8,16 ga ikkilantiramiz.

Xuddi shu yondashuvni tahlil qilishning yana bir usuli: $n = 1$ holatda, agar element qo‘shmoqchi bo‘lsak, biz massivning joriy hajmini ikki barobarga oshirishimiz va eski massivning barcha elementlarini yangi massivga nusxalashimiz kerak.

$n = 1$ holatda biz 1 nusxa ko‘chirish amalini bajaramiz, $n = 2$ holatda 2 nusxa ko‘chirish amalini va $n = 4$ holatda 4 nusxa ko‘chirish amalini bajaramiz va hokazo. $n = 32$ ga yetganda, nusxa ko‘chirish amallarining umumiy soni $1 + 2 + 4 + 8 + 16 = 31$ bo‘ladi, bu taxminan $2n$ qiymatiga teng (32). Endi muhokamani umumlashtiramiz. n ta element qo‘shish amali

uchun massiv o‘lchami *logn* marta ikkilantirildi. *n* ta qo‘shish amallari uchun umumiy vaqt sarfi *T(n)* quyidagiga teng bo‘ladi:

$$1 + 2 + 4 + 8 \ldots + \frac{n}{4} + \frac{n}{2} + n = n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} \ldots + 4 + 2 + 1$$
$$= n\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \ldots + \frac{4}{n} + \frac{2}{n} + \frac{1}{n}\right)$$
$$= n(2) \approx 2n = O(n)$$

Ya’ni *T(n)* - O(*n*) ga teng.

Push() va *Pop*() usullarni kodi quyidagicha bo‘ladi:

```
public void Push(T item)
{
  // stek o‘lchamini ikkilantiramiz
  if (count == items.Length)
    Resize(items.Length * 2);
    items[count++] = item;
}
public T Pop()
{
  // Stek bo‘sh bo‘lsa istisno holatini yuzaga keltirish
  if (IsEmpty)
    throw new InvalidOperationException("Stek bo‘sh");
  T item = items[--count];
  items[count] = default(T); //havolani qayta o‘rnating
  return item;
}
```

Stek BATni dinamik massiv orqali amalga oshirish misoli ***3.2 ilovada*** keltirilgan.

Samaradorligi

Stekdagi elementlar soni *n* bo‘lsin. Ushbu o‘lchamdagi stek amallari uchun xotira va vaqt bahosi quyidagicha:

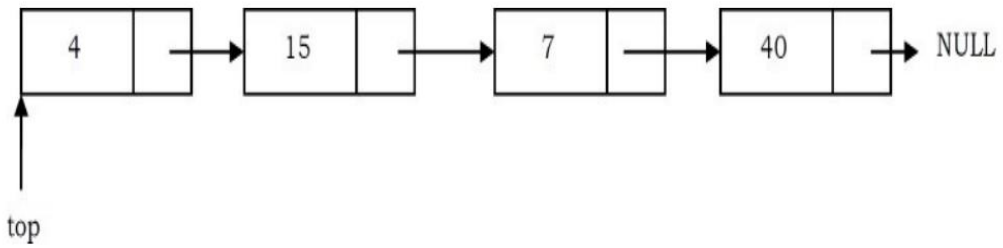
- Xotira sarfi (*n* ta kiritish amali uchun): O(*n*);
- Stekga element qo‘shish *Push*() amali uchun vaqt sarfi: O(1);
- Stekdan elementni olib tashlash *Pop*() amali uchun vaqt sarfi: O(1);
- *Peek*() orqali elementni olish uchun vaqt sarfi: O(1);
- *IsEmpty*() uchun vaqt sarfi: O(1);

- *n* ta elementni qo‘shish uchun umumiy vaqt sarfi : O(*n*);
- Stekdagi elementlarni erkanga chiqarish *Print*() amali uchun vaqt sarfi: O(*n*).

Ajratilgan xotiraning ikki barobarga oshirishi xotiraning to‘lib ketishiga olib kelishi mumkin.

4.2.3. Stekni bir bog‘lamli ro‘yxat asosida amalga oshirish

Bog‘langan ro‘yxatlardan foydalanganda, *Push()* qo‘shish amali elementni ro‘yxatning boshiga kiritish sifatida amalga oshiriladi. *Pop()* amali boshlang‘ich tugunni olib tashlash orqali amalga oshiriladi.



Push() va Pop() usullarni kodi quyidagicha bo‘ladi:

```
public void Push(T item)
{
  // Stekni kattalashtiramiz
  Node<T> node = new Node<T>(item);
  // stek uchini yangi elementga qayta o‘rnatish
  node.Next = head;
  head = node;
  count++;
}
public T Pop()
{
 // Stek bo‘sh bo‘lsa istisno holatini yuzaga keltirish
  if (IsEmpty)
    throw new InvalidOperationException("Stek bo‘sh");
  Node<T> temp = head;
  // stek uchini keyingi elementga qayta o‘rnatish
  head = head.Next;
```

```
    count--;
    return temp.Data;
}
```

Ro‘yxat orqali stekni amalga oshirish dasturi ***3.3 - ilovada*** keltirilgan.

Samaradorligi:

Stekdagi elementlar soni *n* bo‘lsin. Ushbu o‘lchamdagi stek amallari uchun xotira va vaqt bahosi quyidagicha:

- Xotira sarfi (*n* ta kiritish amali uchun): O(*n*);
- Stekga element qo‘shish *Push*() amali uchun vaqt sarfi: O(1);
- Stekdan elementni olib tashlash *Pop*() amali uchun vaqt sarfi: O(1);
- *Peek*() orqali elementni olish uchun vaqt sarfi: O(1);
- *IsEmpty*() uchun vaqt sarfi: O(1);
- n ta elementni qo‘shish uchun umumiy vaqt sarfi: O(*n*);
- Stekdagi elementlarni ekranga chiqarish *Print*() amali uchun vaqt sarfi: O(*n*).

Stek BATni massiv va bog‘langan ro‘yxat asosida amalga oshirishni solishtirish

Massiv asosida amalga oshirish

• Amallar doimiy vaqtni oladi O(1);

• Vaqti o‘tishi bilan qimmat ikkilantirish amali.

Bog‘langan ro‘yxat asosida amalga oshirish

• Chiroyli tarzda o‘sib boradi va qisqaradi;

• Har bir amal doimiy O(1) vaqtni oladi.

• Har bir amal havolalar bilan ishlash uchun qo‘shimcha joy va vaqt talab qiladi.

4 – bob bo‘yicha nazorat savollari

1. Stek nima?
2. Stekga element qo‘shish qanday amalga oshiriladi?
3. Stekdan element olib tashlash qanday amalga oshiriladi?
4. Stekni statik massiv asosida qanday amalga oshiriladi?
5. Stekni dinamik massiv asosida qanday amalga oshiriladi?
6. Stekni bog‘langan ro‘yxatlar asosida qanday amalga oshiriladi?

5. Navbat

Navbat - bu berilganlarni saqlash uchun ishlatiladigan berilganlar tuzilmasidir.

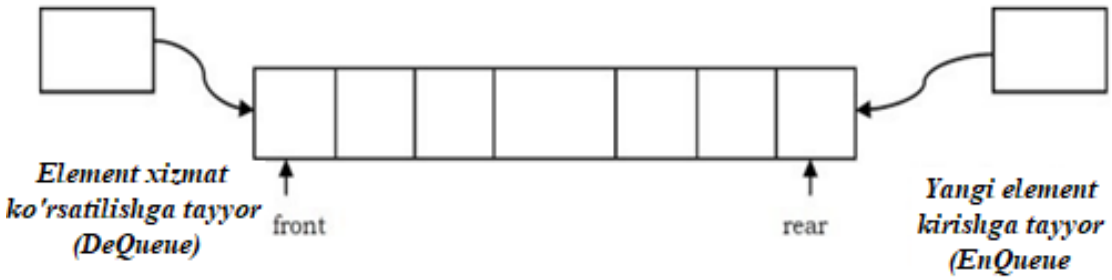
Navbatda ham xuddi stekdagi kabi berilganlarning kelish tartibi muhim ahamiyatga ega. Umuman olganda, navbat - bu ketma-ketlikning boshidan boshlab, ketma-ketlikda xizmat ko‘rsatishni kutayotgan odamlar yoki narsalarning navbatidir.

Ta'rif: **Navbat - bu tartiblangan ro‘yxat bo‘lib, unda kiritish bir uchida (dumidan) va o‘chirish ikkinchi uchida (boshdan) amalga oshiriladi. Birinchi kiritilgan element birinchi bo‘lib o‘chiriladi.**

Navbat “birinchi kirgan birinchi chiqadi” (FIFO) yoki “oxirgi kirgan oxirgi chiqadi” (LILO) tamoyiliga amal qiladi.

Steklardagi singari, navbatda ham bajarilishi mumkin bo‘lgan ikkita jarayonga maxsus nomlar beriladi. Element kiritiladigan jarayonga **EnQueue** va elementni navbatdan olib tashlash jarayoniga **DeQueue** deyiladi.

Elementni bo‘sh navbatdan olib tashlashga urinish “bo‘shab qolish” (underflow) deb ataladi va elementni to‘liq navbatga qo‘shishga urinish “to‘lib ketish” (overflow) deb ataladi. Odatda biz ularni istisno deb hisoblaymiz. Misol tariqasida rasmni ko‘rib chiqamiz:



Navbat tushunchasini supermarketda oziq-ovqat maxsulotlari uchun to‘lovni amalga oshirish jarayonidagi kassani kuzatish orqali tushuntirish mumkin. Biz kassaga kelganimizda, biz navbatning oxirida turamiz va navbat boshida turgan mijoz navbatdagi xizmat ko‘rsatiladigan kishidir. U navbatni tark etadi va xizmat ko‘rsatilgan hisoblanadi. Undan keyin turgan odam navbatning boshida turadi va bir muncha vaqt o‘tgach, navbatni tark

etadi va unga ham xizmat ko‘rsatilgan hisoblanadi. Navbatning oldingi qismidagi har bir kishi navbatdan chiqishda davom etar ekan, biz navbatning old tomoniga qarab harakat qilamiz. Nihoyat, biz navbatning boshiga yetib boramiz, navbatdan chiqamiz va bizga ham xizmat ko‘rsatilgan hisoblanadi. Bu xatti-harakatlar kelish tartibini saqlash zarur bo‘lgan hollarda juda samaralidir.

5.1. Navbat – BAT

Quyidagi keltirilgan amallar navbat BATni yaratish uchun zarur hisoblanadi. Navbatga element qo‘shish va olib tashlash FIFO tamoyiliga amal qilishi kerak.

Navbatning asosiy amallari:

• *EnQueue*(T *data*): elementni navbat oxiriga qo‘shadi;

• T *DeQueue*(): navbat boshidagi elementni olib tashlaydi va uni qiymat sifatida qaytaradi.

Navbatning qo‘shimcha amallari:

• T *First*(): elementni navbat boshidan olib tashlamasdan qaytaradi;

• T *Last*(): elementni navbat oxiridan olib tashlamasdan qaytaradi;

• bool *IsEmpty*: navbat bo‘sh yoki yo‘qligini bildiradi.

Istisnolar

Boshqa BATlarda bo‘lgani kabi, DeQueue-ni bo‘sh navbat bilan bajarish "Bo‘sh navbat" istisnosini keltirib chiqaradi va EnQueue-ni to‘liq navbat bilan bajarish "To‘liq navbat" istisnosini keltirib chiqaradi.

Navbatdan foydalanadigan ilovalarga misol

Quyida navbatlardan foydalanadigan ba'zi ilovalar keltirilgan.

Navbatdan to‘g‘ridan-to‘g‘ri foydalanadigan ilovalar

• Operatsion tizimlar kirib kelish tartibi asosida (masalan, chop etish navbati) vazifalarni (teng ustuvorlik bilan) rejalashtiradi;

• Haqiqiy navbatlarni simulyatsiya qilish, masalan, chiptalar kassasidagi navbatlar yoki birinchi kelganga birinchi xizmat ko‘rsatiladigan boshqa ssenariylar navbatni talab qiladi;

• Multidasturlash;

• Ma'lumotlarni asinxron uzatish (fayl kiritish-chiqarish, kanallar, soketlar);

• Call-markazda mijozlarni kutish vaqti.

Navbatdan bilvosita foydalanadigan ilovalar

• Algoritmlar uchun yordamchi berilganlar tuzilmasi;

• Boshqa berilganlar tuzilmalarining komponenti.

5.2. Navbat BATni amalga oshirilishi

Navbat BATni amalga oshirishning ko‘plab usullari (steklarga o‘xshash) mavjud. Eng ko‘p ishlatiladigan usullar quyida keltirilgan:

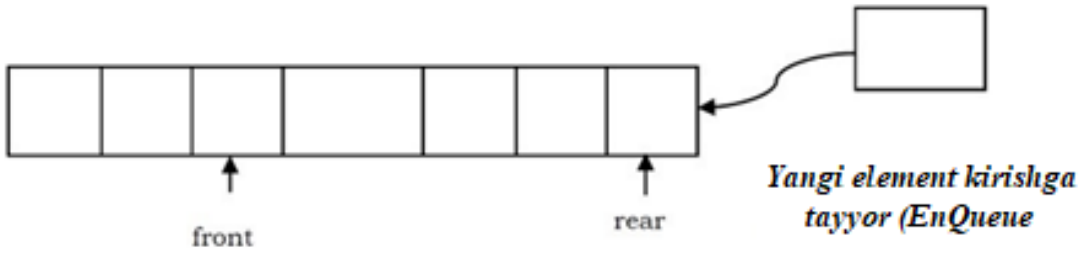
• Statik siklik massiv asosida oddiy amalga oshirish;

• Dinamik siklik massiv asosida amalga oshirish;

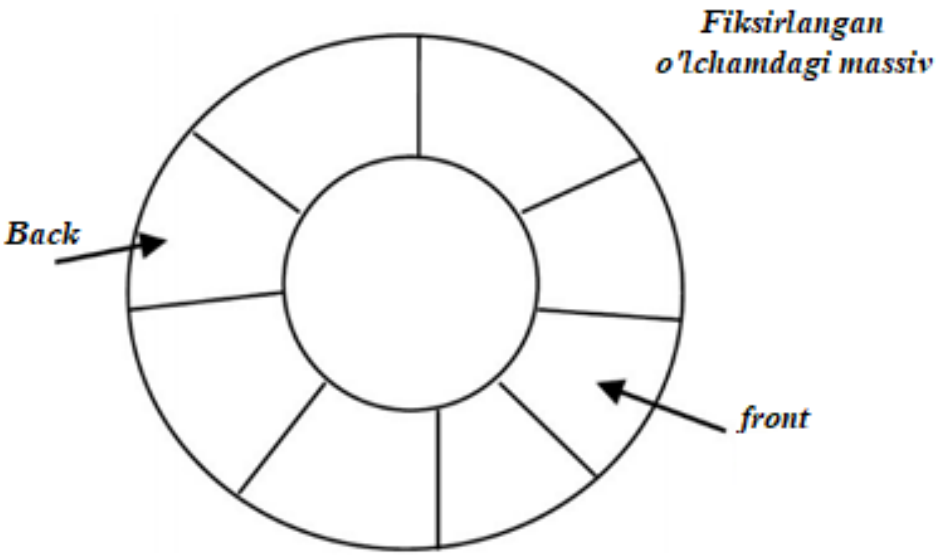
• Bog‘langan ro‘yxat asosida amalga oshirish.

Nima uchun aynan siklik massivlar kerak?

Keling dastlab, stek uchun bajarganimiz kabi, navbatlarni amalga oshirish uchun oddiy massivlardan foydalanishimiz mumkinmi yoki yo‘qmi ko‘rib chiqaylik. Biz bilamizki, navbatda element qo‘shish bir uchidan, ularni olib tashlash boshqa uchidan amalga oshiriladi. Quyidagi rasmda ko‘rsatilgan misolda massivning boshlang‘ich elementlari bo‘sh ekanligi va massiv tez o‘sib borishi aniq ko‘rinib turibdi, bu esa ajratilgan massiv hajmining to‘lib ketishiga olib keladi, garchi navbat uzunligi o‘rtacha barqaror bo‘lib qolsa ham.



Shunday qilib, navbat uchun oddiy massivdan foydalanib amalga oshirish samarasiz. Ushbu muammoni hal qilish uchun biz siklik massivlardan foydalanamiz. Bu shuni anglatadiki, biz massivning oxirgi elementi va birinchi elementini qo‘shni elementlar sifatida ko‘rib chiqamiz. Bunday holda, agar boshida bo‘sh elementlar mavjud bo‘lsa, u holda oxirgi ko‘rsatgich keyingi bo‘sh elementga o‘tadi.



5.2.1. Navbat BATni oddiy siklik massiv asosida amalga oshirish

Navbatning bunday amalga oshirilishida navbat elementlarini saqlash uchun yetarli bo‘lgan ma'lum o‘lchamdagi massiv ishlatiladi. Elementlar massivga aylana shaklida qo‘shiladi va boshlang‘ich (front) va tugatish (back) elementlarini kuzatib boradigan ikkita o‘zgaruvchidan foydalaniladi. Odatda ***front (oldi)*** boshlang‘ich elementni belgilash uchun, ***back (orqa)*** - navbatdagi oxirgi elementni belgilash uchun ishlatiladi. Navbat elementlari qatori to‘lib ketishi mumkin. U holda *EnQueue* amali (navbatga element qo‘shish) ***to‘liq navbat*** turidagi istisnoni keltirib chiqaradi. Xuddi shunday, agar bo‘sh navbatdan elementni olib tashlashga harakat qilinsa, u ***bo‘sh navbat*** turidagi istisnoni keltirib chiqaradi. E'tibor bering, dastlab boshlang‘ich va oxirgi elementlar -1 ga ishora qiladi, ya'ni navbat bo‘sh.

Siklik massiv asosida *EnQueue* va *DeQueue* usullarni amalga oshirish kodi quyidagicha:

```
// Element qo‘shish
public void EnQueue(T item)
{
  // Agar navbat to‘la bo‘lsa, istisno yuzaga keltiramiz
  if (IsFull)
    throw     new     InvalidOperationException("Navbat
to‘la");
  back = (back + 1) % size;
```

```
  items[back] = item;
  if (front == -1)
    front = back;
}
// Elementni olib tashlash
public T DeQueue()
{
// Agar navbat bo‘sh bo‘lsa, istisno yuzaga keltiramiz
  if (IsEmpty)
   throw new InvalidOperationException("Navbat bo‘sh");
  T item = items[front];
  if (front == back)
    front = back = -1;
  else
    front = (front + 1) % size;
  return item;
}
```

Statik siklik massiv orqali navbat BATni amalga oshirish misoli ***4.1 - ilovada*** keltirilgan.

Samaradorligi:

Navbatdagi elementlar soni *n* bo‘lsin. Bunday o‘lchamdagi navbat amallari uchun xotira va vaqt baholari quyidagilardan iborat:

- Xotirani sarfi (*n* ta qo‘shish amali uchun): O(n);
- *EnQueue*() elementni kiritish: O(1);
- *DeQueue*() navbatdan elementni olib tashlash: O(1);
- *Count* navbat hajmini olish: O(1);
- *First*() navbatdan birinchi elementni olish: O(1);
- *Last*() navbatdan oxirgi elementni olish: O(1);
- *IsEmpty* navbatni bo‘shlikka tekshirish uchun: O(1);
- *IsFull* navbatni to‘lalikka tekshirish uchun: O(1).

Cheklovlar:

Dastlab statik massivining maksimal hajmi aniqlanishi kerak va uni o‘zgartirib bo‘lmaydi. To‘liq navbatga yangi element qo‘shishga urinish istisno (xato) keltirib chiqaradi.

5.2.2. Navbat BATni dinamik siklik massiv asosida amalga oshirish

Bu holda, xuddi dinamik massivli stekdagi kabi har safar navbat oxiriga element qo‘shilganda, navbatning to‘lib ketishi tekshiriladi. Agar navbat to‘la bo‘lsa, massivning o‘lchami ikki barobar oshiriladi va keyin yangi element qo‘shiladi.

Dinamik siklik massiv asosida *EnQueue* va *DeQueue* usullarni amalga oshirish kodi quyidagicha:

```
// Element qo‘shish
public void EnQueue(T item)
{
// Agar navbat to‘lib ketsa, istisno yuzaga keltirish
  if(IsFull) Resize();
  back = (back + 1) % size;
  items[back] = item;
  if (front == -1) front = back;
}
// Elementni olish tashlash
public T DeQueue()
{
// Agar navbat bo‘sh bo‘lsa, istisno yuzaga keltirish
  if (IsEmpty)
   throw new InvalidOperationException("Navbat bo‘sh");
  T item = items[front];
  if (front == back)
    front = back = -1;
  else
    front = (front + 1) % size;
  return item;
}
```

Dinamik siklik massiv asosida navbatni BATni amalga oshirish dasturi ***4.2 - ilovada*** keltirilgan.

Samaradorligi:

Aytaylik, *n* - navbat elementlar soni bo‘lsin. Bu o‘lcham uchun xotira va ishlash vaqti baholari quyidagicha:

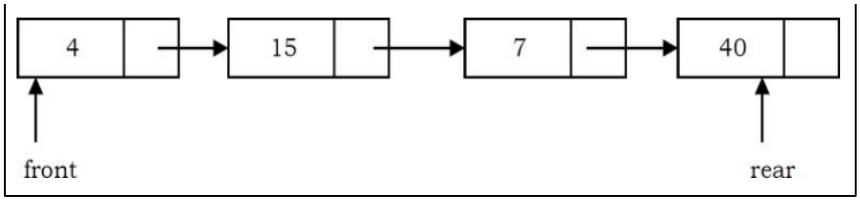
- Xotirani sarfi (*n* ta kiritish amallari uchun): O(*n*);
- *EnQueue*() element kiritish amali uchun: O(1);
- *DeQueue*() elementni o‘chirish amali uchun: O(1);
- *Count* navbat hajmini olish uchun: O(1);
- *First*() navbatdan birinchi elementni olish uchun: O(1);
- *Last*() navbatdan oxirgi elementni olish uchun: O(1);
- *IsEmpty* navbatni bo‘shlikka tekshirish amali uchun: O(1);
- *IsFull* navbatni to‘lalikka tekshirish amali uchun: O(1);
- Navbatga *n* ta elementni kiritish uchun umumiy vaqt bahosi: O(n).

Cheklovlar:

Navbat uchun ajratilgan xotira hajmi ikki baravarga oshirilganda, xotira yetishmaslik muammosi yuzaga kelishi mumkin.

5.2.3. Navbat BATni bir bog‘lamli ro‘yxat asosida amalga oshirish

Bog‘langan ro‘yxatlardan foydalanganda *EnQueue* amali ro‘yxat oxiriga element kiritish sifatida amalga oshiriladi. *DeQueue* amali ro‘yxat boshida joylashgan elementni olib tashlash orqali amalga oshiriladi, shuning uchun ro‘yxat boshiga va ro‘yxat oxiriga ko‘rsatkichlarni saqlash qulay.



Ro‘yxat asosida *EnQueue* va *DeQueue* usullarni amalga oshirish kodi quyidagicha:

```
// Navbatga element qo‘shish
public void EnQueue(T data)
{
  Node<T> node = new Node<T>(data);
  Node<T> temp = tail;
  tail = node;
```

```
  if (count == 0)
    head = tail;
  else
    temp.Next = tail;
  count++;
}
// Navbatdan elementni o‘chirish
public T DeQueue()
{
  if (count == 0)
    throw new InvalidOperationException();
  T output = head.Data;
  head = head.Next;
  count--;
  return output;
}
```

Ro‘yxat bo‘yicha navbatni amalga oshirish dasturi ***4.3 - ilovada*** keltirilgan.

Samaradorligi:

Navbatdagi elementlar soni *n* bo‘lsin. Bunday o‘lchamdagi navbat amallari uchun xotira va vaqt baholari quyidagicha:

- Xotira sarfi (*n* ta qo‘shish amali uchun): O(*n*);
- *Enqueue*() element kiritish amali uchun: O(1);
- *DeQueuePop*() elementni o‘chirish uchun: O(1);
- *Contains*() navbatdagi berilgan qiymatni qidirish uchun: O(*n*).

Amalga oshirishlarni taqqoslash stekni amalga oshirishga o‘xshaydi.

Eng maqbul va oqlangan usul - bog‘langan ro‘yxatni navbat sifatida ishlatishdir.

5.3. Dek

Navbatning yana bir turi **dek** (deque). Deque - bu ikki tomonlama navbat bo‘lib, unda elementlar navbat boshiga yoki oxiriga qo‘shilishi mumkin. O‘chirish ham navbat boshidan, ham oxiridan amalga oshirilishi mumkin.

Ikki tomondan qo‘shish va o‘chirishni amalga oshirish zarur bo‘lganligi sababli, deklar odatda ikki bog‘lamli ro‘yxat asosida tashkil etiladi. Shunday qilib, dekda har bir element ikki bog‘lamli tugun bilan ifodalanadi.

```
public class DNode<T>
{
  public DNode(T data)
  {
    Data = data;
  }
  public T Data { get; set; }
  public DNode<T> Prev { get; set; }
  public DNode<T> Next { get; set; }
}
```

Dek BATda quyidagi usullar amalga oshiriladi:

- *Push_front*() – elementni dek boshidan qo‘shish;
- *Pop_front*() - dek boshidan elementni olib tashlash va qaytarish;
- *Push_back*() - elementni dek oxiridan qo‘shish;
- *Pop_back*() - dek oxiridan elementni olib tashlash va qaytarish.

Qo‘shimcha xususiyatlar:

- *Front*() - elementni dek boshidan qaytaradi, lekin uni olib tashlamaydi;
- *Back*() - elementni dek oxiridan qaytaradi, lekin uni olib tashlamaydi;
- *Count* - dekdagi elementlar sonini qaytaradi;
- *IsEmpty* - dekni bo‘sh yoki yo‘qligini tekshiradi;

Misol tariqasida *Push_Front* va *Pop_Back* usullarni kodi keltirilgan:

```
public void Push_Front(T data)
{
  DNode<T> node = new DNode<T>(data);
```

```
  DNode<T> temp = head;
  node.Next = temp;
  head = node;
  if (count == 0)
    tail = head;
  else
    temp.Prev = node;
  count++;
}
public T Pop_Back()
{
  if (count == 0)
    throw new InvalidOperationException("Dek bo‘sh");
  T output = tail.Data;
  if (count == 1)
  {
    head = tail = null;
  }
  else
  {
    tail = tail.Prev;
    tail.Next = null;
  }
  count--;
  return output;
}
```

Dekni BATi ***5 - ilovada*** keltirilgan.

Dek samaradorligi stek va navbatga o‘xshaydi.

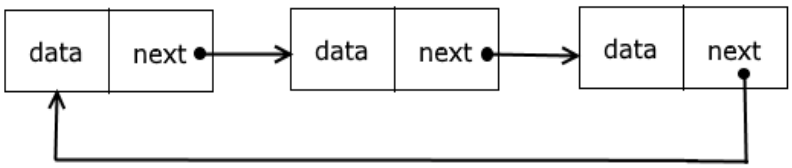
5 – bob bo‘yicha nazorat savollari

1. Navbat nima?
2. Navbatga element qo‘shish qanday amalga oshiriladi?
3. Navbatdan element olib tashlash qanday amalga oshiriladi?
4. Navbatning qanday asosiy amallari mavjud?
5. Siklik statik massiv bilan navbatlar qanday amalga oshiriladi?
6. Siklik dinamik massiv bilan navbatlar qanday amalga oshiriladi?
7. Bog‘langan ro‘yxatlar orqali navbatlar qanday amalga oshiriladi?
8. Dek nima?
9. Dek qanday amalga oshiriladi?

6. Halqasimon bo‘glangan ro‘yxat

6.1. Halqasimon bir bog‘lamli ro‘yxat

Halqasimon (aylana, siklik) ro‘yxatlar bog‘langan ro‘yxatlarning bir turi hisoblanadi. Ular bir yoki ikki bog‘lamli bo‘lishi mumkin. Ularning ajralib turadigan xususiyati shundaki, shartli oxirgi element birinchi elementga havolani saqlaydi, shuning uchun ro‘yxat yopiq yoki halqa shaklida bo‘ladi.



Misol uchun, agar bizning ro‘yxatimiz bitta bosh element *head*dan iborat bo‘lsa, unda biz bunday ro‘yxatni quyidagicha yopishimiz mumkin:

```
head.next = head;
```

Amalga oshirish uchun bir bog‘lamli tugunda ishlatiladigan element sinfini olamiz.

```
using System;
using System.Collections.Generic;
using System.Text;
namespace CircleList
{
  // Halqasimon bog‘langan ro‘yxat
  class CList<T>
  {
    Node<T> head; // bosh/birinchi element
    Node<T> tail; // oxirgi/dum element
    int count;  // ro‘yxatdagi elementlar soni
    public int Count { get { return count; } }
    public bool IsEmpty { get { return count == 0; } }
  }
}
```

`Print()` usuli barcha tugunlarni ekranga aks ettiradi:

```
public void Print()
{
  Node<T> current = head;
  do
  {
    Console.WriteLine(current.Data);
    current = current.Next;
  }
  while (current != head);
}
```

Halqasimon ro‘yxatda element halqani oxiriga qo‘shish.

```
// element qo‘shish
public void Add(T data)
{
  Node<T> node = new Node<T>(data);
  // agar element bo‘sh bo‘lsa
  if (head == null)
  {
    head = node;
    tail = node;
    tail.Next = head;
  }
  else
  {
    node.Next = head;
    tail.Next = node;
    tail = node;
  }
  count++;
}
```

Halqasimon ro‘yxatdan ko‘rsatilgan elementni o‘chirish usuli:

```
public bool Remove(T data)
{
```

```
  Node<T> current = head;
  Node<T> prev = null;
  if (IsEmpty) return false;
  do
  {
    if (current.Data.Equals(data))
    {
    // agar tugun o‘rtada yoki oxirda bo‘lsa
      if (prev != null)
      {
// current tugunini olib tashlaymiz, endi prev currentga emas,
        prev.Next = current.Next;
// Agar tugun oxirgi bo‘lsa tail ni o‘zgartiramiz
        if (current == tail)
          tail = prev;
      }
      else // agar 1-element o‘chirilsa
      {
// agar ro‘yxatda faqat 1 ta element bo‘lsa
        if (count == 1)
        {
          head = tail = null;
        }
        else
        {
          head = current.Next;
          tail.Next = current.Next;
        }
      }
      count--;
      return true;
    }
    prev = current;
    current = current.Next;
```

```
    }
    while (current != head);
    return false;
}
```

Halqasimon ro‘yxatni tozalash

Ro‘yxatni tozalash uchun bosh va oxirgi tugunni tozalash kifoya.

```
public void Clear()
{
    head = null;
    tail = null;
    count = 0;
}
```

Halqasimon ro‘yxat uchun BAT **6.1 - ilovada** to‘liq keltirilgan.

Halqasimon ro‘yxatning samaradorligi xuddi bir bog‘lamli ro‘yxatdagidek bo‘ladi.

6.2. Halqasimon ikki bog‘lamli ro‘yxat

Halqasimon ikki bog‘lamli ro‘yxat — yopiq ro‘yxat bo‘lib, unda elementga ko‘rsatgich aylana bo‘ylab ham oldinga, ham orqaga harakatlanishi mumkin.

Bunday ro‘yxatning har bir tuguni keyingi va oldingi tugunlarga ko‘rsatgichlarni saqlaydigan elementni ifodalaydi.

Halqasimon ro‘yxat rasmiy ravishda yopiq bo‘lishiga qaramay, ya’ni uning boshlanishi va oxiri yo‘q bo‘lishiga qaramay, baribir, shartli ravishda bunday ro‘yxat hali ham birinchi elementga havolani saqlaydi, unga nisbatan yangi elementlar qo‘shiladi.

Shu bilan birga, halqasimon bir bog‘lamli ro‘yxatdan farqli o‘laroq, endi ko‘rsatgichni ro‘yxatning rasmiy oxirgi elementiga saqlash shart emas.

Amalga oshirish uchun bir bog‘lamli tugunda ishlatiladigan element sinfini olamiz.

```
using System;
using System.Collections.Generic;
using System.Text;
namespace CircularDoubleList
{
```

```
  // halqasimon bog‘langan ro‘yxat
  class CDList<T>
  {
    DNode<T> head; // bosh/birinchi element
    int count;  //ro‘yxatdagi elementlar soni
    public int Count { get { return count; } }
    public bool IsEmpty { get { return count == 0; } }
  }
}
```

`Print()` usuli barcha tugunlarni ekranga aks ettiradi:

```
public void Print()
{
  DNode<T> current = head;
  do
  {
    Console.WriteLine(current.Data);
    current = current.Next;
  }
  while (current != head);
}
```

Halqasimon ro‘yxatda elementni halqani oxiriga qo‘shish.

```
// element qo‘shish
public void Add(T data)
{
  DNode<T> node = new DNode<T>(data);
  if (head == null)
  {
    head = node;
    head.Next = node;
    head.Prev = node;
  }
  else
  {
    node.Prev = head.Prev;
```

```
    node.Next = head;
    head.Prev.Next = node;
    head.Prev = node;
  }
  count++;
}
```

Halqasimon ro‘yxatdan ko‘rsatilgan elementni o‘chirish usuli:

```
// elementni o‘chirish
public bool Remove(T data)
{
  DNode<T> current = head;
  DNode<T> removedItem = null;
  if (count == 0) return false;
// o‘chiriladigan tugunni qidirish
  do
  {
    if (current.Data.Equals(data))
    {
      removedItem = current;
      break;
    }
    current = current.Next;
  }
  while (current != head);
  if (removedItem != null)
  {
//agar ro‘yxatning yagona elementi o‘chirilsa
    if (count == 1)
      head = null;
    else
    {
// agar birinchi elementi o‘chirilsa
      if (removedItem == head)
```

```
      {
        head = head.Next;
      }
      removedItem.Prev.Next = removedItem.Next;
      removedItem.Next.Prev = removedItem.Prev;
    }
    count--;
    return true;
  }
  return false;
}
```

Halqasimon ro‘yxatni tozalash

Ro‘yxatni tozalash uchun bosh va oxirgi tugunni tozalash kifoya.

```
public void Clear()
{
  head = null;
  count = 0;
}
```

Halqasimon ikki bog‘lamli ro‘yxat uchun BAT ***6.2 - ilovada*** to‘liq keltirilgan.

Halqasimon ikki bog‘lamlik ro‘yxatning samaradorligi xuddi ikki bog‘lamli ro‘yxatdagidek bo‘ladi.

6 – bob bo‘yicha nazorat savollari

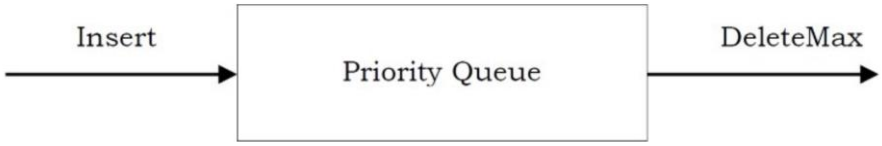
1. Halqasimon bir bog‘lamli ro‘yxat nima?
2. Halqasimon bir bog‘lamli ro‘yxat qanday amalga oshiriladi?
3. Halqasimon ikki bog‘lamli ro‘yxat nima?
4. Halqasimon ikki bog‘lamli ro‘yxat qanday amalga oshiriladi?
5. Halqasimon bir bog‘lamli va ikki bog‘lamli ro‘yxat farqi nimada?

7. Ustuvor navbat

Ba'zi hollarda elementlar to‘plamidan minimal/maksimal elementni topishimiz kerak bo‘lishi mumkin. Buni ustuvor navbat BAT bilan amalga oshirishimiz mumkin.

Ustuvor navbat BAT - *Insert* (navbatga element qo‘shish) va *DeleteMin* (minimal elementni olib tashlaydi va uni qiymat sifatida qaytaradi) yoki *DeleteMax* (maksimal elementni olib tashlaydi va uni qiymat sifatida qaytaradi) amallarini qo‘llab-quvvatlaydigan berilganlar tuzilmasi hisoblanadi.

Bu amallar navbat BATdagi *EnQueue* va *DeQueue* amallariga ekvivalent hisbolanadi. Farqi shundaki, ustuvor navbatlarda elementlarning navbatga kirish tartibi ularni qayta ishlash tartibiga mos kelmasligi mumkin. Ustuvor navbatlarni qo‘llanilishiga misol sifatida, navbat bo‘yicha xizmat ko‘rsatishdan ko‘ra ustunligi (muhimligi) bo‘yicha masalalarni rejalashtirishni keltirish mumkin.



Agar eng kichik kalitga ega bo‘lgan element eng yuqori ustuvorlikka ega bo‘lsa (ya'ni eng kichik element har doim o‘chiriladi) ustuvorlik navbati ortib boruvchi ustuvor navbati deb ataladi.

Xuddi shunday, agar eng katta kalitga ega bo‘lgan element eng yuqori ustuvorlikka ega bo‘lsa (maksimal element har doim olib tashlanadi) ustuvorlik navbati kamayib boruvchi ustuvor navbati deb ataladi.

Ushbu ikki tur simmetrik bo‘lgani uchun biz ulardan biriga e'tibor qaratamiz. Ortib boruvchi ustuvor navbatni qarab chiqamiz.

7.1. Ustuvor navbat - BAT

Ustuvor navbatlar ustidagi asosiy amallar:

Ustuvor navbat - elementlar konteyneri bo‘lib, ularning har biri o‘zi bilan bog‘langan kalitga ega bo‘lib, konteynerda quyidagi amallar bajariladi:

• *Insert*(key, *data*): *data*ni *key* yordamida ustuvor navbatga kiritadi. Elementlar kalit asosida tartibga solinadi;

• *DeleteMin/DeleteMax*: Eng kichik/katta kalitli elementni o‘chiradi va qiymat sifatida qaytaradi;

• *GetMinimum*/*GetMaximum:* Eng kichik/eng katta kalitga ega elementni qaytaradi, lekin uni ustuvor navbatdan o‘chirib tashlamaydi.

Ustuvor navbatdagi yordamchi amallar:

• *Count*: ustuvor navbatdagi elementlar sonini qaytaradi;

• *IsEmpty*: elementlar ustuvor navbatda mavjud yoki yo‘qligini bildiradi.

Ustuvor navbat ilovalari

Ustuvor navbatlar asosida qurilgan masalalar amaliyotda ko‘p uchraydi, ulardan ba’zilari quyida keltirilgan:

• Berilganlarni siqish: Haffman kodlash algoritmi;

• Eng qisqa yo‘l algoritmlari: Deykstra algoritmi;

• Minimal qoldiq daraxti algoritmi: Prim algoritmi;

•Hodisaga asoslangan simulyatsiya: Navbatdagi mijozlar;

• Tanlov muammosi: k-chi eng kichik elementni topish.

7.2. Ustuvor navbatni amalga oshirish

Ustuvor navbatlarni amalga oshirishdan oldin, keling mumkin bo‘lgan variantlarni muhokama qilamiz.

Tartibsiz massiv ko‘rinishida amalga oshirish

Elementlar massivga tartibidan qat'iy nazar kiritiladi. *Delete* (*DeleteMax*) kalitni qidirib, keyin o‘chirish orqali amalga oshiriladi.

Kiritish murakkabligi: O(1);

Olib tashlash murakkabligi: O(n).

Tartibsiz ro‘yxat ko‘rinishida amalga oshirish

Bu massivni amalga oshirishga juda o‘xshaydi, lekin massivlar o‘rniga bog‘langan ro‘yxatlardan foydalaniladi.

Kiritish murakkabligi: O(1);

Olib tashlash murakkabligi: O(n).

Tartiblangan massiv ko‘rinishida amalga oshirish

Elementlar massivga kalit maydoni asosida tartiblangan holda kiritiladi. Olib tashlash faqat bir uchidan amalga oshiriladi.

Kiritish murakkabligi: O(n);

Olib tashlash murakkabligi: O(1).

Tartiblangan ro‘yxat ko‘rinishida amalga oshirish

Elementlar ro‘yxatga kalit maydonlari asosida tartiblangan tartibida kiritiladi. O‘chirish faqat bir uchidan amalga oshiriladi, shuning uchun ustuvor navbatning holati saqlanib qoladi. Ro‘yxat bilan bog‘liq barcha boshqa funksiyalar xuddi bog‘langan ro‘yxatlardagi kabi o‘zgartirilmagan holda amalga oshiriladi.

- Kiritish murakkabligi: O(n);
- Olib tashlash murakkabligi: O(1).

Ikkilik qidirish daraxtlari orqali amalga oshirish

Agar qo‘shish tasodifiy bo‘lsa, qo‘shish va o‘chirish o‘rtacha O(log*n*) murakkablikka ega bo‘ladi.

Muvozanatlangan ikkilik qidirish daraxtlari orqali amalga oshirish

Qo‘shish ham, o‘chirish ham eng yomon holatda O(log*n*) murakkablikka ega bo‘ladi.

Uyum (piramida) orqali amalga oshirish

Ikkilik uyum qidirish, kiritish va o‘chirish uchun O(log*n*) murakkabligini va maksimal yoki minimal elementni topish uchun O(1) murakkabligiga ega bo‘ladi.

Ustuvor navbatni amalga oshirishni uyum (heap) mavzusida ko‘rib chiqamiz.

Quyidagi jadvalda qidirish algoritmlarini amalga oshirishlarni solishtirish keltirilgan:

Jadval №3. Turli BAT uchun murakkablik baholari.

№	Qo‘llanilishi	Qo‘shish	O‘chirish	Min topish
1	Tartiblanmagan massiv	*1*	*n*	*n*
2	Tartiblanmagan ro‘yxat	*1*	*n*	*n*
3	Tartiblangan massiv	*n*	*1*	*1*
4	Tartiblangan ro‘yxat	*n*	*1*	*1*
5	Binar qidirish daraxti	*logn*	*logn*	*logn*
6	Muvozanatlashtirilgan binar qidirish daraxti	*logn*	*logn*	*logn*
7	Binar uyum	*logn*	*logn*	*1*

Tartiblangan ro‘yxat ko‘rinishida amalga oshirish

Ustuvor ro‘yxatlarni tartiblangan ro‘yxatlar asosida amalga oshirish dasturi ***6.3 - ilovada*** keltirilgan.

7 – bob bo‘yicha nazorat savollari

1. Ustuvor navbat nima?
2. Ustuvor navbat qanday amalga oshiriladi?
3. Ustuvor navbatlarga element qo‘shish qanday amalga oshiriladi?
4. Ustuvor navbatlardan elementni olib tashlash qanday amalga oshiriladi?
5. Ustuvor navbatlarni tartibsiz massiv ko‘rinishida amalga oshirish.
6. Ustuvor navbatlarni tartibsiz ro‘yxat ko‘rinishida amalga oshirish.
7. Ustuvor navbatlarni tartiblangan massiv ko‘rinishida amalga oshirish.
8. Ustuvor navbatlarni tartiblangan ro‘yxat ko‘rinishida amalga oshirish.
9. Ustuvor navbatlarni ikkilik qidirish daraxtlari orqali amalga oshirish.
10. Ustuvor navbatlarni uyum (piramida) orqali amalga oshirish.

8. Rekursiv usul

Misol tariqasida 1 dan n gacha natural sonlar yig‘indisini ko‘rib chiqaylik. C# dasturlash tilida sikl operatori yordamida hal qilish algoritmi quyidagicha yozilishi mumkin:

```
int Sum(int n)
{
  int s = 0;
  for (int i = 1; i <= n; ++i) s += i;
  return s;
}
```

Yoki funksiyani quyidagicha qayta yozish mumkin:

```
int Sum(int n)
{
  int s = 0;
  while (n > 0) { s += n; --n; }
  return s;
}
```

Oxirgi algoritmda yig‘indini hisoblaymiz, masalan, n=5 uchun u 5+4+3+2+1 ga teng bo‘ladi, uni 5+(4+(3+(2+(1)))) shaklida yozish mumkin, ya'ni biz 5 soniga kamaygan ketma-ketlikning yig‘indisini qo‘shamiz. Yuqoridagilarni hisobga olib algoritmimizni yozamiz:

```
int Sum(int n)
{
  return n + Sum(n - 1);
}
```

Muammoning rekursiv yechimiga ega bo‘ldik, ya’ni Sum funksiyasi o‘zini argumentning kamaygan qiymati bilan chaqiradi.

Rekursiv yechim estafetali uzatishga o‘xshaydi. Kimdir sizga 5 soni uchun yig‘indini hisoblashni aytadi. Siz bu yig‘indi 5 soni va undan oldingi 4 ta sonlar yigindisiga teng ekanligini bilasiz, shuning uchun siz Boburga qo‘ng‘iroq qilasiz va unga 4 soni uchun yig‘indisini hisoblashni aytasiz. Uning javobini olgach, siz unga 5 qo‘shasiz va yakuniy natijaga erishasiz.

Bobur 4 sonining yig‘indisi 4 soni va undan oldingi 3 ta sonlar yig‘indisiga teng ekanligini biladi. U Rustamga qo‘ng‘iroq qiladi va unga 3 soni uchun yig‘indini hisoblashni aytadi.

Jarayon davom etadi va har bir ishtirokchi estafetani keyingisiga uzatadi.

Uzatish qachon tugaydi? Ma’lum bir vaqtga kelib kimdir yordam so‘ramasdan javobni aniqlay oladi. Agar bu sodir bo‘lmaganida, so‘rovlar bilan bir-biriga murojaat qiladigan cheksiz odamlar zanjiri - hech qachon tugamaydigan matematik "piramida" paydo bo‘lar edi.

Sum() funksiyasi uchun bu usul dastur ishdan chiqmaguncha o‘zini cheksiz chaqirishini bildiradi.

Cheksiz uzatishni oldini olish uchun n = 1 uchun summani hisoblash buyurilgan kishi javob 1 ekanligini bilishi kerak va u hech kimdan bu javobni olishni so‘ramaydi.

Minimal songacha yetib borildi, shuning uchun estafetani uzatish to‘xtatildi. Vaziyat *Sum*() funksiyasiga qo‘shimcha shart qo‘shish orqali hal qilinadi:

```
int Sum(int n)
{
  if (n == 1) return 1;
  return n + Sum(n - 1);
}
```

Rekursiv usulning boshqaruvini keyingi rekursiv chaqiruvsiz qaytarishiga olib keladigan shart tayanch cheklash deb ataladi.

Har bir rekursiv usul cheksiz rekursiyani va keyinchalik dasturning ishdan chiqishini oldini olish uchun tayanch cheklovga ega bo‘lishi kerak.

Rekursiv funksiyadan foydalangan holda to‘liq dastur quyidagicha bo‘ladi:

```
using System;
namespace Recursiya
{
  class Program
  {
    static void Main(string[] args)
    {
```

```
        int Sum(int n)
        {
          if (n == 1) return 1;
          return n + Sum(n - 1);
        }
        int n;
        Console.WriteLine("Kiriting n = :");
        n = int.Parse(Console.ReadLine());
        Console.WriteLine($"Yig‘indi: {Sum(n)}");
        Console.ReadKey();
     }
  }
}
```

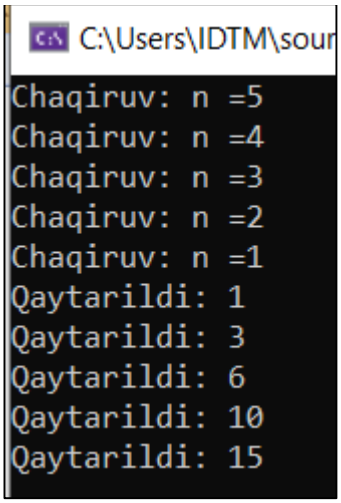
Bu qanday ishlaydi?

Keling, *Sum*() funksiyasini biroz o‘zgartiraylik, shunda u bajarilishi paytida nima sodir bo‘lishi haqida ma'lumot beradi.

Qo‘shimcha chop qilish buyruqlari argumentlarni va qaytariluvchi qiymatlarni kuzatishga yordam beradi:

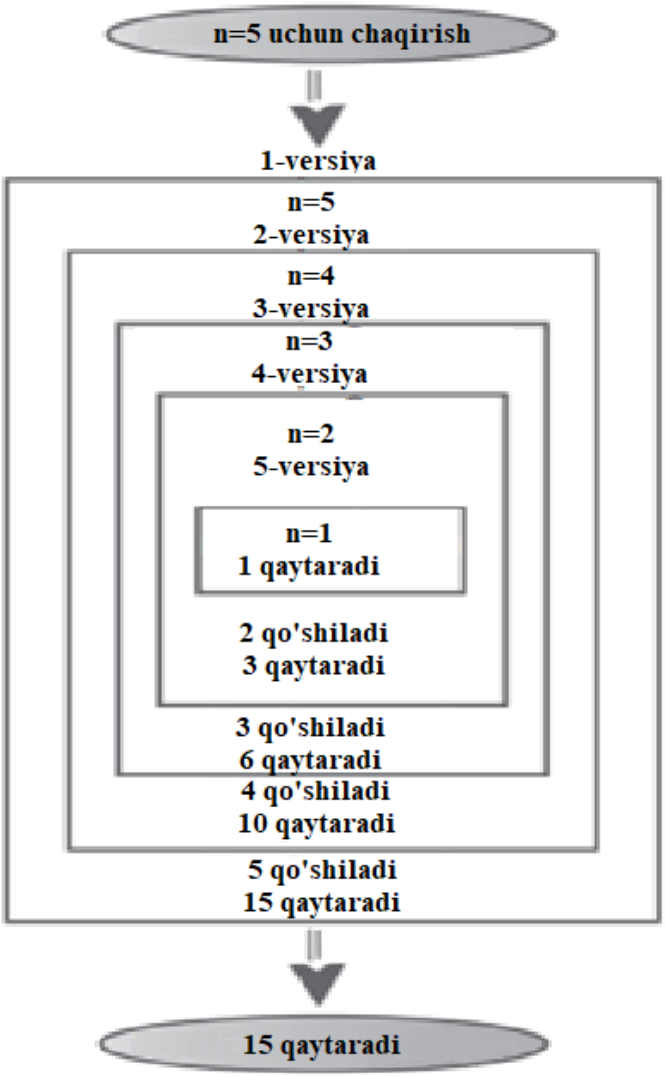
```
int Sum(int n)
{
  Console.WriteLine($"Chaqiruv: n ={n}");
  if (n == 1)
  {
    Console.WriteLine("Qaytarildi: 1");
    return 1;
  }
  int s = n + Sum(n - 1);
  Console.WriteLine($"Qaytarildi: {s}");
  return s;
}
```

Sum(n) ning yangi versiyasi foydalanuvchi 5 sonini kiritganda qanday ko‘rinishda bo‘ladi:



Sum() funksiyasi har safar o‘zini chaqirganda, uning dastlab 5 bo‘lgan argumenti 1 ga kamayadi. Rekursiv chaqiruvlar argument 1 ga teng bo‘lguncha davom etadi va shu nuqtada funksiya qaytadi. Bu kaskadli qaytarishlarning butun bir seriyasiga olib keladi.

Qaytish qiymatlari javob *Main*() usuliga qaytarilgunga qadar sonlar yig‘indisini takrorlaydi. Rasmda *Sum*() ga har bir rekursiv chaqiruv oldingi chaqiruvning "ichidan" qanday amalga oshirilganligi ko‘rsatilgan. Ichki versiya 1 ni qaytarishdan oldin, dasturda bir vaqtning o‘zida beshta *Sum*() chaqiruvlari mavjud. Tashqi chaqiruv qiymati 5 ga teng argumentni, ichki chaqiruv esa 1 ga teng argumentni oladi.



Sum() funksiyasi soddaligiga qaramay, rekursiv usullarning barcha xususiyatlariga ega:

- U o‘zini o‘zi chaqiradi;
- Rekursiv chaqiruv soddalashtirilgan masalani hal qilish uchun mo‘ljallangan;
- Muammoning yetarlicha sodda versiyasi mavjud bo‘lib, bu usul uni hal qilishi va rekursiyasiz qaytishi mumkin.

Rekursiv funksiyaning har bir keyingi chaqiruvida argument qiymati kamayadi (yoki bir nechta argumentlar bilan belgilangan diapazon torayadi) - bu muammoni bosqichma-bosqich "soddalashtirish"ni namoyon qiladi. Argument yoki diapazon ma'lum bir minimal o‘lchamga yetganda, tayanch cheklov ishga tushadi va usul rekursiyasiz qaytadi.

8.1. Rekursiya samaradorligi

Usulni chaqirish biroz qo‘shimcha xarajatlarni o‘z ichiga oladi. Boshqaruv joriy chaqiruv nuqtasidan funksiyaning boshiga o‘tkazilishi kerak.

Bundan tashqari, usul argumentlari va qaytish manzili ichki stekga joylanadi, shunda funksiya argument qiymatlariga murojaat qila olishi va boshqaruvni qaysi manzil bo‘yicha qaytarishni bilishi mumkin.

Sum() funksiyasi bilan bog‘liq vaziyatda, ehtimol, shunday xarajatlar tufayli masalaning sikl yordamida yechimi rekursiv yechimga qaraganda tezroq bo‘ladi. Kechikish ahamiyatsiz bo‘lishi mumkin, ammo ko‘p sonli chaqiruvlar bilan rekursiyadan xalos bo‘lish maqsadga muvofiqdir.

Samaradorlikni pasaytirishning yana bir omili - bu ichki stekdagi barcha oraliq argumentlar va qaytish qiymatlarini saqlash uchun xotira sarfi hisoblanadi. Katta hajmdagi berilganlar muammolarni keltirib chiqarishi va stekning to‘lib ketishiga olib kelishi mumkin.

Rekursiya odatda tabiatan samaraliroq bo‘lgani uchun emas balki muammoni kontseptual soddalashtirish uchun ishlatiladi.

Rekursiya - bu matematik induksiyaning dasturiy analogi - aniqlanayotgan obyekt kontekstida biror narsani aniqlash usuli. Matematik darajada arifmetik progressiya yig‘indisini aniqlash imkonini beradi:

1) n = 1 uchun sum(n) = 1;

2) n > 1 uchun sum(n) = n + sum(n–1).

Yig‘indiga o‘xshab, n! faktorial uchun ham rekursiv funksiyani yozish mumkin:

```
int Factorial(int n)
{
  if (n == 0) return 1;
  return n * Factorial(n - 1);
}
```

Faktorial hisoblash rekursiyaning klassik namunasidir. Rekursiv usullarni boshqa hisoblash masalalariga ham qo‘llash mumkin. Masalan, ikki sonning eng katta umumiy karralisini hisoblash (kasrlarni qisqartirish uchun ishlatiladi), sonni darajaga ko‘tarish va hokazo.

Biroq, avvalgi holatda bo‘lgani kabi, bu yechimlar ham rekursiyani namoyish qilish uchun juda mos keladi, lekin amalda sikllarga asoslangan yechimlar samaraliroq bo‘ladi.

8.2. Anagrammalar

Rekursiyadan foydalanib masalaning yechimini topish uchun eng ajoyib bo‘lgan boshqa sohani ko‘rib chiqamiz. O‘zgartirish - obyektlarni ma'lum bir tartibda joylashtirish masalasini qaraymiz. Aytaylik, biz ma'lum bir so‘zning anagrammalarining to‘liq ro‘yxatini, ya'ni harflarning barcha mumkin bo‘lgan almashtirishlarini (ulardan haqiqiy so‘zlar chiqish yoki chiqmasligidan qat'iy nazar) tuzmoqchimiz.

Masalan, ***cat*** so‘zi uchun dastur quyidagi anagrammalar ro‘yxatini chiqarishi kerak:

cat
cta
atc
act
tca
tac

Anagrammada variantlar soni harflar sonining faktorialiga teng. Uch harfli so‘zda 6 ta, to‘rt harfli so‘zda 24 ta, besh harfli so‘zda 120 ta anagramma va hokazo. So‘z uchun anagrammalar ro‘yxatini tuzish dasturini qanday yozish mumkin? *n* harfli so‘zning mumkin bo‘lgan usullaridan biri:

1. O‘ngdagi *n*–1 harf uchun anagrammalar tuzing;
2. Barcha *n* harfni siklik siljiting;
3. Ushbu amallarni *n* marta takrorlang.

Siklik siljishda eng chapda joylashgan harfdan tashqari barcha harflar bir o‘rin chapga siljiydi. Eng chapdagi harf so‘zning oxiriga o‘tkaziladi.

Agar so‘z *n* marta siklik siljitilsa, so‘zdagi harflarning har biri birinchi o‘rinda bo‘ladi. Tanlangan harf birinchi o‘rinni egallab turgan bir paytda, qolgan barcha harflar uchun anagrammalar ro‘yxati tuziladi (ya'ni harflar barcha mumkin bo‘lgan kombinatsiyalarda qayta tartibga solinadi). “cat” so‘zi bor-yo‘g‘i uchta harfdan iborat, shuning uchun oxirgi ikki harfni siklik siljitish deganda ularni o‘rinlarini almashtirish tushuniladi.

O‘ngdagi *n* - 1 ta harflar uchun anagrammalar ro‘yxati qanday tuziladi? Rekursiv chaqiruv orqali. *Anagram*() rekursiv protsedurasi anagrammalar tuziladigan so‘z uzunligidagi bitta parametrini oladi. Bu so‘z dastlabki so‘zning o‘ng tarafidagi *n* ta harfi deb faraz qilinadi. Har safar *Anagram*() protsedurasi o‘zini rekursiv chaqirganda, u bitta kam sondagi harf uchun shunday qiladi.

Tayanch cheklov anagrammalar tuziladigan so‘z faqat bitta harfdan iborat bo‘lganda yuzaga keladi. Buning uchun yangi almashtirishlarni yaratishdan ma’no yo‘q, shuning uchun protsedura darhol boshqaruvni qaytaradi. Aks holda, u berilgan so‘zning birinchi harfidan tashqari barcha harflarning anagrammalarini tuzadi va butun so‘zni siklik siljitadi. Bu ikki harakat *n* marta bajariladi, bu yerda *n* - so‘zning uzunligi. *Anagram*() rekursiv protsedura kodi:

```
void Anagram(int newSize)
{
 if (newSize == 1) { return; }
 for (int j = 0; j < newSize; j++)
 {
  Anagram(newSize - 1);
  if (newSize == 2)
   Console.WriteLine($"Variant:N{++count} so‘z:{str}");
  rotate(newSize);
 }
}
```

Anagram() protsedurasi har safar o‘zini chaqirganda, so‘zning o‘lchami bir harfga qisqaradi va boshlang‘ich o‘rni bir katakcha o‘ngga siljiydi.

Dasturning to‘liq matni ***7.1 - ilovada*** keltirilgan.

Rekursiv usul yordamida yechiladigan keyingi masala butun sonlarning tartiblangan massividan kalitni qidirish masalasidir.

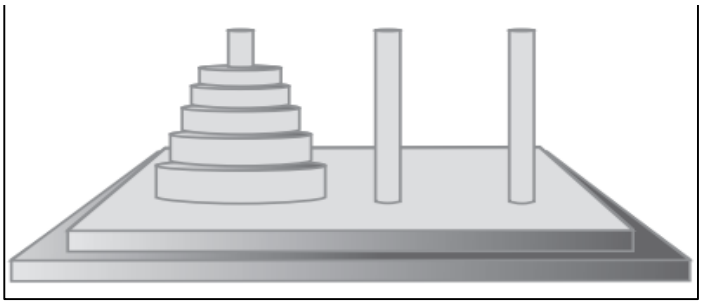
Qidirish uchun ikkilik qidirish algoritmi qo‘llaniladi, ya'ni massivni teng ikkiga bo‘lish va kalit qiymati o‘rta elementdan katta yoki kichik bo‘lishiga qarab, o‘ng yoki chap yarmini teng ikkiga bo‘lish davom etadi, va hokazo.

Qidirish funksiyasi:

```
int recFind(int [] a,int Key, int lowerBound, int
upperBound)
{
  int curIn;
  curIn = (lowerBound + upperBound) / 2;
  if (a[curIn] == Key) return curIn;
  if (lowerBound > upperBound) return -1;
  if (a[curIn] < Key) // Yuqori yarmida
    return recFind(a,Key, curIn + 1, upperBound);
  else
    return recFind(a,Key, lowerBound, curIn - 1);
}
```

Dasturning to‘liq kodi ***7.2 - ilovada*** keltirilgan.

8.3. Xanoy minorasi



Xanoy minorasi - bu matematik boshqotirma. U uchta minora (yoki qoziqlar yoki novda) va har qanday shtanga ustida siljishi mumkin bo‘lgan turli o‘lchamdagi bir qator disklardan iborat.

Boshqotirma bitta minora ustidagi disklar hajmining ortib borish tartibida, yani eng kichigi tepada va shu tariqa konus shaklini hosil qilgan holatdan boshlanadi.

Jumboqning maqsadi quyidagi qoidalarga rioya qilgan holda butun disklar to‘plamini boshqa minoraga o‘tkazishdan iborat:

• Bir vaqtni o‘zida faqat bitta diskni ko‘chirish mumkin;

• Har bir harakat qaysidir minoradan eng tepadagi diskni olib, uni boshqa minora ustiga qo‘yishdan iborat. Qo‘yish paytida mazkur minorada allaqachon boshqa disklar bo‘lishi mumkin, u holda hamma disklar ustiga qo‘yishdan iborat bo‘ladi;

• Diskni kichikroq disk ustiga qo‘yib bo‘lmaydi.

Algoritmi:

• Yuqori *n* - 1 ta disklarni dastlabki minoradan yordamchi minoraga o‘tkazing;

• Dastlabki minoradagi *n*-diskni maqsad minoraga o‘tkazing;

• *n* - 1 ta diskni yordamchi minoradan maqsad minoraga o‘tkazing.

• Yuqoridagi *n* - 1 disklarni dastlabki minoradan yordamchisiga qayta ko‘chirishni o‘ylab ko‘ring. Ko‘chirishga xuddi yuqoridagi kabi hal qilinishi mumkin bo‘lgan yangi masala sifatida qaraladi. Hanoy minorasi masalasini uchta disk uchun yechganimizdan so‘ng, biz bu muammoni yuqorida ko‘rsatilgan algoritm asosida istalgan sondagi disklar uchun hal qilishimiz mumkin.

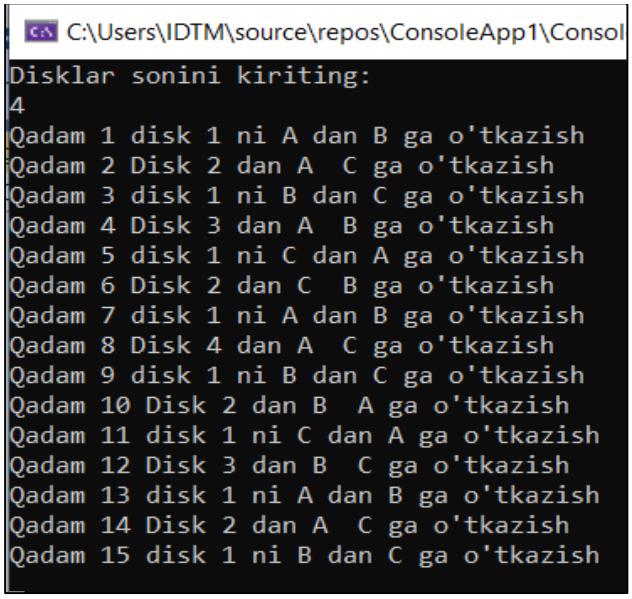
Funksiyasi quyida keltirilgan:

```
void Hanoy(int n, char from, char inter, char to)
{
  if (n == 1)
    Console.WriteLine($"Qadam {++count} disk 1 ni {from} dan {to} ga o‘tkazish");
  else
  {
    Hanoy(n - 1, from, to, inter); // from-->inter
    Console.WriteLine($"Qadam {++count} Disk {n} dan {from} {to} ga o‘tkazish");
    Hanoy(n - 1, inter, from, to); // inter-->to
  }
}
```

Hanoy() funksiyasi argumentlari ko‘chiriladigan disklar sonini, shuningdek dasklabki(from), yordamchi(inter) va maqsad (to) minora raqamlarini o‘z ichiga oladi. Har bir rekursiv chaqiruvda disklar soni bittaga kamayadi. Minoralar ham o‘zgaradi.

To‘liq dastur ***7.3 - ilovada*** berilgan:

Dasturni ishga tushirish natijasi:



Ushbu natijalarni *Hanoy*() manba kodi bilan birga tahlil qiling, qo‘shimcha chop qilishlar qo‘shing va usul qanday ishlashini aniq tushunib olasiz.

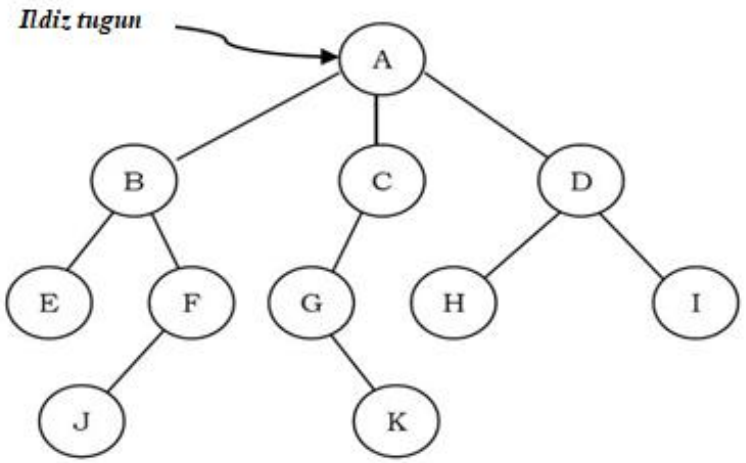
8 – bob bo‘yicha nazorat savollari

1. Rekursiv usul qanday ishlaydi?
2. Rekursiya samaradorligi.
3. Anagrammalarni rekursiya bilan xosil qilish.
4. Xanoy minorasi masalasini rekursiya yordamida yechish.

9. Daraxt

Daraxt - bu bog‘langan ro‘yxatga o‘xshash berilganlar tuzilmasi bo‘lib, unda har bir tugun boshqa bir tugunni emas, balki bir nechta tugunlarni ko‘rsatadi. Daraxt - chiziqli bo‘lmagan berilganlar tuzilmasiga namunadir. Daraxt ko‘rinishidagi tuzilma - bu tuzilmaning ierarxik xususiyatini grafik shaklda ifodalash usulidir.

Daraxtlarda elementlarning tartibi muhim emas. Agar bizga tartiblangan ma'lumot kerak bo‘lsa, biz bog‘langan ro‘yxatlar, steklar, navbatlar va boshqalar kabi chiziqli tuzilmalardan foydalanamiz. Daraxtga misol:



9.1. Daraxtning asosiy xususiyatlari

Daraxt ildiz tugunga ega, hamda, daraxtning ildizi - ota-onasiz tugundir. Undan daraxt boshlanadi. Daraxtda faqat bitta ildiz tugun bo‘lishi mumkin (yuqoridagi misolda A tugun).

• Ota-onani bolalar bilan bog‘lash uchun qirralar (novdalar) ishlatiladi;

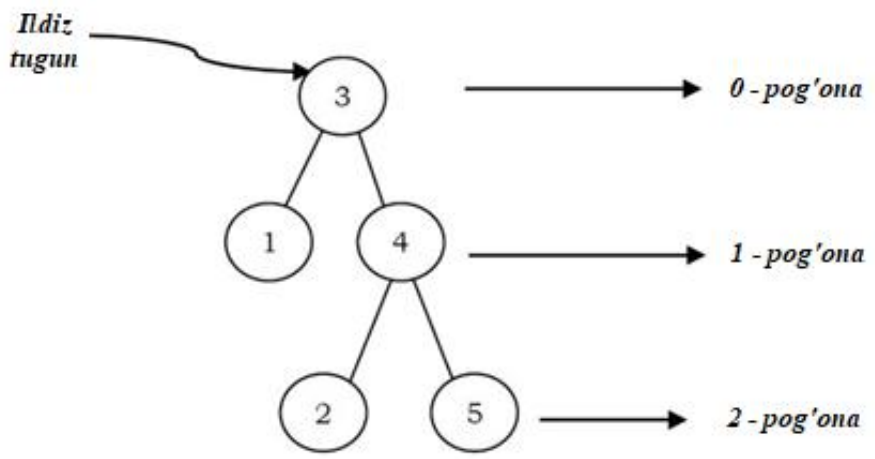
• Avlodlari (bolalari) bo‘lmagan tugunlar barglar deb ataladi (E, J, K, H va I).

• Bir ota-onaning farzandlari aka-uka yoki opa-singil deb ataladi, masalan, A ota-onasining (B, C, D) bolalari bor. O‘z navbatida, B tugunida bolalar (E, F) bor, ular aka-uka hisoblanadi;

• Agar ildizdan *q* gacha bo‘lgan yo‘l bo‘lsa va yo‘lda *p* paydo bo‘lsa, *p* tugun *q* tugunining ajdodi hisoblanadi. *q* tuguniga *p* ning avlodi deyiladi. Masalan, A, C va G tugunlar K ning ajdodlari;

• Berilgan chuqurlikdagi barcha tugunlar to‘plami daraxt pog‘onasi deb ataladi (B, C va D bitta pog‘ona tugunlari hisoblanadi). Ildiz tugun nol pog‘onada joylashgan bo‘ladi;

Tugunning chuqurligi - ildiz tugundan ushbu tugungacha bo‘lgan yo‘lning uzunligiga aytiladi (G tugun chuqurligi 2 ga teng ya’ni, A - C - G);

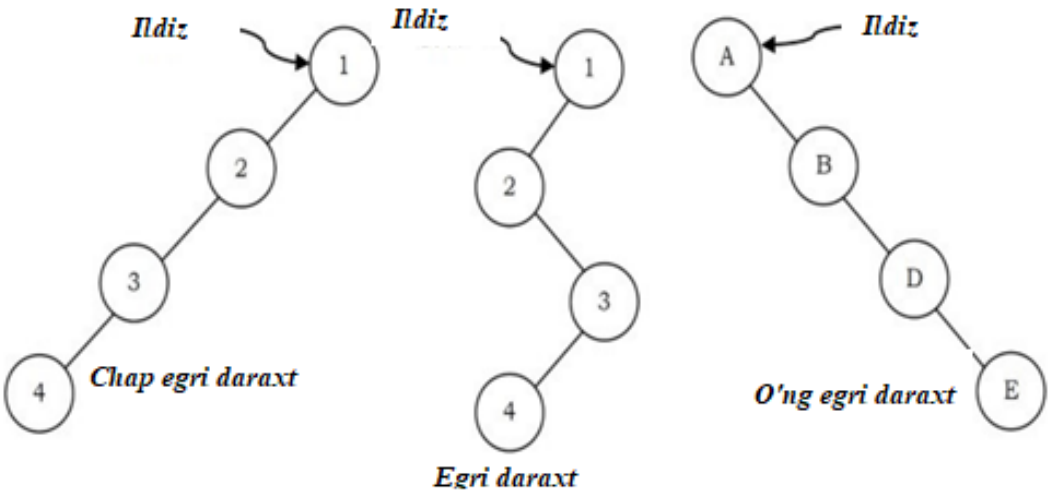


• Tugunning balandligi - bu tugundan eng chuqur tugungacha bo‘lgan yo‘lning uzunligi hisoblanadi. Daraxtning balandligi - bu daraxtning ildizidan eng chuqur tugungacha bo‘lgan yo‘lning uzunligidir. Faqat bitta tuguni (ildizi) bo‘lgan daraxtning balandligi nolga teng. Oldingi misolda B tugunining balandligi 2 ga teng (B - F - J);

• Daraxt balandligi daraxtdagi barcha tugunlar orasidagi maksimal balandlik, daraxt chuqurligi esa barcha daraxt tugunlari orasidagi maksimal chuqurlikdir. Berilgan daraxt uchun chuqurlik va balandlik bir xil qiymatni qaytaradi. Ammo individual tugunlar uchun biz turli xil natijalarni olishimiz mumkin;

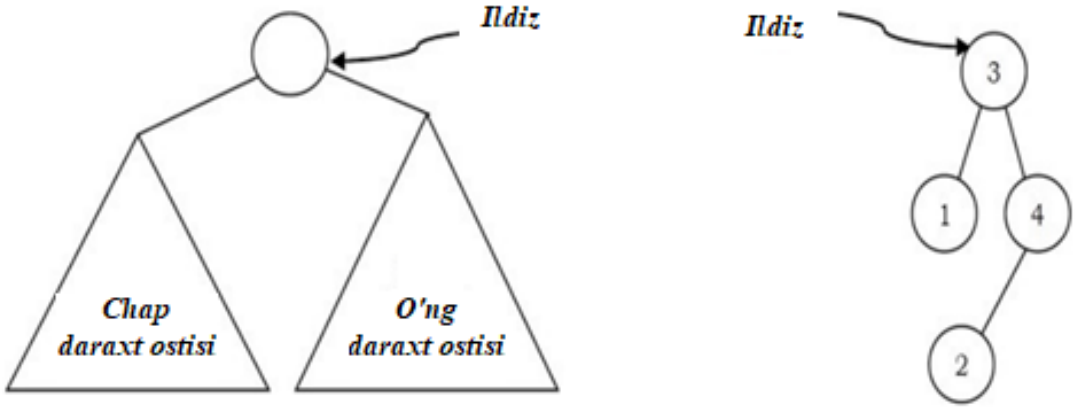
• Tugunning o‘lchami - bu tugun avlodlari soniga (hisobga o‘zi ham kiradi) aytiladi (masalan, C daraxt ostining o‘lchami 3 ga teng);

• Daraxtning har bir tugunida faqat bittadan bola bo‘lsa (barg tugunidan tashqari), bunday daraxtni egri daraxt deb ataymiz. Har bir tugunning faqat chap bolasi bo‘lsa, biz bunday daraxtni chap egri deb ataymiz. Xuddi shunday, agar har bir tugunning faqat o‘ng bolasi bo‘lsa, biz ularni o‘ng egri daraxti deb ataymiz.



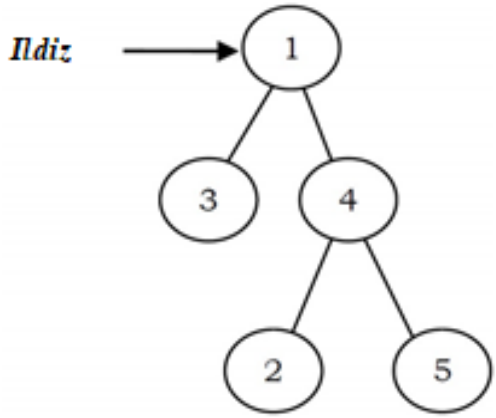
9.2. Binar (Ikkilik) daraxtlar

Agar har bir tugunda ikkitadan ortiq bola tugun bo‘lmasa, bunday daraxt binar (ikkilik) daraxt deb ataladi. Bo‘sh daraxt ham haqiqiy binar daraxt hisoblanadi. Binar daraxtni ildiz tugun va ildizdan chiquvchi, o‘zaro kesishmaydigan ikkita binar daraxt, ya’ni ildizning chap va o‘ng daraxt ostilaridan iborat deb tasavvur qilish mumkin.

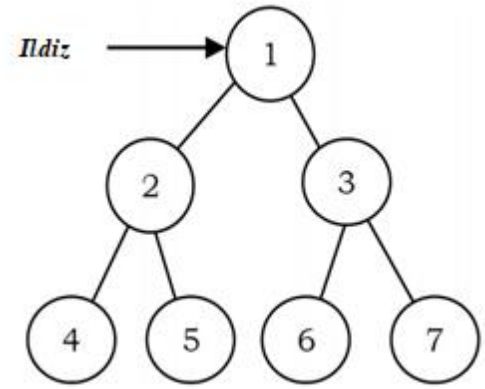


9.2.1. Binar daraxtlarning turlari

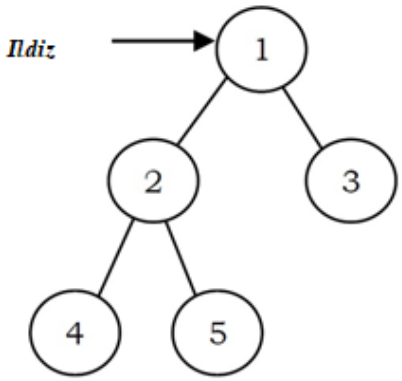
Qat’iy binar daraxt: Agar binar daraxtning har bir tuguni faqat ikkita bola tugundan iborat bo‘lsa yoki umuman tugunga ega bo‘lmasa, bunday binar daraxtga qat’iy binar daraxt deb aytiladi.



To‘liq binar daraxt: Agar har bir tugunda faqat ikkitadan bola tugun bo‘lsa va barcha barg tugunlari bir xil pog‘onada joylashgan bo‘lsa, bunday binar daraxt to‘liq binar daraxt deb ataladi.



Mukammal binar daraxt. Mukammal binar daraxtni aniqlashdan oldin, ikkilik daraxtning balandligi h deb faraz qilaylik. Mukammal binar daraxtlarda, agar biz tugunlar uchun raqamlashni ildizdan boshlab beradigan bo‘lsak (aytaylik, ildiz tugunida 1 bor), u holda biz 1 dan daraxtdagi tugunlar soniga qadar to‘liq ketma-ketlikni raqamlab chiqamiz (quyida joylashgan rasmdagi kabi). Daraxt bo‘ylab o‘tib chiqish paytida biz *NULL* ko‘rsatkichlari uchun ham raqamlashimiz kerak. Agar binar daraxtning barcha barg tugunlari *h* yoki *h* - 1 balandlikda joylashgan bo‘lsa, shuningdek tugunlar ketma-ketligini raqamlashda hech qanday qiymat yetishmayotgan bo‘lsa, bunday binar daraxt mukammal binar daraxt deb ataladi.



9.2.2. Binar daraxtlarning xususiyatlari

Batafsil fikr yuritish uchun daraxtning balandligi *h* deb faraz qilinadi. Shuningdek, ildiz tugunni (root) nolinchi balandlikda joylashgan deb hisoblanadi.

• To‘liq binar daraxtdagi tugunlar soni $n = 2^{h+1} - 1$. Ya’ni *h* pog‘ona bo‘lganligi uchun har bir pog‘onadagi barcha tugunlarni qo‘shishimiz kerak bo‘ladi: $[2^0 + 2^1 + 2^2 + + 2^h = 2^{h+1} - 1]$;

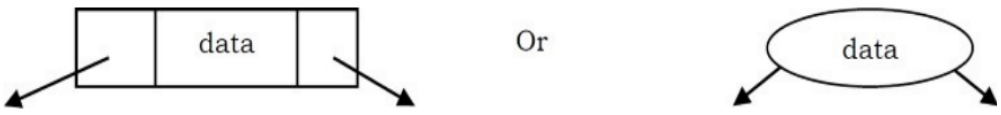
• Mukammal binar daraxtdagi tugunlar soni n 2^h (minimal) va $2^{h+1} - 1$ (maksimal) oralig‘ida yotadi.

• To‘liq binar daraxtdagi barg tugunlari soni 2^h ga teng;

• *n* ta tugunli mukammal binar daraxtidagi *NULL* havolalar soni (bo‘sh ko‘rsatkichlar) $n + 1$ ga teng.

9.2.3. Binar daraxt BATi

Endi binar daraxtning tuzilmasini aniqlaymiz. Tugunni (berilganlarni o‘z ichiga olgan) tasvirlash usullaridan biri quyida ko‘rsatilganidek, berilganlar maydonlari bilan birga chap va o‘ng bola tugunlarga ishora qiluvchi ikkita havolaga ega bo‘lishdir:




```
using System;
using System.Collections.Generic;
```

```
using System.Text;
namespace BinaryTree
{
  // Ota-onaga nisbatan tugun joylari
  public enum Side
  {
    Left,
    Right
  }
  public class BTNode<T> where T : IComparable
  {
    // Binar daraxt tuguni
    // Konstruktor
    public BTNode(T data)
    {
      Data = data;
    }
    // Tugunda saqlanadigan berilganlar
    public T Data { get; set; }
    // Chap shoxi
    public BTNode<T> Left { get; set; }
    // O‘ng shoxi
    public BTNode<T> Right { get; set; }
    // Ota-onasi
    public BTNode<T> Parent { get; set; }
    // Tugunning ota-onasiga nisbatan joylashuvi
    public Side? NodeSide => Parent == null? (Side?)null
: Parent.Left == this ? Side.Left : Side.Right;
    // Sinf ekzamplyarini satrga aylantirish
   public override string ToString() => Data.ToString();
  }
}
// Binar daraxtni aniqlovchi sinf
public class BinaryTree<T> where T : IComparable
```

```
{
  // Binar daraxt ildizi
  public BTNode<T> Root { get; set; }
  // usullar tavsifi
}
```

9.2.4. Binar daraxt ustida bajariladigan amallar

Asosiy amallar:

• Daraxtga element qo‘shish;

• Daraxtdan elementni olib tashlash;

• Daraxtdan elementni qidirish;

• Daraxt bo‘ylab o‘tib chiqish.

Yordamchi amallar:

• Daraxtning o‘lchamini aniqlash;

• Daraxtning balandligini aniqlash;

• Maksimal yig‘indili pog‘onani aniqlash;

• Berilgan tugunlar juftligi uchun eng kichik umumiy ajdodni (LCA) topish.

Binar daraxt ilovalari

Quyida binar daraxtlar muhim ro‘l o‘ynaydigan ba’zi ilovalar keltirilgan:

• Sintaksis tahlil daraxtlari kompilyatorlarda qo‘llaniladi;

• Ma'lumotlarni siqish algoritmlarida qo‘llaniladigan Haffman kodlash daraxtlari;

• O(log*n*) (o‘rtacha) amallarida elementlarni qidirish, kiritish va o‘chirishni qo‘llab-quvvatlaydigan ikkilik qidirish daraxti (BST);

• Logarifmik vaqt (eng yomon holat) ichida elementlar to‘plamidan minimal (yoki maksimal) elementni topish va o‘chirishni qo‘llab-quvvatlaydigan Priority Queue (PQ).

9.2.5. Binar daraxt bo‘ylab o‘tish

Daraxt bo‘ylab o‘tib chiqish uchun bizga daraxt bo‘ylab o‘tish mexanizmi kerak bo‘ladi.

Daraxtning barcha tugunlariga tashrif buyurish jarayoni daraxt bo‘ylab o‘tish deb ataladi. Har bir tugun faqat bir marta qayta ishlanadi, lekin unga

bir necha marta tashrif buyurish mumkin. Chiziqli tuzilmalarda (masalan, bog‘langan ro‘yxatlar, steklar, navbatlar va boshqalar) ko‘rganimizdek, elementlar ketma-ket tartibda qarab chiqiladi. Ammo, daraxt tuzilmalarida o‘tishning ko‘plab turli usullari mavjud.

Daraxt bo‘ylab o‘tish daraxtdan elementni qidirishga o‘xshaydi, faqat o‘tish maqsadi daraxtni ma'lum bir tartibda kesib o‘tishdir. Bundan tashqari, daraxt bo‘ylab o‘tishda barcha tugunlar qayta ishlanadi va qidirish jarayonida esa kerakli tugun topilganda qidirish jarayoni to‘xtaydi.

Mumkin bo‘lgan o‘tish turlari

Ikkilik daraxtning ildizidan boshlab, bajarilishi mumkin bo‘lgan uchta asosiy bosqichi mavjud va ularni bajarish tartibi o‘tish turini belgilaydi.

Bu qadamlar quyidagilardan iborat:

Harakat joriy tugun ustida amalga oshiriladi ("tashrif buyuriladigan" tugun deb nomlanadi va bu harakat "D" harfi bilan belgilanadi),

Chap bola tugunga o‘tish bajariladi ("L" harfi bilan belgilanadi),

O‘ng bola tuguniga o‘tish bajariladi ("R" harfi bilan belgilanadi).

Ushbu jarayonni rekursiya yordamida osongina tavsiflash mumkin. Yuqoridagi ta'rifga asoslanib, 6 ta imkoniyat mavjud:

1.LDR: chap daraxt ostini qayta ishlash, joriy tugunning ma'lumotlarini qayta ishlash va keyin o‘ng daraxt ostini qayta ishlash;

2. LRD: chap daraxt ostini, o‘ng daraxt ostini qayta ishlash va keyin joriy tugunning ma'lumotlarini qayta ishlash;

3. DLR: joriy tugunning ma'lumotlarini qayta ishlash, chap daraxt ostini qayta ishlash va keyin o‘ng daraxt ostini qayta ishlash;

4. DRL: joriy tugunning ma'lumotlarini qayta ishlash, o‘ng daraxt ostini qayta ishlash va keyin chap daraxt ostini qayta ishlash;

5. RDL: o‘ng daraxt ostini qayta ishlash, joriy tugunning ma'lumotlarini qayta ishlash va keyin chap daraxt ostini qayta ishlash;

6. RLD: o‘ng daraxt ostini, chap daraxt ostini qayta ishlash va so‘ngra joriy tugunning ma'lumotlarini qayta ishlash.

9.2.6. Binar daraxt bo‘ylab o‘tishlar tasniflanishi

Ushbu obyektlar (tugunlar) qayta ishlanadigan ketma-ketlik o‘tishning aniq uslubini aniqlaydi.

Tasniflash joriy tugunning ishlash tartibiga asoslanadi.

Bu shuni anglatadiki, agar biz joriy tugun (D) asosida tasniflasak va D o‘rtada bo‘lsa, unda L va R qayerda bo‘lishining ahamiyati yo‘q. Bu borada mumkin bo‘lgan variantlarning umumiy soni 3 taga kamayadi va bu quyidagicha:

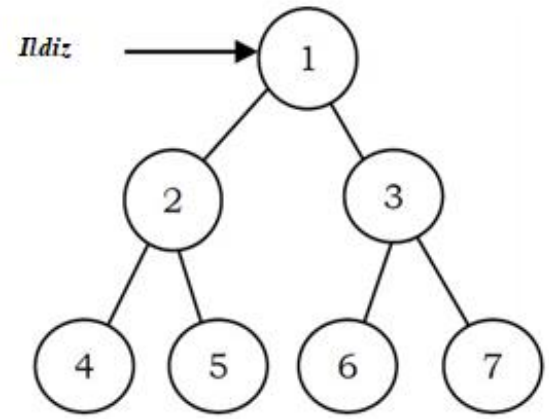
• o‘tishning prefiksli tartibi (DLR) (to‘g‘ridan-to‘g‘ri o‘tish)

• o‘tishning infiksli tartibi(LDR) (simmetrik o‘tish)

• o‘tishning postfiksli tartibi (LRD) (teskari o‘tish)

Yuqoridagi tartiblarga bog‘liq bo‘lmagan o‘tishning yana bir usuli mavjud bo‘lib, bu o‘tishning daraja bo‘ylab tartibidir: bu usul “Kenglik bo‘ylab o‘tish” (BFS) ga asoslangan.

Keling, keyingi muhokama uchun quyidagi diagrammadan foydalanamiz.



9.2.6.1. Binar daraxt bo‘ylab o‘tishning prefiksli tartibi

Prefiks o‘tishida har bir tugun uning har qanday daraxt ostigacha o‘tadi. Bu tushunish uchun eng oddiy yechim. Biroq, har bir tugun daraxt ostigacha qayta ishlangan bo‘lsa ham, daraxt bo‘ylab pastga siljish paytida ba'zi ma'lumotlarni saqlab qolish talab etiladi. Yuqoridagi misolda birinchi navbatda 1, keyin chap daraxt osti, so‘ngra o‘ng daraxt osti qayta ishlanadi. Shuning uchun, ishlov berish chap daraxt ostini qayta ishlashni tugatgandan so‘ng o‘ng daraxt ostiga qaytishi kerak. Chap daraxt ostini qayta ishlagandan so‘ng o‘ng daraxt ostiga o‘tish uchun o‘ng daraxt osti haqida ma'lumot olish uchun ildizga qaytishimiz kerak. Ushbu ma'lumotni saqlashning aniq usuli

bu stek hisoblanadi. LIFO tuzilmasi tufayli teskari tartibda to‘g‘ri daraxt ostlari haqida ma'lumot olish mumkin.

O‘tishning prefiksli tartibi quyidagicha belgilanadi:

• Ildiz tuguni qayta ishlanadi;

• Chap daraxt osti bo‘ylab prefiksli tartibda o‘tiladi;

• O‘ng daraxt osti bo‘ylab prefiksli tartibda o‘tiladi.

Daraxt tugunlariga quyidagi tartibda tashrif buyuriladi: 1 2 4 5 3 6 7

Daraxt bo‘ylab o‘tishning prefiksli tartibi uchun rekursiv funksiya:

```
// Daraxt bo‘ylab o‘tishninig Prefiks tartibi
// Rekursiv chiqish usuli
//  Belgilangan tugundan boshlab ikkilik daraxtni chiqarish
// parametr startNode - Chop etish boshlanadigan tugun
// parametr indent -  chekinishi
// side parametr - Daraxt tomoni
private void PrintPreRec(BTNode<T> startNode, string indent = " ", Side? side = null)
{
  if (startNode != null)
  {
    var nodeSide = side == null ? "+" : side == Side.Left ? "L" : "R";
    Console.WriteLine($"{indent}          [{nodeSide}]-{startNode.Data}");
    indent += new string(' ', 5);
    //chap va o‘ng shoxlari uchun rekursiv chaqiruv
    PrintPreRec(startNode.Left, indent, Side.Left);
    PrintPreRec(startNode.Right, indent, Side.Right);
  }
}
```

Algoritmning murakkabligi O(*n*) ga teng.

Rekursiv bo‘lmagan versiyada stek kerak, chunki biz chap daraxt ostini bosib o‘tishni tugatgandan so‘ng, o‘ng daraxt ostiga o‘tishimiz uchun hozirgi tugunni eslashimiz kerak. Buni modellashtirish uchun avvalo joriy

tugun qayta ishlanadi va chap daraxt ostiga borishdan oldin, joriy tugunni stekka saqlaymiz. Chap daraxt ostini qayta ishlashni tugatgandan so‘ng, element stekdan chiqarib olinadi va o‘ng daraxt ostiga o‘tiladi. Bu jarayon stek bo‘sh bo‘lgunga qadar davom ettiriladi.

Rekursiv bo‘lmagan funksiya:

```
// Daraxt bo‘ylab o‘tishninig Prefiks tartibi
// Rekursiv bo‘lmagan chiqish usuli
public void PrintPreNoRec()
{
  string indent = " ";
  Side? side = null;
  Stack<BTNode<T>> S = new Stack<BTNode<T>>();
  Stack<string> Sin = new Stack<string>();
  BTNode<T> temp = Root;
  while (true)
  {
    while (temp != null)
    {
//joriy tugunni qayta ishlash
      side = temp.NodeSide;
      var nodeSide = side == null ? "+" : side ==
Side.Left ? "L" : "R";
      Console.WriteLine($"{indent}        [{nodeSide}]-
{temp.Data}");
      indent += new string(' ', 5);
      S.Push(temp);
      Sin.Push(indent);
      temp=temp.Left;
    }
    if (S.Count == 0) break;
      indent = Sin.Pop();
      temp = S.Pop();
      temp = temp.Right;
  }
```

```
}
```

Ushbu o‘tishda tugunlarga yuqoridan pastga va chapdan o‘ngga ishlov beriladi.

Algoritmning murakkabligi O(*n*) ga teng.

9.2.6.2. Binar daraxt bo‘ylab o‘tishning infiksli tartibi.

Infiksli o‘tish tartibida, ildizni qayta ishlash daraxt ostlarining orasida amalga oshiriladi.

O‘tish tartibi quyidagicha:

- Chap daraxt osti bo‘ylab infiksli tartibda o‘tiladi;
- Ildiz tugun qayta ishlanadi;
- O‘ng daraxt osti bo‘ylab infiksli tartibda o‘tiladi.

Daraxt tugunlariga quyidagi tartibda tashrif buyuriladi: 4 2 5 1 6 3 7

Daraxt bo‘ylab o‘tishning prefiksli tartibi uchun rekursiv funksiya:

```
// Daraxt bo‘ylab o‘tishninig Infiks tartibi
// Rekursiv chiqish usuli

private void PrintInfRec(BTNode<T> startNode, string
indent = " ", Side? side = null)
{
  if (startNode != null)
  {
    PrintInfRec(startNode.Left, indent, Side.Left);
    var nodeSide = side == null ? "+" : side == Side.Left
? "L" : "R"
    Console.WriteLine($"{indent}          [{nodeSide}]-
{startNode.Data}");
    indent += new string(' ', 5);
    PrintInfRec(startNode.Right, indent, Side.Right);
  }
}
```

Algoritmning murakkabligi O(*n*) ga teng.

Rekursiv bo‘lmagan infiksli tartibda o‘tish

O‘tishning rekursiv bo‘lmagan versiyasi prefiks versiyasiga o‘xshaydi. Faqatgina o‘zgarish shundaki, chap daraxt ostiga o‘tishdan oldin tugunni

qayta ishlashning o‘rniga, uni chap daraxt osti qayta ishlangandan so‘ng va u stekdan chiqqandan keyin qayta ishlanadi.

Rekursiv bo‘lmagan funksiya:

```
// Daraxt bo‘ylab o‘tishning Infiks tartibi
// Rekursiv bo‘lmagan chiqish usuli
public void PrintInfNoRec()
{
  string indent = " ";
  Side? side = null;
  Stack<BTNode<T>> S = new Stack<BTNode<T>>();
  Stack<string> Sin = new Stack<string>();
  BTNode<T> temp = Root;
  while (true)
  {
    while (temp != null)
    {
      S.Push(temp);
      Sin.Push(indent);
      temp = temp.Left;
    }
    if (S.Count == 0) break;
      indent = Sin.Pop();
    temp = S.Pop();
//Joriy tugunni qayta ishlash
    side = temp.NodeSide;
    var nodeSide = side == null ? "+" : side == Side.Left
? "L" : "R";
    Console.WriteLine($"{indent}          [{nodeSide}]-
{temp.Data}");
    indent += new string(' ', 5);
    temp = temp.Right;
  }
}
```

Algoritmning murakkabligi O(*n*) ga teng.

Ushbu o‘tishda tugunlarga ildizga nisbatan simmetrik tarzda ishlov beriladi.

9.2.6.3. Binar daraxt bo‘ylab o‘tishning postfiksli tartibi

Postfiks o‘tish tartibida daraxt ostlari qayta ishlangandan keyin ildiz tugun qayta ishlanadi.

O‘tish tartibi quyidagicha:

- Postfiksli tartibda chap daraxt osti bo‘ylab o‘tiladi;
- Postfiksli tartibda o‘ng daraxt osti bo‘ylab o‘tiladi;
- Ildiz tugun qayta ishlanadi.

Daraxt tugunlariga quyidagi tartibda tashrif buyuriladi: 4 5 2 6 7 3 1

Postfiksli tartibda daraxt bo‘ylab o‘tishning rekursiv funksiyasi:

```
// Daraxt bo‘ylab o‘tishning Postfiks tartibi
// Rekursiv chiqish usuli
private void PrintPosRec(BTNode<T> startNode, string
indent = " ", Side? side = null)
{
  if (startNode != null)
  {
    PrintPosRec(startNode.Left, indent, Side.Left);
    PrintPosRec(startNode.Right, indent, Side.Right);
    var nodeSide = side == null ? "+" : side == Side.Left
? "L" : "R";
    Console.WriteLine($"{indent}          [{nodeSide}]-
{startNode.Data}");
    indent += new string(' ', 5);
  }
}
```

Algoritmning murakkabligi O(*n*) ga teng.

Rekursiv bo‘lmagan postfiksli tartibda o‘tish.

Postfiksli o‘tishda, navbatdagi ishlov berishdan so‘ng, har bir tugunga ikki marta tashrif buyuriladi. Darhaqiqat, chap daraxt ostini qayta ishlagandan so‘ng, biz joriy tugunga tashrif buyuramiz va o‘ng daraxtni qayta ishlagandan so‘ng, biz yana joriy tugunga qaytamiz. Ammo biz ikkinchi tashrifdan keyin tugunni qayta ishlashimiz kerak. Bu yerda

muammo chap daraxt ostidan yoki o‘ng daraxt ostidan qaytib kelayotganimizni qanday farqlashda. Ushbu muammoni hal qilish uchun avval o‘zgargan tugunni kuzatib borish uchun oldingi o‘zgaruvchidan foydalanamiz.

Rekursiv bo‘lmagan funksiya:

```
// Daraxt bo‘ylab o‘tishning Postfiks tartibi
// Rekursiv bo‘lmagan chiqish usuli
public void PrintPosNoRec()
{
  string indent = " ";
  Side? side = null;
  Stack<BTNode<T>> S = new Stack<BTNode<T>>();
  Stack<string> Sin = new Stack<string>();
  BTNode<T> temp = Root;
  BTNode<T> prev = Root;
  do
  {
    while (temp != null)
    {
      S.Push(temp);
      Sin.Push(indent);
      temp = temp.Left;
    }
    while (temp  == null && S.Count >0)
    {
      temp = S.Peek();
      if (temp.Right==null || temp.Right==prev)
      {
      //Joriy tugunni qayta ishlash
        side = temp.NodeSide;
        var nodeSide = side == null ? "+" : side ==
Side.Left ? "L" : "R";
        Console.WriteLine($"{indent}      [{nodeSide}]-
{temp.Data}");
```

```
            indent += new string(' ', 5);
            S.Pop(); prev = temp; temp= null;
          }
          else temp = temp.Right;
        }
    } while (S.Count != 0);
}
```

Algoritmning murakkabligi O(*n*) ga teng.

Ushbu o‘tishda tugunlarni pastdan yuqoriga, chapdan o‘ngga ishlov beriladi.

9.2.6.4. Binar daraxt kengligi bo‘yicha o‘tish

O‘tish tartibi quyidagicha:

• Ildiz tugun qayta ishlanadi;

• Navbati bilan joriy darajadagi barcha tugunlar bo‘yicha o‘tiladi;

• Keyingi darajaga o‘tiladi va ushbu darajadagi barcha tugunlarga tashrif buyuriladi;

• Ushbu jarayon barcha darajalardan o‘tib bo‘linguncha takrorlanadi.

Daraxt tugunlariga quyidagi tartibda tashrif buyuriladi: 1 2 3 4 5 6 7

```
// Daraxt kengligi bo‘ylab o‘tish
// Rekursiv bo‘lmagan xulosa chiqarish usuli
public void PrintBFS()
{
  BTNode<T> temp;
  string indent = " ";
  Side? side = null;
  Queue<BTNode<T>> Q = new Queue<BTNode<T>>();
  Queue<string> Qin = new Queue<string>();
  Qin.Enqueue(indent);
  Q.Enqueue(Root);
  while (Q.Count > 0)
  {
    temp = Q.Dequeue();
    indent = Qin.Dequeue();
    side = temp.NodeSide;
```

```
    var nodeSide = side == null ? "+" : side == Side.Left
? "L" : "R";
    Console.WriteLine($"{indent}          [{nodeSide}]-
temp.Data}");
    indent += "     ";
    if (temp.Left != null)
    {
      Q.Enqueue(temp.Left);
      Qin.Enqueue(indent);
    }
    if (temp.Right != null)
    {
      Q.Enqueue(temp.Right);
      Qin.Enqueue(indent);
    }
  }
}
```

Algoritmning murakkabligi O(*n*) ga teng.

9 – bob bo‘yicha nazorat savollari

1. Daraxt - berilganlarni abstrakt turi sifatida.
2. Daraxtning qanday asosiy xususiyatlari mavjud?
3. Binar daraxt tushunchasi.
4. Qat’iy binar daraxt nima?
5. To‘liq binar daraxt nima?
6. Mukammal binar daraxt nima?
7. Binar daraxtlarning xususiyatlari.
8. Binar daraxt bo‘ylab o‘tish turlari.
9. Binar daraxt bo‘ylab o‘tishlar tasniflanishi.
10. Binar daraxt bo‘ylab o‘tishning prefiksli tartibi qanday amalga oshiriladi?
11. Binar daraxt bo‘ylab o‘tishning infiksli tartibi qanday amalga oshiriladi?
12. Binar daraxt bo‘ylab o‘tishning postfiksli tartibi qanday amalga oshiriladi?

10. Tartiblash.

Tartiblash - bu ro‘yxat elementlarini ma'lum bir tartibda [o‘sish yoki kamayish bo‘yicha] joylashtiradigan algoritmdir.

Tartiblash natijasi kiruvchi berilganlarning qayta tartiblangan ro‘yxati hisoblanadi.

Tartiblash bu kompyuter ilmida ko‘p tadqiqotlarni talab qiladigan algoritmlarning muhim toifalaridan biri hisoblanadi. Tartiblash yordamida masala murakkabligini sezilarli darajada kamaytirish mumkin va u ko‘pincha berilganlar bazasi va qidirish algoritmlarida keng qo‘llaniladi.

10.1. Tartiblash algoritmlarni tasniflash

Tartiblash algoritmlari odatda quyidagi parametrlar asosida toifalarga ajratiladi.

Taqqoslash amallari soni bo‘yicha

Bunday holda, tartiblash algoritmlari taqqoslash amallari soniga qarab tasniflanadi va eng yaxshi holatda O(*nlogn*) va eng yomon holatda O(n^2) hisoblash murakkabligiga ega bo‘ladi.

Taqqoslashlardan foydalanadigan algoritmlardan tashqari, taqqoslashlarsiz (chiziqli) algoritmlar ham mavjud bo‘lib, biz ularni ham muhokama qilamiz.

Almashtirishlar soni bo‘yicha

Bunday holda, tartiblash algoritmlari tartiblash jarayonida amalga oshirilgan almashtirishlar soni bo‘yicha tasniflanadi.

Tezkor xotiradan foydalanishi bo‘yicha

Ba'zi tartiblash algoritmlari "joyida" bajariladi va ularga vaqtinchalik berilganlarni saqlash uchun kichik hajmdagi O(1) yoki O(*logn*) joy talab qilinadi, ya'ni kichik xotira talab qilinadi. Bunday algoritmlar joyida tartiblash algoritmlari deyiladi.

O(*n*) - kirish berilganlari o‘lchamida qo‘shimcha xotiradan foydalanadigan boshqa algoritmlar ham mavjud.

Rekursiya bo‘yicha

Tartiblash algoritmlari rekursiv (tezkor tartiblash) yoki rekursiv bo‘lmagan (tanlash orqali tartiblash, kiritish orqali tartiblash, pufakchali

tartiblash) bo‘lishi mumkin va ikkala usuldan ham foydalanadigan bir nechta algoritmlar mavjud (birlashtirish orqali tartiblash).

Turg‘unlik bo‘yicha

Agar tartiblash algoritmi elementlarning nisbiy tartibini saqlasa u holda bunday tartiblash algoritmlari barqaror hisoblanadi. Ya'ni, agar barcha *i* va *j* indekslari uchun A[i] kalit A[j] kalitiga teng bo‘lsa, hamda dastlabki faylda R[i] yozuvi R[j] yozuvidan oldin bo‘lib, tartiblangan ro‘yxatda ham R[i] yozuvi R[j] yozuvidan oldin kelsa bunday tartiblash barqaror hisoblanadi. Ushbu mezon yozuvlarni kalitlar bo‘yicha tartiblash uchun ishlatiladi va bir xil kalitga ega yozuvlar tartiblangan massivda o‘z ketma-ketligini saqlab qolishini anglatadi.

Moslashuvchanlik bo‘yicha

Tartiblash algoritmidan foydalanganda uning murakkabligi dastlabki kiruvchi berilganlarning oldindan tartiblanganligiga (tezkor tartiblash) qarab o‘zgarishi, ya'ni kirivchi berilganlarning oldindan tartiblanganligi tartiblashning bajarilish vaqtiga ta'sir qilishi mumkin. Buni hisobga oladigan algoritmlar adaptiv algoritmlar deb ataladi.

Boshqa tasniflash

Tartiblash algoritmlarining boshqa tasniflash usullari ham mavjud:

• Ichki tartiblash;

• Tashqi tartiblash.

Ichki tartiblash

Tartiblash paytida faqat asosiy xotiradan foydalanadigan tartiblash algoritmlari ichki tartiblash algoritmlari deyiladi.

Bunday algoritmlar barcha xotiraga yuqori tezlikdagi kirish ruxsatiga ega deb faraz qilinadi.

Tashqi tartiblash

Tartiblash paytida lenta yoki disk kabi tashqi xotiradan foydalanadigan tartiblash algoritmlariga tashqi tartiblash algoritmlari deyiladi.

10.2. Tartiblash algoritmlari

Tartiblashning bir nechta algoritmlarini ko‘rib chiqaylik.

10.2.1. Pufakchali tartiblash (almashtirish orqali tartiblash)

Pufakchali tartiblash eng oddiy tartiblash algoritmidir. U kirish massivining birinchi elementdan oxirgi elementga qadar o‘tib chiqadi va har bir elementlar juftligini solishtiradi va kerak bo‘lganda ularni almashtiradi. Pufakchali tartiblash boshqa almashtirishlar kerak bo‘lmaguncha itaratsiyani davom ettiradi. Algoritm o‘z nomini katta qiymatli elementlarning ro‘yxatning oxiriga "suzib chiqishi" tufayli oldi. Bu eng oson tartiblash usuli, lekin ayni paytda eng ko‘p vaqt talab qiladi. Bu “joyida” tartiblash algoritmidir.

Pufakchali tartiblashning boshqa tartiblash algoritmlarga nisbatan yagona muhim afzalligi shundaki, u kiruvchi berilganlar ro‘yxatining tartiblangan yoki yo‘qligini darhol aniqlashi mumkin.

Amalga oshirilishi:

```
void BubbleSort(int[] A)
{
  int tmp;
  for (int k = A.Length - 1; k >= 0; k--)
  {
    for (int i = 0; i <= k - 1; i++)
      if (A[i] > A[i + 1])
      {
        tmp = A[i];
        A[i] = A[i + 1];
         A[i + 1] = tmp;
      }
  }
}
```

Algoritm O(n^2) murakkablikka ega (hatto eng yaxshi holatda ham).

Joriy o‘tish joyida almashtirishlar bo‘lmasa, tartiblashni tugatishga ruxsat berish uchun biz uni bitta qo‘shimcha o‘zgaruvchi yordamida yaxshilashimiz mumkin.

Ushbu o‘zgartirilgan versiya pufakchali tartiblashning eng yaxshi holatini O(n) ga yaxshilaydi.

```
void BubbleSort(int[] A)
{
  int tmp, k;
  bool swp = true;
  for (k = A.Length - 1; k >= 0 && swp; k--)
  {
    swp = false;
    for (int i = 0; i <= k - 1; i++)
      if (A[i] > A[i + 1])
      {
        tmp = A[i];
        A[i] = A[i + 1];
        A[i + 1] = tmp;
        swp = true;
      }
  }
}
```

Algoritmning murakkabligini baholash:

- Eng yomon holatdagi murakkabligi: O(n^2);
- Eng yaxshi holatdagi murakkabligi (kengaytirilgan versiya): O(n);
- O‘rtacha murakkabligi (Asosiy versiya): O(n^2);
- Qo‘shimcha xotiradan foydalanilgan eng yomon holatda: O(1) .

10.2.2. Tanlash orqali tartiblash

Tanlash orqali tartiblash joyida tartiblash algoritmidir. Tanlash orqali tartiblash kichik fayllar uchun yaxshi ishlaydi. Katta hajmdagi berilganlarga ega yozuvlarni tartiblash uchun samarali hisoblanadi, chunki almashtirishlar faqat zarur bo‘lganda, kalit qiymatlarga qarab amalga oshiriladi.

Afzalliklari:

• Amalga oshirishning soddaligi;

• Joyida tartiblash (saqlash uchun qo‘shimcha joy talab qilmaydi).

Kamchiliklari:

• Yomon masshtablashgan: O(n^2), ya’ni n ortishi bilan tartiblash vaqti ortadi.

Algoritm quyidagicha:

1. Ro‘yxatdagi minimal qiymat topiladi;

2. Uni joriy o‘rindagi qiymat bilan almashtiriladi;

3. Butun ro‘yxat tartiblangunga qadar bu jarayon barcha elementlar uchun takrorlanadi.

Bu algoritm eng kichik elementni qayta-qayta tanlaganligi uchun tanlash orqali tartiblash deb ataladi.

Amalga oshirish:

```
void SelectionSort(int[] A)
{
  int min, tmp;
  int n = A.Length;
  for (int i = 0; i < n; i++)
  {
    min = i;
    for (int j = i + 1; j < n; j++)
      if (A[j] < A[min]) min = j;
    tmp = A[min];
    A[min] = A[i];
    A[i] = tmp;
  }
}
```

Algoritm murakkabligi tahlili.

- Eng yomon holatdagi murakkabligi: O(n^2);
- Eng yaxshi holatdagi murakkabligi: O(n^2);
- O‘rtacha murakkabligi: O(n^2);
- Eng yomon holatda qo‘shimcha xotira sarfi: O(1).

10.2.3. Kiritish orqali tartiblash

Kiritish orqali tartiblash - bu oddiy va samarali taqqoslash orqali tartiblash usuli hisoblanadi. Ushbu algoritmda itaratsiyaning har bir bosqichida kiruvchi berilganlardan element olib tashlanadi va u tartiblangan ro‘yxatning to‘g‘ri joyiga qayta kiritiladi. Kiruvchi berilganlardan elementni olib tashlash tasodifiy bo‘lib, bu jarayon dastlabki ro‘yxatning barcha elementlari o‘tib ketguncha takrorlanadi.

Afzalliklari

• Amalga oshirishning oddiyligi;

• Kichik berilganlar uchun samarali;

• Moslashuvchan: agar kiruvchi berilganlar ro‘yxati oldindan saralangan bo‘lsa (to‘liq bo‘lmasligi mumkin), u holda kiritish orqali tartiblash O($n + d$) amal bajaradi, bu yerda d – kiritishlar soni;

• Tanlash orqali tartiblash va pufakchali tartiblashdan ko‘ra amalda samaraliroq, garchi ularning barchasi O(n^2) - eng yomon murakkablik darajasiga ega bo‘lsa ham;

• Barqaror: agar kalitlar mos kelsa, kiruvchi berilganlarning nisbiy tartibini saqlaydi;

• Joyida tartiblash hisoblanadi, ya'ni faqat doimiy miqdorda O(1) qo‘shimcha xotira talab qiladi;

• Tezkor: Kiritish orqali tartiblash ro‘yxatni kiritilgan berilganlarga qarab tartiblashi mumkin.

Kiritish orqali tartiblash algoritmning tavsifi

Kiritish orqali tartiblashning har bir siklida elementni kiruvchi berilganlardan olib tashlaydi (uni vaqtinchalik xotirada saqlaydi) va kiruvchi berilganlar ro‘yxatida element qolmaguncha uni tartiblangan ro‘yxatning kerakli joyiga qo‘yadi. Tartiblash odatda mahalliy darajada amalga oshiriladi. k takrorlashdan keyin hosil bo‘lgan massivning birinchi $k + 1$ ta yozuvlari tartiblangan bo‘ladi.

Misol:

Kiruvchi berilganlar massivi berilgan: 6 8 1 4 5 3 7 2 va maqsad ularni o‘sish tartibida joylashtirishdir.

6 8 1 4 5 3 7 2 birinchi element joyida qoladi;

6 8 1 4 5 3 7 2 ikkinchi element joyida qoladi;

1 6 8 4 5 3 7 2 uchinchi element olib tashlanadi va birinchi o‘ringa qo‘yiladi;

1 4 6 8 5 3 7 2 jarayon har bir keyingi element bilan davom etadi;

1 4 5 6 8 3 7 2

1 3 4 5 6 8 7 2

1 3 4 5 6 7 8 2

1 2 3 4 5 6 7 8 oxirgi element joyiga kiritilgan.

Amalga oshirish

```
void InsertionSort(int [] A)
{
  int i, j, v;
  int n = A.Length;
  for (i = 1; i < n; i++)
  {
    v = A[i]; j = i;
    while (j >= 1 && A[j - 1] > v )
    {
      A[j] = A[j - 1]; j--;
    }
    A[j] = v;
  }
}
```

Algoritm tahlili

Eng yomon holat tahlili:

Eng yomon holatda har bir *i* uchun barcha elementlarini siljitishi kerak *A[1],. . . , A [i - 1]* (ya'ni *A[i]* qiymati ularning barchasidan kichik) , bu θ(i - 1) vaqtni oladi.

Umumiy vaqt quyidagicha hisoblanadi:

$T(n)= \theta(1)+ \theta(2)+ \theta(3) \ldots + \theta(n - 1)= \theta(1+2+3+ n- 1) = \theta(n(n-1)/2) \approx \theta(n^2)$

O‘rtacha holat tahlili:

O‘rtacha holatda ichki sikl *A[i]* ni *A[1] . . . , A [i - 1]* ning o‘rtasiga kiritadi. Bu $\theta(i/2)$ vaqtni oladi. Umumiy vaqt sarfi quyidagiga teng bo‘ladi:

$$T(n) = \sum_{i=1}^{n} \Theta(i/2) \approx \Theta(n^2)$$

Eng yaxshi holat tahlili:

Eng yaxshi holatda, ro‘yxat dastlab tartiblangan holatda bo‘lib, ichki siklda hech qanday amal bajarilmaydi, ya'ni, $T(n)= \theta(n)$.

Murakkablikni baholash:

- Eng yomon holatdagi murakkabligi: $O(n^2)$;
- Eng yaxshi holatdagi murakkabligi: $O(n)$;

- O‘rtacha murakkabligi: O(n^2);
- Eng yomon holatda qo‘shimcha xotira sarfi: O(1).

Boshqa tartiblash algoritmlari bilan solishtirish

Kiritish orqali tartiblash algoritmi vaqt bo‘yicha murakkabligi eng yomon holatda O(n^2) teng bo‘lgan eng sodda algoritmlardan biri hisoblanadi.

Kiritish orqali tartiblash dastlabki berilganlar deyarli saralanganda (uning moslashuvi tufayli) yoki kiruvchi berilganlar hajmi nisbatan kichik bo‘lganda (kam yuk tufayli) qo‘llaniladi. Shu sabablarga ko‘ra va uning barqarorligi tufayli kiritish orqali tartiblash birlashtirish orqali tartiblash (merge sort) va tezkor tartiblash (quick sort) kabi “bo‘l va hukmronlik qil” tamoyiliga asoslangan murakkabroq rekursiv algoritmlar uchun asosiy holat sifatida (muammo hajmi kichik bo‘lsa) qo‘llaniladi.

Eslatma:

• Pufakchali tartiblashda o‘rtacha va eng yomon holatda $\frac{n^2}{2}$ taqqoslash va $\frac{n^2}{2}$ almashtirish (inversiya) bajariladi;

• Tanlash orqali tartiblashda $\frac{n^2}{2}$ taqqoslash va n almashtirish bajariladi;

• Kiritish orqali tartiblashda o‘rtacha holatda $\frac{n^2}{4}$ taqqoslash va $\frac{n^2}{8}$ almashtirish bajaradi, eng yomon holatda esa ular ikkilanadi;

• Qisman tartiblangan kiruvchi berilganlar uchun kiritish orqali tartiblash deyarli chiziqli hisoblanadi;

• Tanlash orqali tartiblash kattaroq qiymatlar va kichik kalitlarga ega elementlar uchun eng mos keladi.

10.2.4. Shell tartiblash

Shell tartiblash (shuningdek, asta kamayib boruvchi tartiblash deb ham aytiladi) Donald Shell tomonidan ixtiro qilingan. Bu tartiblash kiritish orqali tartiblashning umumlashmasidir. Kiritish orqali tartiblash qisman tartiblangan kiruvchi berilganlarda samarali ishlaydi. Shell tartiblash, shuningdek, *n*-oraliqli kiritish orqali tartiblash sifatida ham tanilgan.

Kiritish orqali tartiblashda faqat qo‘shni elementlar juftliklari solishtiriladi, Shell tartiblashda bir nechta o‘tishlarni amalga oshiradi va har bir o‘tishda solishtiriladigan elementlar o‘rtasidagi oraliq turlicha bo‘ladi.

Kiritish orqali tartiblash qo‘shni elementlarni taqqoslaydi va *n* ta siljish amalini bajaradi. Shell tartiblashda qo‘llaniladigan variant quyidagicha:

Qo‘shni taqqoslash algoritmning oxirgi bosqichida amalga oshiriladi, shuning uchun Shell tartiblashning oxirgi bosqichi aslida kiritish orqali tartiblash algoritmidir. Bu bir-biridan uzoqda joylashgan elementlarni solishtirish va almashish imkonini berib, kiritish orqali tartiblashni yaxshilaydi.

Bu solishtirish orqali tartiblash algoritmlari orasida kvadratik vaqtdan kamroq vaqt ichida bajariladigan birinchi algoritmdir.

Shell tartiblash aslida kiritish orqali tartiblashning oddiy kengaytmasi hisoblanadi. Uning asosiy farqi - bu elementlarning o‘rin almashishini sezilarli darajada tezlashtirish uchun bir-biridan uzoqda joylashgan elementlarni almashtirish qobiliyati hisoblanadi.

Misol uchun, agar eng kichik element massiv oxirida bo‘lsa, kiritish orqali tartiblash ushbu elementni massivning boshiga joylashtirish uchun to‘liq siljish qadamlarini talab qiladi. Biroq, Shell tartiblash bilan bu element bir vaqtning o‘zida bir necha qadam sakrashi va kamroq almashtirishlarda to‘g‘ri manzilga yetib borishi mumkin.

Shell tartiblashning asosiy g‘oyasi massivdagi har bir *h* - elementni almashishdan iborat bo‘lib, *h* elementlar almashinuvi qanchalik uzoqda sodir bo‘lishi mumkinligini aniqlaydi.

Katta massivlar uchun oldindan tartiblash ancha kattaroq oraliqda boshlanishi kerak, keyin asta-sekin 1 gacha qisqaradi.

Masalan, 1000 ta elementdan iborat massivda 364 - tartiblash, keyin 121 - tartiblash , 40 - tartiblash, 13 - tartiblash, 4 - tartiblash va nihoyat 1 - tartiblash amalga oshiriladi. Oraliqni hosil qilish uchun foydalaniladigan sonlar ketma - ketligi (bu misolda 364, 121, 40, 13, 4, 1) oraliqli ketma-ketlik deyiladi.

Knut tomonidan taklif qilingan berilgan oraliqli ketma-ketlik juda mashhur hisoblanadi. Teskari shaklda, ushbu ketma - ketlik 1 dan boshlanib, u $h = 3 \times h + 1$ formula bo‘yicha (boshlang‘ich qiymati $h=1$) hosil qilinadi. Oraliqli ketma-ketliklarni qurishning boshqa usullari ham mavjud.

Knut ketma-ketligidan foydalangan holda Shell tartiblash algoritmini ko‘rib chiqamiz.

Tartiblash algoritmida dastlabki oraliq birinchi navbatda qisqa siklda hisoblanadi. Dastlab rekurent formula asosida *h* ning massiv o‘lchamidan oshmaydigan maksimal qiymati topiladi.

Keyin tartiblash usulining tashqi siklining har bir itaratsiyasida oraliq yuqoridagiga teskari formula bo‘yicha kamayadi:

$h = (h–1)/3$

Teskari formuladan foydalanib, bizning misolimizda teskari ketma-ketlik hosil bo‘ladi, bular 364, 121, 40, 13, 4, 1. Bu raqamlarning har biri 364 dan boshlab *n*-massivni tartiblash uchun ishlatiladi. 1-tartibni bajarish algoritmni tugatadi.

Shell tartiblash, qiymatlarni belgilangan joyga tezda ko‘chirishi sababli kiritish orqali tartiblashning samaradorligini oshiradi.

Amalga oshirish

```
void ShellSort(int []A)
{
  int i, j, v;
  int h = 1;
  int n = A.Length;
  while (h <= n / 3) h = h * 3 + 1;
  while (h > 0)
  {
    for (i = h; i < n; i++)
    {
      v = A[i]; j = i;
      while (j > h - 1 && A[j - h] >= v )
      {
        A[j] = A[j - h]; j -= h;
      }
      A[j] = v;
    }
    h /= 3;
  }
}
```

Shell tartiblash tahlil

Shell tartiblash o‘rta o‘lchamdagi ro‘yxatlar uchun samarali. Katta hajmli ro‘yxatlar uchun ushbu algoritm eng yaxshi tanlov emas. Bu barcha O(n^2) murakkablikka ega tartiblash algoritmlari ichida eng tezkori hisoblanadi.

Shell tartiblashning kamchiligi shundaki, u murakkab algoritm bo‘lib, birlashtirish orqali tartiblash, uyum tartiblash va tezkor tartiblash kabi samarali emas. Shell tartiblash yuqoridagi turlarga qaraganda ancha sekinroq, lekin bu nisbatan oddiy algoritm bo‘lib, tezlik muammo bo‘lmasa, elementlari soni 5000 dan ortiq bo‘lmagan ro‘yxatni tartiblash uchun yaxshi tanlov qiladi. Ushbu tartiblash qayta tartiblash uchun ham yaxshi tanlovdir. Ya'ni, tartiblangan massivga o‘zgartirishlar kiritilganda va qayta tartiblash bajarilganda samaralidir.

Shell tartiblashning bajarilish vaqti o‘sish ketma-ketligini tanlashga bog‘liq.

Shell tartiblashning murakkabligi

- Eng yomon holatning murakkabligi bo‘shliqlar ketma-ketligiga bog‘liq. Eng mashxuri: O($nlog_2n$).
- Eng yaxshi holatdagi murakkabligi: O(n);
- O‘rtacha murakkabligi o‘tkazib yuborish ketma-ketligiga bog‘liq va taxminan: O($n^{3/2}$);
- Eng yomon holatdagi xotira sarfi: O(n)

10.2.5. Birlashtirish orqali tartiblash

Birlashtirish orqali tartiblash algoritmi oldingi boblarda keltirilgan algoritmlarga qaraganda, hech bo‘lmaganda tezlik jihatidan ancha samaralidir. Pufakchali tartiblash, kiritish orqali tartiblash va tanlash orqali tartiblash O(n^2) vaqtni olsa, birlashtirish orqali tartiblash O(n*$logn$) vaqtni oladi, bu esa tartiblash jarayonini sezilarli darajada tezlashtiradi.

Misol uchun, agar n (tartiblash uchun obektlar soni) 10 000 bo‘lsa, n^2 = 100 000 000, $n \times \log n$ esa 40 000 teng bo‘ladi. Bundan tashqari, birlashtirishni amalga oshirish nisbatan oson. Agar ushbu sondagi obyektlarni tartiblash uchun kiritish orqali tartiblash 28 soat talab qilsa, birlashtirish orqali tartiblash 40 sekund vaqt talab qiladi. Bundan tashqari, birlashtirish orqali tartiblashni amalga oshirish nisbatan oson. Kontseptual

darajada u Tezkor tartiblash va Shell tartiblash algoritmlariga qaraganda oddiyroq.

Birlashtirish orqali tartiblash algoritmi “bo‘l va hukmronlik qil” tamoyiliga asoslanadi, ya’ni dastlabki masala ikkita kichik qism masalaga bo‘linadi va har bir kichik masala mustaqil yechiladi, so‘ngra natijalar dastlabki masalaga qaytariladi.

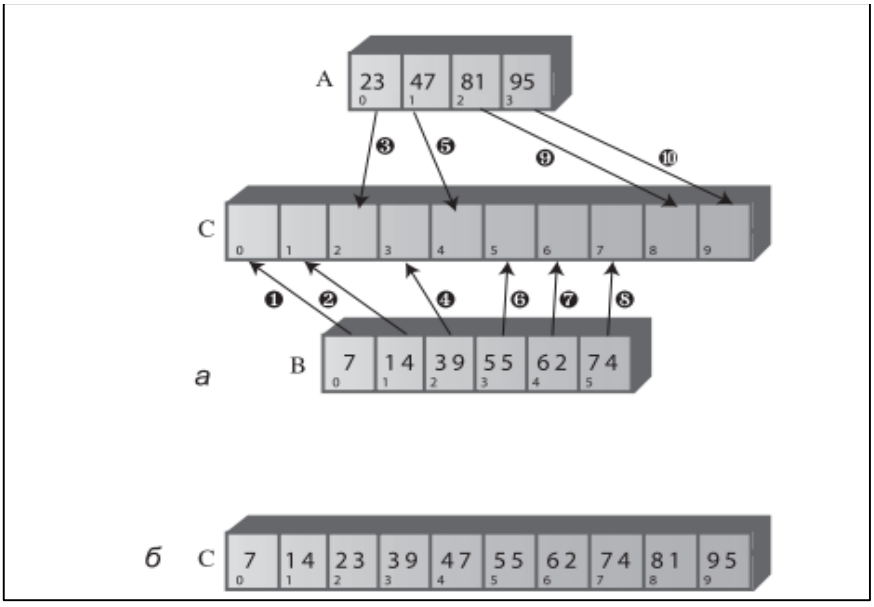
Birlashtirish orqali tartiblashning kamchiligi shundaki, u xotirada tartiblanayotgan massivning o‘lchamiga teng bo‘lgan qo‘shimcha massivni ajratishni talab qiladi. Agar dastlabki massivni xotiraga sig‘dirish qiyin bo‘lsa, unda birlashtirish orqali tartiblash algoritmidan foydalanib bo‘lmaydi, lekin yetarli bo‘sh joy bo‘lsa, bu algoritm juda samarali.

To‘g‘ridan-to‘g‘ri birlashtirish orqali tartiblashga o‘tishdan oldin, ikkita tartiblangan massivni birlashtirish algoritmini ko‘rib chiqaylik.

Ikki tartiblangan massivni birlashtirish

Birlashtirish orqali tartiblash algoritmining markaziy qismi avvaldan tartiblangan ikkita massivni birlashtirishdir. Ikki tartiblangan *A* va *B* massivlarini birlashtirish bilan *C* massiv yaratiladi. *C* massiv *A* va *B* ning barcha elementlarini o‘z ichiga olgan va tartibni saqlaydi. Dastlab, biz birlashtirish jarayonining o‘zini ko‘rib chiqamiz, keyin esa tartiblashda qanday ishlatilishini aniqlaymiz.

Faraz qilamiz, bizda ikkita oldindan tartiblangan massiv bor (bir xil o‘lchamda bo‘lishi shart emas). Aytaylik, *A* massivida 4 ta element, *B* massivda 6 ta element bo‘lib, massivlar *C* massiviga birlashtirilsin. *C* massiv dastlab 10 ta bo‘sh katakchadan iborat bo‘ladi.



Ikki massivni birlashtirish: a - birlashtirishdan oldin; b - birlashtirgandan so‘ng.

Rasmda doiralardagi raqamlar elementlarning A va B dan C ga ko‘chirilish tartibini ko‘rsatadi. 8-bosqichdan keyin B massivida hech qanday element yo‘qligi sababli, qo‘shimcha taqqoslashlar kerak emas; qolgan barcha elementlar oddiygina A dan C ga ko‘chiriladi.

Endi ushbu algoritmni A massiv berilganda qanday qo‘llashni ko‘rib chiqamiz. Buning uchun algoritmni biroz o‘zgartiramiz, A massivi yarmidan ikki qismga ajratamiz, ularning har birini mustaqil tartiblaymiz, so‘ngra bu qismlarni birlashtirishimiz kerak bo‘ladi va buning uchun bizga qo‘shimcha massiv zarur. Quyida birlashtirish funksiyasi berilgan:

```
// Massivni ikki qismini birlashtirish
void Merge(int []A, int []tmp, int left, int mid, int
right)
{
  int i, left_end, size, pos;
  left_end = mid-1; pos = left; size = right - left + 1;
  while ((left <= left_end) && (mid <= right))
  {
    if (A[left] <= A[mid]) { tmp[pos++] = A[left++]; }
    else { tmp[pos++] = A[mid++]; }
  }
  while (left <= left_end) { tmp[pos++] = A[left++]; }
  while (mid <= right) { tmp[pos++] = A[mid++]; }
  for (i = 0; i < size; i++) A[right] = tmp[right--];
}
```

Merge() funksiyasi uchta *while* sikldan iborat. Birinchi sikl A massivning chap va o‘ng yarmida takrorlanadi, elementlarni taqqoslaydi va ulardan kichikroqlarini *tmp* massiviga ko‘chiradi.

Ikkinchi sikl chap massivdagi barcha elementlar *tmp* massivga olinmagan bo‘lsa ishlaydi. Bunday holda, chap yarmining qolgan elementlari *tmp* ga ko‘chiriladi.

Uchinchi massiv xuddi ikkinchi massiv singari ishlaydi ya’ni o‘ng yarmining barcha elementlari *tmp*ga olinmagan bo‘lsa, qolgan elementlar *tmp* ga ko‘chiriladi.

Birlashtirish orqali tartiblash

Birlashtirib orqali tartiblash ortidagi g‘oya massivni ikkita qismga bo‘lish, har bir yarmini tartiblash va keyin *Merge*() funksiyasidan ikki yarmini bitta tartiblangan massivga birlashtirishdan iborat. Har bir yarmi ikki chorakka bo‘linadi, har bir chorak alohida tartiblanadi, shundan so‘ng ikki chorak tartiblangan yarmiga birlashtiriladi.

Xuddi shunday, 8 qismdan iborat har bir juftlik tartiblangan chorakka birlashtiriladi va hokazo. Massiv toki bitta elementdan iborat qism massiv olinmaguncha qayta - qayta bo‘linaveradi. Bu rekursiyaning asosiy cheklovi bo‘lib, bir elementdan iborat massiv allaqachon tartiblangan deb hisoblanadi.

Birlashtirish orqali tartiblash rekursiv funksiya yordamida juda samarali amalga oshiriladi:

```
void MergeSort(int []A, int []tmp, int left, int right)
{
  int mid;
  if (right > left)
  {
    mid = (right + left) / 2;
    MergeSort(A, tmp, left, mid);
    MergeSort(A, tmp, mid + 1, right);
    Merge(A, tmp, left, mid+1, right);
  }
}
```

Tahlil

Yuqoridagi dasturdan ko‘rinib turibdiki, umuman olganda, rekursiv usulga har bir chaqiruv qaysidir parametrning kamayishiga olib keladi va har safar funksiya qaytganida bu parametr yana ortadi. *MergeSort*() da, har bir rekursiv chaqiruvda diapazon ikkiga bo‘linadi va har bir qaytishda ikkita yarmi bitta kattaroq diapazonga birlashtiriladi.

MergeSort() funksiyasi har biri bitta elementdan iborat ikkita massivni topgandan keyin qaytsa, ularni ikkita elementdan iborat tartiblangan massivga birlashtiradi. Bundan tashqari, hosil bo‘lgan 2 elementli massivlarning har bir jufti 4 ta elementdan iborat massivga birlashtiriladi.

Jarayon butun massiv tartiblangunga qadar doimiy o‘lchamdagi massivlar bilan davom etadi.

Algoritmning murakkabligini baholash:

- Eng yomon holatdagi murakkabligi: O(*n*log*n*);
- Eng yaxshi holatdagi murakkabligi: O(*n*log*n*);
- O‘rtacha murakkabligi: O(*n*log*n*);
- Eng yomon holatdagi xotira sarfi: O(*n*).

10.2.6. Tezkor tartiblash (Quicksort)

n ta elementdan iborat kirish massivini tartiblash uchun tezkor tartiblash algoritmining eng yomon holatdagi murakkabligi O(n^2) ga teng. Eng yomon holatda juda sekin bo‘lishiga qaramay, bu algoritm amalda ko‘pincha eng yaxshisidir, chunki u o‘rtacha hisobda juda samarali: uning kutilayotgan ishlash vaqti O(*n*lg*n*) ni tashkil qiladi va baholashdagi yashiringan doimiy koeffitsientlar juda kichikdir. Shuningdek, uning afzalligi shundaki, u joyida tartiblashni amalga oshiradi va virtual xotira muhitida ham yaxshi ishlaydi.

Tezkor tartiblash algoritmining tavsifi

Tezkor tartiblash xuddi birlashtirish orqali tartiblash kabi, “bo‘l va hukmronlik qil” tamoyilidan foydalanadi. Uch bosqichli tartiblash jarayoni quyida tasvirlangan:

Bo‘lish : A[l..r] massivi shunday ikkita (ehtimol bo‘sh) A[l. . q-1] va A[q+1..r] qismlarga ajratiladiki bu yerda, A[l . . q-1] qism massivning har bir elemetli A[q] qiymatdan kichik, A[q] esa o‘z navbatida A[q+1..r] qism massivdagi har bir elementdan kichik yoki tengdir. q indeksi bo‘linish jarayoni davomida hisoblanadi.

Hukmronlik qilish: A[l . . q-1] va A[q+1..r] massiv ostilari tezkor tartiblash algoritmini rekursiv ravishda chaqirish bilan tartiblanadi.

Kombinatsiya: Massiv ostlari tartiblanganligi sababli ularni bilrashtirish talab qilinmaydi: butun A[1..r] massivi tartiblangan holatga kelgan bo‘ladi.

Quyidagi protsedura tezkor tartiblashni amalga oshiradi:

```
void QuickSort(int []A, int l, int r)
{
  if(r > l) {
    int q = Partition(A, l, r);
```

```
    QuickSort(A, l, q - 1);
    QuickSort(A, q + 1, r);
  }
}
```

Massivni qismlarga ajratish.

Ko‘rib chiqilayotgan tartiblash algoritmining muhim qismi *Partition()* funksiyasi bo‘lib, u A[l..r] massiv osti elementlarining tartibini qo‘shimcha xotirani jalb qilmasdan o‘zgartiradi.

Bo‘lish algoritmi quyidagicha:

Dastlab, massiv osti oxirgi elementi A[r] tayanch element sifatida tanlanadi va u aniq to‘g‘ri o‘ringa joylashtiriladigan element hisoblanadi.

Keyinchalik, u massivning chap uchidan tayanch element qiymatidan katta element topilgunga qadar qarab chiqiladi, so‘ngra massivning o‘ng uchidan tayanch element qiymatidan kichik element topilguncha qarab chiqadi. Undan keyin to‘xtashga olib kelgan shu ikkita element o‘rinlari almashtiriladi, chunki ular tayanchga nisbatan o‘z joylarida emas. Jarayon chap tomonidagi barcha elementlar tayanchdan kichik yoki teng, o‘ngdagilari esa tayanchdan katta bo‘lguncha davom etadi.

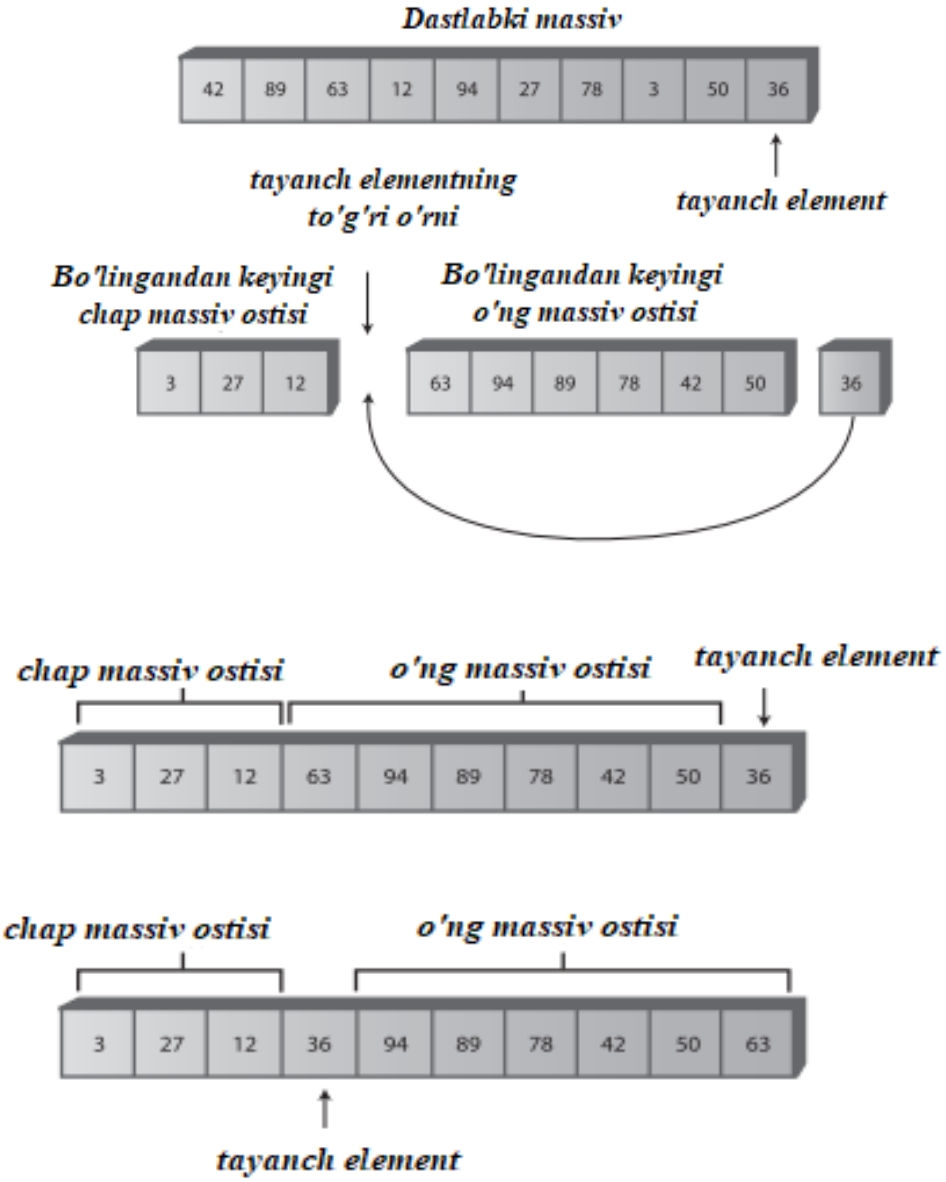
Bo‘linishni yakunlash:

Jarayonni yakunlash uchun A[r] elementi o‘ng tomonning eng chap elementi bilan almashtirilishi kerak.

Ushbu algoritmni amalga oshirish dasturi:

```
int Partition(int [] A, int l, int r)
{
  int i = l - 1, j = r, v = A[r];
  int t;
  for (; ; )
  {
    while (A[++i] < v) ;
    while (v < A[--j]) if (j == l) break;
    if (i >= j) break;
    t = A[i]; A[i] = A[j]; A[j]=t;
  }
  t = A[i]; A[i] = A[r]; A[r] = t;
  return i; }
```

Rasmda funksiya qanday ishlashi ko‘rsatilgan:

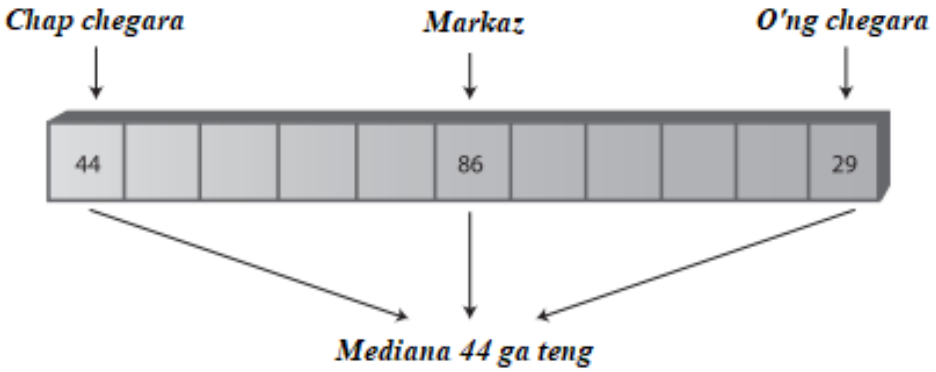


Medianani uch nuqta bilan aniqlash

Tayanch qiymatini tanlash uchun ko‘plab sxemalar mavjud. Bunday sxema oddiy bo‘lishi kerak, bundan tashqari eng katta yoki eng kichik qiymatni tanlashdan katta ehtimol bilan qochish kerak. Elementni tasodifiy tanlash oddiy, lekin har doim ham yaxshi natijaga olib kelmaydi. Albatta, siz barcha elementlarni tahlil qilishingiz va medianani hisoblashingiz mumkin. Tanlov ideal bo‘ladi, lekin jarayonning o‘zini amaliy deb hisoblash qiyin - chunki u tartiblashdan ko‘ra ko‘proq vaqt talab etadi.

Medianani tanlashning eng murosali yo‘li massivdagi birinchi, oxirgi va o‘rta elementlar orqali aniqlash hisoblanadi. Olingan natija tayanch

qiymat sifatida ishlatiladi. Birinchi, oxirgi va o‘rta elementlarning medianasini tanlash uchta nuqtadan medianani aniqlash deb ataladi.



Albatta, uchta nuqta bo‘yicha medianani aniqlash massivning barcha elementlari orasidan aniqlashdan ancha tezroq. Biroq, berilganlar to‘g‘ri yoki teskari tartibda tartiblangan bo‘lsa, u eng katta yoki eng kichik elementni tanlashdan muvaffaqiyatli qochadi. Bu eng yomon holatning yuzaga kelish ehtimolini kamaytiradi. Ehtimol, ba'zi patologik holatlarda bu sxema ham yomon ishlaydi, ammo oddiy vaziyatda bu sizga tayanch nuqta qiymatini tez va samarali tanlash imkonini beradi va algoritmning o‘rtacha bajarilish vaqtini 5-10% ga qisqartiradi.

Tanlangan elementlarni tartiblash mumkin va keyin elementlarning o‘rtacha qiymatini A[r-1] bilan almashish va keyin A[l+2 ... r-2] bo‘yicha bo‘lish algoritmini bajarish mumkin.

Bu takomillashtirishga uchta nuqta orqali mediana usuli deb ataladi.

Kichik o‘lchamli massiv ostilari.

Rekursiv dastur kichik o‘lchamli massiv ostlari uchun o‘zini qayta-qayta chaqiradi. Bunga yo‘l qo‘ymaslik uchun kichik o‘lchamli massiv ostlari uchun oddiy tartiblash algoritmlaridan birini qo‘llashingiz mumkin, masalan, kiritish orqali tartiblashni:

```
if(r-l<=M) InsertionSort(A,r-l+1);
```

Bu yerda M - bu qandaydir parametr, uning aniq qiymati amalga oshirishga bog‘liq. Odatda qiymat 5 dan 20 gacha bo‘lgan oraliqda olinadi.

Kichik o‘lchamli massiv ostlarini boshqarishning biroz soddaroq usuli bu rekursiyadan quyidagi shaklda qaytishni amalga oshirishdir:

```
if(rl<=M) return;
```

Ya'ni, bo‘linishda kichik o‘lchamli massiv ostlari uchun shunchaki inkor etish kerak bo‘ladi. Natijada, bo‘lish oxirida deyarli tartiblangan massiv olinadi. Kichik o‘lchamli massiv ostlarini tartiblash uchun kiritish orqali tartiblashni qo‘llash mumkin, bu deyarli tartiblangan massivlarda juda yaxshi ishlaydi.

Uchta nuqtadan median qiymatni topish usuli hamda kichik o‘lchamli massiv ostlari qirqishni birlashtirib, rekursiv amalga oshirish bilan solishtirganda tezkor tartiblash vaqtini 20-25% ga yaxshilash mumkin.

Takomillashtirilgan tezkor tartiblash quyidagi ko‘rinishga ega:

```
static const int M=10;
void median3(int [] A, int l, int r)
{
  int q = (l + r) / 2;
  int t;
  if (A[l] < A[q]) { t = A[l]; A[l] = A[q]; A[q] = t; };
  if (A[l] < A[r]) { t = A[l]; A[l] = A[r]; A[r] = t; };
  if (A[q] < A[r]) { t = A[q]; A[q] = A[r]; A[r] = t; };
  t = A[q]; A[q] = A[r - 1];A[r-1]=t;
}
void QuickSortM(int []A, int l, int r)
{
  if(r - l <= M) return;
  median3(A, l, r);
  int q = Partition(A, l + 1, r - 1);
  QuickSortM(A, l, q - 1);
  QuickSortM(A, q + 1, r);
}
void SQuickSort(int []A, int l, int r)
{
  QuickSortM(A, l, r);
  InsertionSort(A, r - l + 1);
}
```

Algoritm murakkabligini baholash:

- Eng yomon holatdagi murakkabligi: O(n^2);

- Eng yaxshi holatdagi murakkabligi: O(*n*log*n*);
- O‘rtacha murakkabligi: O(*n*log*n*);
- Eng yomon holatdagi xotira sarfi: O(1).

10.2.7. Chiziqli tartiblash algoritmlari

Biz taqqoslash asosidagi bir qancha tartiblash algoritmlari bilan tanishib chiqdik. Ular orasida eng yaxshilari O(*n*log*n*) murakkabligiga ega. Ushbu ma’ruzada chiziqli murakkablikka ega bo‘lgan boshqa turdagi algoritmlarni ko‘rib chiqamiz. Biroq, ular kiritilgan berilganlarga ma'lum cheklovlar qo‘yadi.

Chiziqli tartiblash algoritmlariga quyidagilar misol bo‘la oladi:

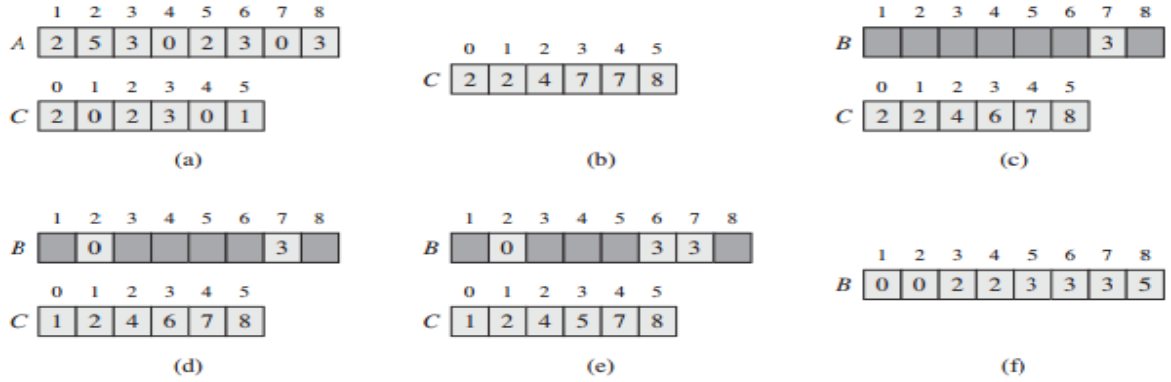
• Hisoblash orqali tartiblash (*CountingSort*);

• Razryadali tartibalash (*RadixSort*);

• Cho‘ntak tartiblash (*BucketSort*).

10.2.7.1. Hisoblash orqali tartiblash

Elementlari qiymatlari 0 dan *M* gacha bo‘lgan *N* o‘lchamli butun sonlar massivni saralash zarur bo‘lsin. Tartiblash usuli quyidagicha: ma'lum bir qiymatga ega bo‘lgan elementlarning sonini hisoblash va keyin tartiblangan massivdan ikkinchi o‘tish paytida tegishli o‘rinlarga o‘tish uchun hisoblagichlardan foydalanish. Dastlab, har bir qiymat uchun elementlar soni hisoblab chiqiladi, so‘ngra har bir bunday qiymatdan kichik yoki unga teng bo‘lgan elementlar soniga teng bo‘lgan qismiy summalar hisoblanadi. Bundan tashqari, bu sonlar qiymatlarni taqsimlashda indeks sifatida ishlatiladi. Har bir qiymat uchun unga bog‘langan hisoblagichning qiymati bir xil qiymatga ega bo‘lgan elementlar blokining oxirini ko‘rsatadigan indeks sifatida ko‘rib chiqiladi. Ushbu indeks qiymatni qo‘shimcha massivga joylashtirishda ishlatiladi, shundan so‘ng u o‘ngdagi keyingi o‘ringa o‘tadi. Ta'riflangan jarayon rasmda ko‘rsatilgan:



Dastlab, har bir qiymat uchun massivdagi ushbu qiymatga ega bo‘lgan elementlar soni hisoblanadi. Ushbu misolda 0 qiymati 2 ta, 1 qiymati 0 ta, 2 qiymati 2 ta, 3 qiymati 3 ta, 4 qiymati 0 ta va 5 qiymati 1 ta uchradi. Keyin qismiy yig‘indilar hisoblab chiqiladi, ya'ni, qiymatlari berilgan qiymatdan katta bo‘lmagan elementlar soni: 0 dan katta bo‘lmagan 2 ta element, 1 dan katta bo‘lmagan 2 ta element, 2 dan katta bo‘lmagan 4 ta element, 3 dan katta bo‘lmagan 7 ta element, 4 dan katta bo‘lmagan 7 ta element va 5 dan katta bo‘lmagan 8 ta element mavjud. So‘ngra ushbu yig‘indi qo‘shimcha massivdagi dastlabki massivdan teskari tartibda yoziladigan sonning o‘rnini aniqlashda ishlatiladi, 7 indeksda 3 yoziladi, shundan so‘ng 3 uchun ko‘rsatgich bittaga kamayadi, so‘ngra dastlabki massivdagi keyingi element 0 2-o‘ringa yoziladi va hokazo.

Dasturni C# da amalga oshirish:

```
void CountSort(int []A, int m)
{
  int i, j,n=A.Length;
  int []C = new int[m];
  int []B = new int[n];
  for (j = 0; j < m; j++) C[j] = 0;
  for (i = 0; i < n; i++) C[A[i]]++;
  for (j = 1; j < m; j++) C[j] += C[j - 1];
  for (i = n - 1; i >= 0; i--) B[C[A[i]]-- - 1] = A[i];
  for (i = 0; i < n; i++) A[i] = B[i];
}
```

Algoritm murakkabligini baholash:

- Umumiy bajarish vaqti: $2O(M)+3O(n)\approx O(n)$;

- Eng yomon holatdagi murakkabligi: O(*n*);
- Eng yaxshi holatdagi murakkabligi: O(*n*);
- O‘rtacha murakkabligi: O(*n*);
- Eng yomon holatdagi fazoning murakkabligi : O(*n*).

10.2.7.2. Razryadli tartiblash

Razryadli tartiblash butun sonning har bir raqami bo‘yicha barqaror tartiblashga asoslanadi, tartiblashning ikki turi mavjud, katta razryaddan boshlab tartiblash va kichik razryaddan boshlab tartiblash.

Odatda kichik razryad bo‘yicha tartiblash qo‘llaniladi.

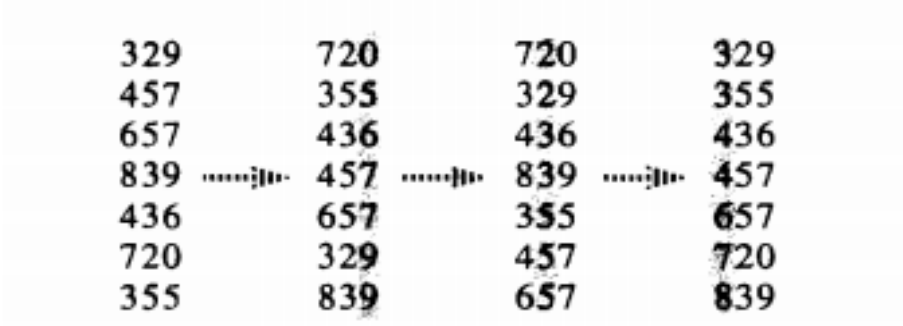
Sondagi ma'lum bir razryaddagi raqamini olish uchun ma'lum bir dasturlash tiliga va ma'lum bir uskunaga moslashtirilishi mumkin bo‘lgan turli usullardan foydalanish mumkin, masalan, bitlar ustida amallardan foydalanishingiz mumkin. *d* razryaddagi raqamni aniqlashning eng oson usuli - $x=(A/R^d)\%R$

```
int digit(int A,int d,int R) { return A/pow(R,d)%R;}
```

Saralash eng kichik razryaddan boshlanadi. Misol uchun, siz hisoblash orqali tartiblash algoritmini qo‘llashingiz mumkin, bu algoritm barqaror hisoblanadi.

Keyin olingan massiv ikkinchi razryad bo‘yicha tartiblanadi va hokazo.

Tartiblash misoli rasmda ko‘rsatilgan:



O‘nlik sonlar uchun algoritmni amalga oshirish.

```
void RadixSort(int [] A, int M)
{
  int i, j, k, d,n=A.Length;
  int[] C = new int[10];
  int[] B = new int[n];
```

```
  for (k = 1, d = 0; d < M; d++, k *= 10)
  {
    for (j = 0; j < 10; j++) C[j] = 0;
    for (i = 0; i < n; i++) C[A[i] / k % 10]++;
    for (j = 1; j < 10; j++) C[j] += C[j - 1];
    for (i = n - 1; i >= 0; i--) B[C[A[i] / k % 10]-- -
1] = A[i];
    for (i = 0; i < n; i++) A[i] = B[i];
  }
}
```

Algoritmning murakkabligini baholash:

- Umumiy bajarilish vaqti: (2O(10)+3O(*n*))O(*M*) ≈ O(n);
- Eng yomon holatdagi murakkabligi: O(*n*);
- Eng yaxshi holatdagi murakkabligi: O(*n*);
- O‘rtacha murakkabligi: O(*n*);
- Eng yomon fazoviy murakkablik: O(*n*).

10.2.7.3. Cho‘ntak (Bucket) tartiblash

Hisoblash orqali tartiblash kabi, Bucket tartiblash ham uning murakkabligini yaxshilash uchun kiritishga cheklovlar qo‘yadi. Saralanadigan berilganlar tekis taqsimlangan bo‘lishi kerak.

Boshqacha qilib aytganda, agar kiruvchi berilganlar fiksirlangan to‘plamdan tanlansa, Cho‘ntak tartiblash yaxshi ishlaydi. Cho‘ntak tartiblash aslida hisoblash orqali tartiblashning umumlashmasidir.

Cho‘ntak tartiblashning o‘ziga xos farqi shundaki, u bir nechta qiymatlar bitta cho‘ntagiga tushishi uchun kirish massivining qiymatlarini ajratish uchun maxsus funksiyadan foydalanadi. Shuning uchun har bir cho‘ntak joylashuv bo‘yicha saralash kabi kengaytiriladigan ro‘yxat bo‘lishi kerak.

Cho‘ntak tartiblash dasturi:

```
void BucketSort(int []A)
{
  int max = A.Length;
  int[][] bucket=new int[10][];
  for (int i = 0; i < 10; i++)
```

```
    bucket[i] = new int[max + 1];
  for (int i = 0; i < 10; i++) bucket[i][max] = 0;
  for (long digit = 1; digit <= 1000000; digit *= 10)
  {
    for (int i = 0; i < max; i++)
    {
      int d = (int)((long)A[i] / digit) % 10;
      bucket[d][bucket[d][max]++] = A[i];
    }
    int idx = 0;
    for (int i = 0; i < 10; i++)
    {
      for (int j = 0; j < bucket[i][max]; j++)
      {
        A[idx++] = bucket[i][j];
      }
      bucket[i][max] = 0;
    }
  }
}
```

Algoritmning murakkabligini baholash:

- Umumiy bajarilish vaqti: 6* (O(10)+11*O(n)) ≈ O(n);
- Eng yomon holatdagi murakkabligi: O(n);
- Eng yaxshi holatdagi murakkabligi: O(n);
- O‘rtacha murakkabligi: O(n);
- Eng yomon fazoning murakkabligi: O(n).

10 – bob bo‘yicha nazorat savollari

1. Tartiblash algoritmlarni tasniflash.
2. Pufakchali tartiblash qanday amalga oshiriladi?
3. Tanlash orqali tartiblash qanday amalga oshiriladi?
4. Kiritish orqali tartiblash qanday amalga oshiriladi?
5. Shell tartiblash qanday amalga oshiriladi?
6. Birlashtirish orqali tartiblash qanday amalga oshiriladi?
7. Tezkor tartiblash qanday amalga oshiriladi?
8. Chiziqli tartiblash algoritmlari qanday amalga oshiriladi?
9. Hisoblash orqali tartiblash (CountingSort) qanday amalga oshiriladi?
10. Razryadali tartibalash (RadixSort) qanday amalga oshiriladi?
11. Cho‘ntak tartiblash (*BucketSort*) qanday amalga oshiriladi?

11. Qidirish

Informatika fanida qidirish - bu elementlar to‘plamidan berilgan xususiyatlarga ega elementni topish jarayonidir. Elementlar berilganlar bazasida yozuvlar, massivdagi oddiy berilganlar elementlari, fayllardagi matnlar, daraxtlardagi tugunlar, graflardagi uchlar va qirralar va boshqalar sifatida saqlanishi mumkin.

Qidirish informatika fanining asosiy algoritmlaridan biridir. Biz bilamizki, bugungi kunda kompyuter olamida juda ko‘p ma'lumotlar mavjud. Ushbu ma'lumotni to‘g‘ri olish uchun bizga juda samarali qidirish algoritmi kerak.

Qidirish jarayonini yaxshilaydigan berilganlarni tartibga solishning ma'lum usullari mavjud.

Bu shuni anglatadiki, agar biz berilganlarni to‘g‘ri tartibda saqlasak, kerakli elementni topish oson bo‘ladi. Saralash - elementlarni tartiblash usullaridan biri hisoblanadi.

Qidirish turlari

Quyida biz ushbu ma'ruzada muhokama qiladigan qidirish turlari keltirilgan:

• Tartiblanmagan chiziqli qidirish;

• Saralangan/tartiblangan chiziqli qidirish;

• Ikkilik qidirish;

• Interpolyatsiyali qidirish;

• Ikkilik (binar) qidirish daraxtlari (binar qidirish daraxtlari bilan ishlaydi);

• Belgilar jadvallari va xeshlash;

• Satrlarni qidirish algoritmlari: satr osti, ternar qidirish va qo‘shimchalar daraxtlari.

11.1. Tartiblanmagan chiziqli qidirish

Faraz qilaylik, elementlarning tartibi noma'lum bo‘lgan massiv berilgan. Bu massiv elementlari tartiblanmaganligini bildiradi. Bunday hollarda, elementni topish uchun biz massivning har bir elementini qarab chiqishimiz va biz qidirayotgan element berilgan ro‘yxatda bor yoki

yo‘qligini aniqlashimiz kerak. Butun sonlar massividan elementni qidirish funksiyasi quyidagi ko‘rinishga ega:

```
int NoOrderSearch(int[] A,  int data)
{
  for (int i = 0; i < A.Length; i++)
    if (A[i] == data) return i;
  return -1;
}
```

Butun massivni qarab chiqishda (elementni qidirish uchun) algoritmning murakkabligi (mehnat unumdorligi) eng yomon holatda O(n), eng yaxshi holatda O(1), o‘rtacha O(n/2).

11.2. Tartiblangan chiziqli qidirish

Agar massivning elementlari oldindan tartiblangan bo‘lsa, ko‘p hollarda element berilgan massivda bor yoki yo‘qligini bilish uchun butun massivni qarab chiqish shart emas. Quyidagi algoritmda siz istalgan nuqtada, agar A[i] dagi qiymat qidirish uchun berilgan qiymatdan katta bo‘lsa, qolgan massivni qidirmasdan shunchaki -1 ni qaytaramiz:

```
int OrderSearch(int[] A,  int data)
{
  for (int i = 0; i < A.Length; i++)
  {
    if (A[i] > data) return -1;
    if (A[i] == data) return i;
  }
  return -1;
}
```

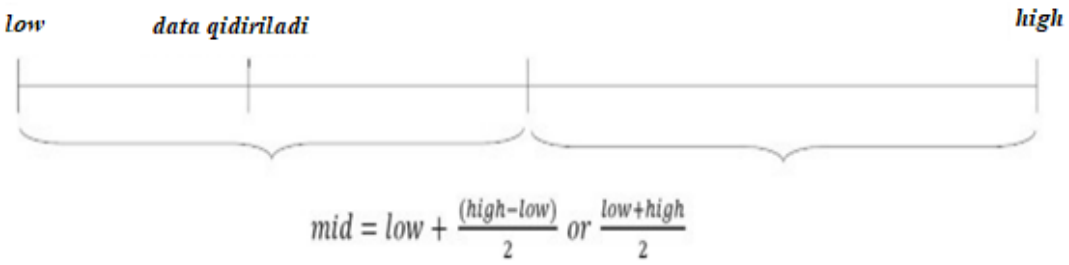
Bu algoritmning vaqt murakkabligi O(*n*), chunki eng yomon holatda biz butun massivni qarab chiqishimiz kerak. Ammo o‘rtacha hisobda, algoritmning murakkabligi massivning tartiblanganligi sababli kamayadi.

Eslatma. Yuqoridagi algoritm uchun biz siklni kattaroq miqdorga oshirish orqali uni yaxshilashimiz mumkin (aytaylik, 2 ga). Bu tartiblangan ro‘yxatni qidirish uchun taqqoslashlar sonini kamaytiradi.

11.3. Binar qidirish

Lug‘atdagi so‘zni qidirish masalasini ko‘rib chiqaylik. Odatda biz to‘g‘ridan-to‘g‘ri taxminiy sahifaga o‘tamiz (aytaylik, o‘rta sahifa) va qidirishimizni shu nuqtadan boshlaymiz. Agar biz izlayotgan nom bir xil bo‘lsa (shu sahifada joylashgan bo‘lsa), qidirish tugaydi. Agar sahifa tanlangan sahifalardan oldin bo‘lsa, unda biz birinchi yarmi uchun xuddi shunday jarayonni bajaramiz; aks holda, xuddi shu jarayonni ikkinchi yarmiga qo‘llaymiz.

Binar qidirish shunga o‘xshash tarzda ishlaydi. Bunday strategiyadan foydalanadigan algoritm binar (ikkilik) qidirish algoritmi deb ataladi.




```
// Iterativ binar qidirish algoritmi
int BinarySearchImerativ(int []A, int data)
{
  int low = 0;
  int high = A.Length - 1;
  while (low <= high)
  {
    int mid = (low + high) / 2;
    if (A[mid] == data) return mid;
    if (A[mid] < data) low = mid + 1;
    else high = mid - 1;
  }
  return -1;
}
// Rekursiv binar qidirish algoritmi
int BinarySearchRecursive (int [] A, int low, int high,
int data)
```

```
{
  int mid = (low + high) / 2;
  if (low > high) return -1;
  if (A[mid] == data) return mid;
  if (A[mid] < data)
    return BinarySearchRecursive(A, mid + 1, high,
data);
  else
    return BinarySearchRecursive (A, low, mid - 1,
data);
}
```

“Bo‘l va hukmronlik qil” usuliga ko‘ra, T(n) = O($log_2 n$) ni olamiz. Shunday qilib, algoritmning vaqt bo‘yicha murakkabligi: O(log$n$).

Iterativ amalga oshirish uchun fazoviy murakkablik: O(1).

11.4. Interpolyatsion qidirish

Albatta, binar qidirish log(n) o‘rtacha vaqt murakkabligi bilan qidirish uchun ajoyib algoritmdir.

Ushbu algoritm har doim qolgan qidirish joyining o‘rtasini tanlaydi, o‘rta o‘rinda topilgan qiymatni kerakli qiymat bilan taqqoslashga qarab u yoki boshqa yarmini tashlab yuboradi. Qolgan qidirish maydoni to‘liq yarmiga kamayadi. Biroq, boshqa qidirish usullari ham mavjud.

Matematikada interpolyatsiya - bu ma'lum berilgan nuqtalarning chekli to‘plamlari doirasida yangi nuqtalarini qurish jarayonidir. Informatika fanida ko‘pincha erkli o‘zgaruvchilar qiymatlarining cheklangan to‘plamida funksiya qiymatlari berilgan bo‘ladi.

Ko‘pincha erkli o‘zgaruvchining qiymatlari oralig‘ida ushbu funksiyaning qiymatini interpolyatsiya qilish (ya'ni, taxmin qilish) talab qilinadi. Yoki aksincha, berilgan qiymatli funksiya uchun erkli o‘zgaruvchining qiymatini aniqlash talab qilinadi.

Interpolyatsiya usulining juda ko‘p turlari mavjud bo‘lib, eng oddiy usullardan biri chiziqli interpolyatsiyadir.

Aytaylik, quyidagi qiymatli *f(x)* funksiyasi mavjud:

f(1)=10; f(2)=20;…f(10)=100. Aytaylik, agar *f(x)*=55 bo‘lsa, *x* ning qiymati qayerda ekanligini aniqlashimiz kerak bo‘lsin. 55 qiymati 50 va 60

larning o‘rtasida joylashgan bo‘lganligi uchun f(5) = 50 va f(6) = 60 oralig‘ida f(55) ni olish mantiqan to‘g‘ri keladi, bu esa 55 ni beradi.

Chiziqli interpolyatsiya ikkita berilganlar nuqtasini oladi, deylik $(x_1; y_1)$ va (x_2, y_2) va x nuqtadagi interpolyant y quyidagicha aniqlanadi:

$$y = y_1 + (y_2 - y_1)\frac{(x - x_1)}{(x_2 - x_1)}$$

Xuddi shunday x nuqta y ning berilgan qiymati bilan aniqlanadi:

$$x = x_1 + (x_2 - x_1)\frac{(y - y_1)}{(y_2 - y_1)}$$

Aytaylik, quyidagi doimiy berilgan bo‘lsin:

$$K = \frac{(y - y_1)}{(y_2 - y_1)}$$

K (0<=K<=1) doimiy kerakli elementga qanchalik yaqinlashganimiz mumkinligini aniqlaydi. Binar qidirish usulida berilgan kalitning qiymatidan qat’iy nazar biz K=½ ko‘rinishida tanlaymiz. Interpolyatsiya qidirishida berilgan kalitning qiymatiga qarab nuqta tanlashimiz mumkin.

Bu algoritm biz telefon kitobidan ismni qanday qidiradigan bo‘lsak, xuddi shu tarzda qidirishni amalga oshirishga harakat qiladi. Biz, insonlar, qidirayotgan ismimiz, masalan, "Maxmud" kabi "M" harfi bilan boshlansa, telefon kitobining o‘rtasidan qidirishni boshlashimiz kerakligini oldindan bilamiz. Agar biz "Anvar" so‘zini qidirayotgan bo‘lsak, uni kitobning boshidan qidiramiz. Buning sababi shundaki, biz harflarning tartibini oldindan bilamiz, biz oraliqlarni (a-z) bilamiz va qandaydir tarzda biz intuitiv ravishda so‘zlar teng taqsimlanganligini bilamiz. Bu faktlar binar qidirish yomon tanlov bo‘lishi mumkinligini tushunish uchun yetarli. Haqiqatan ham, binar qidirish ro‘yxatni ikkita teng qism ro‘yxatlarga ajratadi, agar biz qidirilayotgan element ro‘yxatning boshida yoki oxirida ekanligini oldindan bilsak ushbu usul (binar qidirish) samarali bo‘lmaydi.

Interpolyatsion qidirish algoritmi binar qidirishni yaxshilashga harakat qiladi. Savol shundaki, bu qiymatni qanday topish mumkin? Bizning holatimizda K doimiyni quyidagi formula bo‘yicha hisoblashimiz mumkin:

$$K = \frac{(data - A[low])}{(A[high] - A[low])}$$

Ajratilgan indeks qiymati quyidagi formula bo‘yicha aniqlanadi:

$$mid = low + (high - low) * K$$

K doimiysi qidirish maydonini toraytirish uchun ishlatiladi.

Endi biz tezda kerakli qiymatga yaqinlashamiz. O‘rtacha, interpolyatsiya qidirishi taxminan log(log*n*) taqqoslashlarini amalga oshiradi (agar elementlar teng taqsimlangan bo‘lsa), bu yerda *n* - ro‘yxatdagi elementlar soni. Eng yomon holatda (masalan, kalitlarning raqamli qiymatlari eksponent ravishda oshganida), bu O(*n*) gacha taqqoslash mumkin.

Interpolyatsion qidirishda qidirilayotgan element unga yaqin elementlar orasidan qidiradi, chiziqli qidirishda esa berilgan qiymatga mos keladigan elementni qidiradi. Ushbu algoritm yaxshi natija berishi uchun berilganlarning tartiblangan to‘plami teng taqsimlangan bo‘lishi kerak.

```
// Interpolyatsiyani qidirish algoritmi
int BinarySearchImerativ(int[] A, int data)
{
  int low = 0;
  int high = A.Length - 1;
  int mid;
  while (low <= high)
  {
    mid = low + (high - low) * (data - A[low]) / (A[high]
- A[low]);
    if (A[mid]== data) return mid;
    if (A[mid]< data) low = mid + 1;
    else high = mid - 1;
  }
  return -1;
}
```

Asosiy qidirish algoritmlarini solishtirish

Amalga oshirish	**Eng yomon holat**	**O‘rtacha holat**
Tartiblanmagan massiv	*n*	*n/2*
Tartiblangan massiv (binar qidirish)	*logn*	*logn*

Tartiblanmagan ro‘yxat	*n*	*n/2*
Tartiblangan ro‘yxat	*n*	*n/2*
Binar qidirish daraxti	*n*	*logn*
Interpolyatsion qidirish	*n*	*logn(logn)*

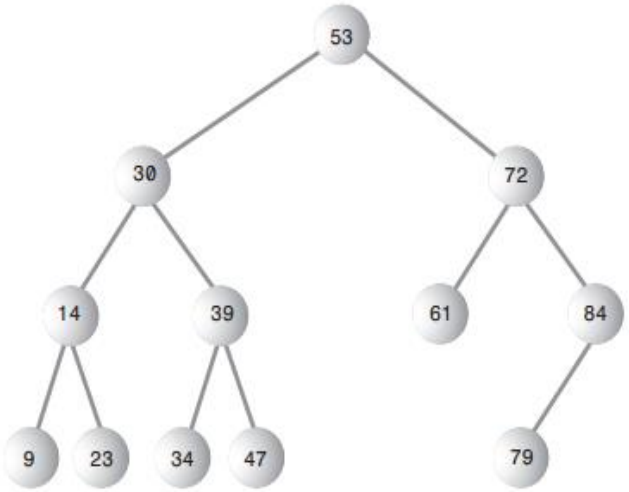
Boshqa qidirish usullarini tegishli bo‘limlarni o‘rganayotganda ko‘rib chiqamiz.

11.5. Binar qidirish daraxti

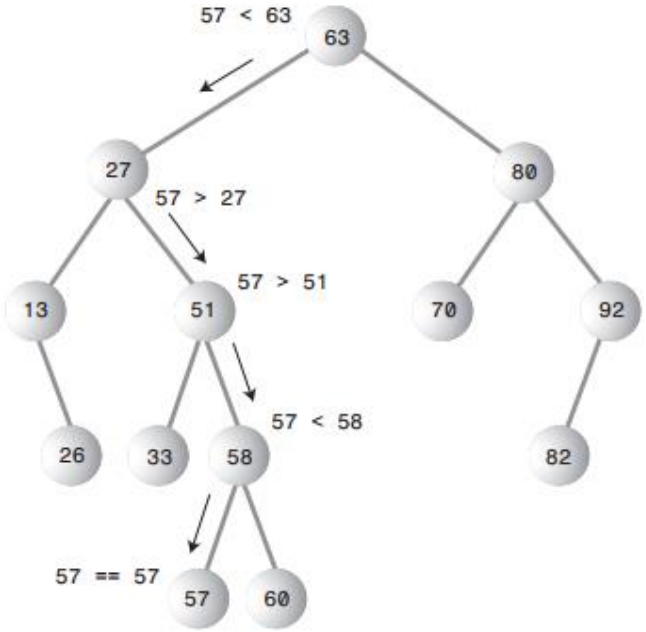
Ko‘p ishlatiladigan daraxtlardan biri binar qidirish daraxtidir.

Bunday daraxtda tugunning chap bolasining berilganlar maydonining (kalit) qiymati ota-ona tugunida berilgan qiymatdan kichik bo‘lishi kerak, o‘ng bolasining berilganlar maydonining (kalit) qiymati esa ota-onadagidan katta yoki teng bo‘lishi kerak.

Binar qidirish daraxtiga misol:



Berilgan kalit bilan tugunni qidirish. Misol tariqasida quyidagi binar qidirish daraxtini ko‘rib chiqamiz:



Aytaylik, kaliti 57 ga teng bo‘lgan tugunni topish kerak bo‘lsin.

Qidirish ildiz tugunidan boshlanadi va 57 qiymatni ildiz tugunining kaliti 63 ga solishtiradi. Qidirilayotgan qiymat kichik, shuning uchun qidirish daraxtning chap tomonida ya'ni ildiz tugunining chap qismi yoki chap pastki daraxtning avlodlaridan biri tomonida davom ettiriladi. Ildiz tugunining chap bolasining kaliti 27; 57 va 27 solishtiriladi, bu esa o‘z navbatida qidirilayotgan tugun 27 kalitli tugunning o‘ng pastki daraxtiga tegishli ekanligini ko‘rsatadi. Shu tarzda 51 bilan solishtiriladi undan katta, shuning uchun qidirish jarayoni avval o‘nga boradi va 58 ga, so‘ngra chapga 57 ga o‘tadi. Bu safar 57 kaliti aynan qidirilayotgan qiymatga teng. Shu bilan qidirish nihoyasiga yetadi**.**

11.5.1. Binar daraxtda tugunni qidirishni amalga oshirish.

Tugunni qidirish usuli quyida keltirilgan:

```
public BTNode<T> Find(T key)
{
  // berilgan kalitli tugunni qidirish
  var temp = Root; // ildiz tugundan boshlanadi
  while (temp != null)
  {
    int result = key.CompareTo(temp.Data);
    if (result == 0) break;
    if (result < 0) // Chapga harakatlanadi?
      temp = temp.Left;
    else temp = temp.Right; yoki o‘nga?
  }
  return temp;
}
```

Muvaffaqiyatsiz qidirish

Agar vaqtinchalik o‘zgaruvchi *temp* = *null* bo‘lsa, keyingi bola tugun topilmaganini bildiradi; qidirish daraxtning oxiriga yetdi, kerakli tugun topilmadi va shuning uchun bunday kalitli tugun mavjud emas.

Muvaffaqiyatli qidirish

Agar temp - vaqtinchalik o‘zgaruvchi qiymati null bo‘lmasa, bu kerakli tugun muvaffaqiyatli topilganligini bildiradi. *Find*() usuli ushbu tugunga havolani qaytaradi.

11.5.2. Binar daraxt bo‘ylab qidirish samaradorligi

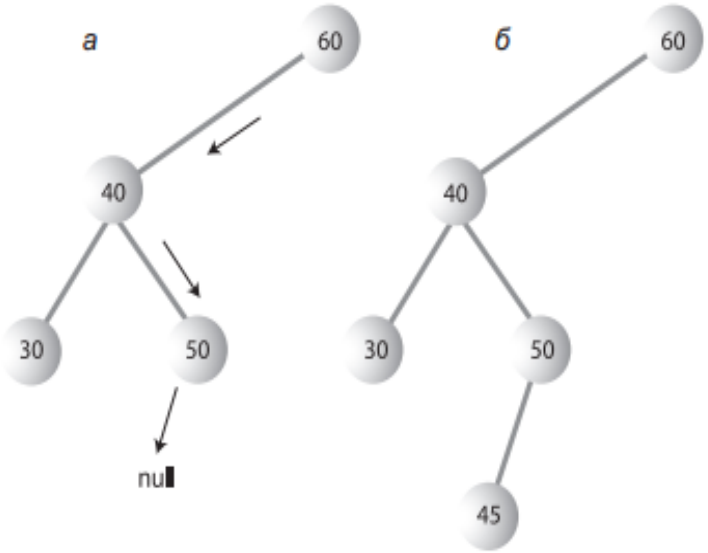
Tavsifdan ko‘rinib turibdiki, tugunni qidirish vaqti daraxt pog‘onalari soniga bog‘liq. Ya’ni o‘rtacha qidirish vaqti O(*logN*) va aniqrog‘i O($\log_2 N$) teng bo‘ladi.

11.5.3. Binar qidirish daraxtiga tugunni kiritish

Tugunni kiritish uchun avval unga joy topishingiz kerak. Bu jarayon deyarli mavjud bo‘lmagan tugunni topishga teng. Funksiya yangi tugunning ota-onasi bo‘ladigan tugunlarni ildizdan tugungacha kuzatib boradi. Ota-ona topilganda, yangi tugunning kaliti ota-ona kalitidan kichik yoki kattaligiga qarab, yangi tugun chap yoki o‘ng tomon sifatida kiritiladi.

Tugunni kiritishga misol:

Aytaylik, biz qiymati 45 ga teng bo‘lgan yangi tugunni qo‘shmoqchimiz. Tugunni kiritish tartib-qoidasi kiritish uchun joyni qidirishdan boshlanadi.



Tugun kiritish: a-qo‘shishdan oldin; b-qo‘shgandan keyin

45 qiymati 60 dan kichik, lekin 40 dan katta - 50 qiymatli tugunga qarab o‘tadi. Endi biz chapga o‘tishimiz kerak, chunki 45 qiymat 50 dan kichik, lekin 50 qiymatli tugunda chap bola tugun yo‘q; uning chap ko‘rsatkichi *NULL*. *NULL* topilsa, kiritish algoritmi yangi tugunni biriktirish uchun joy topilgan deb hisoblaydi. Funksiya 45 kalitli yangi tugunni yaratadi va rasmda ko‘rsatilganidek, uni chap bola sifatida 50 kalitli tugun bilan bog‘laydi.

Tugunni kiritishni amalga oshirish

Tugunni qo‘shishning rekursiv bo‘lmagan usuli:

```
// Binar qidirish daraxtiga tugunni kiritish
public void Add(T data)
{
  // Yangi tugun yarating
  var newNode = new BTNode<T>(data);
  // Ildiz tugun mavjud emas
  if (Root == null) Root = newNode;
  else
  {
  // Ildiz tugunidan boshlang
    var temp = Root;
    while (true)
    {
    // (ichki sikldan chiqish)
      var parent = temp;
      int result = data.CompareTo(temp.Data);
      if (result == 0) return;
      if (result < 0)
      {
        temp = temp.Left;
//Agar  zanjirning  oxirigacha  yetib  borsa,  chapdan
//kiriting
        if (temp == null)
        {
          parent.Left = newNode;
          newNode.Parent = parent;
          return;
        }
      }
      else
      { // Yoki to‘g‘rimi?
        temp = temp.Right;
```

```
//Agar zanjirning oxirigacha yetib borsa, o‘ng tomonga
//kiriting
        if (temp == null)
        {
          parent.Right = newNode;
          newNode.Parent = parent;
          return;
        }
      }
    }
  }
}
```

Yangi *parent* o‘zgaruvchisi (*temp* ning ota-onasi) daraxt bo‘ylab o‘tishdagi so‘ngi *null* bo‘lmagan tugunni ushlab turadi (rasmda 50 qiymatli tugun). Bu tugunni saqlash zarur, chunki *temp* o‘zgaruvchi qiymati *null* ga teng bo‘lsa, u holda *parent* o‘zgaruvchi va yangi tugun bir - biriga bog‘lanadi. Yangi tugunni bog‘lashni *parent* siz amalga oshirishning iloji bo‘lmasdi.

Yangi tugunni kiritish *parent* (*null* dan farqli oxirgi tashrif buyurilgan tugun) dagi bolaga mos ko‘rsatgichni o‘zgartirish orqali amalga oshiriladi. Agar *parent*ning chap bolasini izlash muvaffaqiyatsiz yakunlangan bo‘lsa, yangi tugun *parent*ga chap tomon sifatida, o‘ng bolani qidirishda esa mos ravishda o‘ng bola sifatida biriktiriladi. Rasmda 45 qiymati 50 qiymatli tugunga chap bola sifatida biriktirilgan, chunki u 50 dan kichik.

11.5.4. Binar qidirish daraxtidan tugunni olib tashlash

Tugunni o‘chirish algoritmi ancha murakkab va ko‘plab omillarga bog‘liq.

Berilgan ildizli bitta daraxt ostini boshqa ildizli boshqa daraxt ostiga qo‘yishimiz kerak bo‘ladi. Buning uchun bizga rekursiv usul kerak bo‘ladi:

```
//  currentnode  daraxt  ostiga  node  daraxt  ostini
//kiritishning rekursiv usuli
public  void  AddNode(BTNode<T>  node,  BTNode<T>
currentNode = null)
{
```

```
  if (Root == null)
  {
    node.Parent = null;
    Root = node;
  }
  currentNode = currentNode ?? Root;
  node.Parent = currentNode;
  int result = node.Data.CompareTo(currentNode.Data);
  if (result == 0) return;
  if (result < 0)
  {
    if (currentNode.Left == null)
    {
      currentNode.Left = node;
    }
    else
    {
      AddNode(node, currentNode.Left);
    }
  }
  else
  {
    if (currentNode.Right == null)
    {
      currentNode.Right = node;
    }
    else
    {
      AddNode(node, currentNode.Right);
    }
  }
}
```

Binar qidirish daraxti tugunini o‘chirish kodi quyida berilgan:

```
public void Remove(T data)
```

```
{
  BTNode<T> node = Find(data);
  if (node == null) { return; }
//agar tugun avlod tugunga ega bo‘lmasa uni o‘chirish
mumkin
  if (node.Left == null && node.Right == null)
  {
    if (node.NodeSide == Side.Left)
    {
      node.Parent.Left = null;
    }
    else
    {
      node.Parent.Right = null;
    }
    return;
  }
// agar chap bola tugun yo‘q bo‘lsa, u holda //o‘ng bola
tugun o‘chirilgan tugun o‘rniga qo‘yiladi
  if (node.Left == null)
  {
    if (node.NodeSide == Side.Left)
    {
      node.Parent.Left = node.Right;
    }
    else
    {
      node.Parent.Right = node.Right;
    }
    node.Right.Parent = node.Parent;
    return;
  }
//agar o‘ng bola tugun yo‘q bo‘lsa, u holda chap //bola
tugun o‘chirilgan tugun o‘rniga qo‘yiladi
```

```
  if (node.Right == null)
  {
    if (node.NodeSide == Side.Left)
    {
      node.Parent.Left = node.Left;
    }
    else
    {
      node.Parent.Right = node.Left;
    }
    node.Left.Parent = node.Parent;
    return;
  }
//agar ikki bola tugun ham mavjud bo‘lsa,
//o‘ng bola tugun o‘chiriladigan tugun o‘rniga //chap
bola tugun esa o‘ng bola tugun o‘rniga o‘tadi
  switch (node.NodeSide)
  {
    case Side.Left:
      node.Parent.Left = node.Right;
      node.Right.Parent = node.Parent;
      AddNode(node.Left, node.Right);
      break;
    case Side.Right:
      node.Parent.Right = node.Right;
      node.Right.Parent = node.Parent;
      AddNode(node.Left, node.Right);
      break;
    default:
      node.Data = node.Right.Data;
      node.Right = node.Right.Right;
      node.Left = node.Right.Left;
      AddNode(node.Left, node);
      break;
```

```
  }
  return;
}
```

O‘chirilgan tugunlarni hisobga olishning osonroq yo‘li mavjud. Buning uchun BTNode tugunni ifodalovchi berilganlar tuzilmasiga *isDeleted* nomli yangi maydon - mantiqiy turdagi xususiyat kiritish kerak bo‘ladi. Tugunni olib tashlash uchun bu xususiyatga *true* qiymati beriladi. *Find*() kabi boshqa amallar tugun ustida ishlashdan oldin tugun o‘chirilgan yoki yo‘qligiga ishonch hosil qilish uchun ushbu maydonni tekshiradi. Ushbu yondashuv bilan tugunni o‘chirish daraxtning tuzilishini o‘zgartirmaydi. Albatta, bu xotirani "o‘chirilgan" tugunlar bilan to‘ldirishi mumkinligini ham anglatadi. Ushbu yondashuv murosaliroq bo‘lishi mumkin, lekin u daraxtdan nisbatan kam sonli o‘chirish uchun mos bo‘lishi mumkin. (Masalan, agar sobiq xodimlarning ma'lumotlari xodimlar bo‘limi ma'lumotlar bazasida abadiy qolsa.)

Daraxtni ekranga chop qilish

Daraxtni ekranda ko‘rsatish uchun quyidagi *Print*() usulidan foydalaniladi. U daraxt bo‘ylab o‘tishning turli usullarini chaqirishi mumkin, masalan:

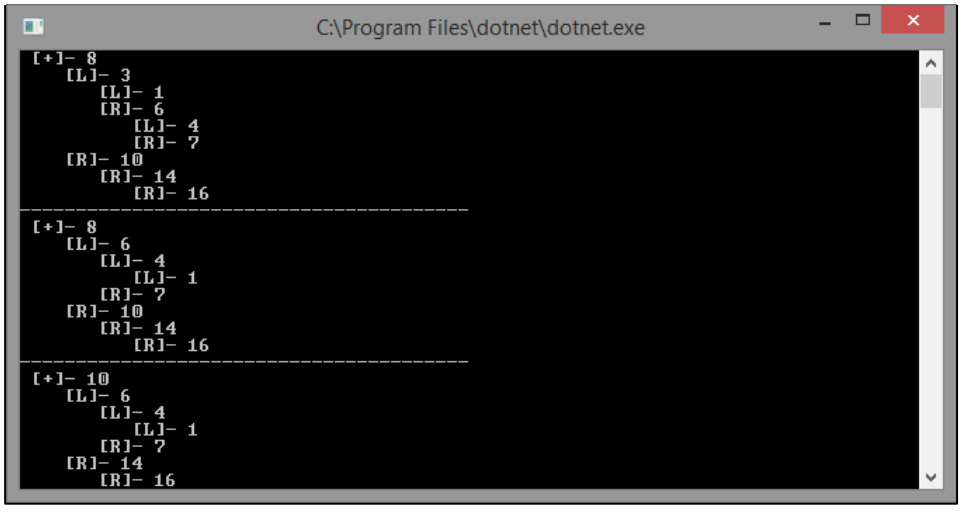
```
// Ikkilik daraxtni chiqarish
public void Print()
{
  PrintPreRec(Root);
// PrintPreNoRec();
// PrintInfRec(Root);
// PrintPosRec(Root);
// PrintPosNoRec();
// PrintBFS();
}
```

Daraxtlarni asosiy dasturda ishlatilishi:

```
using System;
namespace BinaryTree
{
  class Program
  {
```

```
    static void Main(string[] args)
    {
      var binaryTree = new BinaryTree<int>();
      binaryTree.Add(8);
      binaryTree.Add(3);
      binaryTree.Add(10);
      binaryTree.Add(1);
      binaryTree.Add(6);
      binaryTree.Add(4);
      binaryTree.Add(7);
      binaryTree.Add(14);
      binaryTree.Add(16);
      binaryTree.Print();
      Console.WriteLine(new string('-', 40));
      binaryTree.Remove(3);
      binaryTree.Print();
      Console.WriteLine(new string('-', 40));
      binaryTree.Remove(8);
      binaryTree.Print();
      Console.ReadLine();
    }
} }
```

Dastur natijasi:



Daraxt BATining to‘liq dasturi ***8 - ilovada*** keltirilgan.

11.5.5. Binar qidirish daraxtlarning samaradorligi

Yuqorida aytib o‘tganimizdek, ko‘plab daraxt amallari ma'lum bir tugunni topish uchun pog‘onadan pog‘onaga tushishni talab qiladi. Bunday o‘tish qancha vaqt talab qiladi? To‘liq daraxtda tugunlarning taxminan yarmi eng pastki pog‘onada bo‘ladi. (Aniqrog‘i, daraxt to‘la bo‘lsa, daraxtning qolgan qismiga qaraganda pastki pog‘onada bitta tugun ko‘proq bo‘ladi.) Shuning uchun qidirishlar, qo‘shishlar va o‘chirishlar kabi barcha amallar eng pastki pog‘onadan tugunni qidirishni talab qiladi. Amallar soni daraxtning balandligiga bog‘liq. Standart daraxt amallarining bajarilish vaqti $\log_2 N$ ga mutanosib, bajarilish vaqti bo‘yicha murakkabligi $O(\log_2 N)$ ga teng bo‘ladi.

Keling, daraxtni hozirgacha ko‘rib chiqilgan boshqa berilganlar tuzilmalari bilan taqqoslaylik. Elementlari soni 1 000 000 ta bo‘lgan tartiblanmagan massiv yoki bog‘langan ro‘yxatda kerakli elementni topish uchun o‘rtacha 500 000 ta taqqoslash kerak bo‘ladi.

Ammo 1 000 000 tugunli daraxt uchun 20 ta (yoki undan kam) taqqoslash yetarli bo‘ladi.

Tartiblangan massivda elementlar tezda topiladi, lekin kiritish uchun o‘rtacha 500 000 ta amal kerak bo‘ladi. 1 000 000 ta tugunli daraxtga element kiritish uchun 20 yoki undan kam taqqoslash va tugunni ulash uchun ahamiyatsiz vaqt talab etiladi.

Xuddi shunday, 1 000 000 ta elementli massivdan elementni olib tashlash uchun o‘rtacha 500 000 ta elementni siljitish kerak bo‘lsa, 1 000 000 ta tugunli daraxtdan elementni olib tashlash uchun 20 yoki undan kam taqqoslash kerak bo‘ladi va o‘rindoshini topish uchun (ehtimol) yana bir necha taqqoslash kerak va elementni ajratish va vorisni biriktirish uchun ahamiyatsiz vaqt talab qiladi.

Shunday qilib, daraxt berilganlarni saqlashning barcha asosiy amallari uchun yuqori samaradorlikni ta'minlaydi. O‘tish boshqa amallar kabi tez emas. Biroq, odatdagi katta berilganlar bazasi uchun o‘tish odatiy amallar toifasiga kirmaydi. Daraxt bo‘ylab o‘tish ko‘proq algebraik va boshqa kamdan-kam uzunlikdagi ifodalarni tahlil qilishda qo‘llaniladi.

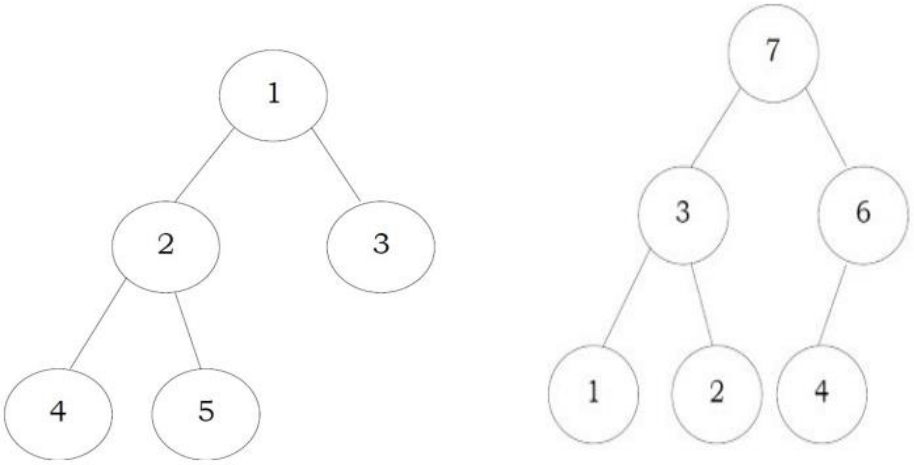
Binar qidirish daraxtining kamchiligi shundaki, kiritiladigan berilganlar tartiblangan ketma-ketlikda bo‘lsa, unda kiritish va qidirish amallari uzoq vaqt $O(n)$ oladi, chunki bu holda muvozanatsiz daraxt olinadi, ya'ni chap egri yoki o‘ng egri daraxt hosil bo‘ladi.

11 – bob bo‘yicha nazorat savollari

1. Qidirish turlari.
2. Tartiblanmagan chiziqli qidirish qanday amalga oshiriladi?
3. Tartiblangan chiziqli qidirish qanday amalga oshiriladi?
4. Binar qidirish qanday amalga oshiriladi?
5. Interpolyatsion qidirish qanday amalga oshiriladi?
6. Binar qidirish daraxti qanday amalga oshiriladi?
7. Binar qidirish daraxtlarning samaradorligi qanday amalga oshiriladi?

12. Uyum *(Heap)*

Uyum (piramida) - bu o‘ziga xos xususiyatlarga ega bo‘lgan daraxtdir. Uyum uchun asosiy talab tugundagi qiymat uning bola tugunlari qiymatlaridan katta (≥) yoki kichik (≤) bo‘lishi kerak. Bu uyum xususiyati deb ataladi. Uyumning qo‘shimcha xususiyati ham mavjud: qandaydir h > 0 uchun barcha barg tugunlari *h* yoki *h - 1* (bu yerda *h* - daraxt balandligi) pog‘onada bo‘lishi kerak. Masalan:



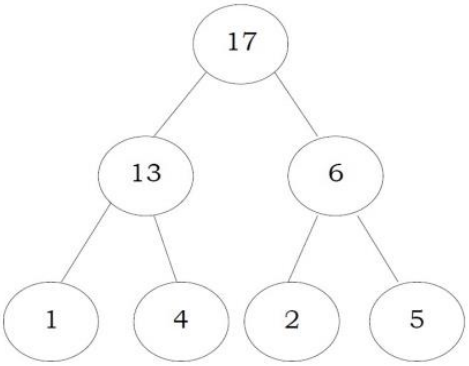
12.1. Uyumlar turlari

Uyumning xususiyatlaridan kelib chiqib, biz uyumlarni ikki turga bo‘lishimiz mumkin:

• *min*-uyum: tugunning qiymati uning bola tugunlari qiymatlaridan kichik yoki teng bo‘lishi kerak;

• *max*-uyum: tugunning qiymati uning bola tugunlari qiymatlaridan katta yoki teng bo‘lishi kerak.

Binar (ikkilik) uyumda har bir tugunda ikkitagacha bola tugun bo‘lishi mumkin. Biz faqat ikkilik uyumlarni ko‘rib chiqamiz va keyingi muhokamalarda minimal ikkilik uyumlarga va maksimal ikkilik uyumlarga e'tibor qaratamiz.



12.2. Uyumni tasvirlash

Uyum amallarini ko‘rib chiqishdan oldin, keling, uyumlarni qanday amalga oshirish mumkinligini ko‘rib chiqaylik.

Imkoniyatlardan biri dinamik massivlardan foydalanishdir. Faraz qilaylik, elementlar 0 indeksidan boshlanadigan massivlarda saqlangan deylik. Yuqoridagi *max*-uyumi quyidagicha ifodalanishi mumkin:

17	13	6	1	4	2	5
0	1	2	3	4	5	6

Uyumni ifodalovchi *Heap* sinfi ikki atributga ega: *size* – massiv o‘lchami, *count*- uyumdagi elementlar soni bo‘lib, uyumda *arr* massivning nechta elementi mavjudligini ifodalaydi. Ya'ni, uyum o‘lchamini *count*<=*size* o‘zgartirish mumkin. Daraxtning ildizi *arr*[0] va element (tugun) indeksini (*i* ni) bilgan holda quyidagi formuladan foydalanib, uning ota-onasi, chap va o‘ng bolalarining massividagi indekslarni osongina hisoblashimiz mumkin:

```
parent=(i-1)/2
left=2*i+1
right=2*i+2
```

Keyinchalik biz faqat *max* uyum bilan ishlaymiz.

Heap sinfi quyidagi ko‘rinishga ega:

```
class  Heap<T> where T: IComparable
{
```

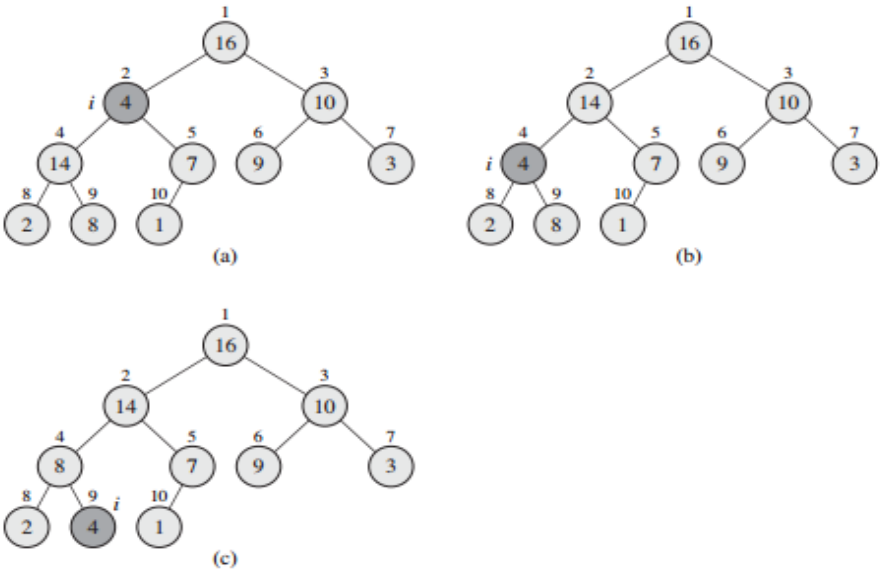
```
  T [] arr;// Tugundagi qiymatlarni saqlovchi massiv
  int count;// uyum elementlari soni
  int size; // uyum massivi o‘lchami
  public int Count { get { return count; } }
  public bool IsEmpty { get { return count == 0; } }
  public Heap(int n=100)
  {
    count = 0;
    size = n;
    arr = new T[n];
  }
  //  uyumni qayta ishlash usuli
}
```

12.3. Daraxtni uyumga keltirish

Uyumdagi elementlarni qayta tiklagandan so‘ng, u uyum xususiyatini qanoatlantirmasligi mumkin. Bunday holda, daraxtning tugunlarini shunday qayta joylashtirilishi kerakki, u uyum xususiyatiga ega bo‘lishi kerak. Ushbu jarayon uyush (uyumga keltirish, heapify) deb ataladi.

ax uyumda tugunni to‘g‘ri to‘ldirish uchun biz uning eng katta qiymatli tugunlarini topib, uni joriy tugun bilan almashtirishimiz kerak va bu jarayonni har bir tugunda uyum xususiyati qanoatlanguncha davom ettirishimiz kerak bo‘ladi.

Quyidagi rasmda qiymati 4 bo‘lgan tugun uyum xususiyatini qanoatlantirmaydi. Keling, ushbu tugunni ko‘chirishga harakat qilaylik. Buning uchun maksimal qiymatga ega bo‘lgan bola tugunini topamiz va uni joriy tugun bilan almashtiramiz. Keyin bu jarayonni tugun uyum xususiyatlarini qanoatlantirguncha davom ettiramiz. Ya'ni, 4 ni 8 ga almashtirsak, biz quyidagilarni olamiz:



Endi daraxt uyum xususiyatini qanoatlantiradi. Yuqorida keltirilgan uyum xususiyatini tiklash jarayoniga uyumni tartiblash (elakdan o‘tkazish) deb ataladi.

Uyush usuli quyidagicha ko‘rinadi:

```
private void Heapify(int i)
{
  int l, r,max;
  T tmp;
  l = 2 * i + 1;
  r = 2 * i + 2;
  max =(l < count && arr[l].CompareTo(arr[i])>0) ? l:i;
  if (r < count && arr[r].CompareTo(arr[max])>0) max=r;
  if (max != i)
  {
    tmp = arr[i];
    arr[i] = arr[max];
    arr[max] = tmp;
    Heapify(max);
  }
}
```

Vaqt bo‘yicha murakkabligi: O(*logn*) teng. Uyum to‘liq ikkilik daraxt bo‘lib, eng yomon holatda biz ildizdan boshlaymiz va bargga qadar harakat

qilamiz. Bu to‘liq ikkilik daraxtning balandligiga teng. Xotira sarfi (fazoviy murakkabligi): O(1).

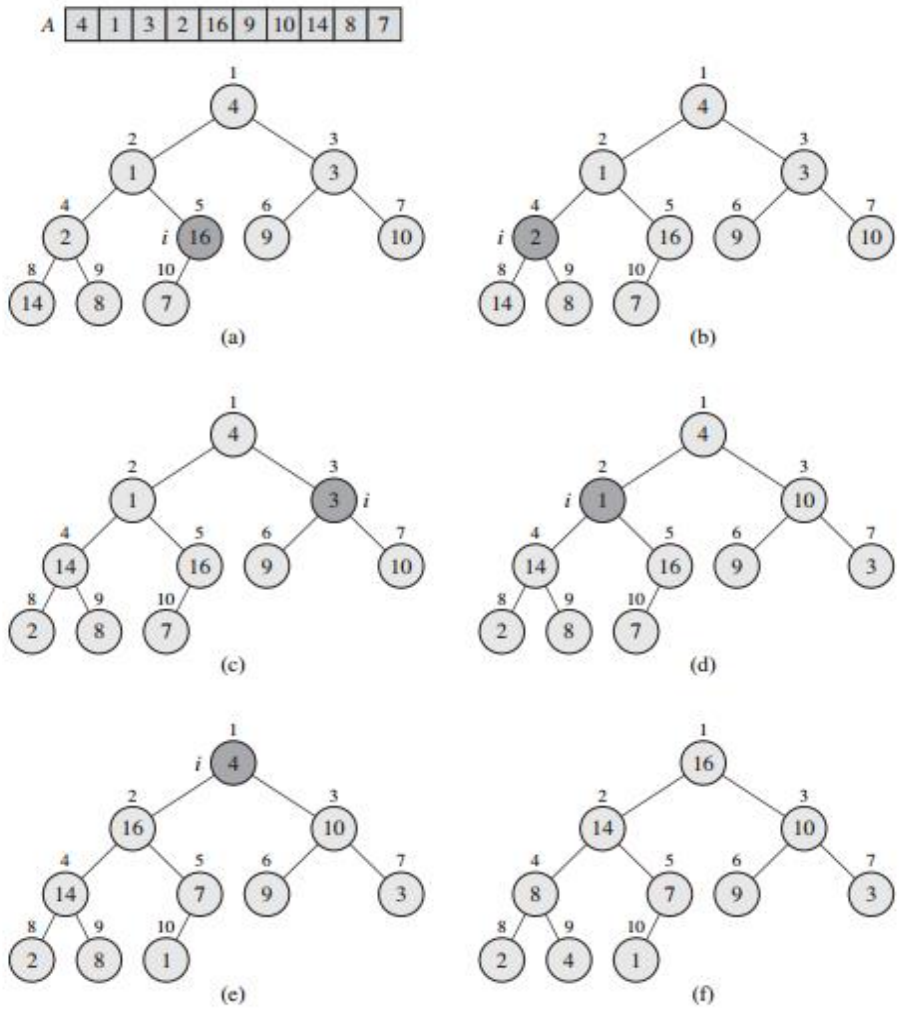
12.4. Dastlabki massivdan uyum qurish.

Dastlabki massivdan uyum yaratish uchun konstruktorni qayta yuklaymiz va unga parametr sifatida massivni oladigan o‘zgaruvchini beramiz.

Uyumning barg tugunlari massiv oxirida joylashganligini hisobga olsak, uyumning oxirgi tugunining ota-onasi bo‘lgan tugun indeksi (n-2)/2 ga teng. Keyinchalik, siklda har bir ota-tugun uchun biz *Heapify*() usulini qo‘llaymiz.

```
public Heap(int[]A)
{
  count = A.Length;
  size = count;
  arr = new int[count];
  for (int i = 0; i < count; i++) arr[i] = A[i];
  for (int i = (count - 2) / 2; i >= 0; i--)
  {
    Heapify(i);
  }
}
```

Quyida massivdan uyum yaratish misoli keltirilgan:



12.5. Maksimal elementni olish

max-uyimdagi maksimal element har doim ildizda bo‘lgani uchun u *arr[0]* da saqlanadi, uni olish usuli:

```
public T GetMaximum()
{
  if (IsEmpty)
    throw new InvalidOperationException("Uyum bo’sh");
  return arr[0];
}
```

Vaqt bo‘yicha murakkabligi: O(1) ga teng.

12.6. Maksimal elementni o‘chirish va qaytarish

Maksimal elementni uyumdan olib tashlash uchun biz shunchaki elementni ildizdan olib tashlashimiz kerak. Ildiz elementini olib tashlaganingizdan so‘ng, uyumning oxirgi elementini ildiz joyiga ko‘chiriladi. Elementni almashtirgandan so‘ng, daraxt uyum xususiyatini qanoatlantirmasligi mumkin. Uyumni qayta yaratish uchun *Heapify*(0) usulini chaqirish kerak bo‘ladi.

Elementni o‘chirish usuli quyidagicha bo‘ladi:

```
public int DeleteMax()
{
  int data;
  if (IsEmpty)
    throw new InvalidOperationException("Uyum bo‘sh");
  data = arr[0];
  arr[0] = arr[count - 1];
  Heapify(0);
  count--;
  return data;
}
```

12.7. Uyumga element qo‘shish

Elementni uyumga qo‘shish xuddi elementli uyumdan olib tashlash jarayoniga o‘xshaydi. Uyumga yangi element kiritish uchun quyidagi amallarni bajarish kerak:

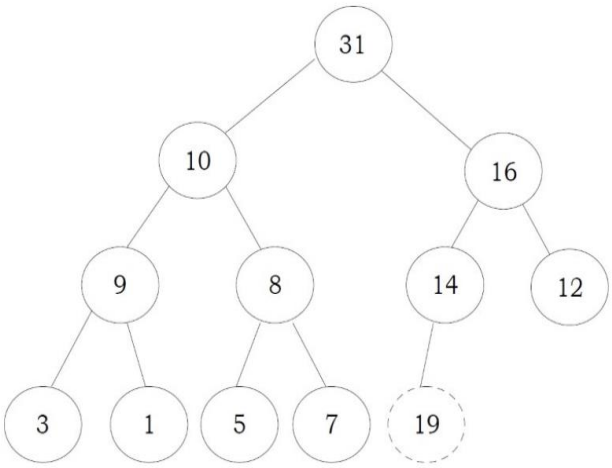
• Agar massiv o‘lchami yetarli bo‘lmasa, kattalashtirish (ikkilantirish);

• Yangi elementni uyumning (daraxt) oxirida saqlash;

• Elementni pastdan yuqoriga toki u o‘z joyiga tushgunga qadar siljitish.

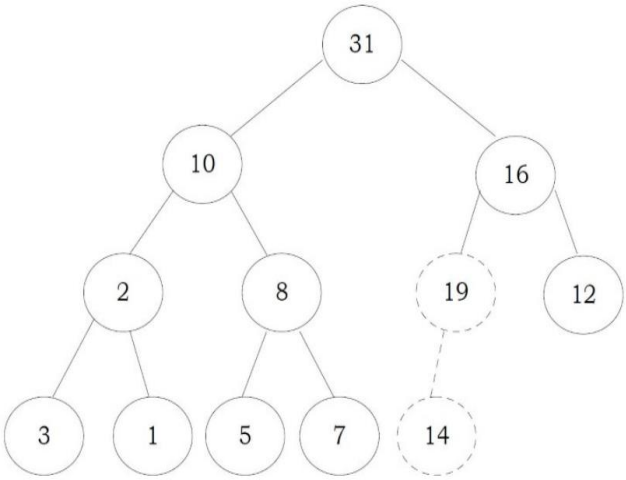
Kodga o‘tishdan oldin, keling, bir misolni ko‘rib chiqaylik.

Biz 19 qiymatli elementni uyumning oxiriga joylashtirdik va bu uyum xususiyatini qanoatlantirmaydi:

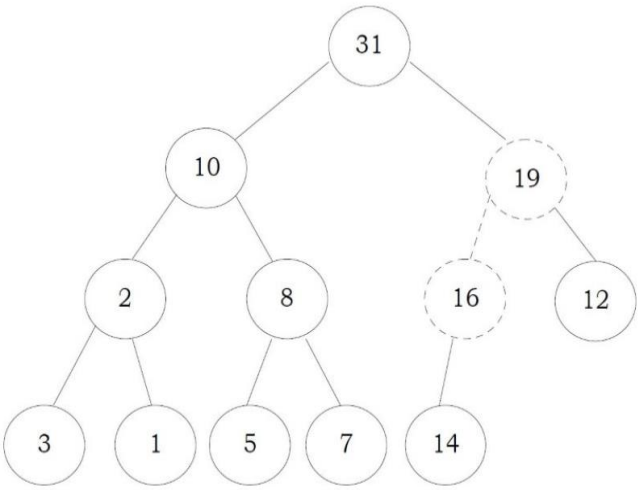


Ushbu elementni (19) joyiga o‘rnatish uchun uni ota-onasi tuguni bilan solishtirishimiz va ularni qayta qurishimiz kerak bo‘ladi.

19 va 14 o‘rinlarini alshmashtiramiz va quyidagini olamiz:



19 va 16 qiymatlar o‘zaro almashtiriladi:



Endi xosil bo‘lgan daraxt uyum xususiyatlarini qanoatlantiradi. Biz pastdan yuqoriga harakatlanayotganimiz sababli, bu jarayon yuqoriga o‘tish deb ataladi.

Elementni uyumga kiritish usuli quyidagi ko‘rinishga ega:

```
public void Insert(int data)
{
  int i=count;
  if (count == size) Resize();
    count++;
  while (i > 0 && data.CompareTo(arr[(i - 1) / 2])>0)
  {
    arr[i] = arr[(i - 1) / 2];
    i = (i - 1) / 2;
  }
  arr[i] = data;
}
```

Massiv o‘lchamini oshirish usuli quyidagi ko‘rinishga ega:

```
private void Resize()
{
  int[] tmp = arr;
  size *= 2;
  arr = new int[size];
  for (int i = 0; i < count; i++) arr[i] = tmp[i];
}
```

12.8. Uyum elementlarini chop etish

Uyum elementlarini ekranga chiqarish uchun *Print*() usulidan foydalanish mumkin:

```
public void Print()
{
  Console.WriteLine();
  for (int i = 0; i < count; i++)
    Console.Write($"{arr[i]}  ");
  Console.WriteLine();
}
```

12.9. Uyumni tozalash

Uyumni tozalash uchun *Clear*() usuli qo‘llaniladi:

```
public void Clear()
{
  count = 0;
  size = 10;
  arr = new T[size];
}
```

12.10. Uyumni tartiblash algoritmi HeapSort()

Uyumni tartiblashni amalga oshirish uchun *A[n]* kirish massividan uyum qurish kerak, bu yerda *n* - massivning o‘lchami. Buning uchun parametri massiv bo‘lgan konstruktordan foydalanamiz. Uyum massivi o‘lchami *count* = *n* ga teng.

Insert(*data*) usulidan foydalanib, kiritiladigan yoki hisoblanadigan qiymatlarni ketma - ket uyumga qo‘shish orqali uyum yaratilishi mumkin.

Massivning maksimal elementi *arr[0]* ildizida saqlanganligi uchun uni massivning oxirgi elementi *arr[count-1]* bilan almashtiramiz.

Keyinchalik, biz oxirgi tugunni uyumdan olib tashlaymiz, ya’ni uyum hajmini qisqartiramiz *count=count-1*;

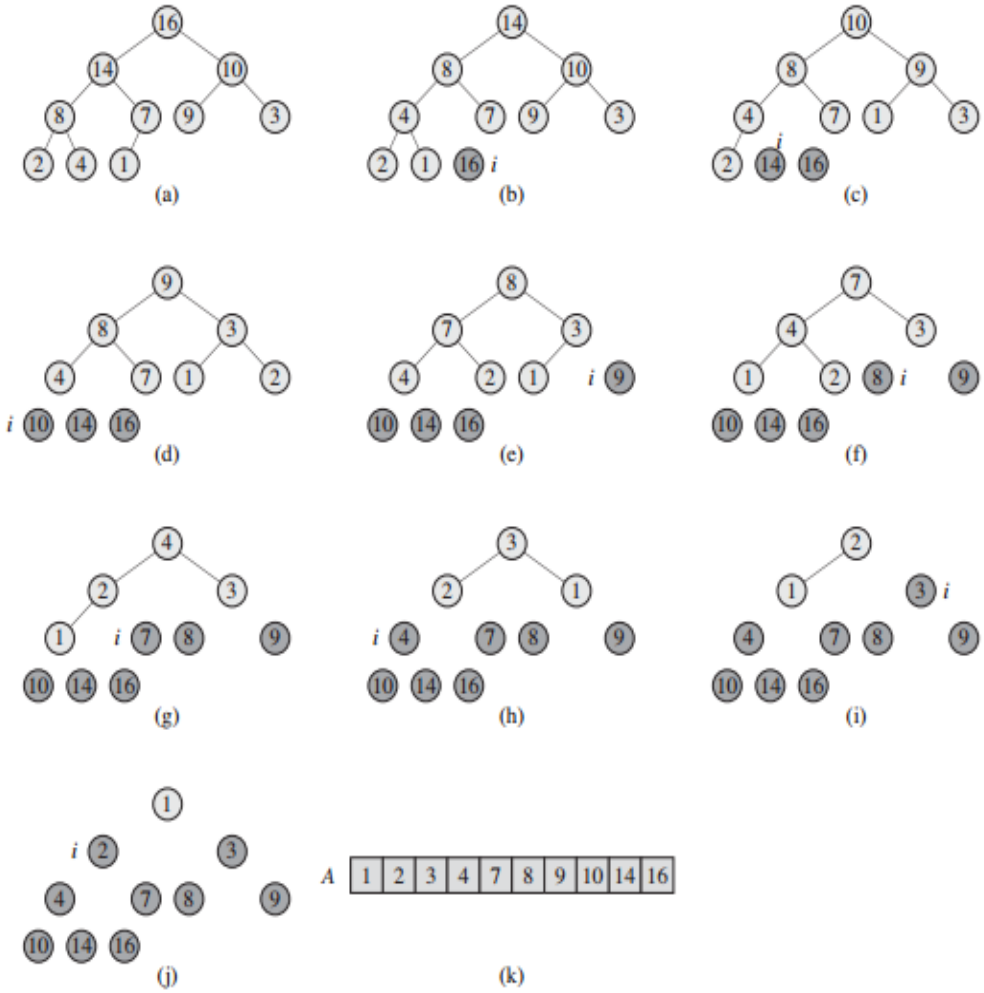
O‘zgartirishdan so‘ng, ildiz tugunining yangi qiymati *max*-uyum xususiyatini buzishi mumkin. Vaziyatni to‘g‘irlash uchun *Heapify*(0) usulini chaqirish kifoya, bu esa yangi qayta qurilgan uyumni yaratadi.

Keyin algoritm bu jarayonni *count*-1 = 2 bo‘lgunga qadar takrorlaydi:

```
public void HeapSort()
{
  T tmp;
  int n = count;
  for (int i = count- 1; i>0; i--)
  {
    tmp = arr[0];
    arr[0] = arr[count - 1];
    arr[count - 1] = tmp;
    count--;
```

```
    Heapify(0);
  }
  count = n;
}
```

Rasmda boshlang‘ich *max*-uyumi qurilgandan keyin *HeapSort* qanday ishlashiga misol ko‘rsatilgan.



HeapSort protsedurasi O(*nlgn*) vaqtni oladi, chunki *Heap*(*A*) konstruktori bajarish uchun O(*n*) vaqt oladi va *Heapify*() funksiyasini *n*-1 ta chaqiruvining har biri O(*lgn*) vaqt oladi.

Algoritmning murakkabligi baholash:

Eng yomon holatdagi murakkabligi: O(*n*log*n*)

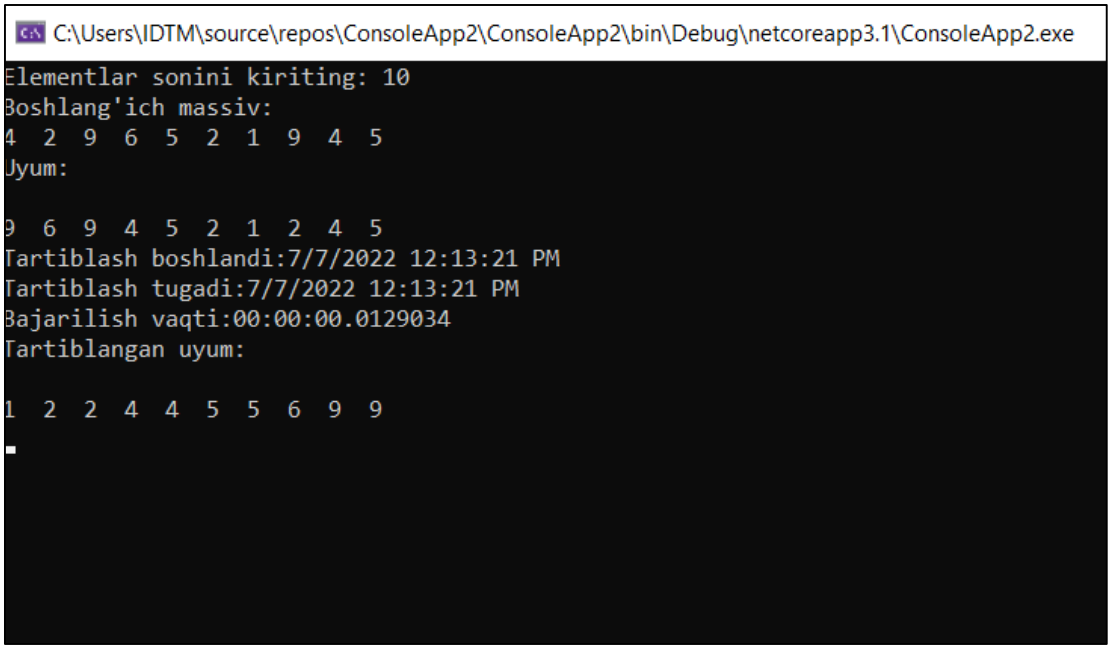
Eng yaxshi holatdagi murakkabligi: O(*n*log*n*)

O‘rtacha murakkabligi: O(*n*log*n*)

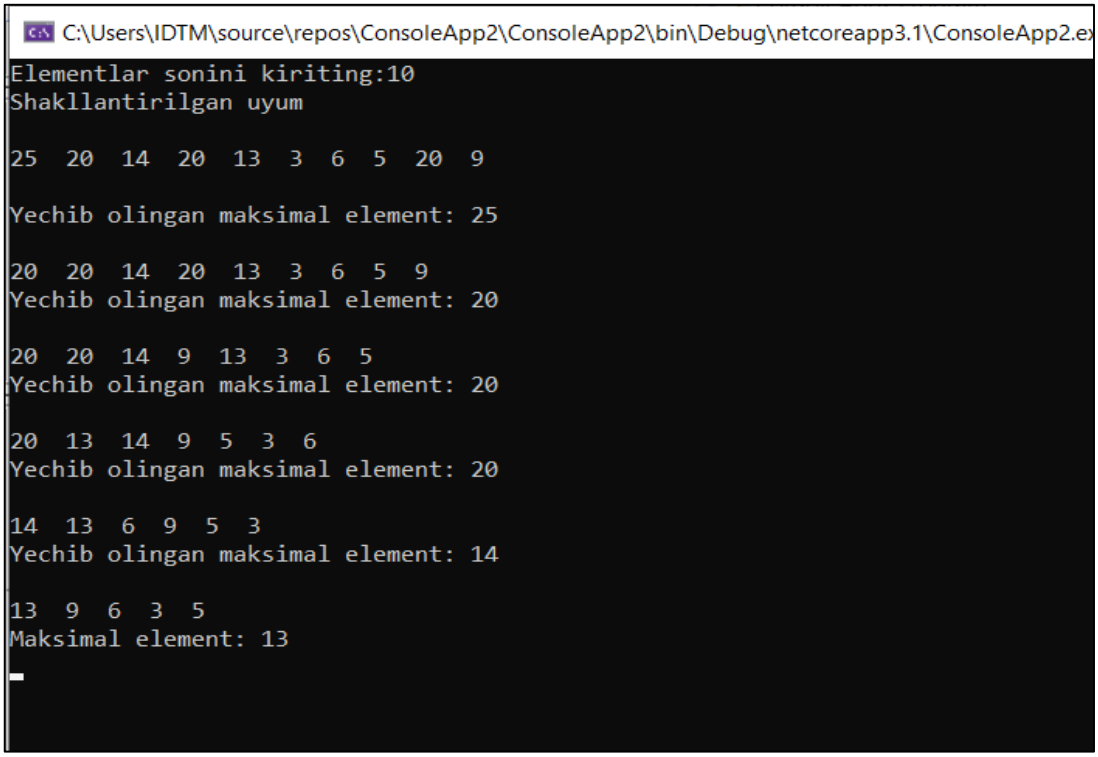
Eng yomon holatdagi fazosining murakkabligi: O(*n*) jami, O(1) qo‘shimcha.

Sinf tavsifining to‘liq matni ***9 - ilova***da keltirilgan.

Tartiblash bajarilishi natijasi:



Navbat bajarilishi natijasi:



12 – bob bo‘yicha nazorat savollari

1. Uyum nima?
2. Uyum turlari.
3. Uyum qanday tasvirlanadi?
4. Daraxtni uyumga qanday keltirish.
5. Maksimal elementni o‘chirish va qaytarish qanday amalga oshiriladi?
6. Uyumni tartiblash algoritmi.

Glossariy

Algoritm - kerakli natijaga erishish uchun hisoblab chiqiladigan qadamlar to’plami.

Algoritm murakkabligi – algoritmning kiruvchi berilganlarga nisbatan sarflaydigan resurslar (vaqt,xotira) miqdori.

Algoritmlar tahlili – muayyan masalani yechish uchun qo’llanilgan algoritmlarning murakkabliklarini solishtirish.

Bajarilish vaqti tahlili - bu masala hajmi ortishi (kiruvchi berilganlarning hajmi oshganda) bilan bajarilish vaqti qanday o‘zgarishini aniqlash jarayonidir

Berilganlarning abstrakt(mavhum) turi - har qanday muayyan amalga oshirishdan qat'i nazar, aniq belgilangan ma'lumotlar qiymatlari va ular bilan bog'liq operatsiyalar to'plami.

Binar daraxt - har bir tugunida ko'pi bilan ikkita bolasi bor daraxt.

Berilganlar tuzilmalari - bu berilganlarni samarali ishlatish uchun kompyuterda saqlash va tartibga solishning maxsus usuli.
Bog‘langan ro‘yxat - bu berilganlar to‘plamini saqlash uchun ishlatiladigan berilganlar tuzilmasi hisoblanadi.
Daraxt - chiziqli bo‘lmagan berilganlar tuzilmasi bo’lib, unda har bir tugun boshqa bir tugunni emas, balki bir nechta tugunlarni ko‘rsatadi.
Dek (deque) - bu ikki tomonlama navbat bo‘lib, unda elementlar navbat boshiga yoki oxiriga qo‘shilishi va o’chirilishi mumkin.
Dinamik massiv - o‘zgaruvchan o‘lchamdagi ixtiyoriy kirish mumkin bo‘lgan berilganlar tuzilmasidir.
Massiv - butun sonlar (indekslari) orqali tasodifiy kirish mumkin bo'lgan elementlar to'plami.

Mukammal binar daraxt - agar binar daraxtning barcha barg tugunlari h yoki h - 1 balandlikda joylashgan bo‘lsa, shuningdek tugunlar ketma-ketligini raqamlashda hech qanday qiymat yetishmayotgan bo‘lsa, bunday binar daraxt mukammal binar daraxt deb ataladi.
Navbat - bu berilganlarni saqlash uchun ishlatiladigan berilganlar tuzilmasi bo’lib, unda kiritish bir uchida (dumidan) va o‘chirish ikkinchi uchida (boshdan) amalga oshiriladi.

O‘zgaruvchi - bu berilganlarni saqlash uchun ishlatilib, algoritm ishlashi davomida har xil berilganlarni o‘zgaruvchilarda saqlashi mumkin.

O‘sish tezligi - kiruvchi berilganlarning hajmiga qarab algoritmning ishlash vaqti ortib borish tezligi.
Qat’iy binar daraxt - har bir tuguni faqat ikkita bola tugun yoki umuman tugunga ega bo‘lmagan binar daraxt.
Qidirish - bu elementlar to‘plamidan berilgan xususiyatlarga ega elementni topish jarayonidir.
Stek - berilganlarni saqlash uchun ishlatiladigan oddiy berilganlar tuzilmasi bo‘lib, stekda berilganlarning kelish tartibi muhim ahamiyatga ega.
Tartiblash - bu ro‘yxat elementlarini ma'lum bir tartibda [o‘sish yoki kamayish bo‘yicha] joylashtiradigan algoritmdir.

Tayanch berilganlar turlari - tizim tomonidan aniqlanadigan berilganlar turlari tayanch turlar deyiladi.
To‘liq binar daraxt - agar har bir tugunda faqat ikkitadan bola tugun bo‘lsa va barcha barg tugunlari bir xil pog‘onada joylashgan bo‘lsa, bunday binar daraxt to‘liq binar daraxt deb ataladi.
Uyum (piramida) - bu o‘ziga xos xususiyatlarga ega bo‘lgan daraxtdir.

Adabiyotlar

1. R. Lafore / Структуры данных и алгоритмы на Java , 2 издание, 2017.
2. N. Karumanchi / Data Structures and Algorithms Made Easy: Data structures and Algorithmic Puzzles, Second edition, 2015.
3. M. A. Weiss / Data Structures & Algorithm Analysis in C++, 4th edition, 2014.
4. Aho A.V., Hopkroft J.E., Ullman J.E. / Data Structures and Algorithms, 2003.
5. Bucknall J./ The Tomes of Delphi™ Algorithms and Data Structures, 2003.

Ilovalar

1-ilova. Bir bog‘lamli ro‘yxat.

```
using System;
using System.Collections.Generic;
using System.Text;
namespace LList
{
  public class LList<T> where T:IComparable
  {
    Node<T> head; // bosh/birinchi element
    public void Print()
    {
      Node<T> current = head;
      while (current != null)
      {
        Console.WriteLine(current.Data);
        current = current.Next;
      }
    }
    public int Count()
    {
      int count = 0;
      Node<T> current = head;
      while (current != null)
      {
        count++;
        current = current.Next;
      }
      return count;
    }
    // ro‘yxat boshiga qo‘shish
    public void AppendFirst(T data)
    {
```

```
  Node<T> node = new Node<T>(data);
  node.Next = head;
  head = node;
}
public void Add(T data)
{
  Node<T> node = new Node<T>(data);
  Node<T> current = head;
  if (current == null)
    head = node;
  else
  {
    while (current.Next != null)
    { current = current.Next; }
    current.Next = node;
  }
}
public void Insert(T data, int pos)
{
  int k = 1;
  Node<T> node = new Node<T>(data);
  Node<T> current = head;
  while (current.Next != null && k < pos)
  { k++; current = current.Next; }
  node.Next = current.Next;
  current.Next = node;
}
public void Delete(int pos)
{
  int k = 1;
  Node<T> current = head;
  Node<T> previous = null;
  while (current.Next != null && k < pos)
  {
```

```
        k++; previous = current; current=current.Next;
      }
      if (pos <= 1) head = head.Next;
      else previous.Next = current.Next;
    }
    public void Clear()
    {
      head = null;
    }
  }
}
```

Asosiy dastur:

```
using System;
namespace LList
{
  class Program
  {
    static void Main(string[] args)
    {
      LList<string> linkedList = new LList<string>();
      // Elementlar qo‘shish
      linkedList.Add("Anvar");
      linkedList.Add("Rustam");
      linkedList.Add("Botir");
      linkedList.Add("Salim");
      // Ro‘yxat elementlarini ekranga chop etish
      linkedList.Print();
      // Ko‘rsatilgan tugunni o‘chirish
      linkedList.Delete(2);
      // Ro‘yxat boshiga element qo‘shish
      linkedList.AppendFirst("Bilol");
      linkedList.Print(); } } }
```

2 – ilova. Ikki bog‘lamli ro‘yxat.

```
using System;
using System.Collections.Generic;
using System.Text;
namespace DoubleList
{
  class DList<T> where T: IComparable
  {
    // bosh/birinchi element
    DNode<T> head;
    // dum / oxirgi element
    DNode<T> tail;
    int count;  // ro‘yxatdagi elementlar soni
    public int Count { get { return count; } }
    public bool IsEmpty { get { return count == 0; } }
    // ro‘yxat elementlarini chiqarish
    public void Print()
    {
      if (!IsEmpty)
      {
        DNode<T> current = head;
        while (current != null)
        {
          Console.WriteLine(current.Data);
          current = current.Next;
        }
      }
      else
        Console.WriteLine("Bo‘sh ro‘yxat");
    }
    public void PrintBack()
    {
      if (!IsEmpty)
      {
```

```
    DNode<T> current = tail;
    while (current != null)
    {
      Console.WriteLine(current.Data);
      current = current.Prev;
    }
  }
  else
    Console.WriteLine("Bo‘sh ro‘yxat");
}
// Elementni ro‘yxat boshidan boshlab qidirish
public bool ContainsH(T data)
{
  DNode<T> current = head;
  while (current != null)
  {
    if (current.Data.Equals(data))
      return true;
    current = current.Next;
  }
  return false;
}
// Elementni ro‘yxat oxiridan boshlab qidirish
public bool ContainsT(T data)
{
  DNode<T> current = tail;
  while (current != null)
  {
    if (current.Data.Equals(data))
      return true;
    current = current.Prev;
  }
  return false;
}
```

```
// ro‘yxat boshiga element qo‘shish
public void AddFirst(T data)
{
  DNode<T> node = new DNode<T>(data);
  if (IsEmpty)
    head = tail = node;
  else
  {
    head.Prev = node;
    node.Next = head;
    head = node;
  }
  count++;
}
// ro‘yxat oxiriga element qo‘shish
public void Add(T data)
{
  DNode<T> node = new DNode<T>(data);
  if (IsEmpty)
    head = tail = node;
  else
  {
    tail.Next = node;
    node.Prev = tail;
    tail = node;
  }
  count++;
}
// Ko‘rsatilgan o‘ringa element kiritish
public void Insert(T data, int pos)
{
  int k = 1;
  DNode<T> node = new DNode<T>(data);
  DNode<T> current = head;
```

```
        while (current.Next != null && k < pos)
        { k++; current = current.Next; }
        node.Next = current.Next;
        node.Prev = current;
        current.Next.Prev = node;
        current.Next = node;
        count++;
      }
      //Ikki bog‘lamli ro‘yxat bosh elementini o‘chirish
      public void DeleteFirst()
      {
        if (!IsEmpty)
        {
          head = head.Next;
          head.Prev = null;
          count--;
        }
        else
          Console.WriteLine("Ro‘yxat bo‘sh");
      }
      //Ikki  bog‘lamli  ro‘yxatning  oxirgi  elementini
//o‘chirish
      public void DeleteLast()
      {
        if (!IsEmpty)
        {
          tail = tail.Prev;
          tail.Next = null;
          count--;
        }
        else
         Console.WriteLine("Ro‘yxat bo‘sh");
      }
      // Ikki bog‘lamli ro‘yxatdan berilgan o‘rindagi
```

```
    //elementni o‘chirish
    public void Delete(int pos)
    {
      if (!IsEmpty)
      {
        int k = 1;
        DNode<T> current = head;
        while (current.Next != null && k < pos)
        {
          k++; current = current.Next;
        }
        if (k == 1)
        {
          head = head.Next;
          head.Prev = null;
        }
        else
        {
          current.Prev.Next = current.Next;
          current.Next.Prev = current.Prev;
        }
        count--;
      }
      else
        Console.WriteLine("Ro‘yxat bo‘sh");
    }
    // Ro‘yxatni tozalash
    public void Clear()
    {
      head = null; tail = null; count = 0;
    }
  }
}
```

Asosiy dastur:

```
using System;
namespace DoubleList
{
  class Program
  {
  static void Main(string[] args)
    {
      DList<string> linkedDList = new DList<string>();
      // Element qo’shish
      linkedDList.Add("Anna");
      linkedDList.Add("Anvar");
      linkedDList.Add("Rustam");
      linkedDList.Add("Oleg");
      linkedDList.Insert("Petr", 3);
      linkedDList.Print();
      // O’chirish
      linkedDList.Delete(3);
      linkedDList.DeleteFirst();
      linkedDList.DeleteLast();
      linkedDList.PrintBack();
    }
  }
}
```

3.1 – ilova. Statik siklik massiv asosidagi stek dasturi

```
using System;
using System.Collections.Generic;
using System.Text;
namespace ArrayStack
{
  public class ArrStack<T>
  {
    private T[] items; // stek elementlari
    private int count;  // elementlar soni
    // kelishuv bo‘yicha massivdagi elementlar soni
    const int n = 10;
    public ArrStack(int length = n)
    {
      items = new T[length];
    }
    // stekni bo‘shlikka tekshirish
    public bool IsEmpty
    {
      get { return count == 0; }
    }
    // stek o‘lchami
    public int Count
    {
      get { return count; }
    }
    // Stekka element qo‘shish
    public void Push(T item)
    {
      // agar stek to‘la bo‘lsa istisno holatini yuzaga
keltirish
      if (count == items.Length)
        throw    new    InvalidOperationException("Stek
to‘lgan");
```

```
      items[count++] = item;
    }
    // Stekdan elementni olish
    public T Pop()
    {
      // Agar stek bo‘sh bo‘lsa istisno holatini yuzaga
keltirish
      if (IsEmpty)
        throw    new    InvalidOperationException("Stek
bo‘sh");
      T item = items[--count];
      items[count] = default(T); //
      return item;
    }
    // Stek oxiridan element qaytarish
    public T Peek()
    {
      // Agar stek bo‘sh bo‘lsa istisno holatini yuzaga
keltirish
      if (IsEmpty)
        throw    new    InvalidOperationException("Stek
bo‘sh");
      return items[count - 1];
    }
    // Stek elementlarini konsol ekraniga chiqarish
    public void Print()
    {
      for (int i=0;i<count;i++)
      {
        Console.WriteLine(items[i]);
      }
    }
  }
}
```

Statik siklik massiv ko‘rinishida stekdan foydalanishga doir misol:

```
using System;
namespace ArrayStack
{
  class Program
  {
    static void Main(string[] args)
    {
      try
      {
        ArrStack<string> stack = new
ArrStack<string>(8);
        // 4 ta element qo‘shamiz
        stack.Push("Anvar");
        stack.Push("Rustam");
        stack.Push("Diana");
        stack.Push("Sattor");
        Console.WriteLine("Stekdagi elementlar
soni:{0}",stack.Count);
        Console.WriteLine("Stekdagi elementlar
ro‘yxati");
        stack.Print();
        Console.WriteLine("Stekdan olib tashlash");
        // Stekdan 1 ta elementni olib tashlaymiz
        var head = stack.Pop();
        Console.WriteLine(head);  // Sattor
        head = stack.Pop();
        Console.WriteLine(head);
        head = stack.Pop();
        Console.WriteLine(head);
```

```
        // Stekdan element olib tashlamasdan oxiridan
olish
        head = stack.Peek();
        Console.WriteLine(head);  // Diana
        Console.WriteLine("Stekdagi elementlar
ro‘yxati");
        stack.Print();
      }
      catch (InvalidOperationException ex)
      {
        Console.WriteLine(ex.Message);
      }
      Console.ReadKey();
    }
  }
}
```

3.2 – ilova. Dinamik siklik massiv asosidagi stek dasturi

```
using System;
using System.Collections.Generic;
using System.Text;
namespace DinArrayStack
{
  public class DArrStack<T>
  {
    private T[] items;
    private int count;
    const int n = 1;
    public DArrStack(int length = n)
    {
      items = new T[length];
    }
    public bool IsEmpty
    {
      get { return count == 0; }
    }
    public int Count
    {
      get { return count; }
    }
    public void Push(T item)
    {
      // Stekni ikkilantiramiz
      if (count == items.Length)
        Resize(items.Length * 2);
        items[count++] = item;
    }
    public T Pop()
    {
      // Agar stek bo‘sh bo‘lsa, istisno
      //yuzaga keltirish
```

```
      if (IsEmpty)
        throw    new    InvalidOperationException("Stek
bo‘sh");
      T item = items[--count];
      items[count]  =  default(T);//kelishuv  bo‘yicha
qiymat
      return item;
    }
    public T Peek()
    {
      return items[count - 1];
    }
    private void Resize(int max)
    {
      T[] tempItems = new T[max];
      for (int i = 0; i < count; i++)
        tempItems[i] = items[i];
      items = tempItems;
    }
    public void Print()
    {
      for(int i=0;i<count;i++)
      {
        Console.WriteLine(items[i]);
      }
    }
  }
}
```

3.3 - ilova: Ro‘yxat asosidagi stek dasturi

```
using System;
using System.Collections.Generic;
using System.Text;
namespace ListStack
{
  public class Node<T>
  {
    public Node(T data)
    {
      Data = data;
    }
    public T Data { get; set; }
    public Node<T> Next { get; set; }
  }
  public class LStack<T>
  {
    Node<T> head;
    int count;

    public bool IsEmpty
    {
      get { return count == 0; }
    }
    public int Count
    {
      get { return count; }
    }
    public void Push(T item)
    {
      // Stekni kattalashtiramiz
      Node<T> node = new Node<T>(item);
      node.Next = head; // stek uchini yangi elementga
qayta o‘rnatish
```

```
      head = node;
      count++;
    }
    public T Pop()
    {
      // Stek bo‘sh bo‘lsa istisno holatini yuzaga
keltirish
      if (IsEmpty)
        throw new InvalidOperationException("Stek
bo‘sh");
      Node<T> temp = head;
      head = head.Next; // stek uchini keyingi elementga
qayta o‘rnatish
      count--;
      return temp.Data;
    }
    public T Peek()
    {
      if (IsEmpty)
        throw new InvalidOperationException("Stek
bo‘sh");
      return head.Data;
    }
    public void Clear()
    {
      head = null;
      count = 0;
    }
    public bool Contains(T data)
    {
      Node<T> current = head;
      while (current != null)
      {
        if (current.Data.Equals(data))
```

```
            return true;
          current = current.Next;
        }
        return false;
      }
      public void Print()
      {
        Node<T> current = head;
        while (current != null)
        {
          Console.WriteLine(current.Data);
          current = current.Next;
        }
      }
    }
}
```

Ro‘yxat asosidagi stekdan foydalanishga misol:

```
using System;
namespace ListStack
{
  class Program
  {
    static void Main(string[] args)
    {
      try
      {
        LStack<string> stack = new LStack<string>();
        stack.Push("Petr");
        stack.Push("Alice");
        stack.Push("Rustam");
        stack.Push("Sarvar");
        stack.Print();
        Console.WriteLine();
        string header = stack.Peek();
```

```
            Console.WriteLine($"Stek uchi: {header}");
            Console.WriteLine();
            header = stack.Pop();
            stack.Print();
        }
        catch (InvalidOperationException ex)
        {
            Console.WriteLine(ex.Message);
        }
        Console.Read();
    }
  }
}
```

4.1 – ilova. Siklik massiv asosidagi navbat

```
using System;
using System.Collections.Generic;
using System.Text;
namespace ArrayQueue
{    // Statik siklik massiv orqali navbat BAT ni
tavsiflovchi sinf
  public class ArrQueue<T>
  {
    private int front; // navbat boshi
    private int back;  // navbat oxiri
    private T[] items; // elementlarni saqlash uchun
siklik massiv
    private int size;   // massivdagi elementlar soni
    public ArrQueue(int length = 10)
    {
      items = new T[length];
      front = -1;
      back = -1;
      size = length;
    }
    // navbat bo‘shmi
    public bool IsEmpty
    {
      get { return front == -1; }
    }
    // navbat to‘lami
    public bool IsFull
    {
      get { return (back + 1) % size == front; }
    }
  // navbatdagi elementar soni
    public int Count
    {
```

```
      get
      {
        return ((size - front + back + 1) % size);
      }
    }
    // Element qo‘shish
    public void EnQueue(T item)
    {
    //  Agar  navbat  to‘la  bo‘lsa,  istisno  yuzaga
keltiramiz
      if (IsFull)
        throw  new  InvalidOperationException("Navbat
to‘la");
      back = (back + 1) % size;
      items[back] = item;
      if (front == -1) front = back;
    }
    // Elementni olib tashlash
    public T DeQueue()
    {
    //  Agar  navbat  bo‘sh  bo‘lsa,  istisno  yuzaga
keltiramiz
      if (IsEmpty)
         throw  new  InvalidOperationException("Navbat
bo‘sh");
      T item = items[front];
      if (front == back) front = back = -1;
      else front = (front + 1) % size;
      return item;
    }
    // Birinchi elementni olish
    public T First
    {
      get
```

```
      {
        if (IsEmpty) throw new
InvalidOperationException("Navbat bo‘sh");
        return items[front];
      }
    }
    // Oxirgi elementni olish
    public T Last
    {
      get
      {
        if (IsEmpty) throw new
InvalidOperationException("Navbat bo‘sh");
        return items[back];
      }
    }
    // Navbat elementlarini chop qilish
    public void Print()
    {
      for (int i = front; i <= back; i++)
      {
        Console.WriteLine(items[i]);
      }
    }
  }
}
```

Navbatdan foydalanishga misol.

```
using System;
namespace ArrayQueue
{
  class Program
  {
    static void Main(string[] args)
```

```
    {
      try
      {
      ArrQueue<string> queue = new ArrQueue<string>();
        queue.EnQueue("Alim");
        queue.EnQueue("Salim");
        queue.EnQueue("Anvar");
        queue.EnQueue("Ziyoda");
        Console.WriteLine("Shakllangan navbat");
        queue.Print();
        Console.WriteLine();
        string firstItem = queue.DeQueue();
        Console.WriteLine($" {firstItem} elementni
olib tashlash:");
        Console.WriteLine();
        queue.Print();
        Console.WriteLine($"Birinchi element:
{queue.First}");
        Console.WriteLine($"Oxirgi element:
{queue.Last}");
      }
      catch (InvalidOperationException ex)
      {
        Console.WriteLine(ex.Message);
      }
      Console.ReadKey();
    }
  }
}
```

4.2 – ilova. Dinamik siklik massiv asosidagi navbat dasturi

```
using System;
using System.Collections.Generic;
using System.Text;
namespace DinArrayQueue
{
  // Dinamik siklik massiv asosida navbat BAT ni
tasvirlovchi sinf
  public class DArrQueue<T>
  {
    private int front; // navbat boshi
    private int back;  // navbat oxiri
    private T[] items; // navbat elementlari uchun
siklik massiv
    private int size;   //massivdagi elementlar soni
    public DArrQueue(int length = 10)
    {
      items = new T[length];
      front = -1;
      back = -1;
      size = length;
    }
    // Navbatni bo‘shlikka tekshirish
    public bool IsEmpty
    {
      get { return front == -1; }
    }
    // Navbatni to‘liqlikka tekshirish
    public bool IsFull
    {
      get { return (back + 1) % size == front; }
    }
    // Navbatdagi elementlar soni
    public int Count
```

```
    {
      get
      {
        return ((size - front + back + 1) % size);
      }
    }
    // Element qo‘shish
    public void EnQueue(T item)
    {
      // Agar navbat to‘lib ketsa, istisno yuzaga
keltirish
      if(IsFull) Resize();
      back = (back + 1) % size;
      items[back] = item;
      if (front == -1) front = back;
    }
    // Elementni olish tashlash
    public T DeQueue()
    {
      // Agar navbat bo‘sh bo‘lsa, istisno yuzaga
keltirish
      if (IsEmpty)
        throw new InvalidOperationException("Navbat
bo‘sh");
      T item = items[front];
      if (front == back) front = back = -1;
      else front = (front + 1) % size;
      return item;
    }
    // Birinchi elementni olish
    public T First
    {
      get
      {
```

```
        if (IsEmpty) throw new
InvalidOperationException("Navbat bo‘sh");
        return items[front];
      }
    }
    // Oxirgi elementni olish
    public T Last
    {
      get
      {
        if (IsEmpty) throw new
InvalidOperationException("Navbat bo‘sh");
        return items[back];
      }
    }
    // Navbat elementlarini chop qilish
    public void Print()
    {
      for (int i = front; i <= back; i++)
      {
        Console.WriteLine(items[i]);
      }
    }
    private void Resize()
    {
      size *= 2;
      T[] tempItems = new T[size];
      for (int i = 0; i < Count; i++)
        tempItems[i] = items[i];
      items = tempItems;
    }
  }
}
```

Dinamik siklik massiv ko‘rinishidagi navbatdan foydalanishga misol:

```
using System;
namespace DinArrayQueue
{
  class Program
  {
    static void Main(string[] args)
    {
      try
      {
        DArrQueue<string> queue = new
DArrQueue<string>(3);
        queue.EnQueue("Alim");
        queue.EnQueue("Salim");
        queue.EnQueue("Anvar");
        queue.EnQueue("Ziyoda");
        Console.WriteLine("Shakllantirilgan navbat");
        queue.Print();
        Console.WriteLine();
        string firstItem = queue.DeQueue();
        Console.WriteLine($" {firstItem} elementni
olish:");
        Console.WriteLine();
        queue.Print();
        Console.WriteLine($"Birinchi element:
{queue.First}");
        Console.WriteLine($"Oxirgi element:
{queue.Last}");
      }
      catch (InvalidOperationException ex)
      {
        Console.WriteLine(ex.Message);
      }
      Console.ReadKey();} } }
```

4.3 – ilova. Ro‘yxat bo‘yicha navbatni amalga oshirish dasturi

```
using System;
using System.Collections.Generic;
using System.Text;
namespace ListQueue
{
  public class Node<T>
  {
    public Node(T data)
    {
      Data = data;
    }
    public T Data { get; set; }
    public Node<T> Next { get; set; }
  }
  public class LQueue<T>
  {
    Node<T> head; // bosh/birinchi element
    Node<T> tail; // oxirgi/dum element
    int count;
    // Navbatga qo‘shish
    public void EnQueue(T data)
    {
      Node<T> node = new Node<T>(data);
      Node<T> temp = tail;
      tail = node;
      if (count == 0)
        head = tail;
      else
        temp.Next = tail;
      count++;
    }
    // Navbatdan o‘chirish
    public T DeQueue()
```

```
{
  if (count == 0)
    throw new InvalidOperationException();
  T output = head.Data;
  head = head.Next;
  count--;
  return output;
}
// Birinchi elementni olish
public T First
{
  get
  {
    if (IsEmpty)
      throw new InvalidOperationException();
    return head.Data;
  }
}
// oxirgi elementni olish
public T Last
{
  get
  {
    if (IsEmpty)
      throw new InvalidOperationException();
    return tail.Data;
  }
}
public int Count { get { return count; } }
public bool IsEmpty { get { return count == 0; } }
public void Clear()
{
  head = null;
  tail = null;
```

```
      count = 0;
    }
    public bool Contains(T data)
    {
      Node<T> current = head;
      while (current != null)
      {
        if (current.Data.Equals(data))
          return true;
        current = current.Next;
      }
      return false;
    }
    public void Print()
    {
      Node<T> current = head;
      while (current != null)
      {
        Console.WriteLine(current.Data);
      }
    }
  }
```

Ro‘yxat asosidagi navbatlardan foydalanishga misol:

```
using System;
namespace ListQueue
{
  class Program
  {
    static void Main(string[] args)
    {
      try
      {
        LQueue<string> queue = new LQueue<string>();
```

```
            queue.EnQueue("Alim");
            queue.EnQueue("Salim");
            queue.EnQueue("Anvar");
            queue.EnQueue("Ziyoda");
            Console.WriteLine("Shakllantirilgan navbat");
            queue.Print();
            Console.WriteLine();
            string firstItem = queue.DeQueue();
            Console.WriteLine($" {firstItem} Elementni
olib tashlash: ");
            Console.WriteLine();
            queue.Print();
            Console.WriteLine($"Birinchi element:
{queue.First}");
            Console.WriteLine($"Oxirgi element:
{queue.Last}");
        }
        catch (InvalidOperationException ex)
        {
            Console.WriteLine(ex.Message);
        }
        Console.ReadKey();
    }
  }
}
```

5 -ilova. Dek

```
using System;
using System.Collections.Generic;
using System.Text;
namespace DoubleQueue
{
  public class DNode<T>
  {
    public DNode(T data)
    {
      Data = data;
    }
    public T Data { get; set; }
    public DNode<T> Prev { get; set; }
    public DNode<T> Next { get; set; }
  }

  public class Deque<T> // ikki bog‘lamli dek sinfi
  {
    DNode<T> head; // asos/birinchi element
    DNode<T> tail; // oxirgi/dim element
    int count;  // ro‘yxatdagi elementlar soni
    // Element qo‘shish
    public void Push_Back(T data)
    {
      DNode<T> node = new DNode<T>(data);
      if (head == null)
        head = node;
      else
      {
        tail.Next = node;
        node.Prev = tail;
      }
      tail = node;
```

```
      count++;
    }
    public void Push_Front(T data)
    {
      DNode<T> node = new DNode<T>(data);
      DNode<T> temp = head;
      node.Next = temp;
      head = node;
      if (count == 0)
        tail = head;
      else
        temp.Prev = node;
      count++;
    }
    public T Pop_Front()
    {
      if (count == 0)
        throw    new    InvalidOperationException("Dek
bo‘sh");
      T output = head.Data;
      if (count == 1)
      {
        head = tail = null;
      }
      else
      {
        head = head.Next;
        head.Prev = null;
      }
      count--;
      return output;
    }
    public T Pop_Back()
    {
```

```
      if (count == 0)
        throw     new     InvalidOperationException("Dek
bo‘sh");
      T output = tail.Data;
      if (count == 1)
      {
        head = tail = null;
      }
      else
      {
        tail = tail.Prev;
        tail.Next = null;
      }
      count--;
      return output;
    }
    public T First
    {
      get
      {
        if (IsEmpty)
          throw    new    InvalidOperationException("Dek
bo‘sh");
        return head.Data;
      }
    }
    public T Last
    {
      get
      {
        if (IsEmpty)
          throw    new    InvalidOperationException("Dek
bo‘sh");
        return tail.Data;
```

```
      }
    }
    public int Count { get { return count; } }
    public bool IsEmpty { get { return count == 0; } }
    public void Clear()
    {
      head = null;
      tail = null;
      count = 0;
    }
    public bool Contains(T data)
    {
      DNode<T> current = head;
      while (current != null)
      {
        if (current.Data.Equals(data))
          return true;
        current = current.Next;
      }
      return false;
    }
    public void Print()
    {
      if (IsEmpty)
        throw    new    InvalidOperationException("Dek
bo‘sh");
      DNode<T> current = head;
      while (current != null)
      {
        Console.WriteLine(current.Data);
        current = current.Next;
      }
    }
  }
```

```
}
```

Dekni qo‘llanilishi

```
using System;
namespace DoubleQueue
{
  class Program
  {
    static void Main(string[] args)
    {
      try
      {
        Deque<string> deck = new Deque<string>();
        deck.Push_Front("Andrey");
        deck.Push_Front("Katya");
        deck.Push_Front("Vanya");
        deck.Push_Back("Fedor");
        deck.Print();
        string removedItem = deck.Pop_Front();
        Console.WriteLine("\n  O‘chirildi:  {0}  \n",
removedItem);
        removedItem = deck.Pop_Back();
        Console.WriteLine("\n  O‘chirildi:  {0}  \n",
removedItem);
        deck.Print();
      }
      catch (InvalidOperationException ex)
      {
        Console.WriteLine(ex.Message);
      }
      Console.Read();
    }
  }
}
```

6.1- ilova. Halqasimon bir bog‘lamli ro‘yxat

```
using System;
using System.Collections.Generic;
using System.Text;
namespace CircleList
{
  public class Node<T>
  {
    public Node(T data)
    {
      Data = data;
    }
    public T Data { get; set; }
    public Node<T> Next { get; set; }
  }
    // Halqasimon bog‘langan ro‘yxat
  class CList<T>
  {

      Node<T> head; // bosh/birinchi element
      Node<T> tail; // dum/oxirgi element
      int count;  // ro‘yxatdagi elementlar soni
      // element qo‘shish
      public void Add(T data)
      {
        Node<T> node = new Node<T>(data);
        // agar element bo‘sh bo‘lsa
        if (head == null)
        {
          head = node;
          tail = node;
          tail.Next = head;
        }
        else
```

```
    {
      node.Next = head;
      tail.Next = node;
      tail = node;
    }
    count++;
  }
  public bool Remove(T data)
  {
    Node<T> current = head;
    Node<T> prev = null;
    if (IsEmpty) return false;
    do
    {
      if (current.Data.Equals(data))
      {
       // agar tugun o‘rtada yoki oxirda bo‘lsa
        if (prev != null)
        {
          prev.Next = current.Next;
          if (current == tail)
            tail = prev;
        }
        else//agar birinchi element o‘chirilsa
        {
//agar ro‘yxatda faqat bitta element bo‘lsa
          if (count == 1)
          {
            head = tail = null;
          }
          else
          {
            head = current.Next;
            tail.Next = current.Next;
```

```
            }
          }
          count--;
          return true;
        }
        prev = current;
        current = current.Next;
      } while (current != head);
      return false;
    }
    public int Count { get { return count; } }
    public bool IsEmpty { get { return count == 0; }
}
    public void Clear()
    {
      head = null;
      tail = null;
      count = 0;
    }
    public bool Contains(T data)
    {
      Node<T> current = head;
      if (current == null) return false;
      do
      {
        if (current.Data.Equals(data))
          return true;
        current = current.Next;
      }
      while (current != head);
      return false;
    }
    public void Print()
    {
```

```
        Node<T> current = head;
        do
        {
          Console.WriteLine(current.Data);
          current = current.Next;
        }
        while (current != head);
      }
    }
  }
```

Asosiy dastur:

```
using System;
namespace CircleList
{
  class Program
  {
    static void Main(string[] args)
    {
      CList<string> cList = new CList<string>();
      cList.Add("Amir");
      cList.Add("Rustam");
      cList.Add("Said");
      cList.Add("Botir");
      cList.Add("Guzal");
      cList.Print();
      cList.Remove("Rustam");
      Console.WriteLine("\n O‘chirishdan keyin: \n");
      cList.Print();
      Console.Read();
    }
  }
}
```

6.2 – ilova. Halqasimon ikki bog‘lamli ro‘yxat

```
using System;
using System.Collections.Generic;
using System.Text;
namespace CircularDoubleList
{
  public class DNode<T>
  {
    public DNode(T data)
    {
      Data = data;
    }
    public T Data { get; set; }
    public DNode<T> Prev { get; set; }
    public DNode<T> Next { get; set; }
  }

  class CDList<T>
  {
    DNode<T> head; // bosh/birinchi element
    int count;  //ro‘yxatdagi elementlar soni
    // element qo‘shish
    public void Add(T data)
    {
      DNode<T> node = new DNode<T>(data);
      if (head == null)
      {
        head = node;
        head.Next = node;
        head.Prev = node;
      }
      else
      {
        node.Prev = head.Prev;
```

```
    node.Next = head;
    head.Prev.Next = node;
    head.Prev = node;
  }
  count++;
}
// elementni o‘chirish
public bool Remove(T data)
{
  DNode<T> current = head;
  DNode<T> removedItem = null;
  if (count == 0) return false;
  // o‘chiriladigan tugunni qidirish
  do
  {
    if (current.Data.Equals(data))
    {
      removedItem = current;
      break;
    }
    current = current.Next;
  }
  while (current != head);
  if (removedItem != null)
  {
  //agar ro‘yxatning yagona elementi o‘chirilsa
    if (count == 1)
      head = null;
    else
    {
      // agar birinchi elementi o‘chirilsa
      if (removedItem == head)
      {
        head = head.Next;
```

```
        }
        removedItem.Prev.Next = removedItem.Next;
        removedItem.Next.Prev = removedItem.Prev;
      }
      count--;
      return true;
    }
    return false;
  }
  public int Count { get { return count; } }
  public bool IsEmpty { get { return count == 0; } }
  public void Clear()
  {
    head = null;
    count = 0;
  }
  public bool Contains(T data)
  {
    DNode<T> current = head;
    if (current == null) return false;
    do
    {
      if (current.Data.Equals(data))
        return true;
      current = current.Next;
    }
    while (current != head);
    return false;
  }
  public void Print()
  {
    DNode<T> current = head;
    do
    {
```

```
            Console.WriteLine(current.Data);
            current = current.Next;
        }
        while (current != head);
    }
  }
}
```

Halqasimon ikki bog‘lamli ro‘yxatning qo‘llanilishiga misol:

```
using System;
namespace CircularDoubleList
{
  class Program
  {
    static void Main(string[] args)
    {
      CDList<string> cDList = new CDList<string>();
      cDList.Add("Amir");
      cDList.Add("Rustam");
      cDList.Add("Said");
      cDList.Add("Guzal");
      cDList.Print();
      cDList.Remove("Rustam");
      Console.WriteLine("\n O‘chirilgandan keyin: \n");
      cDList.Print();
      Console.Read();
    }
  }
}
```

6.3 - ilova. Ustuvor ro‘yxat

```
using System;
using System.Collections.Generic;
using System.Text;
namespace ListPQueue
{
  public class Node<T>
  {
    public Node(T data)
    {
      Data = data;
    }
    public T Data { get; set; }
    public Node<T> Next { get; set; }
  }
  public class LPQueue<T> where T:IComparable
  {
    Node<T> head; // Bosh/birinchi element
    int count;
    public int Count { get { return count; } }
    public bool IsEmpty { get { return count == 0; } }
    // Navbatga qo‘shish
    public void Insert(T data)
    {
      Node<T> node = new Node<T>(data);
      if (IsEmpty)  head = node;
      else {
        Node<T> current = head;
        Node<T> prev = null;
        while (current != null &&
current.Data.CompareTo(data) > 0)
        {
          prev = current;
          current = current.Next;
```

```
        }
        if (prev == null)
        {
          node.Next = current;
          head = node;
        }
        else {
          node.Next = current;
          prev.Next = node;
        }
      }
      count++;
    }
    // Navbatdan olib tashlash (o‘chirish)
    public T DeLeteMax()
    {
      if (IsEmpty)
        throw   new   InvalidOperationException("Navbat
bo‘sh");
      T output = head.Data;
      head = head.Next;
      count--;
      return output;
    }

    // Birinchi elementni olish
    public T GetMaximum
    {
      get
      {
        if (IsEmpty)
          throw  new  InvalidOperationException("Navbat
bo‘sh");
        return head.Data;
```

```
      }
    }
    public void Clear()
    {
      head = null;
      count = 0;
    }
    public bool Contains(T data)
    {
      Node<T> current = head;
      while (current != null)
      {
        if (current.Data.Equals(data))
          return true;
        current = current.Next;
      }
      return false;
    }
    public void Print()
    {
      Console.WriteLine();
      Node<T> current = head;
      while (current != null)
      {
        Console.Write($"{current.Data}  ");
        current = current.Next;
      }
      Console.WriteLine();
    }
  }
}
```

Ustuvor ro‘yxatlardan foydalanishga misol:

```
using System;
namespace ListPQueue
{
  class Program
  {
    static void Main(string[] args)
    {
      try
      {
        LPQueue<int> pqueue = new LPQueue<int>();
        Random rnd = new Random();
        Console.Write("Elementlar sonini kiriting: ");
        int n = Int32.Parse(Console.ReadLine());
        for (int i = 0; i < n; i++)
        {
          pqueue.Insert(rnd.Next(0, 100));
        }
        Console.WriteLine("Shakllangan navbat");
        pqueue.Print();
        Console.WriteLine();
        int firstItem = pqueue.DeLeteMax();
        Console.WriteLine($"Maksimal          elementni
chiqaramiz: {firstItem}");
        Console.WriteLine();
        pqueue.Print();
        Console.WriteLine($"Navbatning         maksiman
elementi: {pqueue.GetMaximum}");
      }
      catch (InvalidOperationException ex)
      {
        Console.WriteLine(ex.Message);
      }
      Console.ReadKey();
    }   } }
```

7.1 - ilova. Anagramma

```
using System;
using System.Text;
namespace Anagram
{
  class Program
  {
    static void Main(string[] args)
    {
      char[] arrChar = new char[100];
      StringBuilder str;
      int size;
      int count;
      void Anagram(int newSize)
      {

        if (newSize == 1) { return; }
        for (int j = 0; j < newSize; j++)
        {
          Anagram(newSize - 1);
          if (newSize == 2)
          Console.WriteLine($"Variant:N{++count}  so‘z:
{str} ");
          rotate(newSize);
        }
      }
      void rotate(int newSize)
      {
        int j;
        int position = size - newSize;
        char temp = str[position];
        for (j = position + 1; j < size; j++)
          str[j - 1] = str[j];
        str[j - 1] = temp;
```

```
      }
      Console.WriteLine("Satrni kiriting");
      str = new StringBuilder();
      str.Append(Console.ReadLine());
      size = str.Length;
      count = 0;
      Anagram(size);
      Console.ReadKey();
    }
  }
}
```

7.2 – ilova. Butun sonlarning tartiblangan massividan kalitni qidirish

```
using System;
namespace RecFind
{
  class Program
  {
    static void Main(string[] args)
    {
      int recFind(int [] a,int Key, int lowerBound, int
upperBound)
      {
        int curIn;
        curIn = (lowerBound + upperBound) / 2;
        if (a[curIn] == Key) return curIn;
        if (lowerBound > upperBound) return -1;
        if (a[curIn] < Key) // Yuqori yarmida
          return recFind(a,Key, curIn + 1, upperBound);
        else
          return recFind(a,Key, lowerBound, curIn - 1);
```

```
        }
        Console.WriteLine("Massivni kiriting:");
        var                         p                         =
Array.ConvertAll(Console.ReadLine().Split(new[]  {  '  '
},    StringSplitOptions.RemoveEmptyEntries),    s    =>
int.Parse(s));
        Console.WriteLine("Qidirilayotgan            sonni
kiriting:");
        int key = int.Parse(Console.ReadLine());
        int ind;
        ind = recFind(p,key,0, p.Length-1);
        if          (ind>=0)          Console.WriteLine($"
p[{ind}]={key}");

        else Console.WriteLine($" {key} : qiymat massivda
yo‘q");
        Console.ReadKey();
    }
  }
}
```

7.3 – ilova. Hanoy minorasi

```
using System;
namespace Hanoy
{
  class Program
  {
    static void Main(string[] args)
    {
      int count = 0;
      void Hanoy(int n, char from, char inter, char to)
      {
        if (n == 1)
          Console.WriteLine($"Qadam {++count} disk 1 ni
{from} dan {to} ga o‘tkazish");
        else
        {
          Hanoy(n - 1, from, to, inter); // from-->inter
           Console.WriteLine($"Qadam {++count} Disk {n}
dan {from}  {to} ga o‘tkazish");
          Hanoy(n - 1, inter, from, to); // inter-->to
        }
      }
      Console.WriteLine("Disklar sonini kiriting:");
      int nDisks = int.Parse(Console.ReadLine());
      Hanoy(nDisks, 'A', 'B', 'C');
      Console.ReadKey();

    }
  }
}
```

8 – ilova. Binar daraxt

```
using System;
using System.Collections.Generic;
using System.Text;
namespace BinaryTree
{
  // Ota-onaga nisbatan tugun joylari
  public enum Side
  {
    Left,
    Right
  }
  public class BTNode<T> where T : IComparable
  {
    // Binar daraxt tuguni
    // Konstruktor
    public BTNode(T data)
    {
        Data = data;
    }
    // Tugunda saqlanadigan berilganlar
    public T Data { get; set; }
      // Chap shoxi
    public BTNode<T> Left { get; set; }
      // O‘ng shoxi
    public BTNode<T> Right { get; set; }
      // Ota-onasi
    public BTNode<T> Parent { get; set; }
      // Tugunning ota-onasiga nisbatan joylashuvi
       Parent == null? (Side?)null : Parent.Left == this
          ? Side.Left : Side.Right;
    // Sinf ekzamplyarini satrga aylantirish
    public    override    string    ToString()    =>
Data.ToString();
```

```
  }
}

using System;
using System.Collections.Generic;
namespace BinaryTree
{
  // Binar daraxtni aniqlovchi sinf
  public class BinaryTree<T> where T : IComparable
  {
    // Binar daraxt ildizi
    public BTNode<T> Root { get; set; }
    // Usul tavsifi
    public BTNode<T> Find(T key)
    {
      // berilgan kalitli tugun qidirish
      var temp = Root; // Ildiz tugundan boshlash
      while (temp != null)
      {
        int result = key.CompareTo(temp.Data);
        if (result == 0) break;
        if (result < 0) // Chapga yurish?
          temp = temp.Left;
        else temp = temp.Right; // yoki o‘nga?
      }
      return temp;
    }
// Ikkilik qidiruv daraxtiga element qo‘shish
    public void Add(T data)
    {
      // Yangi tugun yaratish
      var newNode = new BTNode<T>(data);
      // Ildiz tugun mavjud emas
      if (Root == null) Root = newNode;
```

```
        else
        {
          // Ildiz tugundan boshlash
          var temp = Root;
          while (true)
          {
            // (sikldan ichki chiqish)
            var parent = temp;
            int result = data.CompareTo(temp.Data);
            if (result == 0) return;
            if (result < 0)
            {
              temp = temp.Left;
// Agar zanjir oxiriga etib borilsa, chapga qo‘yish
              if (temp == null)
              {
                parent.Left = newNode;
                newNode.Parent = parent;
                return;
              }
            }
            else
            { //Yoki o‘nga?
              temp = temp.Right;
// Agar zanjir oxiriga etib borilsa, o‘nga qo‘yish
              if (temp == null)
              {
                parent.Right = newNode;
                newNode.Parent = parent;
                return;
              }
            }
          }
        }
```

```
    }
// node qism daraxtni currnode qism daraxtga qo‘yishning
//rekursiv usuli
    public  void  AddNode(BTNode<T>  node,  BTNode<T>
currentNode = null)
    {
      if (Root == null)
      {
        node.Parent = null;
        Root = node;
      }
      currentNode = currentNode ?? Root;
      node.Parent = currentNode;
      int                    result                    =
node.Data.CompareTo(currentNode.Data);
      if (result == 0) return;
      if (result < 0)
      {
        if (currentNode.Left == null)
        {
          currentNode.Left = node;
        }
        else
        {
          AddNode(node, currentNode.Left);
        }
      }
      else
      {
        if (currentNode.Right == null)
        {
          currentNode.Right = node;
        }
        else
```

```
    {
      AddNode(node, currentNode.Right);
    }
  }
}
//Bi qidiruv daraxtidan tugunni o‘chirish
//usuli ko‘rinishi
public void Remove(T data)
{
  BTNode<T> node = Find(data);
  if (node == null) { return; }
  //agar tugunda ajdodlar mavjud bo‘lmasa,
  //uni o‘chirish mumkin
  if (node.Left == null && node.Right == null)
  {
    if (node.NodeSide == Side.Left)
    {
      node.Parent.Left = null;
    }
    else
    {
      node.Parent.Right = null;
    }
    return;
  }
  //agar chap yo‘q bo‘lsa, o‘ngdagini
  //o‘chiriladiganning o‘rniga qo‘yish
  if (node.Left == null)
  {
    if (node.NodeSide == Side.Left)
    {
      node.Parent.Left = node.Right;
    }
    else
```

```
    {
      node.Parent.Right = node.Right;
    }
    node.Right.Parent = node.Parent;
    return;
  }
  // agar o‘ng yo‘q bo‘lsa, chapdagini
  //o‘chiriladiganning o‘rniga qo‘yish
  if (node.Right == null)
  {
    if (node.NodeSide == Side.Left)
    {
      node.Parent.Left = node.Left;
    }
    else
    {
      node.Parent.Right = node.Left;
    }
    node.Left.Parent = node.Parent;
    return;
  }
  //agar ikki avlod ham mavjud bo‘lsa
  //o‘ngi o‘chiriladigan o‘rniga bo‘ladi
  //chap esa o‘ngni o‘rniga qo‘yiladi
  switch (node.NodeSide)
  {
    case Side.Left:
      node.Parent.Left = node.Right;
      node.Right.Parent = node.Parent;
      AddNode(node.Left, node.Right);
      break;
    case Side.Right:
      node.Parent.Right = node.Right;
      node.Right.Parent = node.Parent;
```

```
            AddNode(node.Left, node.Right);
            break;
          default:
            BTNode<T> tmp = node.Left;
            node.Data = node.Right.Data;
            node.Right = node.Right.Right;
            node.Left = node.Right.Left;
            AddNode(tmp, node);
            break;
        }
        return;
      }
      // Binar daraxt chiqarilishi
      public void Print()
      {
        PrintPreRec(Root);
         // PrintPreNoRec();
         // PrintInfRec(Root);
         // PrintPosRec(Root);
         // PrintPosNoRec();
        // PrintBFS();
      }
      private   void   PrintPreRec(BTNode<T>   startNode,
string indent = " ", Side? side = null)
      {
        if (startNode != null)
        {
          var nodeSide = side == null ? "+" : side ==
Side.Left ? "L" : "R";
          Console.WriteLine($"{indent}      [{nodeSide}]-
{startNode.Data}");
          indent += new string(' ', 5);
          // chap va o‘ng novdalarini rekursiv chaqirish
          PrintPreRec(startNode.Left, indent, Side.Left);
```

```
        PrintPreRec(startNode.Right,            indent,
Side.Right);
      }
    }
    private void SumOdd(BTNode<T> startNode, Side? side
= null)
    {
      if (startNode != null)
      {
        var nodeSide = side == null ? "+" : side ==
Side.Left ? "L" : "R";
        Console.WriteLine($"{indent}      [{nodeSide}]-
{startNode.Data}");
        indent += new string(' ', 5);
        // chap va o‘ng novdalarini rekursiv chaqirish
        PrintPreRec(startNode.Left, indent, Side.Left);
        PrintPreRec(startNode.Right,            indent,
Side.Right);
      }
    }
    // O‘tishning prefi shakli
    // Chiqarishning norekursiv usuli
    public void PrintPreNoRec()
    {
      string indent = " ";
      Side? side = null;
      Stack<BTNode<T>> S = new Stack<BTNode<T>>();
      Stack<string> Sin = new Stack<string>();
      BTNode<T> temp = Root;
      while (true)
      {
        while (temp != null)
        {
          //Joriy tugunni qayta ishlaymiz
```

```
            side = temp.NodeSide;
            var nodeSide = side == null ? "+" : side ==
Side.Left ? "L" : "R";
            Console.WriteLine($"{indent}    [{nodeSide}]-
{temp.Data}");
            indent += new string(' ', 5);
            S.Push(temp);
            Sin.Push(indent);
            temp=temp.Left;
          }
          if (S.Count == 0) break;
          indent = Sin.Pop();
          temp = S.Pop();
          temp = temp.Right;
        }
      }
      // Daraxt bo‘ylab o‘tishning infiks shakli
      // Chiqarishning infiks shakli
      private   void   PrintInfRec(BTNode<T>   startNode,
string indent = " ", Side? side = null)
      {
        if (startNode != null)
        {
          PrintInfRec(startNode.Left, indent, Side.Left);
          var nodeSide = side == null ? "+" : side ==
Side.Left ? "L" : "R";
          Console.WriteLine($"{indent}      [{nodeSide}]-
{startNode.Data}");
          indent += new string(' ', 5);
          PrintInfRec(startNode.Right,            indent,
Side.Right);
        }
      }
      // daraxt bo‘ylab o‘tishning infiks tartibi
```

```
    // Chiqarishning norekursiv usuli
    public void PrintInfNoRec()
    {
      string indent = " ";
      Side? side = null;
      Stack<BTNode<T>> S = new Stack<BTNode<T>>();
      Stack<string> Sin = new Stack<string>();
      BTNode<T> temp = Root;
      while (true)
      {
        while (temp != null)
        {
          S.Push(temp);
          Sin.Push(indent);
          temp = temp.Left;
        }
        if (S.Count == 0) break;
        indent = Sin.Pop();
        temp = S.Pop();
        //Joriy tugunni qayta ishlash
        side = temp.NodeSide;
        var nodeSide = side == null ? "+" : side ==
Side.Left ? "L" : "R";
        Console.WriteLine($"{indent}      [{nodeSide}]-
{temp.Data}");
        indent += new string(' ', 5);
        temp = temp.Right;
      }
    }
    // daraxt bo‘ylab o‘tishning postfiks tartibi
    // Rekursiv chiqarish usuli
    private   void   PrintPosRec(BTNode<T>   startNode,
string indent = " ", Side? side = null)
    {
```

```
      if (startNode != null)
      {
        PrintPosRec(startNode.Left, indent, Side.Left);
        PrintPosRec(startNode.Right,            indent,
Side.Right);
        var nodeSide = side == null ? "+" : side ==
Side.Left ? "L" : "R";
        Console.WriteLine($"{indent}      [{nodeSide}]-
{startNode.Data}");
        indent += new string(' ', 5);
      }
    }
    // daraxt bo‘ylab o‘tishning postfiks tartibi
    // Norekursiv chiqarish usuli
    public void PrintPosNoRec()
    {
      string indent = " ";
      Side? side = null;
      Stack<BTNode<T>> S = new Stack<BTNode<T>>();
      Stack<string> Sin = new Stack<string>();
      BTNode<T> temp = Root;
      BTNode<T> prev = Root;
      do
      {
        while (temp != null)
        {
          S.Push(temp);
          Sin.Push(indent);
          temp = temp.Left;
        }
        while (temp  == null && S.Count >0)
        {
          temp = S.Peek();
          if (temp.Right == null || temp.Right == prev)
```

```
            {
              //Joriy tugunni qayta ishlash
              side = temp.NodeSide;
              var nodeSide = side == null ? "+" : side ==
Side.Left ? "L" : "R";
              Console.WriteLine($"{indent}  [{nodeSide}]-
{temp.Data}");
              indent += new string(' ', 5);
              S.Pop(); prev = temp; temp= null;
            }
            else temp = temp.Right;
          }
        } while (S.Count != 0);
    }
    // Daraxt kengligi bo‘yicha o‘tish
    // Norekursiv chiqarish usuli
    public void PrintBFS()
    {
      BTNode<T> temp;
      string indent = " ";
      Side? side = null;
      Queue<BTNode<T>> Q = new Queue<BTNode<T>>();
      Queue<string> Qin = new Queue<string>();
      Qin.Enqueue(indent);
      Q.Enqueue(Root);
      while (Q.Count > 0)
      {
        temp = Q.Dequeue();
        indent = Qin.Dequeue();
        side = temp.NodeSide;
        var nodeSide = side == null ? "+" : side ==
Side.Left ? "L" : "R";
        Console.WriteLine($"{indent}      [{nodeSide}]-
{temp.Data}");
```

```
        indent += "    ";
        if (temp.Left != null) { Q.Enqueue(temp.Left);
Qin.Enqueue(indent); }
        if (temp.Right != null) { Q.Enqueue(temp.Right);
Qin.Enqueue(indent); }
      }
    }
  }
}
using System;
namespace BinaryTree
{
  class Program
  {
    static void Main(string[] args)
    {
      var binaryTree = new BinaryTree<int>();
      var rand = new Random();
      int i, j,k, n=10;
      /*binaryTree.Add(8);
      binaryTree.Add(3);
      binaryTree.Add(10);
      binaryTree.Add(1);
      binaryTree.Add(6);
      binaryTree.Add(4);
      binaryTree.Add(7);
      binaryTree.Add(14);
      binaryTree.Add(16);
      */
      i = 0;
      while (i < n)
      {
        k= rand.Next(0, 100);
        binaryTree.Add(k);
```

```
            Console.Write("   " + k);
            i++;
        }
        Console.WriteLine();
        binaryTree.Print();
        //  Console.WriteLine(binaryTree.Find(45).Data);
        Console.WriteLine(new string('-', 40));
        binaryTree.Remove(8);
        binaryTree.Print();
        Console.WriteLine(new string('-', 40));
        binaryTree.Remove(3);
        binaryTree.Print();
        Console.ReadLine();
    }
  }
}
```

9 – ilova. Heap sinf dasturi

```
using System;
using System.Collections.Generic;
using System.Text;
namespace HeapSort
{
  class  Heap<T> where T: IComparable
  {
    T [] arr;// tugunlardagi qiymatlarni saqlash uchun
// massiv
    int count;// uyum elementlari soni
    int size; // uyumdagi massiv o‘lchami
    public int Count { get { return count; } }
    public bool IsEmpty { get { return count == 0; } }
    public Heap(int n=10)
    {
      count = 0;
      size = n;
      arr = new T[size];
    }
    public Heap(T []A)
    {
      count = A.Length;
      size = count;
      arr = new T[count];
      for (int i = 0; i < size; i++) arr[i] = A[i];
      for (int i = (count - 2) / 2; i >= 0; i--)
      {

        Heapify(i);
      }
    }
    public T GetMaximum()
    {
```

```
      if (IsEmpty)
        throw    new    InvalidOperationException("Uyum
bo‘sh");
      return arr[0];
    }
    public T DeleteMax()
    {
      T data;
      if (IsEmpty)
        throw    new    InvalidOperationException("Uyum
bo‘sh");
      data = arr[0];
      arr[0] = arr[count - 1];
      Heapify(0);
      count--;
      return data;
    }
    public void Insert(T data)
    {
      int i=count;
      if (count == size)Resize();
      count++;
      while (i > 0 && data.CompareTo(arr[(i - 1) / 2])>0)
      {
        arr[i] = arr[(i - 1) / 2];
        i = (i - 1) / 2;
      }
      arr[i] = data;
    }
    public void Print()
    {
      Console.WriteLine();
      for (int i = 0; i < count; i++)
        Console.Write($"{arr[i]}  ");
```

```
  Console.WriteLine();
}
public void Clear()
{
  count = 0;
  size = 1;
  arr = new T[size];
}
public void HeapSort()
{
  T tmp;
  int n = count;
  for (int i = count- 1; i>0; i--)
  {
    tmp = arr[0];
    arr[0] = arr[count - 1];
    arr[count - 1] = tmp;
    count--;
    Heapify(0);
  }
  count = n;
}
private void Resize()
{
  T[] tmp = arr;
  size *= 2;
  arr = new T[size];
  for (int i = 0; i < count; i++) arr[i] = tmp[i];
}
private void Heapify(int i)
{
  int l, r,max;
  T tmp;
  l = 2 * i + 1;
```

```
      r = 2 * i + 2;
      max =(l < count && arr[l].CompareTo(arr[i])>0) ?
l:i;
      if (r < count && arr[r].CompareTo(arr[max])>0) max
= r;

      if (max != i)
      {
        tmp = arr[i];
        arr[i] = arr[max];
        arr[max] = tmp;
        Heapify(max);
      }
    }
  }
}
```

Uyumlarni qo‘llanilishiga misol:

```
using System;
namespace HeapSort
{
  class Program
  {
    static void Main(string[] args)
    {
      // massivni tartiblash funksiyasi
      void HeapSortMax()
      {
        int[] mas;
        Random rnd = new Random();
        DateTime tStart, tFinish;
        Console.Write("Elementlar sonini kiriting: ");
        int n = Int32.Parse(Console.ReadLine());
        mas = new int[n];
```

```
            Console.WriteLine("Boshlang‘ich massiv: ");
            for (int i = 0; i < n; i++)
            {
              mas[i] = rnd.Next(1, n);
              Console.Write($"{mas[i]}  ");
            }
            Console.WriteLine();
            Heap<int> heap = new Heap<int>(mas);
            Console.WriteLine("Uyum:");
            heap.Print();
            tStart = DateTime.Now;
            Console.WriteLine($"Tartiblash
boshlandi:{tStart}");
            heap.HeapSort();
            tFinish = DateTime.Now;
            Console.WriteLine($"Tartiblash
tugadi:{tFinish}");
            Console.WriteLine($"Bajarilish
vaqti:{tFinish.Subtract(tStart)}");
            Console.WriteLine("Tartiblangan uyum:");
            heap.Print();
          }
          // Ustunlik navbatlarni yaratish funksiyasi
          void HeapPQueue()
          {
            Random rnd = new Random();
            Console.Write("Elementlar sonini kiriting:");
            int n = Int32.Parse(Console.ReadLine());
            int k;
            Heap<int> heapPQ = new Heap<int>(n);
            for (int i = 0; i < n; i++)
            {
              k = rnd.Next(1, n * 3);
              heapPQ.Insert(k);
```

```
        }
        Console.WriteLine("Shakllantirilgan uyum");
        heapPQ.Print();
        Console.WriteLine();
        for (int i = 0; i < n-5; i++)
        {
          k = heapPQ.DeleteMax();
          Console.WriteLine($"Yechib olingan maksimal
element: {k}");
          heapPQ.Print();
        }

        Console.WriteLine($"Maksimal element:
{heapPQ.GetMaximum()}");
      }
      //HeapSortMax();
      HeapPQueue();
      Console.ReadKey();
    }
  }
}
```

SULTON MAMANAZAROVICH GAYNAZAROV,
DONIYOR YUSUPOVICH SAIDOV

ALGORITMLAR VA BERILGANLAR TUZILMASI

1-qism

O‘QUV QO‘LLANMA

Muharrir Z.N.Buranov

Bosishga ruxsat etildi 07.03.2023y. Bichimi 60X84 $^{1}/_{16}$.
Bosma tabog‘i 16,0. Shartli bosma tabog‘i 16,0. Adadi 10 nusxa.
Buyurtma № 31. Bahosi kelishilgan narxda.
“Ma’rifat” nashriyoti. Toshkent, Salorbo‘yi kochasi, 35A.
O‘zbekiston Milliy universiteti bosmaxonasida bosildi.
Toshkent, Talabalar shaharchasi, O‘zMU.