1. Algoritm. Algoritmi keerukus. Ajalise keerukuse asümptootiline hinnang. Erinevad keerukusklassid: kirjeldus, näited.

1.1 Algoritm

- Mingi meetod probleemi lahendamiseks, mida saab realiseerida arvutiprogrogrammi abil.
- Algoritm on õige, kui kõigi sisendite korral, mis vastavalt algoritmi kirjeldusele on lubatud, lõpetab ta töö ja annab tulemuse, mis rahuldab ülesande tingimusi. Öeldakse, et algoritm lahendab arvutusülesande.
- Selline programm, mis annab probleemile õige vastuse piiratud aja jooksul.
- Kindlalt piiritletud sisendi korral vastab ta järgmistele kriteeriumitele:
    - lõpetab töö piiratud aja jooksul;
    - kasutab piiratud hulka mälu;
    - annab probleemile õige vastuse.
- Parameetrid, mille järgi hinnata algoritmide headust:
    - vastava mälu hulk;
    - töötamise kiirus ehk vajatava aja hulk.

Omadused:

1. **Lõpplikkus** – töö peab lõppema peale lõpliku arvu sammude läbimist.
2. **Määratletus** – iga samm peab olema nii täpselt määratud (rangelt ja ühemõtteliselt), et seda suudaks täita isegi arvuti. Kui täidetavaid samme on liiga palju, siis algoritm ei ole praktiliselt täidetav.
3. **Sisend** – algoritmil on sisendandmed, mis pärinevad alati kindlat liiki objektide hulgast. Nende hulk võib olla ka null. Mida rohkem andmeid, seda rohkem aega kulub nende töötlemiseks.
4. **Väljund** – üks või mitu töötulemust.
5. **Efektiivsus**
    a. Korrektne – peab andma õige tulemuse
    b. Efektiivne – peab leidma vastuse mõistliku ajaga
    c. Programmi efektiivsusele hinnangu andmist nim. algoritmi keerukuse uurimiseks
        i. Ajaline
        ii. Mahuline
6. **Korrektsus** – lahendab etteantud ülesande õigesti.

Algoritmi **tegelik tööaeg** ja **efektiivsus** sõltub andmete hulgast, protsessori kiirusest, arvutist, kompilaatorist jne.

1.2 Algoritmi keerukus

- On hinnang sellele, kuidas algoritmi poolt esitatavad nõudmised ajale muutuvad näiteks siis, kui probleemi mõõt kasvab. Keerukus mõjutab jõudlust, kuid mitte vastupidi.
- *On põhioperatsioonide arvu sõltuvusfunktsioon sisendi suurusest.*
- ***Põhioperatsioon:*** *üks tehe, üks tsüklitingimus või üks rida*
- Keerukusprobleemidega tegeleb vastav teadus – **arvutuslik keerukusteooria.**

Ajalise keerukuse uurimine	**Mahulise keerukuse** uurimine
algoritmi alusel koostatud programi **tööaja hindamine**	programmi tööks kasutatava **mälu mahu hindamine**

- **Keerukusfunktsiooni kasvukiirus** – kui kiiresti kasvab antud algoritmi järgi koostatud programmi ressursivajadus töödeldavate andmete mahu suurenemisel.
- **Keerukusfunktsiooni** leidmiseks on võimalik kokku arvutada kõik sammud, mida arvuti teeb ülesande lahendamiseks. Pole võimalik väljendada konkreetse arvuga, vaid andmete hulgast (n) sõltuva valemina.

Algoritmi keerukus **halvimal** juhul	Algoritmi **keskmine** keerukus	Algoritmi keerukus **parimal** juhul
Teatud algandmete komplekti juures võib kuluda aega tunduvalt rohkem keskmisest; selgitakse välja, millised on halvimad variandid ja kui tihti nad esinevad. **Kriitilistel juhtudel** peetakse õigeks arvestada ennekõike just halvima juhu keerukusega.	Igasuguste halvemate ja paremate juhuste keskmine, mis aga tavaliselt kipub suvaliselt tekkivate andmete puhul olema halvimale üsna lähedal.	Mingi algandmete komplekti puhul võib algoritm töötada tunduvalt kiiremini. Need juhused on huvitav väljaselgitada, paraku pakub see enamasti vaid teoreetilist huvi, sest kasutada saab seda teadmist harva.

1.3 Ajalise keerukuse asümptootiline hinnang

Asümptootiline hinnanguga väljendatakse funktsiooni väärtuse muutumise üldist trendi – funktsiooni kasvamise kiirust. **Asümptoot** on sirge, millele funktsiooni graafik lõpmatult läheneb, kuid millega ta ei lõiku. **Suure O** tähistust kasutakse algoritmide keerukuse tähistamiseks. Reeglina antakse hinnang halvima juhu jaoks ja tegelik tööaeg peaks olema parem. Suure O järgi **saab hinnata algoritmi tööaja suhtelist kasvu andmehulga suurenemisel**. Kuna hinnang on ligikaudne kaob mõte täpselt kõiki tehteid üle lugeda. Oluline on N-i järk (kus **N on töödeldavate andmete hulk** ehk probleemi mõõt). Hinnangud hakkavad kehtima alles N-i suurte väärtuste korral.

O. def: Funktsioon g(N) on O(f(N)), kui leiduvad konstandid C0 ja N0, nii et g(N)<c0*f(N) kõigi N>N0 korral

- O(g(n)) on funksioonide hulk, mis ei kasva kiiremini kui g(n).

- g(n) on funktsioon, mis kirjeldab algoritmi saamude arvu ja sellest tulenevalt tööaja seost sisendi mahuga (n). Näiteks võib funktsiooniks g(n) olla n, n^2 jms

Konstant C0-ga:

- püütakse likvideerida vead, mis tekivad matemaatiliselt sammude väljaarvutamisel või programmi analüüsides ebaoluliste lausete vahelejätmise tõttu
- et võimaldada klassifitseerida algoritmid tööaja ülemise piiri järgi

1.4 Erinevad keerukusklassid: kirjeldus, näited

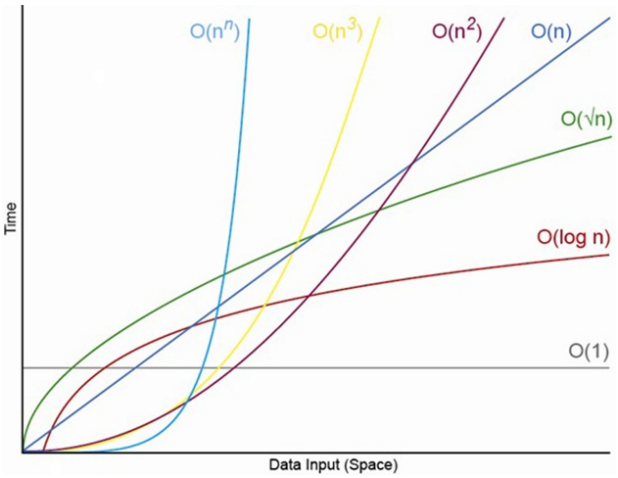
Tööaja hindamiseks on vaja peamist tähelepanu pöörata kasutavatele keelekonstruktsioonidele – st algoritmi või programmi struktuurile.

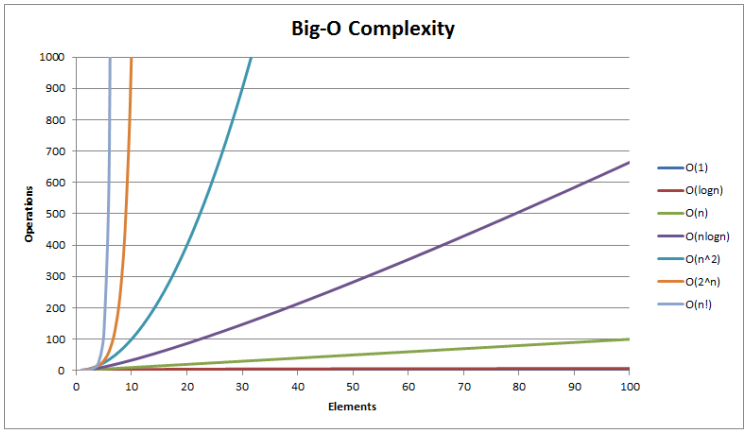
O(1)	O($\log_2 n$) või O(log n)	O(n)	O(n $\log_2 n$) või O(n log n)
Konstantne	Logaritmiline	Lineaarne	Linearitmeetiline?
• Andmehulgast ei sõltu algoritmi tööaeg. • Kõiki programmi lauseid täidetakse üks kord. • Lahendamiseks on tavaliselt olemas valem.	• Tööaeg kasvab väga aeglaselt N-i kasvades. • Log n kasvab vaid 2 korda kui n^2. • Probleemi lahendakse selle järkjärgulisel vähendamisel kordades või muutmisel väiksemaks probleemiks	• Kui N kasvab 2 korda, kasvab ka tööaeg 2 korda • Iga elementi on vaja töödelda mingil määral.	• Tavaliselt tähendab seda, et lineaarses algoritmis on $\log_2 n$ algoritm. • Algoritm lahendab probleemi nii, et alguses tükeldab väiksemateks osadeks. • Kui kõik nn alamprobleemid on lahendatud, siis kogu lahenduse saamiseks need ühendatakse kokku.
Paisksalvestus; tuvastamaks, kas täisarv on paaris või paaritu	Kahendotsimine – otsitav piirkond aheneb igal sammul 2 korda, Parallelsed ja jagatud algoritmid	Lineaarne otsimine (leida kõige väiksem number massiivis) Vektorite sisestamine, väljastamine, liitmine, lahutamine ja skalaarkorrutis	Poolitussortimine, kiirsortimine, mestimisega sorteerimine, kuhjaga sorteerimine, timsort

O(n^2)	O(n^3)	O(2^n)	O(n!)
Ruutkeerukus	Kuupkeerukus	Eksponentsiaalne	Faktoriaalne
• Andmehulga kasvamisel 10 korda suureneb tööaeg 100 korda. • Enamasti 2 tsüklit üksteise see ja mõlemad sõltuvad algandmete hulgast.	Enamasti 3 tsüklit üksteise sees, mis kõik sõltuvad algandmete hulgast.	• Kui N=10 on aeg 1000, • Suurendades N-i 20 korda, kasvab tööaeg 1000000-ni. • Ebapraktiline algoritm.	

• Sobilik on selline algoritm suhteliselt väikest mõõtu probleemide lahendamiseks. • Nullsortimine, valiksortimine, maatriksite sisestamine, väljastamine, liitmine ja lahuta...	• Sobilik vaid väikeste andmetehulkade korral. • Maatriksite korrutamine.	Jõumeetodil lahendused, kõigi variantide täisläbivaatused.	n elemendi kõigi võimalike järjestuste leidmine

Arvuti töökiirus (operatsioonid sekundis)	Probleemi mõõt 10^6		
	O(n)	O(n log n)	$O(n^2)$
10^6	tunnid	tunnid	lootusetu
10^9	sekundid	sekundid	aastad
10^12	kohe	kohe	nädalad





Legend

Excellent | Good | Fair | Bad | Horrible

Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Timsort	O(n)	O(n log(n))	O(n log(n))	O(n)
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)

2. Algoritmimise strateegiate lühike iseloomustus ja kasutamise näide (Brute-force ehk jõumeetod, Greedy method ehk ahne algoritm, Divide and Conquer ehk jaga ja valitse).

2.1 Brute-force ehk jõumeetod

- Leiab lahenduse **ebaefektiivselt**, tavaliselt vaadates **läbi kõikvõimalikud** lahendused ja teed
- **Kergesti arusaadav ja väljamõeldav**
- Sõltub **lähteandmete** iseloomust, hulgast ja sellest, mida otsitakse, kas selline meetod on sobiv
- Tuleb läbi proovida **kõik võimalused** ja valida välja **parim**

2.1.1 Nõrgad küljed:

- Enamasti on tegemist **aeglase** meetodiga
- **Nõuavad paljude sammude sooritamist**
- **Tihti võimatu täita**, sest keerukusklass võib kerkida O(N!)-ni

2.1.2 Tugevad küljed:

- Jõumeetodil lahenduse uurimine viib tavaliselt **probleemist parema arusaamise juurde ehk ta on kui mõtlemise strateegia.**
- **Väikeste algandmete hulga** juures võib sellist lahendust **paberil läbi mängida** ja muutub probleem arusaadavaks
- Jõumeetodil töötavad algoritmid on **lihtsad, paremini arusaadavad, kergemini realiseeritavada ja veakindlamad**

2.1.3 Näide:

Valiksorteerimine, mullisosrteerimine, Sequential search

Leida arvu 625 kõik tegurid. Lahenduskäik: alustatades 1-st ja lõpetades 625 jagada arv läbi kõigi arvudega. Kui arv jagub (jääk on 0), siis on järgmine tegur leitud.

2.2 Greedy method ehk ahne algoritm

- Algoritmitüüp on sobiv **optimiseerimisülesannete lahendamiseks**.
- Optimiseerimisül. Otsib kõigi kandidaatide hulgast mingit alamhulka (valitute hulka), mis rahuldaks teatud tingimusi. Tingimuseks on enamasti **mingi max või min väärtuse leidmine** ja vastavalt on ka tehtud valikufunktsioon.

2.2.1 Nõrgad küljed:

- **Ei anna alati vastuseks optimaalset tulemust** ja kui tulemus on ka optimaalne, on seda väga **raske tõestada**.

2.2.2 Tugevad küljed:

- Paljudel juhtudel on teda kergem koostada
- Töötab kiiremini kui DP algoritm

Optimiseerimise juures on vajalikud teatud tingimused:

1. **Kandidaatide hulk** (graafi tipud, teede pikkused, rahatähtede suurused...)
2. **Valitute hulk**, mis või kes on juba **kasutatud** (sobivaks tunnistatud, tagasi antud rahatähed, läbitud graafi tipud...)
3. **Eeldatav lahendus,** otsitav summa vms, mille järgi saab otsustada, kas välja valitud kandidaadid moodustavad lahendused (ei pruugi olla optimaalne)
4. **Jätkamise näitaja,** mille järgi saab otsustada, kas kandidaatide hulka saab suurendada, et lahendust leida.
5. **Valikufunktsioon,** mille abil valitakse uusi kandidaate väljavalitute hulka
6. **Vastusefunktsioon,** mis annab lõpliku väärtuse lahendusele

2.2.3 Näide kasutamisest:

Ahnet algoritmi on sobiv kasutada siis, kui **alamülesannete optimaalsed lahendused** annavad tulemuseks **kogu probleemi optimaalse lahenduse**. **Valiksorteerimise algoritm** (igal sammul otsitakse vähimat arvu, mida sorteerimata massiiviosa algusesse tõsta). **Seljakoti probleem**. Varga eesmärk on seljakotti sisse panna võimalikult suure summa eest kraami. (kaks variatsiooni: diskreetne – asju ei ole võimalik tükeldada ja pidev – osade kaupa)

2.3 Divide and Conquer ehk jaga ja valitse

Kogu meetod on oma olemuselt rekkursiivne (st jaga ülesanne tükkideks ja tükid omakorda tükkideks kuni saadakse sobiva suursega tükid, mida on paras lahendada)

- Probleem jagatakse mitmeks alamprobleemiks, mis lahendatakse üksteisest sõltumatult.
- Seejärel ühendateakse alamprobleemide lahendused nö alt üles ja saadakse lahendus kogu probleemile.

2.3.1 Omadused:

- **Minimaalne sisendi mõõt n0** – kui probleemi suurs on alla selle ei hakata probleemi jagama.
- **Alamprobleemi suurus**, milleks kogu probleem jagatakse – milline suurus on paras?
- Jagamisel saadavate **alamprobleemide arv** – liiga palju alamprobleeme pole ka hea
- **Algoritm**, mida kasutakse alamprobleemide lahenduste ühendamiseks, sellest sõltub ka lahenduse efektiivsus

2.3.2 Tugevad küljed:

- Konseptuaalselt raskete probleemide lahendamine
- Paralleelsus – mitmetuumaliste protsessorite rakendamisel
- Aitab avastada efektiivseid algoritme
- Cachemälu kasutamine on efektiivsem

2.3.3 Nõrgad küljed:

- Tugevate külgede vastandid ☺

2.3.4 Näide kasutamisest:

- **Kiirsorteerimine** ja **mestimisega sorteerimine**. Mõlemad algoritmid on rekursiivsed ja jaotavad mingi skeemi järgi kogu ülesannet tükkideks, et need sorteerida ja hiljem osad ühendada.
- Jaga ja valitse tüüpi strateegiat kasutavad ka **otsimiskahendpuu** ja **kahendotsimise** algoritmis.

3. Andmestruktuur. Andmestruktuuri loogiline tase ja realisatsiooni tase.

3.1 Andmestruktuur

- **Andmete talletamise ja organiseerimise viis**
- Vahend **suure hulga andmete organiseerimiseks ja salvestamiseks arvutis** ning neile efektiivse juurdepääsu tagamiseks
- Andmestruktuurid jaotuvad üldise ülesehituse järgi: **lineaarsed ja mittelineaarsed**. Nad tuginevad arvuti võimetele salvestada ja võtta andmeid **mälust aadressi järgi.**
- Lineaarsed andmestruktuurid on loendid, kus **elementide vahel on järgnevussuhe**
- **Lihtsaim füüsiline struktuur** andmete mälus hoidmiseks **on masiiv(id).**

- **Loogiliseks struktuuriks** on **andmete jada** – andmed on järjestatud, lineaarsed, igale andmeelemendile eelneb ja järgnev alati üks element. On oluline, kes või mis on jadas esimene ja viimane jne.
- Ühemõõtmeline massiiv, kus on üliõpilaste nimekiri (loend): seoseks võib olla järjestus tähestistiku alusel.

Oluline on vahet teha andmestruktuuri kahel aspektil: loogilisel ja realisatsiooni tasemel.

Andmestruktuuri elemendi jaoks kasutatakse tavaliselt järgmisi mõisteid:

- **Sõlm** *(node)* – andmeelement tabelis, üldisemalt struktuuris (ka **kirje**, **objekt, element**). Koosneb ühest või mitmest **infoväljast** ja ühest või mitmest **viidaväljast.**
- **Väli** (võtme- ja infoväli) – sõlm koosneb mitmest baidist arvutimälus ja loogilisel tasemel mitmest väljast info hoidmiseks. Lisaks on vajalikud väljad, et **luua seoseid struktuuri teiste elementidega**.
- **Võtmeväli** – selle järgi saab näiteks **sorteerida ja otsida**, vastavalt eesmärgile võib võtmeväljaks olla kord üks ja kord teine **infoväli**.
- **Sõlme aadress** (ka link, viit sõlmele) – sõlme esimese baidi aadress arvuti mälus. *→*Mummuga nool sümboliseerib aadressi, mumm peaks paiknema seal, kuhu aadress kirjutatud on.*

3.2 Andmestruktuuri loogiline tase

- **Teatud omadustega struktuur:**
  - Andmete järjestus (eelmine, järgmine, ...)
  - Operatsioonid (elemendi lisamine, eemaldamine, ...)
- Kirjeldab struktuuri **loogilist ülesehitust** ja selle esitamiseks sobivad hästi nooled, kastid jms graadfilised võtted.
- Operatsioonide selgitamiseks aga **pseudokood** või muud üldised vahendid algoritmi üleskirjutamiseks.
- On kirjeldus struktuuri käitumisest ja sellest, kuidas me teda **tajume**.

3.3 Realisatsiooni tase – andmestruktuuri füüsiline esitus

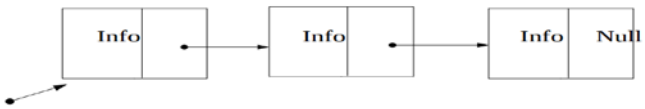
- Näitab, kuidas vastav struktuur tegelikult arvutis üles ehitatakse ja kuidas tegelikult toimuvad tema peal vajalikud operatsioonid.
- Sama loogilist struktuuri saab tihti **realiseerida mitmel erineval viisil** ning sõltub keelest ja olukorrast, milline variant on otstarbekam.
- **Tavaliselt staatiline või dünaamiline**; lähtub mäluaadresside kasutamisest
- Aadressid arvutakse välja või salvestatakse koos andmetega

4. Ühe viidaga ahelad. Peamised tegevused ahelaga: elemendi lisamine, elemendi kustutamine, ahela läbimine - tekstiline ja pildiline kirjeldus.

4.1 Ühe viidaga ahelad

Ahel on madala taseme struktuur, mis koosneb viitade abil ühendatud elementidest (nn sõlmedest) Ahela abil saab ehitada lineaarse loendit dünaamiliselt.

Ühe viidaga ahel koosneb **peast** (head), mis viitab ahela esimesele elemendile ja selle külge aheldatud ning omavahel seotud **ahela ülejäänud elementidest**.



Joonis 1 Pildiline kirjeldus ühe viidaga ahelast ning selle läbimisest.

- **Esimene sõlm: pea (head), viimane sõlm (tail)**.
- Ahela kohta on teada **esimese sõlme aadress**, järgmised sõlmed saadakse kätte esimesest sõlmest lähtudes.
- Viimase elemendi (saba) viidaväljas on **tühi aadress** (NIL või Null), selle järgi saab tuvastada **ahela lõppemisest.**
- Seega viidaväljade abil ühendatakse sõlmed ühte struktuuri: ahelasse.

4.2 Ahela läbimine

4.2.1 Tekstiline

Alustades ahela peast liigutakse sõlm haaval ahela lõpuni. Vajaduse korral tehakse iga sõlmega midagi (uuritakse võtmevälja väärtust vms). Abimuutuja **current** peab olema viidatüüpi. Ta tuleb kasutusele võtta selleks, et kogu ahelale viitav **Head** kaotsi ei läheks.

4.2.2 Pildiline

Joonis 1 Pildiline kirjeldus ühe viidaga ahelast ning selle läbimisest.

4.3 Elemendi (sõlme) lisamine

4.3.1 Tekstiline

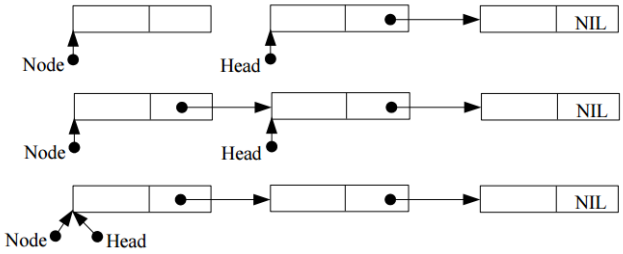
Sõlme lisamine võib toimuda ahela algusesse, keskele või lõppu:

- **Esimese** elemendi lisamisel muutub **ahela pea väärtus** (kõige lihtsam ja kiirem)
- Elemendi **keskele** lisamisel tuleb jälgida, et **ahel ei katkeks** (kui on vaja säilitada näiteks sorteeritud järjekorda)
    - Abiks tuleb võtta üks või kaks viita ja läbida ahel õige kohani

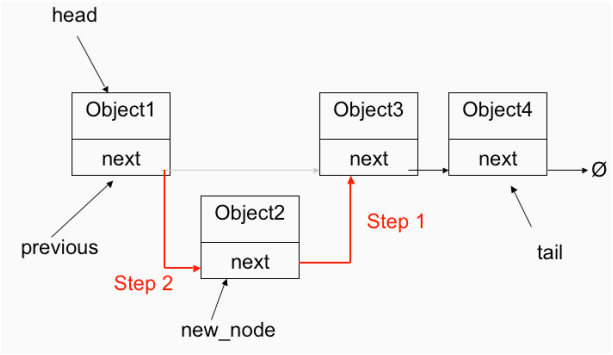
- Võib abiks võtta 2 viita: Current ja Prev. Viimane on pidevalt ühe elemendi võrra Curr-ist tagapoolt. Lisamine toimub Prev-i ja Current-i vahele.

- **Viimase** elemendi lisamisel tuleb uue sõlme **viidavälja NIL** kirjutada

4.3.2 Pildiline

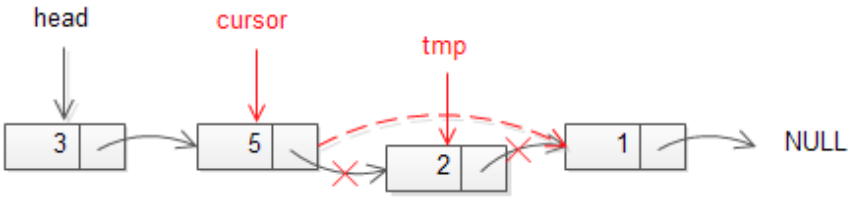


Pilt 2. Kolm sammu elemendi lisamisel ahela algusesse.

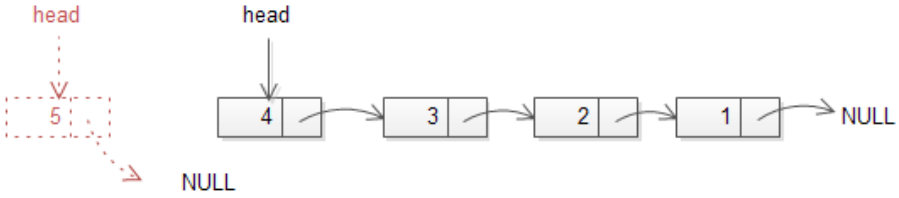


4.4 Elemendi kustutamine

Kustutav sõlm otsitakse tavaliselt üles infovälja või võtmevälja väärtuse järgi. Mugavam on kustutada kasutades kahte abiviita **(sarnaselt lisamisele)**

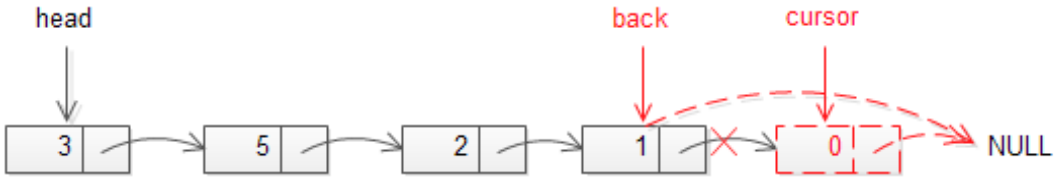


We point the head to the next node and remove the node that the head pointed to.



To remove a node from the back of the linked list, we need to:

- Use two pointers: cursor and back to track the node.
- Start from the first node until the cursor pointer reaches the last node and the back pointer reaches the node before the last node.
- Set the next pointer of the back to NULL and delete the node that the cursor points to.
- If the node has only 1 element, set the head pointer to NULL before removing the node.



5. Pinu: omadused, operatsioonid. Näited pinu kasutamisest. Pinu realiseerimine arvutis.

5.1 Pinu: omadused, operatsioonid.

Magasin ehk **pinu** (stack) on lineaarloend, kuhu elemente lisatakse ja kust elemente kustutatakse ühest ja samast otsast, pinu tipust (top). Andmete kättesaamine toimub reeglina sel teel, et element eemaldatakse pinust.

- Omadus: mis esimesena sisse pandi, saab kätte kõige viimasena ja vastupidi – viimasena paigaldatud eseme saab kätte esimesena.(*LIFO*)
- Operatsioonid.
  - Elemendi **lisamine** pinusse (push)
  - Elemendi **eemaldamine** pinust (pop)
  - Uue pinu **loomine**.
  - **Kontroll**, kas pinu on **tühi**.
  - **Kontroll**, kas pinu on **täis** (ruum uute elementide lisamiseks on otsa saanud).

5.2 Näited pinu kasutamisest

- Funktsioonide väljakutse organiseerimine
- Avaldiste teisendamine, kontrollimine, arvutamine ja muu aritmeetiliste avaldiste töötlemine (sulge mugavalt uurda)
- Igasuguse info tagurpidi pööramiseks
- Abivahend andmete hoidmisel, kus viimati listud väärtust tuleb kohe töötlema hakata
- Alamprogrammide väljakutse organiseerimine arvutis
  - Uue alamprogrammi väljakutse tähendab seda, et programmi täitmine jääb teatud kohas pooleli ja peab peale alamprogrammi töö lõppemist jätkuma samalt kohalt. Selleks pannakse pooleli jäänud alamprogramm pinusse koos oma muutujate komplektiga. Kui momendil töötav alamprogramm oma töö lõpetab, võetakse pinust välja kõige peal olev alamprogramm ja kogu väljakutse loogika on just selline, et see on see õige, millega edasi minna.

5.3 Pinu realiseerimine arvutis

5.3.1 Staatiline meetod kasutades massiivi

- Pinu maht **piiratud**
- Pinu tipu meeles pidamiseks tuleb arvet pidada täidetud massiivelementide üle
- **Staatiline mälukasutusega**
- Elemendid on **füüsiliselt järjestikku**
- Massiivi lahter sisaldab infot ühe pinu elemendi kohta

5.3.2 Dünaamiline meetod kasutades ühe viidaga ahelat

- Tuleb otsustada, **kumba pidi viidad jooksevad**, et *push* ja *pop* protseduure lihtsam kirjutada oleks
- **Piisab ühest viidast** pinu tipule, et temaga peamisi operatsioone teha
- Ei pea paiknema füüsiliselt järjestikku, vaid järgnevusseose määravad viidad
- Elemendid võiksid viidata olemasoleva pinu poole (nö alla poole)

6. Postfiks avaldis ehk Pööratud Poola kuju (Reverse Polish Notation). Mis see on, kuidas teisendatakse tavaliseks infiks avaldiseks ja vastupidi.

Nii loogika kui ka aritmeetikaavaldisi saab kirja panna kolmel erineval kujul: **prefiks** (+ab , Poola kuju, operandid on avaldises ees.), **postfks** (ab+ , pööratud poola kuju, operandid järel) ja **infiks** (a+b) kujul.

6.1 Postfiks avaldis ehk pööratud Poola kuju

– (ab+) viis kuidas panna kirja loogikaavaldisi **sulge kasutamata**. **Operatorid pannakse operandide järele**.

Avaldise **postfiks kujule teisendamine** (teisendusalgoritm eeldab, et kõigi tehete järjekord on määratud **sulgudega**):

- Arv kirjuta väljundisse
- Vasakut sulgu ignoreeri
- Operaator (tehtemärk) pane pinusse
- Parempoolse sulu puhul võta operaator pinust ja kirjuta väljundisse

Postfiksilt infiks kujule:

- Postfikskujul avaldis kirjutada pinusse (vasakpoolseim liige pinusse kõige ülemiseks(top) ja parempoolseim liige kõige alumiseks(bottom))
- Võtta pinust ülevalt poolt järjest operande ning kirjutada välja(nii kaua kui tuleb operaator)
- Kui tuleb operaator, kirjutada see kahe viimati väljakirjutatud operandi vahele, ning lisada sulud ümber
- Jätka tegevust

(5*((9*8)+(7*(4+6))))					
	Väljund	Pinu			
1. samm	5				
2. samm	5	*	push		
3. samm	59	*			
4. samm	59	**	push		
5. samm	598	**	pop		
6. samm	598*	*			
7. samm	598*	*+	push		
8. samm	598*7	*+			
9. samm	598*74	*+*	push		
10. samm	598*74	*+*+	push		
11. samm	598*746	*+*+	pop		
12. samm	598*746+	*+*	pop		
13. samm	598*746+*	*+	pop		
14. samm	598*746+*+	*	pop		
15. samm	598*746+*+*				
1. samm	598*746+*+*				
2. samm	98*746+*+*	5			
3. samm	8*746+*+*	5,9			
4. samm	*746+*+*	5,9,8			
5. samm	746+*+*	5,(9*8)			
6. samm	46+*+*	5,(9*8),7			
7. samm	6+*+*	5,(9*8),7,4			
8. samm	+*+*	5,(9*8),7,4,6			
9. samm	*+*	5,(9*8),7,(4+6)			
10. samm	+*	5,(9*8),(7*(4+6))			
11. samm	*	(5*((9*8)+(7*(4+6))))			

Table 3: An Expression with Parentheses

Infix Expression	Prefix Expression	Postfix Expression
(A + B) * C	* + A B C	A B + C *

Table 4: Additional Examples of Infix, Prefix, and Postfix

Infix Expression	Prefix Expression	Postfix Expression
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+ + + A B C D	A B + C + D +

7. Järjekord: omadused, operatsioonid. Näited järjekorra kasutamisest. Järjekorra realiseerimine arvutis.

7.1 Järjekord: omadused

- **Lineaarloend**, kus elementide lisamine toimub alati loendi ühes otsas ja elementide eemaldamine teisest otsast
- **Andmete vaatamiseks** pääseb ligi vaid selles otsas, kust toimub eemaldamine
- **FIFO**, kes esimesena järjekorda lisatakse, see võetakse ka välja esimesena

7.2 Peamised operatsioonid

1. **Lisada** element järjekorra lõppu
2. **Eemaldada** element järjekorra algusest

Lisaks:

3. Uue tühja järjekorra **loomine**
4. Kontroll, kas järjekord on **tühi**
5. Kontroll, kas järjekord on **täis**

7.3 Näited järjekorra kasutamisest

- Kui andmed saabuvad kindlas ajalises järgnevuses ja saabumise järgnevus on oluline ka andmete töötlemisel
    - Näiteks **operatsioonisüsteem** paneb saabunud käsud/päringus järjekorra lõppu ja võtab neid järjekorra algusest täitmiseks.
    - Näiteks lennuki **piletite broneerimissüsteem**. Broneerimissoovid saabuvad erinevatel ajahetkedel, samas järjekorras tuleks need soovida ka rahuldada ja broneeringud teha.
- Eriliigid:
    - Mitme teenindajaga järjekorrad
    - Eelistusjärjekord – mõni peab saama teenindatud varem, kuid sama prioriteediga tegelaste vahel kehitivad ikka järjekorra reeglid.
    - Piiratud pikkusega järjekord

7.4 Järjekorra realiseerimine arvutis

7.4.1 Staatiline ehk kasutades massiivi

- Massiivina realiseerimiseks peab järjekorra jaoks **meeles pidama kahte välist indeksit** (`algus` ja `lõpp`).
- Andmeid **listakse algusesse** ja **eemaldatakse lõpust**.

- Seega sarnaselt pinule muudetakse nii elemendi lisamisel kui ka kustutamisel **vastavaid indekseid**.
- Kui algus ja lõpp on tagurpidi (`lõpp > algus`), siis on järjekord tühi
- Et massiiv otsa ei saaks, siis tuleb järjekord uuesti massiivi algusesse tuua, kui see on lõppu jõudnud

7.4.2 Dünaamiline ehk kasutades ahelat

- Kasulik ja vajalik, et **viitasid oleks kaks**, mis viitaks ahela **esimesele** ja **viimasele** sõlmele. Nõnda on lisamine kiira ja mugav.
- Põhjus on järjekorra eripäras, sest ligi on **vaja pääseda** nii **järjekoora algusele** kui ka **lõpule**.
- Järjekorda on kasulikum hoida **nö tagurpidi**, sest nii saab palju mugavamalt elementi eemaldada.

8. Puu. Üldine puu. Kahendpuu. Järjestatud (orienteeritud) ja järjestamata (orienteerimata) puu. Puuga seotud erinevad mõisted. Puu ülesmärkimine sulgavaldisena ja Dewey kümnendesitusena. Puu läbimise järjekorrad (pre-, post- ja inorder). Puu realiseerimine arvutis.

8.1 Puu. Üldine puu

- **Graaf** on mittelineaarne struktuur, mille abil saab modelleerida objektide hulgas paari-kaupa esinevaid suhteid ja seoseid.
- **Puu** on graafi erivorm.
- Puus ühendatakse andmeobjektid **hierhilisel** viisil.
- Puu koosneb elementidest, mida nim. **tippudeks** ehk **sõlmedeks** (siia paigutakse andmed/info), ja seosetest tippude (sõlmedes oleva info) vahel, mida nim. **kaarteks**.
- Iga puu sõlm on juureks mõnele alampuule. Sõlme kõigi alampuude arvu nimetatakse selle sõlme **järguks.** Sõlm, mille järk on 0, on **leht,** Ülejäänud sõlmed on **hargnevad sõlmed.**
- Puu sõlmed jagunevad paiknemishierarhia järgi **tasemetesse.** Juur on tasemel 0, juure järglased on tasemel 1 jne. Vastavalt tasemete arvule mõõdetakse ka **puu kõrgust**.
- *Puu on **täielik**, kui tema kõigil tasemetel on max võimalik arv sõlmi ja kõik lehed paiknevad samal tasemel.*
- *Mitmed tegevused puus on O (log N) keerukusega.*

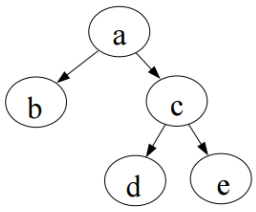
8.2 Järjestamata ja järjestatud puud

- **Järjestamata,** kui ühe tipu laste omavaheline järjestuse ei ole oluline. Järjestamata puu on **orienteeritud**, sest hierarhilised seosed on olulised.
- **Järjestatud**, kui ühe tipu järglaste järjestus on oluline, st räägitakse 1., 2., 3. jne pojast.

8.3 Puuga seotud erinevad mõisted

- **Sõlm** – element, millest puu koosneb ja kus paiknevad andmed
- **Kaar** – seos tippude (sõlmedes oleva info) vahel
- **Juur** – sõlm, millele ei eelne mitte ühtegi sõlme ja seda on ainult üks.
- **Leht** – sõlm, mille järk on 0 (talle ei järgne ühtegi sõlme)
- **Sõlme järk** – sõlme alampuude arv
- **Puu järk** – suurim võimalik sõlme järk antud puus
- **Vanemad** – sõlmed, millel on lapsed/järglased
- **Üks vanem** – igal sõlmel on ainult üks vanem
- **Lapsed** – sõlmed, millel on vanemad
- **Lehed** – sõlmed, millel järglased puuduvad
- **Vend** – sõlm, millel on sama vanem
- **Eellased** – kõik antud sõlmest kõrgemal (juure poole) olevad sõlmed
- **Järglased** – kõik antud sõlmest allpool olevad sõlmed
- **Tee** – ainus, lühim kaarte järgnevus, mis viib puu juurest leheni. Puu juure ja konkreetse lehe vahel on alati ainult üks tee
- **Mets** – järjestatud hulk, mis koosneb 0 või mitmest mittelõikuvast puust
- **n-järku puu** – puu, mille kõigi sõlmede maksimaalne laste arv on piiratud arvuga n

8.4 Puu ülesmärkimine



8.4.1 Sulgavaldisena

(a(b) (c(d) (e)))

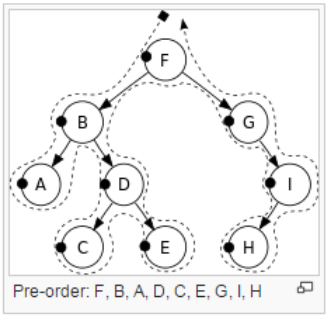
8.4.2 Dewey kümnendesitusena

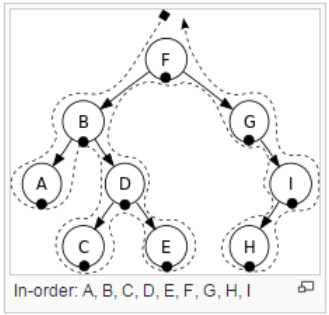
1 a; 1.1 b; 1.2 c; 1.2.1 d; 1.2.2 e

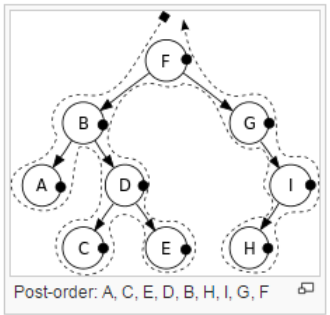
8.5 Kahendpuu

- On **tippude lõplik hulk**, mis on tühi või koosneb juurest ja **kahest mittelõikuvast alampuust**, mida nim vasakuks ja paremaks alampuuks.
- Kahendpuu igal sõlme on **max kaks alampuud** (2-järku puu)
- Iga alampuu puhul on vahe, kas ta on vasakpoolne või parempoolne
- Võrreldes tavalise puuga, siis kahendpuu puhul peetakse ka **tühja alampuud** puuks

8.6 Puu läbimise järjekorrad (pre-, post- ja inorder)







8.6.1 Lõppjärjekord (Postorder e. Endorder)

Vanema ja tema kahe järglase kohta kehtib järgmine läbimise järjestus:
vasak järglane, parem järglane, vanem

1. Läbi vasak alampuu.
2. Läbi parem alampuu.
3. Väljasta (töötle) juur (tegevust korratakse iga alampuu jaoks)

8.6.2 Eesjärjekord (Preorder)

Vanema ja tema kahe järglase kohta kehtib järgmine läbimise järjestus**:**
vanem, vasak järglane, parem järglane.

1. Väljasta (töötle) juur.
2. Läbi vasak alampuu.
3. Läbi parem alampuu (igal alampuul on oma juur, mida väljastada).

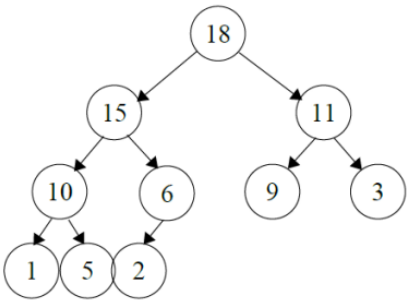
8.6.3 Keskjärjekord (Inorder)

Vanema ja tema kahe järglase kohta kehtib järgmine läbimise järjestus:
vasak järglane, vanem, parem järglane.

1. Läbi vasak alampuu.
2. Väljasta (töötle) juur.
3. Läbi parem alampuu (loomulikult on igal alampuul jälle oma juur, mida väljastada).

8.7 Puu realiseerimine arvutis

8.7.1 Staatiliselt massiivina



Joonis 1 Kahendkuhi puuna

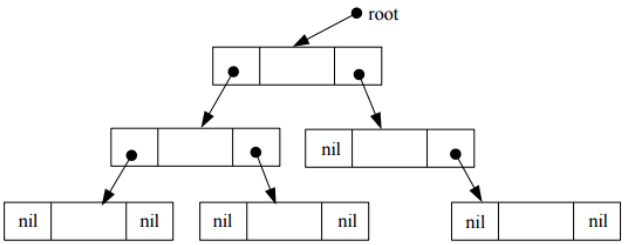
1	2	3	4	5	6	7	8	9	10
18	15	11	10	6	9	3	1	5	2

- Oluline on **indeksite olemasolu**.
    1. Massiivi elemendis hoida lisaks infole ka kummagi järglase asukoha indeksit

2. Kastuda mingit reeglit laste ja vanemate indeksite väljaarvutamiseks. Tüüpilised reeglid i-nda elemendi jaoks on järgmised:
   2.1. Vasak laps – 2*i
   2.2. Parem laps – 2*i +1
   2.3. Vanem – i/2 (täisarvuline jagamine)

8.7.2 Dünaamiliselt ahelana

- Dün. realisatsioon on eelistatud, sest ei nõua esialgset **suurt mälu eraldamist** ja on ka loomulikum.
- Puu iga sõlm sisaldab lisaks infole **kahte viita**: `LLINK` ja `RLINK`
- Puuga on seotud ka **nn puuviit `ROOT`**
- Kui puu on tühi: `T = NULL,` vastasel juhul on `ROOT` väärtuseks on puu juure aadress
- Kui mingi sõlme üks alampuudest on tühi, siis kirjutakse vastavasse viidaväljasse tühja viida tähis `NULL/nil`



8.8 Puude kasutamine

Kasutatakse arvuti mälus andmestruktuurina:

- Avaldised jt keele osad **süntaksipuuna**.
- Erinevad **otsimispuud** otsimise kiirendamiseks (kahendotsimispuu)
- **Kahenkuhi** kiireks elementid paigutamiseks ja kättesaamiseks
- Ka **otsustamispuud**, **koodipuud** jne

9. Graaf. Graafiga seotud mõisted. Suunatud ja suunamata graaf. Atsükliline graaf. Kaalutud graaf. Graafi ülesjoonistamine ja realiseerimine arvutis. Graafi algoritmid: topoloogiline sorteerimine, sügavuti otsimine, laiuti otsimine, lühim tee kaalutud graafis e Dijkstra algoritm): algoritmi kirjeldus koos väikese näitega.

9.1 Graaf

- Graafi võib kirjeldada kui andmestruktuuri, mis **ei pea olema lineaarne**.
- Graaf on kõige üldisem võimalus andmete vaheliste **seoste kujutamiseks**
- Graaf on **struktuur**, mille abil saab modelleerida objektide hulgas **esinevaid paari-kaupa suhteid/seoseid.**

- Graaf koosneb **tippudest** ja tippe ühendavatest **kaartest (servadest).**

9.2 Graafiga seotud mõisted

- **Suunatud graaf (orienteeritud)** – Graaf, mille kaartel **on suund**, st iga kaare jaoks on määratud, millisest tipust ta algab ja millises tipus lõppeb. Teisisõnu või öelda, et graafi tipud on paari kaupa järjestatud.
- **Suunamata graaf (orienteerimata)** – seos kahe tipu vahel on mõlemas suunas. See kehtib kõigi graafi servade kohta. Joonisel nooli ei märgitata. *Tee saab minna läbi kaare mõlemat pidi.*
- **Tee** ehk ahel
    - Saab graafis leida ühest tipust teise tippu.
    - Teel on **pikkus.**
    - Selline kaarte järgnevus, kus ühe kaare lõpp-punkt on järgmise kaare alguspunktiks.
    - Kaks tippu v ja u on **seotud**, kui nende vahel on tee.
        - Kaks tippu on seotud **külgnevussuhtega**, kui ühest tipust läheb kaar teise tippu
- **Elementaarahel** – ahel, mis läheb igast **tipust** vaid ühe korra.
- **Lihtahel** – ahel mis läheb igast **servast** vaid ühe korra.
- **Tsükkel** – ahel, kus algus ja lõpp on samas tipus.
- **Hamiltoni tsükkel** – elementaartsükkel (elementaarahel, mis lõppeb samas tipus), mis läbib **kõiki graafi tippe.**
- **Euleri tsükkel –** lihttsükkel (lihtahel, mis lõppeb samas tipus), mis läbib **kõiki graafi servi** ühe korra.
- **Atsükliline graaf** – graaf, kus puudub tsükkel
- **Suunatud atsükliline graaf** – graaf, kus puudub suunatud tsükkel
- **Graafi kaalud** – kaartel olevad arvud, mis kannavad infot lisaks seoseinfole (nt: seoste tugevus, pikkus vms)

- **Täielik** – kui graafil on kaared kõigi tippude vahe

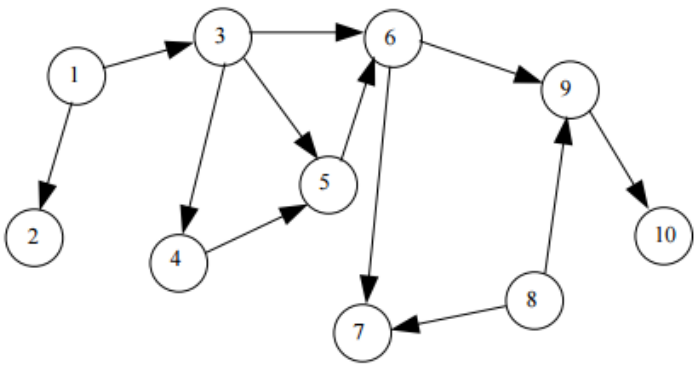
9.3 Graafi ülesjoonistamine ja realiseerimine arvutis

9.3.1 Staatiline realisatsioon

Esitab graafis olevaid tippudevahelise seoseid **külgnevusmaatriksina.** Veerud = read = tipud. Igas lahtris, kas 0 (False) või 1 (True). *Programmeerides on vaja graafi jaoks deklareerida kahemõõtmeline massiiv, mille elemendi on kas täisarvud või ka boolean-tüüpi väärtused.*

9.3.2 Dünaamiline realisatsioon

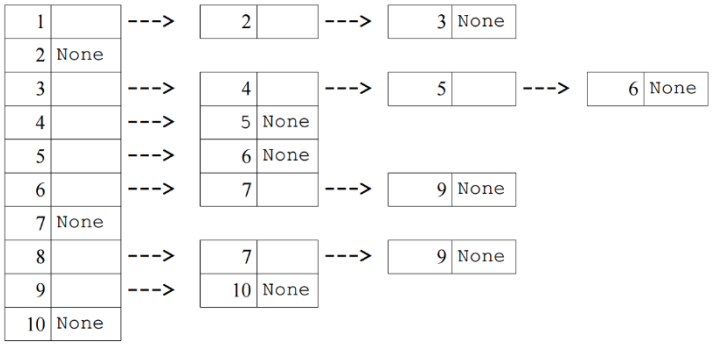
- Hõredama graafi kujutamiseks võetakse kasutusele **külgnevusloend**
- Graafi tippudest moodustatakse massiiv
- Iga tipu jaoks on üks lahter
- Iga tipulahtri külge kinnitakse lineaarahel nendest tippudest, mis külgnevad antud tipuga
- Loendi lõpus on tühi viit (None)
- Mälu hoitakse kokku sellega



Joonis 1. Suunatud graaf

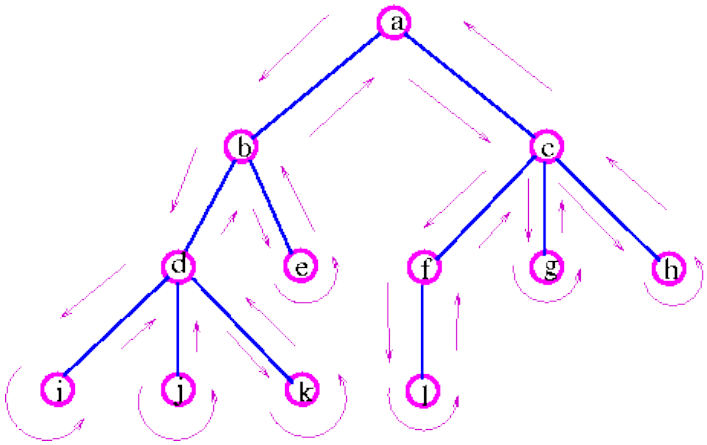
	1	2	3	4	5	6	7	8	9	10
1	0	1	1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	1	1	1	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0
6	0	0	0	0	0	0	1	0	1	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	1	0	1	0
9	0	0	0	0	0	0	0	0	0	1
10	0	0	0	0	0	0	0	0	0	0

Joonis 2. Joonisel 1 oleva graafi külgnevusmaatriks



9.4 Sügavuti otsimine

- **Sarnane labürindi läbimisele**
- Minnakse ühte teed pidi **nii sügavale, kui see võimalik on**
- Kui **naabrid otsas**, siis minnakse tagasi ja otsitakse uut teed
- Nii jätkatakse kuni leidub **veel uurimata tippe**
- Kui sellisel viisil rohkemate tippude juurde ei pääse, kuid on veel uurimata tippe, võetakse neist suvaline ja korratakse tegevust



- Iga tipp saab sattuda vaid ühte **otsimispuusse** ja seega puud ei lõiku
- Algoritmi kasutamiseks sobib nii **orienteeritud** kui ka **orienteerimata** graaf
- Algoritm: nagugi laiuti otsimine otsimisel tippe värvitakse kasutades kolme erinevat värvi: **valge, hall, must**. Lisaks iga tipuga seotakse **kaks ajatemplit**: märgitakse siis, kui tipp esimest korda avastati ja siis, kui tipp on lõplikult töödeldud ja mustaks värvitud

9.4.1 Näide kasutamisest:

- Saab kasutada graafis tsüklite leidmiseks
- Kahe tipu vahelise tee leidmiseks
- Min toespuu leidmiseks

9.5 Laiuti otsimine

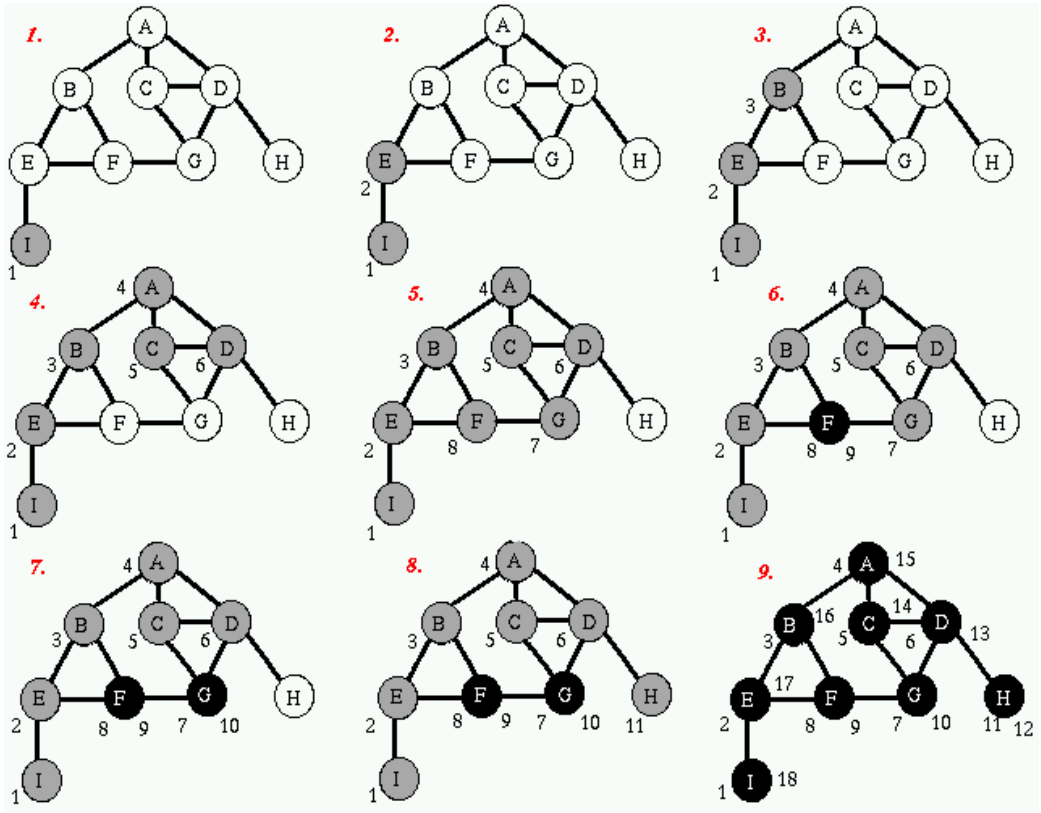
- Kasutada ülesannete puhul, kus otsitakse **parimat lahendust** ja **see tuleks välja sõeluda** paljude võimaluste hulgast
- Kõige tüüpilisemaks ülesandeks on leida **lühimat teed kahe tipu vahel**
- Tippude vahele jäävaid kaarte arvu loetakse tee pikkuseks
- Kõiki lahendusvariante uuritakse paralleelse, mitte ei võrrelda hiljem

9.5.1 Lahenduskäik

- Võetakse **lähtetipp**
- Kõigepealt otsitakse tipud, kuhu saab **ühe kaare läbimisel**, siis **kahe kaare läbimisel** jne
- Kui järgmine tee on pikem, siis seda ei fikseerita
- Nii toimides läbitakse lõpuks kõik tipud ja saadakse teada kaarte arv algustipust igasse tippu.

9.5.2 Algoritm

- **Valge** – tipuni pole veel jõutud
- **Hall** – tipuni on jõutud, tema eellane on fikseeritud, kuid temaga külgnevad tipud pole veel kõik läbi uuritud
- **Must** – tipu järglased on läbi uuritud ja rohkem selle tipu kallale minna ei tohi



9.5.3 Näide kasutamisest

Laiuti otsimiese algoritm on aluseks teiste algoritmide koostamisele graafiprobleemide lahendamiseks.

9.6 Topoloogiline sorteerimine

- Graaf on **atsükliline** ja **orieteeritud**, siis on graafi tippude vahel olemas **osaline järjestus**.
- Topoloogilise sorteerimise eesmärgiks on saada selline tippude järgnevus, kus **iga tippu töödeldakse enne** kui neid tippe, millele ta osutab.
- Õigeks vastuseks tavaliselt mitu erinevat järgnevust.
    - Kõigepealt tuleb leida üles need tipud, kuhu ühtegi kaart ei sisene. Parem on neid leida külgnevusmaatriksist, liikudes selleks mööda veerge. Kui mõnes veerus 1-d puuduvad, ongi eellasteta tipp leitud.
    - Kui tipp on paigutatud sorteeritud jadasse, tuleb kõigilt tema järglastelt üks eellane maha kustutada. (-1)
    - Järgmisel sammul saab sorteeritud jadasse panna taas neid tippe, millel eellaste arv on 0.

0.

2	3	5	7	8	9	10	11
1	0	0	0	2	2	2	2

1.

2	3	5	7	8	9	10	11
1	0	0	0	1	2	2	1

2.

2	3	5	7	8	9	10	11
1	0	0	0	1	2	2	0

3.

2	3	5	7	8	9	10	11
1	0	0	0	0	2	1	0

4.

2	3	5	7	8	9	10	11
0	0	0	0	0	1	0	0

5.

2	3	5	7	8	9	10	11
0	0	0	0	0	0	0	0

6.

2	3	5	7	8	9	10	11
0	0	0	0	0	0	0	0

7.

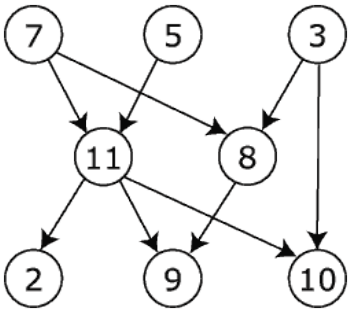
2	3	5	7	8	9	10	11
0	0	0	0	0	0	0	0

8.

2	3	5	7	8	9	10	11
0	0	0	0	0	0	0	0

1. **7**
2. **5**
3. **3**
4. **11**
5. **8**
6. **2**
7. **9**
8. **10**

topoloogiline sorteerimine visuaalselt vasakult-paremale ja ülevalt-alla



9.6.1 Näide kasutamisest

Saab lahendada tööde järgnevuse planeerimise ülesannet. Kui tuleb arvestada sellega, et sorteerimise tulemusi võib olla mitu, seega on võimalikud ka mitu erinevat tööde järjekorda.

9.7 Lühim tee kaalutud graafis e Dijkstra algoritm

- **Suunamata graafist tehakse suunatud graaf**
- Graafis **ei tohi olla tsüklit**, kus kaarte pikkuste summa tuleks negatiivne
- Töötab **ahne algoritmi** põhimõttel, st igal sammul tehakse lokaalselt parim otsus, need otsused viivad kogu probleemi lahenemiseni
- Tööks vajalik **tippude tabel**, kuhu kirjutatakse iga tipu jaoks tema kaugus lähtetipust ja tipu number, kust antud tippu jõuti.
- Kaalutud graafis liidetakse kaalud ning **väljastatakse lühim tee**
- Rakendatakse while-tsükli, töötab seni, kuni kõigi graafi tippude naabrid on läbiuuritud.
- *Üldine idee on väga sarnane laiutiotsimisele, selle vahega, et juba leitud teepikkused võib muuta, juhul kui tuleb välja mõni lühem tee.*

9.7.1 Näide kasutamisest

Kõige lühema tee leidmine kaardil, kui on teada, et punktist A (Tartu) viib punkti B (Tallinn) mitu erinevat teed.

- Kasutatakse kolme massiivi. Massiivid peavad olema nii suured, et oleks ruumi kõigi graafi tippude jaoks.
- **Node** – tipu number koos märkega, kas tipp on "lõpuni" töödeldud
- **Label** – tipu kaugus lähtetipust

- **Prev** – eelmise tipu number (tipp, kust antud tippu satuti).

Dijakstra algoritmi lahendus

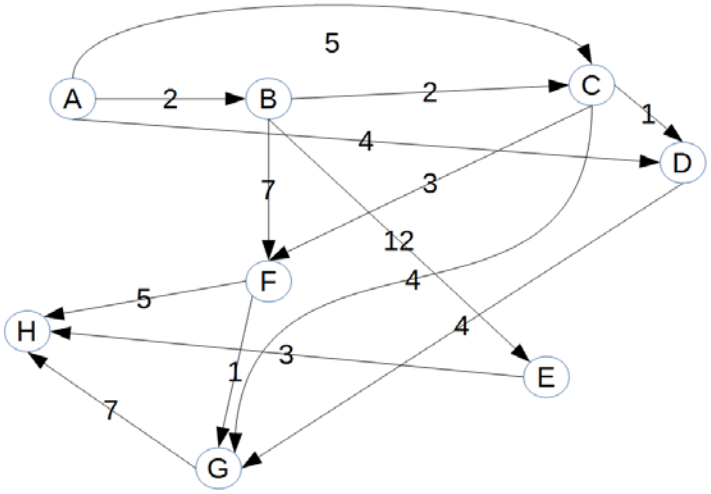
Min kagusega tipp, mida tuleb võtta järgmisel uurimisel aluseks

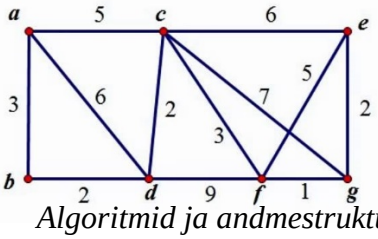
Teepikkuse parandus

0. samm	Node	A	B	C	D	E	F	G	H
	Label	0	999	999	999	999	999	999	999
	Prev								
1. samm	Node	A*	B	C	D	E	F	G	H
	Label	0	0+2	0+5	0+4	999	999	999	999
	Prev		A	A	A				
2. samm	Node	A*	B*	C	D	E	F	G	H
	Label	0	2	**2+2**	4	2+12	2+7	999	999
	Prev		A	**B**	A	B	B		
3. samm	Node	A*	B*	C*	D	E	F	G	H
	Label	0	2	4	4	14	**4+3**	4+4	999
	Prev		A	B	A	B	**C**	C	
4. samm	Node	A*	B*	C*	D*	E	F	G	H
	Label	0	2	4	4	14	7	8	999
	Prev		A	B	A	B	C	C	
5. samm	Node	A*	B*	C*	D*	E	F*	G	H
	Label	0	2	4	4	14	7	8	7+5
	Prev		A	B	A	B	C	C	F
6. samm	Node	A*	B*	C*	D*	E	F*	G*	H
	Label	0	2	4	4	14	7	8	12
	Prev		A	B	A	B	C	C	F
7. samm	Node	A*	B*	C*	D*	E	F*	G*	H*
	Label	0	2	4	4	14	7	8	12
	Prev		A	B	A	B	C	C	F
8. samm	Node	A*	B*	C*	D*	E*	F*	G*	H*
	Label	0	2	4	4	14	7	8	12
	Prev		A	B	A	B	C	C	F

Lühim tee A-st H-sse: H > F > C >B > A

12 = 5 + 3 + 2 + 2





v	*a*	*b*	*c*	*d*	*e*	*f*	*g*
a	0_a	3_a	5_a	6_a	∞_a	∞_a	∞_a
b	0_a	3_a	5_a	5_b	∞_a	∞_a	∞_a
c	...						

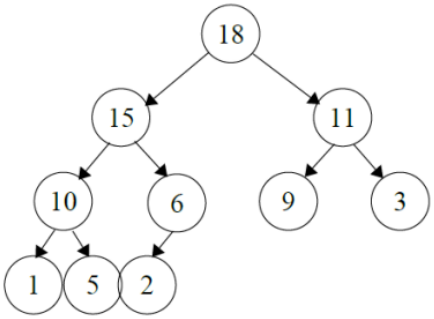
a (läbi b) -> d on lühem, kui otse a->d. sellepärast on 5_b

10. Kahendkuhi - mis teda iseloomustab, kuidas realiseeritakse, milliste ülesannete jaoks

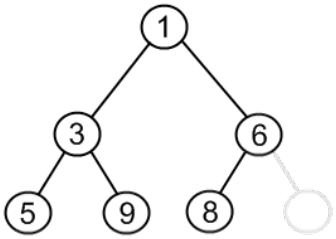
Selline kahendpuud, kus peaksid olemad täidetud järgmised tingimused:

1. Igas tipus olev väärtus ei tohi olla väiksem kui selle tipu järglastel
2. Lehtede sügavus ei tohi erineda rohkem kui 1 taseme võrra
3. Viimane tase täitub vasakult paremale

- Andmestruktuur on tavaliselt realiseeritud **massiivina** ja puu juur on element indeksiga 1, edasi tulevad juure järglased 2 ja 3 jne
- Kahendkuhja kasutatakse **ka prioriteetidega järjekorra realiseerimiseks**.
- Sel juhul paikneb kõrgeima prioriteediga element kuhja tipus (massiivi 1. lahtris) ja peale tema eemaldamist tuleb kuhi ringi ehitada



Joonis 1 Kahendkuhi puuna



1	2	3	4	5	6	7	8	9	10
18	15	11	10	6	9	3	1	5	2

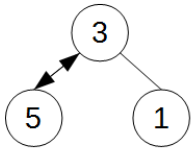
11. Sorteerimisülesanne. Sorteermine kuhjaga (Heaps.), lisamissorteerimine (Insertion s.), mestimisega sorteerimine (Merge s.), loendamissorteerimine (Counting s.). Meetodite keerukus, tugevad ja nõrgad küljed. Mõtle ka näitele algoritmi töö selgitamiseks.

11.1 Sorteerimine kuhjaga (Heaps sort)

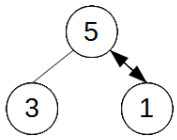
- Meetod kasutab kahendkuhja
- **Suuremat vajadust lisamälu järele ei ole.**
- Sorteerimine toimub kahes etapis:
  1. Arvudemassiivist moodustatakse väärtuste ümberpaigutamise teel kuhi.
  2. Kuhi sorteeritakse vastava algoritmiga.
  - Kui massiivist on sel viisil kuhi moodustatud, järgneb sorteerimine:
  1. Tipmine element võetakse kuhjast ära (tema on kõige suurem), selleks vahetatakse 1. ja viimane element ning kuhja suurust vähendatakse ühe võrra.

2. Kuhi moodustatakse uuesti sel teel, et tippu sattunud väike element viiakse vastava protseduuriga oma õigele kohale.

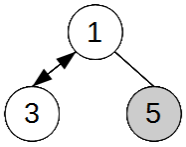
Sorteerida numbrid 351 kahanevas järjekorras.



1. Panen arvud puusse.

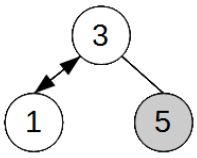


2. Teen maksimaalse kahendkuhja nii, et järglased oleksid vanema tipu väärtusest väiksemad.

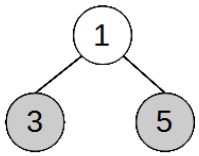


3. Kuhi on moodustatud, hakkan sorteerima. Vahetan omavahel ära viimase ja esimese indeksi olevad väärtused
4. Panen massiivi:

5		

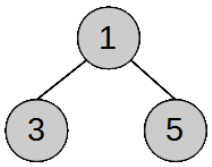


5. Teen uuesti maksimaalse kuhja.



6. Vahetan väärtused ja panen massiivi:

5	3	



7. Kuna element on alles vaid 1, siis puu on maksimaalne ja paneme viimase väärtuse massiivi:

5	3	1

Keerukus: Kahendpuu maksimaalne kõrgus on log n. Seega ajaline keerukus ei saa ületada log n. Siit tulenevalt saame kuhja abil sorteerimise keerukuseks O(n log n).

Eripärad: Sorteeritud massiiv hakkab tekkima massiivi lõpust.

11.1.1 Tugevad küljed

- Põhiline eelis: ta on efektiivne
- **Halvima puhul tõestatud keerukus: O(n log n)**
- **Sorteerib kohapeal, seega nõuab vaid O(1) lisamälu (mälu efektiivsus)**
- The heap sort algorithm is not recursive
- In-place algorithm: an algorithm that transforms input using a data structure with a small, constant amount of storage space

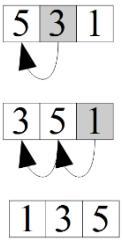
- Best at sorting huge sets of items because it doesn't use recursion
- If the array is partially sorted, Heap Sort generally performs much better than quick sort or merge sort

11.1.2 Nõrgad küljed

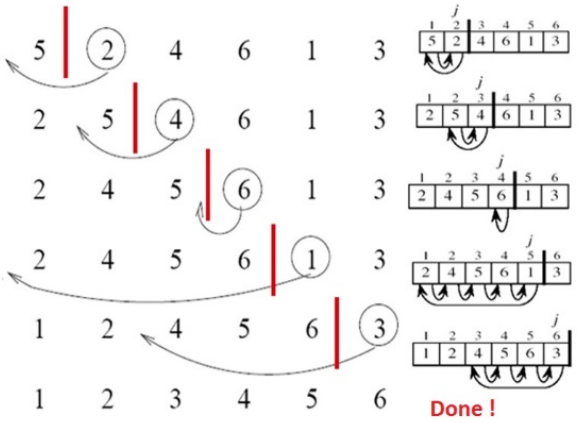
- Aeglasem kui kiirsorteerimine ja mestimisega sorteerimine
- Raske realiseerida
- Ebastabiilne
- Peaaegu sorteeritud massiiviga töötab samakaua kui kaootiliselt sorteeritud
- Algoritmi rakendamine on probleematiline, kui soovitakse kasutada cache mälu
- Ei toimi ahelaga (linked listiga), sest tahab momentaalset ligipääsu saada

11.2 Lisamissorteerimine (Insertion s.)

- Sorteeritud on vasakpoolne massiivi osa, kuid mitte lõplikult.
- Paremalt poolt võetakse järgmine element ja sobitatakse ta sorteeritud poolele õigesse kohta vahele.
- Esimeseks arvuks tuleb massiivi lisada väga väike arv, millest väiksemat sorteeritavate kirjete hulgas ei leidu. Seda on vaja, et töö käigus mitte üle minna massiivi algusest.



- Alustades 2. elemendist:
  1. Võta kirje.
  2. Leia talle sobiv kohta temast vasakul olevate kirjete hulgas (selleks tuleb teda võrrelda vasakule poole jäävate kirjetega, kuni õigekoht on leitud).
  3. Nihuta kirjed eest ära, et paigutada vaatlusalune kirje oma kohale.
  4. Korda tegevust kõigi kirjetega kuni massiivi lõpuni.



- **Keerukus: Halvimal ja keskmisel juhul O(n2) ning parimal juhul O(n) (sõltuvalt massiivi eelnevast sorteeritusest).**
- **Eripärad:**
  - Massiivi sorteerimisel tekitab rohkem raskusi vahelepanekuks ruumi tegemine - kui arv lisatakse rea algusesse, tuleb nihutada kõiki ülejäänud kirjeid. Sobib paremini juhul, kui üksik uus kirje on vaja õigesse kohta lisada.
  - Või dünaamilise nimistu sorteerimiseks, kus kirjeid ei ole vaja füüsiliselt ümber paigutada

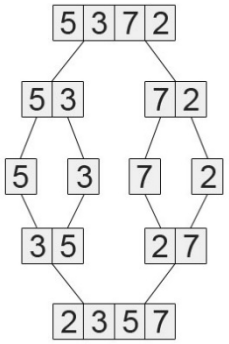
11.2.1 Tugevad küljed

- Lihtne rakendada
- Efektiivne väiksemate andmekomplektide puhul
- Adaptiivne, see on efektiivne nende andmekomplektidega, mis on enam-vähem sorteeritud
- Stabiilne, ei muuda relatiivset järjekorda elementidel, millel on sama väärtus
- Online, st saab sorteerida listi samal ajal kui see suureneb/sisse jookseb

11.2.2 Nõrgad küljed

- Suurte andmestike puhul vähem efektiivne
- Kui elementide arv suureneb, siis programmi kiirus väheneb
- Nõuab palju elementide liigutamist

11.3 Mestimisega sorteerimine (Merge s.)



- Algoritm koosneb kahest osast – massiivi jagamisest ning kahe osa ühendamisest.
    1. Jaga algandmed kaheks enam vähem võrdseks osaks.
    2. Sorteeri kumbki osa eraldi.
    3. Kombineeri mõlemad osad kokku üheks sorteeritud massiiviks.
- Olemuselt on see algortim rekursiivne ja toetub otseselt lähenemisele "Jaga ja valitse" (divide et impera). Rekursioonist väljudes ühendatakse massiivi osi järjest omavahel, saades nii üha pikemad sorteeritud lõigud. Vajab täiendavat mälu ajutiste massiivide tegemiseks (reaalselt sama palju kui on sorteeritavaid andmeid)
- **Keerukus: O(n log2n) nii halvimal kui ka keskmisel juhul.**

11.3.1 Tugevad küljed

- Stabiilne
- Toimib hästi koos virtuaalse ja cache mäluga
- Saab tööd jagada protsessorite vahel
- Ei oma “raskeid” sisendandmeid
- Hea sorteerida suurte andmete hulki, mis ei mahu mälus ära
- Hea aeglaselt ligipääsetavate andmete sorteerimiseks, nt kõvaketas
- Suurepärane nende andmete sorteerimiseks, mis jooksevad järjestikuliselt. Nt kõvaketas, linked list

11.3.2 Nõrgad küljed

- Peaaegu sorteeritud andmetega töötaba samakaua kui kaootiliste andmetega
- Nõuab lisamälu

11.4 Loendamissorteerimine (Counting s.)

1. samm	andmed	1	2	1	0
	loendur	1	2	1	
		0	***1***	***2***	
2. samm	loendur	1	3	4	
		0	***1***	***2***	
3. samm j=3	loendur	0	3	4	
		0	***1***	***2***	
	massiiv	0			
4. samm j=2	loendur	0	2	4	
		0	***1***	***2***	
	massiiv	0		1	
5. samm j=1	loendur	0	2	3	
		0	***1***	***2***	
	massiiv	0		1	2
6. samm j=0	loendur	0	1	3	
		0	***1***	***2***	
	massiiv	0	1	1	2

11.4.1 Keerukus

- **Kirjeldatud algoritm on ajaliselt keerukuselt lineaarne O(N).**
- $O(n+k)$, where n is the size of the sorted array and k is the size of the helper array (range of distinct values).

11.4.2 Tugevad küljed

- Hea ajalise keerukusega

11.4.3 Nõrgad küljed

- Algoritmi kasutusvaldkond on piiratud - tema abil sobib sorteerida **positiivseid täisarve**, mis on kindlates piirides
- Algoritm **nõuab täiendavalt mälu** kahe massiivi jagu: loendamiseks ja uue sorteeritud massiivi moodustamiseks. Seega hea ajalise keerukusega kaasneb suur mäluline keerukus.
- Negatiivsete numbritega ei toimi

11.4.4 Näide algoritmi töö selgitamiseks

1. Loendurmassiivi algväärtustamine
2. Erinevate massiivis olevate väärtuste loendamine
3. Igale arvule eelnevate arvude kokku lugemine
4. Arvude paigutamine uude massiivi vastavalt leitud kohale

3 massiivi: andmete massiiv, loendurmassiiv ja uus massiiv:

12. Otsimisülesanne. Jadaotsimine. Kahendotsimine. Otsimisalgoritmide keerukus.

12.1 Otsimisülesanne

- Otsimine tegeleb probleemida, kuidas koguda andmed arvuti mällu ja meetoditega, kuidas konkreetseid andmeid sealt leida saab.
- Oluline on organiseerida materjal selliselt, et ta oleks võimalikult kiiresti kättesaadav.
- Enamasti kasutatakse andmete otsimiseks mingit identifikaatorit nn **võtit K** (mis on unikaalne).
- **Otsimisülesanne** – on N kirjet ja nende hulgast on vaja leida üks konkreetne kirje, mille võti vastab otsitavale võtmele K. Vastav kirje tagastatakse või antakse teada, et sellise võtmega kirjet ei leitud.

12.2 Jadaotsimine/järjestikotsimine

- **Idee** – alusta algusest ja võrreldes iga kirje võtit otsitava võtmega jätka nii kaua, kuni on leitud otsitav võti, seejärel peatu. Või jõua peale kõigi kirjete läbivaatamist arusaamisele, et sellise võtmega kirje puudub.
- On kõige **lihtsam.** Saab lõppeda edukalt kui ka edutult.
- **Keerukus** on **lineaarne O(N).**
- Kui otsitav element on 1. lahtris, on ta käes esimese sammuga, kui elementi pole, tuleb kogu massiiv läbi vaadata, et selles veenduda.
- **Võtmete paiknemise** kohta tabelis igasugune eeldus puudub.
- Toimib hästi massiiviga, kuid võib kasutada ka lineaarse nimistu korral.

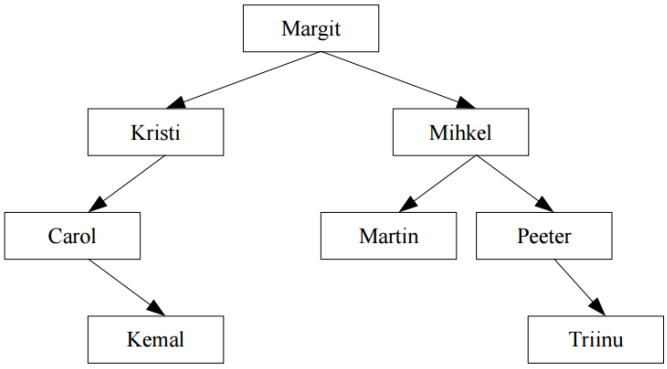
12.3 Kahendotsimine

- Eeldus: tabel peab olema **järjestatud**.
- Võtmed paiknevad nõnda: $k_0<k_1<k_2<\ldots<k_n$
- Massiiv **jagatakse pooleks**, keskel olevat võtit **võrreldakse otsitava võtmega**. Kui otsitav võti on sellest väiksem, võib tabeli ülemise poole kõrvale jätte. Edasi jagatakse tabeli alumine pool jne.
- Sobib **kasutada massiivi** jaoks, kus on kerge indeksi järgi leida nö keskmist kirjet.
- **Keerukus** – O (log n), seega tegemist on päris hea meetodiga.

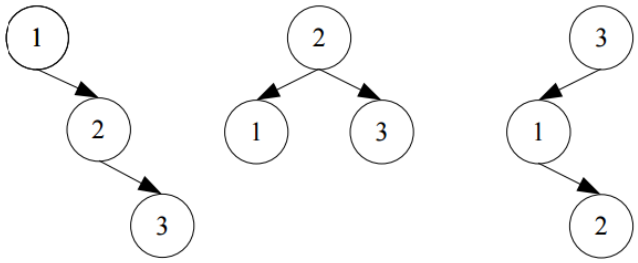
13. Kahendotsimispuu. Andmete lisamine, otsimine ja kustutamine, operatsioonide keerukusklassid. AVL-puu: omadused, kuidas töötab, milleks kasutatakse. Puna-must puu: omadused, kuidas töötab, milleks kasutatakse.

13.1 Otsimiskahendpuu

- Kui infot on vaja kiiresti lisada ja kustutada, kuid ka efektiivselt otsida.
- Dünaamiliste andmete korral sobib paremini viitada abil ehitav kahendpuu, kui dün. struktuur.
- Reeglid:
    - Iga tipu vasakpoolse järglase võti on alati selle tipu võtmest väiksem.
    - Iga tipu parempoolse järglase võti on selle tipu võtmest suurem.



Pilt nr 1 Otsimiskahedpuu nimedest. Nimed lisatakse: Margit, Kristi, Mihkel, Peeter, Martin, Carol, Triinu, Kemal.



Pilt nr 2. Samadest väärtustest koostatud otsimiskahendpuud. Lisamise järjekord: a) 1-2-3, b) 2-1-3 või 2-3-1, c) 3-1-2

13.2 Andmete (tipu) lisamine

- **Alustada puu juurest** ja võrrelda iga tipu võtmeväärtust lisatava elemendi väärtusega
- Kui uue tipu võti on **väiksem**, siis liigutakse **vasakusse** alampuusse
- Kui uue tipu võti on **suurem**, siis liigutakse **paremasse** alampuusse
- Nõnda toimitakse **iga tipu juures**
- Kui **alampuus edasi minna ei saa**, sest see puu on tühi, ongi uuele elemendile koht leitud.
- **Keerukus:** O (log n)

13.3 Otsimine

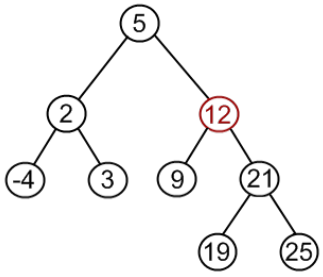
- Alustades puu juurest ning liigutakse vastavalt otsitava võtme väärtusele vasakusse või paremasse alampuusse kuni **võti** leitakse või kuni jõutakse **leheni**.
- Leidmise korral tagastakse **tipu aadress** (tsükkel katkestatakse), kui ei leita, sii **Nil**.
- **Min** leidmisel liigutakse juurest seni vasakule kui saab. Viimases kättesaadavas tipus ongi väikseim võti.
- **Max** leidmiseks liigutakse mööda puud paremale kuni Nil ette tuleb.
- **Keerukus:** O (log n)

13.4 Kustutamine

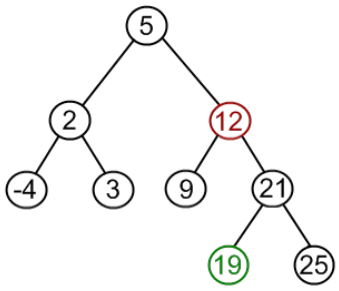
Kustutamine algab otsimisest

1. Lapsed puuduvad, võime tipu lihsalt ära kustutada
2. Kui tipul üks laps, siis see paigutakse kustutava tipu asemele

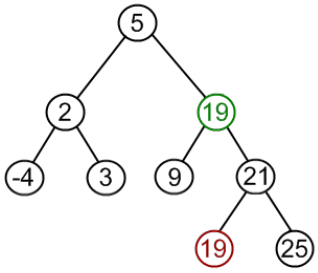
3. Kui mõlemad lapsed on olemas



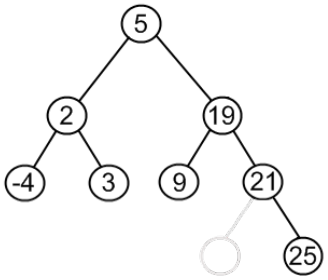
1)



2)



3)



4)

13.5 Keerukusklassid

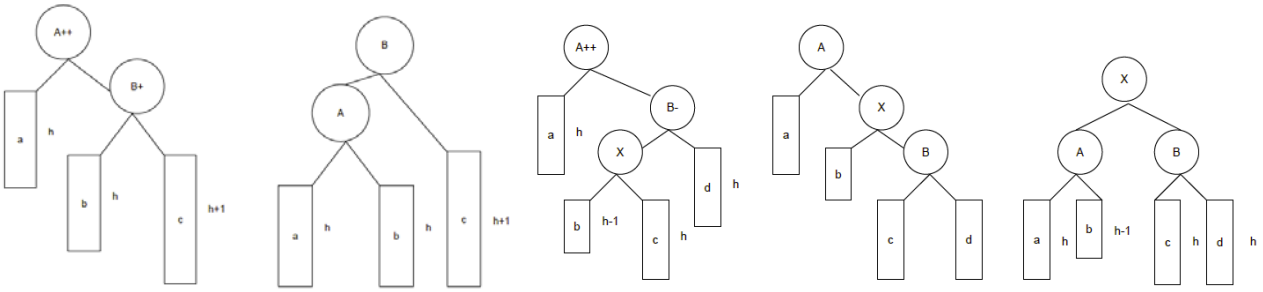
- Üldiselt on ostimiskahendpuust otsimine **logarimilise keerukusega** tegevus (puu tasemed max täidetud).
- **Halvim variant:** kui võtmed on **järjestatud**, siis on otsimise **keerukus lineaarne**, sest saadakse sisuliselt lineaarnimistu.
- **Tasakaalustatud puu** – iga alampuu jaoks vasaku ja parema alampuu kõrguste vahe pole suurem kui üks
- **Tasakaalustatud kahendpuu** tagab log aja nii võtme järgi otsimiseks, elemendi lisamiseks kui ka elemendi eemaldamiseks. Tippe võiks olla vähemalt 256.

13.6 AVL-puu

13.6.1 Omadused ja kuidas töötab

- Otsimiskahendpuu
- Töö tema korrashoimiseks on suur ja algoritm keeruline
- AVL-puu saamiseks on peale iga sõlme lisamist või kustutamist vaja kontrollida tema **tasakaalustatust** ja vajaduse korral puu tasakaalu viia.
- Igal sõlmel on **tasakaalu faktor**, mille abil isel iga tipu vasaku ja parema alampuu kõrguse vahet
    - o - Sõlme vasak alampuu on 1 võrra kõrgem (-1) (kriitiline)
    - o + sõlme parem alampuu on 1 võrra kõrgem (+1) (kriitiline)
    - o * Sõlme mõlema alampuu kõrguse on võrdsed (0)

Tasakaalustamine.Olukord 1 Tasakaalustamine.Olukord 2



13.6.2 Milleks kasutatakse

- Kasutakse siis, kui **otsimine toimub tihedamini**, kui lisamine või kustutamine.
- Saab seal kasutada, kus on kõrge **turvarisk** ning **paralleelse koodi** tegemisel.
- Kasutada kui **mõtlemisestrateegiana**
- Kui teha **uut andmeteeki**, siis sinna implementeerida

13.7 Puna-must puu

13.7.1 Omadused

- Otsimiskahendpuu
- Ei ole nii hästi tasakaalus kui AVL-puu, kuid tema hoidmine on vähem töömahukas ja tulemuslikkus pole oluliselt AVL-puust halvem.
- Iga puu tipp on värvitud kas punaseks või mustaks.
- Tippude lisamisel või kustutamisel tuleb järgida teatud värvide skeemi – nii saab säilitada puu mõistliku seisus
- **n tipuga PM-puu** ei ületa otsimise aeg **2*log(n+1)**
- Võtmega tipu **otsimine**, samuti **vähima** ja **suurima** elemendi leidmine sarnaselt tavalisele otsimiskahendpuule (**keerukus O(log n)**)
- **Must kõrgus** – mustade tippude arv puu juurest leheni. Puu must kõrgus on iga lehe suhtes ühesugune

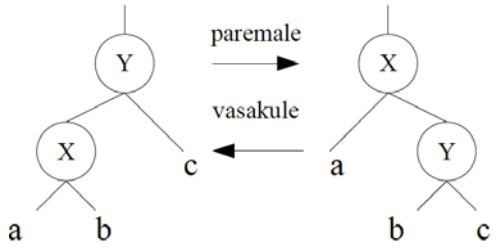
13.7.2 Kuidas töötab

Reeglid:

- Iga sõlm peab olema kas **punane** või **must**
- **Lehtede NIL-viidad** (e tühjad alampuud) on **mustad**
- **Igal punasel sõlmel** peab olema **must vanem**
- **Iga tee**, mis läheb puu juurest mõnda lehte, sisaldab **sama arvu musti sõlmi**

- Puu **korrastamiseks** kasutatakse kolme operatsiooni:

    1. **tippude värvimine** - punane tipp värvitakse mustaks ja must punaseks;
    2. **pööre vasakule** - tipu X parem laps saab uueks (alam)puu juureks ning X ise satub tema vasakuks lapseks
    3. **pööre paremale** - tipu Y vasak laps saab uueks (alam)puu juureks ning Y ise satub tema paremaks lapseks



13.7.3 Milleks kasutatakse

Praktikas üks enam kasutatavatest isebalanseeruvatest otsimispuudest. Konteinerites *"set"* ja *"map"*. C++: STL library, Javas: klass "TreeMap" ja muudes realisatsioonides, kus on vaja kasutada assotsatiivset massiivi.

14. Paisksalvestusmeetod. Paisktabel. Paiskfunktsioon (jäägi meetod ja korrutamise meetod). Kollisisoonide lahendamine (ahelad väljaspool tabelit, avatud paisksalvestus ja erinevad sondeerimismeetodid). Andmete lisamine, otsimine ja kustutamine.

14.1 Paisksalvestusmeetod

Def – algoritm, mis paneb suvalise pikkusega andmehulga vastavusse fikseeritud pikkusega andmehulgaga.

- *Mõistlik kasutada siis, kui struktuur, millega tegeldakse ei pea võimaldama muud kui lisamist, otsimist ja kustutamist, on paisktabel mõitlik lahendus.*
- Otsimise jaoks on kirjes fikseeritud mingi võtmeväärtus, mis peab üle kõigi andmete olema unikaalne - arvete numbrid, isikukoodid jne.
- Kui arvete numbrid oleksid vahemikus 1..100, siis saaks teha tabeli ja paigutada andmed tabeli lahtritesse 1..100 vastavalt arve numbrile.
- Kirjeldatud meetod sobib juhul, kui võtmeid on vähe.
- Meetodit kutsutakse **otsene adresseerimine**. Tabelit, kuhu andmed salvestatakse, nimetatakse **otseadresseerimisega tabeliks.**
- Kirjed võivad olla salvestatud **otse tabelisse** või on tabelis **vastava kirje aadress**.
- Kui aga arveid on küll 100, aga numbrid on vahemikus 1..10000, siis paigutada neid andmeid 10000 lahtriga tabelisse, kus enamik lahtreid tühjaks jääb, on ilmne raiskamine.

- Arve numbritele tuleks rakendada mingit arvutust, et tegelikest väärtustest saaks väärtused vahemikus 1..100. Leitud väärtuse järgi paigutatakse kirje tabelisse.
- Kui võtmete väärtused oleksid 100, 200, 300, ...9900, 10000, siis rakendades tehet Võti div 100 saame väärtused vahemikus 1, 2, 3, ..99, 100 ja nende väärtuste järgi on andmed paigutatavad tabeli lahtritesse
  1..100.
- Keerukus: parimal juhul lisamisel, otsimisel ja kustutamisel O(1). Halvim olukord tekib siis, kui kõigi võtmete jaoks arvutakse sama paiskväärtus. Sel juhul kiiruseks O(N), sõltumata kollisioonide lahendamise meetodist.

14.2 Paisktabel

- Meetodit kutsutakse **paisksalvestamiseks.**
- Igas kirjes eraldatakse üks väli, mis on võtmeks.
- Sellele võtmele rakendatakse **paiskfunktsiooni**, mis vastavalt võtme väärtusele arvutab indeksi e tabeli lahtri aadressi.
- **Paiskfunktsioon** tuleb valida selliselt, et arvutuse tulemus **mahuks tabeli indeksite vahemikku**.
- Paisksalvestamiseks paigutatakse andmed paisktabelisse, mida saab realiseerida massiivina.

14.3 Paiskfunktsioon

- On algoritm, mis arvutab suvalisele väärtusele vasteks täisarvu nii, et see **mahub etteantud vahemikku.**
- Vahemikuks on **paisktabeli pikkus** ehk leitud täisarv peab sobima tabeli indeksiks.
- Leitud indeksit nimetatakse **paiskväärtuseks**.
- **Kollisioon** ehk põrge on olukord, kus paiskfunktsiooni rakendamisel kahele erinevale võtmele tekib sama paiskväärtus.
- **Peab olema kiirelt ja kergelt arvutatav - LIHTNE**
- **Suutma salvestada kirjed võimalikult ühtlaselt tabelisse ära jagada – ÜHETAOLINE**
- **Sama sisend peab alati andma sama väljundi - DETERMINEERITUD**

14.3.1 Jäägi meetod

- Paiskväärtuseks on **jääk**, mis tekib võtme täisarvulisel jagamisel tabeli pikkusega.
- h(k)=k mod M, kus k on võti ja M on paisktabeli pikkus.
- M-i valik ei ole suvaline.
  - Sobivad pigem algarvud.
  - Ei sobi arvusüsteemi alus, paarisarvud jms, mille puhul samasuguste jääkide tekkimise võimalus on suurem.

14.3.2 Korrutamise meetod

- Võti korrutatakse mingi irratsionaalarvuga (0<A<1) ja täisosa lõigatakse ära.
- Järgi jääb arv vahemikus 0 kuni 1.
- Leitud arv korrutatakse tabeli pikkusega M, tulemusest jäetakse alles täisosa.
- See täisosa mahub alati 0 ja M-1 vahele.
- `h(k)=[M(k*T - [k*T])]`
- $T=\frac{\sqrt{5}-1}{2}=0{,}618033$

T 0,618033988749895		*Tabeli kõrgus M=12*
võti	rakendamine	tulemus
1	int[12*(1*T-int[1*T])]	7
2	int[12*(2*T-int[2*T])]	2
3	int[12*(3*T-int[3*T])]	10
4	int[12*(4*T-int[4*T])]	5
5	int[12*(5*T-int[5*T])]	1
6	int[12*(6*T-int[6*T])]	8
7	int[12*(7*T-int[7*T])]	3
8	int[12*(8*T-int[8*T])]	11
9	int[12*(9*T-int[9*T])]	6
10	int[12*(10*T-int[10*T])]	2
11	int[12*(11*T-int[11*T])]	9
12	int[12*(12*T-int[12*T])]	4

14.4 Kollisisoonide/vastuolude lahendamine

Kui mingi võtmega kirjet on vaja otsida, siis tehakse võtmega arvutus, saadakse indeks ja vaadatakse tabelisse.

Võib tekkida olukord, kus kaks erinevat võtmeväärtust arvutatakse samaks indeksiks. Näiteks on võtmed 500 ja 588. Kui mõlemale võtmele rakendada täisarvulist jagamist 100ga, on tulemuseks 5, st mõlemad kirjed tuleks paigutada lahtrisse indeksiga 5, mis pole võimalik. Sellist olukorda kutsutakse **vastuoluks** ehk **kollisiooniks** ehk **põrkeks**.

Esiteks tuleb vähendada kollisiooniohtu:

1. Suurendada tabel 2…3 korda, kui tegelikult on vaja
2. Leida selline paiskfunktsioon, mis indeksid/paiskväärtused võimalikult ühtlaselt üle kogu tabeli jaotab

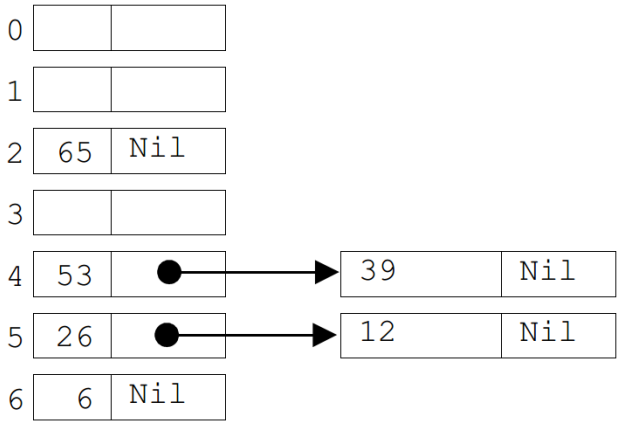
Ükskõik kui kavalalt paiskfunktsiooni ka ei valiks, ei maksa loota, et vastuolusid ei teki. Seega on põhimõtteliselt kaks lahendust, mida nendega peale hakata:

1. Kollisiooniahel

2. Otsida mingi kavala valemi järgi tabelist uus koht, mis ei ole hõivatud, kuid tuleb arvestada sellega, et kõik kirjed mahuksid tabeli piiridesse.

14.4.1 Kollisiooniahelate paigutamine väljaspoole tabelit

1. **Vastuoluliste kirjete** eraldi seostamine



2. **Ostsene seostamine**. Selle meetodi puhul on tabelis ainult viidaväljad, kuhu kirjutatakse aadressid ahelate algusele ja kõik andmeelemendid paiknevad tabelist väljas ahelates. On arusaadav, et **erinevate olukordade arv väheneb** ja koos sellega ka **vajalike kontrollide arv**. Seega kulub vähem aega.

14.4.2 Vaba/avatud paisksalvestus

- **Kõik võtmed** tuleb tabelisse **ära mahutada.**
- Eeldus uue võtme paigutamisel: **vähemalt 1 tabeli lahter on vaba**.
- Kui on tabeli aadres h(K) on juba hõivatud, siis tuleb leida talle uus **vaba aadress**.
- **Järelkatsumise järjekord** - selleks kirjeldatakse iga võtme jaoks mingi **lahtriaadresside järgnevus**, kuhu võtmeid paigutada proovitakse, seni kuni vaba koht leitakse.

14.4.3 Erinevad sondeerimismeetodid

14.4.3.1 Lineaarne järelkatsumine

```
h(k), h(k)-1, h(k)-2, ...0, m-1, ..., h(k)+1

Järelekatsumisfunktsioon: s(j,k) = j
```

Puudus: lineaarne klasterdumine – tekivad pikad hõivatud piirkonnad, millede pikenemise tõenäosus üha kasvab. Esmane kuhjumine. Efektiivsus kahaneb, kui tabeli täidetus läheneb 100%-l, sest väga pikalt on vaja otsida.

Indeks / Lisatav võti	0	1	2	3	4	5	6
26						26	
53					53	26	
12				12	53	26	
65			65	12	53	26	
39		39	65	12	53	26	
6		39	65	12	53	26	6

14.4.3.2 Ruutjärelkatsumine

Meetod seisneb selles, et uut kohta elementile võtmega K otsitakse üha kaugemalt ja see kaugus kasvab kord ühele poole kord teisele poole.

j	s(j,k)
1	0
2	1
3	-1
4	4
5	-4
6	9
7	-9
8	16
9	-16
10	25
11	-25
12	36

h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, ...
Järelekatsumisfunktsioon: s(j,k)=([j/2]) 2(-1) j

		mod 7	0	1	2	3	4	5	6
1. samm	*26*	*5*						26	
2. samm	*53*	*4*					53	26	
3. samm	*12*	*5*					53	26	12
4. samm	*65*	*2*			65		53	26	12
5. samm	*39*	*4*			65	39	53	26	12
6. samm	*6*	*6*	6		65	39	53	26	12

- Pluss: Esmase kuhjumise vältimine
- Miinus: Teisane kujumine, mis siiski pole nii tugev

14.4.3.3 Topelt paisksalvestus

Idee: fikseerime veel teise paiskfunktsiooni h'(k), mida samale võtmele rakendades saame uue aadressi.

Näiteks: h'(k) = 1+k mod (m-2)

h(k), (h(k)-h'(k)) mod m, (h(k)-2·h'(k)) mod m, (h(k)-3·h'(k)) mod m, ...
Järelekatsumisfunktsioon: s(j,k) = j ·h'(k)

Indeks / lisatav võti	0	1	2	3	4	5	6	
26						26		26 mod 7=5
53					53	26		53 mod 7=4
12			12		53	26		(5-(1+12 mod 5)) mod 7=2
65		65	12		53	26		(2-(1+65 mod 5)) mod 7=1
39		65	12		53	26	39	(4-(1+39 mod 5)) mod 7=6
6	6	65	12		53	26	39	(6-(1+6 mod 5)) mod 7=4 (6-2*(1+6 mod 5)) mod 7=2 (6-3*(1+6 mod 5)) mod 7=0

14.5 Andmete lisamine, otsimine ja kustutamine

14.5.1 Võtme K järgi otsimine

- Arvutada välja paiskaadress h(K).
- Kontrollida, kas võti K on tabelis kohal t[h(k)].
- Kui lahter oli tühi, siis otsimine oli ebaedukas.
- Kui lahtris on K-st erinev võti, siis läbi vastav kollisiooniahel.
- Kui K-d ei leitud ka ahelast, siis otsimine oli ebaedukas.
- Kui võti K oli lahtris t[h(k)] või ahelas, siis otsimine oli edukas.

14.5.2 Võtme K lisamine (eeldusel, et kirjet võtmega K tabelis ei ole)

- Arvutada välja paiskaadress h(K).
- Kui tabelis vastava indeksiga lahter on tühi, siis lisa võti sinna.
- Vastasel juhul liigu kollisiooniahela lõppu, tee uuse element ja lisa võti.

14.5.3 Võtmega K kirje eemaldamine

- Otsi võtme K järgi.
- Kui otsimine oli edukas, siis:
    - Kui K on tabelis, tõmba ta sealt maha, kirjuta tema asemele esimene element kol.ahelast (kui ahel olemas) ja viimane eemalda ahelast.
    - Kui K on ahelas, siis kustuta sealt vastav element.

14.6 Paiskemeetod vs otsimispuud

- Mõlemad meetodid on mõeldud otsimiseks.
- Reegline on paiskmeetod parem/kiirem, eeldusel, et võtmed on lihtsamat andmetüüpi ja nendele on hea arvutada paskväärtust.
- Kui võtmete arv ei ole ennustatav on puu parem oma dünamilisuse tõttu.
- Puu on parem, kui eeldada sorteeritust: sort, jada, min, max, naabervõtmed. Nimetatud andmeid paisktabelis kiirelt leida võimalik ei ole. Tabeli saab küll väljastada, kuid see ei anna midagi.

	Paiskmeetod	Otsimispuud
Lisamine	O(1)	O(log N)
Kustutamine	O(1)	O(log N)
Otsimine	O(1)	O(log N)
Min	O(N)	O(log N)
Max	O(N)	O(log N)
Sorterd	O(N log N)	O(N)