3. Rūšiavimo algoritmai

Rūšiavimas yra viena iš bazinių kompiuterių darbo operacijų – kompiuteris vidutiniškai apie 25 procentus viso skaičiavimo laiko sunaudoja rūšiavimui. Rūšiavimo kaip algoritmo tikslas – išdėstyti duomenis tam tikra tvarka. Rūšiavimo darbą sąlygoja daug aspektų: duomenų sutvarkymo apibrėžimas, duomenų struktūra, rūšiuojamų laukų išskyrimas, atminties panaudojimas rūšiavimo operacijai, antrinės ar tretinės atminties poreikis, duomenų pateikimo rūšiavimui vienalaikiškumas ar eiliškumas, lygiagretus rūšiavimas, kiti veiksniai – visa tai turi didelę įtaką rūšiavimo algoritmo darbo efektyvumui.

Duomenys, kuriais disponuojama, paprastai saugomi sąraše, jei jie išsitenka vidinėje atmintyje. Duomenys saugomi faile(-uose), jei jiems reikia antrinės arba tretinės atminties. Kiekvieno duomens užimama vieta kompiuterio atmintyje vadinama įrašu, kiekvienas įrašas gali būti skaidomas į laukus, laukų reikšmės vadinamos raktais.

Rūšiavimas yra labai svarbus paieškos efektyvumui. Pvz., jei įrašai sąraše sutinkami atsitiktine tvarka, tikslinga naudoti nuoseklios paieškos metodą, kurio patobulinti beveik neįmanoma. Tačiau jei įrašai yra surūšiuoti pagal raktinius laukus, reikiamo rakto sąraše galima ieškoti labai efektyviai. Pvz., dvejetainė paieška sąraše `list[0].key ≤ list[1].key ≤ … ≤ list[n-1].key` lyginimo operacijas vykdo su `list[middle].key` reikšme, kur `middle = (n-1)/2`. Tokiu būdu, arba paieška baigiama sėkmingai, arba nepatikrinta sąrašo dalis yra sumažinta maždaug per pusę. Po j raktinių palyginimų, nepatikrinta sąrašo dalis yra $[n/2^j]$, o tai reiškia, kad šis metodas blogiausiu atveju reikalauja O(log n) raktų reikšmių palyginimų.

Savybė: *dvejetainei paieškai galima naudoti dvejetainį sprendimų medį (decision tree), aprašantį paieškos procesą: rakto reikšmės ir sąrašo indeksai talpinami viršūnėse, kelias medyje nuo šaknies iki bet kokios viršūnės reiškia paieškos lyginimų seką; tuomet lengva pastebėti, kad dvejetainė paieška atlieka ne daugiau kaip O(log n) palyginimų.*

Formalizuojant rūšiavimą, tariama kad yra duotas įrašų sąrašas (R_0, R_1, …, R_{n-1}), kuriame kiekvienas įrašas R_i turi rakto reikšmę K_i. Raktų reikšmės sudaro pilnai sutvarkytą aibę, t.y. bet kokiom dviem raktų reikšmėms x ir y, galioja $x = y$ arba $x < y$, arba $x > y$. Šie sąryšiai yra tranzityviniai, t.y. bet kokiai iš trijų reikšmių x, y ir z,

iš $x < y$ ir $y < z$ išplaukia $x < z$. Rūšiavimo uždavinius galima apibrėžti kaip paiešką tokių perstatinių, kad $K_{\sigma(i-1)} \le K_{\sigma(i)}$, $0 < i \le n-1$, norima išdėstymo tvarka yra ($R_{\sigma(0)}$, $R_{\sigma(1)}$, ..., $R_{\sigma(n-1)}$).

Kadangi sąrašas gali turėti kelias vienodas raktų reikšmes, perstatinis nėra vienintelis. Kai kuriuose taikymuose tikslinga surasti vienintelį pakeitimą σ_s, kuris turi „stabilumo“ savybes:

[*surūšiuotas*] $K_{\sigma(i-1)} \le K_{\sigma(i)}$, kai $0 < i \le n-1$

[*stabilus*] Jei $i < j$ ir $K_i = K_j$ įvestame sąraše, tada R_i eina prieš R_j surūšiuotame sąraše.

Rūšiavimo metodas, kuris generuoja stabilų perstatinį σ_s yra vadinamas stabiliu. Stabilumas yra tik vienas iš kriterijų, atskiriant rūšiavimo metodus vienus nuo kitų. Be to, duomenų rūšiavimą galima apibūdinti vietos atmintyje poreikiu (pvz. vidinis ar išorinis rūšiavimas), naudojamu rūšiavimo metodu (pvz., burbuliuko algoritmas) ir pan..

Papildomos atminties (vidinės) kiekis, reikalingas algoritmų darbui, irgi yra svarbus rodiklis. Rūšiavimo algoritmai gali būti skirstomi į grupes pagal tai, kiek papildomos atminties naudoja (nenaudoja visiškai; naudoja papildomai tik atmintį, reikalingą rodyklėms išdėstyti; naudoja papildomai tiek pat atminties, kiek yra duomenų).

3.1. Elementarūs rūšiavimo algoritmai

Nagrinėjant rūšiavimą dažnai galima tarti, kad duomenys, kuriuos algoritmas turi rūšiuoti - tai skaičiai, laisvai telpantys į kintamajam skirtą atmintį. Taip galima geriau sukoncentruoti dėmesį į algoritmo darbo specifiką.

Išrinkimo algoritmas

Vienas iš paprasčiausių rūšiavimo algoritmų - išrinkimo (selection arba selection sort) vadovaujasi taisykle: iš turimo duomenų sąrašo išrenkamas minimalus elementą ir rašomas į pirmą vietą (sukeičiant pirmoje vietoje esantį elementą su rastu minimaliu). Po to tas pats principas taikomas sąrašui be pirmojo elemento, ir t.t. kol sąrašas tampa tuščias. Nors šis algoritmas priklauso „brutalios

jėgos“ algoritmams, jis dažnai naudojamas labai ilgiems įrašams su trumpais laukais rūšiuoti, nes šis algoritmas garantuoja, kad kiekvienas iš elementų bus perkeltas į kitą vietą ne daugiau kaip vieną kartą (algoritmas efektyvus duomenų sukeitimo vietomis – swap operacijos atžvilgiu).

Pagrindinės algoritmo darbo operacijos - tai duomenų lyginimas ir keitimas vietomis.

Savybė: *Vertinant algoritmo efektyvumą, nesunku parodyti, kad jis naudoja apytikriai* $N^2/2$ *lyginimų ir N sukeitimų vietomis*.

Įterpimo algoritmas

Įterpimo algoritmas yra beveik toks pat paprastas kaip ir išrinkimo, bet labiau lankstus. Pradedant nuo pirmojo elemento, kiekvienas sekantis x_i yra lyginamas su prieš jį esančiu, ir, jei jų tvarka netinkama, sukeičiami vietomis. Jei toks keitimas įvyko, lyginama pora (x_{i-1}, x_i), ir t.t. Lyginimų serija nutraukiama, jei kurios nors poros elementų sukeisti vietomis nereikia.

Šis algoritmas gerai žinomas bridžo žaidimo mėgėjams - taip jie rūšiuoja turimas kortas. Taikomuoju požiūriu jis labiau tinkamas situacijai, kai duomenų keitimo vietomis operacija yra lengvai vykdoma. Programuojant šį algoritmą, reikia parinkti tinkamą bazinę duomenų struktūrą (masyvas čia mažiau tinka). Naudojant sąrašus, tikslinga į juos įvesti ir signalinį žymenį (*sentinel*), tai žymiai palengvina programos konstravimą.

Analizuojant algoritmą, galima naudoti elementų įterpimo metidką. Tariant kad rūšiavimas įterpia R_i įrašą į sutvarkytą įrašų seką $R_0, R_1, \ldots, R_{i-1}$, ($K_0 \le K_1 \le \ldots \le K_{i-1}$), taip kad gautoji dydžio i seka taip pat būtų sutvarkyta. Tuomet pradedama nuo sutvarkytos sekos R_0 ir nuosekliai į ją įterpia įrašus R_1, R_2, …, R_{n-1}. Kadangi kiekvienas įterpimas vėl duoda seką sutvarkytą, galima sutvarkyti sąrašą su n įrašų darant $n-1$ įterpimą. Ši strategija yra realizuojama *insertion_sort* pagalba:

```
void insertion_sort(element list[], int n)
{
      int i,j;
      element next;
      for (i = 1; i < n, i++) {
```

```
        next = list[i];
        for (j = i-1; j>= 0 && next.key < list[j].key;j--)
        list[j+1] = list[j];
        }
}
```

Savybė. *Įterpimo algoritmas vidutiniu atveju naudoja apytikriai $N^2/4$ lyginimų ir $N^2/8$ keitimų vietomis ir dvigubai daugiau operacijų blogiausiu atveju. Įterpimo metodas yra tiesinis beveik surūšiuotiems duomenims*.

Burbuliuko algoritmas

Šio algoritmo idėja - nuosekliai iš dešinės į kairę peržiūrint gretimų elementų poras ir jei reikia, elementus sukeičiant vietomis, į sekos pradžią perkelti mažesnius elementus. Taip elgiantis, mažesni elementai pasislenka į duomenų sekos pradžią, o pirmoje sekos vietoje atsiduria mažiausias elementas. Po to galima pritaikyti tą patį principą duomenų posekiui be pirmojo elemento, ir t.t. Tai panašu į virimo procesą, kai oro burbuliukai kyla į paviršių - iš čia kilęs algoritmo pavadinimas.

Savybė: *Burbuliuko algoritmas apytikriai naudoja $N^2/2$ lyginimų ir $N^2/2$ keitimų vietomis ir vidutiniu, ir blogiausiu atvejais.*

Nė vienam iš trijų pateiktų algoritmų nereikia papildomos arba tarpinės atminties. Tolesnis algoritmų darbo arba jų tinkamumo vienai ar kitai situacijai vertinimas turi apimti lyginimo ir keitimo vietomis operacijų "kainą" ir jų santykį. Operacijų vykdymo laikas akivaizdžiai priklauso nuo įrašų ilgio, laukų, pagal kuriuos rūšiuojama, santykio, įrašų ilgių santykio ir pan. Jei šie faktoriai napalankūs, galima keisti algoritmų realizaciją. Pavyzdžiui, rūšiuojant ilgus įrašus, galima įvesti papildomą įrašų nuorodų masyvą ar sąrašą ir lyginimo atveju manipuliuoti šio masyvo reikšmėmis. Taip galima pasiekti, kad bet kuris iš jau nagrinėtų algoritmų naudos tik N įrašų keitimų vietomis.

Rūšiavimas Šelo metodu

Išvardyti algoritmai yra universalūs ta prasme, kad nekreipia dėmesio į duomenų sekos ypatumus. Tačiau jei duomenų ypatumai žinomi, algoritmus galima modifikuoti taip, kad jie dirbtų efektyviau, tokiems atvejams pasiūlyta daug ir įvairių metodu.

Įterpimo algoritmas nors ir dažnai naudojamas, yra lėtas, nes atliekami elementų sukeitimai vyksta tik tarp gretimų elementų, taigi elementai gali saraše pajudėti tik per vieną poziciją vienu metu. Pavyzdžiui, jei elementas su mažiausia reiksme yra masyvo pabaigoje, reikia N žingsnių norint jį patalpinti į jam priklausiančią vietą. Rūšiavimas Šelo metodu (toliau Shellsort) yra modifikuotas įterpimo algoritmas, kuris leidžia sukeisti vietomis toli esančius elementus.

Šio algoritmo idėja - pertvarkyti elementus taip, kad imant kiekvieną h-tąjį elementą, būtų gaunamas surūšiuotas sąrašas (vadinamas h-surūšiuotu), kartojant šį algoritmą skirtingoms h reikšmes (paprastai mažėjančioms pagal tam tikrą dėsnį), galima perkelti elementus sąraše dideliais atstumais ir tuo pagreitinti rūšiavimą.

Šio algoritmo darbo efektyvumas priklauso nuo skaičiaus h (vadinamosios inkrementinės sekos) kitimo dėsnio parinkimo ir nuo duomenų sekos.

Kitas šio metodo ypatumas - jį lengva programuoti netgi sudėtingesniems duomenims. Todėl jis dažnai aptinkamas įvairiuose taikymuose.

Kaip nuspręsti kurią inkrementinę seką naudoti? Paprastai į šį klausima atsakyti sunku. Daugelio skirtingų inkrementinių sekų savybės aprašytos literatūroje [Knuth 1998], pasiūlyta sekų, gerai veikiančių praktiškai, bet nėra įrodomai geriausios sekos. Praktikoje dažnai naudojamos geometriškai mažėjančios sekos, tuomet sukeitimo prieaugių skaičius yra logaritminis. Taip pat reikia atsižvelgti į aritmetinius prieaugių santykius, tokius kaip bendrųjų daliklių kiekis ir kt.

Inkrementinė seka 1 4 13 40 121 364 1093 3280 9841..., rekomenduota Knuth‘o 1969 metais [Knuth 1998], kurios santykis tarp prieaugių yra apie 1/3, duoda palyginti efektyvų rūšiavimo laiką net gana dideliems sąrašams.

Daug kitų inkrementinių sekų duoda efektyvesnį rūšiavimą, bet Knuth minėtą seką sunku pralenkti daugiau nei 20%, net santykinai dideliems N. Viena tokių sekų yra 1 8 23 77 281 1073 4193 16577..., seka $4^{i+1} + 3 \cdot 2^i + 1$, $i>0$, kuri yra greitesnė blogiausiu atveju. Ši, Knuth`o ir daug kitų sekų turi panašias dinamines charakteristikas dideliems failams

Kita vertus, yra keletas blogų inkrementinių sekų, pvz., 1 2 4 8 16 32 64 128 256 512 1024 2048… (tikroji Shell‘o pasiūlyta seka, kai jis pristatė algoritmą 1959 m.). Ši seka veiks blogai, nes nelyginėse pozicijose esantys elementai nelyginami su esančiais lyginėse iki pat paskutinio veiksmo. Lėto rūšiavimo efektas yra pastebimas atsitiktiniams sąrašams ir yra katastrofiškas blogiausiu atveju: algoritmas išsigimsta ir reikalauja kvadratinio laiko jei, pvz., pusė elementų su

mažiausiomis reikšmėmis yra lyginėse pozicijose ir pusė elementų su didžiausiomis reikšmėmis yra nelyginėse pozicijose.

Toliau programa pateikia kompaktišką Shellsort algoritmo realizaciją, naudojančią inkrementinę seką 1 4 13 40 121 364 1093 3280 9841…:

```
void shellsort(Item a[],int l, int r)
    { int i, j, h;
      for (h = 1; h <= (r-l)/9; h= 3*h+1);
      for ( ; h > 0; h/=3)
        for (i = l+h; i<= r; i++)
          { int j = i; Item v = a[i];
            while (j>= l+h && less(v, a[j-h]))
              { a[j] = a[j-h]; j -= h; }
            a[j] = v;
          }
     }
```

Shellsort algoritmo efektyvumo tiksli matematinė analizė yra sudėtinga, ji nėra tiksli, tai apsunkina ne tik įvairių didėjančių sekų įvertinimą, bet ir Shellsort algoritmo analitinį palyginimą su kitais algoritmais. Nežinoma net Shellsort algoritmo veikimo laiko funkcijos išraiška (kuri beje priklauso nuo inkrementinės sekos). Knuth pastebėjo, kad abi funkcinės išraiškos $N(\log N)^2$ ir $N^{1.25}$ gana gerai atitinka eksperimentinius duomenis, o vėlesni testai nurodo sudėtingesnės išraiškos funkciją $N^{1+1/\sqrt{\lg N}}$ kai kurioms inkrementinėms sekoms. Tiriant Shellsort algoritmą, galima pagrįsti tokias savybes.

Savybė: *k-sutvarkyto failo h-rūšiavimo rezultatas yra failas, sutvarkytas pagal h ir k.*

Savybė: *Shellsort algoritmas atlieka mažiau nei N(h-1)(k-1)/g palyginimų g-surūšiuoti failą, kuris yra h- ir k-sutvarkytas, su sąlyga, kad h ir k yra pirminiai.*

Savybė: *Shellsort algoritmas atlieka mažiau nei $O(N^{3/2})$ palyginimų sekai 1 4 13 40 121 364 1093 3280 9841….*

Savybė: *Shellsort algoritmas atlieka mažiau nei $O(N^{4/3})$ palyginimų sekai 1 8 23 77 281 1073 4193 16577….*

Savybė: *Shellsort algoritmas atlieka mažiau nei $O(N(\log N)^2)$ palyginimų sekai 1 2 3 4 6 9 8 12 18 27 16 24 36 54 81….*

Pasiskirstymo skaičiavimas

Labai paprasta rūšiuoti specifinėje situacijoje, kai elementų raktai keičiasi pvz., nuo 1 iki N, – belieka duomenis perkelti į vietas, kurių numeriai lygūs jų reikšmėms:

for i:= 1 **to** N **do** t[a[i]] = a[i];

Jei turima N skaičių, kurie yra intervale nuo 1 iki M, rūšiuoti galima irgi panašiu būdu, tik reikia žinoti, kiek kartų kiekvienas skaičius sekoje pasikartoja. Kai M nėra labai didelis, seką galima pereiti du kartus, pirmą kartą apskaičiuojant elementų pasikartojimus sekoje, o antrą kartą perkeliant tuos elementus į jiems priklausančias vietas.

3.2. Greito rūšiavimo algoritmas

Šis metodas, angliškai vadinamas *quicksort*, pasiūlytas C. A. R. Hoare 1962-ais metais [Sedgewick 1999]. Jis yra labai paplitęs ir aptinkamas daugelyje taikymų. Algoritmo realizavimo paprastumas ir efektyvus atminties naudojimas sąlygojo jo populiarumą. Algoritmo teigiamos savybės:

- papildoma atmintis beveik nenaudojama;
- algoritmo sudėtingumas yra O(NlogN) viduriniu atveju;
- algoritmo realizavime gali būti apibrėžti labai trumpi vidiniai ciklai.

Algoritmo neigiamos savybės:

- algoritmas rekursyvus, todėl diegimas komplikuotas, kai nėra rekursijos mechanizmo;
- algoritmo sudėtingumas yra $O(N^2)$ blogiausiu atveju;
- algoritmas labai jautrus programavimo klaidoms.

Quicksort remiasi paradigma "skaldyk ir valdyk". Pagrindinė idėja - išskaidžius duomenų seką į dvi dalis taip, kad vienoje iš jų visi elementai būtų mažesni už kitos dalies elementus, toliau šios dvi dalys gali būti rūšiuojamos nepriklausomai viena nuo kitos. Todėl parinkus slenkstį, t.y elementą, kuris galima sakyti jau yra savo vietoje, galima skaidyti seką į dvi dalis: vienoje iš jų visi elementai yra mažesni už slenkstį, o kitoje dalyje – didesni. Rūšiavimo programa tuomet gali atrodyti taip:

```
Void quicksort(element list[], int left, int right)
/*rūšiuojame list[left], ..., list[right] nemažėjančia
tvarka pagal raktinius laukus. List[left].key sutartinai
pasirenkamas kaip atskaitos raktas. Tarkime, kad
list[left].key ≤ list[right].key*/
{
      int pivot,i,j;
      element temp;
      if (left < right) {
         i = left;     j = right + 1;
         pivot = list[left].key;
         do {
         /*ieškome raktų iš kairės ir dešinės raktų
dalies,pakeičiant out-of-order elementus tol, kol
kairysis ir dešinysis rėžiai susikerta arba susiliečia*/
            do
               i++;
            while (list[i].key < pivot);
            do
               j--;
            while (list[j].key > pivot;
            if (i < j)
               SWAP(list[i],list[j],temp);
            } while (i < j);
            SWAP(list[left],list[j],temp);
            quicksort(list,left,j-1);
            quicksort(list,j+1,right);
      }
}
```

Įvertinus reikalingus veiksmus ir juos pertvarkius, algoritmą galima vykdyti gan tobulai, pvz., suradus kairėje sekos pusėje elementą, kuris didesnis už slenkstį ir kurį reikia perkelti į dešinę pusę, galima prieš tai dar surasti elementą dešinėje,

kuris mažesnis už slenkstį, ir tik tada abu šiuos elementus sukeisti vietomis - taip yra taupomi sukeitimo veiksmai ir atmintis. Aišku, šis algoritmas yra nestabilus, jis gali ne tik keisti lygių elementų tvarką, bet ir juos dėstyti skirtingose vietose. Algoritmo efektyvaus realizavimo (programavimo) ypatumus sąlygoja tokie faktoriai:

- rekursijos eliminavimas (sudarant galimybę detaliai valdyti algoritmo vykdymo eigą ir iš anksto numatyti nepalankių arba išsigimusių sekų atvejus);
- trumpi posekiai – esant trumpiems sąrašams, greitas rūšiavimas yra neefektyvus, todėl tikslinga trumpą posekiams, pvz., tokiems kad elementų skaičius $\leq$ M, naudoti kokį nors tiesioginį metodą, pvz., įterpimą, ribinį skaičių M parenkant tinkamiausią esamoms sąlygoms;
- sąrašo skaidymo į dalis slenksčio parinkimas - ši problema irgi svarbi greito rūšiavimo algoritme – dažniausiai naudojamas arba didesnio iš pirmųjų dviejų nesutampančių sekos elementų principas, arba vadinamasis medianos iš trijų elementų principas.

Quicksort analizė

Algoritmo elgesys blogiausiu atveju yra lygus $O(n^2)$ – tai akivaizdu, parenkant seką ir skaidymo į dalis slenkstį taip, kad viena iš dalių visada būtų tuščia. Tačiau palankiais atvejais, kai kiekvienas įrašas yra teisingoje pozicijoje, sąrašo dalis į kairę bus tokio paties dydžio kaip ir į dešinę, tuomet kiekvienos dalies dydis apytikriai bus $n/2$. Intuityviai aišku, kad quicksort turėtų veikti optimaliausiai, kai sąrašas skaidomas į dvi lygias dalis. Laikas, reikalingas lokalizuoti įrašą n dydžio sąraše, yra lygus $O(n)$. Jei $T(n)$ yra laikas, reikalingas surūšiuoti sąrašą iš n įrašų, tai kai sąrašas skyla į apytikriai dvi lygias dalis, o kiekvieną kartą įrašas yra tinkamoje vietoje, laiką galima įvertinti:

$T(n) \leq cn + 2T(n/2)$, kai kurioms konstantoms c

$$\leq cn + 2(cn/2 + 2T(n/4))$$

$$\leq 2cn + 4T(n/4)$$

$$\ldots$$

$$\leq cn \log_2 n + nT(1) = O(n \log_2 n)$$

Eksperimentiniai rezultatai rodo, kad kai vidutinis skaičiavimo laikas atitinka nurodytą santykį, tuomet *quicksort* yra geriausias iš iki šiol nagrinėtų vidinio rūšiavimo metodų.

Savybė: *Tegul $T_{avg}(n)$ yra laukiamas laikas, kurį užtruks quicksort, kad surūšiuotų failą iš n įrašų. Tada egzistuoja gokia konstanta k, kad $T_{avg}(n) \leq kn \log_e n$, kai $n \geq 2$.*

Skirtingai nuo įterpimo rūšiavimo, kuriame papildoma vieta atmintyje reikalinga tik vienam įrašui, greitam rūšiavimui reikalingas stekas, realizuojantis rekursiją. Tuo atveju, kai sąrašas skaidomas taip kaip aukščiau paminėtoje savybėje, maksimalus rekursijos gylis (*recursion depth*) bus log *n*, o tai reikalauja steko apimties, vertinamos dydžiu O(log *n*). Blogiausias atvejis būna tada, kai sąrašas suskaidomas į kairę dalį, kurios dydis yra $n - 1$, ir į dešinę dalį, kurios dydis yra 0, kiekviename rekursijos lygyje. Šiuo atveju, rekursijos gylis lygus *n*, kuris reikalauja O(*n*) steko apimties. Esant blogiausiam atvejui reikalinga steko apimtis gali būti sumažintas 4 kartus, tariant kad dešiniosios sąrašo dalies, kurios dydis mažesnis už 2, nebūtinai turi būti saugoma atmintyje, bent jau laikinai.

Greito rūšiavimo algoritmo taikymas ranginės statistikos uždaviniams

Pagrindinė quicksort algoritmo idėja gali būti panaudota ne vien duomenims rūšiuoti. Labai dažnai sutinkami uždaviniai kaip antai: iš skaičių sekos išrinkti k mažiausių elementų, arba apskaičiuoti skaičių aibės medianą. Šitokius ir panašius uždavinius (dažnai minimus ranginės statistikos vardu), aišku, galima spręsti rūšiuojant turimą skaičių seką, tačiau iš esmės tokiems uždaviniams pilnas rūšiavimas nėra reikalingas. Efektyvesnis sprendimo būdas yra panaudoti dalinimo į dvi dalis operaciją (pateikiamoje žemiau programoje ji išskirta atskirai, *partition* vardu, ją valdantys parametrai yra a – slenkstis, l, r – kairiojo ir dešiniojo elementų pozicijos, šie parametrai perduodami procedūrai), t.y. parinkti tinkamą elementą ir suskaidyti sąrašą į dvi dalis taip, kad vienoje iš jų būtų k mažesnių už parinktą elementų, o kitoje (N-k) didesnių. Jei nepavyksta tai atlikti iš karto, skaidymo operacija kartojama kelis kartus, bet tik vienai iš dalių. Programuojant šitokią procedūrą, ji iškviečia save tik vieną kartą, iš esmės ji nėra rekursyvi ir eliminuoti rekursiją nėra sudėtinga. Tam nebūtina naudoti netgi dėklo, nes procedūra, grįždama vėl į savo pradžią, atnaujina parametrų reikšmes. Tuomet rekursyvi programos versija galėtų atrodyti taip:

```
Select(Item a[], int l, int r, int k)
    {  int i;
        if (r <= l) return;
        i = partition(a, l, r);
        if (i > k) select(a, l, i-1, k);
        if (I < k) select(a, i+1, r, k);
    }
```

Savo ruožtu, nerekursyvi programa galėtų atrodyti:

```
Select(Item a[], int l, int r, int k)
      {
         while ( r > l )
              {  int i = partition(a, l, r);
                  if (i >= k)  r = i-1;
                  if (I <= k)  l = i+1;
              }
      }
```

Optimalaus rūšiavimo laiko vertinimas

Rūšiavimo metodų, kurie buvo aptarti, blogiausio atvejo veikimo laikas yra $O(n^2)$, o jo vertinimas priklauso nuo konkrečių algoritmų. Tačiau optimalaus rūšiavimo laiko vertinimas gali būti atsietas nuo konkrečių procedūrų ir nagrinėjamas teoriniame lygmenyje. Teoriškai, vidinio rūšiavimo algoritmai turi dvi pagrindines operacijas: raktų palyginimą ir elementų keitimą vietomis (swap). Tuomet galima teigti, kad geriausias galimas laikas yra $O(n \log_2 n)$.

Toks vertinimas gali būti įrodytas, naudojant sprendimų medį, kuris vizualiai atspindi rūšiavimo procesą. Kiekviena medžio viršūnė vaizduoja dviejų raktų palyginimą, ir kiekviena šaka rodo palyginimo rezultatą. Todėl, kiekvienas kelias, kuriuo galima apeiti medį, vaizduoja skaičiavimų seką, kuriuos rūšiavimo algoritmas galėtų atlikti.

Savybė*: Bet kuris sprendinių medis, kuris rūšiuoja n skirtingų elementų, yra bent* $\log_2(n!) + 1$ *aukščio*.

Iš tikro, rūšiuojant n elementų galima gauti $n!$ skirtingų galimų atsakymų. Todėl bet koks sprendinių medis turi turėti bent $n!$ lapų. Bet sprendinių medis taip

pat yra ir dvejetainis medis, kuris gali turėti daugiausiai 2^{k-1} lapų, jeigu jo aukštis yra k. Todėl aukštis turi būti bent $\log_2(n!) + 1$.

Savybė: *Bet koks algoritmas, kuris rūšiuoja naudodamas palyginimus, turi turėti blogiausio atvejo skaičiavimo laiką ne mažiau $\Omega(n \log_2 n)$.*

Kiekvienam sprendinių medžiui, turinčiam $n!$ lapų, gali (arba turi) egzistuoti kelias, kurio ilgis yra $c\ n\log_2 n$, kur c yra konstanta. Blogiausiam atvejui turi egzistuoti kelias, kurio ilgis $\log_2 n!$. Dabar, $n! = n(n - 1)(n - 2), \ldots, (3)(2)(1) \geq (n / 2)^{n/2}$. Taigi $\log 2n! \geq (n / 2)\log_2 (n / 2) = O(n \log_2 n)$.

3.3. Skaitmeninis rūšiavimas

Rūšiuojami duomenys dažnai yra sudėtingi, pvz., įrašai duomenų bazėse, ar telefonų sąrašai, ar bibliotekiniai katalogai. Paprastai apie šių įrašų laukų reikšmes nieko apibrėžto negalima pasakyti. Tačiau jei apie įrašų ar laukų reikšmes galima pasakyti ką nors papildomo, šią informaciją tikslinga panaudoti rūšiavimo procedūroms.

Skaitmeninio rūšiavimo algoritmuose (*radixsort*) duomenų reikšmės interpretuojamos kaip skaičiai M-ainėje skaičiavimo sistemoje. Priklausomai nuo elementų reikšmių i-oje pozicijoje skaičiai gali būti suskirstyti į M grupių, po to kiekviena iš šių grupių gali būti lygiai taip pat suskirstyta į M pogrupių, priklausomai nuo reikšmių j-oje pozicijoje, ir t.t.. Įvedus vienokį ar kitokį pozicijų parinkimo dėsnį, gaunami išsamiai apibrėžti skaitmeninio rūšiavimo algoritmai.

Skaitmeninio rūšiavimo algoritmai nagrinėjami kai M yra 2 laipsnis (turint omenyje kompiuterinę realizaciją), o pozicijos keičiasi pereinant nuo i prie i+1 arba i-1 arba jas grupuojant po kelias. Šioje situacijoje, programuojant skaitmeninio rūšiavimo algoritmus, tikslinga turėti funkcijas tiesioginiam darbui su bitais, pvz. ekvivalenčią tokiai instrukcijai: $x\ (div\ 2^k)\ mod\ 2^j$.

Reikia pastebėti, kad taikant skaitmeninį rūšiavimą skaičiai turi būti pakankamai dideli. Jei skaičiai nėra dideli ir kiekvieną jų sudaro ne daugiau kaip b bitų, naudojant pasiskirstymo skaičiavimo algoritmą duomenis galima rūšiuoti taip, kad rūšiavimo laikas tiesiškai priklausytų nuo duomenų kiekio. Belieka tik atmintyje išskirti 2^b dydžio lentelę, kad b bitų ilgio skaičiais galima būtų laisvai disponuoti.

Dažniausiai naudojamos dvi skaitmeninio rūšiavimo procedūros: skaitmeninio keitimo (*radix exchange sort* arba *most significant digit - MSD*) ir tiesioginio skaitmeninio rūšiavimo (*straight radix sort* arba *least significant digit - LSD*), besiskiriančios apdorojamų bitų tvarka. Pirmasis metodas (MSD) pagrįstas leksikografine tvarka ir bitų pozicijas numeruoja iš kairės į dešinę. Tai reiškia, kad skaičiai, prasidedantys dvejetainiu nuliu, rūšiuojamoje sekoje pasirodys anksčiau negu skaičiai, prasidedantys dvejetainiu vienetu. Kai reikia dvejetaines pozicijas analizuoti ir skaičius keisti vietomis, galima taikyti keitimo vietomis procedūrą, panašiai kaip *partition* procedūra greito rūšiavimo algoritme.

Tiesioginio skaitmeninio rūšiavimo metodas (LSD), skirtingai nuo skaitmeninio keitimo algoritmo, analizuoja bitus iš dešinės į kairę. Jis iš pradžių rūšiuoja duomenis pagal dešiniausią poziciją, po to pagal prieš tai esančią poziciją, po to dar prieš tai ir t.t. Elementams, kurių bitai j-1 pozicijoje sutampa, šis metodas neturi keisti tarpusavio tvarkos, t.y. jis yra stabilus. Labai panašiai elgdavosi senos skaičiavimo mašinos, rūšiuodamos perfokortas.

LSD rūšiavimas dažnai yra paprastesnis nei MSD, nes elementų grupių nereikia rūšiuoti atskirai, o tai reiškia, kad LSD rūšiavimo resursų sanaudos yra mažesnės nei MSD.

Formalizuojant LSD skaitmeninį rūšiavimą, galima daryti prielaidą, kad įrašai $R_0, ..., R_{n-1}$ turi reikšmes, kurios yra $(x_0, x_1, ..., x_{d-1})$ ir $0 < x_i < r$. Taip pat galima tarti, kad kiekvienas įrašas turi nuorodos lauką ir kad įvedamas sąrašas yra dinaminis sąrašas. Atskiras elementų grupes galima realizuoti kaip eilutes su nuorodomis *front[i]* į i-tos eilutės pradžią ir nuoroda *rear[i]* į jos pabaigą, $0 \le i < r$. Tuomet įvedamus įrašus galima apibrėžti kaip tiesinį sąrašą, pvz. kai $r = 10$ ir $d = 3$, įrašai aprašomi taip:

```
#define max_digit 3 /* skaičiai tarp 0 ir 999 */
#define radix_size 10
typedef struct list_node *list_pointer;
typedef struct list_node {
     int key[max_digit];
     list_pointer link;
     };
```

Tuomet LSD skaitmeninio rūšiavimo programa gali atrodyti taip:

```
list_pointer radix_sort(list_pointer ptr)
/* tiesinio sąrašo skaitmeninis rūšiavimas */
{
     list_pointer front [radix_size], rear
[radix_size];
     int i, j, digit;
     for (i = max_digit - 1; i >= 0; i--) {
       for (j = 0; j < 0; j < radix_size; j++)
           front[j] = rear[]j] = null;
       while (ptr) {
           digit = ptr -> key[i];
           if  (!front[digit])
             front[digit]  = ptr;
           else
             rear[digit] -> link = ptr;
           rear[digit] = ptr;
           ptr = ptr ->link;
     }
     /* iš naujo sukuriamas tiesinis sąrašas kitam
lyginimui */
     ptr = null;
     for (j = radix_size - 1; j >= 0; j--)
       if (front[j]) {
           rear[j] -> link = ptr; ptr = front[j];
       }
     }
     return ptr;
}
```

Algoritmo vykdymo laikas labai priklauso nuo *radix* pagrindo parinkimo. Skaitmeninis rūšiavimas pagrindu 2 ir skaičiais nuo 1 iki 100 milijardų bus atliekamas labai lėtai, o pagrindu 10 ir skaičiais nuo 0 iki 999 bus atliktas labai greitai. Taigi parinkti pagrindą kiekvienam *n* reikia labai kruopščiai.

Savybė*:* *funkcija radix_sort daro max_digit lyginimų, kiekvienas lyginimas užima o(radix_size + n) laiko, o visos programos vykdymo laikas yra o(max_digit(radix_size + n))*.

Jei skaitmeninės sistemos pagrindas yra 2-to laipsnis, tuomet galima pastebėti tokias abiejų skaitmeninio rūšiavimo algoritmų savybes:

1. *radixexchange* metodas naudoja apie NlgN bitų lyginimų,

2. abu skaitmeniniai metodai, rūšiuodami N skaičių, kurių kiekvienas yra b bitų ilgio naudoja mažiau negu Nb bitų lyginimų,
3. tiesioginis metodas rūšiuoja N skaičių, kurių kiekvienas b bitų ilgio, kartodamas algoritmą b/m kartų (jei išskiriama papildoma atmintis 2^m skaitliukių saugoti, o taip pat buferis pertvarkyti sąrašą).

3.4. *Heapsort* algoritmas

Naudojant prioritetinės eilutės operacijas, galima konstruoti *heap* rūšiavimo algoritmą (*heapsort*), kuris remiasi duomenų struktūra *heap*. Algoritmo idėja labai paprasta: turint duomenis heap struktūroje, atspausdinti šakninį elementą (kuris yra didžiausias aibės elementas), po to jį pašalinti, o likusią aibę vėl pertvarkyti į heap struktūrą – taip elgtis tol, kol aibė nebus tuščia. Visi išmetami elementai išsirikiuos mažėjančia tvarka. Formaliai užrašytas algoritmas bus toks:

```
void heapsort(element list[], int n)
/* atliekamas masyvo heap rūšiavimas */
{
   int i, j;
   element temp;
   for (i = n/2; i > 0; i--)
     adjust(list, i, n);
   for (i = n - 1; i > 0; i--) {
     swap(list[1], list [i + 1], temp);
     adjust(list, 1, i);
   }
}
```

Ši programa naudoja funkciją adjust, kuri reikalinga *max-heap* sutvarkymui:

```
   void adjust(element list[], int root, int n)
   /* pertvarkomas dvejetainis medis, kad sukurtume heap
struktūrą */
   {
      int child, rootkey;
      element temp;
      temp = list[root];
      rootkey = list[root].key;
```

```
      child = 2 * root;     /* left child*/
      while (child <= n ) {
        if ((child < n) &&
        (list[child].key < list[list + 1].key))
          child++;
        if  (rootkey > list[child].key)     /*
palyginame šaknį ir didžiausią sūnų */
          break;
        else {
          list[child  /  2]  =  list[child];           /*
pereinam prie tėvo */
          child *= 2;
        }
      }
      list[child/2] = temp;
   }
```

Šioje heapsort algoritmo versijoje yra naudojama *max-heap* struktūra, t.y. šakninis elementas yra didžiausias visoje aibėje. Pirmiausia į tuščią heap struktūrą yra įterpiama *n* įrašų, po to įrašai vienas po kito iš jos šalinami struktūros. Heap iš *n* elementų galima sukurt greičiau, naudojant funkciją *adjust*. Ši funkcija pertvarko dvejetainį medį T, kurio abu pomedžiai patenkina, o šaknis gali ir nepatenkinti heap sąvybės, taip kad visas medis patenkintų heap sąvybę. Jei medžio su šaknimi *i* aukštis yra *d*, tai **while** ciklas yra atliekamas daugiausiai *d* kartų. Taigi funkcijos *adjust* sudėtingumas vykdymo atžvilgiu yra O(*d*).

Rūšiuojant sąrašą, atliekama $n - 1$ žingsnių, per kiekvieną žingsnį sukeičiant pirmą heap įrašą su paskutiniu. Kadangi pirmas įrašas visada turi didžiausią reikšmę, tai jis dabar jau bus savo vietoj (t.y., kur turėtų būti, surūšiavus failą). Sumažinus heap struktūros dydį, ji vėl pertvarkoma, pvz. pirmu žingsniu įrašas su didžiausia reikšme perkeliamas į *n*-tąją poziciją, antru žingsniu perkeliamas įrašas su antra pagal dydį reikšme į $n - 1$ poziciją, i – tuoju žingsniu perkeliamas įrašas su i – taja pagal dydį reikšme į $n - i + 1$ poziciją. Funkcija *heapsort* realizuoja šią strategiją, jos iškvietimas yra *heapsort(list, n)*.

Heap rūšiavimas, kaip tai seka iš jo realizacijos, reikalauja tik fiksuoto papildomos atminties kiekio, o tuo pačiu metu jo sudėtingumas vykdymo atžvilgiu bus O($n\log n$), kaip blogiausiu taip ir vidutiniu atvejais. Heap rūšiavimas yra truputį lėtesnis nei sąlajos, kai naudojama O(*n*) papildomos vietos, tačiau jis greitesnis už sąlajos rūšiavimą, kai naudojama papildoma atmintis yra proporcinga O(1).

***Heapsort* analizė**

Tariant kad $2^{k-1} \le n \le 2^k$, atitinkamas dvejetainis medis turės k lygių ir viršūnių skaičius lygyje *i* yra 2^{i-1}. Pirmuoju programos ciklu **for** *heapsort* iškviečia *adjust* po vieną kartą kiekvienai viršūnei, kuri turi sūnų. Taigi, laikas kiek trunka šis ciklas, yra kiekvieno lygio viršūnių sandaugų su didžiausiu atstumu, kuriuo gali judėti viršūnė, suma. O tai yra ne daugiau kaip:

$$\sum_{i=1}^{k} 2^{i-1}(k-i) = \sum_{i=1}^{k} 2^{k-i-1} i \le n \sum_{i=1}^{k-1} i/2^i < 2n = O(n)$$

Antru **for** ciklu *heapsort* iškviečia *adjust* $n - 1$ kartų , kur maksimalus aukštis yra $\log_2(n + 1)$. Taigi, ciklo vykdymo laikas yra $O(n \log n)$, o tai duoda ir galutinį vykdymo laiką – $O(n \log n)$.

3.5. Sąlajos rūšiavimas

Apdoruojant duomenis dažnai tenka į gan didelį surūšiuotą sąrašą įterpti tam tikrą kiekį naujų duomenų, po to jį vėl surūšiuoti. Galima elgtis keliais būdais: iš pradžių duomenis įterpti į failą, o po to rūšiuoti; galima išlaikyti surūšiuotą sąrašą įterpiant kiekvieną elementą atskirai; galima iš pradžių duomenis surūšiuoti, o po to abu failus sulieti – būtent pastarasis metodas ir bus nagrinėjamas.

Sąlajos rūšiavime esminis yra sąlajos (*merge*) procesas, kuris sujungia dvi arba kelias duomenų aibes į vieną aibę ir tam tikra prasme yra priešingas išrinkimo (*selection*) operacijai. Paprasčiausiai sąlajos procesas atrodo, kai suliejami du surūšiuoti sąrašai. Tuomet per tiesinį laiką, t.y. po vieną kartą nuskaitant abiejų sąrašų elementus ir keičiant tik nuskaitymų eilę, galima sukurti naują surūšiuotą sąrašą.

Pateikiama programa naudoja $O(n)$ papildomos vietos, ji sujungia surūšiuotus sąrašus (*list*[*i*], …, *list*[*m*]) ir (*list*[*m* + 1], …, *list*[*n*]) į vieną surūšiuotą sąrašą (*sorted*[*i*], …, *sorted*[*n*]):

```
Void merge(element list[], element sorted[], int i,
int m, int n)
```

```
   {
      int j,k,t;
      j = m+1;         /*antrosios      sąrašo      dalies
indeksai*/
      k = i            /*surūšiuoto sąrašo indeksai*/

      while (i <= m && j <= n) {
         if (list[i].key <= list[j].key)
            sorted[k++] = list[i++];
         else
            sorted[k++] = list[j++];
      }
      if (i > m)
      /*sorted[k],   ...,   sorted[n]   =   list[j],   ...,
list[n]*/
         for (t = j; t <= n; t++)
            sorted[k+t-j] = list[t];
         else
         /*sorted[k],  ...,  sorted[n]  =  list[i],  ...,
list[m]]*/
            for (t = i; t <= m; t++)
               sorted[k+t-i];
   }
```

Sąlajos proceso analizė: kiekvienoje while ciklo iteracijoje vienas įrašas yra pridedamas prie surūšiuoto sąrašo, tai yra, *k* padidėja 1. Visas įrašų skaičius, pridėtas prie surūšiuoto sąrašo, yra *n* – i +1. Tai reiškia, kad while ciklas kartojamas daugiausiai *n* – i +1 kartą, bendras skaičiavimo laikas yra O(n – i +1). Jeigu įrašų ilgis yra *M*, tau šis laikas iš tikrųjų yra O(*M*(*n* – *i* + 1)). Kai *M* yra didesnis už 1, sujungto sąrašo vaizdavimas panaikina papildomus *n* – i +1 įrašus, tačiau turi būti skiriama vieta *n* – i +1 elementų tarpiniam sąrašui. Tuomet skaičiavimo laikas daugiau nėra priklausomas nuo *M*; jis tiesiog vertinamas O(n – i +1).

Šį sąlajos procesą galima modifikuoti, padarant pačią sąlają sudėtingesnę, tačiau vykdant algoritmą taip kad jis naudotų tik O(1) papildomos vietos. Pagrindiniai algoritmo žingsniai gali būti tokie (darant prielaidą, kad bendras įrašų skaičius *n* yra tiksluss kvadratas, o kiekviename sąraše esančių įrašų, kuriuos reikia sujungti, kiekis yra √*n* kartotinis):

- 1 žingsnis: identifikuojama $\sqrt{n}$ įrašų su didžiausiais raktais, einant iš dešinės į kairę ir kartu jungiant du sąrašus;
- 2 žingsnis: sukeičiami antro sąrašo įrašai, kurie identifikuoti 1 žingsnyje, su tais, kurie yra į kairę nuo identifikuotų iš pirmojo sąrašo, taip, kad $\sqrt{n}$ įrašų su didžiausiais raktais suformuotų vientisą bloką;
- 3 žingsnis: blokas, sudarytas iš $\sqrt{n}$ įrašų su didžiausiais raktais, sukeičiamas su kairiausiuoju bloku (jei jis dar nėra kairiausias blokas), rūšiuojamas dešiniausias blokas;
- 4 žingsnis: pertvarkomi blokai, išskyrus bloką su didžiausiais įrašais, į nemažėjančią pagal paskutinius raktus blokuose seką;
- 5 žingsnis: atliekame tiek sąlajos žingsnių, kiek jų reikia, kad būtų surūšiuoti $\sqrt{(n-1)}$ blokų, išskyrus bloką su didžiausiais raktais;
- 6 žingsnis: surūšiuojamas blokas su didžiausiais raktais.

Detalesnė analizė. Realizuojant algoritmo žingsnius, galima teigti, kad 1 ir 2 žingsniai bei 3 žingsnyje sukeitimo vietomis operacija užtrunka po $O(\sqrt{n})$ laiko ir užima po O(1) vietos. 3 žingsnio rūšiavimas gali būti atliktas per $O(n)$ laiką, užimant O(1) vietos, kai naudojamas įterpimo rūšiavimas. 4 žingsnis gali būti atliktas per $O(n)$ laiką ir reikalauja O(1) vietos, naudojant išrinkimo rūšiavimą. Kiekvienas įrašų keitimas vietomis naudojant išrinkimą, iš tikrųjų perkelia $\sqrt{n}$ dydžio bloką. Šiame žingsnyje įterpimo rūšiavimas būtų prastesnis už išrinkimą. Bendras 5 žingsnio skaičius yra ne daugiau $\sqrt{(n-1)}$, t.y. vykdymo laikas proporcingas $O(n)$. 6 žingsnio rūšiavimas gali būti atliktas per $O(n)$ laiko, naudojant arba išrinkimo arba įterpimo rūšiavimą. Todėl programos bendras laikas, realizuojant ją kaip aprašyta yra $O(n)$, o naudojama papildoma atmintis yra O(1).

Kartotinis sąlajos rūšiavimas

Kartotinio rūšiavimo atveju daroma prielaidą, kad įvedamos sekos turi n surūšiuotų vieneto ilgio sąrašų. Tuos sąrašus suliejamos poromis gaunant $n/2$ sąrašų, kurių ilgis 2 (jei n nelyginis, tada vienas iš naujų sąrašų bus vieneto ilgio). Toliau $n/2$ sąrašų yra suliejami poromis, ir t.t., tęsiant procesą kol lieka vienas sąrašas. Kartotinį algoritmą lengviau realizuoti, jei iš pradžių išskiriama funkcija, kuri atlieka vieną suliejimą:

```
   void merge_pass(element list[], element sorted[], int
n, int length)
{
int i, j;
for (i = 0; i <= n - 2 * length; i += 2 * length)
merge(list, sorted, i, i + length - 1, i + 2 * length -
1);
if (i + length < n)
merge(list, sorted, i, i + length - 1, n - 1);
else
for (j = i; j < n; j++)
sorted[j] = list[j];
}
```

Naudojant šią programą, nesunku tiesmukiškai parašyti rekursyvią programą, kuri naudotų funkciją *merge_pass,* ją iškviečiant surūšiuotiems sąrašams sujungti. Tuo tarpu nerekursyvus programos variantas irgi naudotų tą pačią funkciją ir gali atrodyti taip:

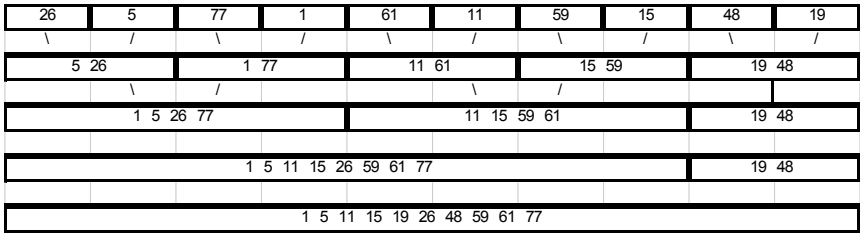
```
   void merge_sort (element list[], int n)
{
int length = 1;
element extra[MAX_SIZE];

while (length < n) {
merge_pass(list, extra, n, length);
length *= 2;
merge_pass(extra, list, n, length);
length *= 2;
}
}
```

Sąlajos rūšiavimo algoritmų sudėtingumas ir savybės yra tokios:

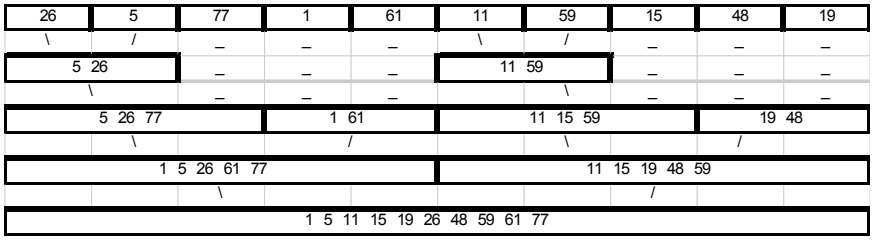
- nepriklausomai nuo duomenų, naudoja apytikriai NlogN lyginimų;
- naudoja papildomą atmintį, kurios tūris proporcingas duomenų kiekiui N, tačiau gali būti pasiektas O(1) vietos poreikis;

- algoritmai yra stabilūs;
- algoritmų darbas nepriklauso nuo duomenų tvarkos, jų sudėtingumas yra O(n log n);
- darbą galima pagerinti, kombinuojant jį su kitais rūšiavimo metodais.

Pav. 3.1. pateiktas pavyzdys parodantis kaip duomenys rūšiuojami ir jungiami po kiekvieno lyginimo, pav. 3.2. pateikia kitokią galimą kartotinio sąlajos rūšiavimo variantą [HSA 1993].



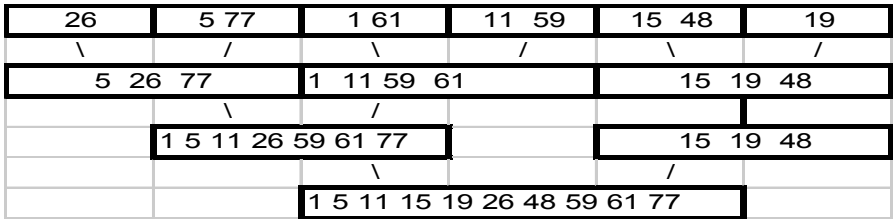
Pav. 3.1. Kartotinas sąlajos rūšiavimas.



Pav. 3.2. Kitas kartotino sąlajos rūšiavimo variantas.

Natūralusis sąlajos rūšiavimas

Algoritmą *merge_sort* galima pakeisti, kad rūšiuodamas atsižvelgtų į įvedamus duomenis. Šiuo atveju duomenys yra peržvelgiami, surandant juose jau surūšiuotas įrašų sekas ir pasižymint jų vietas rūšiuojamame sąraše.. Tada algoritmas gali šiuos surūšiuotus posąrašius interpretuoti kaip jau sulietus. Pav.3.3. iliustruoja, kaip šiuo atveju rūšiuojami pateikti duomenys.



Pav. 3.3. Kartotinas rūšiavimas fiksuojant dalinai surūšiuotus posekius.

3.6. Išorinis rūšiavimas

Apdorojant informaciją labai dažnai pasitaiko atvejų, kai disponuojama labai dideliais duomenų failais ar rinkiniais, kurie yra rūšiuojami ir netelpa kompiuterio vidinėje atmintyje. Tuo atveju anksčiau nagrinėti algoritmai netinka, vietoj jų yra naudojami vadinamieji išoriniai rūšiavimo metodai *(external sorting)*. Du pagrindiniai faktoriai iš esmės skiria vidinio ir išorinio rūšiavimo algoritmus:

- kadangi dauguma duomenų saugomi antrinėje (diskinėje) ar netgi tretinėje atmintyje, elemento išrinkimas iš išorinės atminties trunka kur kas ilgiau, negu tūkstančiai lyginimo ar keitimo vietomis operacijų ar kitokių skaičiavimų vidinėje atmintyje;
- duomenų failai, saugomi išorinėje atmintyje, turi savus duomenų išrinkimo ar paieškos metodus, kurių negalima atsisakyti ar pakeisti (pvz., magnetinėje juostoje duomenis galima išrinkti tik nuosekliai).

Todėl išorinio rūšiavimo atveju, kuriant ar taikant vienokį ar kitokį algoritmą, reikia atsižvelgti į visus duomenų apdorojimo veiksnius ir etapus, pilnos duomenų apdorojimo sistemos sąvoka (operacijų požiūriu) čia lygiai tokia pat svarbi kaip ir algoritmo. Natūralu yra įvesti dar vieną bazinę operaciją – kreipimąsi į išorinę (diskinę) atmintį. Todėl norint parinkti efektyvų išorinio rūšiavimo algoritmą, reikia kreipti dėmesį ne tik į lyginimo ar keitimo vietomis operacijų skaičių, kaip buvo daroma anksčiau, bet ir mažinti įvedimo/išvedimo (I/O) operacijų skaičių ir jų vykdymo laiką.

Pirminiai išorinio rūšiavimo principai ir metodai, kurie buvo sukurti kai duomenys dar buvo saugomi perfokortose ir popierinėse perfojuostose, naudojami ir dabar, kai rūšiuojama diskuose ir magnetinėse juostose, domeninėje atmintyje (*bubble-memory*) ir videodiskuose. Veržlus technologijų vystymasis kelia jiems

daug reikalavimų, verčia juos adaptuoti vis sudėtingesnei kompiuterinei aplinkai. Pvz., rūšiavimo metodai dabartinėse *multimedia* sistemose kompiuterinę atmintį skirsto į pirminę, antrinę, tretinę, iš kurių kiekviena turi didelės įtakos rūšiavimo greičiui ir algoritmams. Iš tikrųjų situacija yra dar sudėtingesnė. Šiuolaikiniuose kompiuteriuose vidinė atmintis nėra vienarūšė, jos sudėtinės ar ją aptarnaujančios dalys yra ir superoperatyvioji atmintis (*cache memory*), ir buferinė atmintis (*buffer memory*). Duomenų išrinkimo greičiai šiose dalyse irgi yra skirtingi. Aišku, kad rūšiavimo metodai galingose kompiuterinėse sistemose turi gerai išnaudoti ir šiuos skirtumus. Be to, I/O operacijas, pvz. diskines, irgi galima detaliau specifikuoti, kai skaitant iš disko ar rašant į jį, kreipiamas dėmesys į tokius svarbius faktorius:

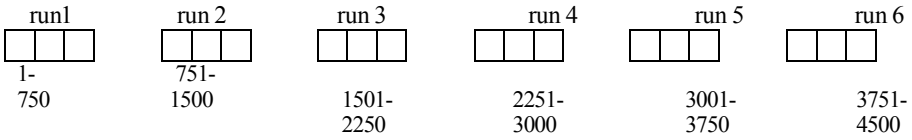
1. *paieškos laikas (seek time):* tai laikas per kurį galvutė nusistato teisingą disko takelį,
2. *gaišties laikas (latency time):* laikas, iki tol, kol skaitymo/rašymo galvutė yra ties reikiamu takeliu;
3. *perdavimo laikas (transmission time):* laikas, per kurį perduodami duomenys į diską arba iš jo.

Populiariausias išorinės atminties duomenų rūšiavimo būdas vadinamas išoriniu sąlajos (*external merge)* rūšiavimu. Šis metodas susideda iš dviejų atskirų fazių. Visų pirma, įvedimo failo dėmenys yra surūšiuojami naudojant tinkamą vidinio rūšiavimo metodą. Surūšiuoti segmentai, dar vadinami srautais, yra perrašomi į išorinės atminties įrenginius. Antra, srautai sugeneruoti pirmoje fazėje, yra sujungiami į sąlajos medį (*merge tree*), po to procesas kartojamas tiek kartų kol lieka tik vienas srautas. Kadangi funkcija *merge* reikalauja, kad tiktai įrašai su didžiausiomis reikšmėmis iš abiejų srautų, kurie yra suliejami, būtų išdėstyti atmintyje vienu metu, todėl yra įmanoma sulieti didelius srautus. Tai žymiai sunkiau pritaikyti vidinio rūšiavimo metodams.

Demonstruojant sąlajos medžio panaudojimą išorinio rūšiavimo procese, tikslinga pateikti pavyzdį, kuriame failas susidedantis iš 4500 įrašų rūšiuojamas kompiuteriu su vidine atmintimi, talpinančioje daugiausiai 750 įrašų. Įvedimo failas yra išsaugotas diske ir jo bloko ilgis yra 250 įrašų. Taip pat yra laisvas kitas diskas, kuris gali būti naudojamas kaip tarpinis. Į įvedimo diską negalima dar kartą ant viršaus įrašyti duomenų. Vienas iš būdų įvykdyti tokį rūšiavimą aprašomas žemiau.

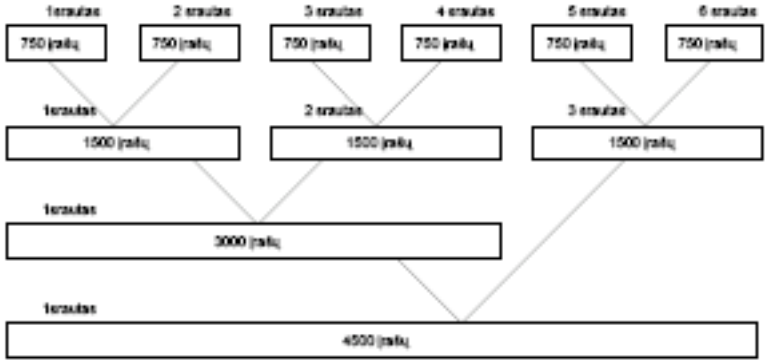
1. Naudojant vidinio rūšiavimo procedūrą surūšiuojami trys blokai vienu metu (t.y., 750 įrašų) gaunant šešis srautus R1-R6. Galime naudoti heap rūšiavimą ar

greitą rūšiavimą. Šie šeši srautai yra įrašomi į tarpinį diską, kaip parodyta pav.3.4.



Pav. 3.4. Srautai tarpiniame diske

2. Tuomet reikia išskirti tris vidinės atminties blokus, kurių kiekvienas gali išsaugoti 250 įrašų. Du iš šių blokų bus naudojami kaip įvedimo duomenų buferiai, o trečias kaip išvedimo duomenų buferis. Atskiriant srautus R1 ir R2, reikia visų pirma nuskaityti kiekvieno srauto vieną bloką į išvedimo duomenis. Kai išvedimo atmintis yra pilna, duomenys įrašomi į diską. Įvedimo atmintis papildoma kitu bloku iš to paties srauto. Tuomet R1 ir R2, R3 ir R4, R5 ir R6 yra suliejami. Šių pertvarkymų rezultatas yra 3 srautai, kurių kiekvieną sudaro 1500 atrūšiuotų įrašų arba 6 blokai. Procesas tęsiamas kaip parodyta pav 3.5.

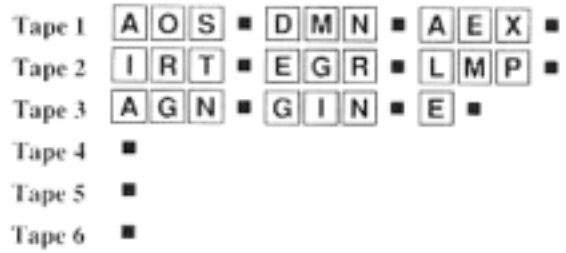


Pav. 3.5. Šešių srautų sąlaja.

Subalansuota daugybinė sąlaja magnetinėms juostoms

Išorinės atminties įrenginiai gali būti paskirstomi sąlajos srautams labai įvairiai, vienas iš būdų – subalansuota daugybinė sąlaja (*balanced multiway merging*). Tarkime, reikia surūšiuoti pakankamai didelio failo įrašus, o vidinėje atmintyje telpa tik trys įrašai. Sakykime, kad turime neribotą kiekį magnetinių juostų (nuoseklaus išrinkimo įrenginių), ir eiliniam suliejimui naudosime bet kurias tris iš jų. Tada pirmiausia iš pirminio failo skaitome nuosekliai po tris įrašus, juos rūšiuojame ir blokus po tris įrašus pakaitomis ir nuosekliai rašome į tris skirtingas juostas. Toliau vykdomos suliejimo procedūros. Po vieną įrašą iš kiekvienos juostos skaitoma į atmintį ir mažiausias iš jų rašomas į naują juostą. Vėl kreipiamąsi į juostą, kurioje buvo mažiausias įrašas, ir iš jos skaitomas naujas įrašas, vėl mažiausias iš jų rašomas į juostą. Taip tęsiama tol, kol nepasibaigs blokas juostoje, iš kurios skaitoma, po to ta juosta ignoruojama, o iš likusių dviejų juostų skaitomi ir suliejami likusieji įrašai. Taip naujoje juostoje suformuojamas blokas iš devynių elementų. Jei dar blokų yra, ši procedūra gali būti tęsiama. Po to vėl tą pačią porcedūrą galima taikyti naujai suformuotoms trims juostoms, kuriose išdėstyti devynių elementų ilgio blokai.

Pavyzdžiui, aprašytoje situacijoje po algoritmo žingsnio:

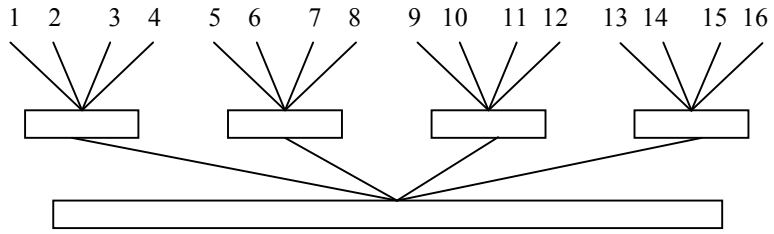


sekančiame žingsnyje gali atrodyti taip:

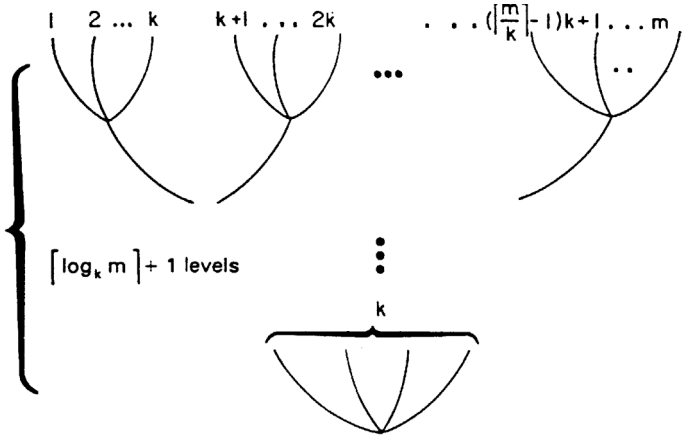
Ši aprašyta procedūra taikoma daugeliui gan efektyviai veikiančių rūšiavimo-sąlajos algoritmų, kurie subalansuotai naudoja nuoseklaus išrinkimo išorinius įrenginius. Balansuotumas reiškia tolygų išorinių atminties įrenginių darbo paskirstymą.

k-daugybinė sąlaja

Srautų sąlaja, apjungianti srautus po du, kaip pavaizduota pav. 3.5, gali būti apibendrinta *m* srautų atvejui. Tuomet sąlajos medis turėtų $\lceil \log_2 m \rceil + 1$ lygius arba $\lceil \log_2 m \rceil$ duomenų failo peržiūros skaičių. Failo peržiūros skaičių galima sumažinti, naudojant didesnį sąlajų skaičių, t.y. *k*-daugybinę sąlają, vienu metu suliejant k-srautų. Pav. 3.6. iliustruoja 4-daugybinę sąlają, kai yra 16 srautų. Peržiūros skaičius dabar yra lygus 2, palyginti su 4 peržiūromis, jei naudojama 2-daugybinė sąlaja. Apskritai, *k*-daugybinė sąlaja, kai yra *m* srautų, reikalauja daugumoje $\lceil \log_k m \rceil$ duomenų peržiūrų, kaip pavaizduota pav. 3.7. Vis dėlto didesnio skaičiaus sąlaja turi ir neigiamų padarinių. Visų pirma, k-srautams, kurių dydis pakankamai didelis, sulieti gali neužtekti vidinės atminties arba jos gali būti per mažai, kad tai atlikti per tam tikrą laiką. Be to didėjant sąlajų skaičiui, didėja lyginimų skaičius, kuris reikalingas kreipiantis į failus kad nustatyti duomenų suliejimo eilinį bloką. Pagrindinis įvertis, nulemiantis palyginimų skaičiaus didėjimą, yra (k-1)/ log2k. Kai *k* didėja, įvedimo/išvedimo laiko sumažėjimas tampa atsvara laikui, kuris yra reikalingas centriniui procesoriui atlikti k-daugybinę sąlają.



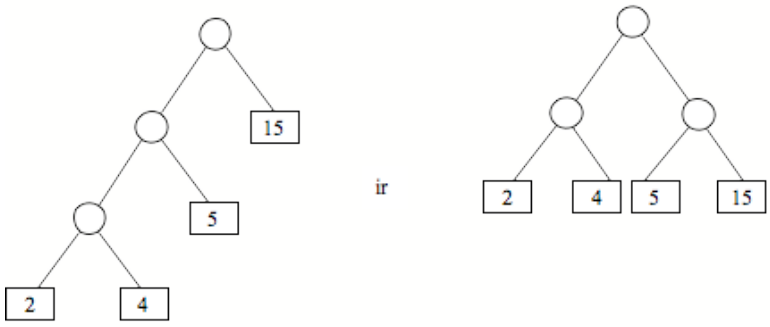
Pav. 3.6. 2-daugybinė sąlaja 16-ai srautų



Pav. 3.7. k-daugybinė sąlaja

Optimali srautų sąlaja

Srautai, kurie yra sugeneruoti sąlajos vykdymo metu arba sudaromi iš duomenų, gali būti ne to paties dydžio. Kai srautai yra skirtingų dydžių, srautų sąlajos strategija gali nepasiduoti minimaliam sąlajos vykdymo laikui. Pavyzdžiui, turint nurodyta tvarka 2, 4, 5 ir 15 ilgių srautus, pav. 3.8. pateikiami du būdai, kaip galima sulieti šiuos srautus, nuosekliai naudojant 2-daugybinę sąlają.



Pav. 3.8. Du skirtingi sąlajos būdai

Apvalios viršūnės parodo 2-daugybinę sąlają duomenimis, kurių vaikų viršūnės yra įvedimo operacija. Kvadratinės viršūnės parodo pradinius srautus. Apvalios

viršūnės nurodomos kaip vidinės viršūnės, o kvadratinės atitinka išorines viršūnes. Kiekvienas iš medžių yra sąlajos medis.

Pirmame sąlajos medyje pradedama suliejant dviejų ir keturių dydžio srautus ir gaunant šešių dydžio srautą. Toliau suliejant šį srautą su penkių dydžio srautu gaunamas 11 dydžio srautas. Pagaliau suliejant 11 ilgio srautą su 15 ilgio srautu, gaunamas norimas surūšiuotas 26 dydžio srautas. Kai sąlaja vykdoma naudojant pirmą sąlajos medį, keli įrašai suliejami po vieną kartą, tuo tarpu kai kiti gali būti suliejami net po tris kartus. Antrajame sąlajos medyje kiekvienas įrašas dalyvauja sąlajoje lygiai du kartus, o suliejamų srautų kiekis yra mažesnis.

Suliejimų skaičius, kuris įtraukia individualų įrašą, yra lygus atitinkamos išorinės viršūnės nuotoliui iki šaknies. Pavyzdžiui, srauto su 15 įrašų įrašai yra suliejami tiktai kartą pirmame medyje (pav. 3.8) ir du kartus antrame medyje tame pačiame pavyzdyje. Kadangi suliejamų įrašų skaičių yra tiesinis, bendras suliejimo laikas bus lygus srautų ilgių sandaugų su atitinkamų išorinių viršūnių nuotoliais iki šaknų sumai. Ši suma vadinama svertiniu išorinio kelio ilgiu (*weighted external path length)*. Medžiams pavyzdyje svertiniai ilgiai yra:

$2 \cdot 3 + 4 \cdot 3 + 5 \cdot 2 + 15 \cdot 1 = 43$ ir $2 \cdot 2 + 4 \cdot 2 + 5 \cdot 2 + 15 \cdot 2 = 52$.

Suliejant N srautų k-daugybinės sąlajos būdu, šį procesą galima minimizuoti, parenkant k-ainį sąlajos medį su minimaliu svertiniu išorinio kelio ilgiu.

Efektyvų minimalaus medžio problemos sprendimą pasiūlė D. Huffman'as [HAS 1993]. Formuluojant jo algoritmą C programavimo kalba, tikslinga pradėti nuo specifikacijų formulavimo:

```
typedef  struct  tree_node  *tree_pointer;
typedef  struct tree_node    {
     tree_pointer  left_child;
     int                   weight;
     tree_pointer  right_child;
} ;
tree_pointer  tree;
int  n;
```

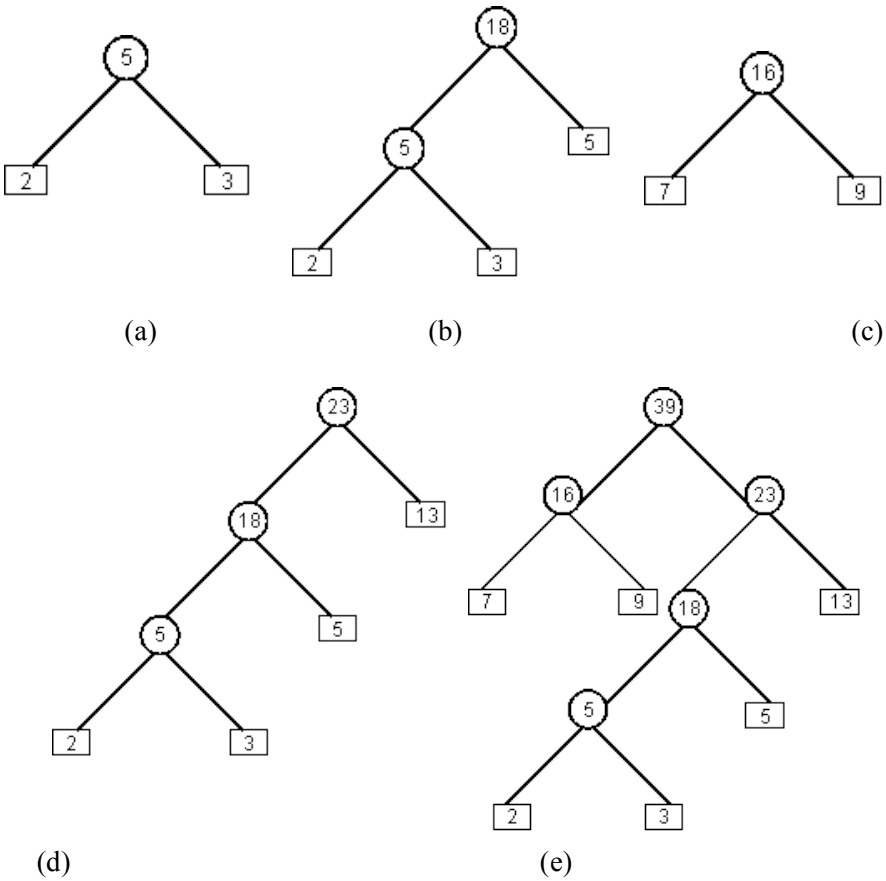
Huffman'o funkcija pradeda darbą su pradiniais duomenimis, kuriuos sudaro n apibendrintų dvejetainių medžių:

- kiekvienas turi po vieną viršūnę
- jie yra masyve *heap*[],
- kiekviena medžio viršūnė turi tris laukus: *weight*, *left_child* ir *right_child*,
- kiekvienos viršūnės svoris yra q_i.

Algoritmo vykdymo metu galios taisyklė: bet kuriam medžiui heap'e, turinčiam šakninę viršūnę *tree* ir didesnį už 1 gylį, visų išorinių viršūnių medyje ilgių suma yra *tree -> weight*. *Huffman'o* funkcija naudoja funkcijas *least* ir *insert*; funkcija *least* pašalina medį, kuris turi mažiausią svorį *heap'e,* o funkcija *insert* įterpia naują medį į *heap'ą*. Šios funkcijos atlieka veiksmus per tiesinį laiką. Huffman'o programa:

```
void huffman ( tree_pointer  heap [ ],  int  n )
{
   tree_pointer  tree;
   int  i;
   initialize (heap ,  n);
   for  (i  =  1;  i  <  n;  i++ )
     {
       tree  =  (tree_pointer)
                   malloc (sizeof (tree_node) ) ;
       if  (IS_FULL (tree)
         fprintf (stderr, "The  memory  is  full/n");
         exit (1) ;
       }
       tree ->left_child  =  least (heap, n – i + 1);
       tree ->right_child = least (heap, n – i );
       tree ->weight = tree ->left_child ->weight +
       tree ->right_child ->weight;
       insert (heap, n – i – 1, tree);
   }
}
```

Iliustruojant algoritmo veikimą, galima pateikti pavyzdį. Tariant, kad svoriai yra q_1=2, q_2=3, q_3=5, q_4=7, q_5=9 ir q_6=13, gaunama medžių seka, pavaizduota pav 3. 9.



Pav. 3.9. Huffman'o funkcijos sudarytas sąlajos medis.

Šio medžio svertinis išorinio kelio ilgis yra:

$$2 \cdot 4 + 3 \cdot 4 + 5 \cdot 3 + 13 \cdot 2 + 7 \cdot 2 + 9 \cdot 2 = 93$$

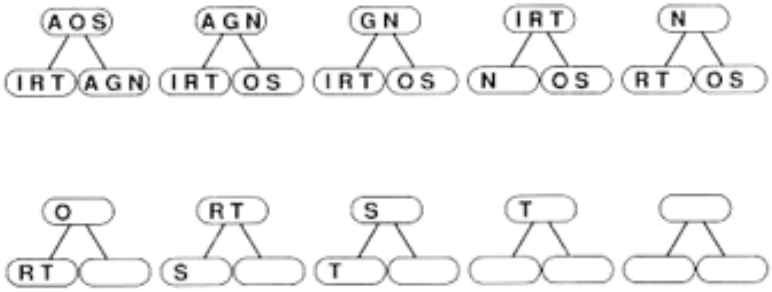
Palyginimui, geriausias pilno binarinio medžio svertinis išorinio kelio ilgis yra 95.

Savybė: *Huffman'o algoritmo analizė – heap'o sudarymas užima O(n) laiko, pagrindinis ciklas for yra vykdomas n-1 kartų, kiekvienas kreipimasis į least užima O(log n) laiko. Asimptotinis algoritmo laikas yra O(n log n).* □

3.8. Pakeitimų išrinkimas

Rūšiavimo-sąlajos algoritmams galima natūraliai ir efektyviai pritaikyti prioritetines eiles. Pirmiausia sąlajos metu, kai reikia išrinkti iš suliejamų elementų minimalų, tikslinga naudoti *heap* struktūrą ir jos operaciją *replace*, kuri pakeičia iš karto ankstesnio algoritmo dvi operacijas - minimalaus elemento rašymą į išorinę atmintį ir naujo elemento įterpimą. Taip sumažinamas atliekamų operacijų skaičius. Aišku, *heap* struktūrą reikia naudoti nuosekliai, t.y. jau pirminiame etape, kai elementai skirstomi į blokus ir rūšiuojami, tikslinga juos išdėstyti į *heap* struktūrą ir surūšiuoti tik iš dalies.

Be to (tai yra dar svarbiau), prioritetinių eilių struktūros panaudojimas įgalina sąlajos metu gauti ilgesnius surūšiuotus blokus, negu jie tilptų į vidinę atmintį. Kai pradiniai duomenys yra pertvarkomi į *heap* struktūrą ir minimalus elementas yra keičiamas nauju, reikia papildomai naudoti tokią taisyklę: jei naujas elementas, rašomas vietoje minimalaus elemento senoje *heap* struktūroje, yra mažesnis už jį, reikia nuo šio elemento pradėti naują bloką ir naują *heap* struktūrą, interpretuojant jį esant didžiausiu šioje struktūroje. Algoritmai, naudojantys šias taisykles, yra vadinami pakeitimo išrinkimo (*replacement selection*) vardu. Pav. 3.10 pateikia pavyzdį, kaip veikia toks algoritmas.



Pav. 3.10. Pakeitimo išrinkimas duomenų blokams iš trijų raidžių.

Algoritmai, sukurti naudojant išdėstytus ir panašius principus, turi tokias jų efektyvumą vertinančias savybes:

- rūšiavimo-sąlajos algoritmai, rūšiuojantys N įrašų, kai vidinė atmintis talpina M įrašų ir duomenys rašomi į (P+1)-ą išorinį įrenginį, perrinks šiuos duomenis $1 + \log_P(N/2M)$ kartų;
- pakeitimo išrinkimo algoritmai, esant atsitiktiniams duomenims, formuoja duomenų blokus dvigubai ilgesnius negu naudojama *heap* struktūra.