Algoritmid ja andmestruktuurid

- Järjekorra tüüpi andmestruktuurid
- Andmestruktuud *dünaamiline massiiv*
- Lihtne *sheduling*
- Andmestruktuur *kuhi* (*heap*)

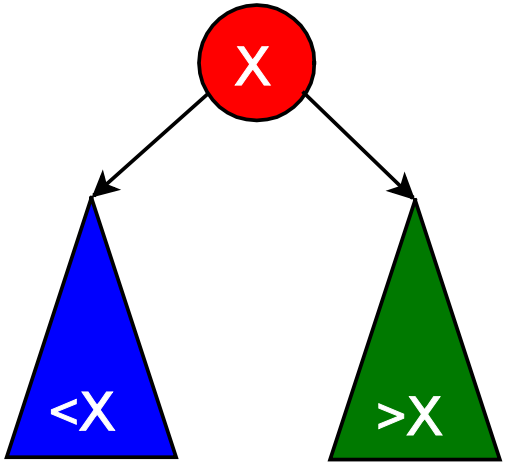
Kokkuvõtteks otsingust

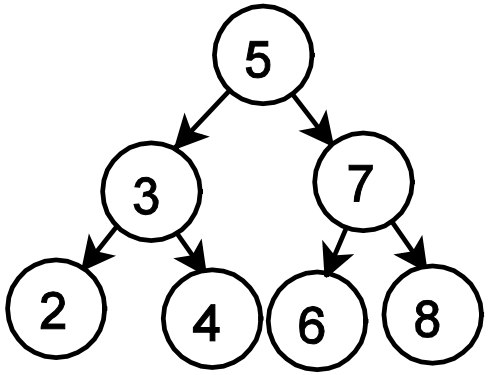
- Otsimine sorteeritud andmetest O(log *n*)
  - Binaarne otsing staatilistel andmetel
  - Otsing otsingupuus dünaamilistel andmetel
- Otsimine sorteerimata andmetest O(*n*)

Binaarne otsingupuu

Binaarne puu (*Binary Search Tree* – BST)

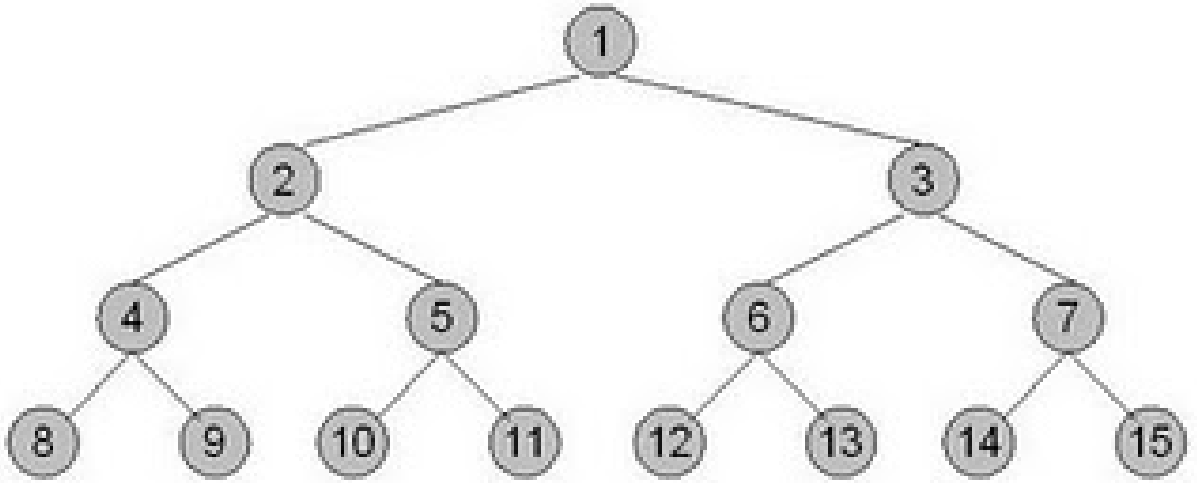
- max 2 järglast
- **Vasak** alampuu < **Tipp** < **Parem** alampuu





Binaarse puu (kahendpuu) sügavus

- Täielikus binaarses puus sügavusega h on 2^h-1 tippu
- Kui n on binaarse puu tippude arv ja h on selle puu sügavus, siis $h \geq \lfloor \lg n \rfloor$



Binaarsed otsingupuud - kokkuvõte

- Operatsioonid (sõltuvalt puu sügavusest h):
  - SEARCH O(h)
  - PREDECESSOR O(h)
  - SUCCESOR O(h)
  - MINIMUM O(h)
  - MAXIMUM O(h)
  - INSERT O(h)
  - DELETE O(h)
- Kiired kui puu sügavus on väike – puu on lame O(lg n)
- Aeglane, kui puu on välja venitatud – lingitud list O(n)

Balanseeruvad otsingupuud

Lisamise ja kustutamise operatsioonid modifitseerivad puud nii, et tasakaal erinevate harude vahel säiliks – harud on ühe sügavad või ei erine palju

- Red-Black tree
- AVL tree
- B-tree
- Splay tree
- Treap

1. Programmeerimistöö
Peotantsu partnerite leidmine

Implementeerib ootejärjekorra ja sobiva partneri leidmise ootejärjekorrast

- Tantsija lisandumisel juhtub üks kahest
  - leitakse talle ootejärjekorrast sobiv partner
  - pannakse tantsija ootejärjekorda, kui ei leita
- Sobivaks partneriks mehele on ootejärjekorras kõige pikem naine, kes on mehest lühem, naisele vastupidi
- Kui sobiv partner leitakse, siis eemaldatakse partner ootejärjekorrast ja väljastatakse
- Iga tantsijat iseloomustab paar (sugu, pikkus)
- Süsteemilt saab küsida ootejärjekorra sisu

1. Programmeerimistöö
Peotantsu partnerite leidmine

- Eesmärgiks on implementatsioon, kus otsing peab toimuma keerukusega O(lg(*n*)), kus *n* on järjekorra pikkus
  - Täispunktid kui halvima juhu keerukus on O(lg(*n*))
    - Balanseeriv otsingupuu
  - Osalised punktid, kui keskmise juhu keeruks on O(lg(*n*))
    - Binaarne otsingupuu
- Andmestruktuurid tuleb ise elementaarsetest andmetüüpidest implementeerida

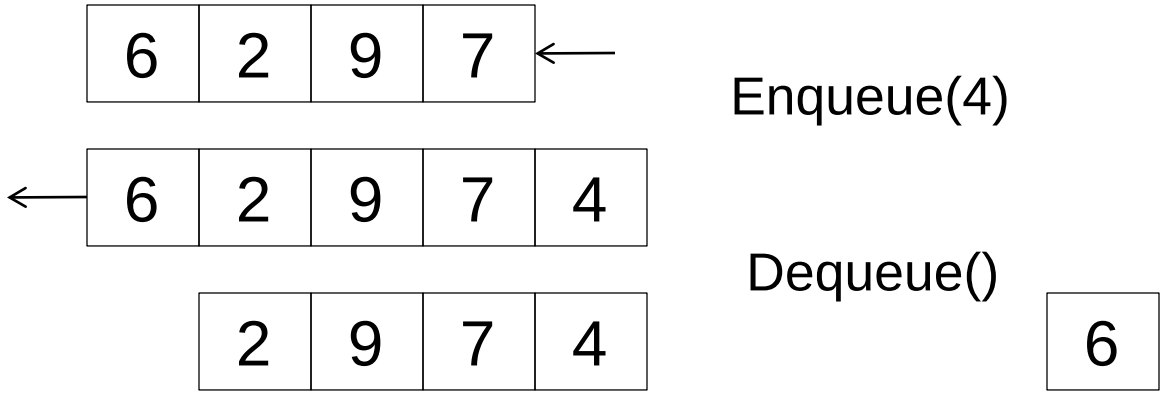
Järjekorra tüüpi andmestruktuurid

- Võimaldavad
  - Andmeid lisada
  - Andmeid eemaldada
  - Kontrollida andmete olemasolu, aga mitte nende hulka
  - Lisada andmeid piiranguta (kuni on mälu)
  - Puudub andmetele otsepöörduse võimalus – andmete eemaldamise järjekord sõltub andmestruktuurist
- Andmestruktuurid
  - *Stack* – pinu, magasin
  - *Queue* – järjekord
  - *Priority queue* – prioriteetjärjekord
  - *Deque* – kahe otsaga järjekord – *stack* ja *queue* ühend

Andmestruktuur *järjekord* (*queue*)

FIFO – *first in first out*

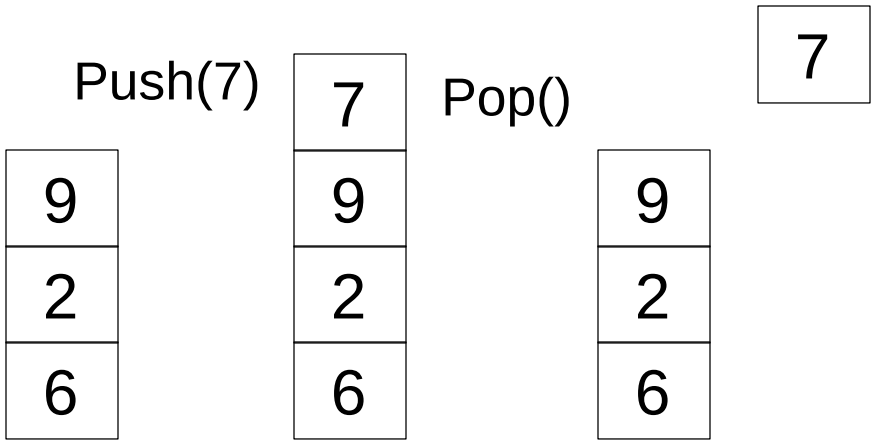
```
DataType Queue
{
  Enqueue(x): insert element x to the
      queue
  Dequeue(): delete the front element
      of the queue and return it
  IsEmpty(): return true if the queue is empty.
      false otherwise
}
```



Andmestruktuur *pinu* (*stack*)

LIFO – *last in first out*

```
DataType Stack
{
  Push(x):     lisa element X pinusse (enqueue)
  Pop(x):      eemalda viimati lisatud element ja
               tagasta selle väärtus (dequeue)
  IsEmpty():   tõene kui pinu on tühi
}
```

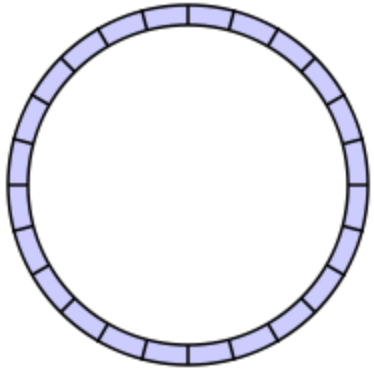


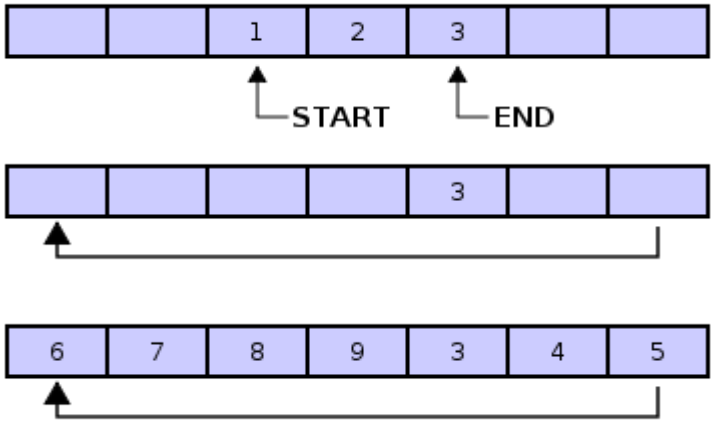
Pinu implementatsiooni idee

- Võimalik implementeerida nii massiivi kui lingitud listi põhisena
- Massiivipõhine implementatsioon
  - Vaja on massiivi M ja muutujat E, mis kannab elementide arvu
  - Pinu põhi (esimene element) on massiivis kohal M[0]
  - M[E] näitab järgmise elemendi asukohta *push* operatsioonis
  - Elementide arv on piiratud massiivi suurusega

Järjekorra implementatsiooni idee

- Vaja on massiivi M ja kahte muutujat *START* ja *END*
- *Enqueue* operatsioon lisab asukohale M[END+1] ja *dequeue* eemaldab asukohast M[START]
- Massiivi kasutatakse ringpuhvrina - kui massiiv saab täis, siis alustatakse algusest
- Elementide arv on piiratud massiivi suurusega.





Dünaamiline massiiv

- Tavalise massiivi probleemiks on mahu piirang

  massiivi suurus tuleb määrata enne kasutamist ja sinna ei saa lisada rohkem andmeid, kui on reserveeritud mälu

- Dünaamiline massiiv muudab ise mälukasutust sõltuvalt vajadusest
  - elementide lisamine ja eemaldamine toimub massiiv lõpus
  - massiiv **suureneb** kui lisatakse täis massiivi
  - massiiv **väheneb**, kui eemaldatakse liiga tühjast massiivist
- Säilib kiire otsepöördumine
- Kasutusvaldkonnad

  pinu, binaarne kuhi, hash tabel

Dünaamiline massiiv - idee

- Mitmetes keeltes juba baasandmetüübina olemas
  - Java, .Net ArrayList
- Abimuutujad
  - *num* elementide arv massiivis
  - *size* reserveeritud massiivi suurus
- Massiivi suurendamine ja vähendamine toob kaasa kõigi massiivi elementide kopeerimise
  - luuakse (reserveeritakse) uus suurem/väiksem massiiv
  - kopeeritakse kõik andmed
  - vabastatakse eelnev massiiv (Javas automaatselt)

Dünaamiline massiiv - keerukus

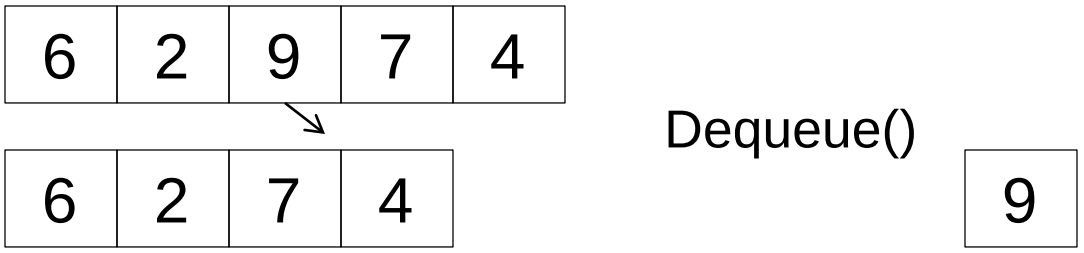
- Eesmärgiks on lisamise ja eemaldamise operatsioonide keerukus O(1) keskmisena üle kõigi operatsioonide
  - konstantse suuruse võrra suurendamine ei taga seda
  - täitumisel (*num* > *size*) suurendatakse 2 korda
  - alatäitumisel (*num* <= *size/4*) vähendatakse 2 korda
- Täpset keerukuse analüüsi saab teha *amortiseeritud analüüsi* (*amortized* analysis) meetodil
  - iga elemendi lisamisel ja eemaldamisel paneme kaks lisaomistusoperatsiooni “panka”
  - täitumisel (*size/2* operatsiooni pärast eelmist suurendamist) kopeerime massiivi *size* elementi “säästude” arvelt
  - vähendamisel kohe peale eelmist suurendamist *size/4* elemendi võrra saame järgijäänud *size/4* elementi kopeerida säästude arvelt

Andmestruktuur *prioriteetjärjekord* (*priority queue*)

Prioriteetjärjekord on lineaarne järjekord, kus saab sisestatud andmeid kätte prioriteedi järjekorras (prioriteetsemad enne)

Andmete "sorteerimist" võidakse teostada sisestamisel, väljastamisel või muud moodi, sõltuvalt realisatsioonist

```
AbstractDataType PriorityQueue
{
  IsEmpty(): return true if the priority queue is empty.
       false otherwise
  Enqueue(x): insert element x to the priority queue
  Dequeue(): delete and return the "best" element of the
       priority queue
  Increase(x): increase the priority of x
  Union(PQ): make a union of two priority queues
}
```



Scheduling ülesanded

- Minimaalse ooteaja ülesanne
  - järjekorras on hulk erineva kestusega töid (kliente)
  - minimeerida summaarset ooteaega
  - rakendused: juuksur, programmeerimisolümpiaadi trahviminutid, kettaoperatsioonide järjestamine
- Lõpptähtajaga ülesannete preemia maksimiseerimine/trahvi minimiseerimine
  - igal ülesandel on kestvus, lõpptähtaeg ning tähtajalise täitmise preemia ja/või tähtaja ületamise trahv
  - järjestada maksimaalselt kasulikul moel
- On veel mitmeid muid *scheduling* ülesandeid
- Ülesanded võivad olla staatilised või dünaamilised

  kõik tööd on alguses teada või ei

Staatiline minimaalse ooteaja ülesanne

töö	kestvus
t1	5
t2	10
t3	4

```
sort (jobs)
while (jobs)
  schedule next job
```

järjestus	koguaeg
1,2,3	39
1,3,2	33
2,1,3	44
2,3,1	43
3,1,2	32
3,2,1	37

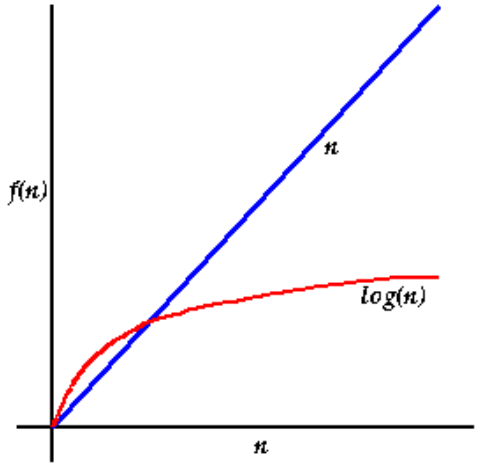
Keerukus:
$O(n \lg n) + O(n) = O(n \lg n)$

Dünaamiline minimaalse ooteaja ülesanne

```
while (job waiting)
      enqueue(job, priority-queue)
while (priority-queue) {
      schedule dequeue(priority-queue)
      while (job waiting)
         enqueue(job, priority-queue)
}
```

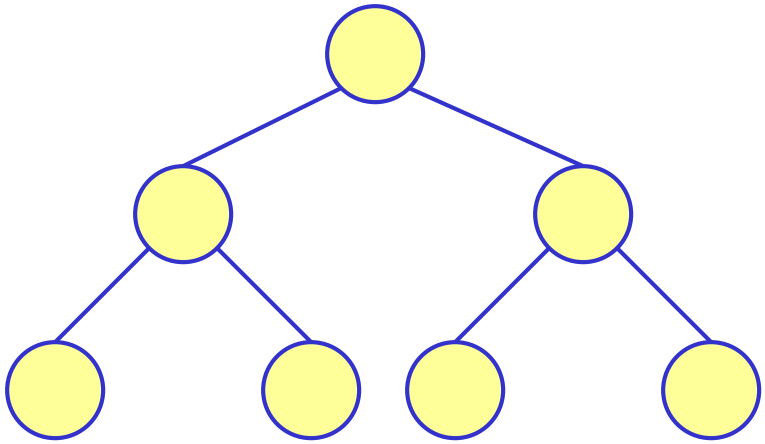
Prioriteetjärjekorra operatsioonide keerukus

	Enqueue()	Dequeue()	isEmpty()
sorteerimata list	O(1)	O(*n*)	O(1)
sorteeritud list	O(*n*)	O(1)	O(1)
kuhi	O(lg *n*)	O(lg *n*)	O(1)



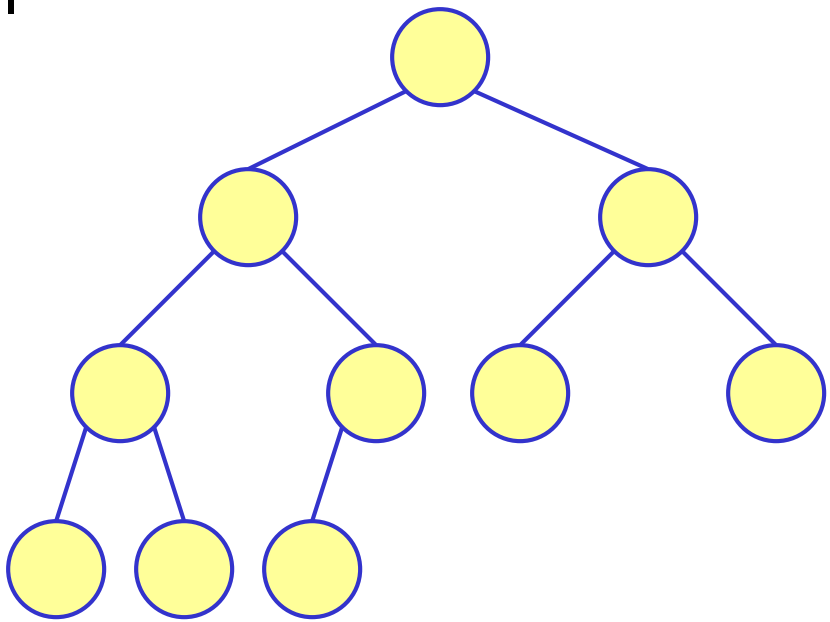
Täielik kahendpuu (*binaarpuu*)

- Kõigil sisemistel sõlmedel on 2 järglast
- Kõik lehed asuvad sügavusel *d*
- Kui puu sügavus (tasemete arv) on *n* siis on puus:
- $2^{n-1} - 1$ sisemist sõlme
  - 2^{n-1} lehte
  - $2^{n} - 1$ tippu kokku
- Puu sügavus, millel on *m* tippu on $\lg(m+1)$



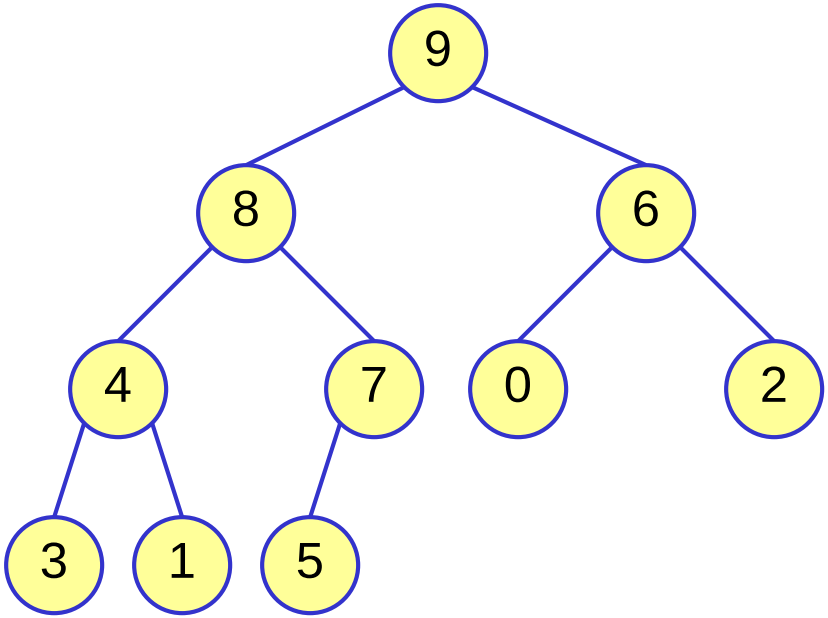
Peaaegu täielik kahendpuu

- täielik kahendpuu tasemeni *d* - 1
- *d* taseme lehed on kõik võimalikult "vasakul"



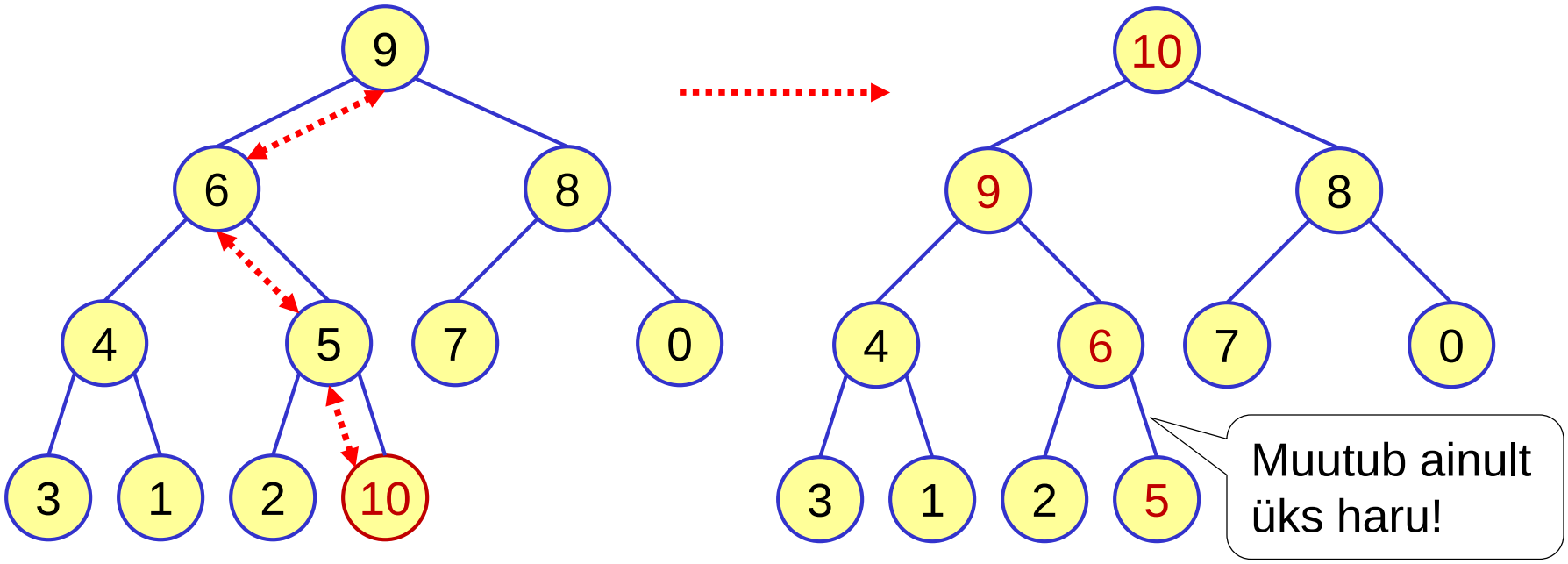
Binaarkuhi (*Binary Heap*)

- peaaegu täielik kahendpuu
- sõlmedes on mingi järjestatud hulga elemendid
- sõlmedes oleva võtme väärtus on suurem (väiksem) või võrdne sõlme järglaste võtme väärtusest
- *max-heap* - suurim tipus
- *min-heap* - väiksem tipus



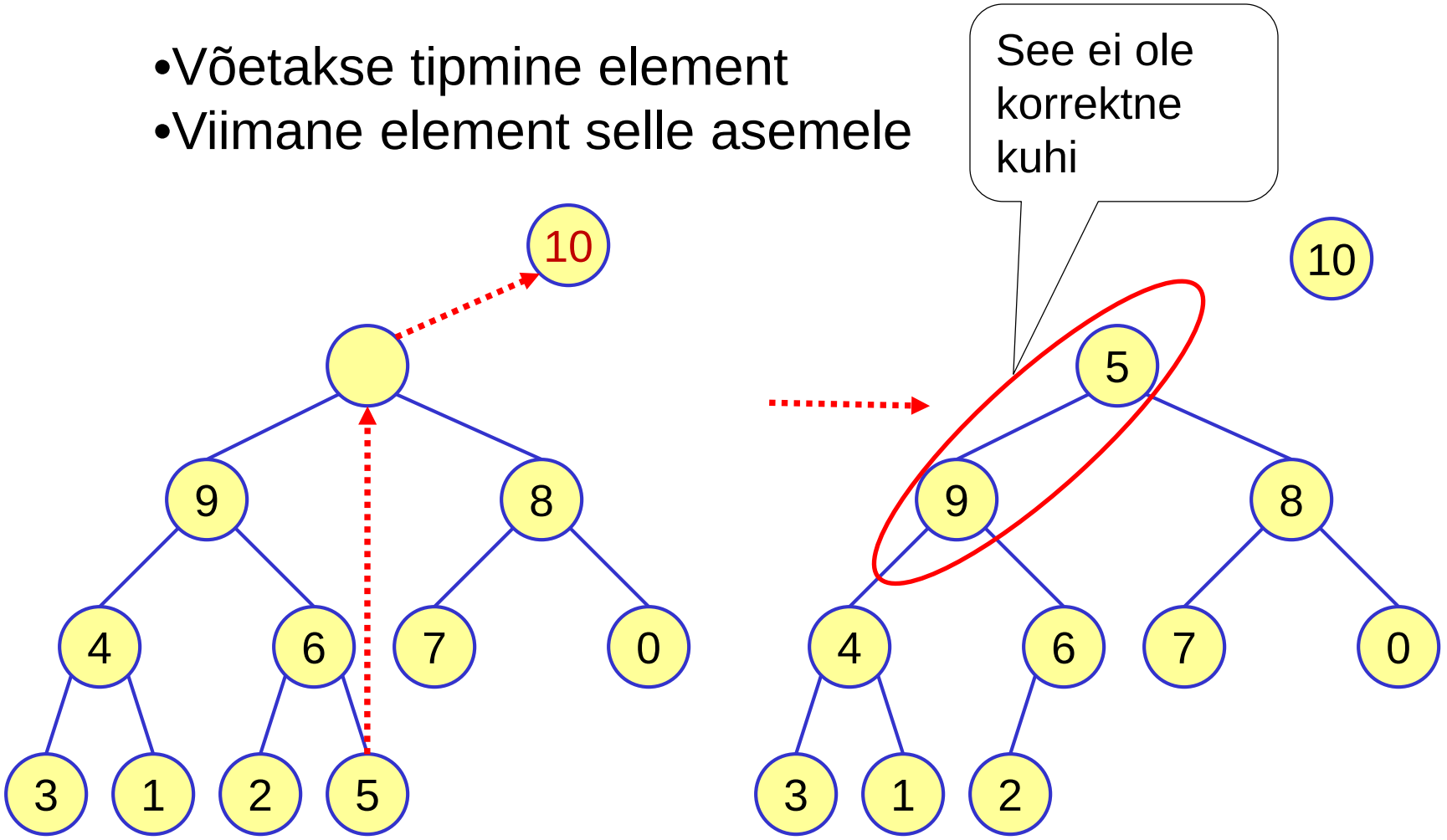
Elemendi X lisamine kuhja

- lisatakse esimesele vabale kohale alumises reas
- vajadusel vahetatakse vanematega, kuni kuhja tingimus on täidetud



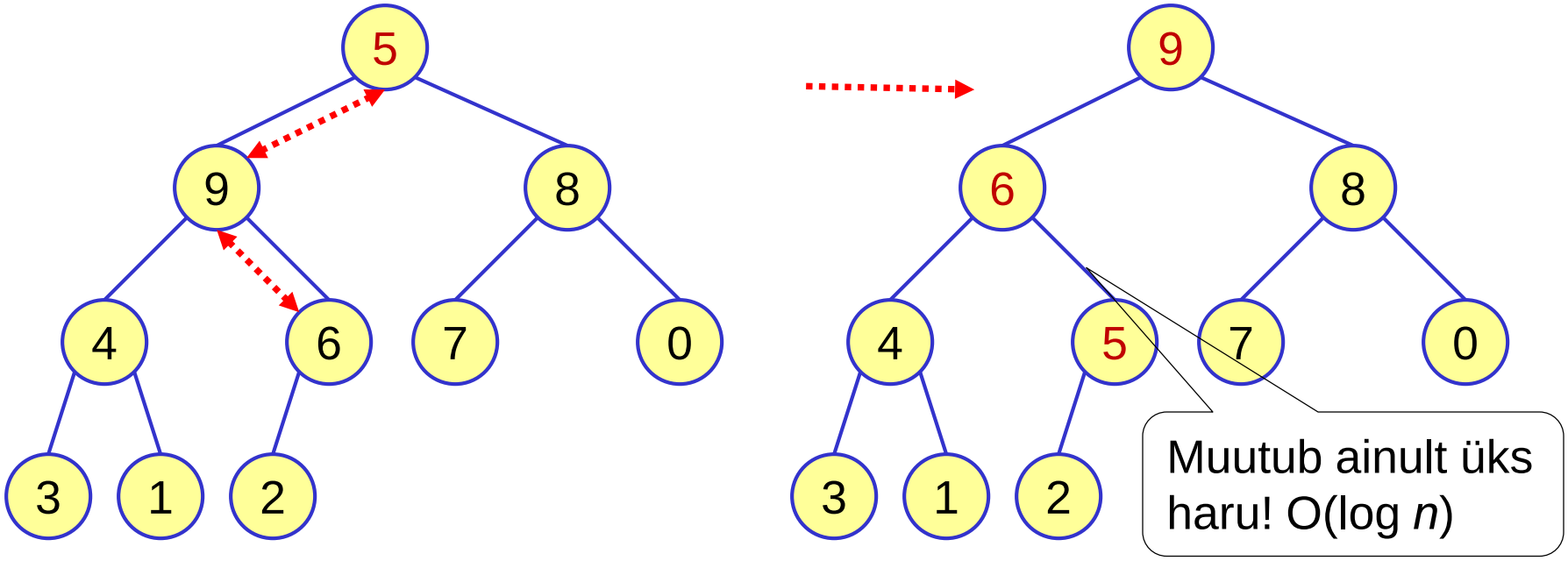
Suurima elemendi väljavõtmine I

- Võetakse tipmine element
- Viimane element selle asemele



Suurima elemendi väljavõtmine II

•Vahetatakse suurema (väiksema) järglasega, kuni kuhja tingimus on rahuldatud



Prioriteetjärjekorra implementeerimine

		Heaps			
Operation	**List**	**Binary**	**Binomial**	**Fibonacci ***	**Relaxed**
make-heap	**1**	**1**	**1**	**1**	**1**
enqueue	**1**	**log N**	**log N**	**1**	**1**
find-max	**N**	**1**	**log N**	**1**	**1**
dequeue	**N**	**log N**	**log N**	**log N**	**log N**
union	**1**	**N**	**log N**	**1**	**1**
increase-key	**1**	**log N**	**log N**	**1**	**1**
delete	**N**	**log N**	**log N**	**log N**	**log N**
is-empty	**1**	**1**	**1**	**1**	**1**

Prioriteetjärjekord binaarkuhjana

```
void enqueue(node new, heap H) {
        put new at the end of the heap //left on last level
        node parent = parent_of(new);
        while(new is not root && parent < new) {
                exchange(parent, new);  // exchange values
                new = parent;
                parent = parent_of(new);
        }
}
```

http://nova.umuc.edu/~jarc/idsv/lesson2.html

Prioriteetjärjekord binaarkuhjana

```
node dequeue(heap H) {
        node_out = root_of(H);
        move the bottom node to the root;
        parent = root_of(H); //former bottom node;
        larger_child = larger_child_of(parent);
        while(larger_child exists && parent < larger_child){
                exchange(parent, larger_child); // exchange values
                parent = larger_child;
                larger_child = larger_child_of(parent);
        }
}
```

http://nova.umuc.edu/~jarc/idsv/lesson2.html

Kuhja massiivesitus

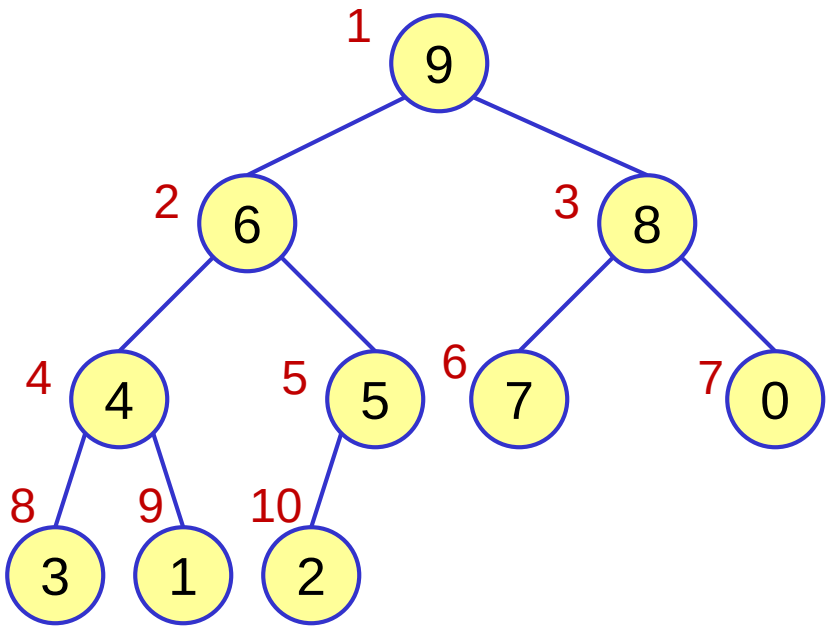
Eeldusel, et massiivi esimene element (kuhja tipp) on indeksiga 1:

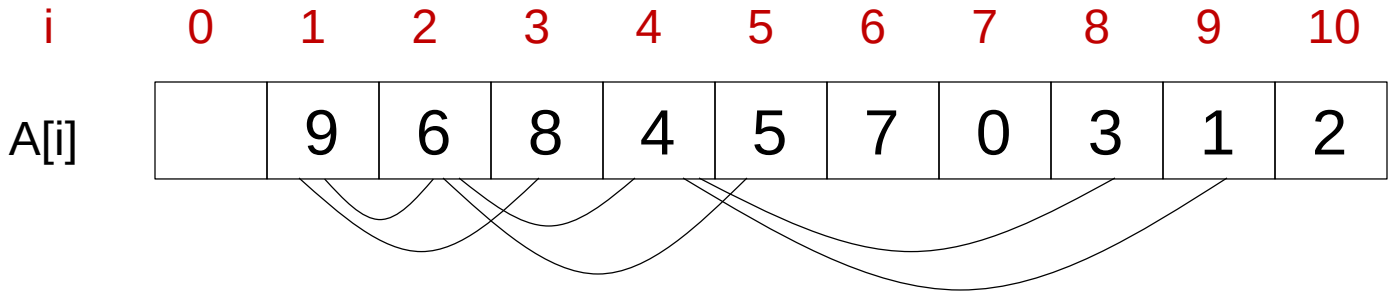
parent(i) = i div 2

left(i) = 2i

right(i) = 2i+1

$A[parent(i)] \geq A[i]$





Kuhi (*heap*)

- Võimaldab leida dünaamilisel andmehulgal maksimaalset (minimaalset) elementi
  - Lisamine O(log *n*)
  - Maksimaalse elemendi eemaldamine O(log *n*)
- Esitab osaliselt sorteeritud andmeid
- Põhiline prioriteetjärjekorra implementatsioon
- Lisaks binaarkuhjale on olemas ka teisi kuhjasid

  Binomiaalne, Fibonaci jt

Kuhja rakendus - Heapsort

- Sisendiks saame peaaegu täieliku kahendpuu
  Andmed on kahendpuus, aga suvalises järjekorras
- Esimese sammuna kuhjastame kahendpuu
- Teise sammuna võtame sealt järjest maksimaalseid elemente ja korrastame kuhja peale iga elemendi äravõtmist

Heapsort massiivesitusega

```
struct heap
    keytype S [ 1 .. n ];  // S is indexed form 1 to n.
    int heapsize;          // heapsize only takes
};                         // the values 0 through n.
```

```
void heapsort ( int n, heap H, // H ends up a heap.
                keytype S[]) {
    makeheap (n, H);
    removekeys (n, H, S);
}
```

Kuhjastamine - *makeheap*

```
void makeheap (int n, heap H)  // H ends-up a heap. {
    index i;
    heap Hsub;                          // Hsub ends up a heap.
    for (i = d - 1; i >= 0; i--)        // Tree has depth d.
        for (all subtrees Hsub whose roots have depth i)
            siftdown (Hsub);
}
```

```
void makeheap (int n, heap H)           // H ends up a heap. {
    index i;                            // It is assumed that n keys
                                        // are in the array H.S.
    H.heapsize = n;
    for (i = [n/2]; i >= 1; i--)        // Last node with depth
        siftdown (H, i);                // d -- 1, which has children
}                                       // is in slot [n/2j in the array.
```

Elemendile koha leidmine - *siftdown*

```
void siftdown (heap H)                  // H starts out having the
{                                       // heap property for all
    node parent, largerchild;           // nodes except the root.
                                        // H ends up a heap.
    parent = root of H;
    largerchild = parent's child containing larger key;
    while (key at parent is smaller than key at largerchild){
        exchange key at parent and key at largerchild;
        parent = largerchild;
        largerchild = parent's child containing larger key;
    }
}
```

Heapsort *siftdown* massiivesitusega

```
void siftdown (heap H, index i)        // To minimize the number
{                                      // of assignment of records,
    index parent, largerchild;         // the key initially at the root
    keytype siftkey;                   // (siftkey) is not assigned to a
    bool spotfound;                    // node until its final position
    siftkey = H.S[i];                  // has been determined.
    parent = i;
    spotfound = false;
    while (2* parent <= H. heapsize && ! spotfound){
        if (2* parent < H. heapsize && H. S[2* parent] < H.S.[2* parent + 1])
            largerchild = 2* parent + 1;       // Index of right child is 1
        else                                   // more than twice child is 1
            largerchild = 2* parent;           // parent. Index of left child
        if (siftkey < H.S[largerchild]){       // is twice that of parent.
            H.S. [parent] = H.S [largerchild];
            parent = largerchild; }
        else spotfound = true; }
    H.S [parent] = siftkey; }
```

Elementide võtmine kuhjast

```
void removekeys (int n, heap H, keytype S[]) {
    index i;
    for (i = n; i >= 1; i--)
        S[i] = root (H);
}
```

```
keytype root (heap H) {
    keytype keyout;
    keyout = key at the root;
    move the key at th bottom node to the root;     // Bottom node is,
    delete the bottom node;                         // far-right leaf.
    siftdown (H);                                   // Restore the
    return keyout;                                  // heap property.
}
```

```
keytype root (heap H) {
    keytype keyout;
    keyout = H.S[1];                    // Get key at the root.
    H.S[1] = H.S[heapsize];             // Move bottom key to root.
    H. heapsize = H. heapsize − 1;      // Delete bottom node.
    Siftdown (H, 1);                    // Restore heap property.
    return keyout;
}
```

Heapsort keerukus

O(siftdown(*n*)) = lg(*n*)

O(makeheap(*n*)) = *n*/2 * O(siftdown(*n*)) =O(*n* log *n*)

! Tegelikult on *makeheap* keerukus O(*n*), kuna *siftdown* kutsutakse välja alampuudele

O(removekeys(*n*)) = *n* * O(siftdown(*n*)) = O(*n* log *n*)

O(heapsort(*n*)) =
max O((makeheap(*n*)), O(removekeys(*n*))) =
O(*n* log *n*)

Algoritmi abstraktsioonitasemed

Üldised algoritmide loomise paradigmad ja abstraktsed andmestruktuurid aitavad organiseerida mõttetööd

Näiteks dünaamilise tööde järjestamise ülesande lahendamiseks:

- vajame prioriteetjärjekorra andmestruktuuri
- prioriteetjärjekorra teeme kuhja andmestruktuurina
- kuhja teeme binaarkuhjana massiivesitusega dünaamilisel massiivil
- dünaamilise massiivi andmestruktuuri implementeerime massiivi ja kahe abimuutujaga

$\Rightarrow$ dekompileeritud koodi või masinkoodi oleks üsna raske mõista