Näidetes kasutatavad kirjed (records)

```
typedef struct date // KUUPÄEV
{
  short int day; // päev
  char month[4]; // kuu (näit. "Jan")
  short int year; // aasta
}
DATE;

struct person // ISIK
{
  char *pName, // viit nimele
  *pAddress; // viit elukohale
  long int Code; // isikukood
  DATE Birthdate; // sünniaeg
  struct person *pNext;// viit järgmisele isikule
};
typedef struct person PERSON;
```

pAddress
Birthdate
pName
Code
pNext

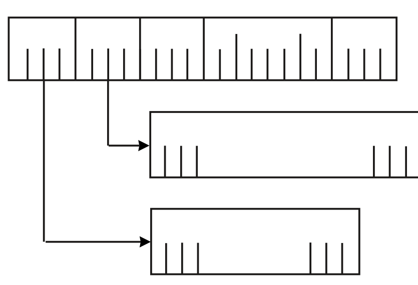
Üksiku kirje loomine ja eemaldamine (1)

```
PERSON *p;
p = CreatePerson("Jaan Tamm",
                 "Tallinn, Tartu mnt 30-10",
                 12345678, 10,"Jan", 1995);
```

```
PERSON *CreatePerson(char* name, char* address,long int code,
short int day,char* month, short int year)
{
   PERSON *pResult = (PERSON *)malloc(sizeof(PERSON));
   pResult->pName = (char*)malloc(strlen(name) + 1);
   strcpy(pResult->pName, name);
   pResult->pAddress = (char*)malloc(strlen(address) + 1);
   strcpy(pResult->pAddress, address);
   pResult->Code = code;
   pResult->Birthdate.day = day;
   strcpy(pResult->Birthdate.month, month);
   pResult->Birthdate.year = year;
   pResult->pNext = 0;
   return pResult;
}
```

➡ Sisendparameetritest tuleb tingimata teha koopiad!

Üksiku kirje loomine ja eemaldamine (2)



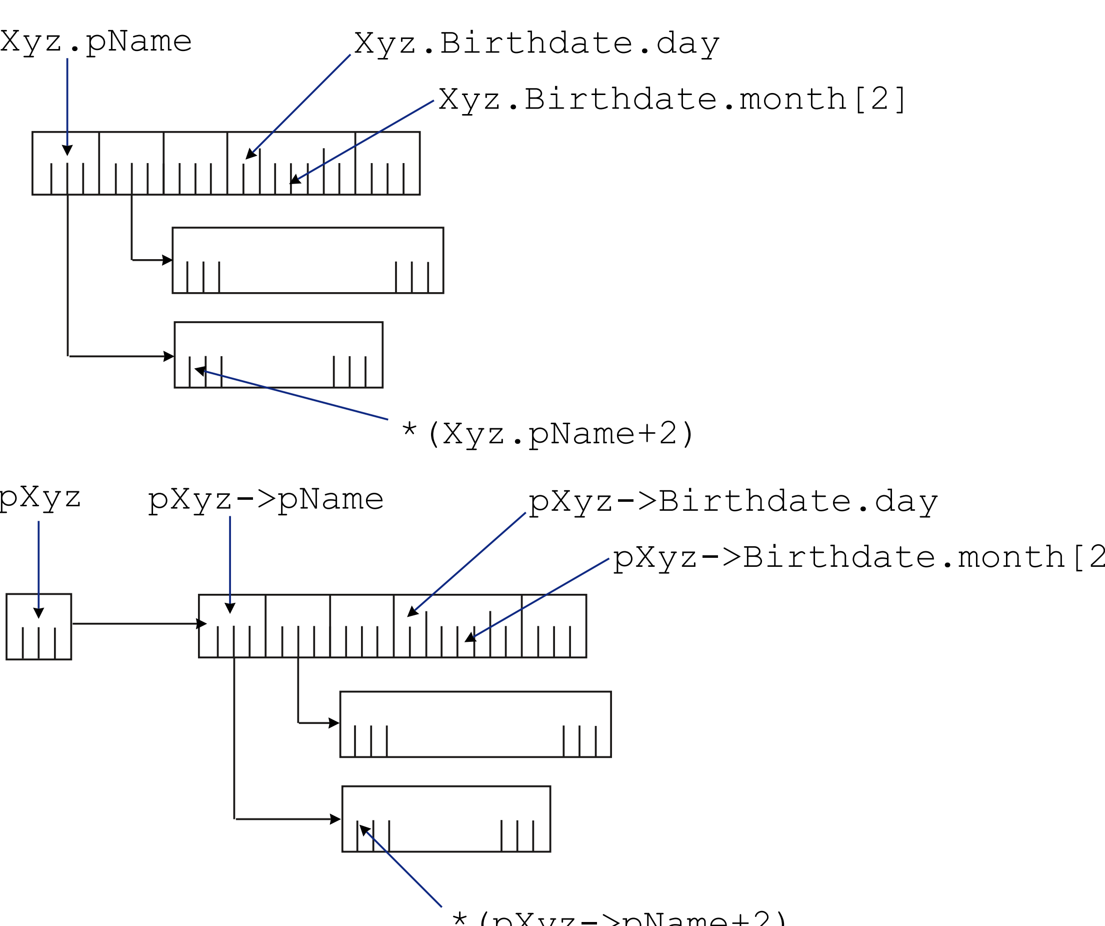
```
PERSON Xyz;
char buf[80];
get_s(buf, 80);
Xyz.pName = (char*)malloc(strlen(buf) + 1);
strcpy(Xyz.pName, buf);
get_s(buf, 80);
Xyz.pAddress = (char*)malloc(strlen(buf) + 1);
strcpy(Xyz.pAddress, buf);
get_s(buf, 80);
Xyz.Code = atol(buf);
get_s(buf, 80);
Xyz.Birthdate.day = (short)atoi(buf);
gets(Xyz.Birthdate.month);
get_s(buf, 80);
Xyz.Birthdate.year = (short)atoi(buf);
```

```
free(Xyz.pName);
free(Xyz.pAddress);
```

- Lihtsad ja kasulikud funktsioonid: get_s, atoi, atol, itoa, ltoa

Juurdepääs (access) kirje üksikutele liikmetele(members)

```
PERSON Xyz, *pXyz;
```



Andmestruktuurid (data structures) ja operatsioonid

Andmestruktuur(kirjete hulk) → Lineaarne (igal kirjel on järjekorranumber)
→ Mittelineaarne

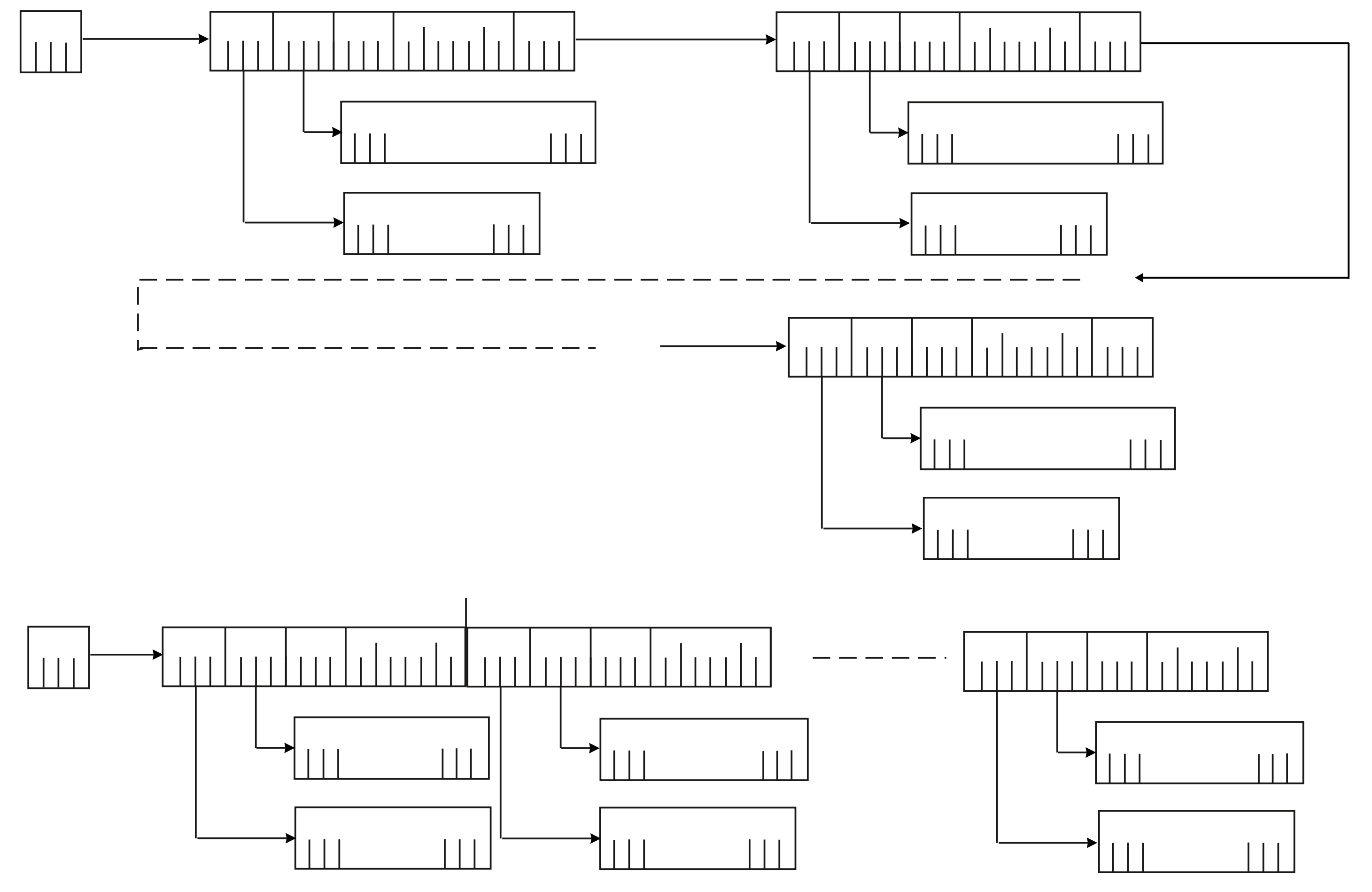
Tüüpilisi operatsioone andmestruktuuridel:

1. Tühja struktuuri loomine
2. Olemasoleva struktuuri hävitamine
3. Kirje lisamine (koos võimalike juhenditega paigutuse kohta)
4. Kirje otsimine struktuurist:
   - asukoha järgi
   - võtme järgi
5. Leitud kirje kopeerimine
6. Leitud kirje lokaliseerimine
7. Leitud kirje eemaldamine struktuurist
8. Leitud kirje ümberpaigutamine teisele positsioonile
9. Leitud kirjele järgneva või eelneva kirje leidmine
10. Suurima või vähima võtmega kirje leidmine
11. Andmestruktuuri sortimine
12. Andmestruktuuri mõõtmine
13. Kahe andmestruktuuri kokkumestimine

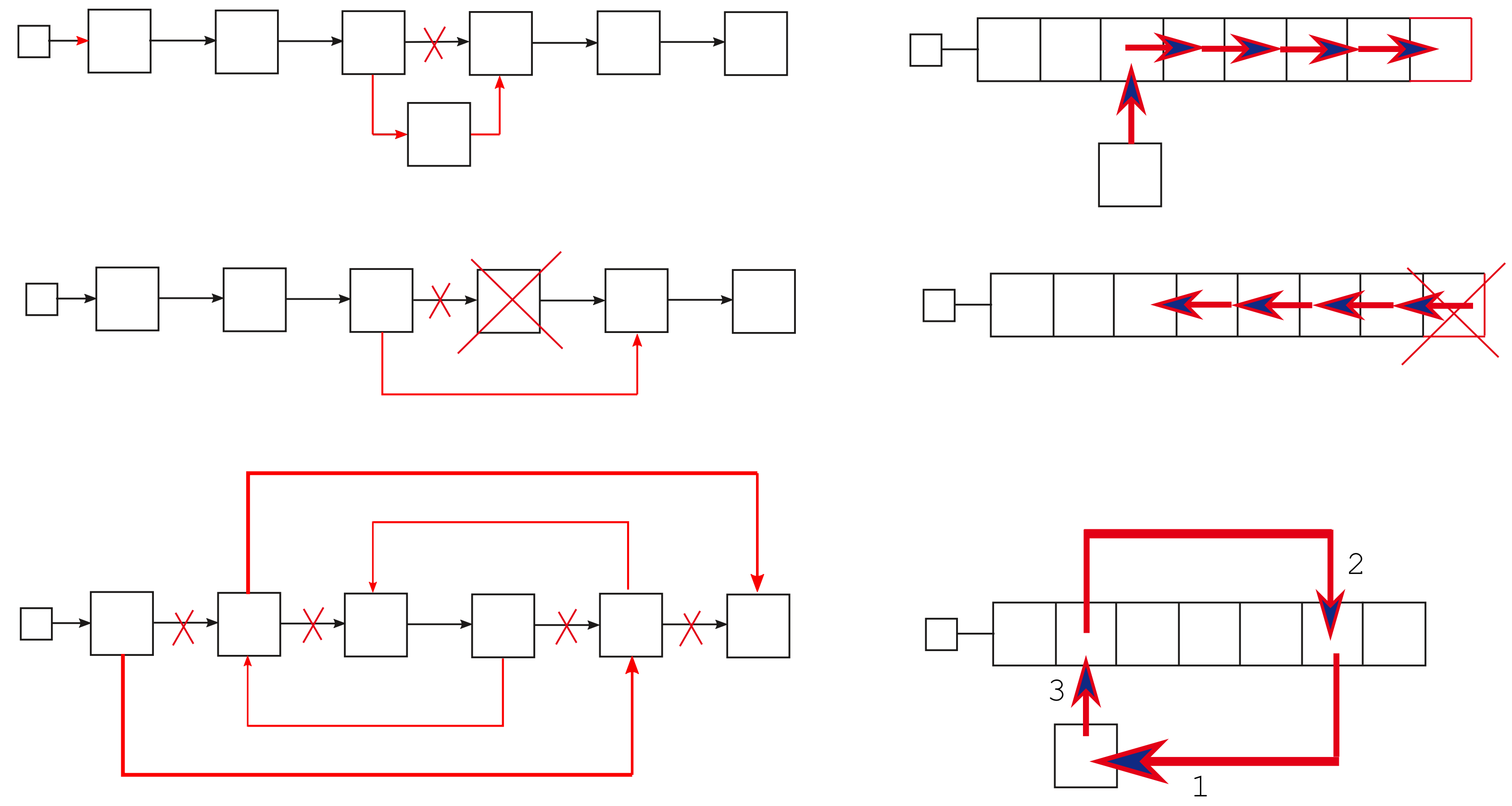
Võti:

- peab olema kas kirjest vahetult väljaloetav või kirje liikmete põhjal arvutatav
- kahe võtme kohta peab olema võimalik kindlaks määrata kas nad on võrdsed või siis kumb neist on suurem ja kumb väiksem
- ideaalsel juhul on igal kirjel oma unikaalne võti

Ahelloend (linked list) ja massiiv (array)

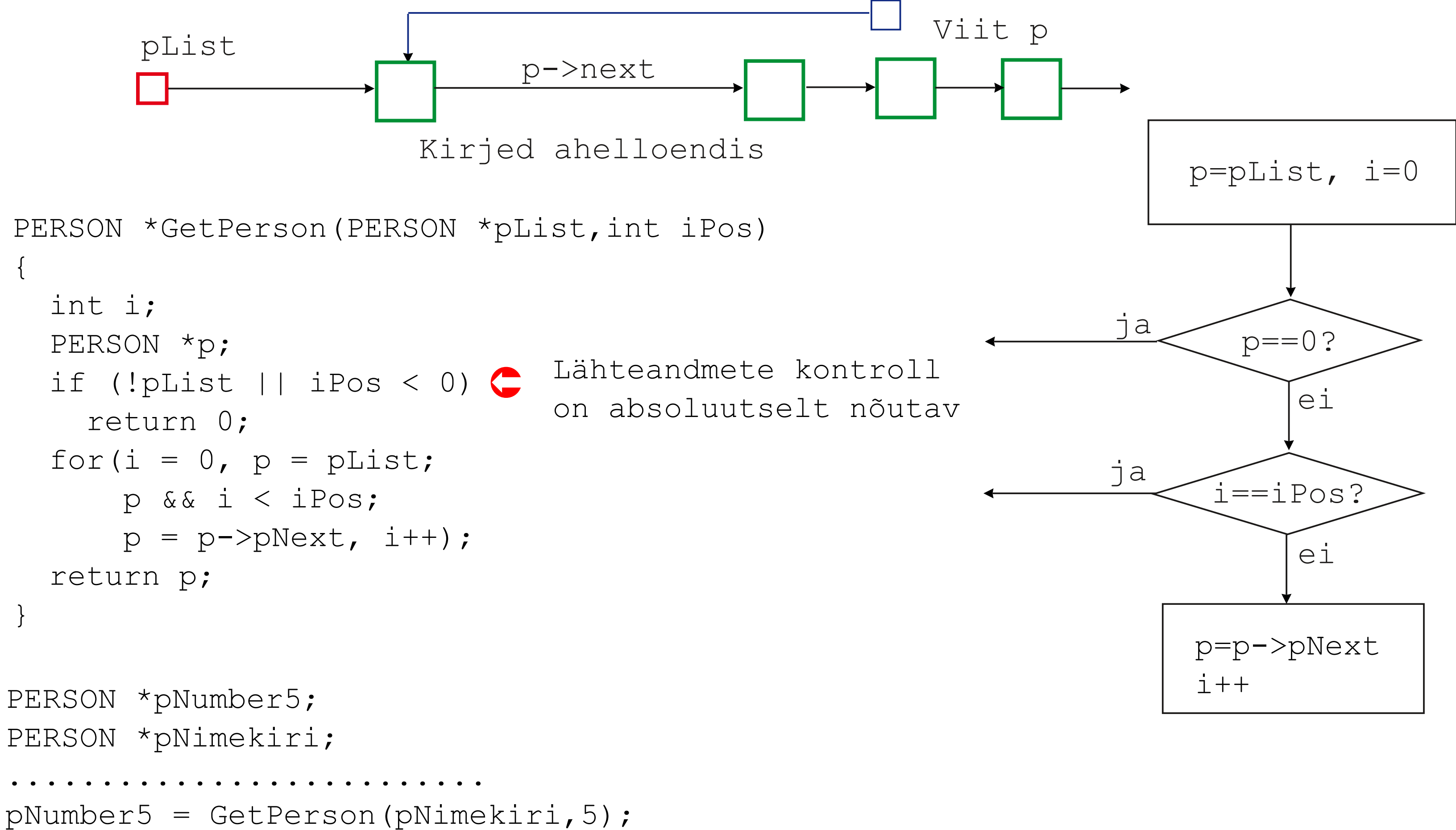


Massiiv ja ahelloend: mõningate operatsioonide võrdlus



➡ Massiivist kustutamisel on ka teine võimalus: märkida mingil kokkulepitud moel, et antud positsioon on tühi

Lineaarotsing (linear search) ahelloendist etteantud indeksi alusel




```
PERSON *GetPerson(PERSON *pList,int iPos)
{
  int i;
  PERSON *p;
  if (!pList || iPos < 0)
    return 0;
  for(i = 0, p = pList;
      p && i < iPos;
      p = p->pNext, i++);
  return p;
}

PERSON *pNumber5;
PERSON *pNimekiri;
..........................
pNumber5 = GetPerson(pNimekiri,5);
..............................
```

Tuleb veenduda, et töötab ka siis kui indeks on liiga suur

Lineaarotsing (linear search) ahelloendist etteantud võtme (key) alusel

pList

Viit p

p->next

Kirjed ahelloendis

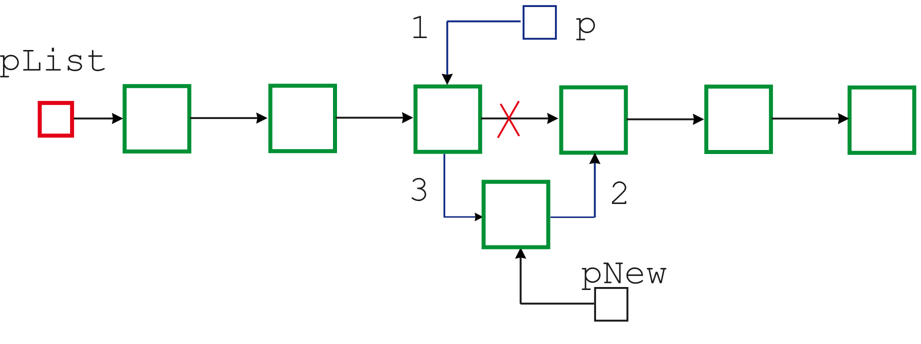
```
PERSON *GetPerson(PERSON *pList,char *pToFind)
{
  int i;
  PERSON *p;
  if (!pList || !pToFind)
    return 0;
  for(i = 0, p = pList;
      p && strcmp(pToFind, p->pName);
      p = p->pNext);
  return p;
}
```

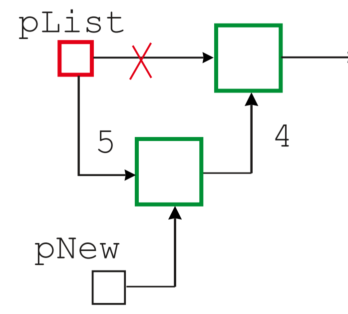
Lähteandmete kontroll on absoluutselt nõutav

```
PERSON *pJaanTamm;
PERSON *pNimekiri;
..........................
pJaanTamm = GetPerson(pNimekiri,"Jaan Tamm");
..............................
```

Tuleb veenduda, et töötab ka siis kui otsitavat isikut polegi

Kirje lisamine ahelloendisse (1)






```
PERSON *Insert(PERSON *pList, PERSON *pNew,int iPos)
{
 PERSON *p;
 if(!pNew)
  return pList;
 if(!iPos)
 {
  pNew->pNext = pList; // 4
  return pNew; // 5
 }
 if(p = GetPerson(pList, iPos - 1)) // 1
 {
  pNew->pNext = p->pNext; // 2
  p->pNext = pNew; // 3
 }
 return pList;
}
```

Tuleb kontrollida, et töötab ka siis kui

- algne nimekiri on tühi
- uus kirje läheb kõige esimeseks
- uus kirje läheb kõige viimaseks
- indeks on vale

Suur puudus: kasutaja ei saa teada, kas operatsioon õnnestus või ei

Kirje lisamine ahelloendisse (2)

```
#include stdlib.h
#include errno.h
PERSON *Insert(PERSON *pList, PERSON *pNew,int iPos)
{
 errno = 0;
 PERSON *p;
 if(!pNew)
 {
  errno = EINVAL;
  return pList;
 }
 if(!iPos)
 {
  pNew->pNext = pList;
  return pNew;
 }
 if(!(p = SearchByIndex(pList, iPos - 1)))
  errno = EINVAL;
 else
 {
  pNew->pNext = p->pNext;
  p->pNext = pNew;
 }
 return pList;
}
```

Pärast funktsioonist lahkumist tuleb väljakutsujal kontrollida standardse globaalse muutuja errno väärtust. Kui see ei ole null, lisamist ei toimunud.

Kirje lisamine ahelloendisse (3)

```
PERSON *Insert(PERSON *pList, PERSON *pNew,int iPos,int *pError)
{
 *pError = 0;
 PERSON *p;
 if(!pNew)
 {
  *pError = 1;
  return pList;
 }
 if(!iPos)
 {
  pNew->pNext = pList;
  return pNew;
 }
 if(!(p = SearchByIndex(pList, iPos - 1)))
  *pError = 1;
 else
 {
  pNew->pNext = p->pNext;
  p->pNext = pNew;
 }
 return pList;
}
```

Pärast funktsioonist lahkumist tuleb väljakutsujal kontrollida muutuja Error väärtust. Kui see ei ole null, lisamist ei toimunud.

Kirje eemaldamine ahelloendist

pResult
pResult
p
1
2
pList
pList
4
3
5

```
PERSON *Remove(PERSON *pList,int iPos PERSON **pResult) {
 PERSON *p;
 if(!pList) {
  *pResult = 0;
  return pList;
 }
 if(!iPos) {
  *pResult = pList;  // 4
  pList = pList->pNext;   // 5
  return pList;
 }
 if(!(p =GetPerson(pList, iPos -1))))   // 1
  *pResult = 0;
 else {
  *pResult = p->pNext;    // 2
  p->pNext = p->pNext->pNext;  // 3
 }
 return pList;
}
```

Tuleb kontrollida, et töötab ka siis kui

- nimekiri on tühi
- eemaldatav kirje on kõige esimene
- eemaldatav kirje on kõige viimane
- indeks on vale

Kasutamise näiteid

```
PERSON *p1 = CreatePerson("Jaan Tamm","Tallinn, Tartu mnt. 10-20",1234567,
10,"Jan", 1995);
PERSON *p2 = CreatePerson("Juhan Kask","Tartu, Tallinna mnt. 20-10",7654321,
11,"Feb", 1995);

PERSON *pList = 0;
pList = Insert(pList, p1, 0);
if (errno)
 printf("Failed!");
int Error = 0;
pList = Insert(pList, p2, 1, &Error);
if (Error)
 printf("Failed!");

PERSON *pNotNeeded;
pList = Remove(pList, 0, &pNotNeeded);
if (!pNotNeeded)
 printf("Failed!");
```

Eritüübiliste kirjete käsitlemine (1)

```
struct book // Raamat
{
 char *pAuthor,
      *pTitle;
 short int Year;
};

struct article // Artikkel ajakirjas
{
 char *pAuthor,
      *pTitle,
      *pJournal;
 short int Year,
           Number;
};
```

pTitle
pAuthor
Year

pJournal
Year
pTitle
Number
pAuthor

Eritüübiliste kirjete käsitlemine (2)

```
union reference // Üldistatud allikas
{
 struct book book; // liige tüübist struct book
 struct article article; // liige tüübist struct article
};

union reference allikas;
// eeldame, et allikas on raamat:
printf("%s\n", allikas.book.pTitle);
printf("%d\n", allikas.book.Year);
// eeldame, et allikas on artikkel:
printf("%s\n", allikas.article.pJournal);
printf("%d\n", allikas.article.Number);
```

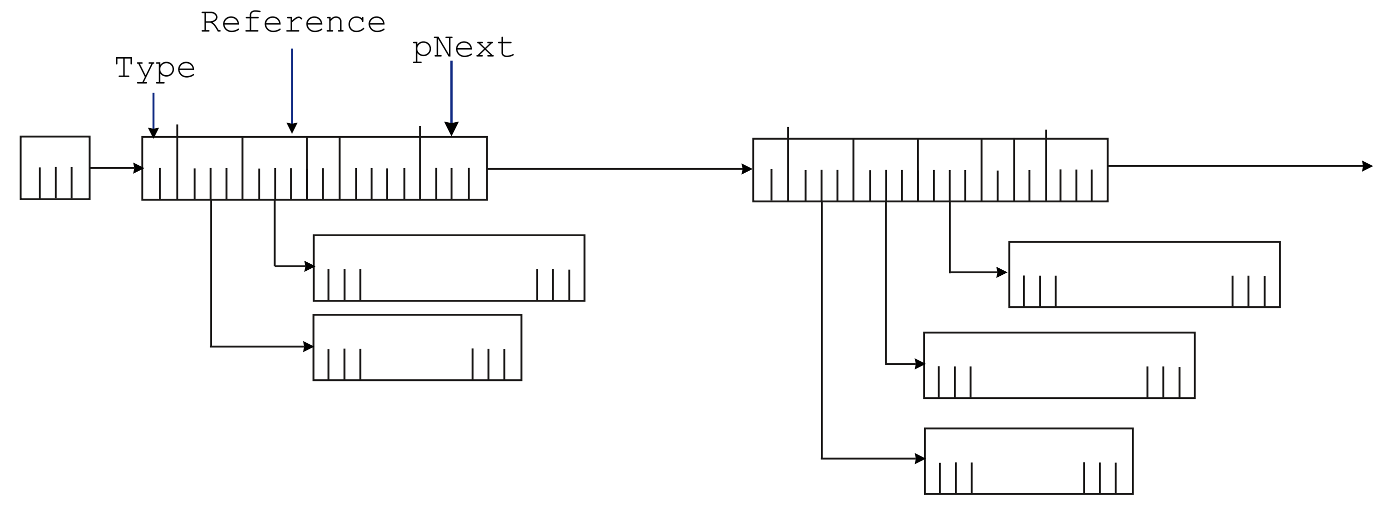
Kui eeldatakse valesti, on tulemuseks kas mõttetud suurused või siis programm kukub kokku.

kasutamata

Eritüübiliste kirjete käsitlemine (3)

```
#define BOOK 1
#define ARTICLE 2

struct entry // Loetelu ühik
{
 short int Type; // BOOK või ARTICLE
 union reference Reference; // ühik
 struct entry *pNext; // viit järgmisele
};
```



Eritüübiliste kirjete käsitlemine (4):

```
struct entry *pXyz;
struct entry *pLoetelu;
..........................
if (!(pXyz=Search(pLoetelu,"Peeter Ülitark")))
{
// Ei ole midagi kirjutanud
...............................
}
else if (pXyz->type==BOOK)
{
// Kirjutas raamatu
...............................
}
else
{
// Kirjutas artikli
..............................
}
```

Ahelloendite variatsioone (1)

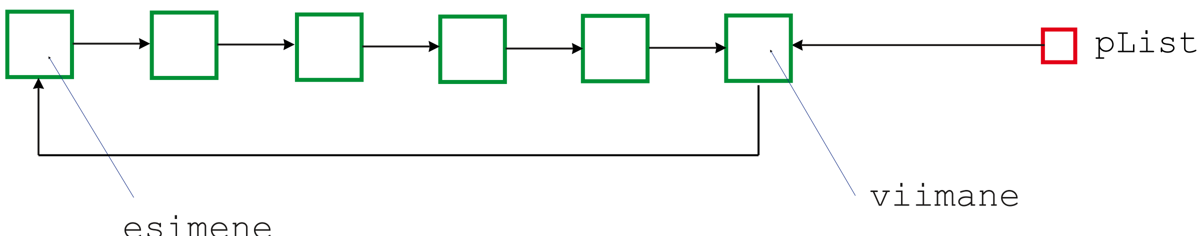
```
struct Person
{
 char *pName,
      *pAddress;
 long int Code;
 DATE Birthdate;
 struct Person *pNext, // viit järgmisele kirjele
               *pPrior; // viit eelmisele kirjele
};
```

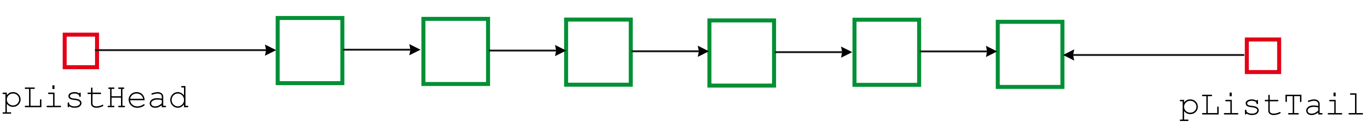
Kahekordselt lingitud ahelloend
(double linked list)

```
struct Header
{
  void *pRecord; // viit suvalist tüüpi kirjele
  int Type; // täisarv konkreetse tüübi markeerimiseks
  struct *Header *pNext;
};
```

Ahelloendite variatsioone (2)

Ringahelloend (circularly linked list)





Need loendid on eriti kasulikud kui uusi kirjeid tuleb alati lisada just ahelloendi lõppu

Viitade kasutamine täiendava info esitamiseks (1)

```
struct Student
{
 PERSON PersonalData;
 double Proficiency; // edukuse koeffitsient
 struct Student *pNextByName,
                *pNextByProficiency;
};
```

Kask 49.7, Kuusk 67.2, Lepp 41.4, Paju 93.8, Tamm 59.1

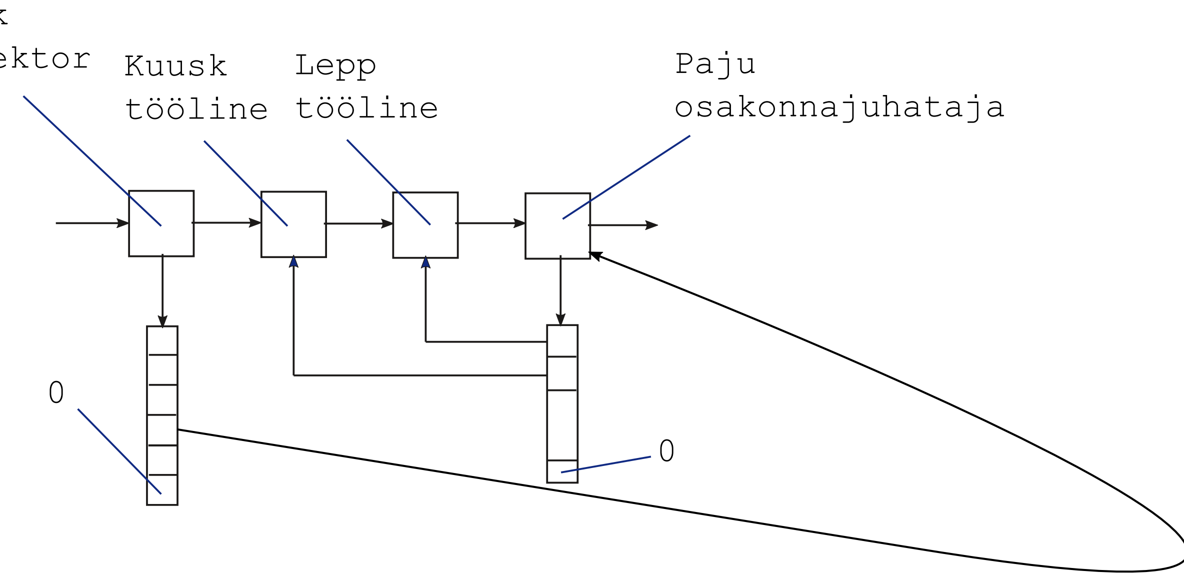
pListName

pListProf

```
void Pingerida(struct Student *pListProf)
{
 for (struct Student *p = pListProf;  p;  p->pNextByProficiency)
     printf("%s %lg\n", p->PersonalData.pName, p->Proficiency);
}
void Nimekiri(struct Student *pListName)
{
 for (struct Student *p = pListName;  p;  p->pNextByName)
     printf("%s %lg\n", p->PersonalData.pName, p->Proficiency);
}
```

Viitade kasutamine täiendava info esitamiseks (2)

```
struct employee // Firma töötaja
{
 PERSON PersonalData;
 char *pPost // Ametikoht
 struct employee *pNextInList, // Viit järgnevale üldnimekirjas
                 **ppSubordinates; // Viit vektorile, kust lähtuvad viidad
                                   // alluvatele, vektor lõpeb nulliga
};
```



Viitade kasutamine täiendava info esitamiseks (3)

```
struct employee // Firma töötaja
{
 PERSON PersonalData;
 char *pPost;
 struct employee *pNextInList, // viit järgnevale üldnimekirjas
                 *pSubordinate, // viit ühele alluvatest
                 *pChief, // viit vahetule ülemusele
                 *pColleague; // viit kaaslasele samast allüksusest
                              // või viit samal postsioonil olijale
};
```

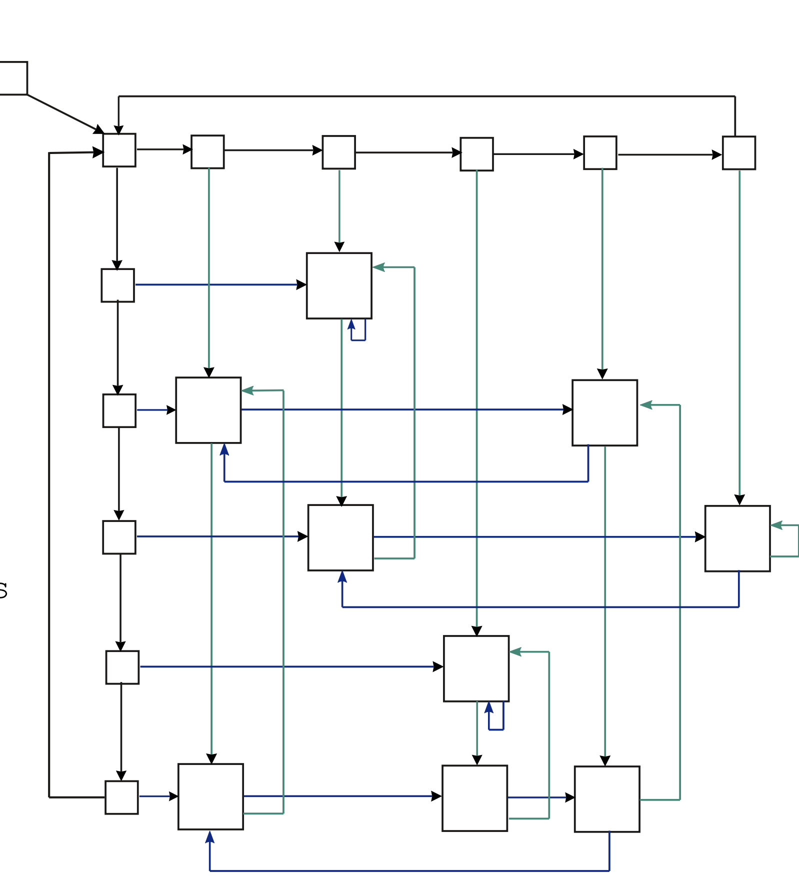
Direktor

Osakonnajuhataja

Tööline

Hõre maatriks (sparse matrix)

```
struct cell // Maatriksi struktuuri element
{
int Row; // rea indeks
int Column; // veeru indeks
RECORD Value; // sisu
struct cell *pNextInRow,
            *pNextInColumn;
};
```



```
0 1 0 0 0
1 0 0 1 0
0 1 0 0 1
0 0 1 0 0
1 0 1 1 0
```

Maatriksile lisatud 0-reas ja 0-veerus on kõik positsioonid täidetud. Ülejäänud mittenullised elemendid on ühendatud ringahelloenditesse.

Serialiseerimine (serialization)

p pName pAddress Code Birthdate pNext 0 0

pSer n pName pAddress Code Birthdate 1 2 0 3 0 4 5 r

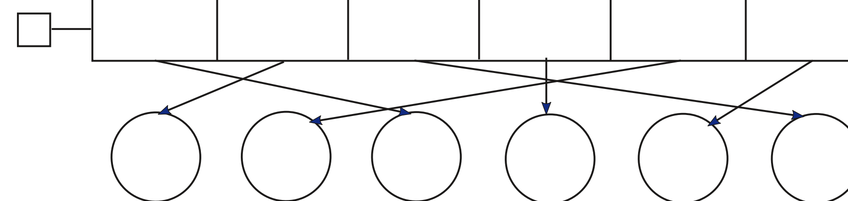
```
char *serialize(PERSON *p)
{
 short int n1,n2,n;
 char *pSer,*r;
 n1=strlen(p->pName)+1;
 n2=strlen(p->pAddress)+1;
 pSer=malloc(n=n1+n2+sizeof(PERSON)+sizeof(short int) - sizeof(PERSON *)-
             2*sizeof(char *));
 memcpy(r=pSer,&n,sizeof(short int)); //1
 memcpy(r+=sizeof(int),p->pName,n1); //2
 memcpy(r+=n1,p->pAddress,n2); //3
 memcpy(r+=n2,&p->Code,sizeof(long int)); //4
 memcpy(r+sizeof(long int),&p->Birthdate.day, sizeof(DATE)); //5
 return pSer;
}
```

Viitade vektor

```
struct Header
{
 void *pRecord;  // viit kirjele (võib olla suvalist tüüpi)
 int Type // täisarv, mis markeerib kirje tüüpi
};
```

Viitade vektor on väga mugav, kuid vaid siis kui me oskame hinnata kirjete arvu. Vektori pikendamine on väga resursse raiskav tegevus.

Kustutamisel on kõige targem asendada viit nulliga



Sortimisel jäävad kirjed paigale ja asukohti vahetavad üksnes viidad

Ringpuhver (circular buffer)

```
typedef struct buffer // Puhvri päis
{
 RECORD *pHead; // viit esimesele kirjele
 int Cursor, // viimase kirje indeks
     Length; //puhvri pikkus baitides
} BUFFER;

BUFFER *Create(int length)
{ // tühja puhvri loomine
 BUFFER *pBuf = (BUFFER *)malloc(sizeof(BUFFER));
 pBuf->Length=length;
 pBuf->pHead=(RECORD *)malloc(length);
 pBuf->Cursor=-1;
 return pBuf;
}
void Insert(BUFFER *pBuf,RECORD *pNew)
{ // kirje lisamine ringpuhvrisse
 if ((++pBuf->Cursor) * sizeof(RECORD) >= pBuf->Length)
   pBuf->Cursor=0; // hüpe puhvri algusesse
 *(pBuf->pHead+Buf->Cursor)=*pNew;
}
```

pBuf

pBuf->pHead

pNew

pNew

Kui kursor on jõudnud puhvri lõppu (s.t. puhver on täitunud), hüpatakse puhvri algusesse ja hakatakse üle kirjutama varem saabunud kirjeid. Ringpuhver eeldab et saabunud kirjeid jõutakse õigeaegselt töödelda. Kasutusel monitooringusüsteemides.

Abstraktsed andmetüübid (abstract data type,ADT)

Abstraktne andmetüüp

Matemaatiline mudel
+
võimalikud operatsioonid

Matemaatilised mudelid:
- järjestatud hulk (iga elemendi kohta saame ütelda, mitmes ta on e. mis numbriga positsioonil ta paikneb)
- järjestamata hulk (järjekorranumbreid pole, elemendi füüsiline asukoht ei oma tähtsust)

Järjestatud hulgal põhinevad abstraktseid andmetüüpe nimetatakse ka lineaarseteks.

Abstraktse andmetüübiga seotud tarkvara:
1. Realisatsioon (implementation) - andmestruktuur ja sellega töötamiseks vajalikud funktsioonid
2. Liides (interface) - andmestruktuuri kasutamiseks vajalikud funktsioonid
3. Kliendi tarkvara - andmestruktuuri liidese funktsioone väljakutsuvad funktsioonid

Lineaarsed abstraktsed andmetüübid

Loend (list): kõik elemendid on kättesaadavad (accessable)

Lubatavad operatsioonid:
Create, destroy, insert,
delete, retrieve, reallocate,
....

Magasin(pinu, stack, LIFO): ainult esimene element on kättesaadav. Uus kirje läheb alati esimeseks, eemaldada saab vaid esimest kirjet.

Lubatavad operatsioonid:
Create, destroy, push, pop, top

Järjekord(queue, FIFO): äärmised elemendid on kättesaadav. Eemaldada saab vaid kõige esimest kirjet, lisada saab ainult lõppu.

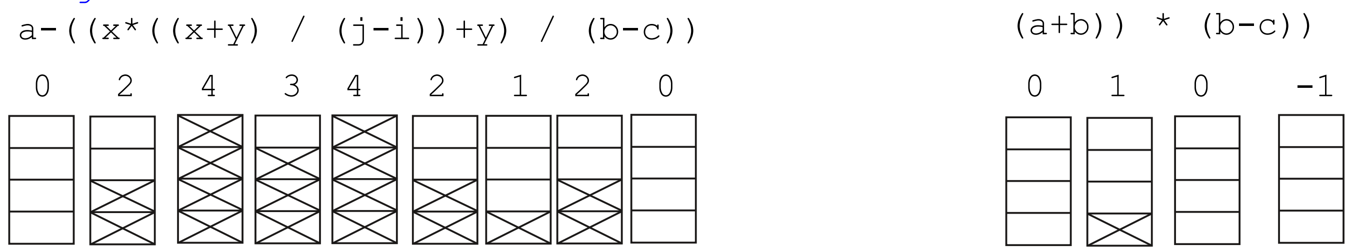
Lubatavad operatsioonid:
Create, destroy, put, get, front

Kahe otsaga järjekord (dequeue): äärmised elemendid on kättesaadavad. Lisada ja eemadada saab mõlemast otsast.

Lubatavad operatsioonid:
Create, destroy, put_front,
put_rear, get_front, get_rear,
front, rear

Näiteid magasini kasutamisest (1)

Sulgude tasakaalu kontroll:

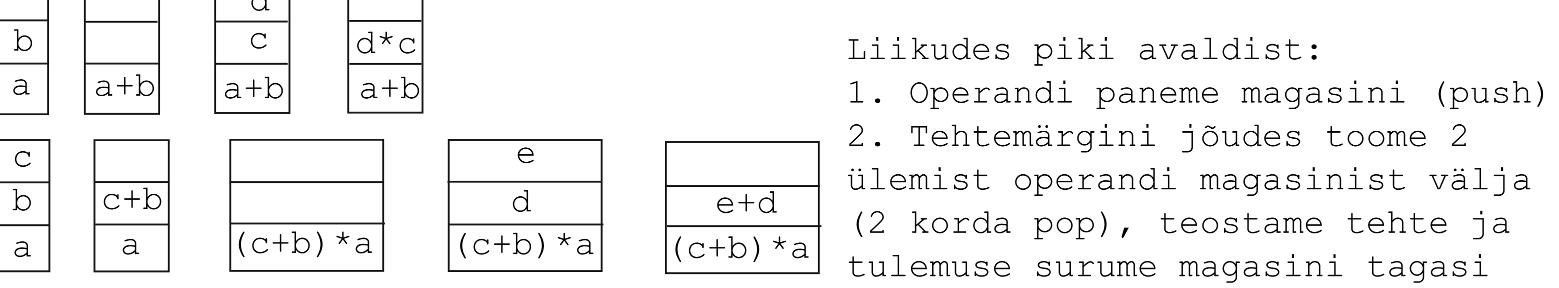


Liikudes piki avaldist paneme viida vasakpoolsele sulule magasini(push). Parempoolset sulgu kohates teeme pop. Nii saame sulud, mis moodustavad paari. Vea tunnus: katsutakse teha pop tühjast magasinist või siis lõppu jõudes magasin ei ole tühi.

Postfikseeritud notatsioon:

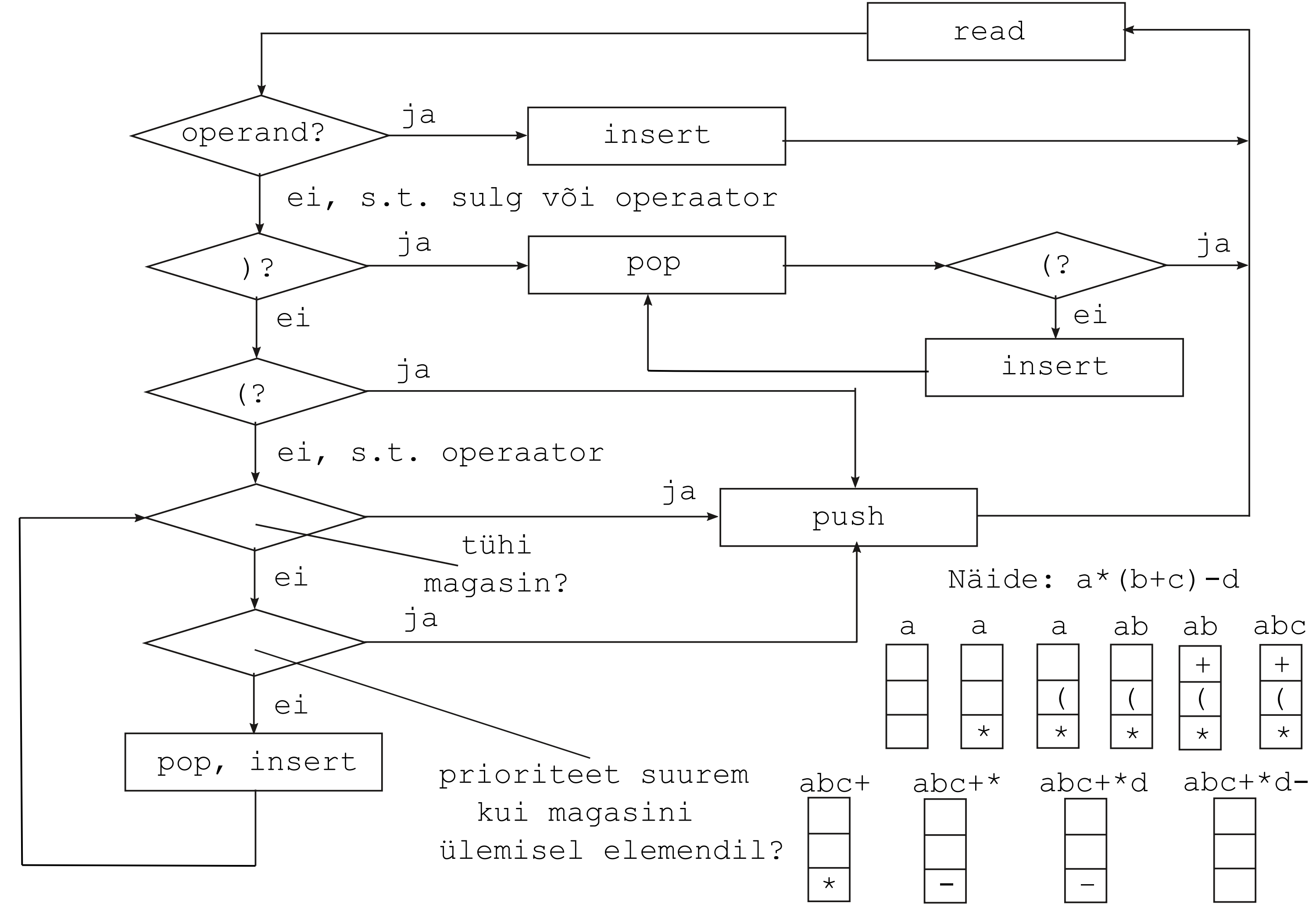
a+b-c*d --> ((a+b)-(c*d)) --> ((ab+)-(cd*)) --> ((ab+)(cd*))-) --> ab+cd*-
a*(b+c)-(d+e) --> ((a*(b+c))-(d+e)) --> ((a*(bc+)-(de+)) -->
((a(bc+)*(de+)) --> abc+*de+-

Arvutamine postfikseeritud kujust:

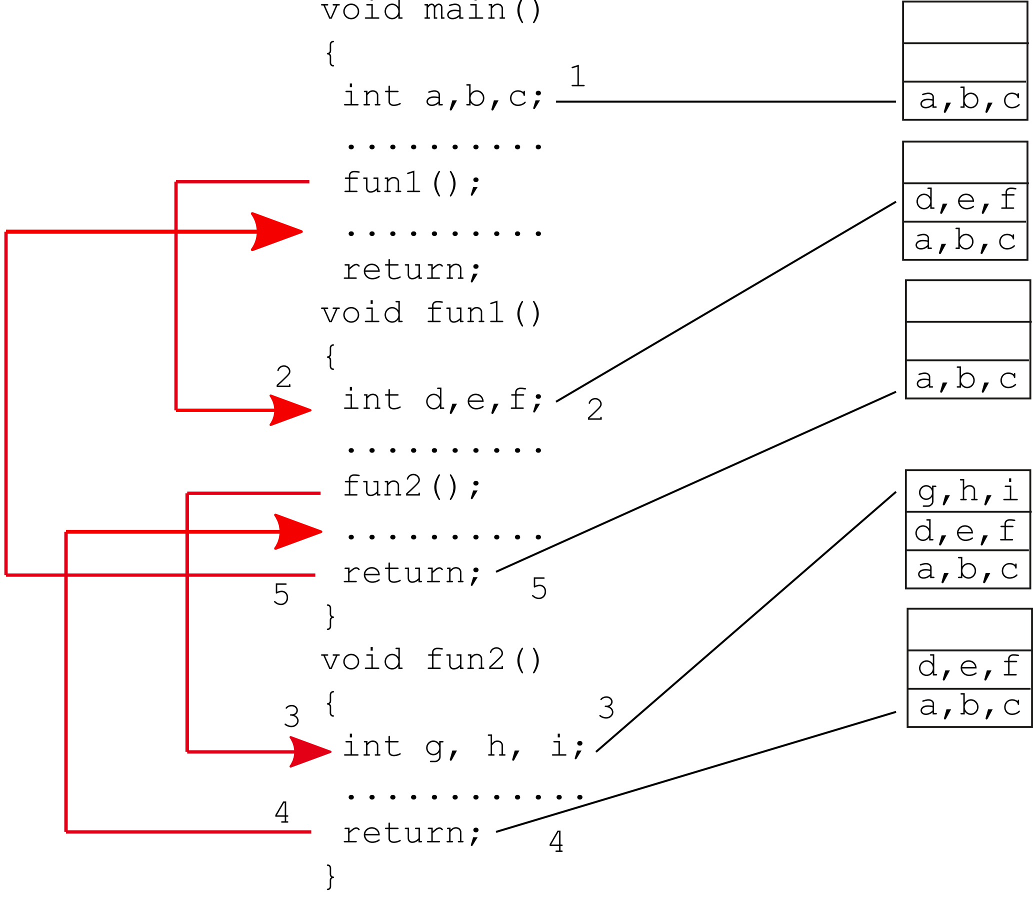


Liikudes piki avaldist:
1. Operandi paneme magasini (push).
2. Tehtemärgini jõudes toome 2 ülemist operandi magasinist välja (2 korda pop), teostame tehte ja tulemuse surume magasini tagasi (push).

Näiteid magasini kasutamisest (2)

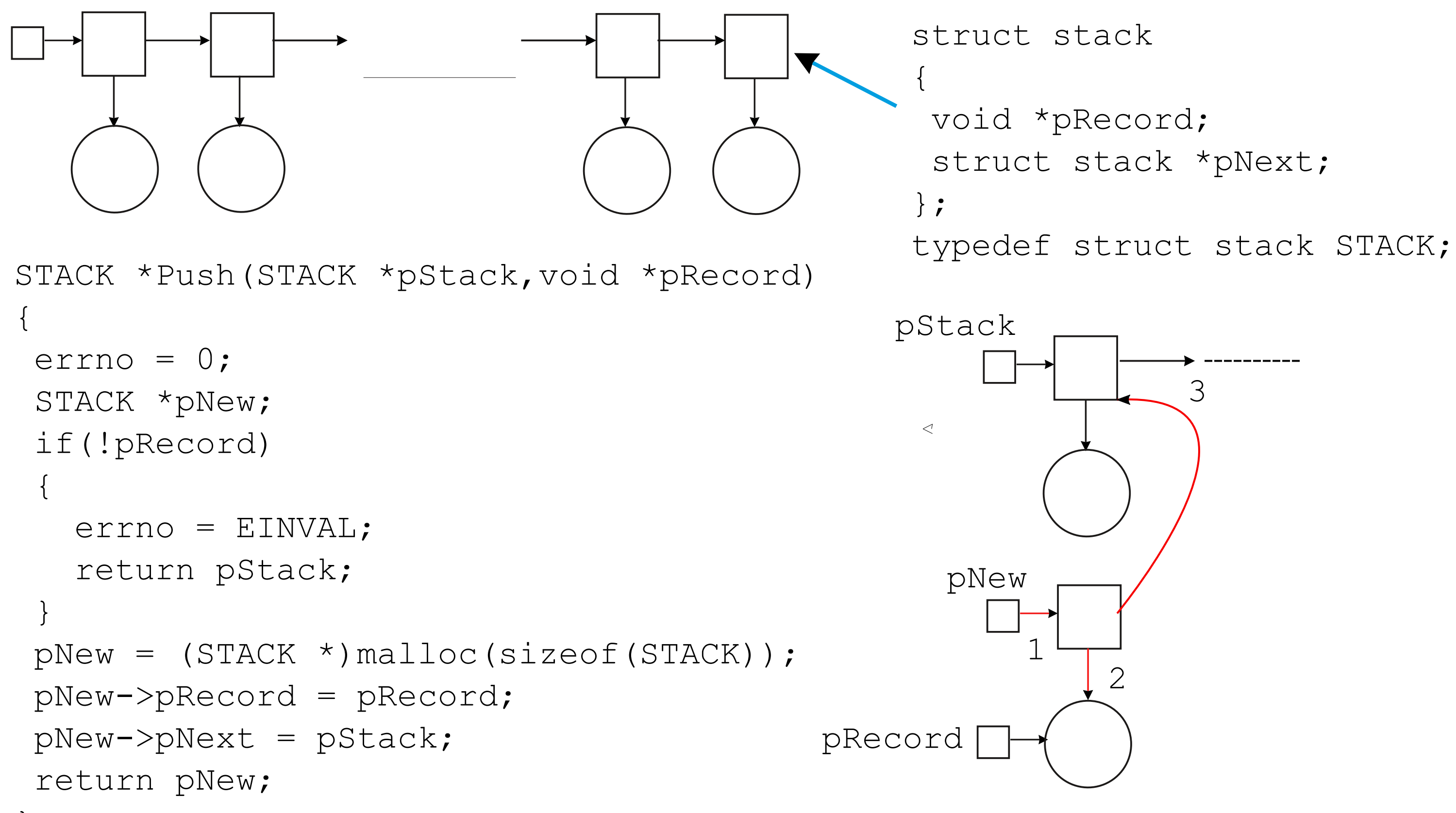


Näiteid magasini kasutamisest (3)



Lokaalmuutujatele eraldatakse mälu piirkonnast nimega "local variable stack"

Näide magasini realisatsioonist (1)



Näide magasini realisatsioonist (2)

```
STACK *Pop(STACK *pStack,void **pResult)
{
 STACK *p;
 if(!pStack)
 {
  *pResult = 0;
  return pStack;
 }
 *pResult = pStack->pRecord;
 p = pStack->pNext;
 free(pStack);
 return p;
}
```

pv

pResult

pStack

```
  void *pv;
  STACK *pMagasin;
  pMagasin = Pop(pMagasin, &pv);
  printf("%s", ((PERSON *)pv)->pName);
```

Rekursioon (recursion) (1)

```
unsigned long int fact(int n)
{
// faktoriaali arvutus
// iteratiivselt
 unsigned long int f;
 int i;
 if (n<=1)
   return 1;
 for (f=1,i=2; i<=n; f*=i, i++);
 return f;
}
```

```
unsigned long int fact(int n)
{
// faktoriaali arvutus
// rekursiivselt
 if (n<=1)
   return 1;
 else
   return n*fact(n-1);
}
```

Faktoriaali rekursiivne definitsioon:

0! = 1

n! = n * (n-1)! kus n = 1, 2, 3, ...

Rekursiivne definitsioon koosneb alati ankrust, mis paneb paika algväärtused ja reeglitest, millede kaudu juba olemasolevatest väärtustes leitakse uued.

Rekursiivne funktsioon kutsub välja iseennast. Samas peab rekursiivsel funktsioonil alati olema vähemalt üks väljund, kus ta iseennast välja ei kutsu.

Rekursioon (2)

```
unsigned long int fib(int i)
{
// Fibonacci arvu leidmine
// iteratiivselt
 unsigned long int fk,fk_1,fk_2,k;
 if (i<=1)
   return i;
 for (fk_1=1,fk_2=0, k=2;
      k<=i;
      fk=fk_2+fk_1, fk_2=fk_1, fk_1=fk, k++);
 return fk;
}
```

```
unsigned long int fib(int i)
{
// Fibonacci arvu leidmine
// rekursiivselt
 if (i<=1)
   return 1;
 else
   return fib(i-1)+fib(i-2);
}
```

Fibonacci rida:

f(0) = f(1) = 1

f(k) = f(k-1) + f (k-2) kui k > 1

1,1,2,3,5,8,13,21,....

f(6)=f(5)+f(4)

f(4)+f(3)

f(3)+f(2)

f(3)+f(2)

f(2)+f(1)

f(1)+f(0)

f(2)+f(1)

f(1)+f(0)

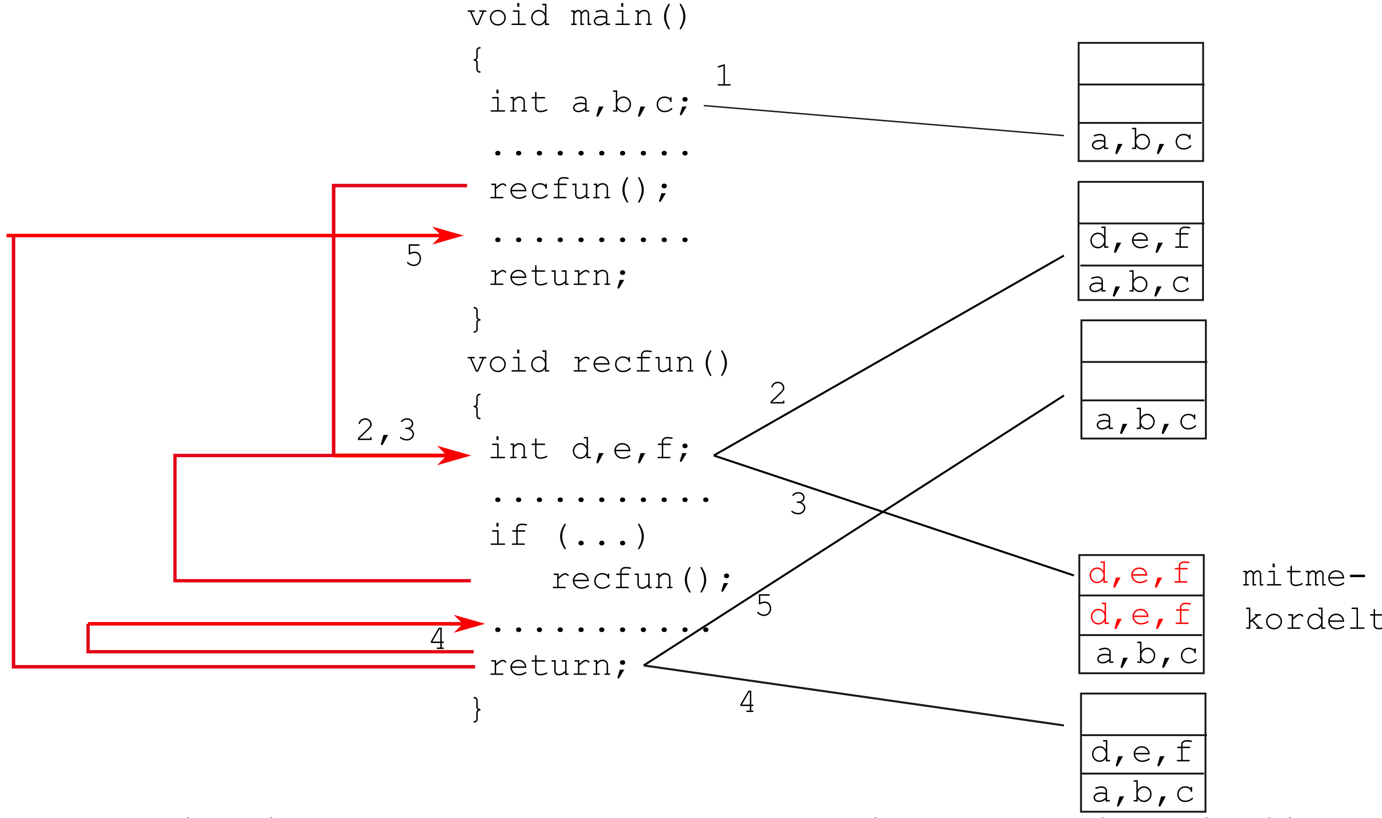
f(1)+f(0)

f(2)+f(1)

f(1)+f(0)

f(1)+f(0)

Rekursioon (3)



Rekursiivsed funktsioonid koormavad lokaalsete muutujate stack-i. Kui viimane täitub üle, võib programm hakata kummaliselt käituma.

Nn. tail recursion funktsioonidel on ainult üks rekursiivne pöördumine iseenda poole ning see asub päris funktsiooni lõpus (näiteks rekursiivne faktoriaal). Sellised funktsioonid lokaalmuutujate mälupiirkonda ei koorma.

Rekursioon (4)

Enamik andmetöötluse algoritme on oma olemuselt rekursiivsed. Et aga:
- rekursiivsed funktsioonid enamasti koormavad lokaalmuutujate stack-i
- neist on väga ebameeldiv vigu otsida

siis püütakse neid asendada iteratiivsete algoritmidega. Tail recursion puhul sobib rekursiooni asendamiseks tavaline tsükkel. Keerukamatel juhtudel tuleb appi võtta omaloodud stack.

Rekursioon (5):

```
char GetChar();
void PutChar();
void Reverse()
{ //rekursiivne variant
 char ch; // 1
 ch=GetChar(); // 2
 if (ch) // 3
 {
   Reverse(); // 4
   PutChar(ch); // 5
 }
 return; // 6
}
```

```
void Push(char);
char Pop();
void CreateStack();
int Empty();
void Reverse(void)
{ // asendav iteratiivne variant
 char ch;
 CreateStack();
 while (ch=GetChar())
   Push(ch);
 while(!Empty())
   PutChar(Pop());
 return;
}
```

GetChar loeb kusagilt märke (0 tähendab, et rohkem ei tule) ja PutChar paneb need kusagile. Meie eesmärgiks on märkide järjekord tagurpidi pöörata (s.t. kui sisse tuleb AB siis välja peab minema BA).

Olgu sisendandmeteks 'A' ja 'B'. Esimesel väljakutsel saab ch väärtuseks'A'. Et see ei ole 0, kutsutakse Reverse() uuesti välja (rida 4) ning ch saab väärtuseks 'B'. Kuna ka see pole 0, kutsutakse Reverse() veel kord välja. Nüüd saab ch väärtuseks 0, Reverse()-st lahkutakse realt 6 ja programmi täitmist jätkatakse realt 5 ch väärtusega 'B'. Pärast 'B'väljastamist lahkutakse jälle realt 6 ning jätkatakse realt 5, kusjures ch väärtuseks on nüüd 'A'. Pärast 'A' väljastamist lahkutakse Reverse()-st lõplikult.

Kahendotsing (binary search) (1)

15 25 28 30 32 36 37 58 61 68 75
[0] [5] [10]
37 58 61 68 75
37 58

n = 11, n / 2 = 5
Kui n on paaritu siis mõlema poole pikkus on n/2

15 25 28 30 32 36 37 58 61 68 75 89
[0] [6] [11]
37 58 61 68 75 89
37 58

n = 12, n / 2 = 6
Kui n on paaris siis vasaku poole pikkus on n/2 ja parema poole pikkus n/2-1

```
PERSON *BinarySearch(PERSON *pArray,int n,char* pKey) {
 PERSON *pCentre;
 int i;
 if(!pArray || !n || !pKey)
   return 0;
 pCentre = pArray + n / 2; // viit keskmisele kirjele
 if(!(i = strcmp(pCentre->pName, pKey)))
   return pCentre; // keskmine oligi otsitav
 else if(i < 0)// keskmise võti otsitavast väiksem, edasi otsime paremalt
   return BinarySearch(pCentre + 1, n % 2 ? n / 2 : n / 2 -1, pKey);
 else // keskmise võti otsitavast suurem, edasi otsime vasakult
   return BinarySearch(pArray, n / 2, pKey);
}
```

Kahendotsing (2)

```
PERSON *BinarySearch(PERSON *pArray,int n,char* pKey)
{ // kahendotsing ilma rekursioonita
 PERSON *pCentre, *pTemp;
 int i;
 if(!pArray || !pKey)
   return 0;
 for (pTemp = pArray; n;)
 {
   pCentre = pTemp + n / 2;
   if (!(i = strcmp(pCentre->pName, pKey)))
      return pCentre;
   else if (i < 0) // keskmise võti otsitavast väiksem
   {
      pTemp = pCentre + 1;
      n = n % 2 ? n / 2 : n / 2 -1;
   }
   else // keskmise võti otsitavast suurem
      n /= 2;
   }
 return 0;
}
```

Kahendotsing (3):

Standardne kahendotsingu funktsioon:

```
void *bsearch(const void *key, const void *base, size_t nitems, size_t size,
              int (*compar)(const void *, const void *));
```

Parameetri compar tüübiks on viit funktsioonile, mile sisendiks on 2 viita void * ja väljundiks täisarv.

```
int CompareKeys(const void *pKey,const void *pRecord)
{
 return strcmp((const char*)pKey, ((const PERSON *)pRecord)->pName);
}


voidTestStandardBinarySearch()
{
 PERSON *pArray = (PERSON *)malloc(8 *sizeof(PERSON));
 *(pArray + 0) = *CreatePerson("Jaak","Kuressare", 123, 10,"Jul", 1995);
 *(pArray + 1) = *CreatePerson("Jaan","Tallinn", 123, 10,"Jan", 1995);
 *(pArray + 2) = *CreatePerson("Juhan","Viljandi", 125, 12,"Mar", 1995);
 *(pArray + 3) = *CreatePerson("Jyri","Tartu", 124, 11,"Feb", 1995);
 *(pArray + 4) = *CreatePerson("Paul","Antsla", 123, 10,"Jun", 1995);
 *(pArray + 5) = *CreatePerson("Peeter ","Rakvere", 127, 14,"May", 1995);
 *(pArray + 6) = *CreatePerson("Priit","Kadrina", 123, 10,"Aug", 1995);
 *(pArray + 7) = *CreatePerson("Toomas","Rapla", 126, 13,"Apr", 1995);
 PERSON *p= (PERSON *)bsearch("Priit", pArray, 8, sizeof(PERSON), CompareKeys);
}
```

Kahendotsing (4):

Kahendotsing eeldab otsimisel kasutatava võtme järgi sorditud andmekogumit, vastasel korral ta ei tööta.

Ahelloendite puhul kahendotsingu kasutamine ei oma mõtet, sest indeksiga opereerimine ahelloendis on ebaratsionaalne.

Sorditud massiivide puhul on kahendotsing suurepärane vahend.

Algoritmide keerukus(Big-O complexity) (1)

Analüüs — teoreetiline / empiiriline

Reaalse aja asemel korduvate operatsioonide arv sõltuvalt kirjete arvust (T(n)).
Keskmise juhu analüüs ja halvima juhu analüüs (average case, worst case).

Järjestikulise otsimise puhul:
$T(n)=(1+2+...+n)/n=(n+1)/2$
$T(n)=n$

T(n) on kasvukiirusega $\mathcal{O}(f(n))$, kui leiduvad konstandid n_0 ja c nii, et kui $n>n_0$, siis $T(n)<c*f(n)$.

$f(n)=n$ - lineaarne algoritm
$f(n)=\log_2 n$ - logaritmiline algoritm
$f(n)=n*\log_2 n$ - lineaarlogaritmiline algoritm
$f(n)=n^2$ - kvadraatne algoritm
$f(n)=2^n$ - eksponentsiaalne algoritm.

Tüüpiliste andmetöötlusoperatsioonide keerukus erinevatel andmestruktuuridel on esitatud lehel http://bigocheatsheet.com/

Algoritmide keerukus (2)

n	lineaarne	logaritmiline	lineaarlogaritmiline	kvadraatne
10^1	10^1	3	33	10^2
10^2	10^2	7	664	10^4
10^3	10^3	10	9966	10^6
10^4	10^4	13	132877	10^8
10^5	10^5	17	1660964	10^{10}
10^6	10^6	20	19931569	10^{12}

Kui eeldada, et arvuti teeb miljon operatsiooni sekundis, siis:

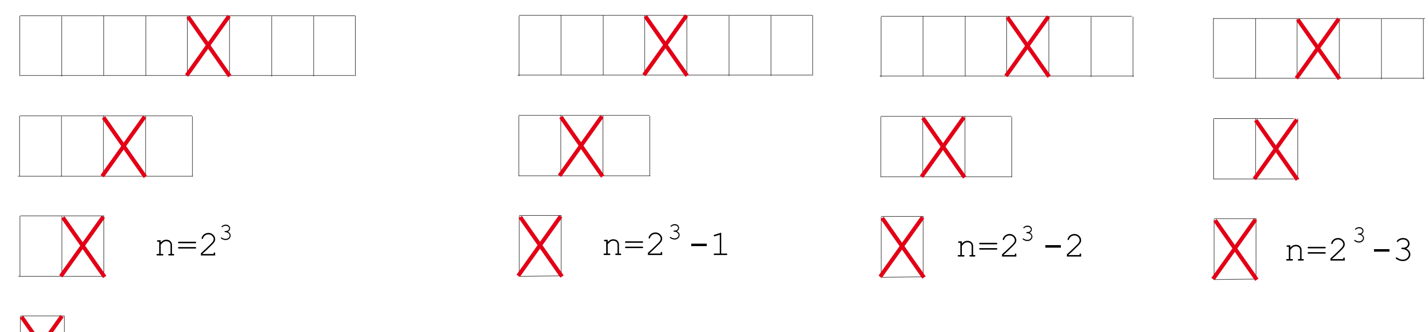
1. Tuhande kirje sortimiseks kulub:
- kvadraatse algoritmiga max 1s
- lineaarlogaritmilisega max 10ms

2. Miljoni kirje sortimiseks kulub:
- kvadraatse algoritmiga max 1.6 nädalat
- lineaarlogaritmilisega max 20s

3. 10 miljoni kirje sortimiseks kulub:
- kvadraatse algoritmiga max 3.1 aastat
- lineaarlogaritmilisega max 5 minutit

Algoritmide keerukus (3)

Tihtipeale saab algoritmide keerukust leida lihtsate aruteludega. Näiteks kahendotsimise puhul:



$T(n) \leq k+1$, kui $n=2^k$ $T(n) \leq k$, kui $n=2^{k-1} < n < 2^k$

Et $k=\log_2 n$, siis $T(n) \leq \log_2 n+1$ või $T(n) \leq \log_2 n$

Siit võime järeldada, et kahendotsimine on logaritmilise iseloomuga

Järjestikulise otsimise täiustamine

Kui loend on sorditud sama võtme järgi, millega ka otsitakse, siis otsitava kirje puudumisel pole vaja lõpuni liikuda. Seega eelnev sortimine võib mõnel juhul otsimise käiku kiirendada.
Loomulikult oleks kasulik koondada need kirjed, mida tõenäoliselt otsitakse sagedamini, loendi algusesse.

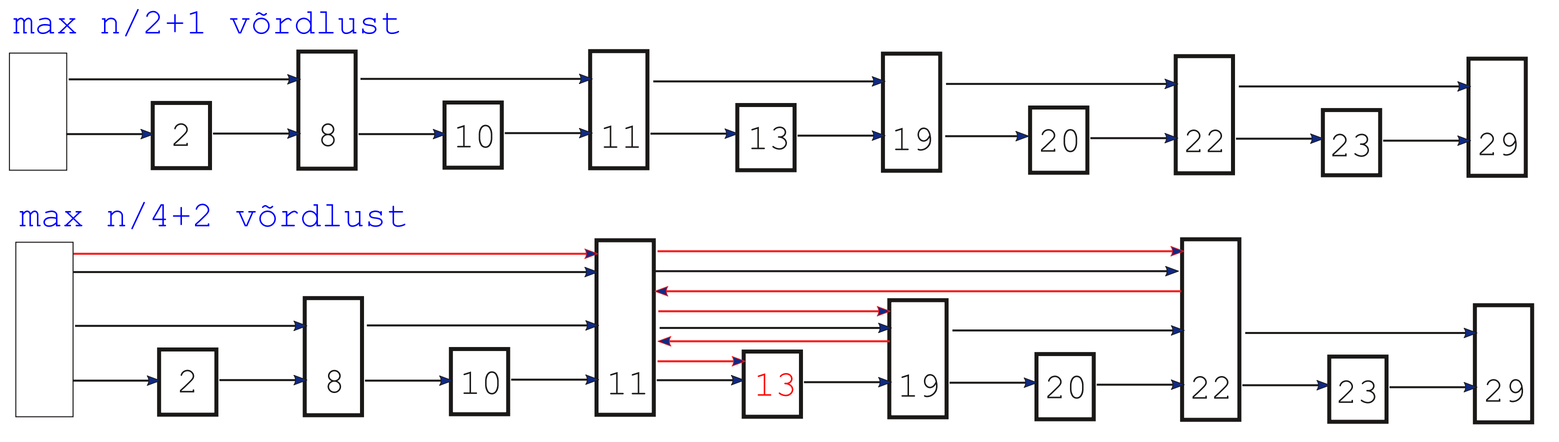
Eksperthinnangud: 80% - 20% reegli alusel
- 80% päringuist hõlmab 20% kirjetest
- 64% päringuist hõlmab 4% kirjetest
- 51.2% päringuist hõlmab 0.8% kirjetest

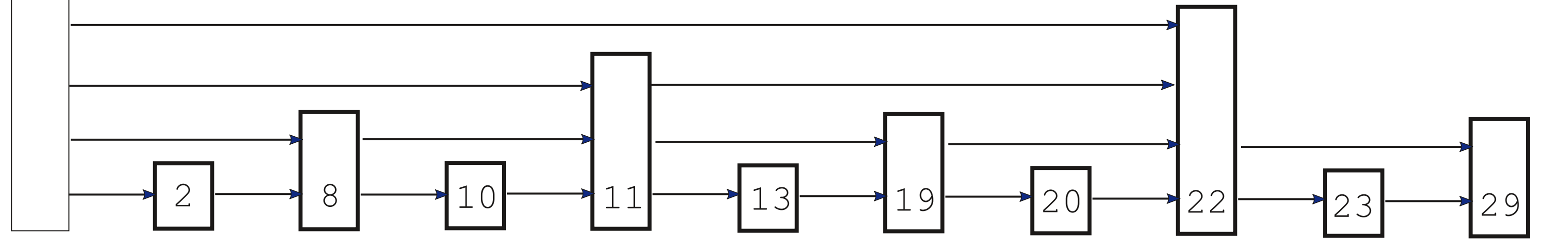
Iseorganiseeruv järjestikuline otsimine:
- “paiguta ette” meetodil tuuakse leitud kirje esimeseks
- “vaheta asukohad” meetodil paigutatakse leitud kirje ühe koha võrra ettepoole

Mõlemal juhul koonduvad sagedamini nõutavad kirjed pikapeale loendi algusesse

Lisaviitadega ahelloendid (skip lists) (1)

2 8 10 11 13 19 20 22 23 29 max n võrdlust





$8 \le n < 15$ max 4 võrdlust

$16 \le n < 23$ max 5 võrdlust

....

$2^k \le n < 2^{k+1} - 1$ max k+1 võrdlust

Alustame liikumist kõige ülemisel nivool. Kui näeme, et oleme juba otsitavast üle hüpanud, läheme üks samm tagasi ja üks nivoo allapoole.

Lisaviitadega ahelloendid (2)

Lisaviitadega ahelloendist otsimine on sama jõudlusega kahendotsimisega (s.t. logaritmilise iseloomuga) kui n kirje puhul nivoode arv k rahuldab võrratust $2^{k-1} \leq n < 2^{k}$

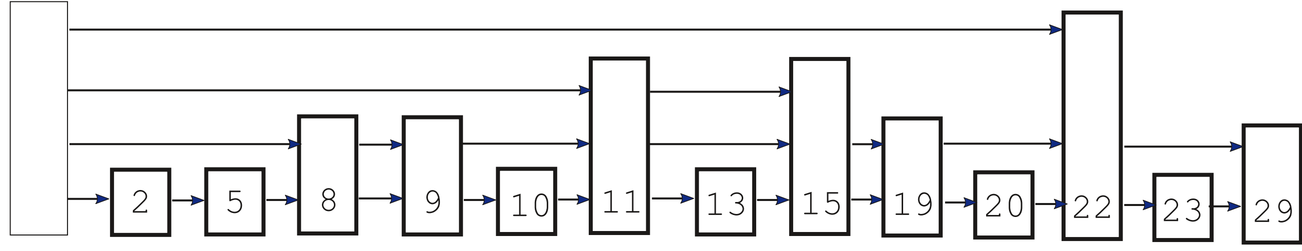
Probleemiks on siin see, et kirjete lisamisel ja eemaldamisel esialgne korrapärane struktuur ei saa säiluda. Probleemi lahendamiseks võib kasutada järgmist mõttekäiku:

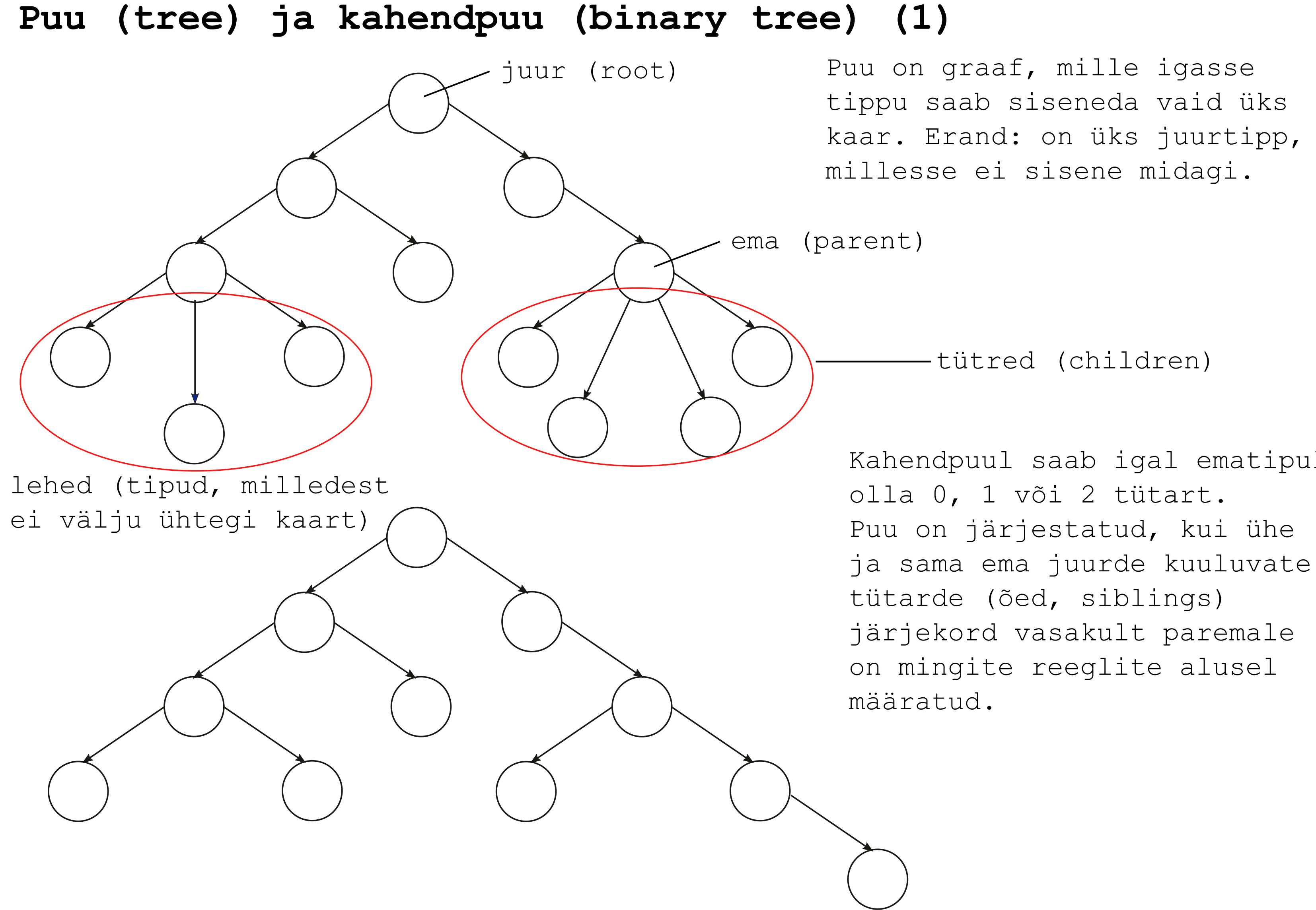
Umbes pooltel kirjetest on 1 viit,
veerandil 2 viita,
....
$1/2^{k}$ k viita

Seega on vaja juhuslikke täisarve genereerivat funktsiooni, mis näiteks neljanivoolise ahelloendi puhul väljastaks:

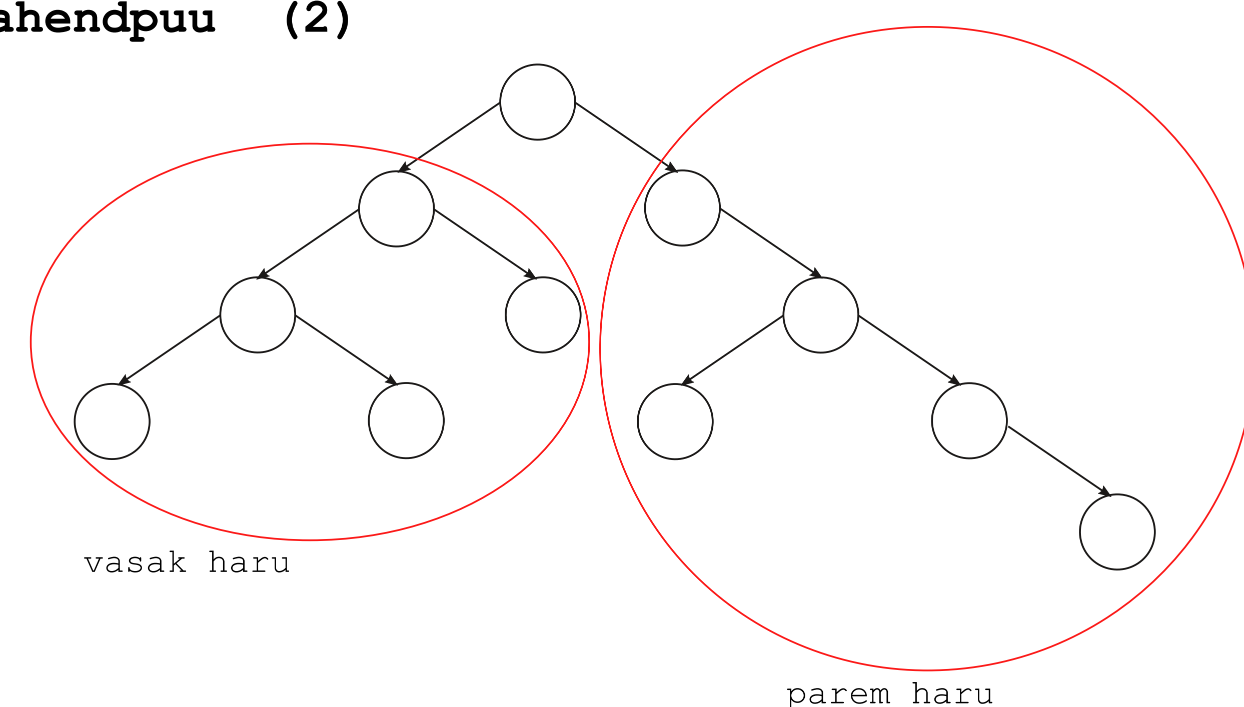
tõenäosusega 0.5 on väljundsuurus 1
0.25 on väljundsuurus 2
0.125 on väljundsuurus 3
0.00625 on väljundsuurus 4

Tulemuseks saaksime struktuuri, mis ei ole küll korrapärane kuid võimaldab siiski kiiremat otsimist, näiteks:




Puu (tree) ja kahendpuu (binary tree) (1)
juur (root)
Puu on graaf, mille igasse tippu saab siseneda vaid üks kaar. Erand: on üks juurtipp, millesse ei sisene midagi.
ema (parent)
tütred (children)
lehed (tipud, milledest ei välju ühtegi kaart)
Kahendpuul saab igal ematipul olla 0, 1 või 2 tütart.
Puu on järjestatud, kui ühe ja sama ema juurde kuuluvate tütarde (õed, siblings) järjekord vasakult paremale on mingite reeglite alusel määratud.

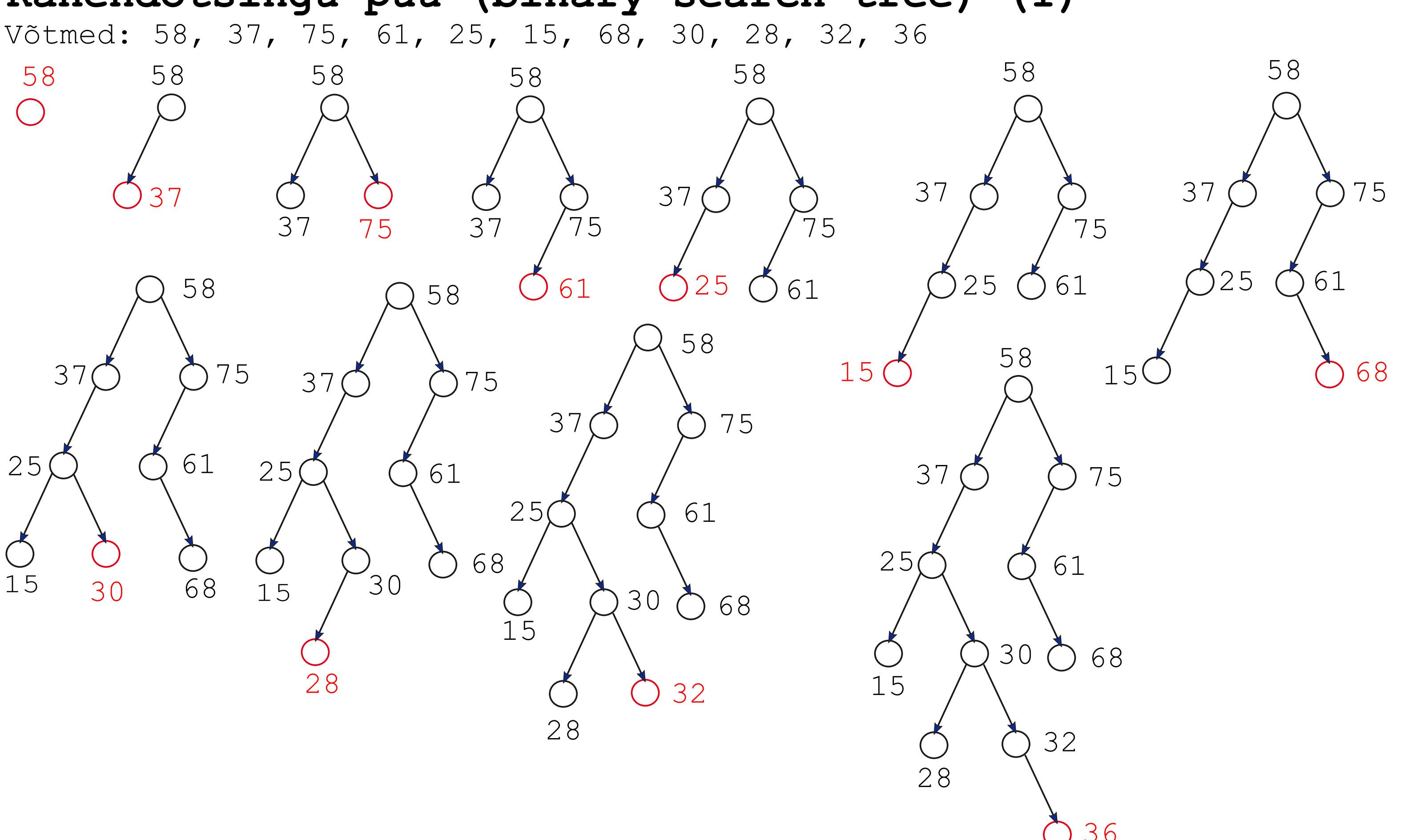
Puu ja kahendpuu (2)



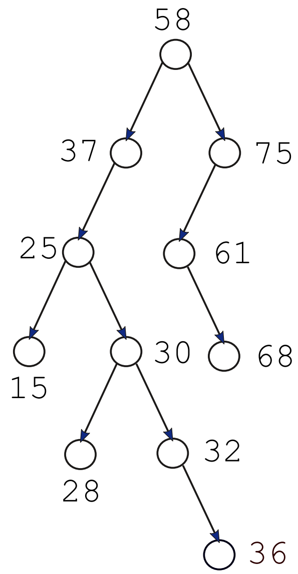
Puu on kas tühi või ta koosneb ainult juurtipust või siis juurtipust ja harudest, viimased omakorda on samuti puud. See on puu rekursiivne definitsioon.
Omavahel sidumata puude hulka nimetatakse metsaks.

Kahendotsingu puu (binary search tree) (1)

Võtmed: 58, 37, 75, 61, 25, 15, 68, 30, 28, 32, 36



Kahendotsingu puu (2)



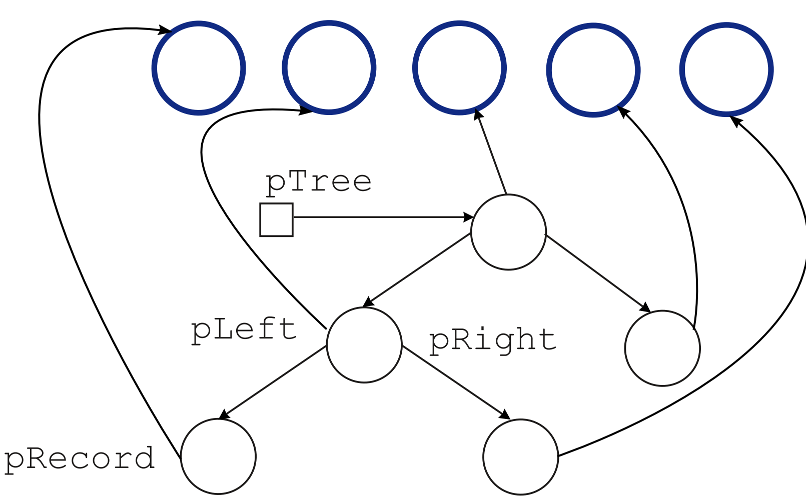
Puu ehitamise reeglid:
1. Ehita puu juur ja seosta see esimese kirjega.
2. Uue kirja lisamiseks liigu piki puud, võrreldes igas tipus uue kirje võtit tipuga seostatud kirje võtmega. Kui uue kirje võti on väiksem, mine vasakule. Kui uue võti on suurem, mine paremale.
3. Kui jõudsid tühja kohta, paiguta sinna uus tipp ja seosta see uue kirjega.

Puust otsimise reeglid:
1. Liigu juurest alates piki puud, võrreldes igas tipus otsitavat võtit antud tipuga seostatud kirje võtmega. Kui otsitav võti on väiksem, mine vasakule. Kui otsitav võti on suurem, mine paremale.
2. Kui võtmed langevad kokku, on otsitav kirje leitud. Kui satud tühja kohta, siis sellise võtmega kirjet ei olegi.

Kahendotsigu puu on järjestatud. Vasakpoolse õe võti on alati väiksem kui parempoolse õe võti.

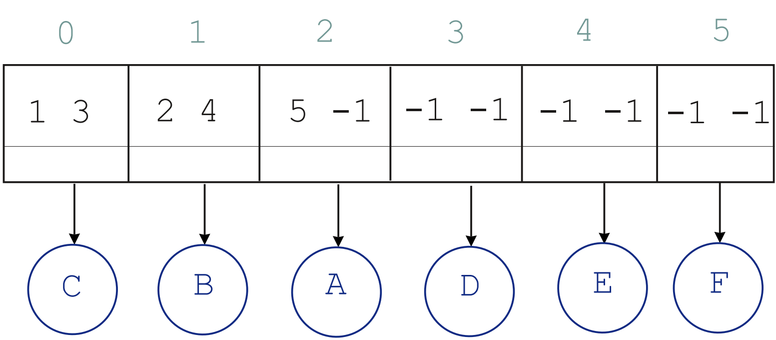
Kahendotsingu puu andmestruktuurina (1)

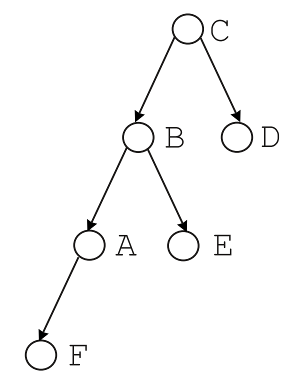
```
struct node // TIPP
{
 void *pRecord; // viit kirjele
 struct node *pLeft, // viit vasakpoolsele tütrele
             *pRight; // viit parempoolsele tütrele
};
typedef struct node NODE;
NODE *pTree = 0;
```



Kahendotsingu puu andmestruktuurina (2)

```
struct node // TIPP
{
 void *pRecord; // viit kirjele
 int left, // vasakpoolse tütre indeks, -1 kui tühi
 right; // parempoolsele tütre indeks, -1 kui tühi
};
typedef struct node NODE;
```





Kirje otsimine kahendpuust (1)

```
void* TreeSearch(NODE *pTree,void *pKey,int(*pCompare)(const void*,const void*))
{ // rekursiivne funktsioon kahendpuust otsimiseks
 inti;
 if(!pTree || !pKey)
   return 0;
 if (!(i = (pCompare)(pKey, pTree->pRecord)))
   return pTree->pRecord;// leidsime
 else if (i < 0)
   return TreeSearch(pTree->pLeft, pKey, pCompare);
 else
   return TreeSearch(pTree->pRight, pKey, pCompare);
}
```

Parameetri pCompare tüübiks on viit funktsioonile, mile sisendiks on 2 viita void * ja väljundiks täisarv. Näide:

```
int CompareKeys(const void *pKey,const void *pRecord)
{
 return strcmp((const char*)pKey, ((const PERSON *)pRecord)->pName);
}
```

```
PERSON *p = (PERSON *)TreeSearch(pTree,"Paul Vaher", CompareKeys);
```

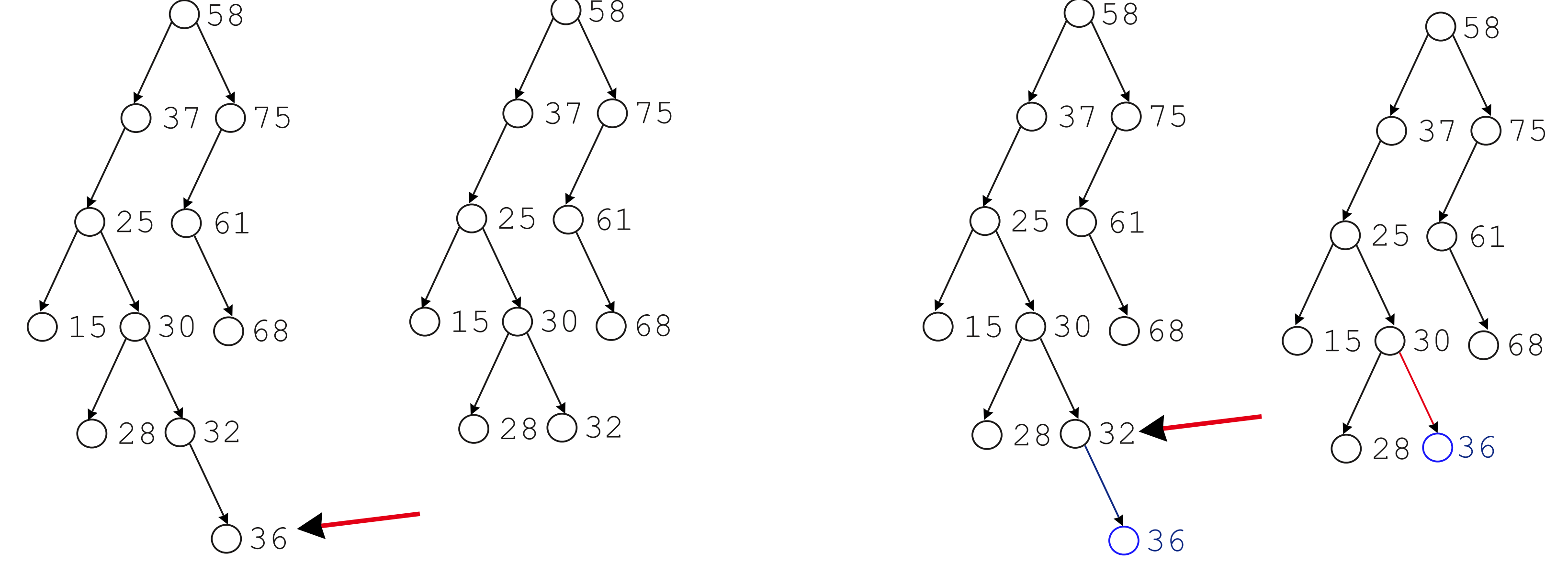
Kirje otsimine kahendpuust (2)

```
void* TreeSearch(NODE *pTree,void *pKey,
                 int(*pCompare)(const void *,const void *))
{ // otsimine kahendpuust ilma rekursioonita
 NODE *p;
 int i;
 if (!pTree || !pKey)
   return 0;
 for(p = pTree; p; )
 {
   if ((i = (pCompare)(pKey, p->pRecord)) < 0)
      p = p->pLeft;// liigume vasakule
   else if (i > 0)
      p = p->pRight;// liigume paremale
   else
      return p->pRecord;// leidsime
 }
 return 0;// ei ole olemas
}
```

Kirje lisamine kahendpuusse

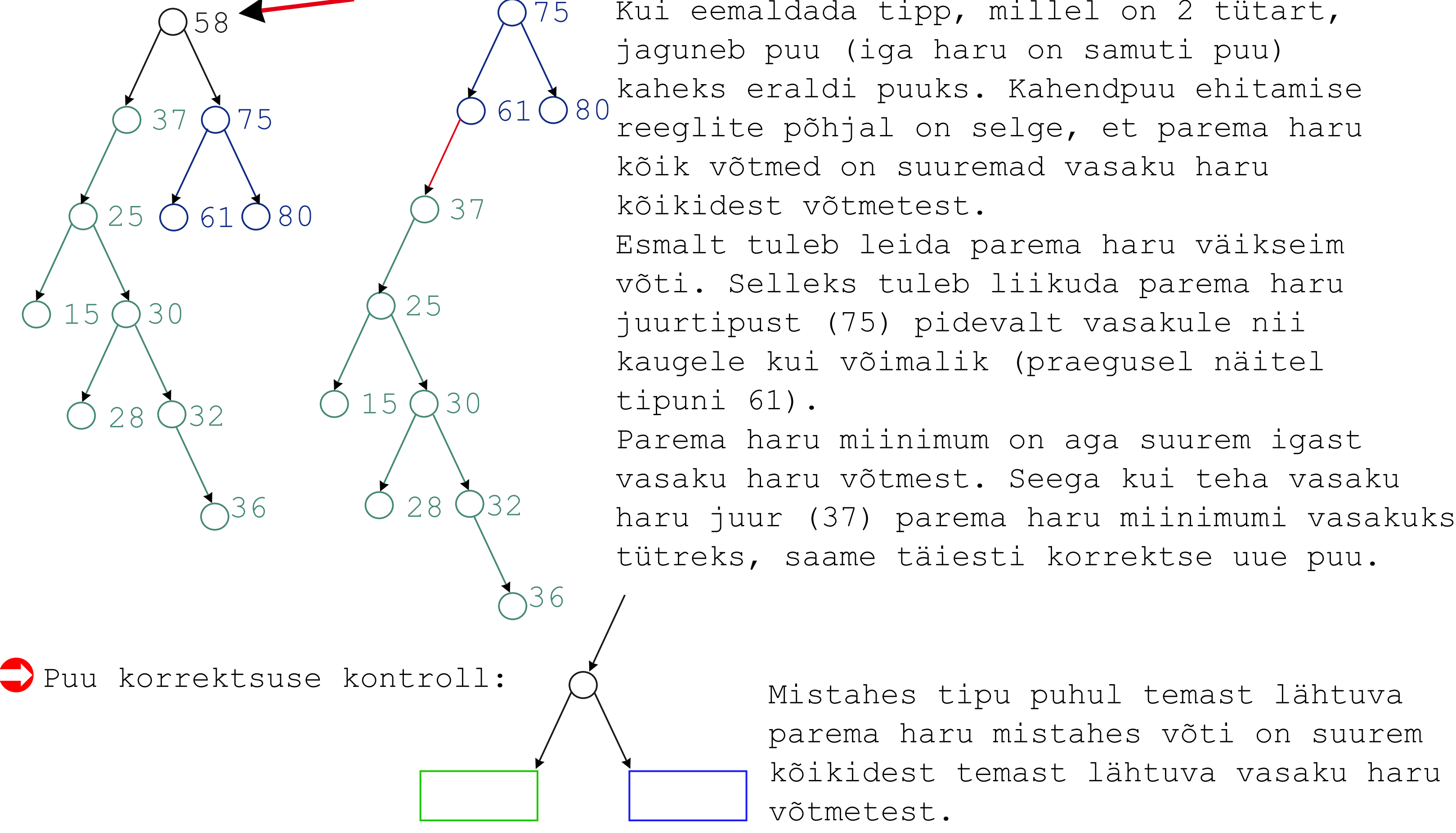
```
NODE *InsertNode(NODE *pTree, void *pNewRecord,
                 int(*pCompare)(const void *, const void *)) {
 NODE *pNew = (NODE *)malloc(sizeof(NODE));// uus tipp
 pNew->pRecord = pNewRecord;
 pNew->pLeft = pNew->pRight = 0;
 if(!pTree)
   return pNew;// puu oli tühi
 for(NODE *p = pTree; 1; ) {
   if ((pCompare)(pNewRecord, p->pRecord) < 0) {
      if (!p->pLeft) {// leidsime tühja koha
         p->pLeft = pNew;
         return Tree;
      }
      else
         p = p->pLeft;// liigume vasakule
   }
   else {
      if(!p->pRight) { // leidsime tühja koha
         p->pRight = pNew;
         return pTree;
      }
      else
         p = p->pRight;// liigume paremale
      }
   }
}
```

Kirje eemaldamine kahendpuust (1)



Lehe eemaldamine ei valmista mingeid probleeme.
Samuti ei tekita erilisi probleeme sellise tipu eemaldamine, millel on ainult üks tütar.

Kirje eemaldamine kahendpuust (2)



Kui eemaldada tipp, millel on 2 tütart, jaguneb puu (iga haru on samuti puu) kaheks eraldi puuks. Kahendpuu ehitamise reeglite põhjal on selge, et parema haru kõik võtmed on suuremad vasaku haru kõikidest võtmetest.
Esmalt tuleb leida parema haru väikseim võti. Selleks tuleb liikuda parema haru juurtipust (75) pidevalt vasakule nii kaugele kui võimalik (praegusel näitel tipuni 61).
Parema haru miinimum on aga suurem igast vasaku haru võtmest. Seega kui teha vasaku haru juur (37) parema haru miinimumi vasakuks tütreks, saame täiesti korrektse uue puu.

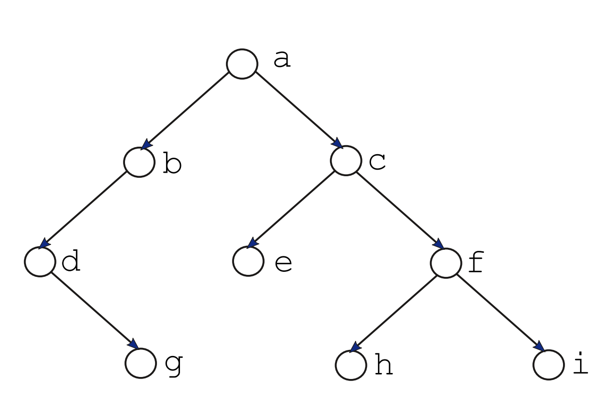
- Puu korrektsuse kontroll:

Mistahes tipu puhul temast lähtuva parema haru mistahes võti on suurem kõikidest temast lähtuva vasaku haru võtmetest.

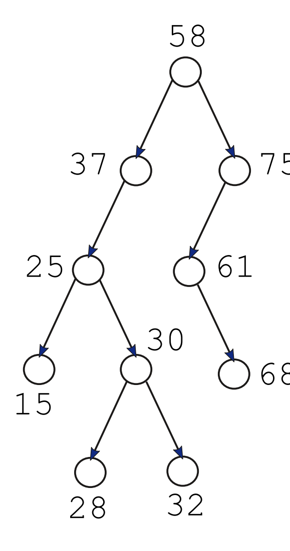
- Miinimum ja maksimum: nende leidmiseks tuleb juurest lähtuvalt liikuda vasakule või paremale seni kuni võimalik.

Puu läbikäik (traversal) (1)

Puu läbikäigu algoritmid (v.a. nivooti) on rekursiivsed . Läbikäigu eesmärgiks on saada puusse paigutatud kirjetest koosnev lineaarne loend



Nivooti: a, b, c, d, e, f, g, h, i
Süviti: g, d, b, e, h, i, f, c, a
(vasak - parem -juur)
Laiuti: a, b, d, g, c, e, f, h, i
(juur - vasak - parem)
Sümmeetriliselt: d, g, b, a, e, c, h, f, i
(vasak - juur -parem)



Nivooti: 58, 37, 75, 25, 61, 15, 30, 68, 28, 32
Süviti: 15, 28, 32, 30, 25, 37, 68, 61, 75, 58
(vasak - parem -juur)
Laiuti: 58, 37, 25, 15, 30, 28, 32, 75, 61, 68
(juur - vasak - parem)
Sümmeetriliselt: 15, 25, 28, 30, 32, 37, 58, 61, 68, 75
(juur - vasak - parem)

Sümmeetriline läbikäik annab sorditud loendi

Puu läbikäik (2)

```
void SymmetricalTraversal(NODE *pTree,void(*pProcess)(NODE *))
{ // rekursiivne funktsioon
 if(pTree)
 {
   SymmetricalTraversal(pTree->pLeft, pProcess);
   (pProcess)(pTree);
   SymmetricalTraversal(pTree->pRight, pProcess);
 }
}
```

pProcess on viit funktsioonile, mille sisendparameetriks on viit puu tipule, väljund aga puudub. Kui me soovime välja trükkida puusse paigutatud isikute nimekirja tähestikulises järjekorras, siis sobiv funktsioon oleks:

```
void ProcessNode(NODE *pNode)
{
 printf("%s\n", ((PERSON *)pNode->pRecord)->pName);
}

void destroy(NODE *pTree)
{ // kahendpuu hävitamine
 if (pTree)
 { // läbikäik süviti meetodil
   destroy(pTree->pLeft);
   destroy(pTree->pRight);
   free(pTree); // juure kustutamine
 }
}
```

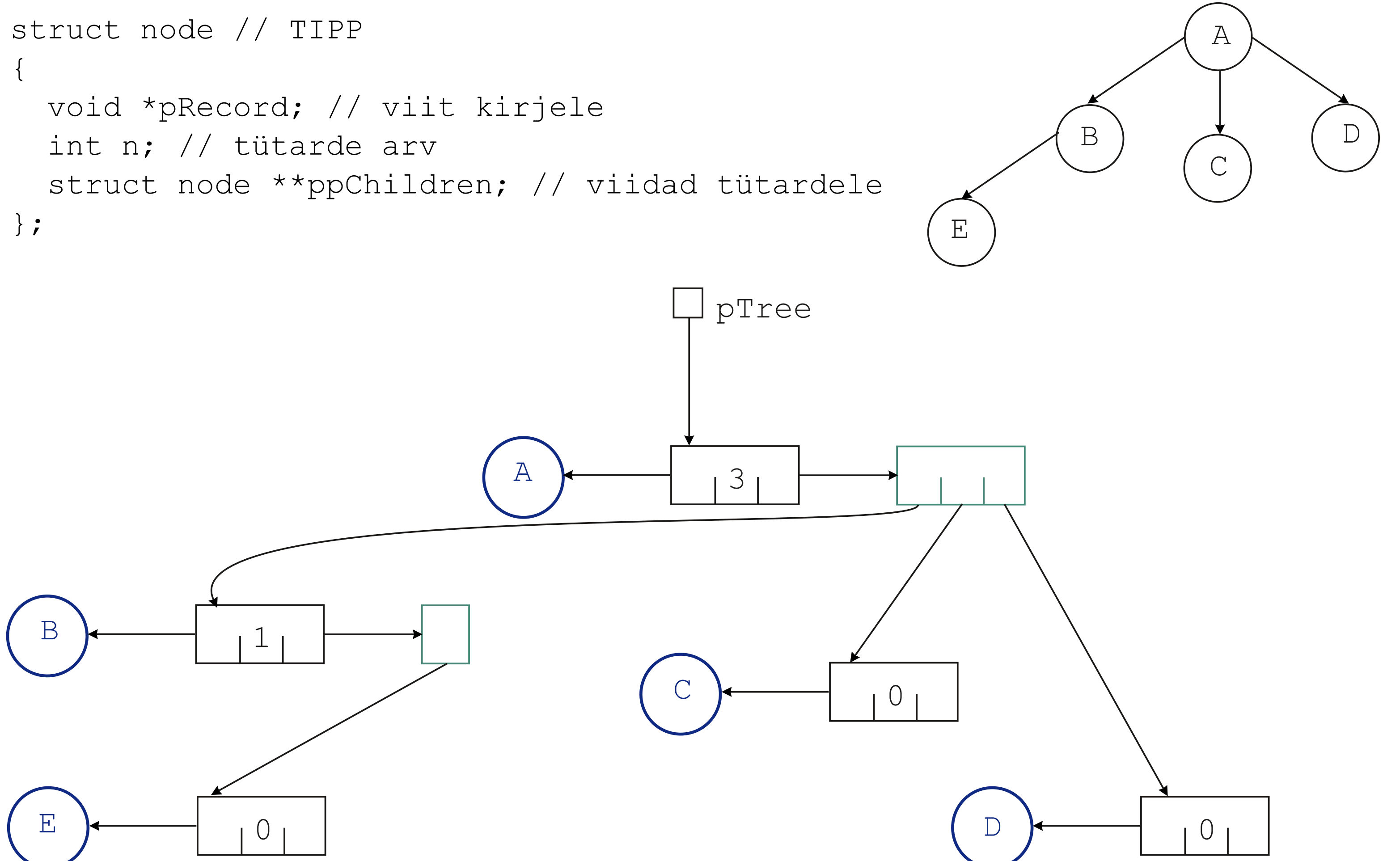
Puu läbikäik (3)

```
void SymmetricalTraversal(NODE *pTree,void(*pProcess)(NODE *))
{
 STACK *pStack = 0;
 NODE *p1 = pTree, *p2;
 if (!pTree)
   return;
 do
 {
   while(p1)
   {
      pStack = Push(pStack, p1);
      p1 = p1->pLeft;
   }
   pStack = Pop(pStack, (void**)&p2);
   (pProcess)(p2);
   p1 = p2->pRight;
 }
 while(!(!pStack && !p1));
}
```

Rekursiivne algoritm on realiseeritud stack-i kasutava mitterekursiivse funktsiooniga

Paljuharuline puu andmestruktuurina (1)

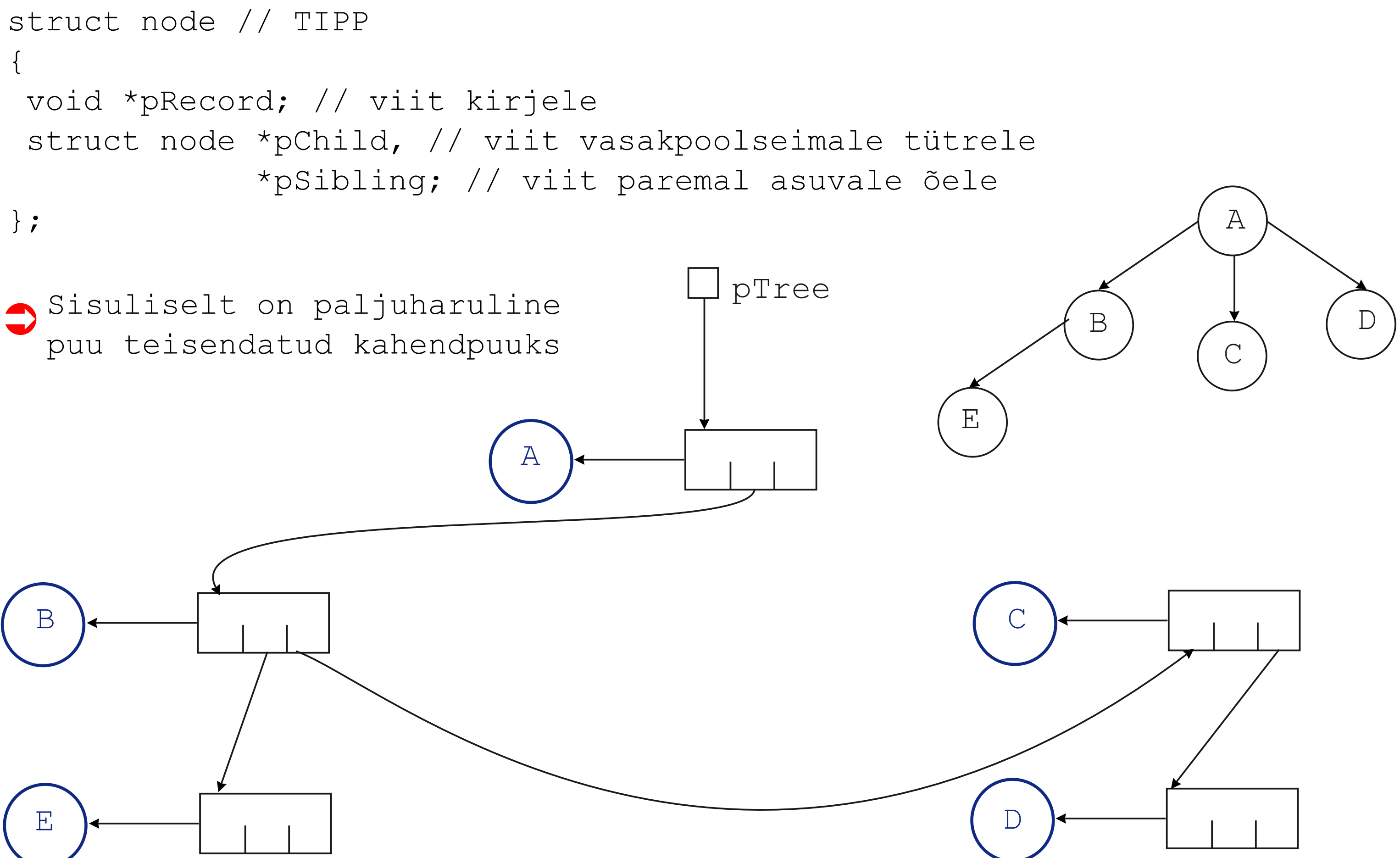
```
struct node // TIPP
{
  void *pRecord; // viit kirjele
  int n; // tütarde arv
  struct node **ppChildren; // viidad tütardele
};
```



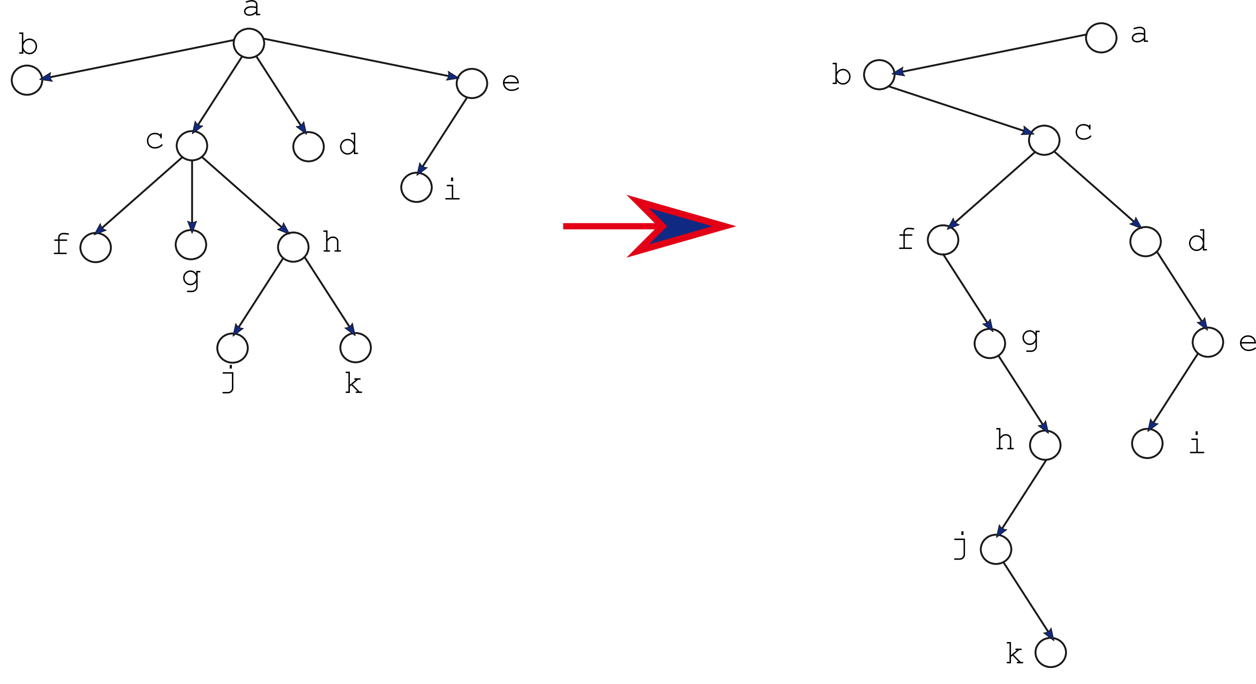
Paljuharuline puu andmestruktuurina (2)

```
struct node // TIPP
{
 void *pRecord; // viit kirjele
 struct node *pChild, // viit vasakpoolseimale tütrele
             *pSibling; // viit paremal asuvale õele
};
```

- Sisuliselt on paljuharuline puu teisendatud kahendpuuks



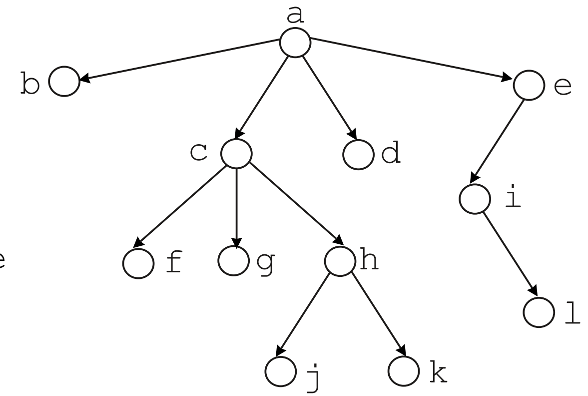
Paljuharuline puu andmestruktuurina (3)



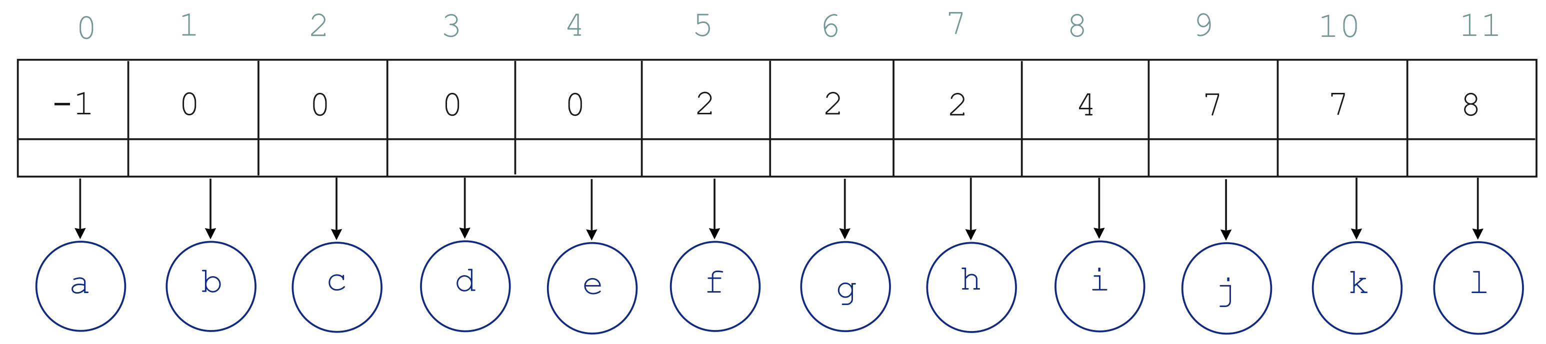
Näide paljuharulise puu teisendamisest kahendpuuks

Paljuharuline puu andmestruktuurina (4)

```
struct node // TIPP
{
 void *pRecord; // viit kirjele
 int parent; // ematipu indeks
};
```

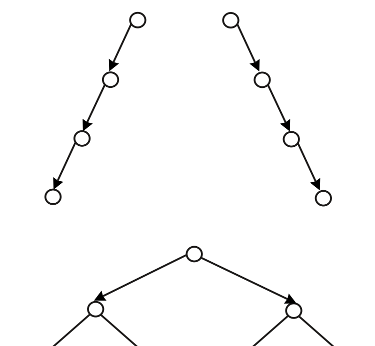


➡ Selline lahendus ei näita enam õdede järjekorda. Samuti puudub võimalus liikuda juurest allapoole.

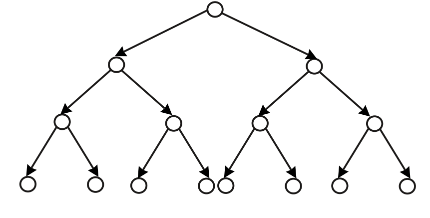


Kahendpuust otsimise parameetrid

Kui andmed saabuvad sorditud järjekorras, taandub puu tavaliseks loendiks. Sel juhul puu kõrgus (s.t. nivoode arv) $h(n) = n$ ja $T(n) \leq n$.



Loendiks taandunud puu vastandiks on absoluutselt tasakaalus puu, mis on võimalik vaid siis kui leidub k nii, et



$n = 2^k - 1$

$h(n) = \log_2(n+1)$

$T(n) \leq \log_2(n+1)$

Juhuslikus järjekorras saabunud andmete põhjal genereeritud puu korral

$\log_2(n+1) < h(n) \leq n$ kui $n \neq 2^k - 1$

$\log_2(n+1) \leq h(n) \leq n$ kui $n = 2^k - 1$

Saab tõestada, et keskmine üle kõigi võimalike konfiguratsioonide on

$T(n) \approx 1.3863 \log_2(n)$

Kuigi keskmine võrdluste arv on logaritmilise iseloomuga, ütleb see vähe, sest too keskmine on võetud üle väga laia diapasooni. Intuitiivselt on selge, et mida ühtlasema pikkusega on harud (s.t. mida enam on puu tasakaalus), seda väiksem on puu kõrgus ning järelikult ka otsitava leidmiseks vajalik võrdluste arv. Praktikas aga kipuvad puu harud tihti välja venima (vt. eemaldamist), kõrgus kasvab ja puust otsimise iseloom muutub pigem lineaarseks.

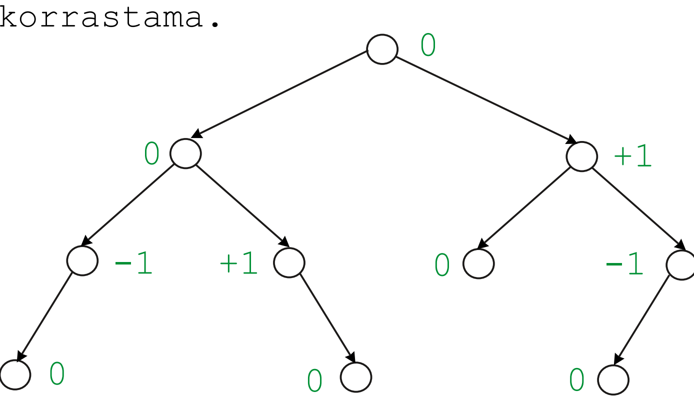
AVL puud (1)

Puu tasakaalustamise probleem: hoolimata sellest, et andmed saabuvad juhuslikus järjekorras, tuleb püüda ehitada puu nii, et ta oleks võimalikult hästi tasakaalus, s.t. võimalikult madal ja enam-vähem ühepikkuste harudega. See tagab kiirema otsimise.

Praktikas tähendab see, et pärast iga uue tipu lisamist tuleb kontrollida puu tasakaalu ja kui vaja, siis rakendada mingeid meetmeid puu konfiguratsiooni muutmiseks.

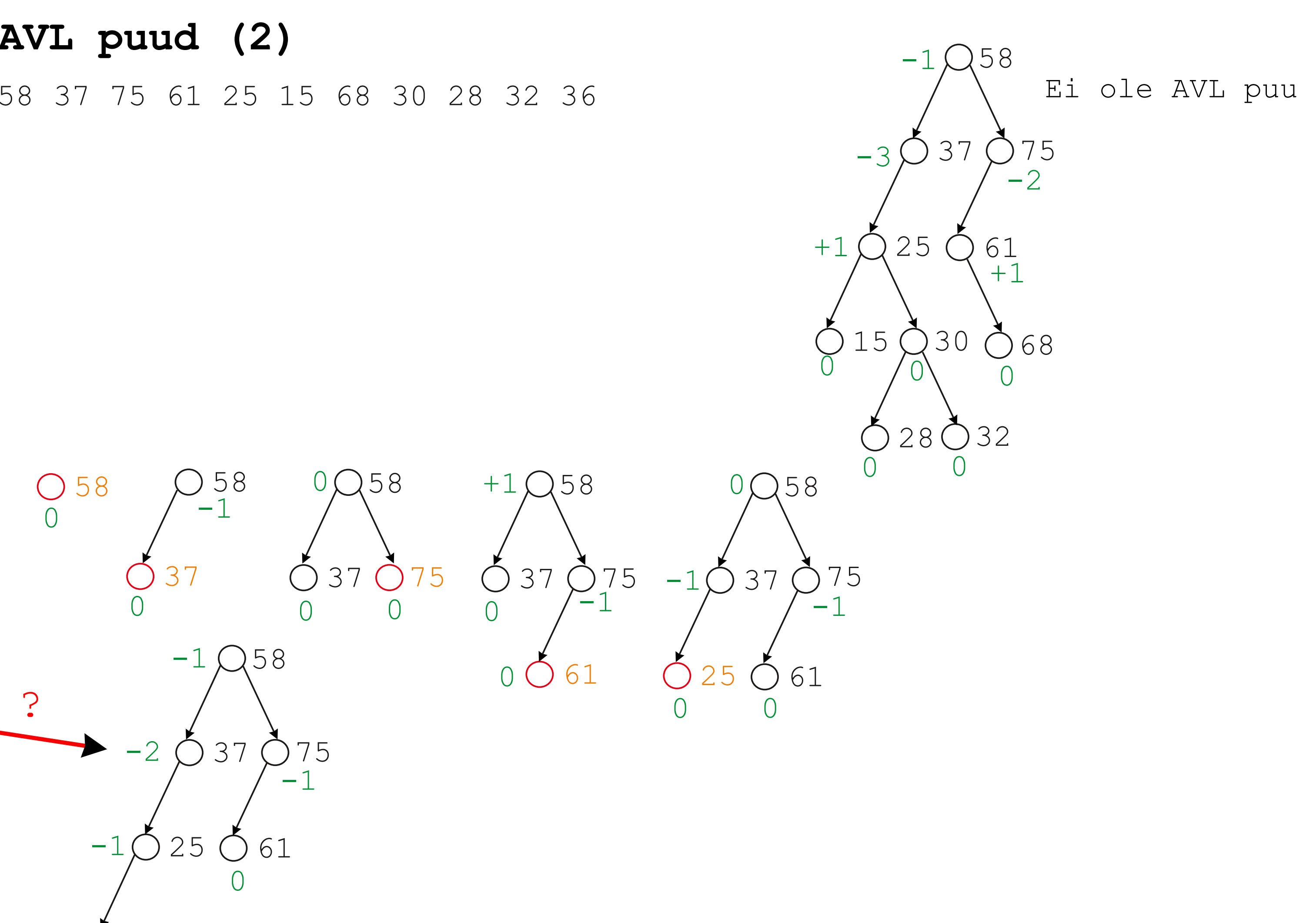
Vanimaks lahendiks selliste iseennast tasakaalustavate puude hulgas on AVL puu (Adelson-Velski ja Landis, Kiiev, 1962).

Puu igal tipul on tasakaalutegur: temast lähtuva parempoolse haru kõrgus miinus vasakpoolse haru kõrgus. Tasakaalutegur võib olla 0, +1 ja -1. Kui pärast uue tipu lisamist kusagil puus mõne tipu tasakaalutegur muutub +2-ks või -2-ks, tuleb asuda puud korrastama.



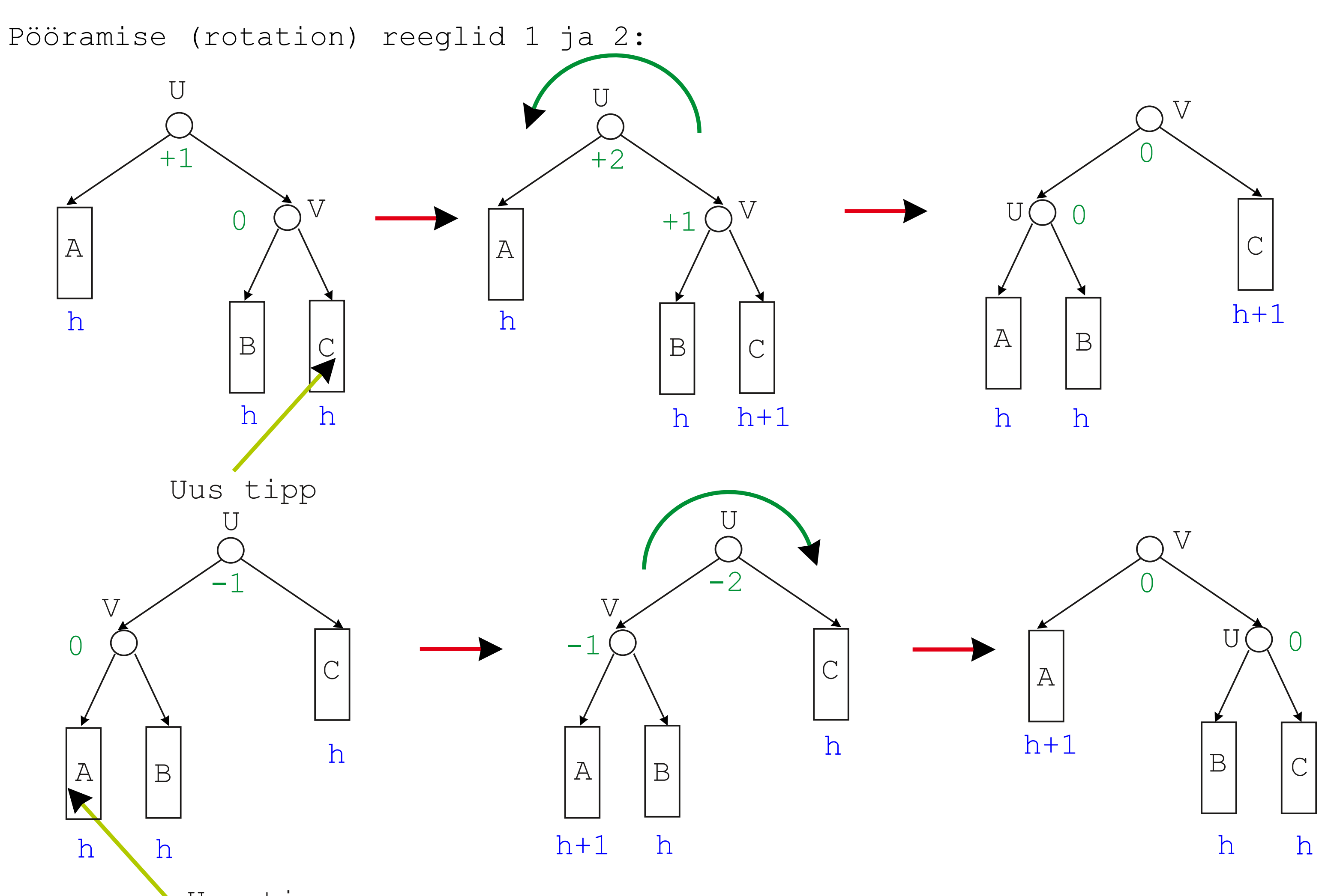
AVL puud (2)

58 37 75 61 25 15 68 30 28 32 36



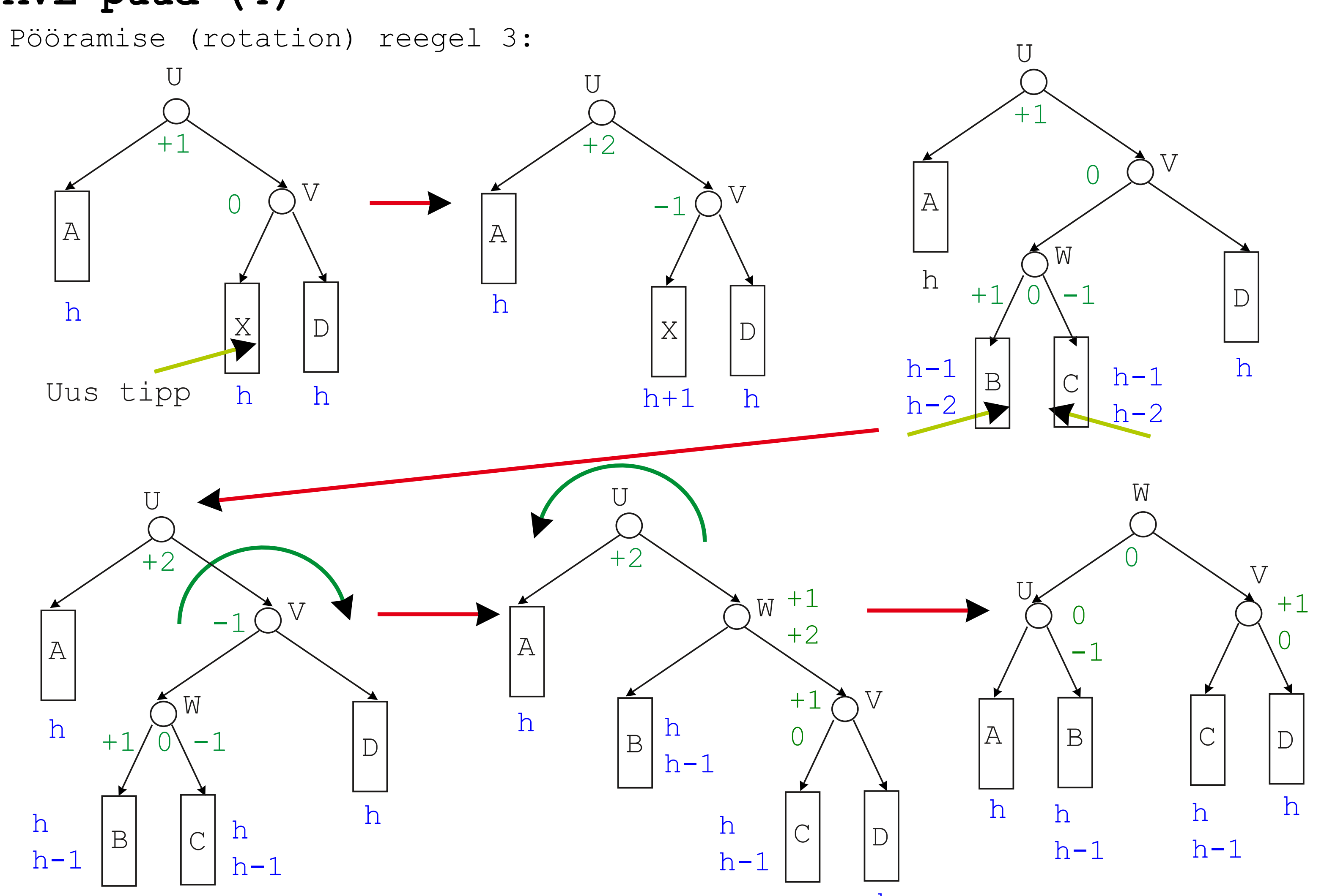
AVL puud (3)

Pööramise (rotation) reeglid 1 ja 2:

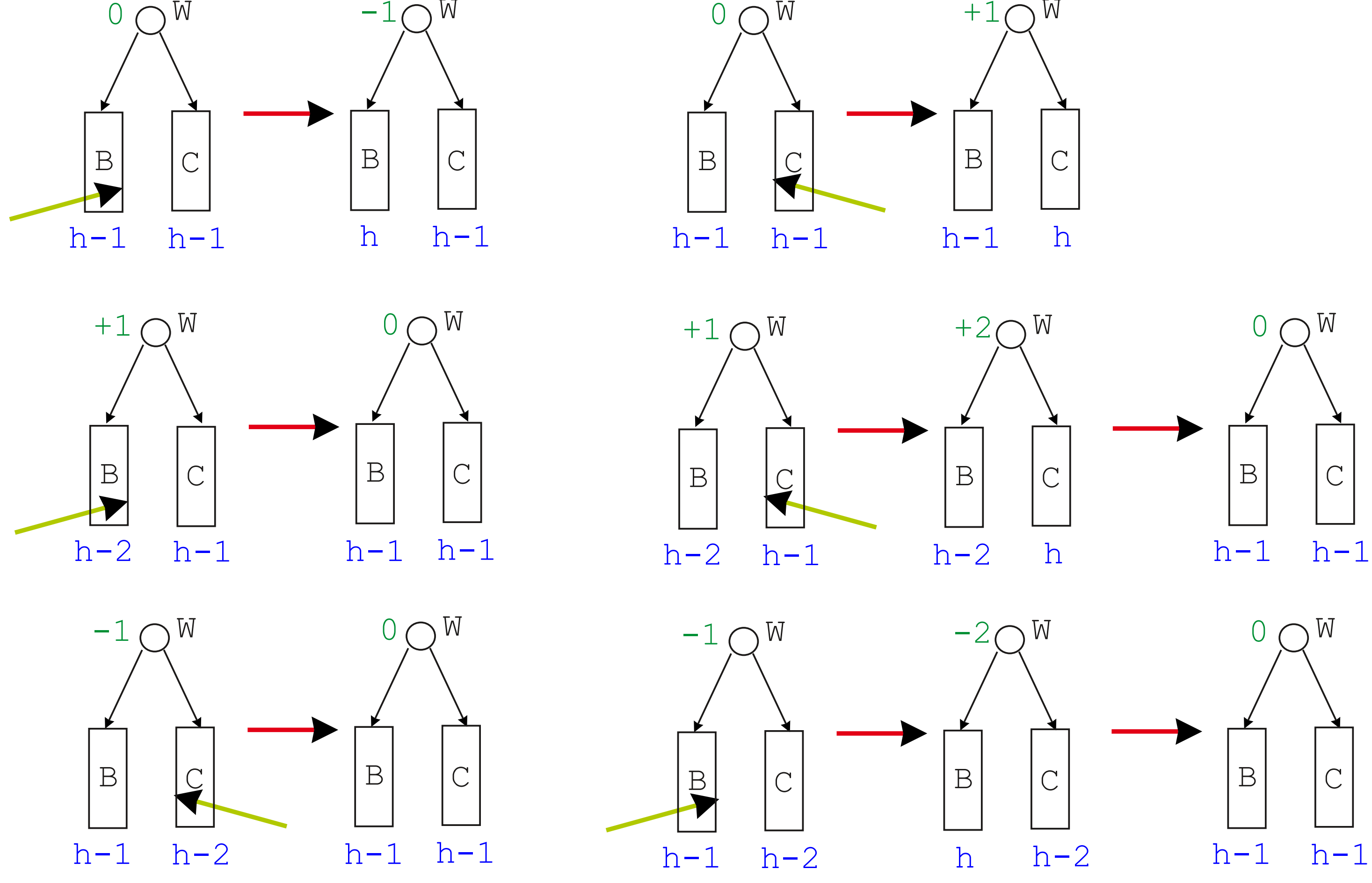


AVL puud (4)

Pööramise (rotation) reegel 3:

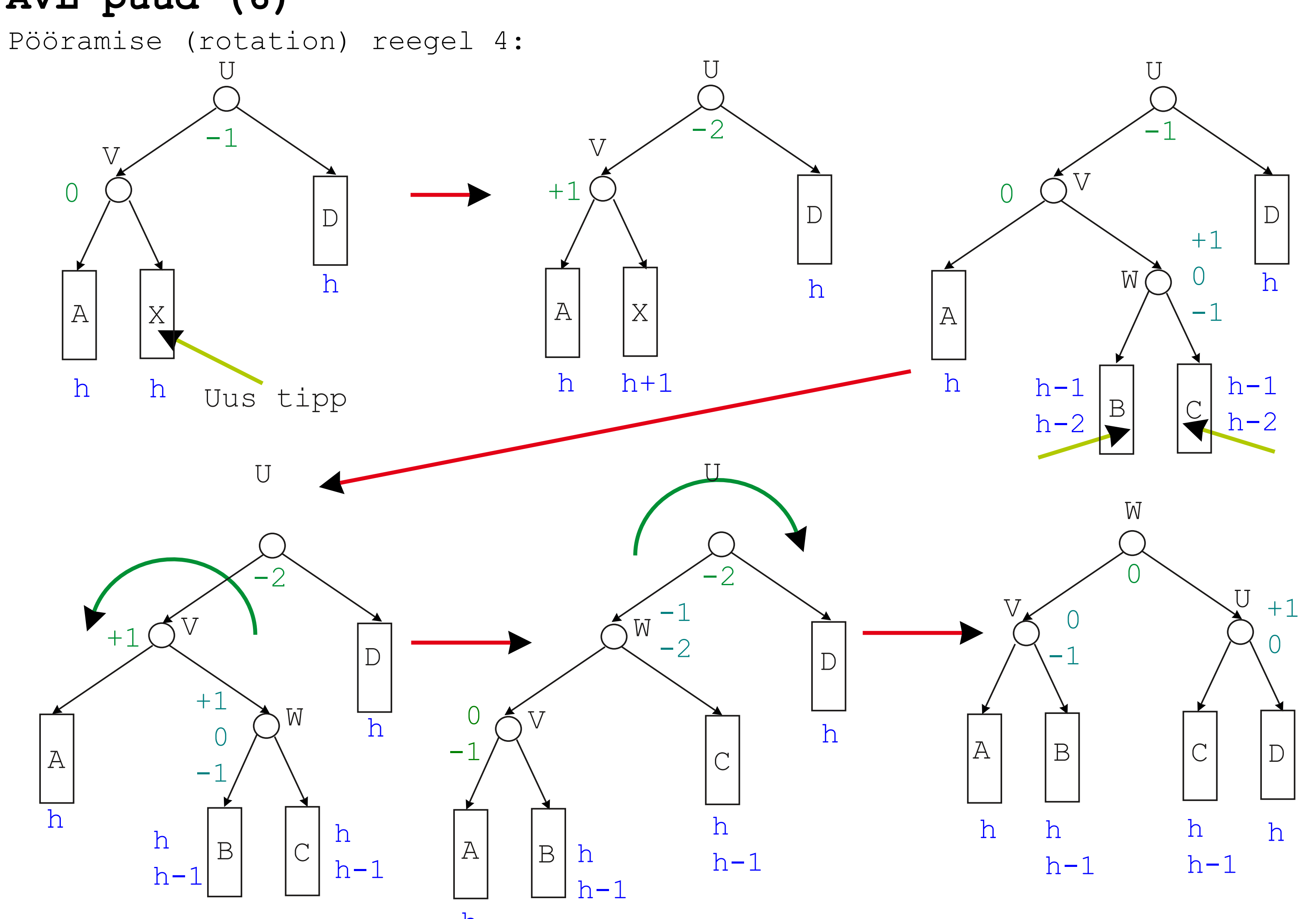


AVL puud (5)



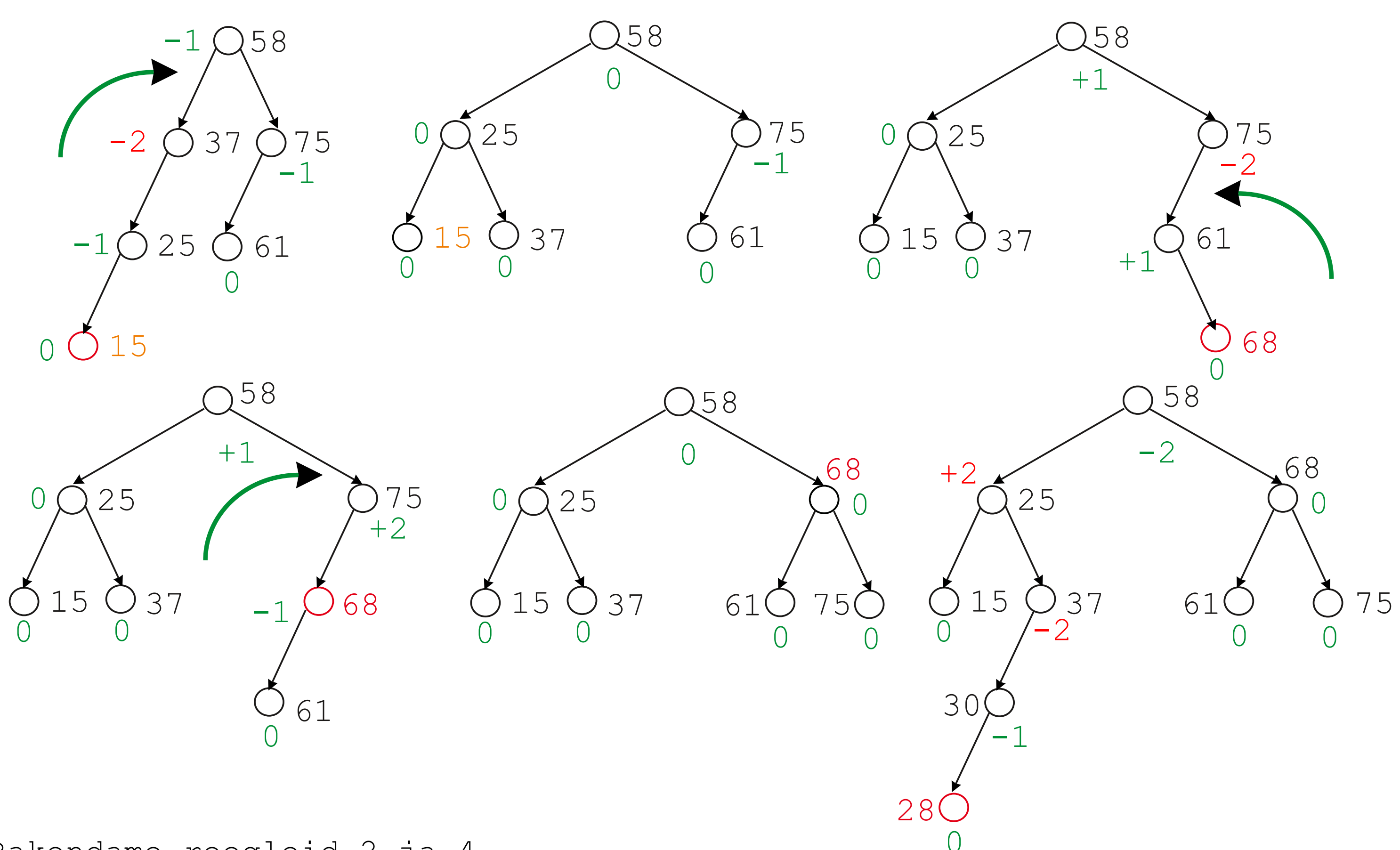
AVL puud (6)

Pööramise (rotation) reegel 4:



AVL puud (7)

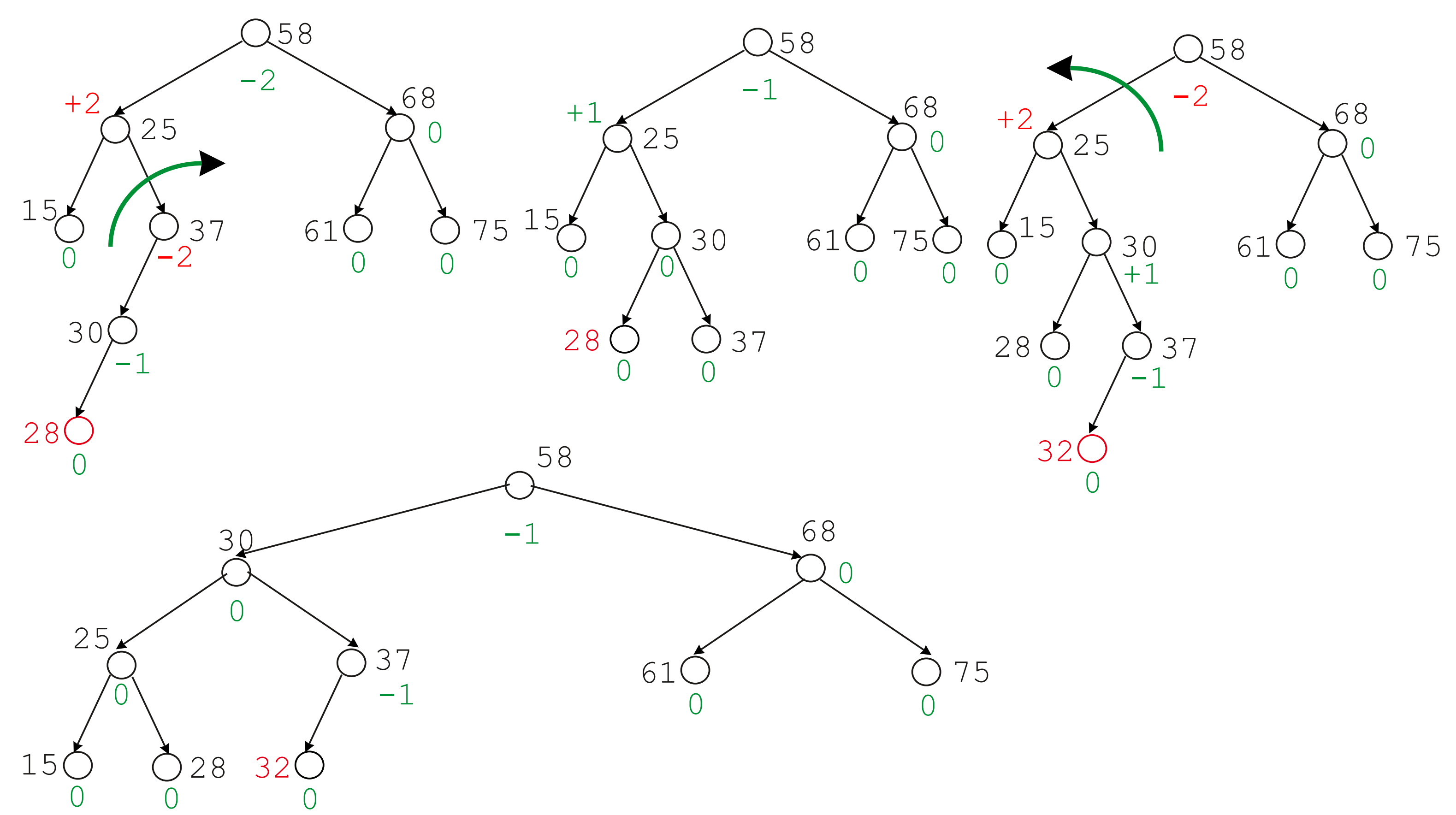
58 37 75 61 25 15 68 30 28 32 36



Rakendame reegleid 2 ja 4

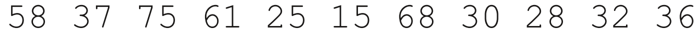
AVL puud (8)

58 37 75 61 25 15 68 30 28 32 36



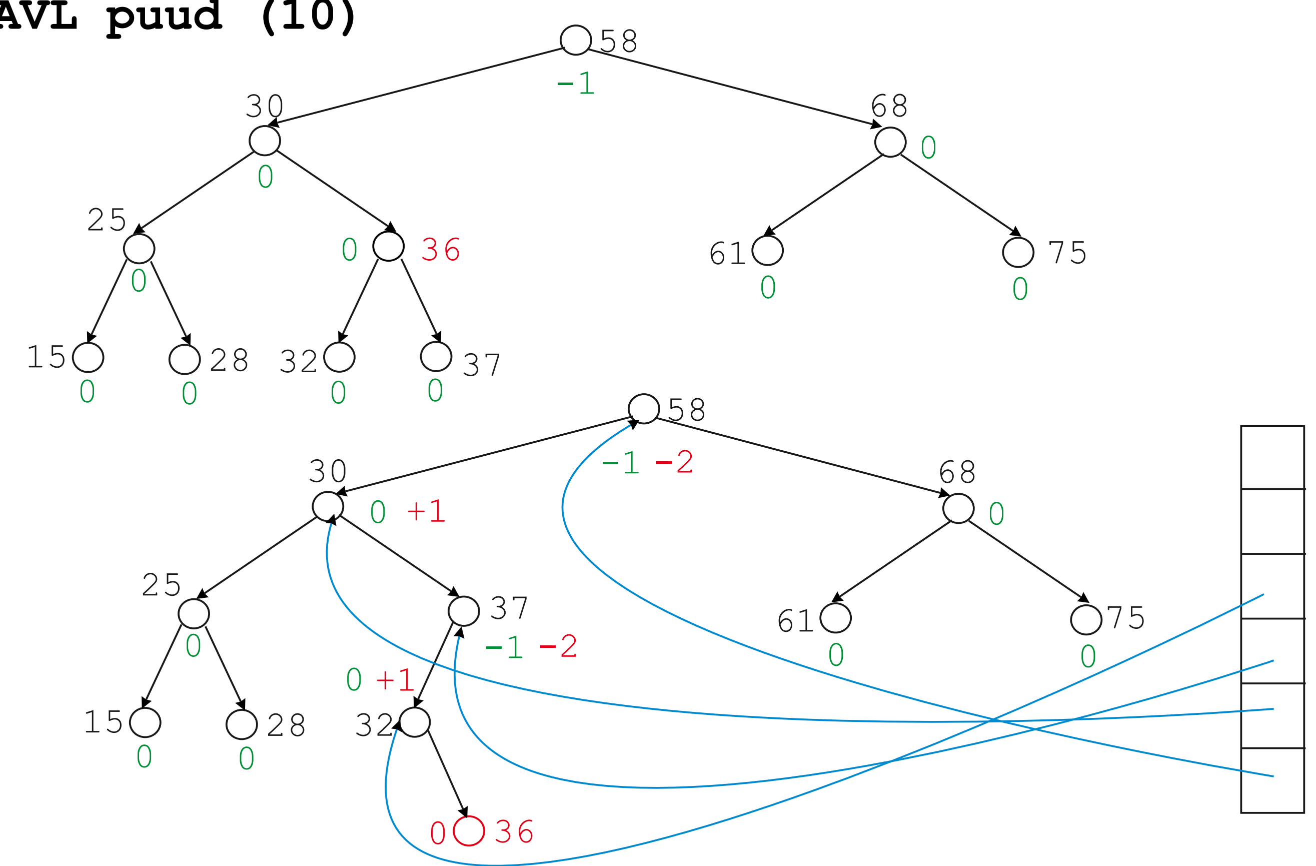
Kasutame reegleid 2 ja 1

AVL puud (9)



Kasutame reeglit 4

AVL puud (10)



Allapoole liikude pushi-ime kõikide läbitud tippude aadressid stack-i. Kui tipust suundume paremale, suurendame tema tasakaalutegurit ühe võrra ja kui vasakule, siis vähendame ühe võrra. Kui tipp on paigas, liigume stack-ist pop-ides tagasi kuni tipuni, mille tasakaalutegur on 2 või -2. Seal asume rakendama pööramise reegleid.

AVL puud (11)

AVL puusse lisamise ja sealt eemaldamise kohta leiab materjali lehtedelt:

https://www.geeksforgeeks.org/avl-tree-set-1-insertion/

https://www.geeksforgeeks.org/avl-tree-set-2-deletion/

Samas on toodud ka vajalike funktsioonide koodid C-s, Java-s ja Python-is.

AVL puude kohta on tõestatud, et:

$T(n) \le 1.4404\log_2(n+2) - 0.328$

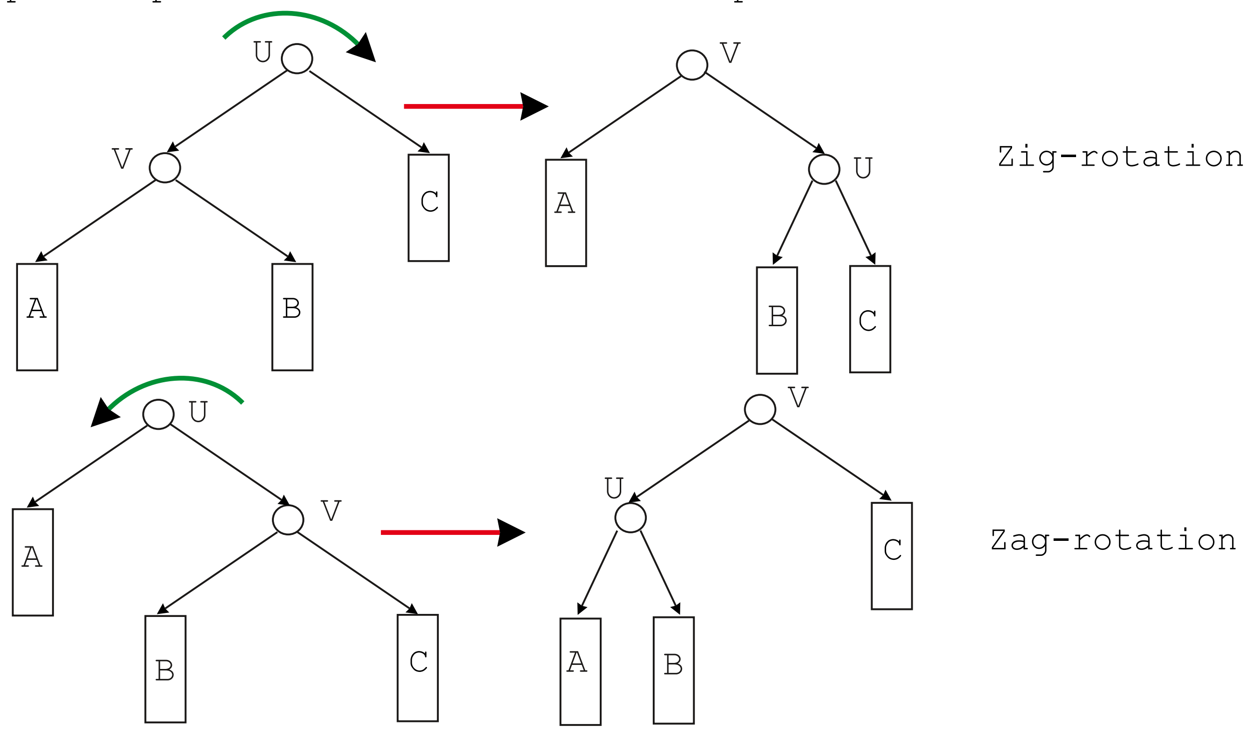
Kui $n > 500$, siis keskmine on $T(n) \approx \log_2(n) - 0.86$

Harali puud (splay trees) (1)

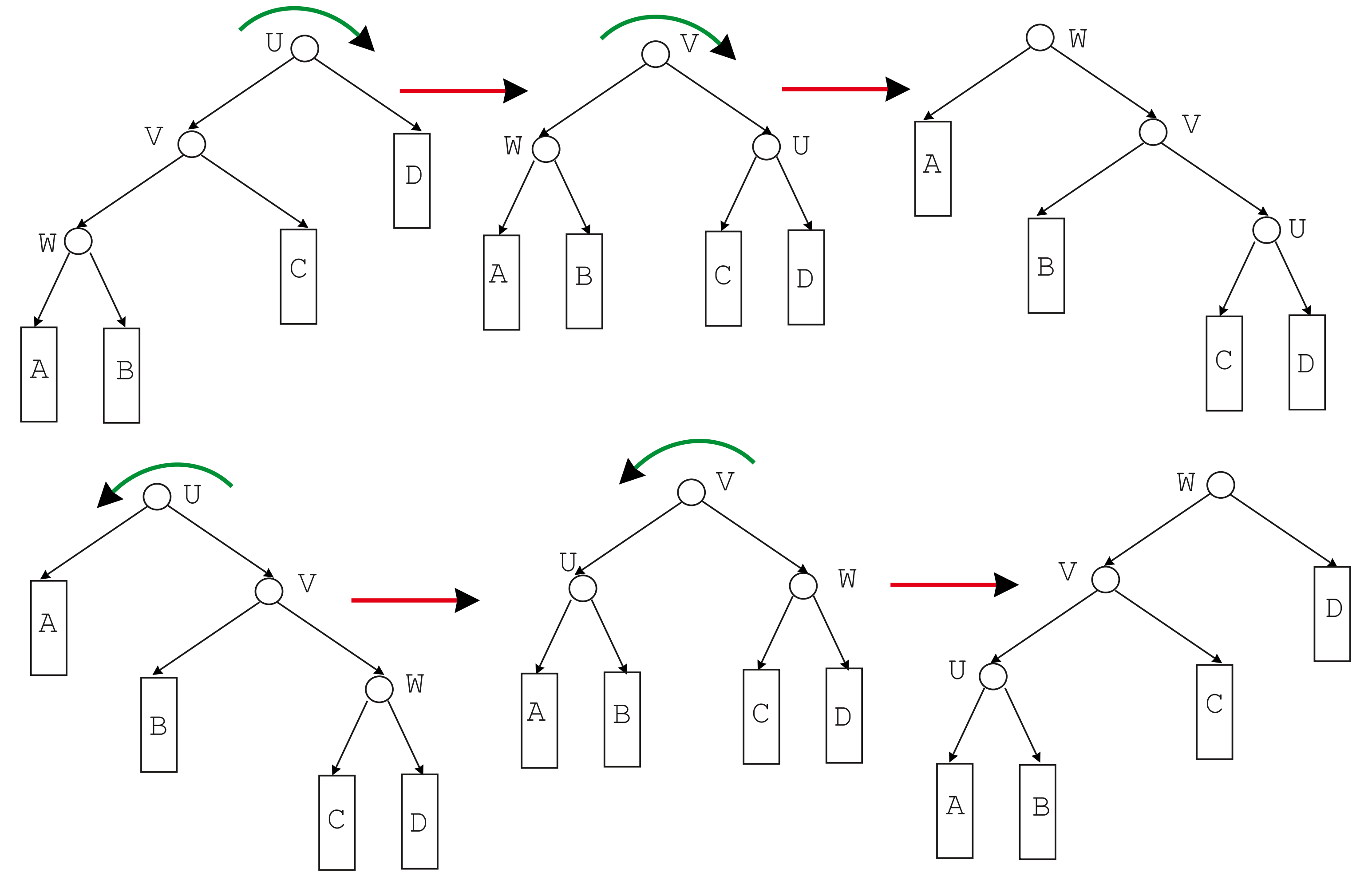
On selge, et kui kirje on puu juurele lähemal, siis tema ülesleidmine võtab vähem aega.

Iseorganiseeruva järjestikuse otsimise puhul pärast iga päringu täitmist toodi leitud kirje loendi etteotsa. Harali puud on samuti iseorganiseeruvad: leitud kirje tuuakse puu juureks. Veel enam - kui uus kirje on tavalise "väiksem vasakule suurem paremale" reegli alusel oma kohale asetatud, tuuakse ka tema puu juureks.

Tipu ülespoole toomiseks kasutatakse pööramist:



Harali puud (2)



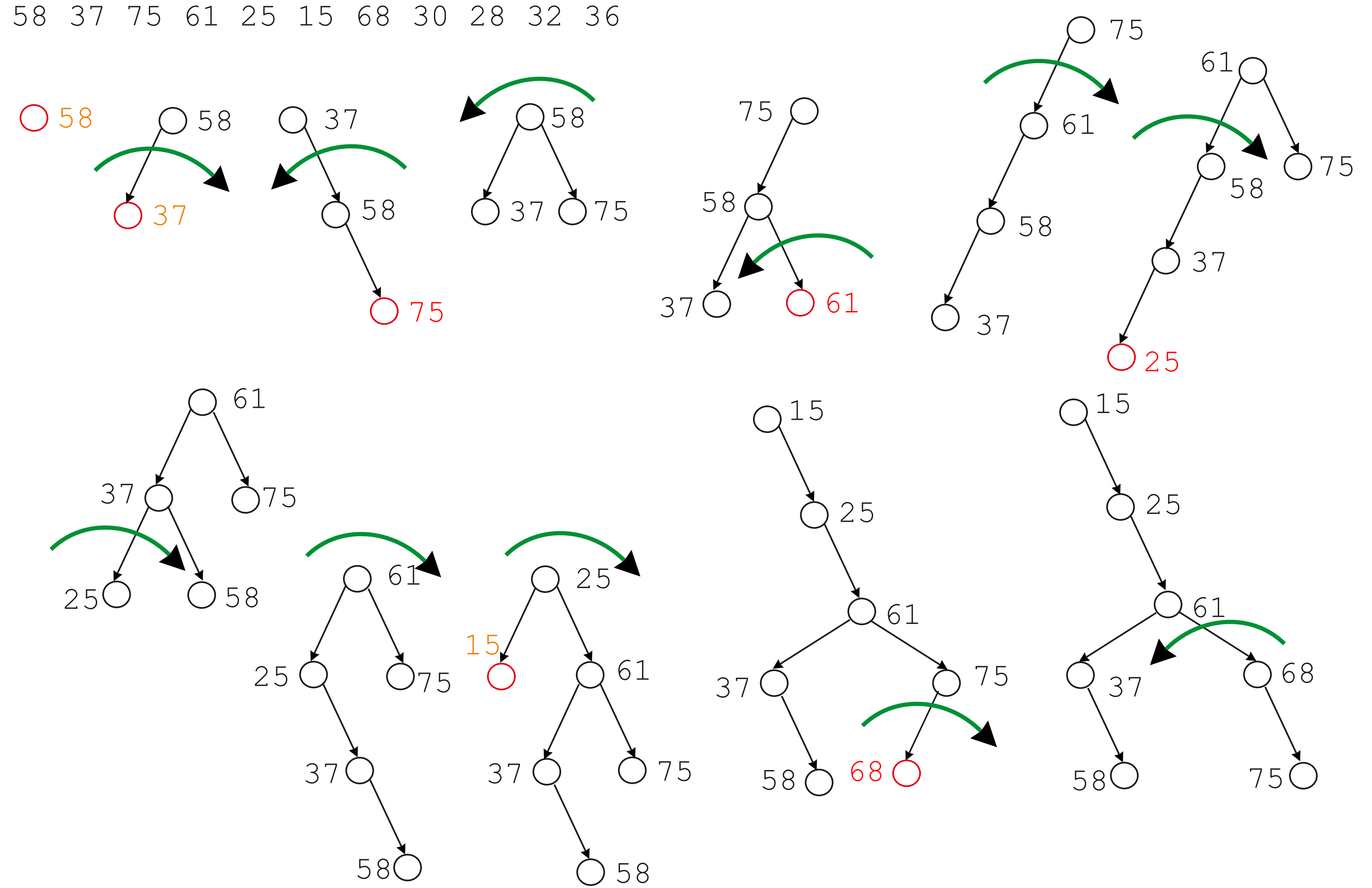
Zig-zig & Zag-zag rotations

Harali puud (3)

Zig-zag & Zag-zig rotations

Harali puud (4)

58 37 75 61 25 15 68 30 28 32 36

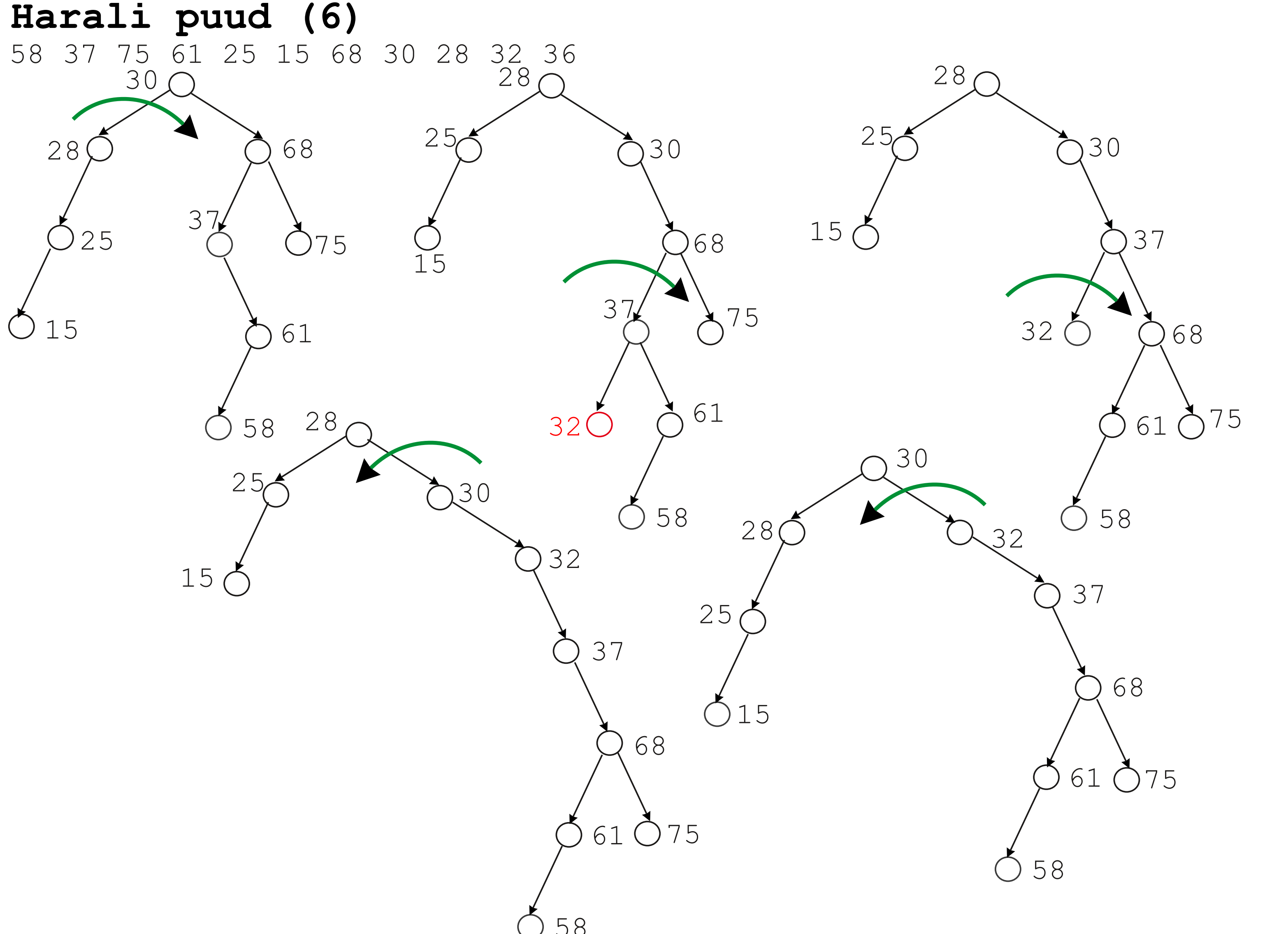


Harali puud (5)

58 37 75 61 25 15 68 30 28 32 36

Harali puud (6)

58 37 75 61 25 15 68 30 28 32 36



Harali puud (7)

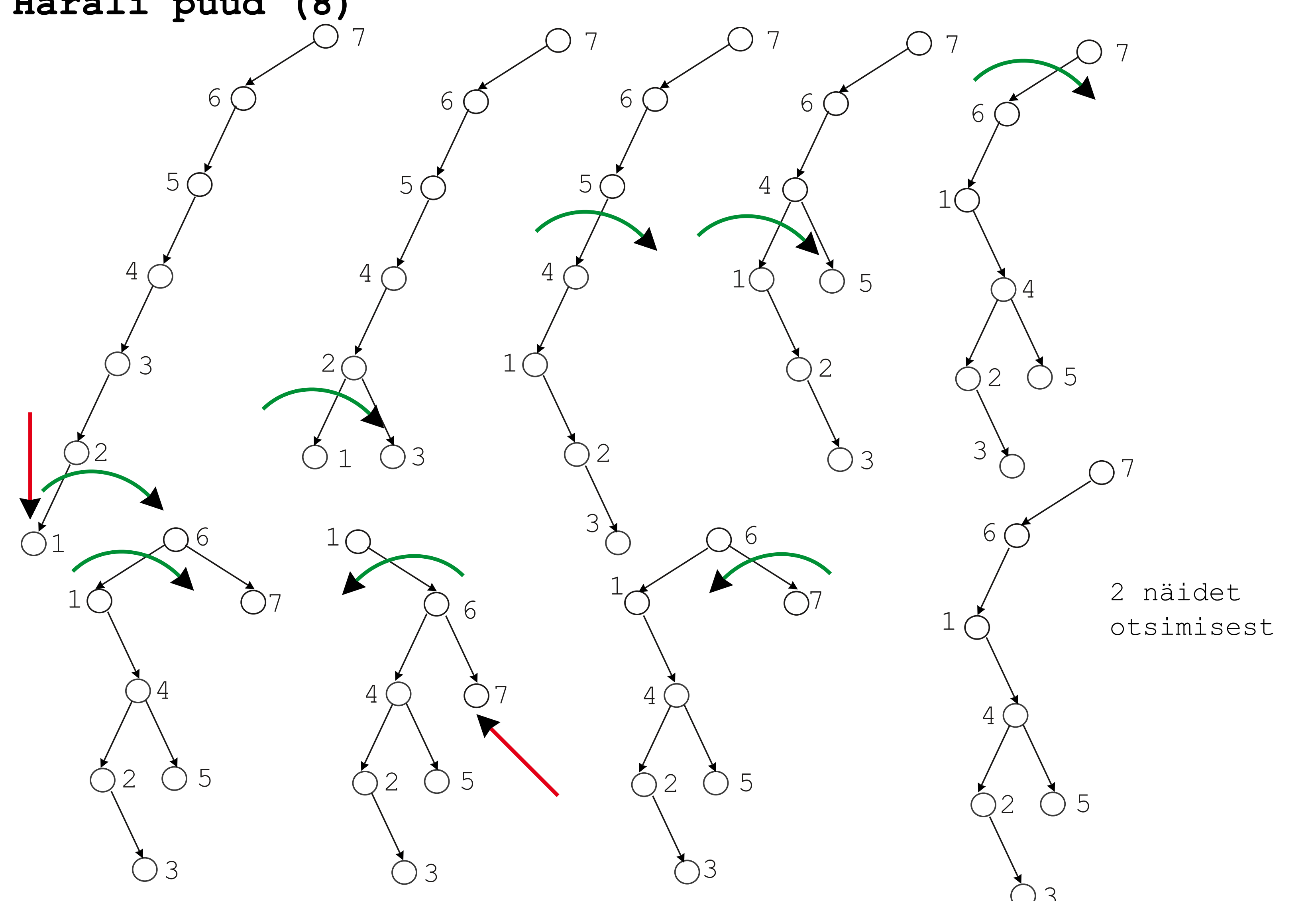
58 37 75 61 25 15 68 30 28 32 36

32
30
37
28
36
68
25
61
75
15
58

32
30
36
28
37
25
68
15
61
75
58

36
32
37
68
30
61
75
28
58
25
15

Harali puud (8)



Harali puud (9)

Harali puust kustutamine toimub samamoodi nagu tavalisest puust.
Kuigi see tundub alguses imelikuna, on harali puuga seotud operatsioonide programmina realiseerimine lihtsam kui AVL-puudega töötamisel. C-keelse koodi võib leida:
https://www.geeksforgeeks.org/splay-tree-set-1-insert/
https://www.geeksforgeeks.org/splay-tree-set-2-insert-delete/
Otsimine harali puudest on logaritmilise iseloomuga protsess.
Harali puud on praktikas (sealhulgas ka operatsioonisüsteemides) sageli kasutuses.

Operatsioonid failidega (1)

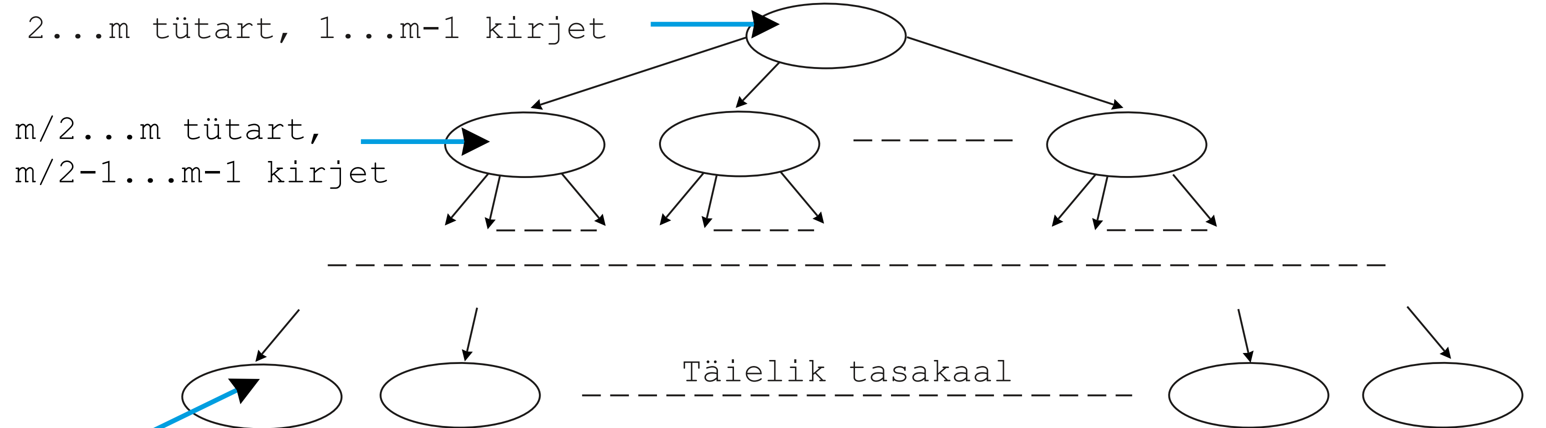
```
#include "stdio.h"
FILE *pfile;
char *filename="c:\\andmed\\andmebaas.bin";
char buffer[sizeof (RECORD)];
int i,m;
pfile=fopen(filename,”rb+”); // avame
// lähemalt vt. http://www.cplusplus.com/reference/cstdio/fopen/
if (!pfile)
{
   ........... // faili ei ole
}
else for (i=0; i<100; i++)
{ // loeme 100 kirjet
   m=fread(buffer,sizeof(RECORD),1,pfile);
   if (m<sizeof(RECORD))
   {
      ........ // vead
   }
   else
   {
      ......... // töötlus
   }
}
fclose(pfile); // sulgeme
```

Operatsioonid failidega (2)

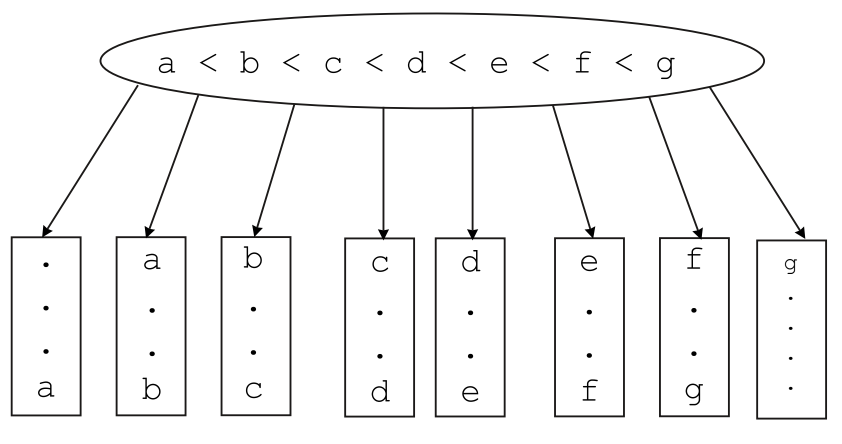
Kuna välisseadmetelt lugemine ja neile kirjutamine nõuab aega, ei ole mõistlik seda teha baithaaval. Seetõttu on lugemise ja kirjutamise operatsioonid puhverdatud.
Lugemisel loetakse kettalt mällu terve plokk ja paigutatakse see lugemise puhvrisse. fread funktsioon loeb andmeid sellest sisemisest puhvrist ja alles siis kui sealt kõik on välja loetud, pöördub uuesti ketta poole.
Kirjutamisel funktsiooniga fwrite paigutatakse andmed esmalt kirjutamise puhvrisse. Alles siis kui see on täis, kantakse terve plokitäis andmed korraga kettale. Osaliselt täitunud puhvri viivitamatut kettale üleviimist saab teostada funktsiooniga fflush.

Hea ülevaate C failidega töötamise funktsioonidest võib leida lehelt
http://www.it.uc3m.es/abel/as/MMC/L2/FilesDef_en.html

B-puud (1)



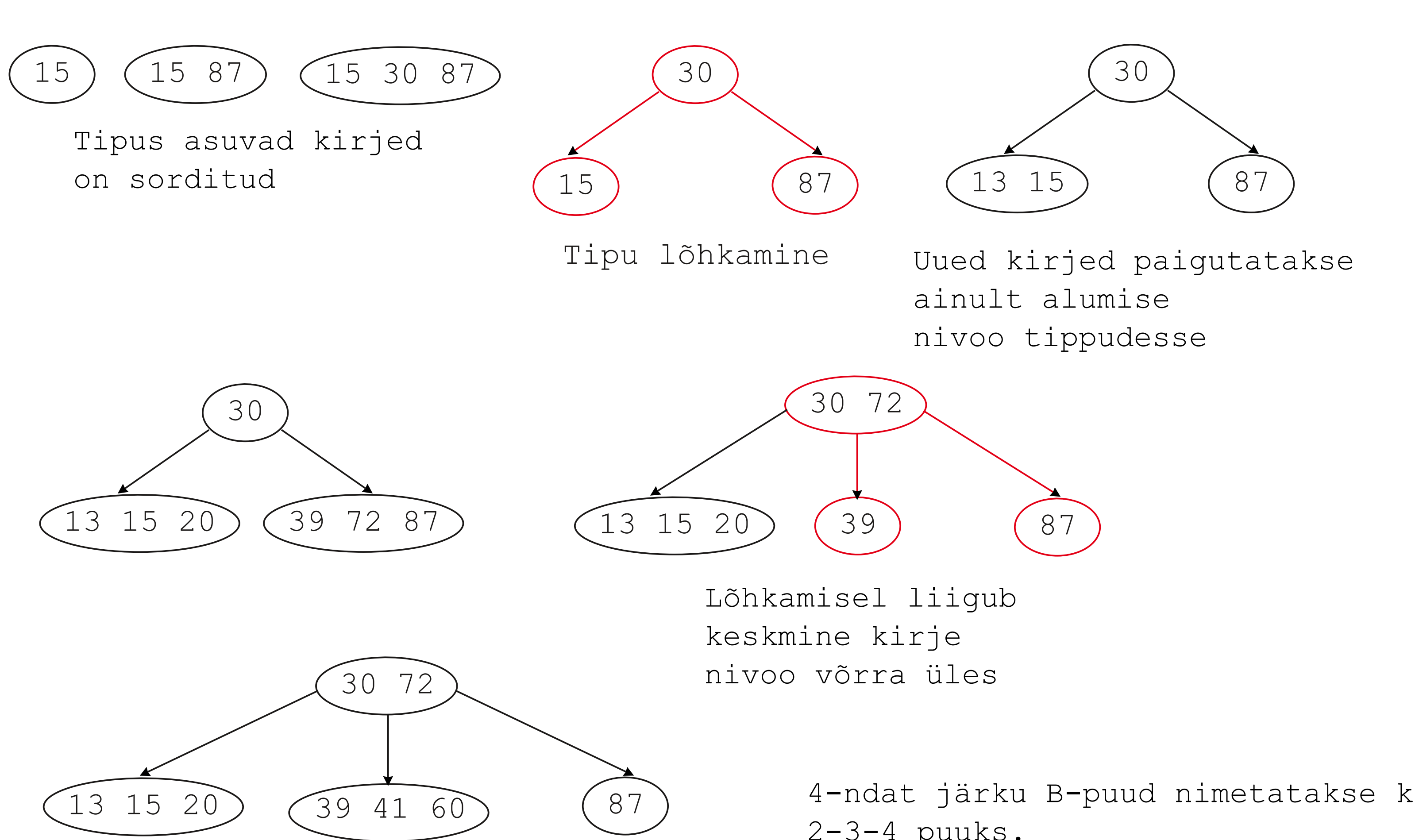
0 tütart,
m/2-1...m-1 kirjet



Et minimiseerida ketta poole pöördumiste arvu, on mõistlik B-puu järk m valida nii, et tipp mahuks tervikuna ühte ketta plokki.
Kirjed tipus on sorteeritud.

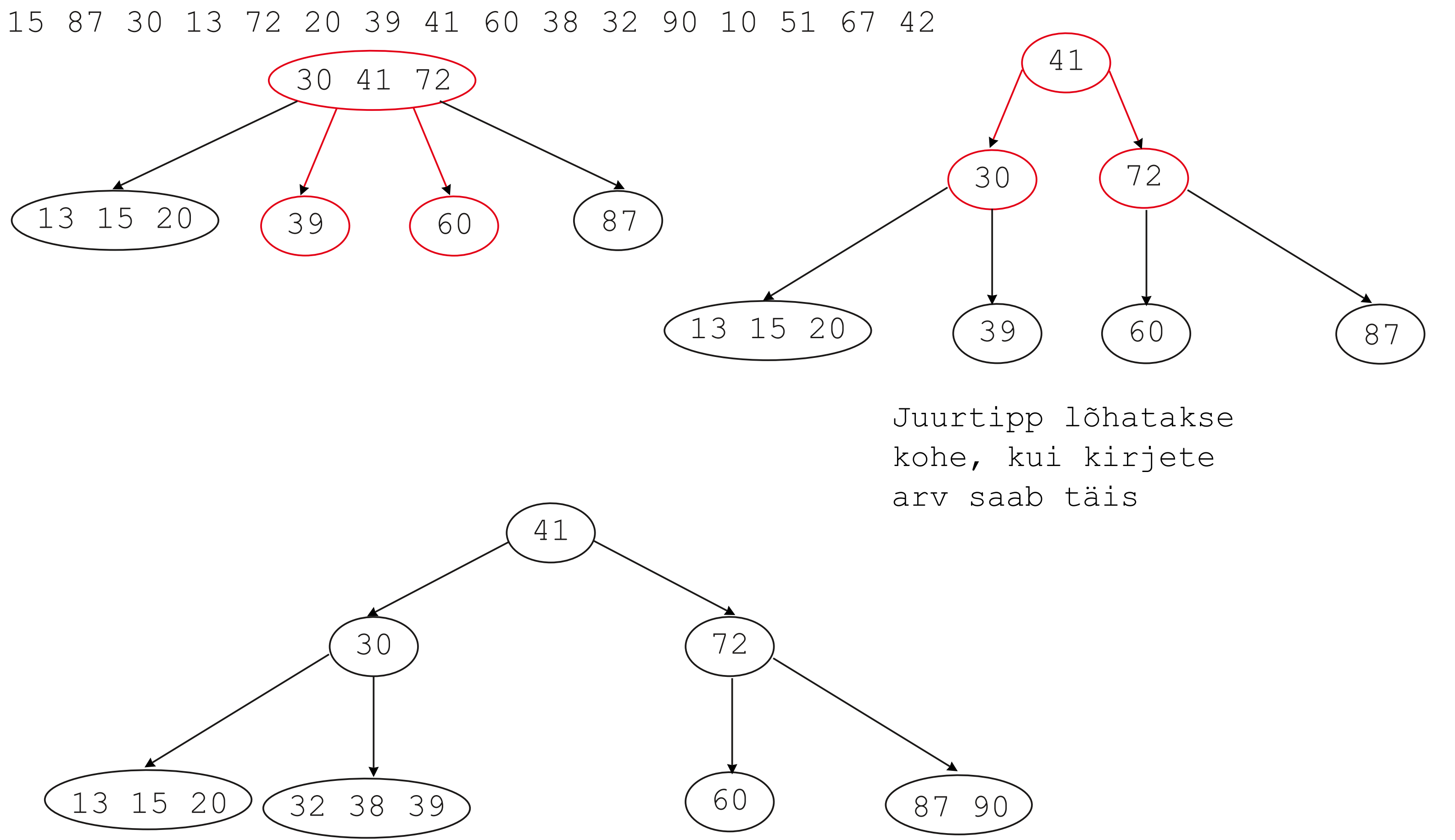
B-puud (2)

15 87 30 13 72 20 39 41 60 38 32 90 10 51 67 42



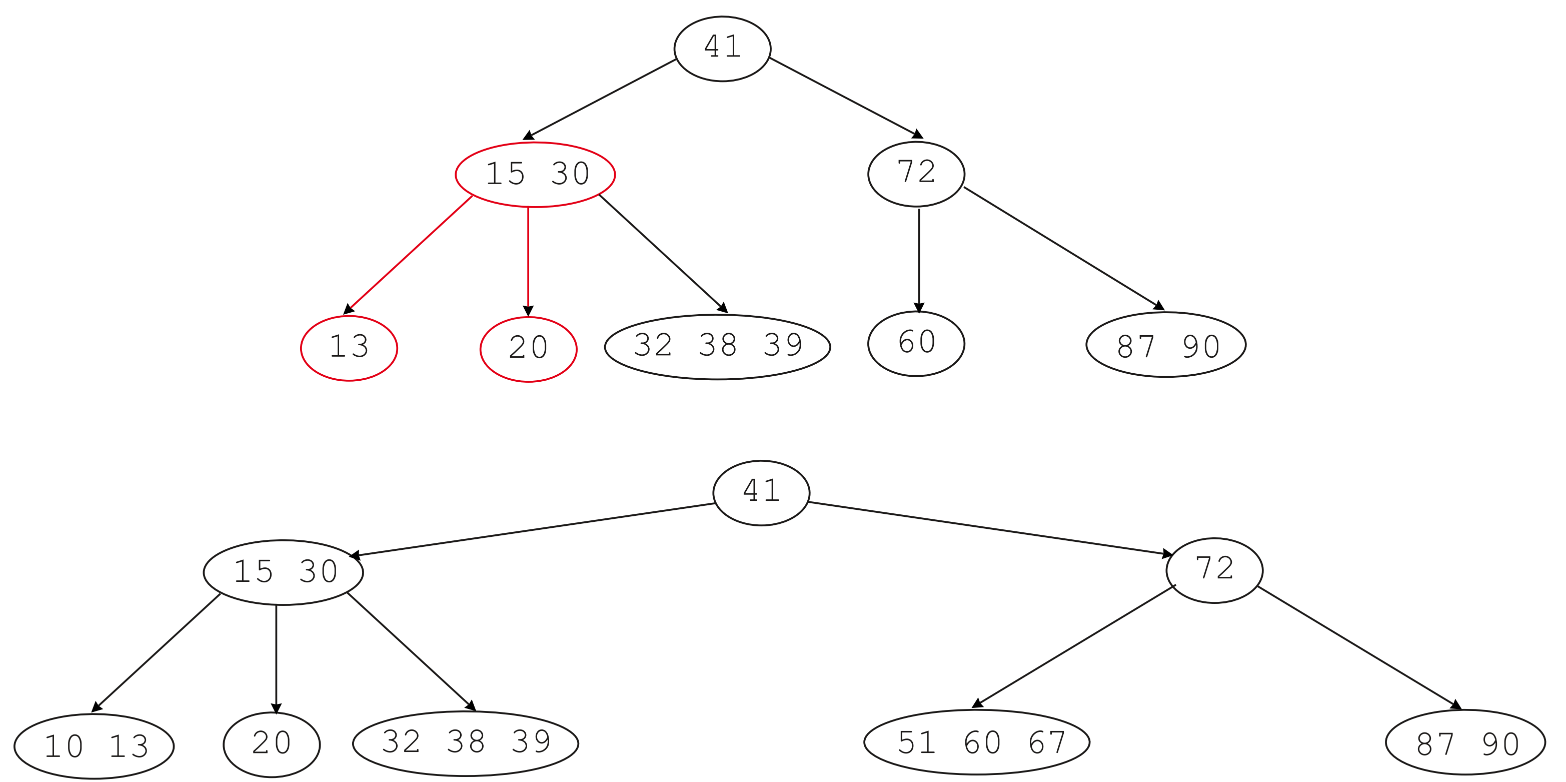
4-ndat järku B-puud nimetatakse ka 2-3-4 puuks.

B-puud (3)



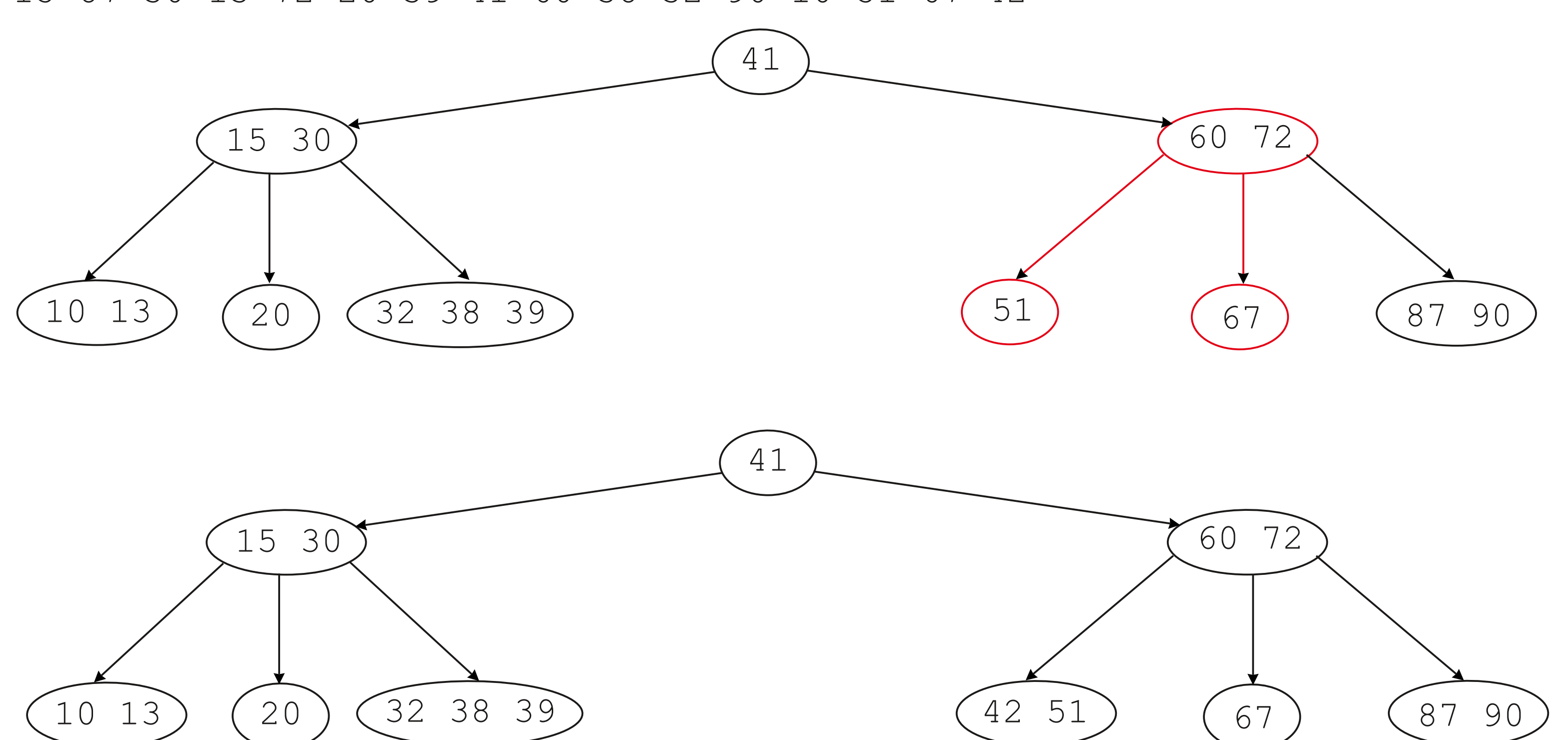
B-puud (4)

15 87 30 13 72 20 39 41 60 38 32 90 10 51 67 42

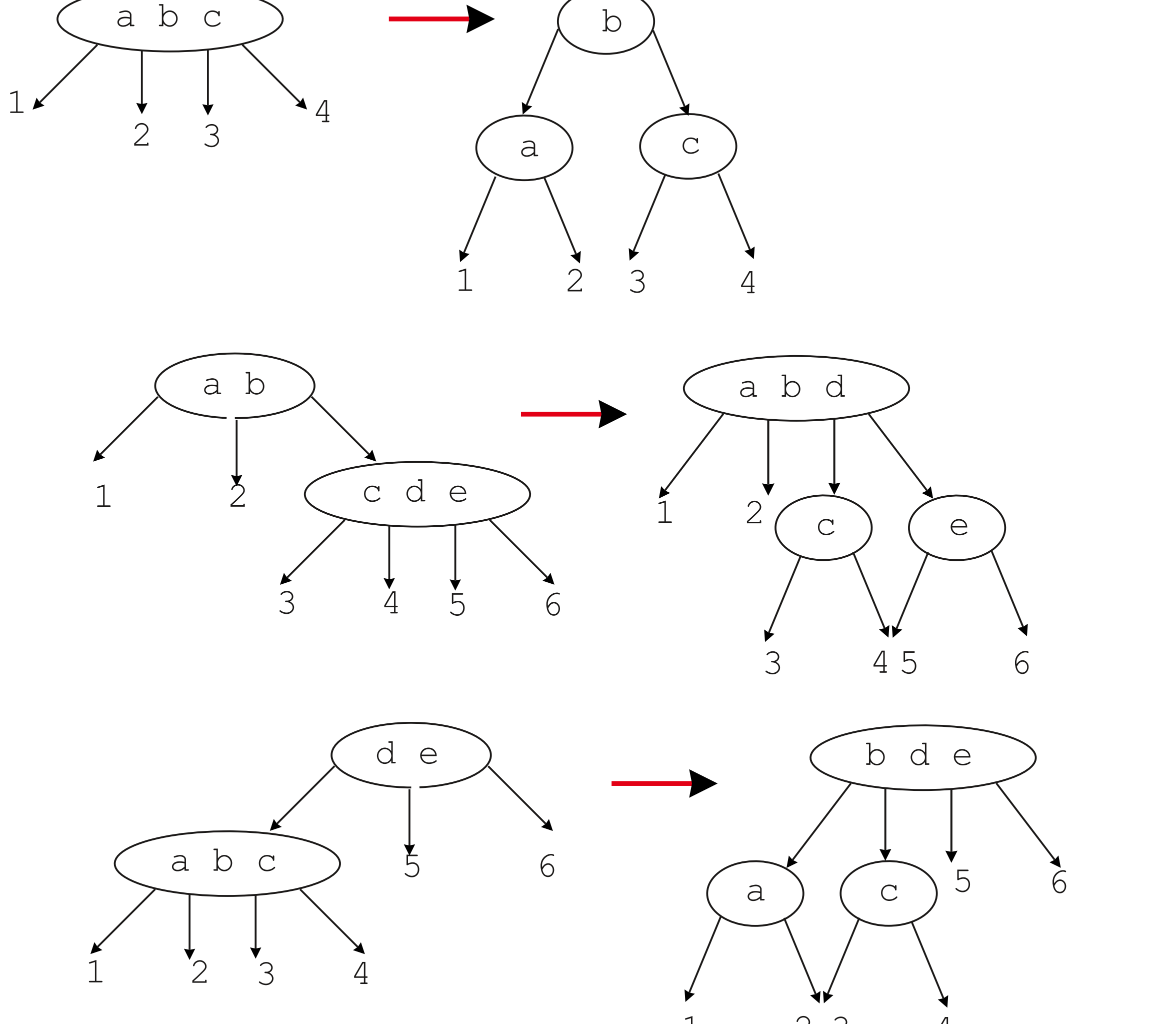


B-puud (5)

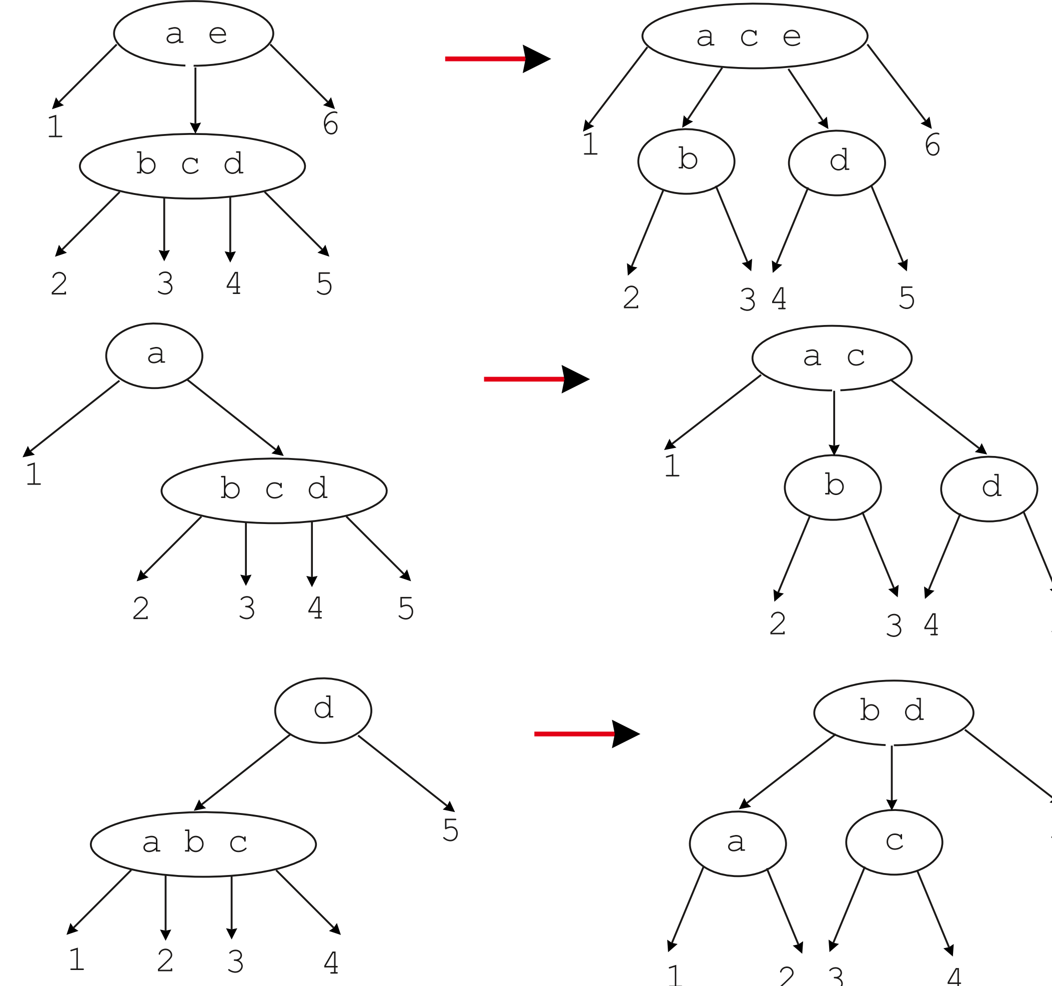
15 87 30 13 72 20 39 41 60 38 32 90 10 51 67 42



B-puud (6)

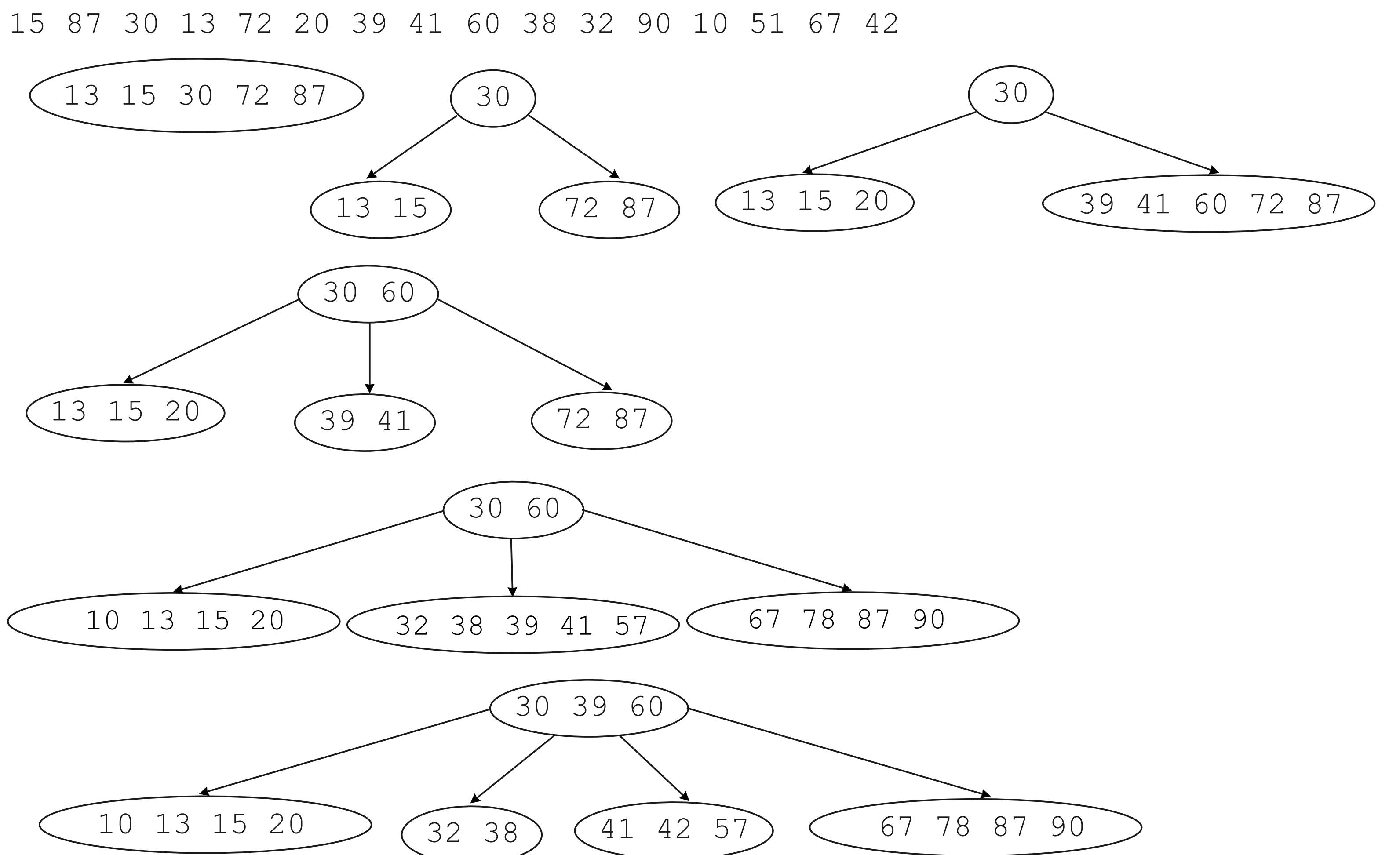


B-puud (7)



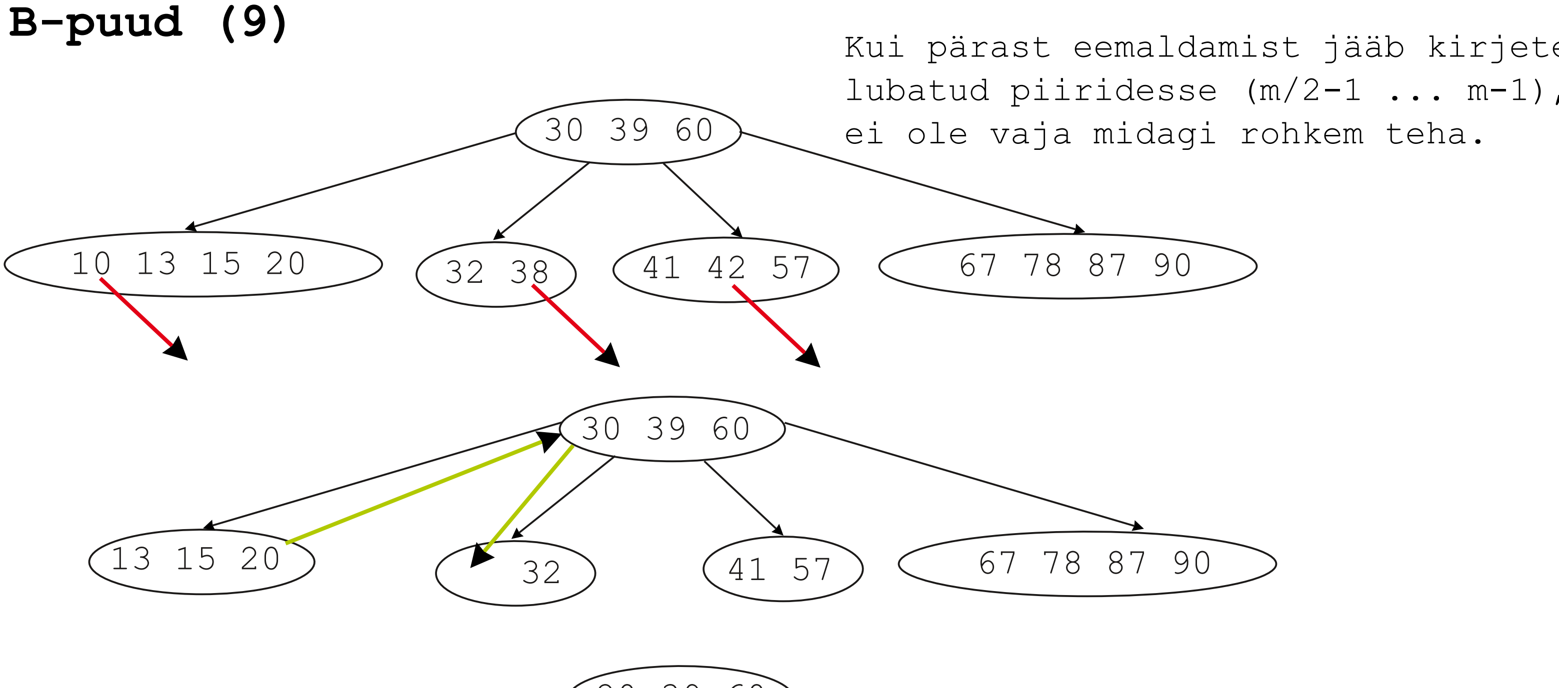
B-puud (8)

15 87 30 13 72 20 39 41 60 38 32 90 10 51 67 42



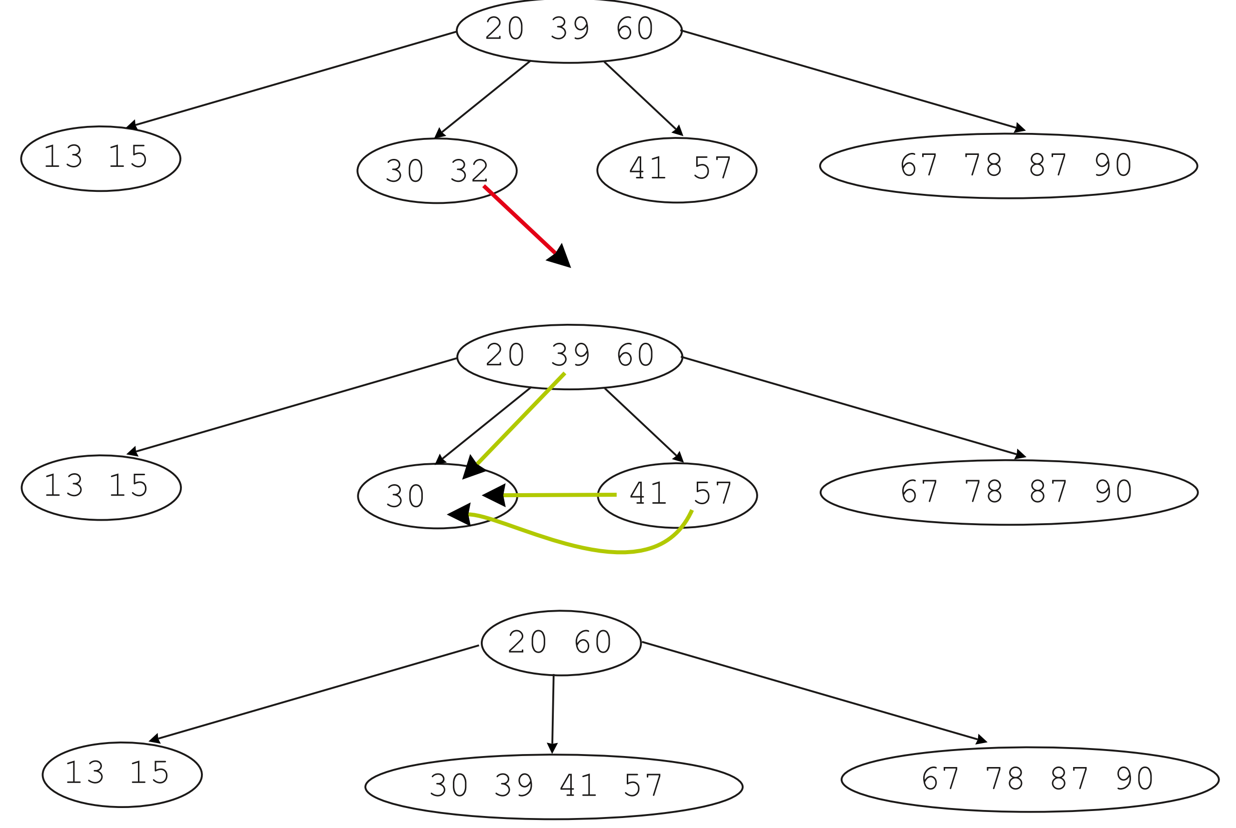
B-puud (9)

Kui pärast eemaldamist jääb kirjete arv lubatud piiridesse (m/2-1 ... m-1), ei ole vaja midagi rohkem teha.



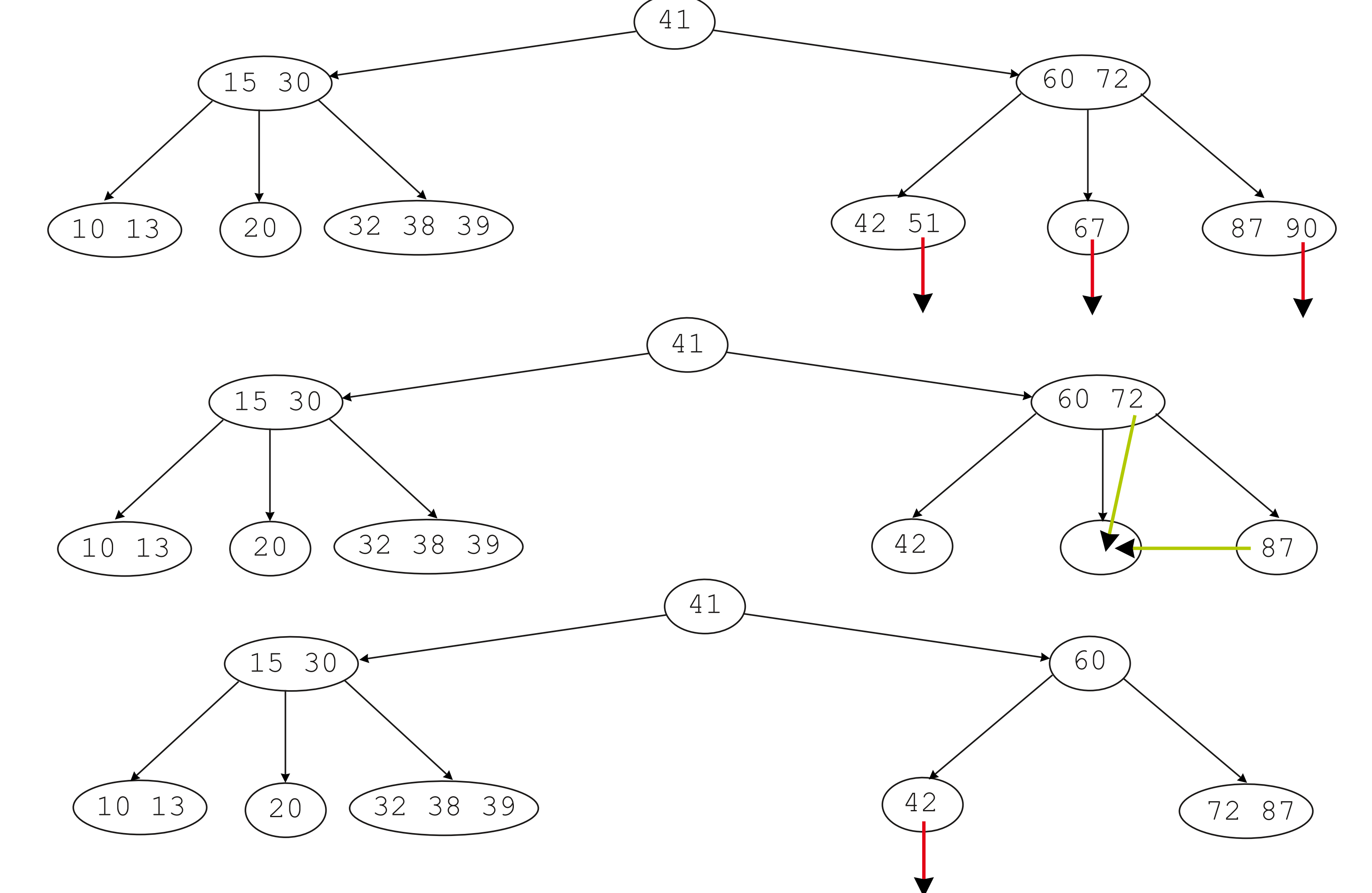
Kui pärast eemaldamist jääb tippu lubatust vähem kirjeid, tuleb vaadata, kas ühel õdedest on kirjete arv miinimumist vähemalt ühe võrra suurem. Kui selline õde leidub, tuleb teha joonisel näidatud ümberpaigutus.

B-puud (10)

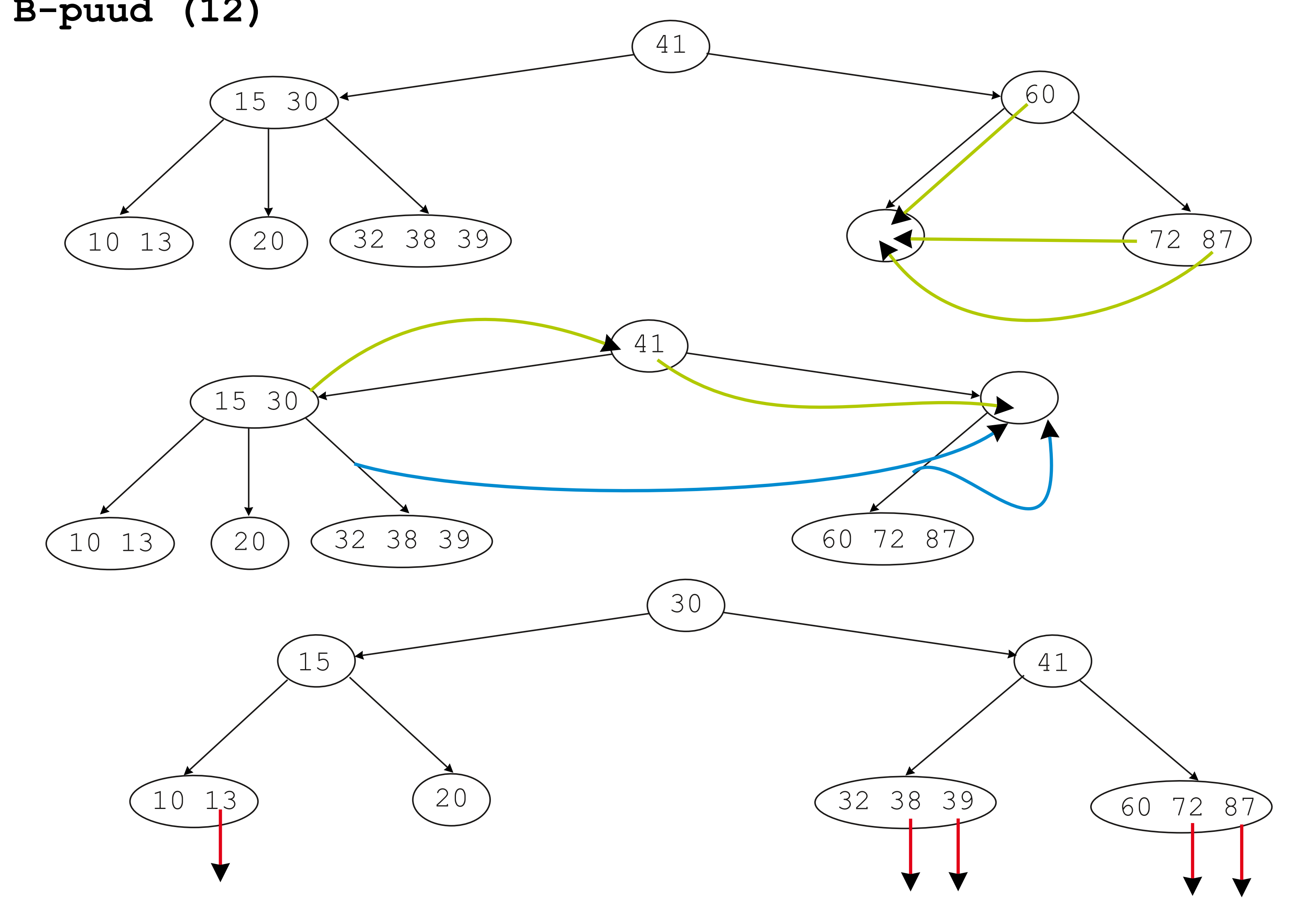


Kui pärast eemaldamist jääb tippu lubatust vähem kirjeid ja mõlemal õel on kirjete arv miinimumi piiril, tuleb tipp ja üks tema õde joonisel näidatud viisil kokku sulatada.

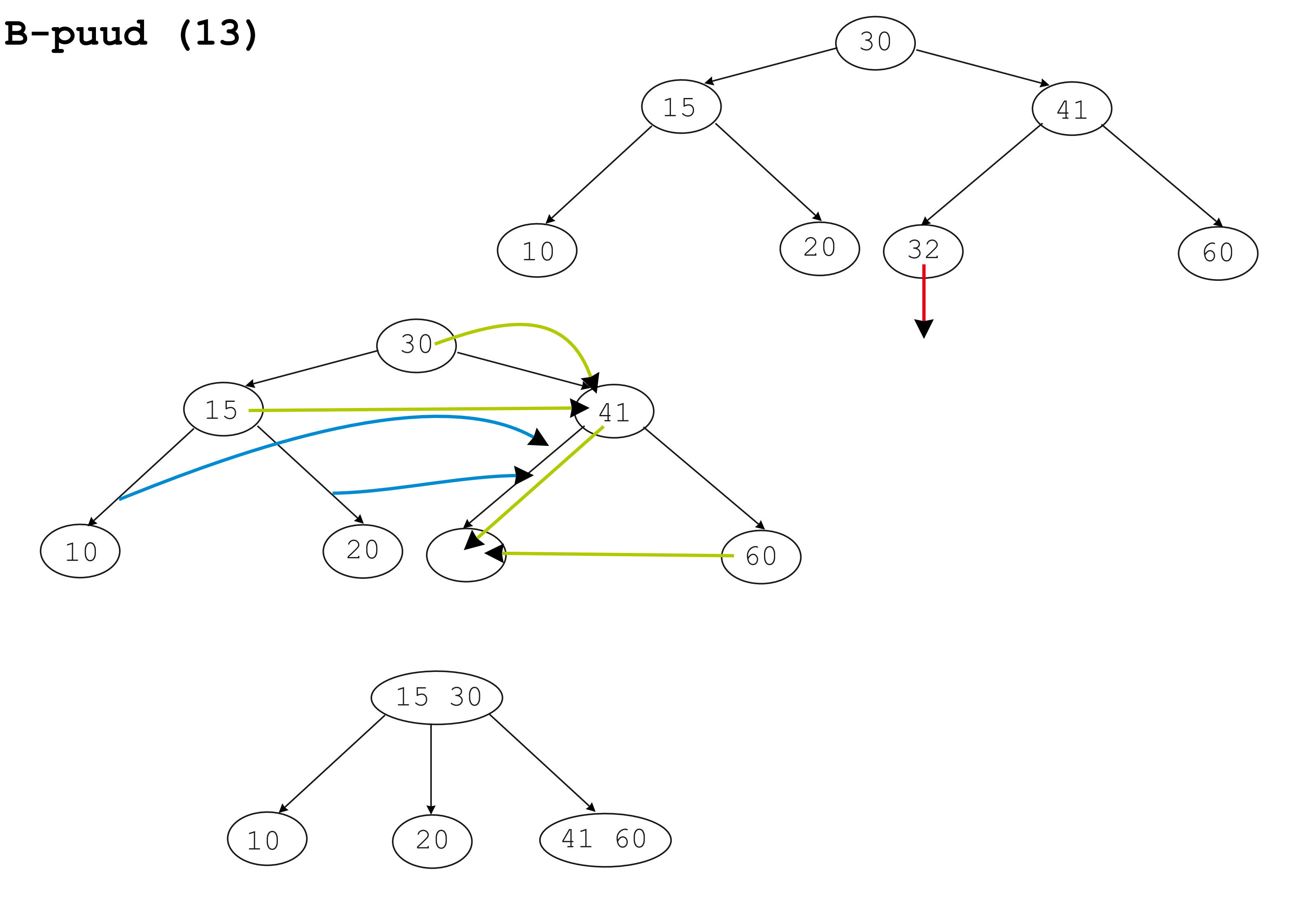
B-puud (11)



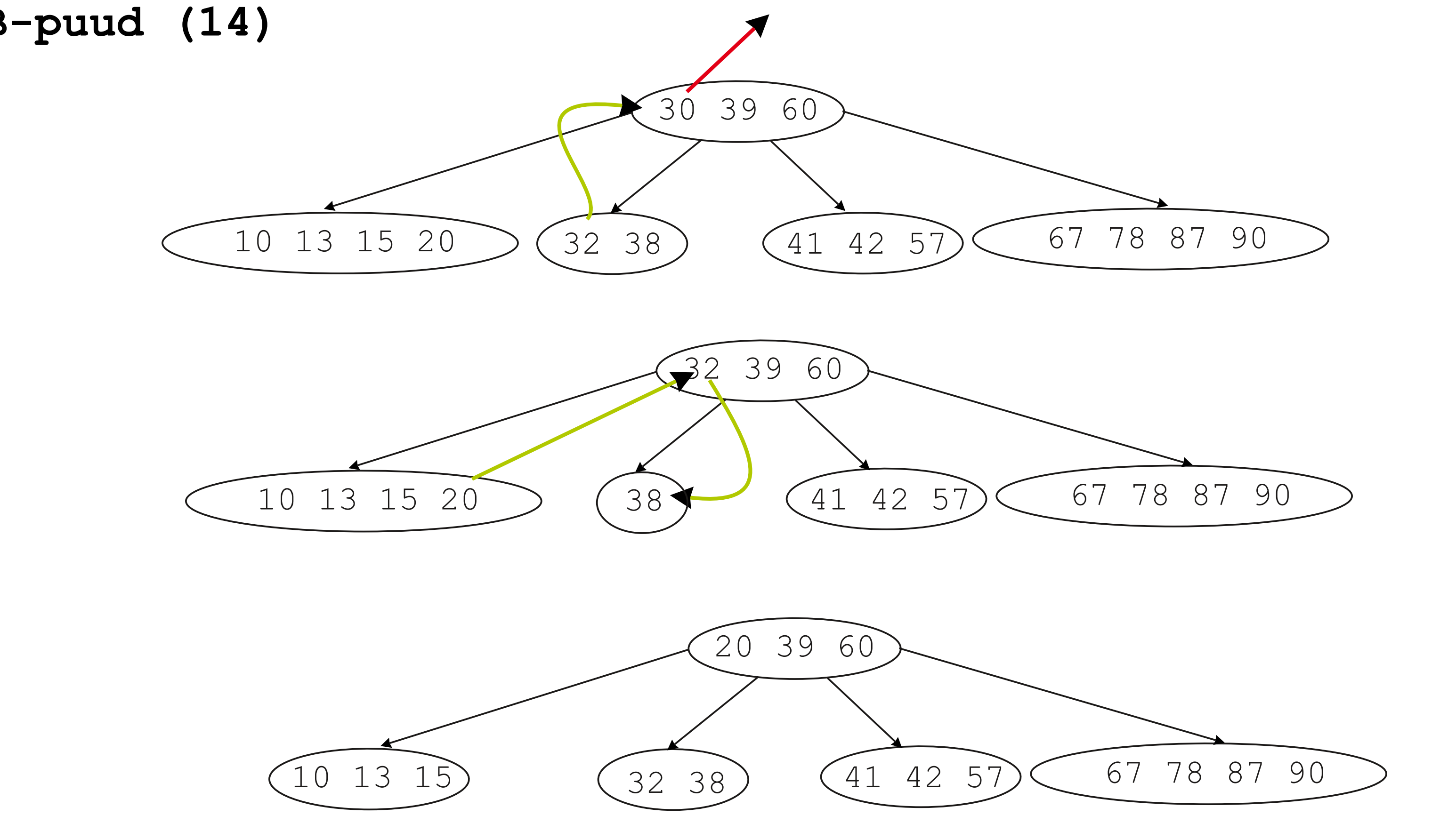
B-puud (12)



B-puud (13)



B-puud (14)



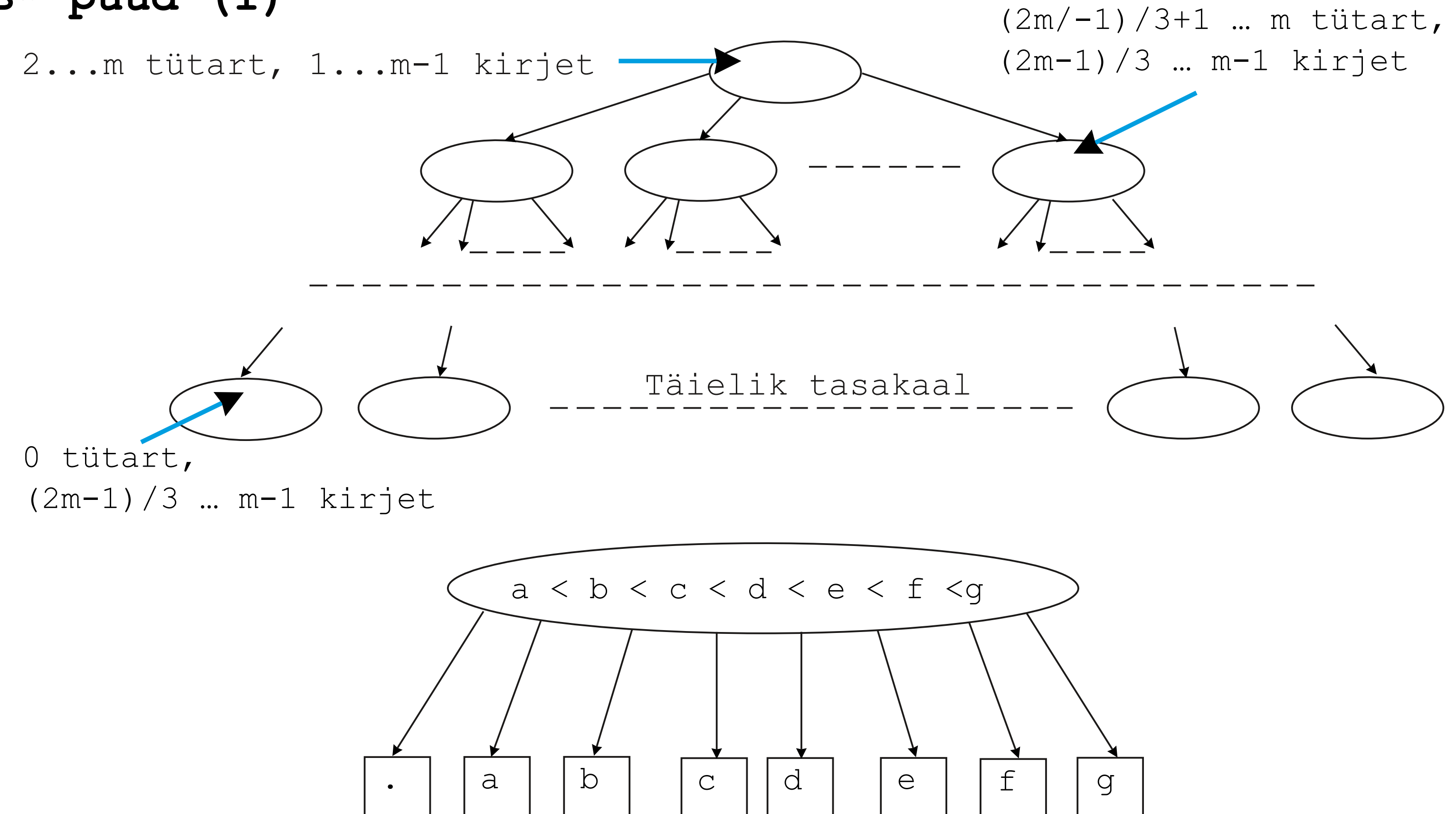
Siiamaani vaatasime eemaldamist lehtedest. Kui eemaldatav kirje asub kõrgemal, tuleb leida talle suuruselt järgnev kirje ja tuua tema asemele. S.t. satume olukorda, kus eemaldamine toimub lehelt.

B-puud (15)

Täiendavat materjali B-puude kohta leiab:
https://www.geeksforgeeks.org/b-tree-set-1-introduction-2/
https://www.geeksforgeeks.org/b-tree-set-1-insert-2/
https://www.geeksforgeeks.org/b-tree-set-3delete/
Samas on toodud ka vastavad C/C++ funktsioonide koodid.

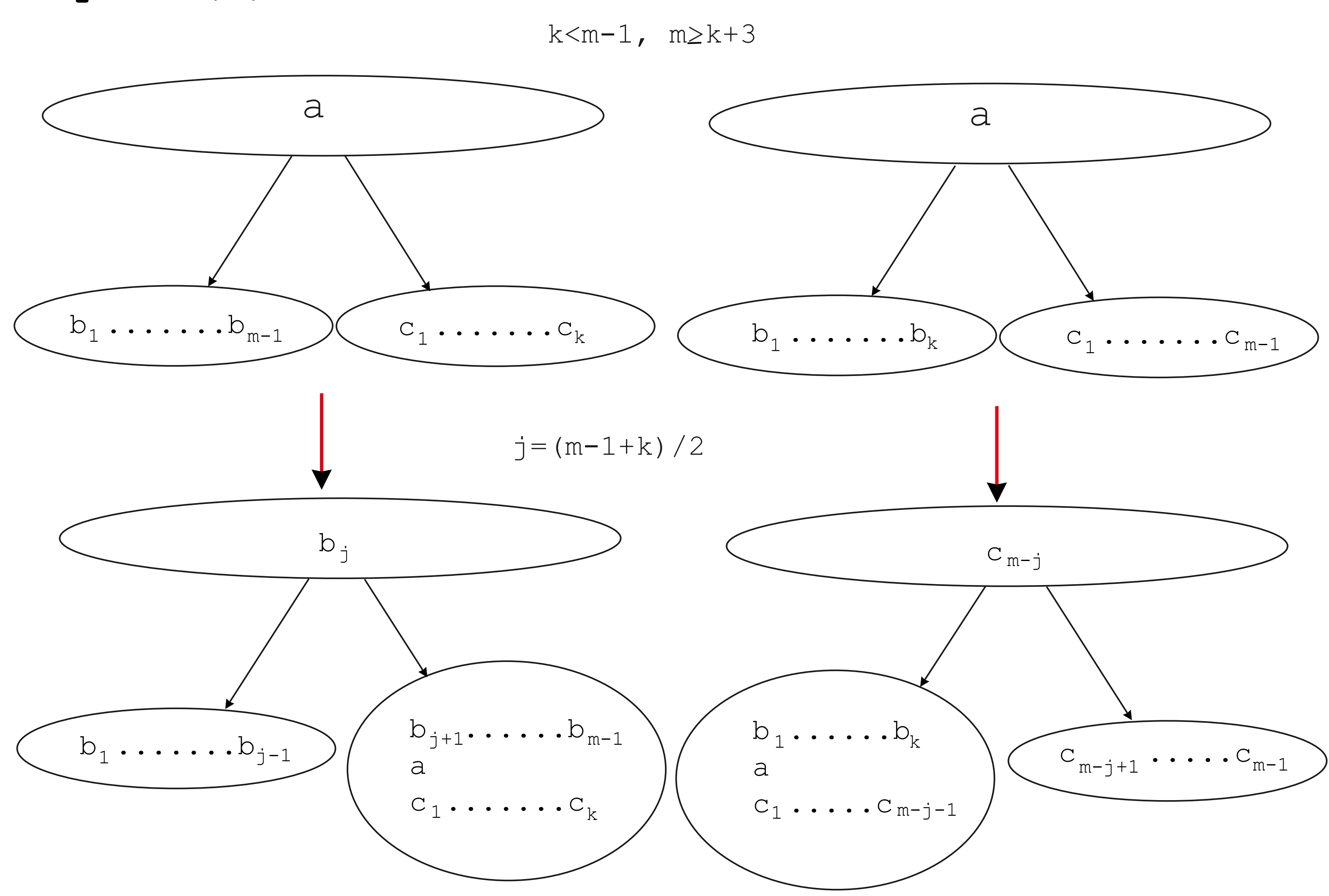
B-puust kustutamise kohta võib lugeda ka
https://medium.com/@vijinimallawaarachchi/all-you-need-to-know-about-deleting-keys-from-b-trees-9090f3334b5c

B*-puud (1)

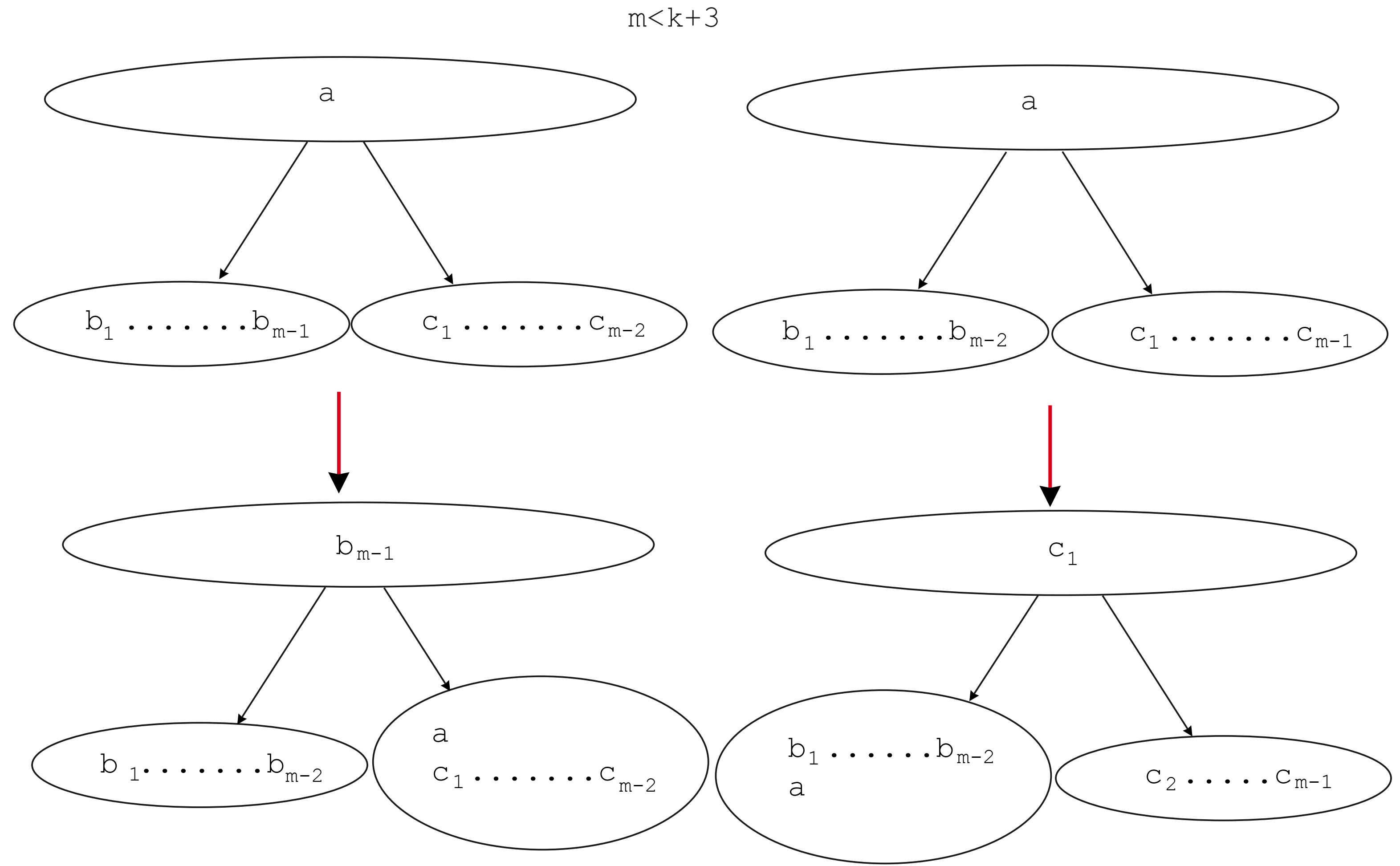


Tipu täituvus on parem: võimalik tühjaks jääv osa ei ole rohkem kui 1/3 ketta plokist (B-puu puhul 1/2).

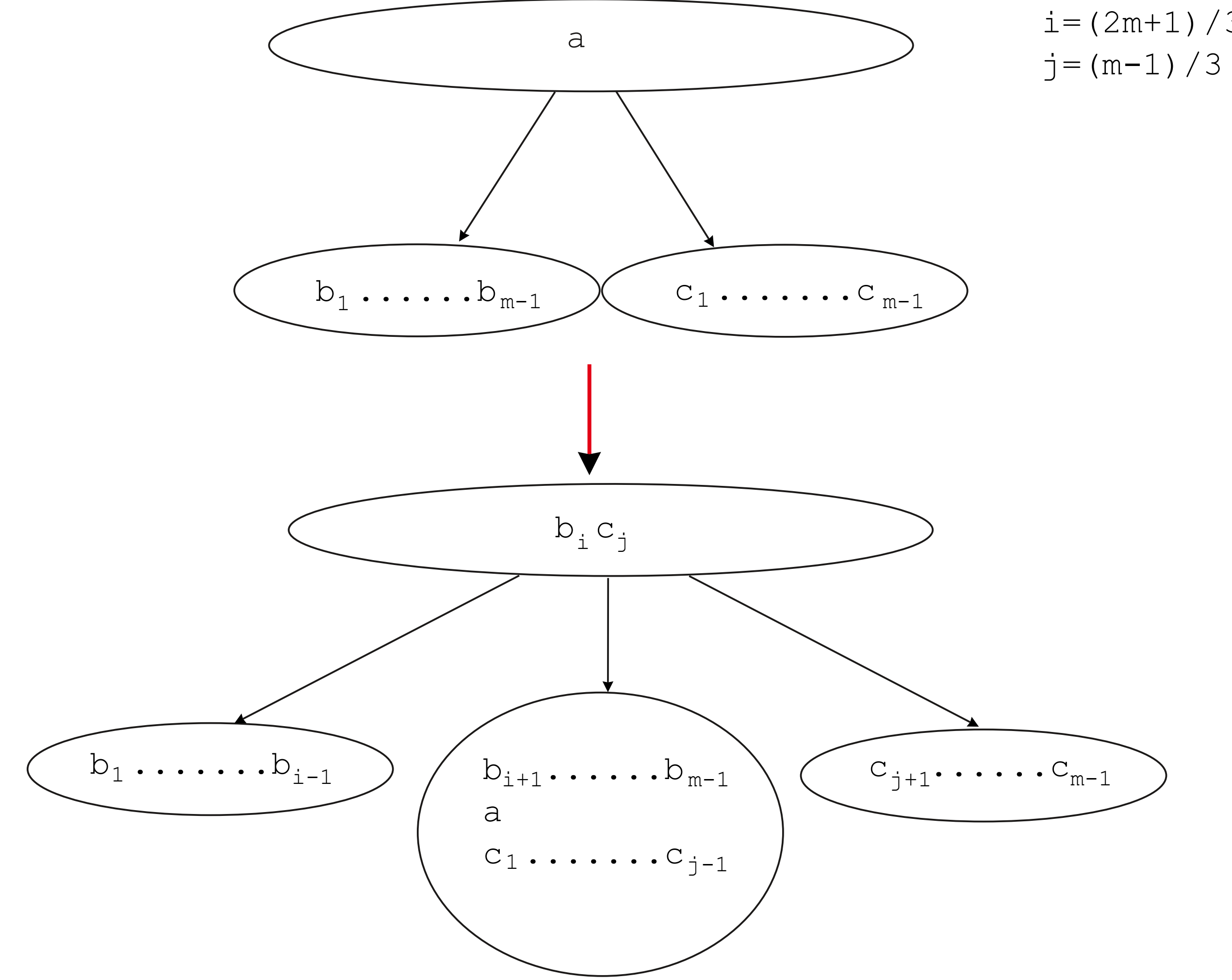
B*-puud (2)



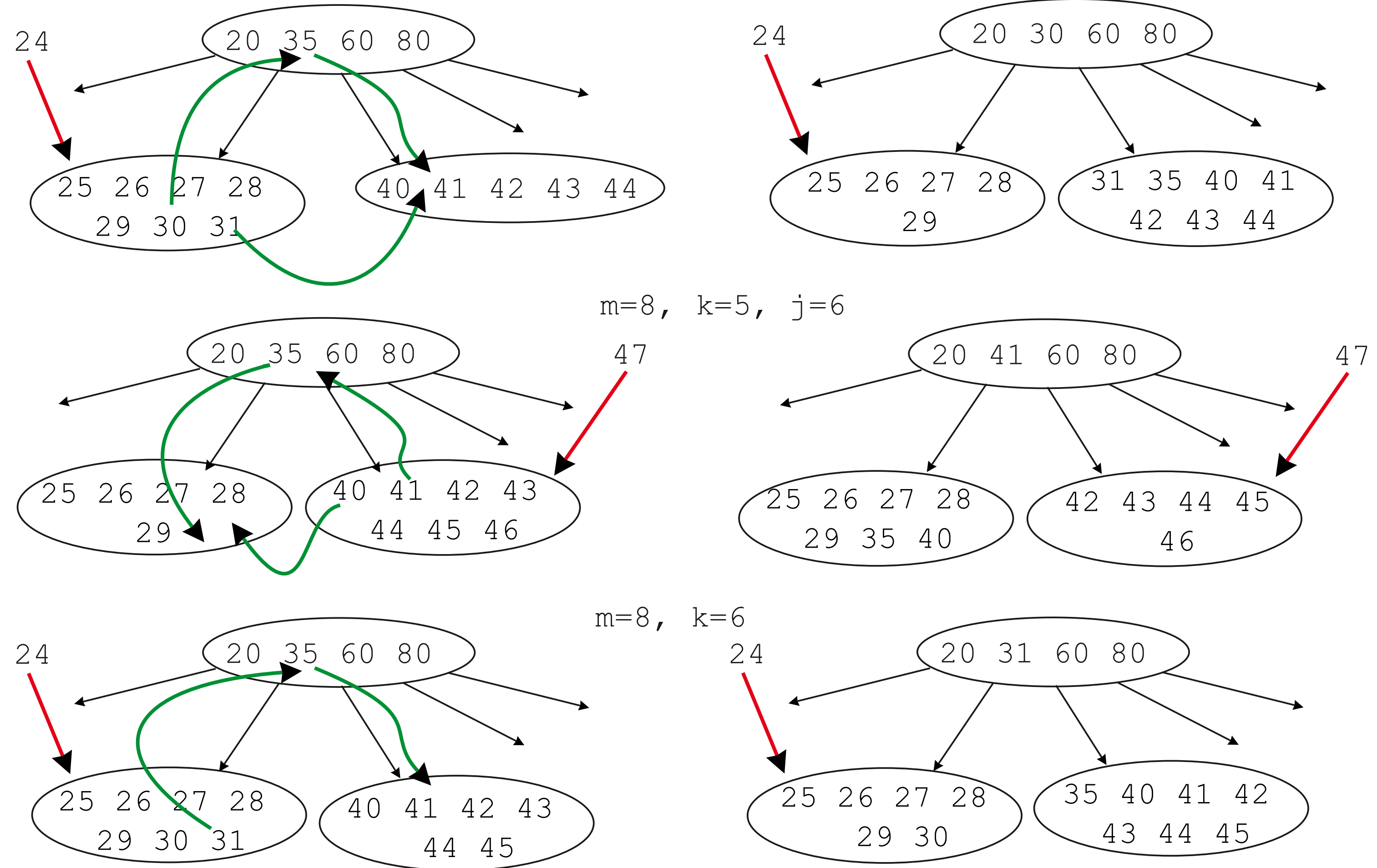
B*-puud (3)



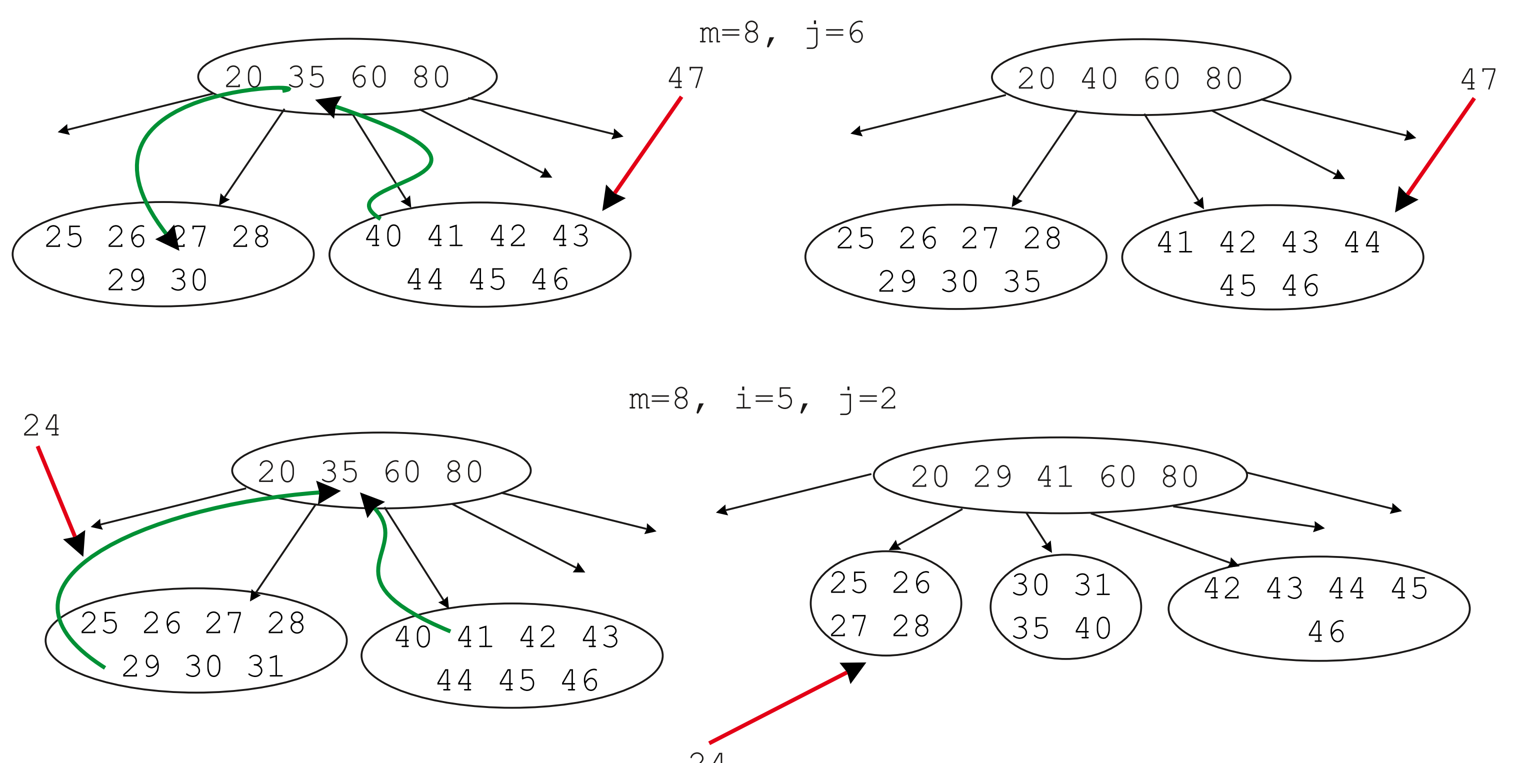
B*-puud (4)



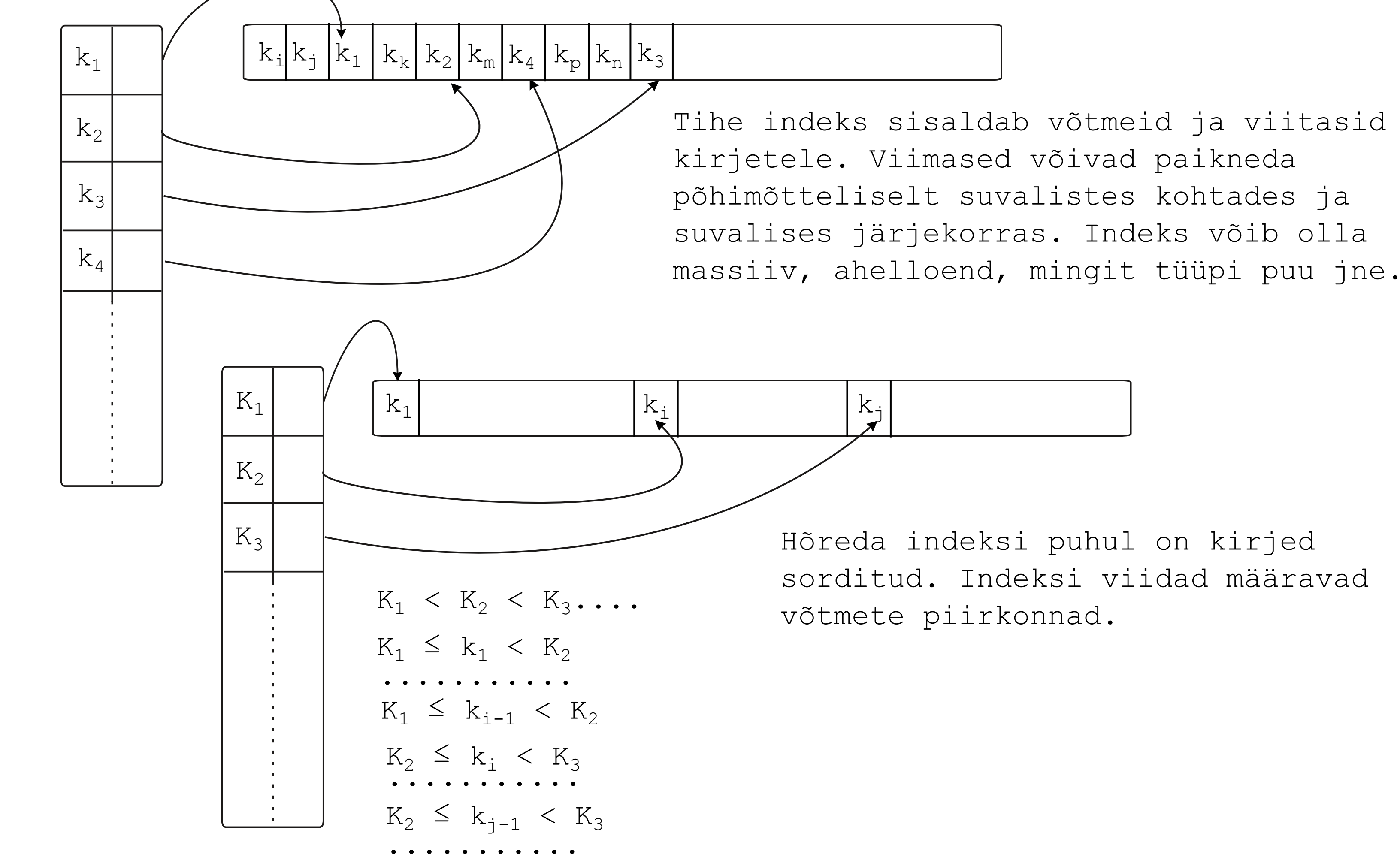
B*-puud (5)



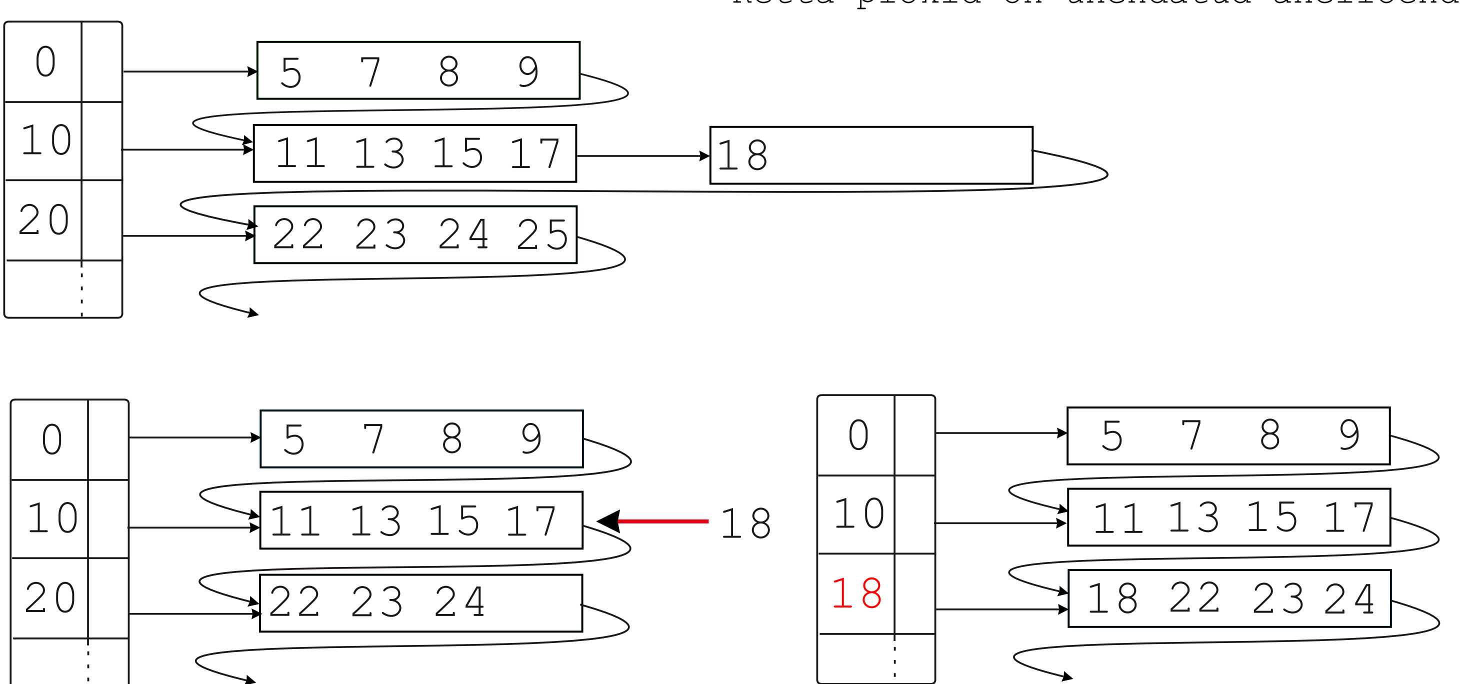
B*-puud (6)



Indekseeritud jadafailid (indexed sequential access method, ISAM) (1)



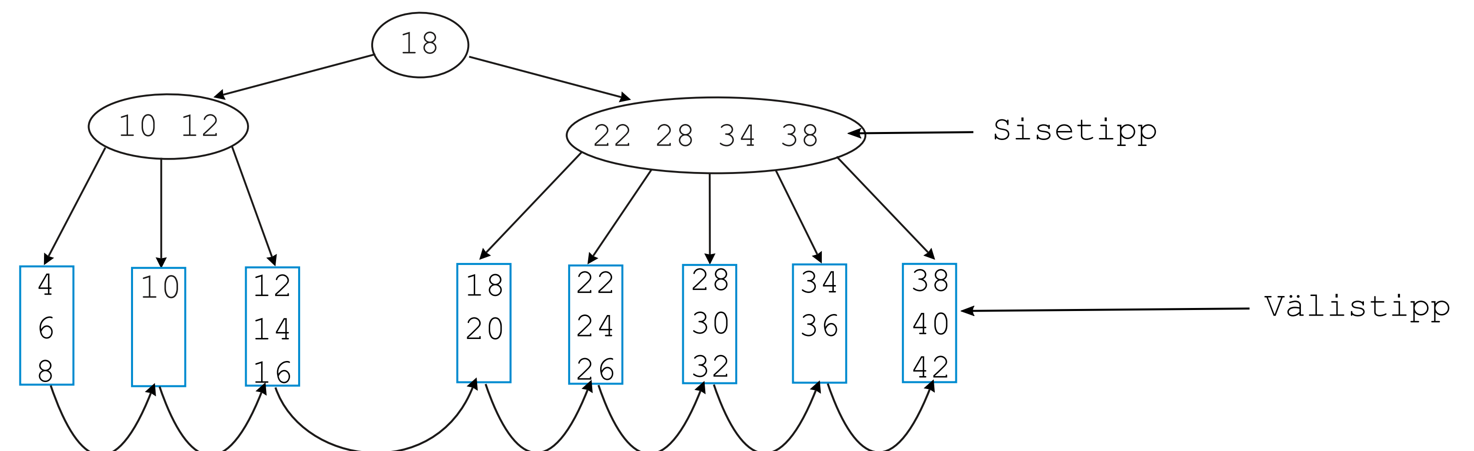
Indekseeritud jadafailid (2)



Uute plokkide vahelepanek loendisse on mõistlik vaid siis kui muud võimalust pole. Sageli on lihtsaimaks lahendiks indeksis olevate piiride muutmine.

B+ puud (1)

m=5, plokki mahub max 3 kirjet

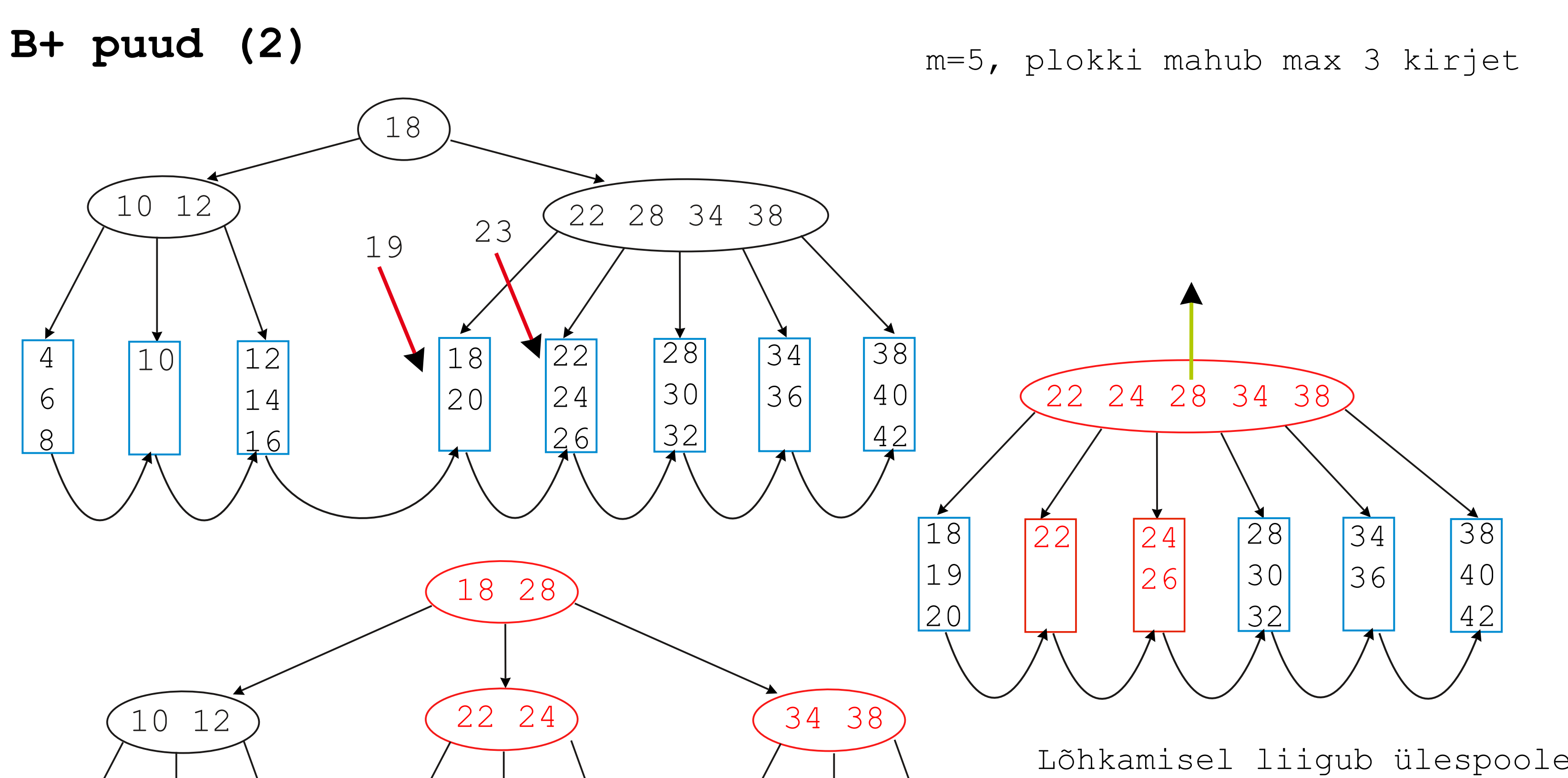


B+ puu on ISAM fail, mille hõre indeks on ehitatud B-puu põhimõttel. Kirjed on välistippudes e. ketta plokkides. Viimased moodustavad sorditud ahelloendi. Samas on nad ka B-puu lehed. B-puu ülejäänud tipud on sisetipud ja sisaldavad hõreda indeksi separaatoreid.

B+ puu sisetipud asuvad samuti kettal.

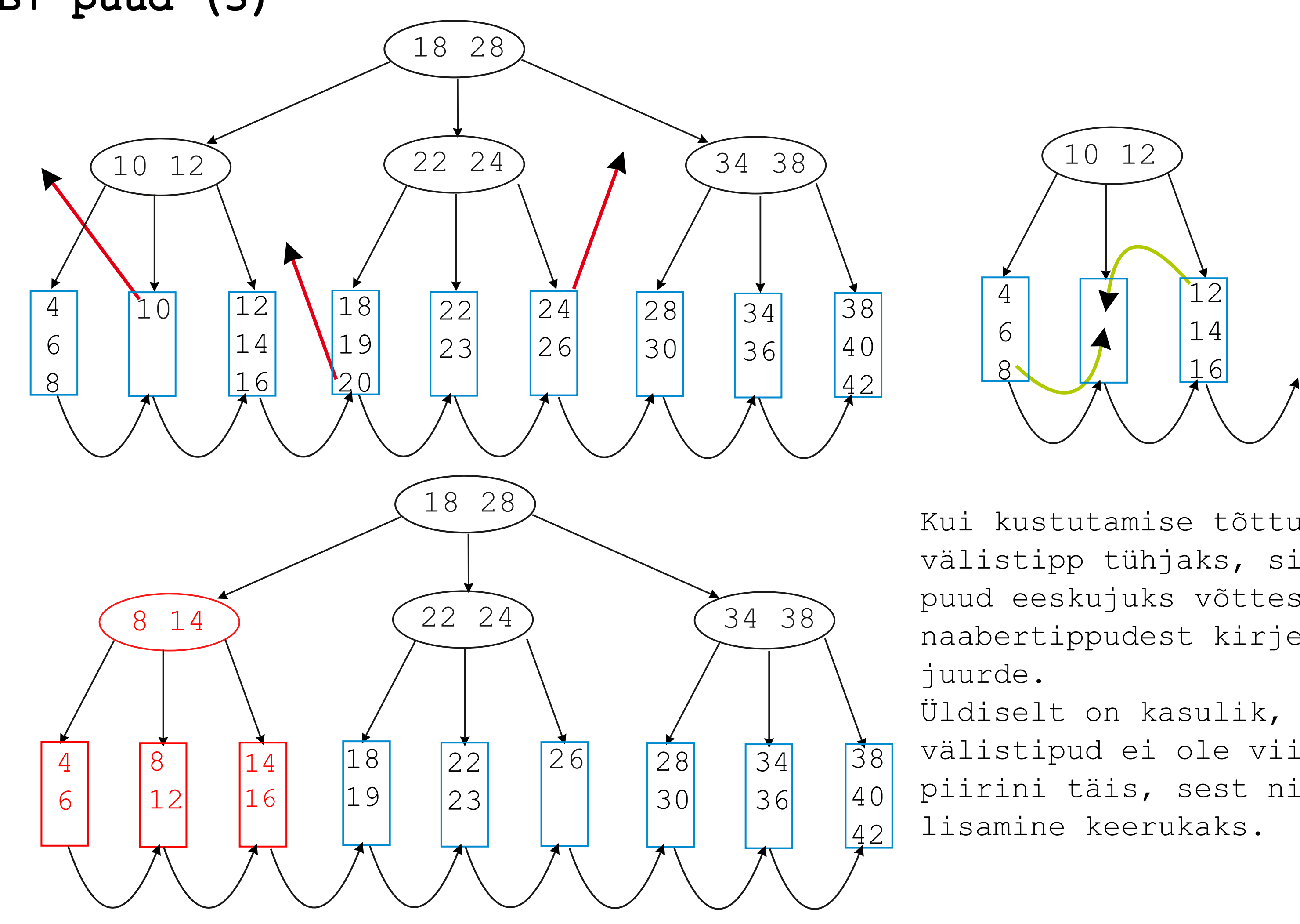
B+ puud (2)

m=5, plokki mahub max 3 kirjet



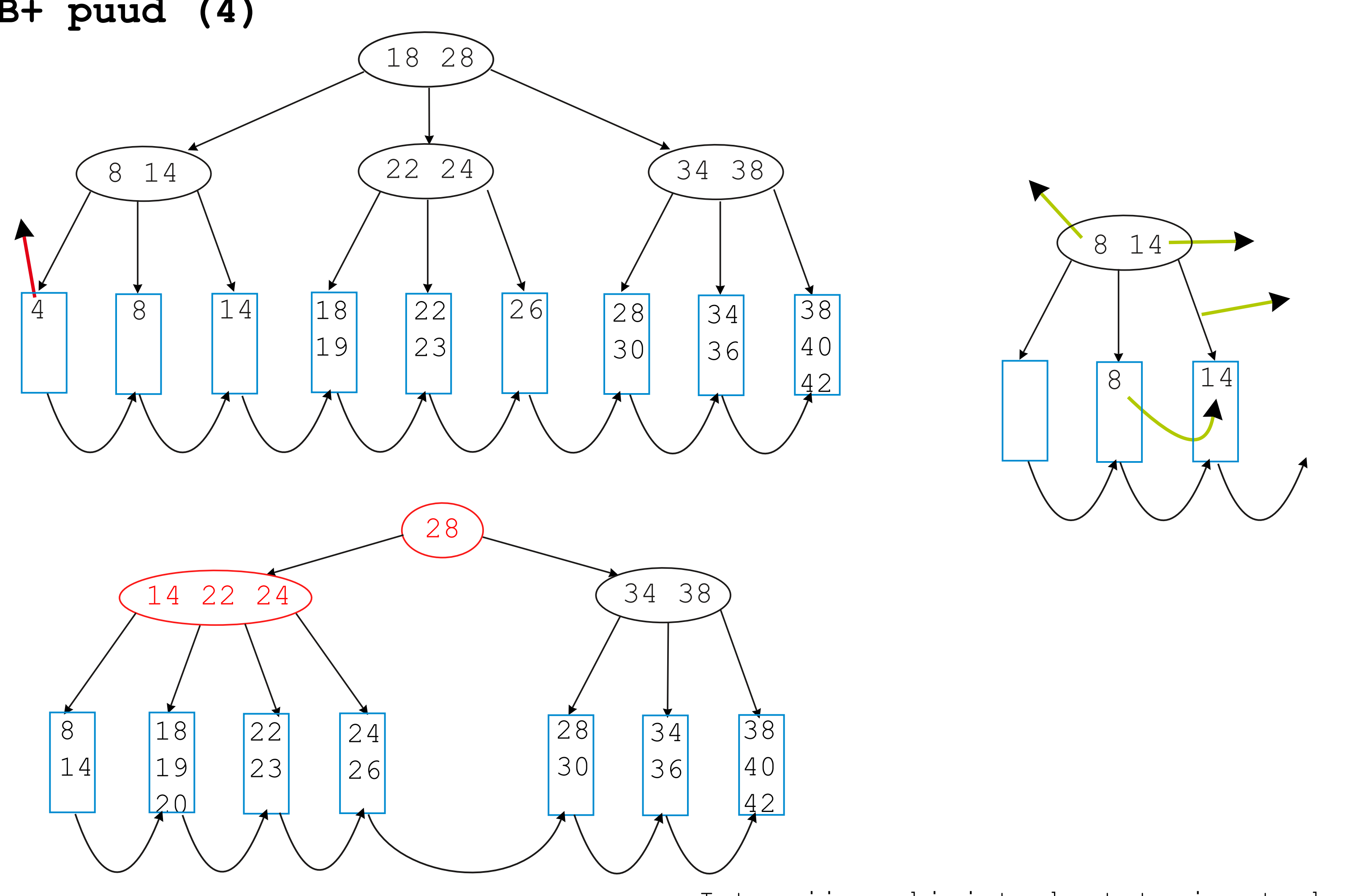
Lõhkamisel liigub ülespoole mitte kirje vaid tema võtme koopia. Kirje ise jääb loomulikult ühte välistippu edasi.

B+ puud (3)



Kui kustutamise tõttu jääb välistipp tühjaks, siis B* puud eeskujuks võttes tuuakse naabertippudest kirjeid juurde.
Üldiselt on kasulik, kui välistipud ei ole viimase piirini täis, sest nii muutub lisamine keerukaks.

B+ puud (4)

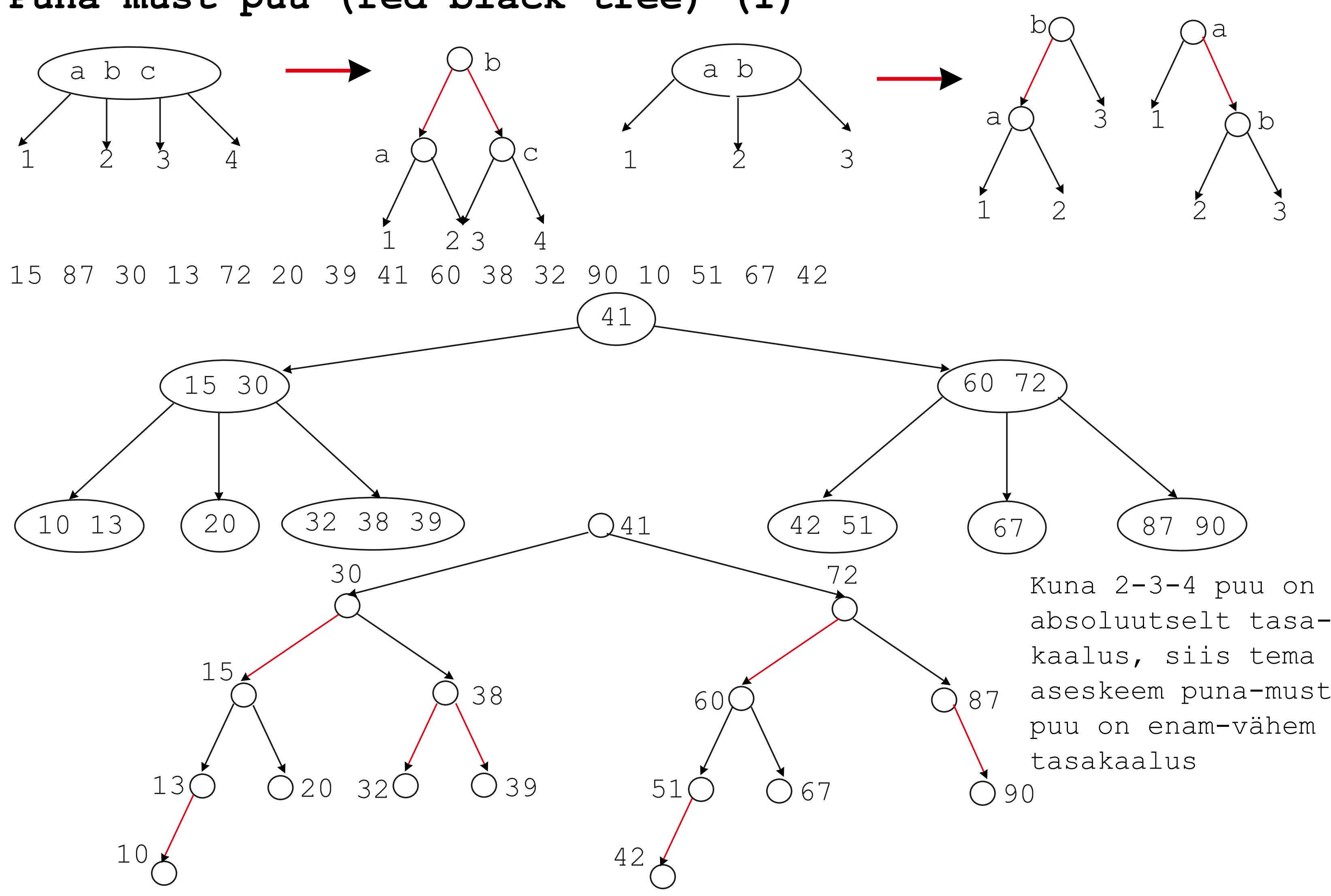


Intensiivne kirjete kustutamine toob kaasa ka indeksi ümberkorraldamise

Konteineri pakkimine (bin packing)

Konteinerite all võib mõtelda ketta plokke. Probleem seisneb nende täitmises kirjetega, millede pikkused võivad olla väga erinevad.
Eeldame, et koos pakkimisega luuakse ka tihe indeks. Sel juhul võib iga kirje panna põhimõtteliselt mistahes plokki.
Algus on alati üks: esimesse konteinerisse pannakse niipalju kirjeid kui saab.
Edasi on aga kolm strateegiat:
1. Next fit: kui järjekordne kirje jooksvasse konteinerisse enam ei mahu, see suletakse ja hakatakse täitma järgmist konteinerit.
2. First fit: ühtegi konteinerit ei suleta. Iga kirje puhul käiakse läbi kõik konteinerid alates esimesest kuni leitakse konteiner, kuhu uut kirjet on võimalik paigutada.
3. Best fit: ühtegi konteinerit ei suleta. Iga uue kirje puhul käiakse läbi kõik olemasolevad konteinerid. Uus kirje pannakse sinna, kuhu pärast uue kirje sisestamist jääb kõige vähem vaba ruumi.

Puna-must puu (red-black tree) (1)

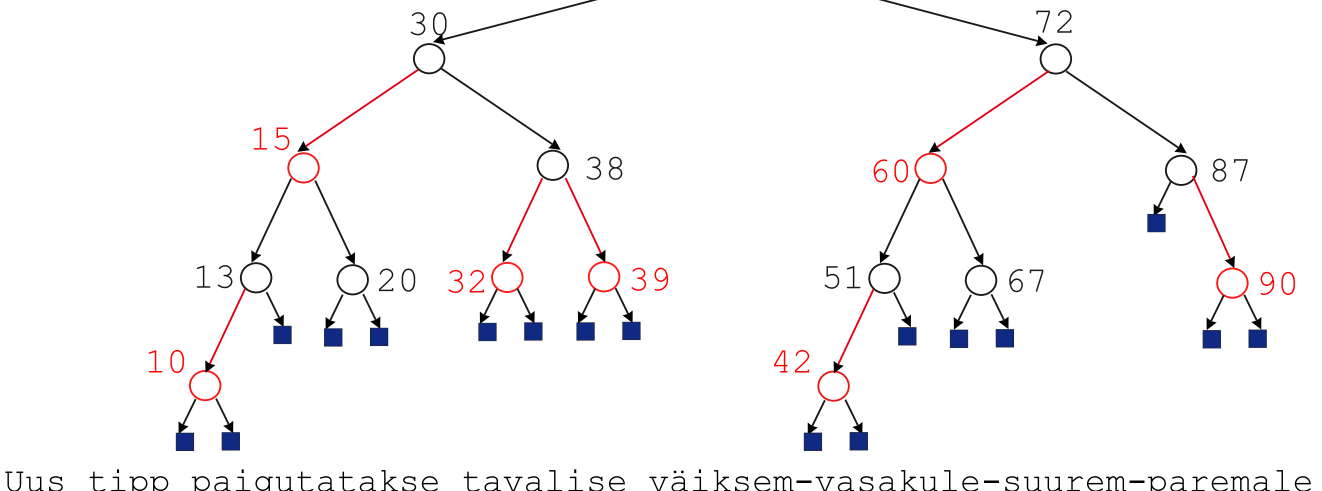


Puna-must puu (2)

Puna-musta puud ehitatakse muidugi mitte eelnevalt konstrueeritud 2-3-4 puud asendades vaid nullist alates ja kindlate reglite alusel:

1. Tipp on punane, kui temasse siseneb punane kaar ning must, kui temasse siseneb must kaar.
2. Juur on alati must.
3. Kui tipp ise on punane, siis ta tütred peavad olema mustad.
4. Kujutame ette, et igal lehel on veel eraldi mustade välistippude paar.

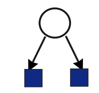
Vaatleme kõiki teekondi mistahest sisetipust nendesse välistippudesse,kuhu antud sisetipust võib jõuda. Siis mustade kaarte arv kõigil neil teekondadel peab olema sama.



Uus tipp paigutatakse tavalise väiksem-vasakule-suurem-paremale reegli alusel. Talle omistatakse punane värv.Kui reeglid (3) ja (4) on pärast uue tipu paigutamist rikutud,tuleb asuda puud korrastama. Selleks on 2 võtet: pööramine ja värvi vahetus (color flip).

Puna-must puu (3)

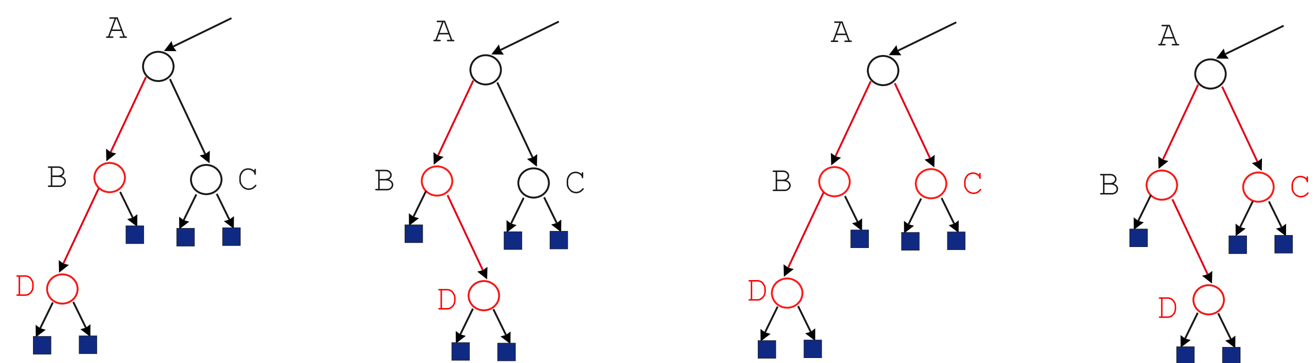
Ehitamine algab juurest, viimast loetakse alati mustaks tipuks.



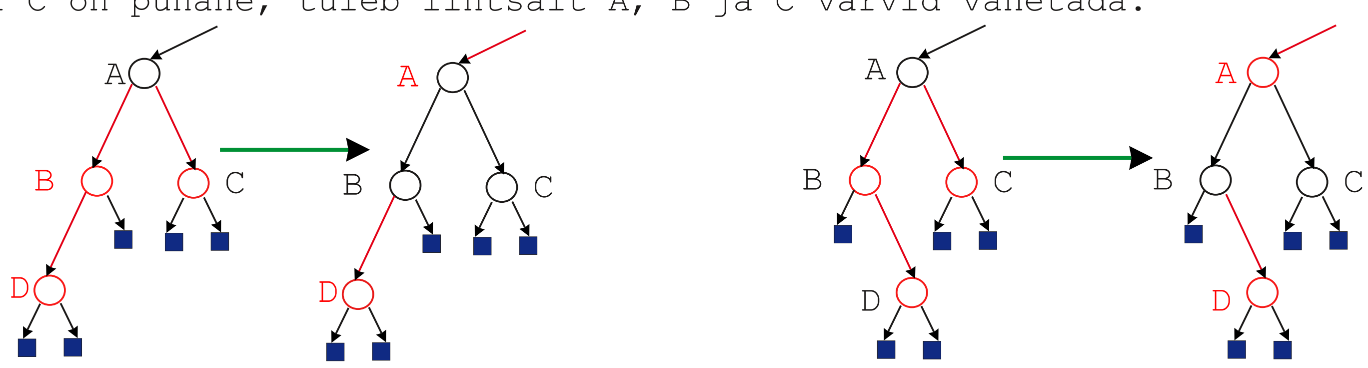
Kui musta tipu A külge kinnitub uus tipp B, on ta punane. Pärast seda puu korrastamist ei vaja, sest kõik reeglid on täidetud.

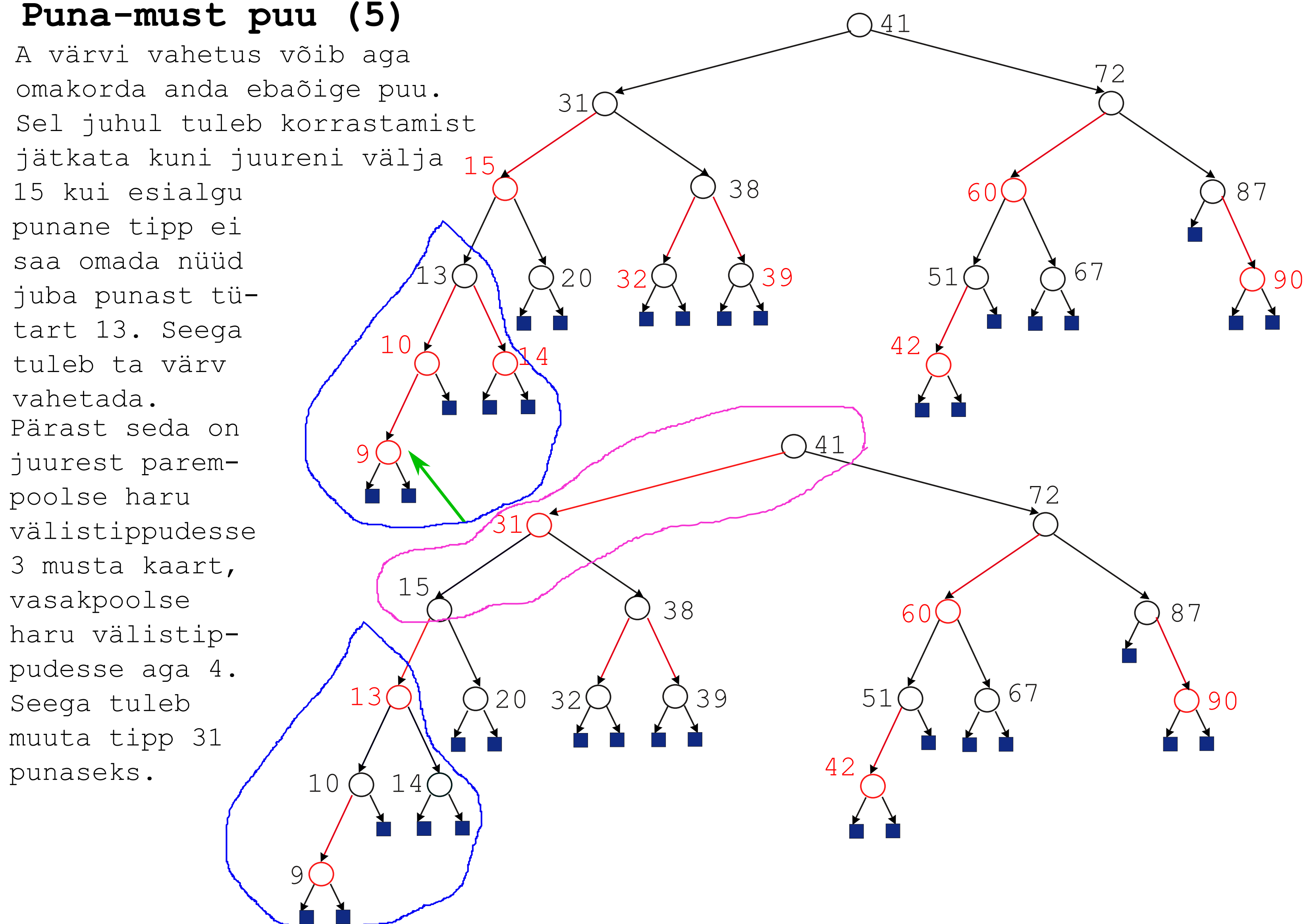
Puna-must puu (4)

Kui punase tipu B külge kinnitub uus tipp D, on ta ka punane. Pärast seda tuleb puud korrastada. Korrastamise reeglite valik sõltub tipu B õetipu C värvist. Saame neli erinevat juhtu.



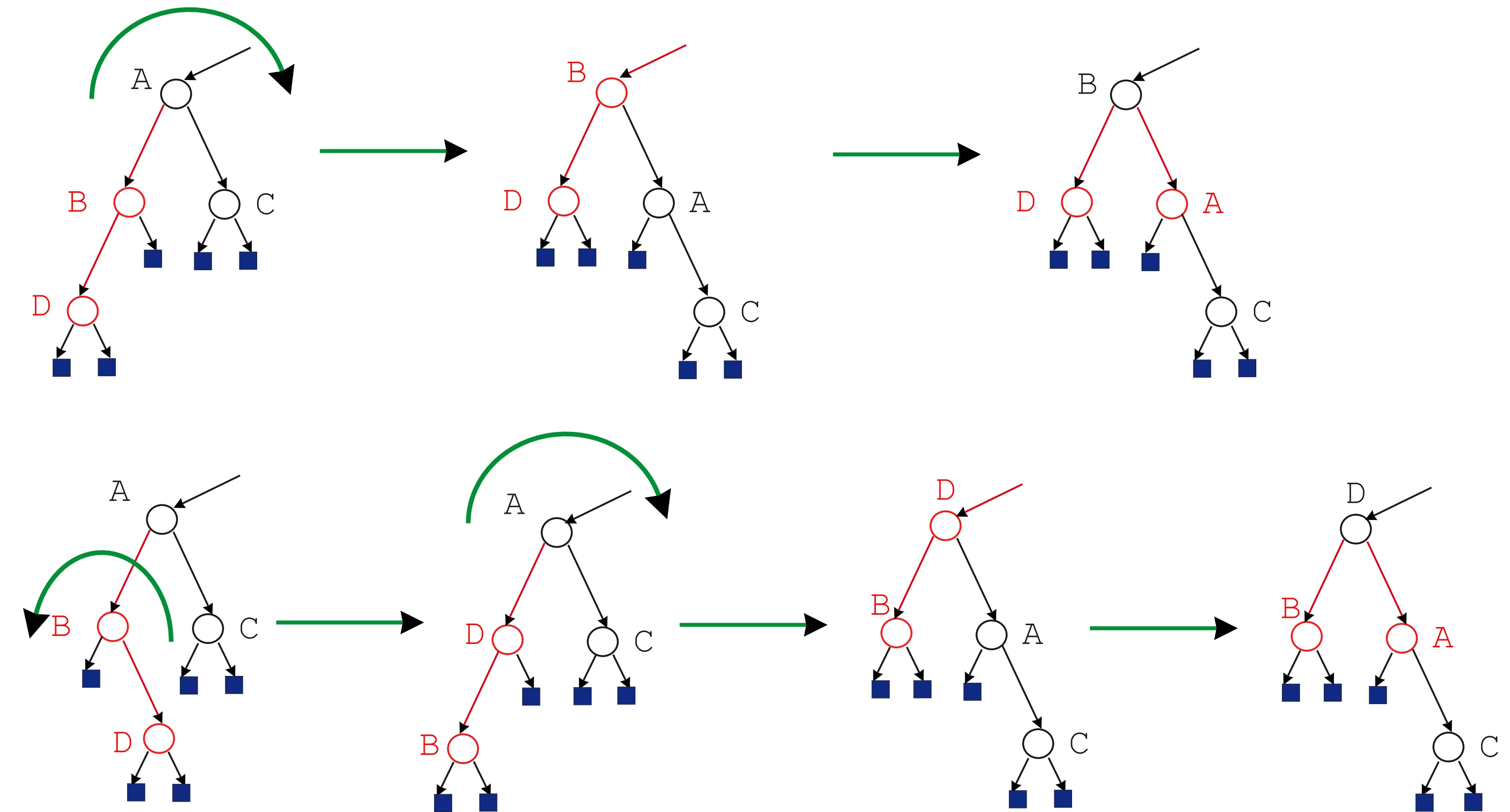
Kui C on punane, tuleb lihtsalt A, B ja C värvid vahetada.




Puna-must puu (5)
A värvi vahetus võib aga omakorda anda ebaõige puu. Sel juhul tuleb korrastamist jätkata kuni juureni välja
15 kui esialgu punane tipp ei saa omada nüüd juba punast tütart 13. Seega tuleb ta värv vahetada.
Pärast seda on juurest parempoolse haru välistippudesse 3 musta kaart, vasakpoolse haru välistippudesse aga 4. Seega tuleb muuta tipp 31 punaseks.
41
31
72
15
38
60
87
13
20
32
39
51
67
90
10
14
42
9
41
31
72
15
38
60
87
13
20
32
39
51
67
90
10
14
42
9

Puna-must puu (6)

Kui C on must, toimub nii pööramine kui ka värvide vahetus.



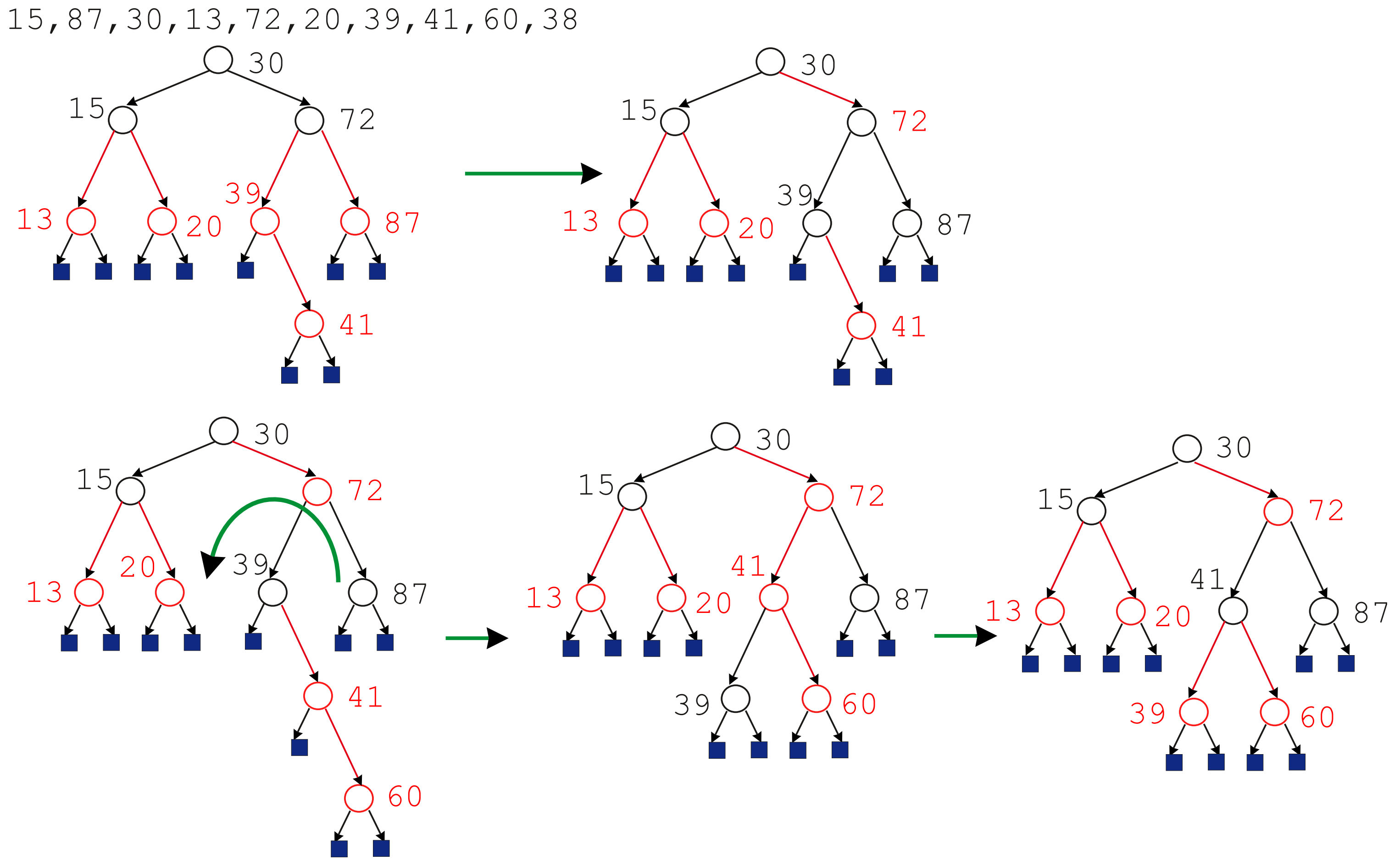
Puna-must puu (7)

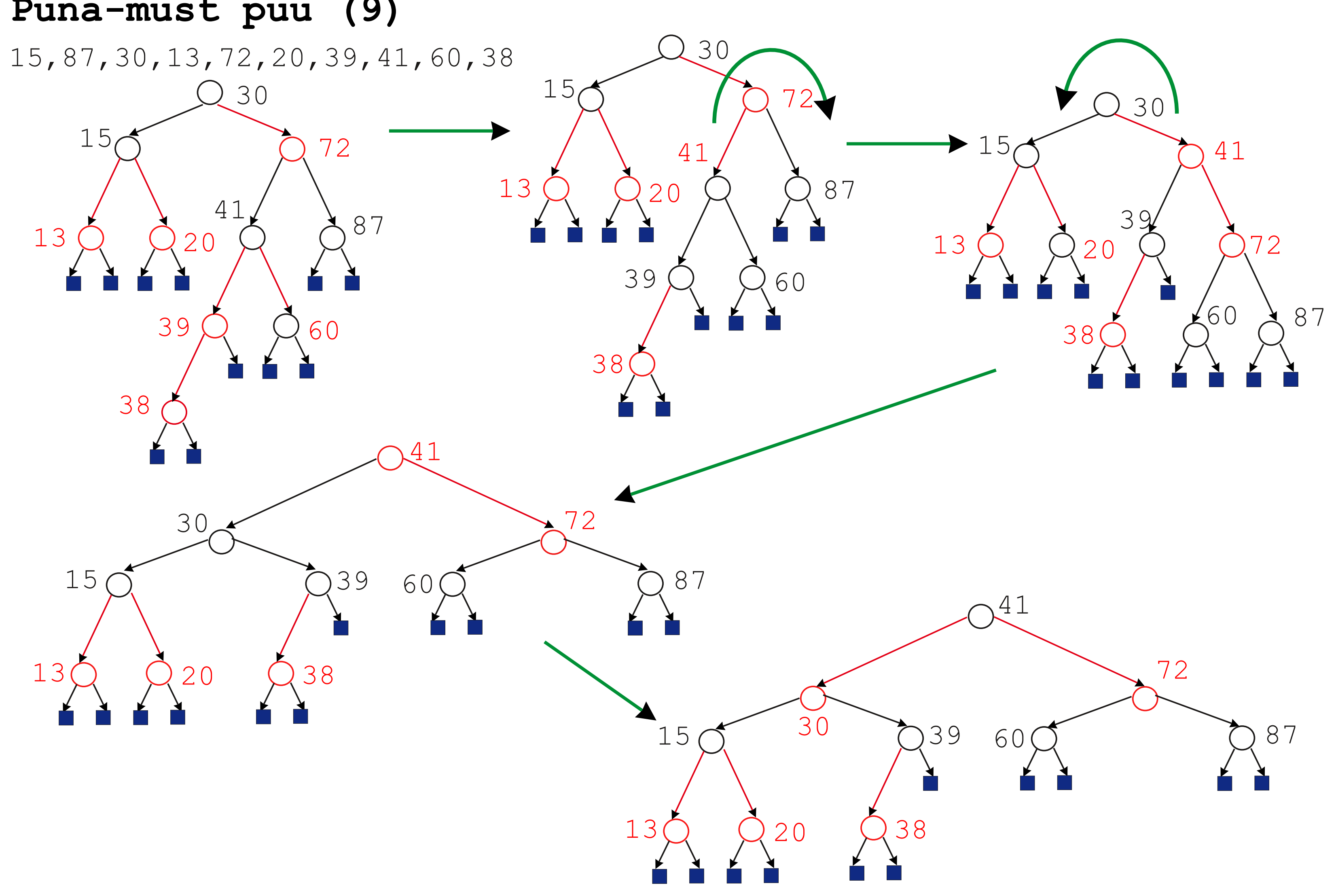
15,87,30,13,72,20,39,41,60,38

Juur jääb mustaks!

Puna-must puu (8)

15,87,30,13,72,20,39,41,60,38





Puna-must puu (9)
15,87,30,13,72,20,39,41,60,38
30
15
72
13
20
41
87
39
60
38
30
15
72
41
13
20
87
39
60
38
30
15
41
13
20
39
72
38
60
87
41
30
72
15
39
60
87
13
20
38
41
72
30
15
39
60
87
13
20
38

Puna-must puu (10)

Puna-mustast puust otsimine on logaritmiline protsess:

$T(n) \leq 2*\log_2 n+2$

Kui n on küllalt suur, siis

$T(n) \approx 1.002*\log_2 n$

Ülevaate puna-mustast puust leiab:

https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/

https://www.geeksforgeeks.org/red-black-tree-set-2-insert/

https://www.geeksforgeeks.org/c-program-red-black-tree-insertion/

https://www.geeksforgeeks.org/red-black-tree-set-3-delete-2/

Samast leiab ka vastavate C-funktsioonide koodid.

Puna-must puu ja AVL puu on praktiliselt sama head. Kumba neist eelistada on pigem maitse küsimus.

Primaarsed ja sekundaarsed võtmed

Võtmete kohta on üksnes 2 üldist nõuet:

- võtit saab kas otseselt või kaudselt seostada konkreetse kirjega
- kahe võtme kohta peab saama määrata, kas nad on võrdsed ja kui ei, siis kumb on suurem ja kumb väiksem.

Kui mingis andmehulgas leidub 2 või enam kirjet, mille võtmed langevad kokku, siis puudega opereerimiseks tuleb sisse tuua sekundaarsed võtmed, s.t. mingid muud kirjega seotud parameetrid. Nii näiteks kui võtmeks on inimese nimi, siis samanimeliste isikute puhul võib lugeda väiksemaks noorema isiku võtit.
Kui andmehulgas on täiesti identseid kirjeid, tuleb puu tippu lisada täiendav väli, mis sisaldab korduste arvu.

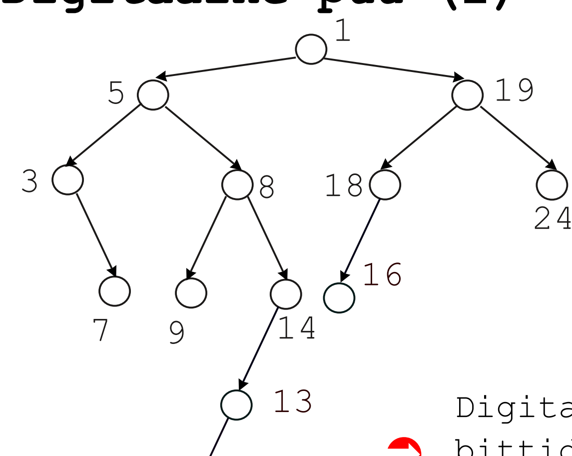
Teine rühm puudel rajanevaid andmestruktuure lähtub aga mitte abstraktsest võtmest vaid lahutab võtme komponentideks. Nii näiteks võib 4-baidist täisarvu vaadelda kui 32-st bitist koosnevat jada, 16-st kahebitisest segmendist koosnevat jada jne. Stringi käsitletaks sümbolite jadana. Puu ehitamisel võrreldakse mitte võtmeid tervikuna vaid võtmete komponente.
Selliste puude (radix search trees) puhul peavad võtmed olema unikaalsed.

Digitaalne puu (digital search tree) (1)

1	00001
19	10011
5	00101
18	10010
3	00011
8	01000
9	01001
14	01110
7	00111
24	11000
13	01101
16	10000
12	01100

0-bit: vasakule, 1-bit: paremale.
Järjekordset tippu paigutades loeme üksteise järel bitte kuni jõuame tühjale kohale. Rohkem bitte välja lugeda pole vaja, sest koht tipu jaoks on käes.

Digitaalne puu (2)

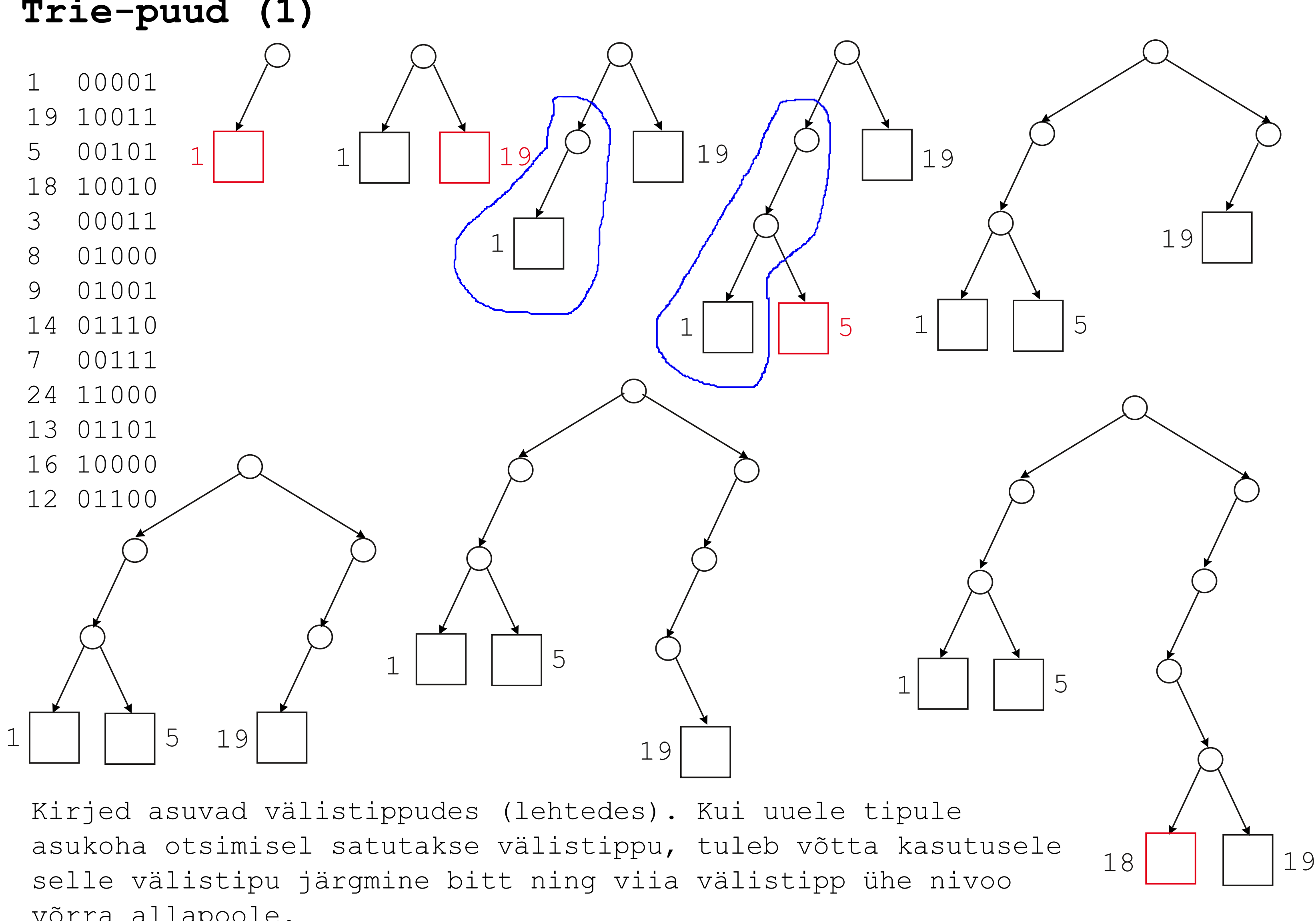


Kui võtmeteks on täisarvud, mis ei ületa suurust 2^n, siis puu kõrgus ei saa ületada arvu n+1. S.t kui näiteks võtmed on tüübist short int, siis max 65536 kirjet mahub puusse kõrgusega kuni 17. Seega on digitaalne puu üsna hästi tasakaalus.

Digitaalsest puust otsimisel liigutakse küll lähtuvalt bittidest, kuid igas tipus tuleb teha ka täielik võtmete võrdlus.

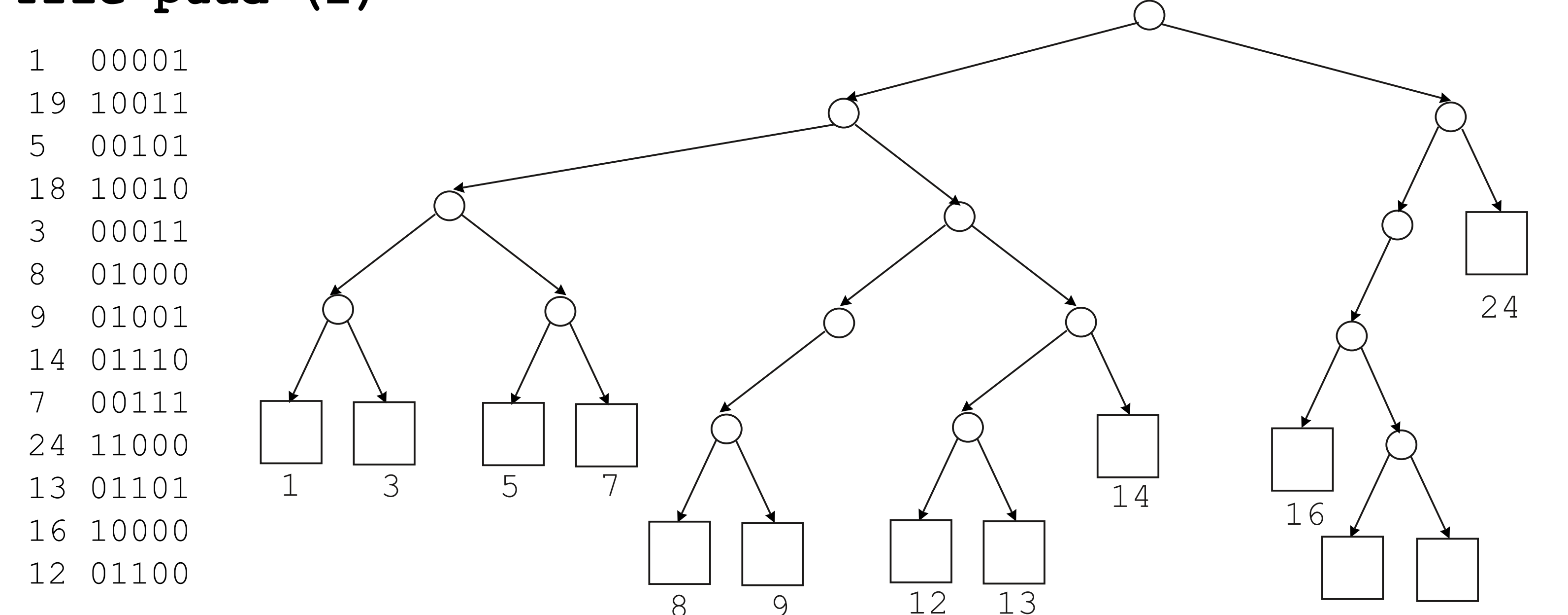
Kui võtmeteks on stringid, saab neid vaadelda kui sümbolite jadasid ja ehitada puu lähtudes sarnasest loogikast. Siin on aga eelduseks, et kõik stringid oleksid ühe ja sama pikkusega.

Trie-puud (1)



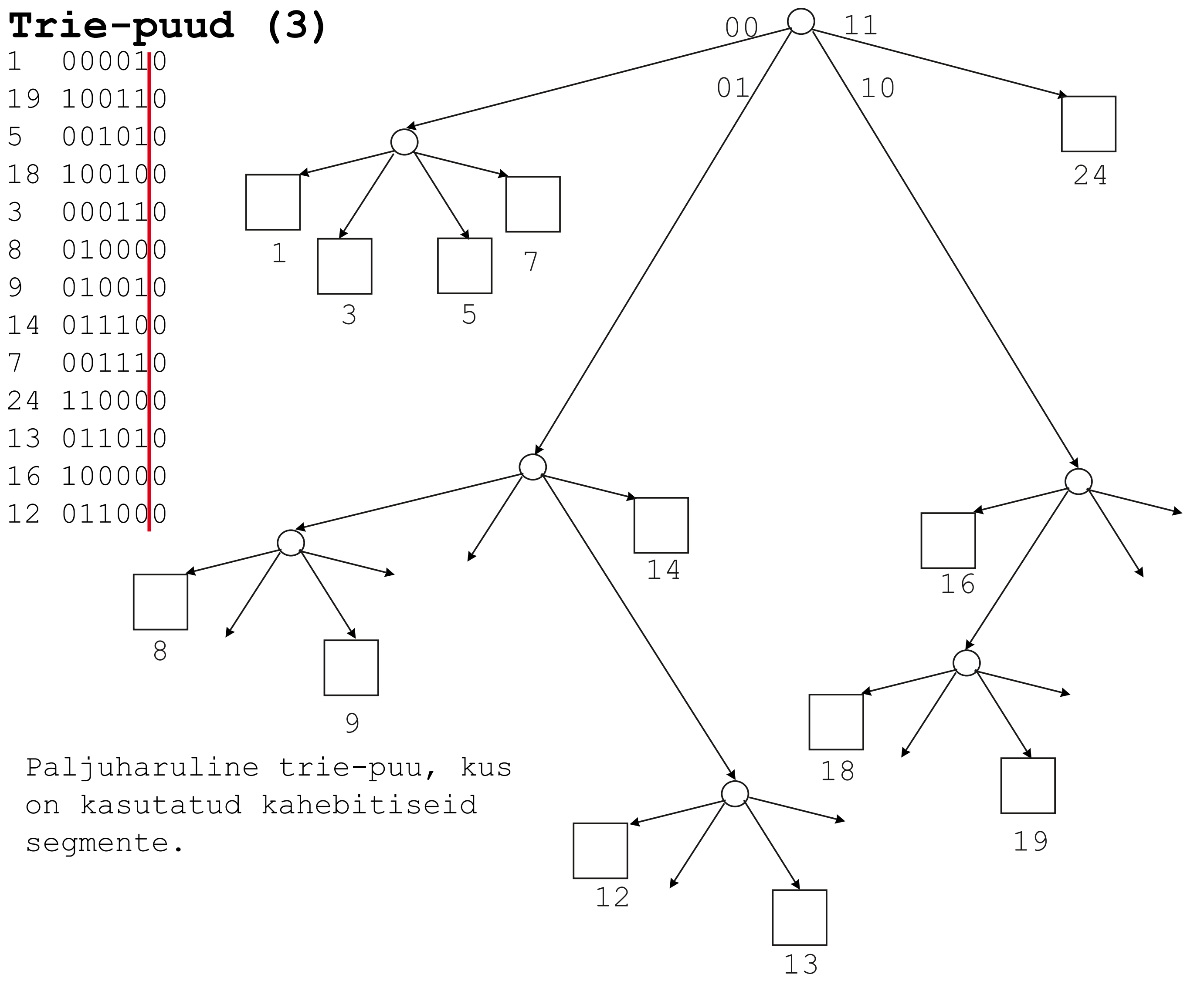
Kirjed asuvad välistippudes (lehtedes). Kui uuele tipule asukoha otsimisel satutakse välistippu, tuleb võtta kasutusele selle välistipu järgmine bitt ning viia välistipp ühe nivoo võrra allapoole.

Trie-puud (2)



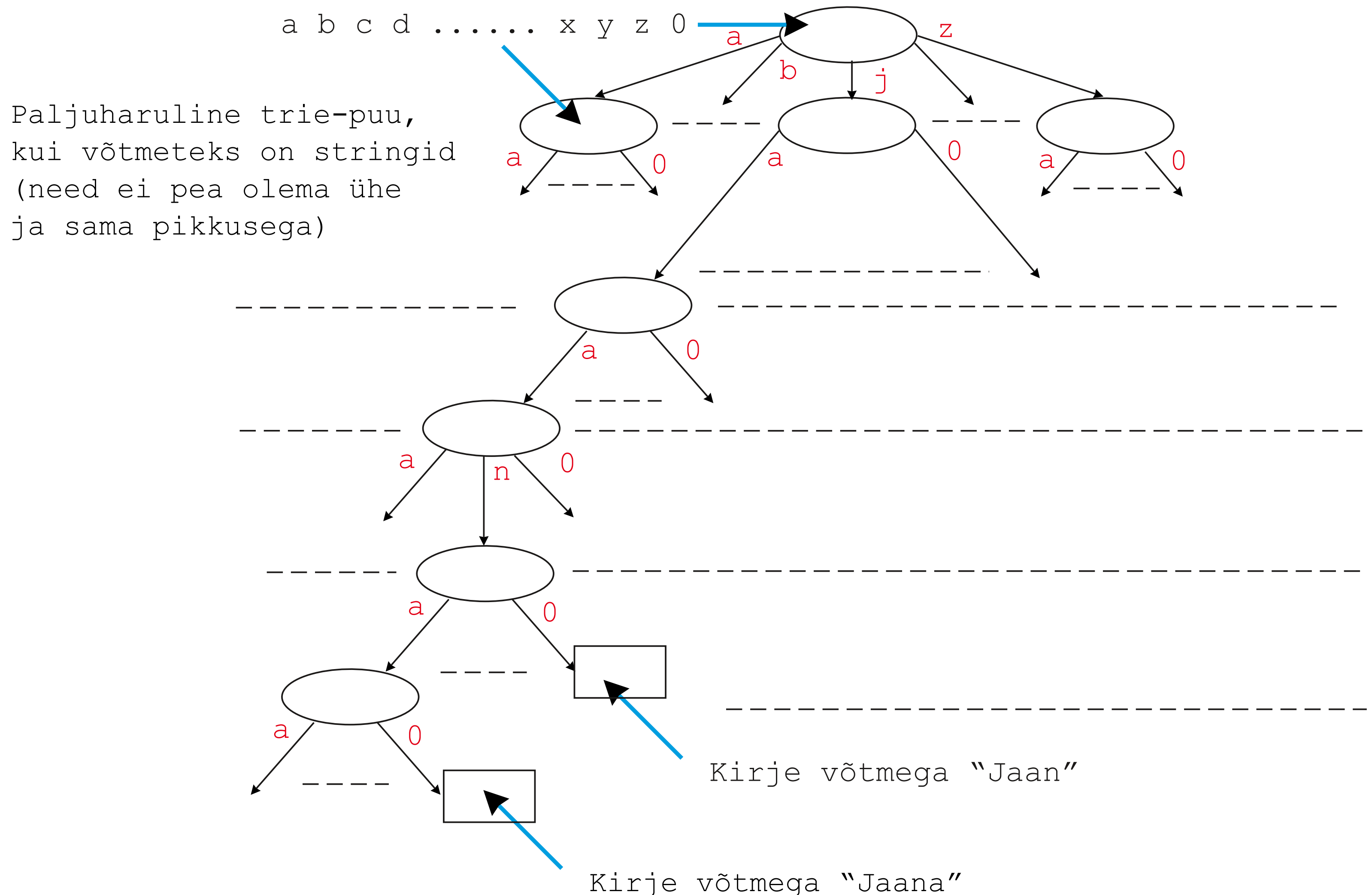
Kui võtmeteks on täisarvud, mis ei ületa 2^n, siis puu kõrgus (ilma välistippudeta) ei ületa n.
Puust otsimisel liigume küll lähtuvalt bittidest, kuid välistippu jõudes tuleb teha täielik võtmete võrdlus.

Trie-puud (3)

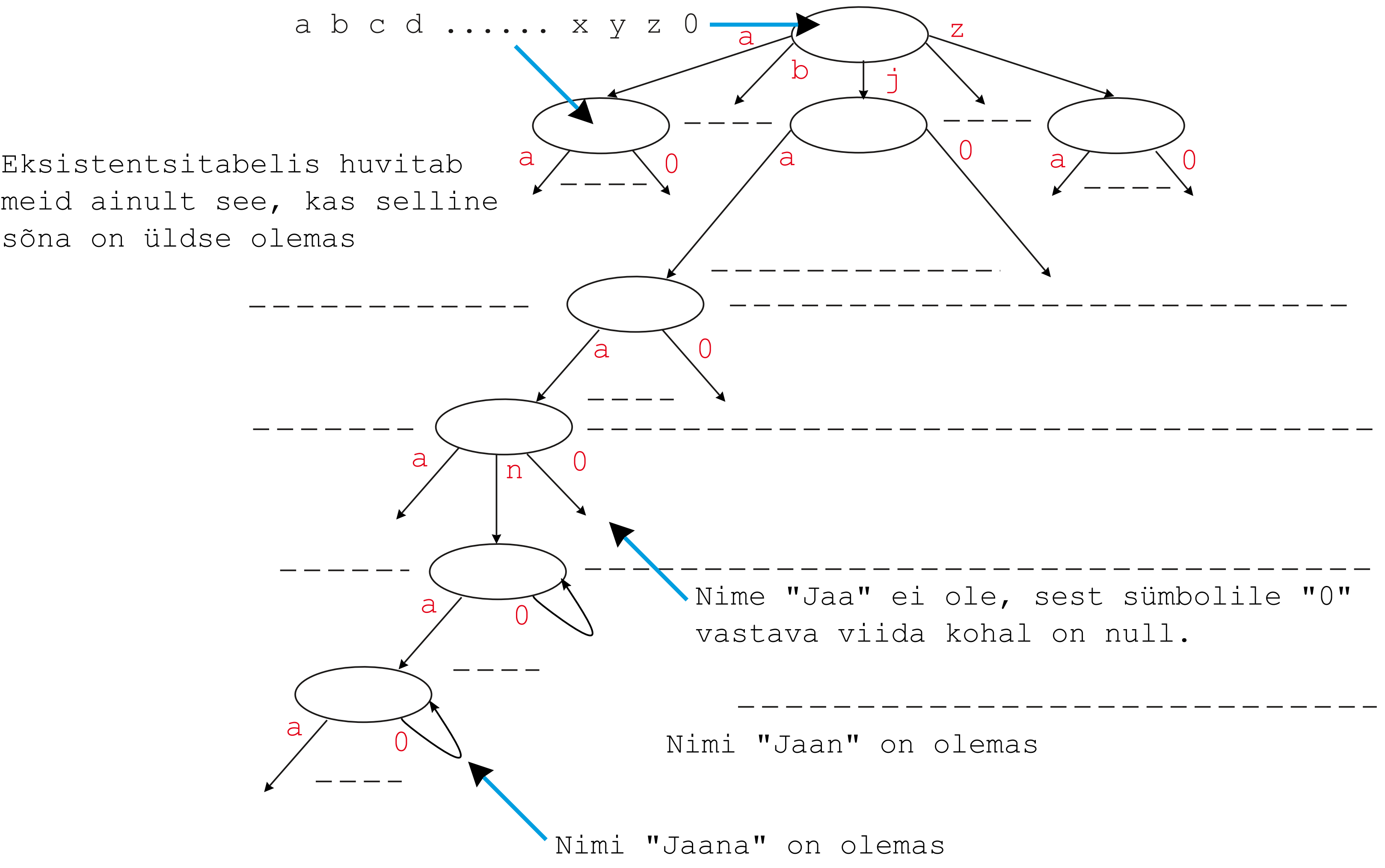


Paljuharuline trie-puu, kus on kasutatud kahebitiseid segmente.

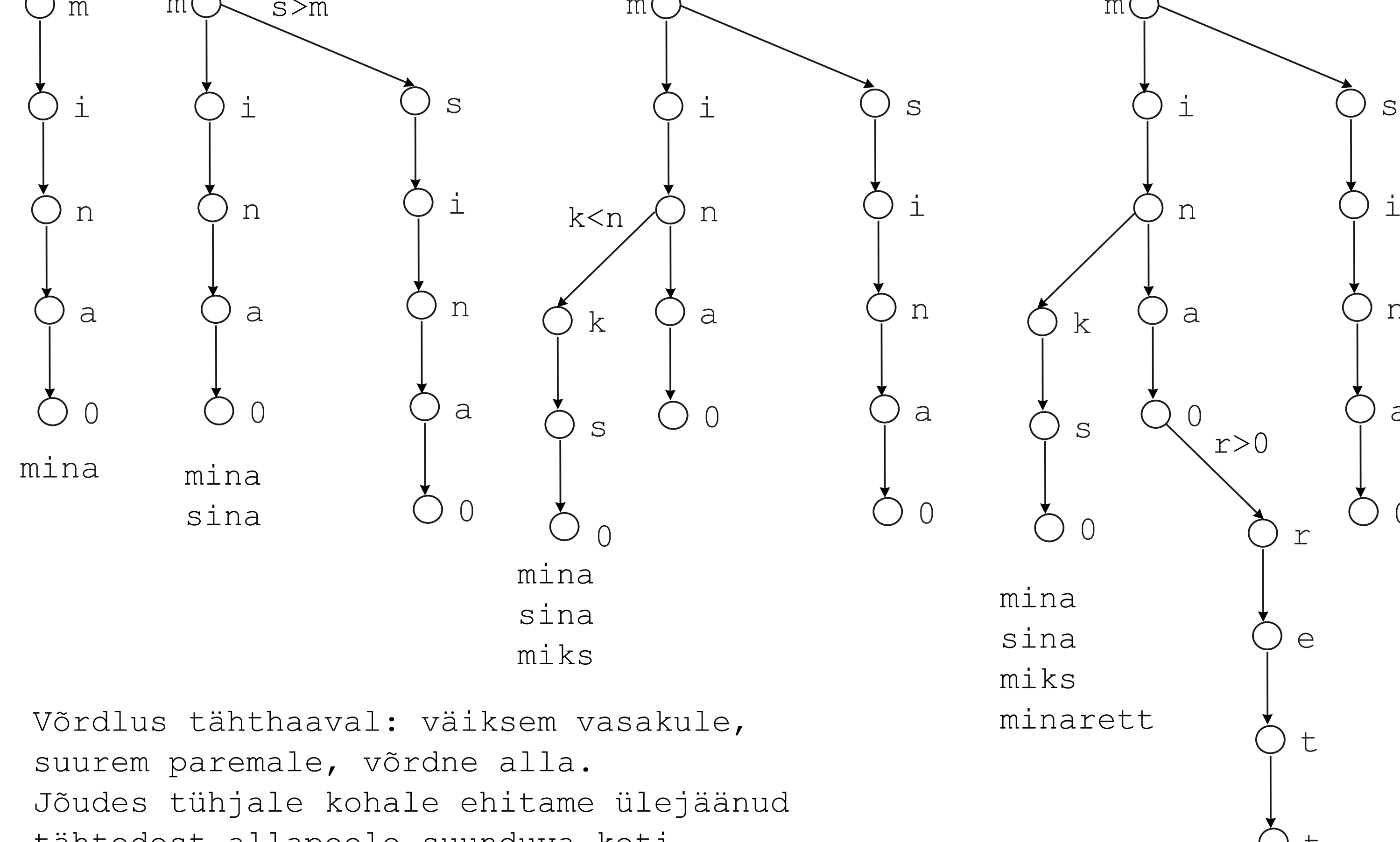
Trie-puud (4)



Trie-puud (5)

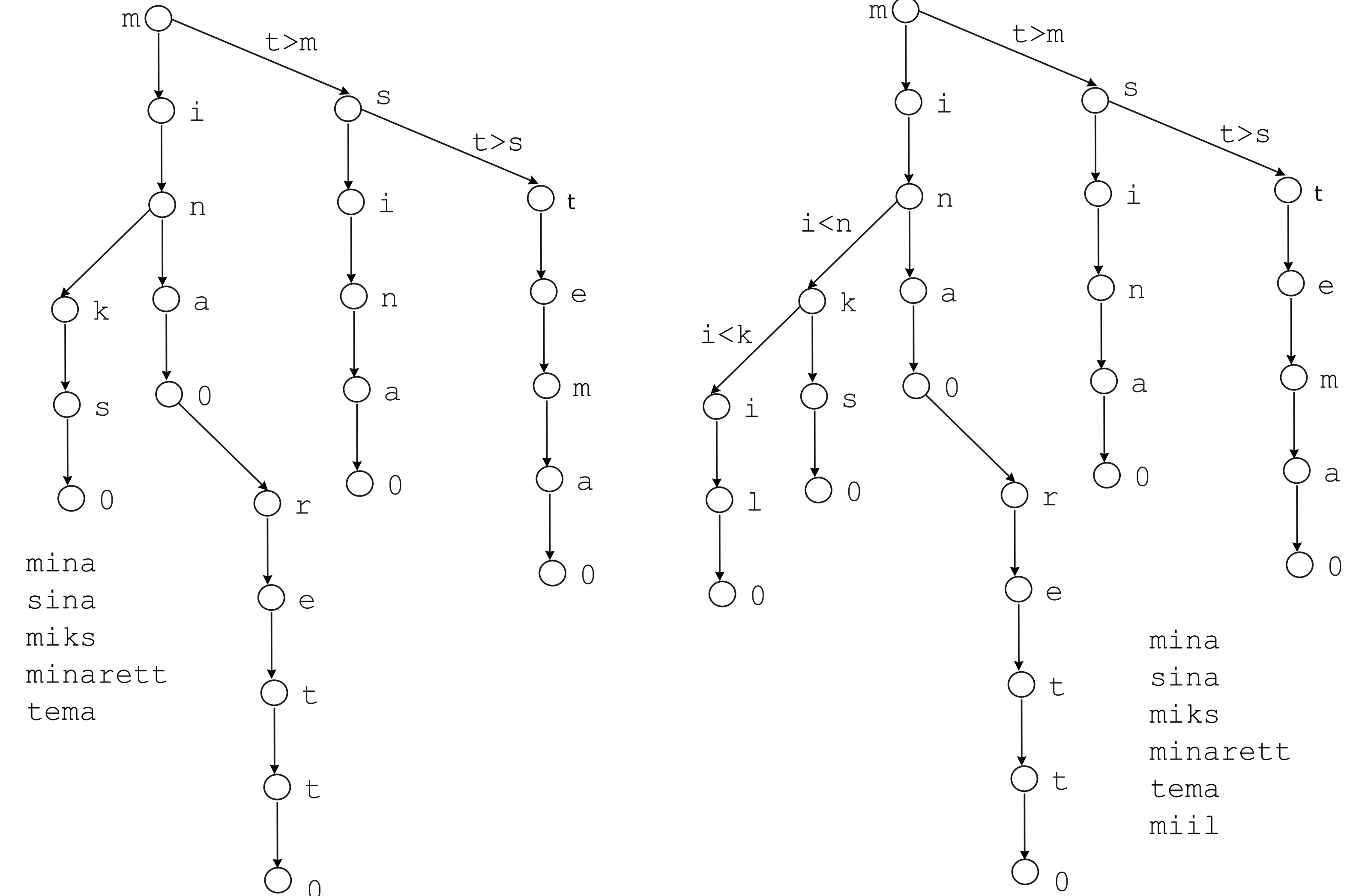


Kolmikpuud (ternary search trees) (1)

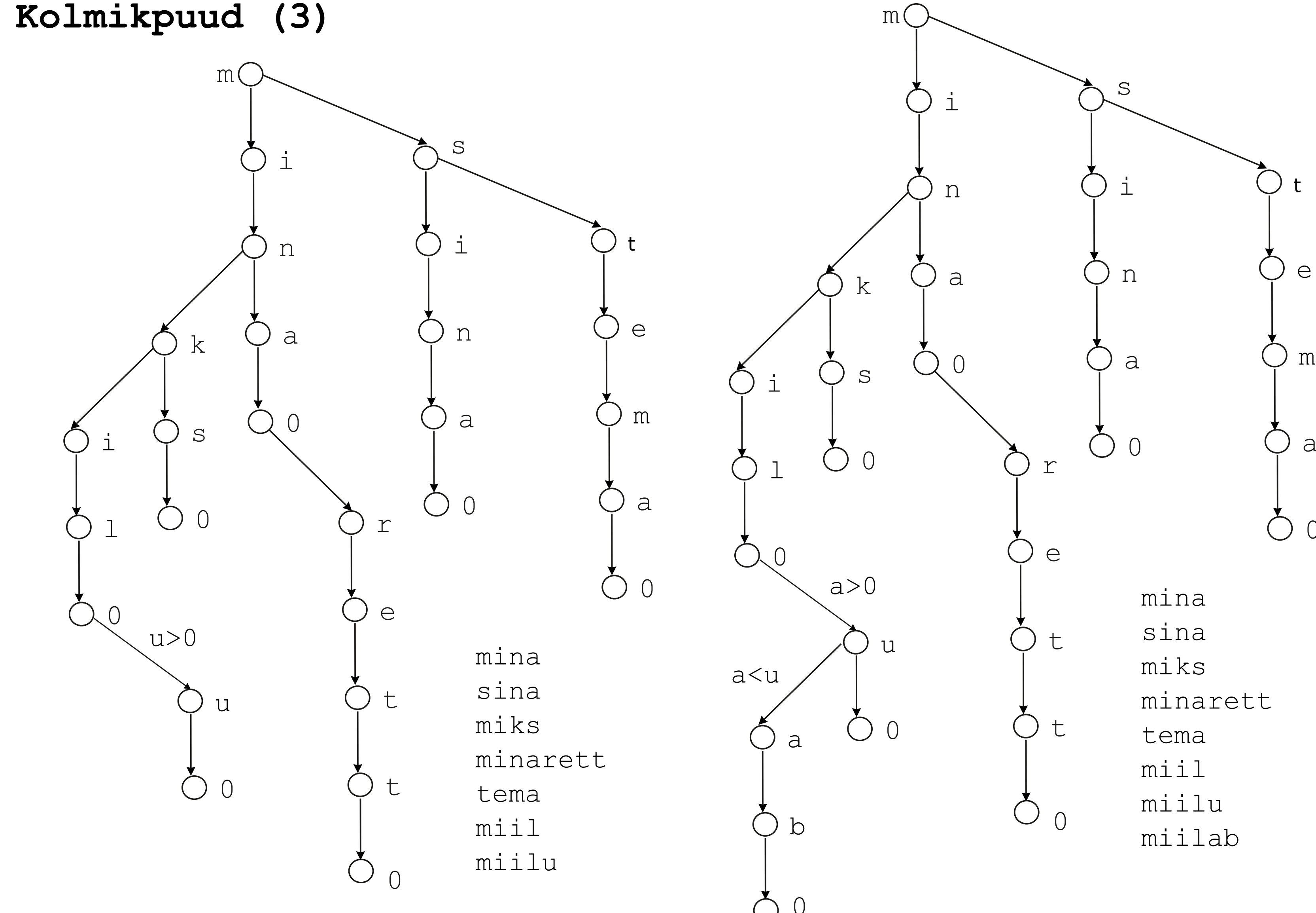


Võrdlus tähthaaval: väiksem vasakule, suurem paremale, võrdne alla.
Jõudes tühjale kohale ehitame ülejäänud tähtedest allapoole suunduva keti.

Kolmikpuud (2)



Kolmikpuud (3)



Kolmikpuud (4)

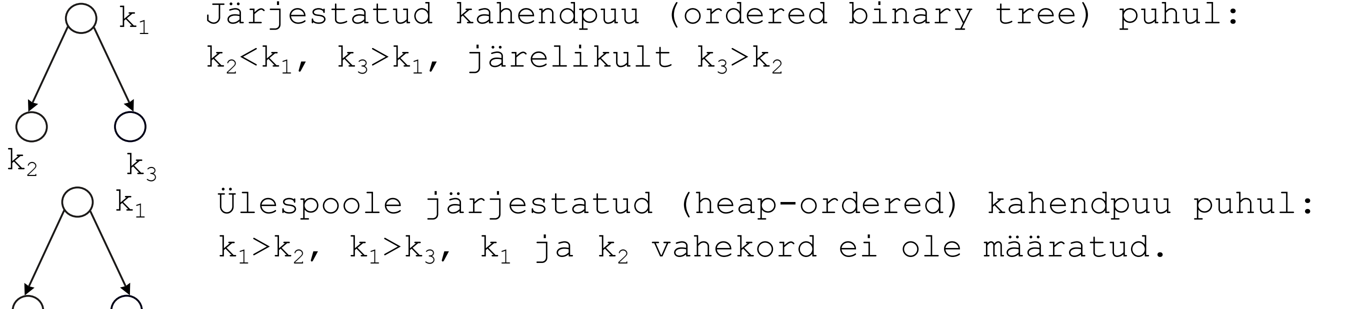
Võrreldes trie-puul baseeruva eksistentsitabeliga on kolmikpuu suureks eeliseks see, et tema tippudes on ainult kolm viita. Samuti ei nõua kolmikpuu, et tähestik oleks ette teada: kui tegu on ühebaidiste sümbolitega, siis on kõik 256 märki lubatud. Lõpuks on kolmikpuu mälus kompaktsem.

Kolmikpuu kohta leiab lisateavet lehelt
https://www.geeksforgeeks.org/ternary-search-tree/
Samas on toodud ka vastavate C-funktsioonide koodid.

Prioriteetidega järjekord (priority queue)

Abstraktse andmetüübi "järjekord" puhul võis eemaldada ainult esimese kirje. Uus kirje tuli paigutada alati järjekorra lõppu.
Abstraktse andmetüübi "prioriteetidega järjekord" puhul on igal kirjel oma prioriteet. Nagu võtitki, nii ka prioriteeti peab saama kirjest välja lugeda. Samuti peab olema võimalik prioriteete võrrelda.
Prioriteetidega järjekord ei ole lineaarne, sest kirjed võivad paikneda ükskõik mis moel. Uue kirje võib paigutada ükskõik mis kohale. Eemaldada saab aga vaid kõige kõrgema prioriteediga kirjet.

Abstraktse andmetüübi "prioriteetidega järjekord" realiseerimiseks sobivad kõige paremini andmestruktuurid, milledest kõrgeima prioriteediga kirje leidmine on kõige hõlpsam.



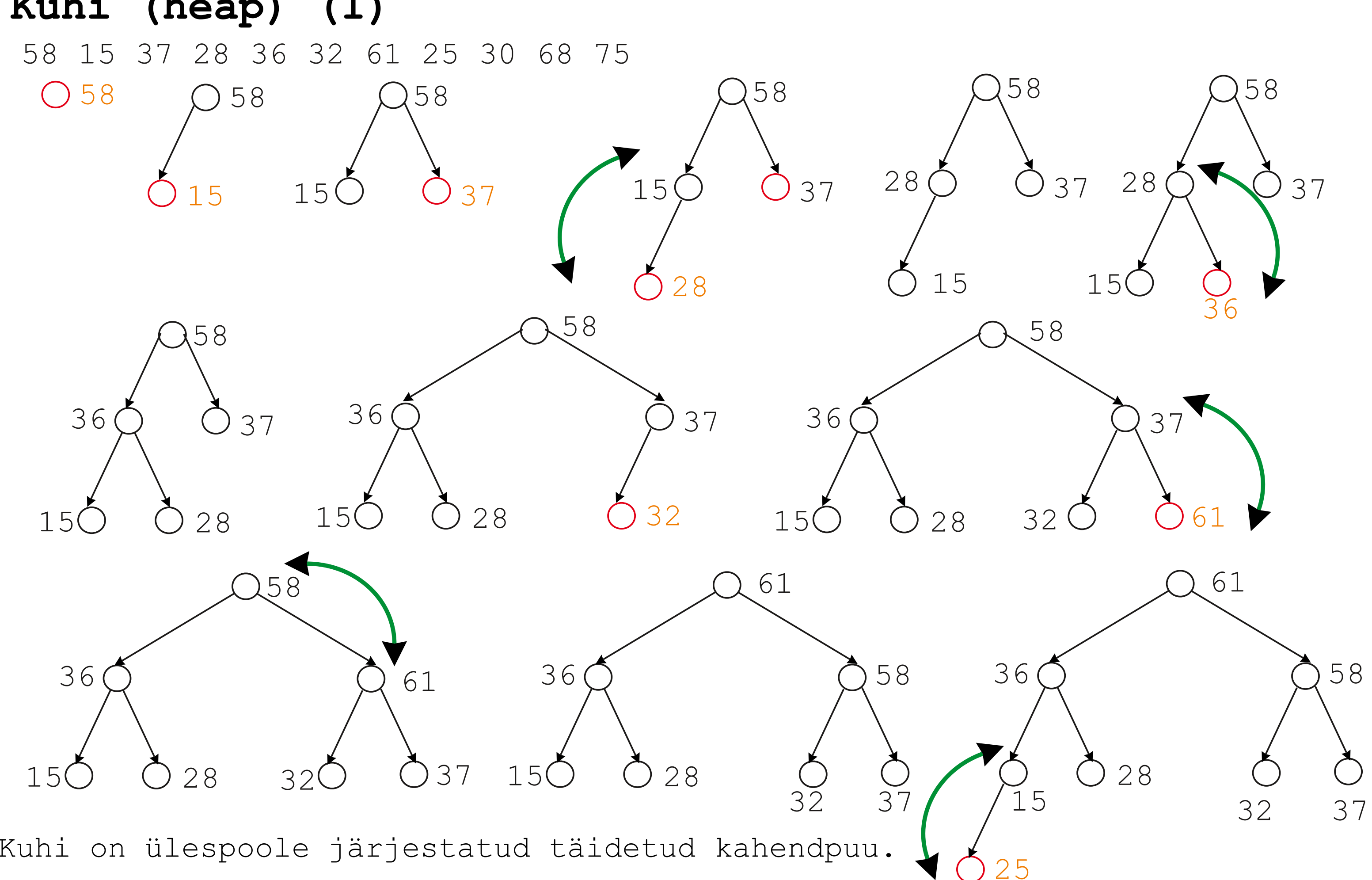
Järjestatud kahendpuu (ordered binary tree) puhul:
$k_2<k_1$, $k_3>k_1$, järelikult $k_3>k_2$

Ülespoole järjestatud (heap-ordered) kahendpuu puhul:
$k_1>k_2$, $k_1>k_3$, k_1 ja k_2 vahekord ei ole määratud.

Täidetud kahendpuu (complete binary tree) on kahendpuu, mis on:
- absoluutselt tasakaalus
- või kui ta ei ole, siis kõige alumise nivoo tippude eemaldamisel saame absoluutselt tasakaalus puu. Samuti peab viimane nivoo olema täidetud ühtlaselt vasakult paremale.

Kuhi (heap) (1)

58 15 37 28 36 32 61 25 30 68 75

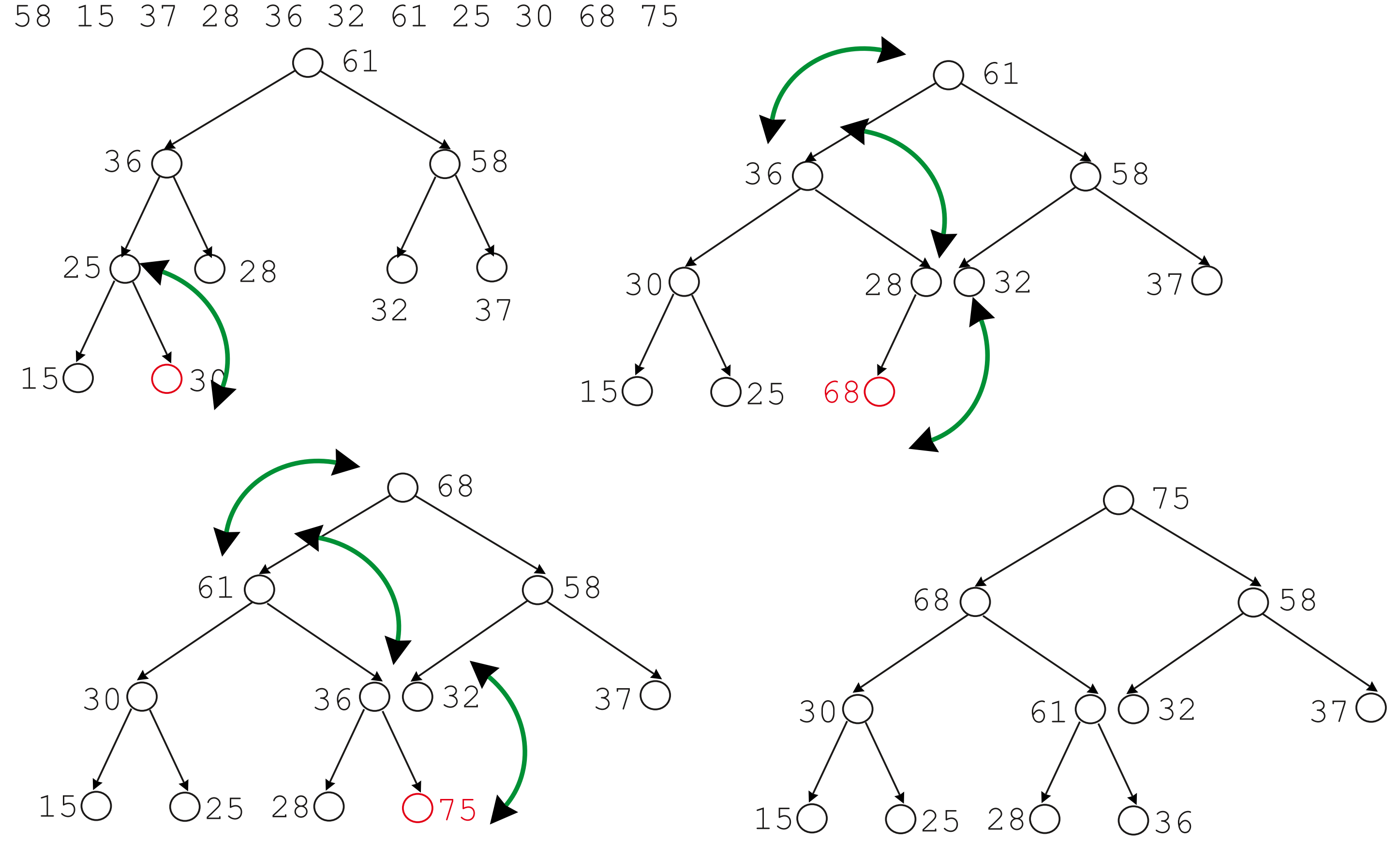


Kuhi on ülespoole järjestatud täidetud kahendpuu.

Kuhja ehitamisel lisatakse uued tipud kõige viimasele nivoole alates kõige vasakpoolsemast positsioonist. Kui ülespoole järjestatuse reegel pärast uue tipu lisamist on rikutud, teostatakse vajalikud vahetamised.

Kuhi (2)

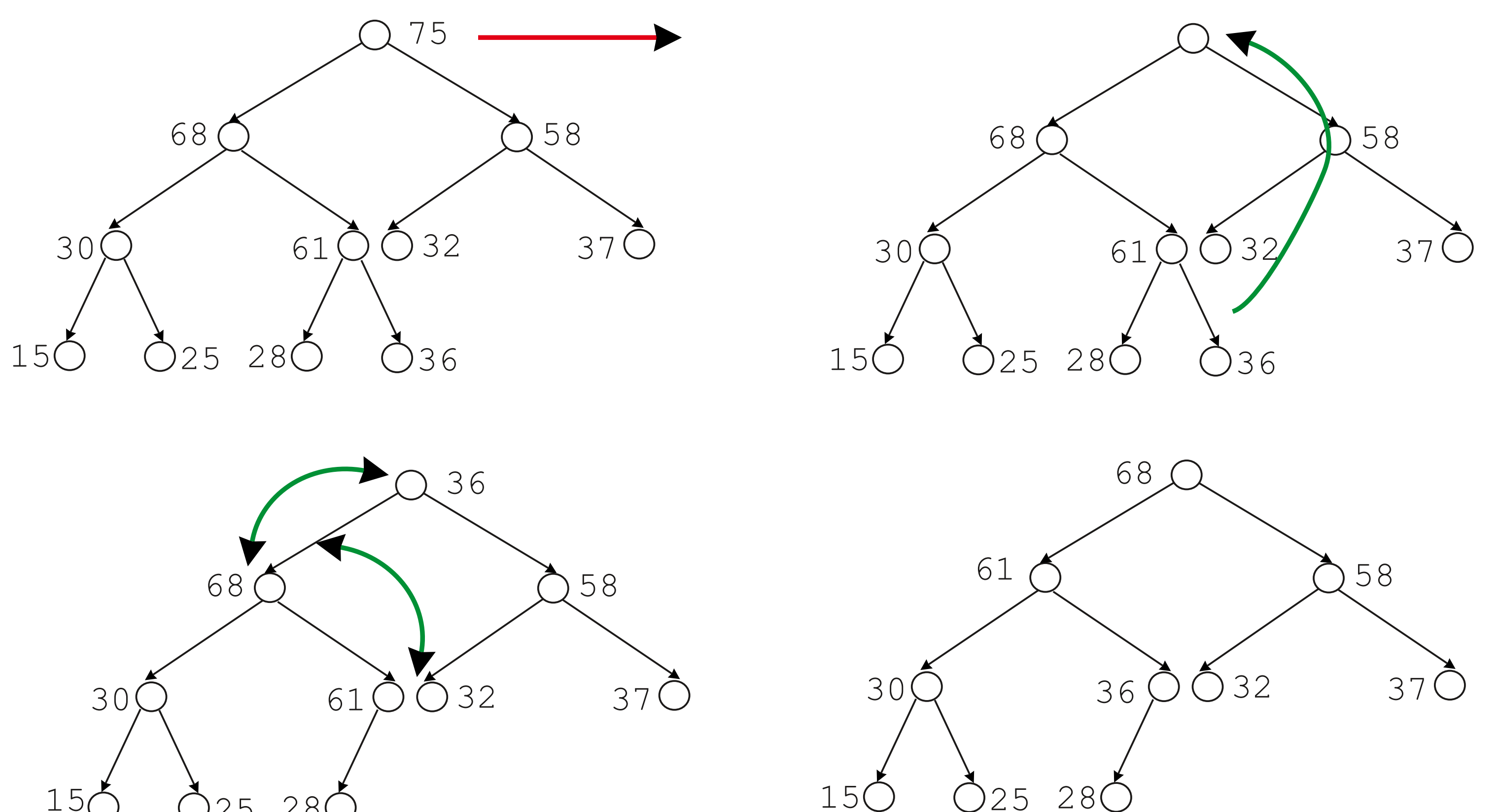
58 15 37 28 36 32 61 25 30 68 75



Suurima prioriteediga kirje on alati kuhja juureks

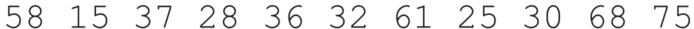
Kuhi (3)

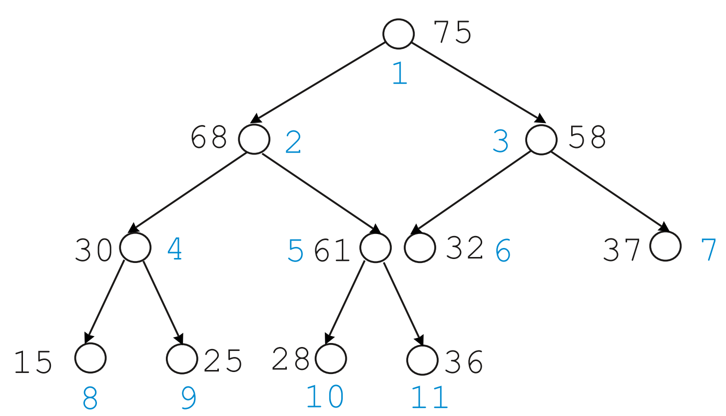
58 15 37 28 36 32 61 25 30 68 75



Pärast suurima prioriteediga kirje eemaldamist toome kõige alumise nivoo parempoolseima kirje uueks juureks ja siis korrastame puu

Kuhi (4)





Ülespoole järjestatud täidetud kahendpuu on küll paberil atraktiivne, kuid tema realiseerimine arvutis on üsna ebameeldiv, sest vajalikud on viidad ka tütardelt emale ja õelt õele.

Käime puu läbi nivooti:

75 68 58 30 61 32 37 15 25 28 36

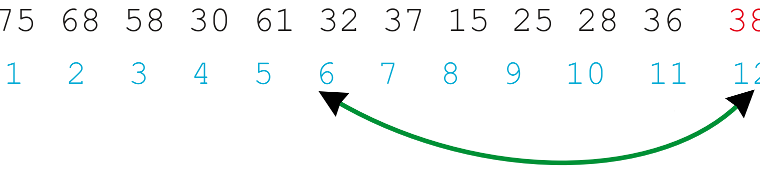
1 2 3 4 5 6 7 8 9 10 11

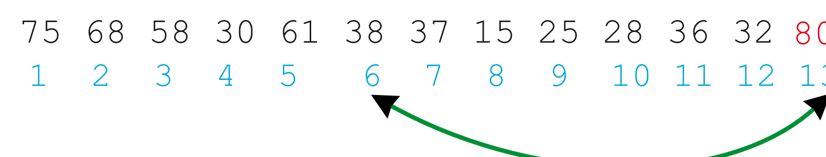
Puu omadustest järgneb, et selles massiivis, kui $2i<n$ ja $2i+1<n$ ja $i/2>0$, siis:

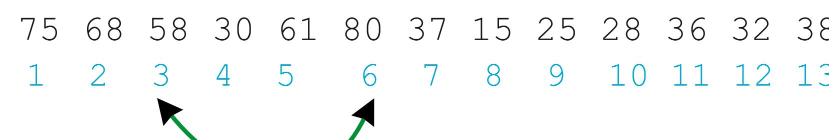
- $a_i > a_{2i}$
- $a_i > a_{2i+1}$
- $a_i < a_{i/2}$

Seda massiivi nimetatakse samuti kuhjaks

Kuhi (5)







80 68 75 30 61 58 37 15 25 28 36 32 38

1 2 3 4 5 6 7 8 9 10 11 12 13

Kui $2i<n$ ja $2i+1<n$ ja $i/2>0$, siis:

- $a_i > a_{2i}$
- $a_i > a_{2i+1}$
- $a_i < a_{i/2}$

Pärast iga lisamist kontrollime, kas need tingimused on täidetud. Eitava vastuse korral teeme vajalikud vahetused.

Suurima prioriteediga kirje on kuhja esimesel positsioonil.

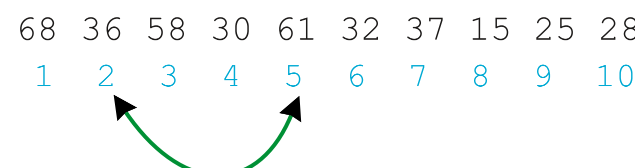
Kuhi (6)

Kui $2i<n$ ja $2i+1<n$ ja $i/2>0$, siis:

- $a_i>a_{2i}$
- $a_i>a_{2i+1}$
- $a_i<a_{i/2}$

Pärast suurima prioriteediga kirje eemaldamist toome viimase kirje esimeseks ja siis vahetame asukohti seni kuni saavutame reegleid täitva kuhja.

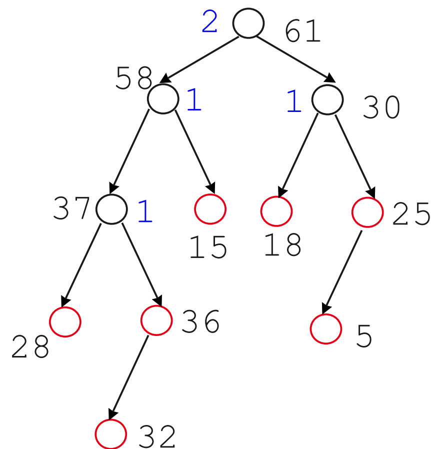
36 68 58 30 61 32 37 15 25 28
1 2 3 4 5 6 7 8 9 10



68 61 58 30 36 32 37 15 25 28
1 2 3 4 5 6 7 8 9 10

Vasakule kaldus puu (leftist tree) (1)

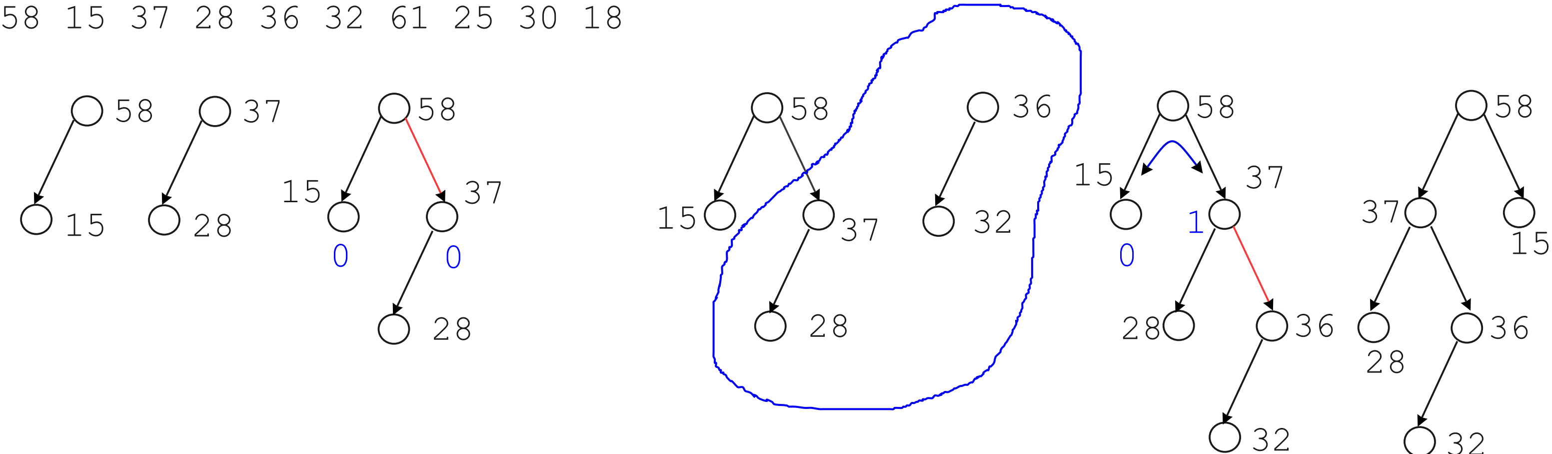
Siinkohal nimetame välistipuks iga tippu, millel on vähem kui 2 tütartippu.
Kahe tipu vahelise teekonna pikkus on võrdne sellel teekonnal läbitavate kaarte arvuga.
Mingi tipu kaugustegur on määratud teekonna pikkusega sellest tipust kuni temale lähima välistipuni. Välistipu kaugustegur on 0.



Vasakule kaldus puu on kahendpuu, mille puhul:
1. Ematipu võti on alati suurem kui tema tütartippude võtmed (s.t. puu on ülespoole järjestatud).
2. Kui ematipul on 2 tütart, siis vasaku õe kaugustegur on kas parempoolse õe kaugustegurist suurem või siis temaga võrdne.

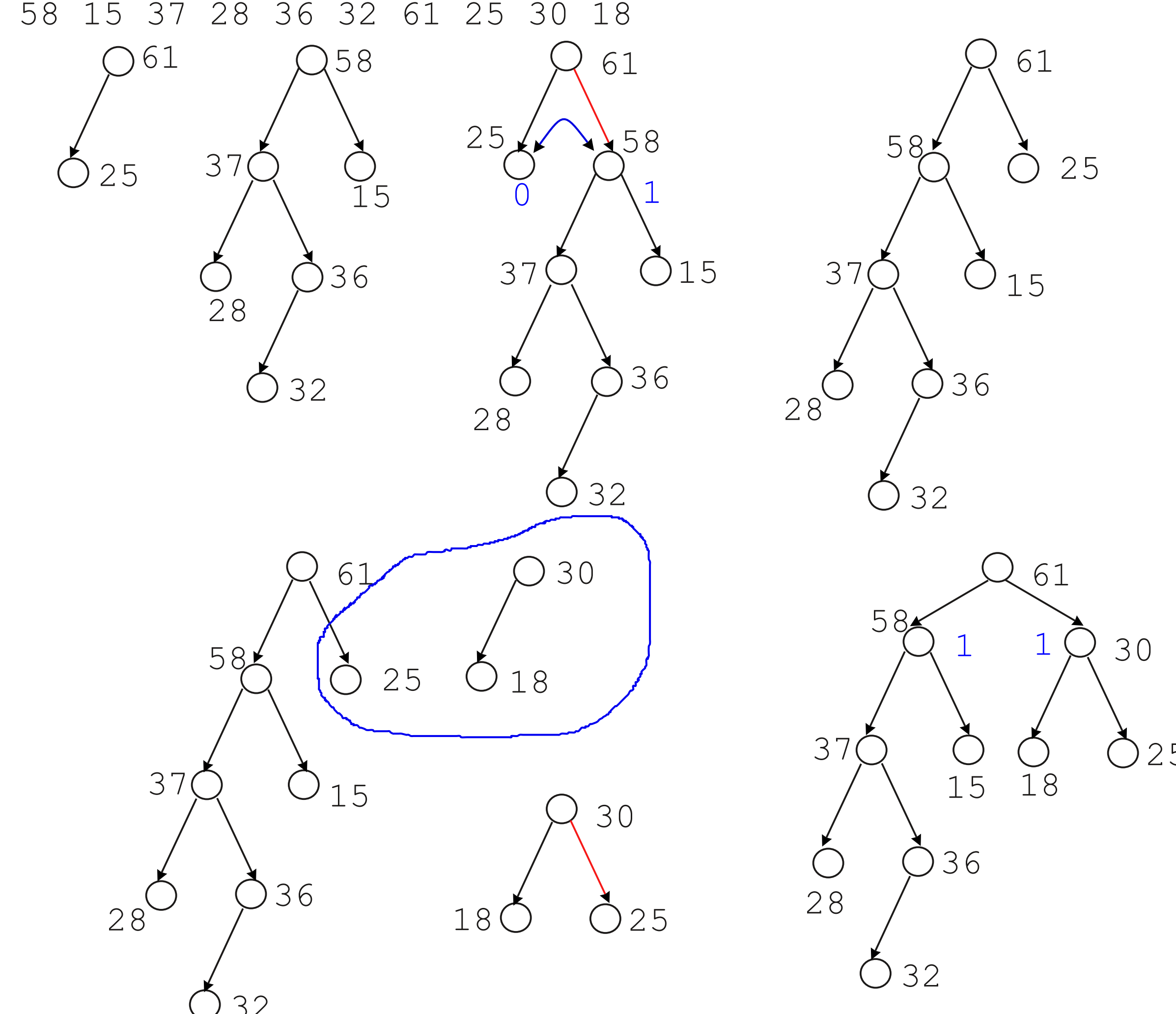
Vasakule kaldus puu (2)

58 15 37 28 36 32 61 25 30 18

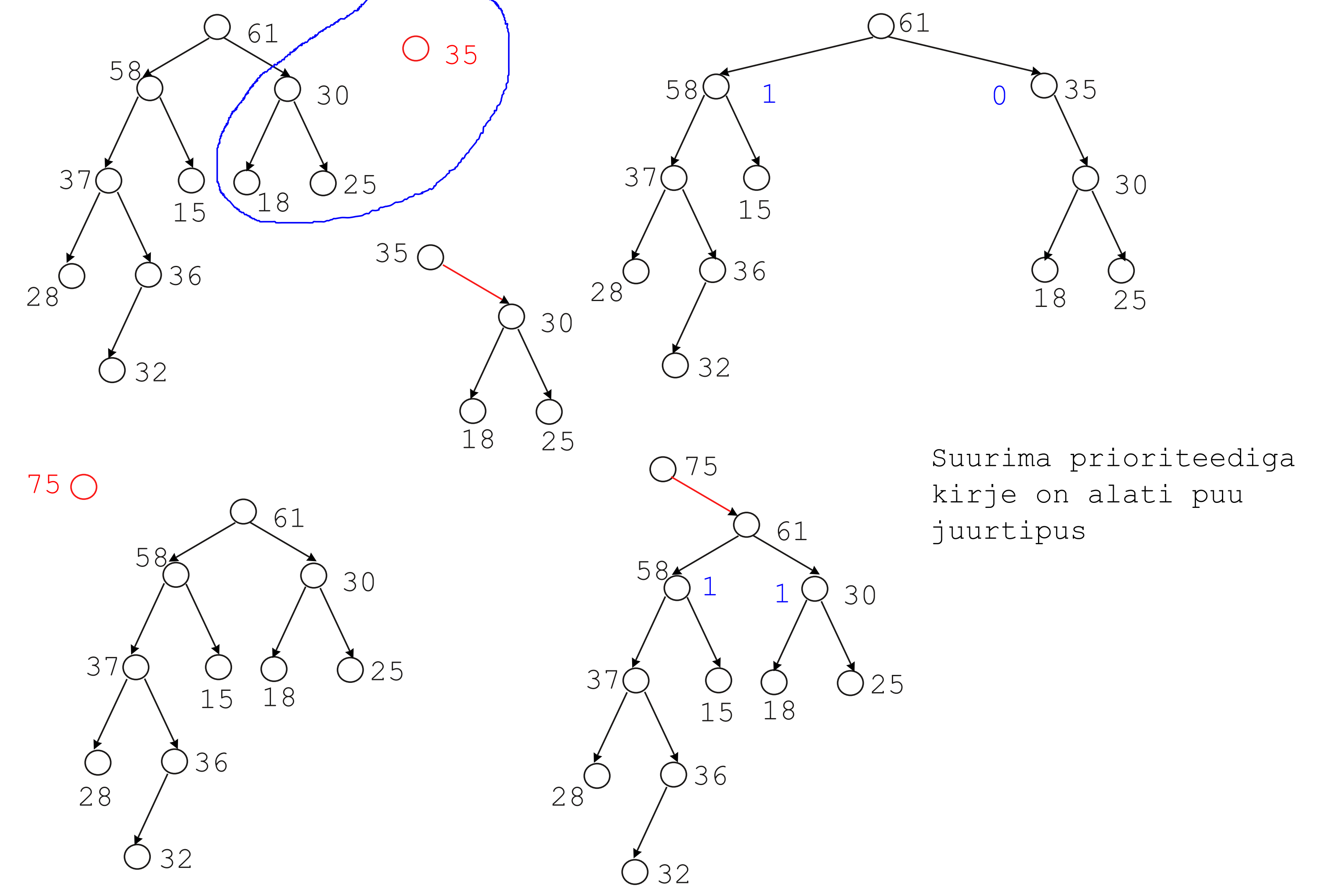


Olgu ühendatavad puud A ja B, kusjuures A juure võti olgu suurem kui B juure võti. Siis B ühineb A parema haruga.
Kui puul A paremat haru polegi, tuleb lihtsalt tõmmata kaar A juurtipust B juurtippu.
Vastasel korral tuleb käsitleda A paremat haru kui iseseisvat puud, millega tuleb mestida puu B. Reeglid on samad. Seega mestimise protsess on rekursiivne.
Pärast mestimist tuleb leida uued kaugustegurid. Kui viimaste vaheline seos on rikutud, tuleb harud vahetada.

Vasakule kaldus puu (3)

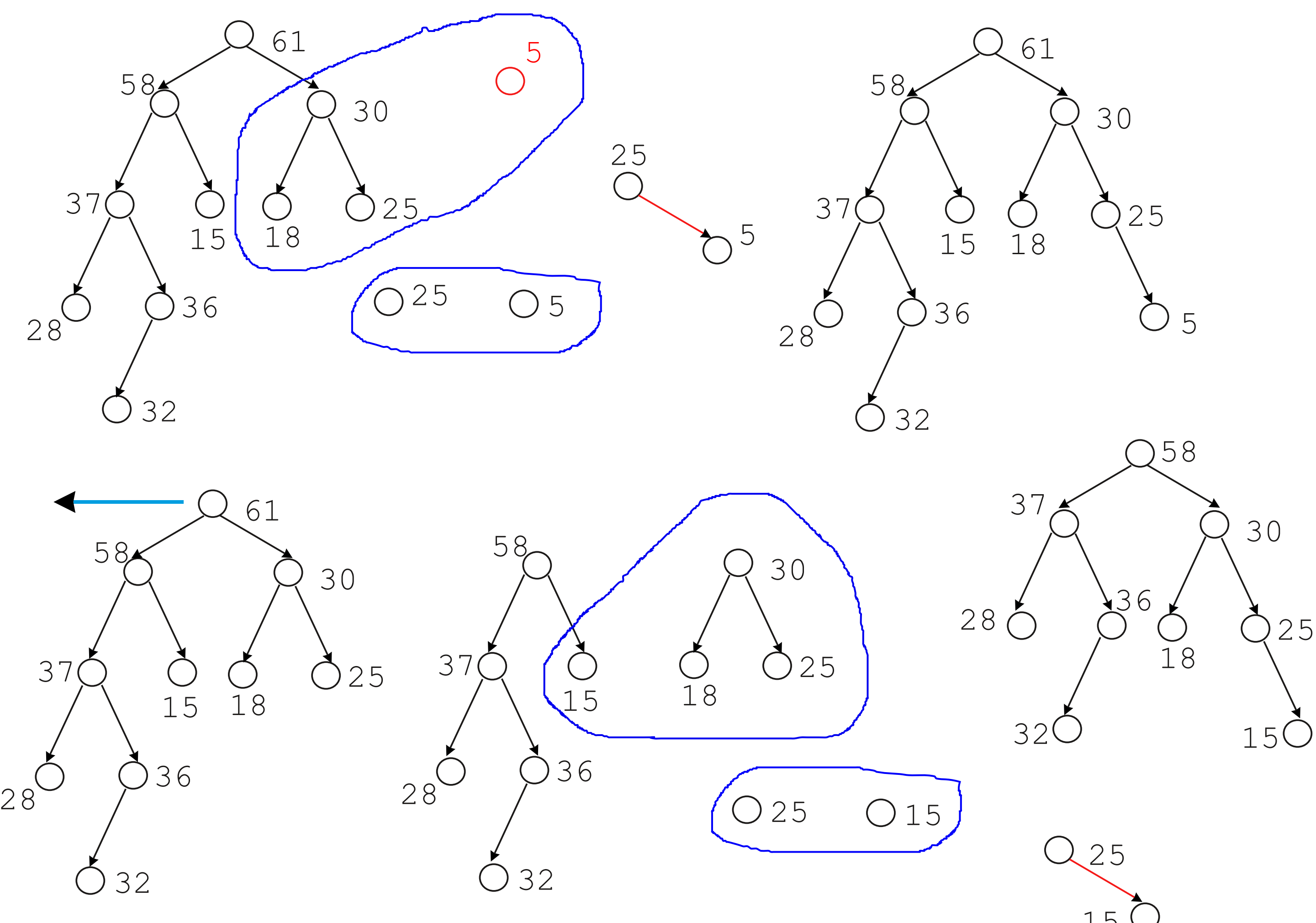


Vasakule kaldus puu (4)



Suurima prioriteediga kirje on alati puu juurtipus

Vasakule kaldus puu (5)

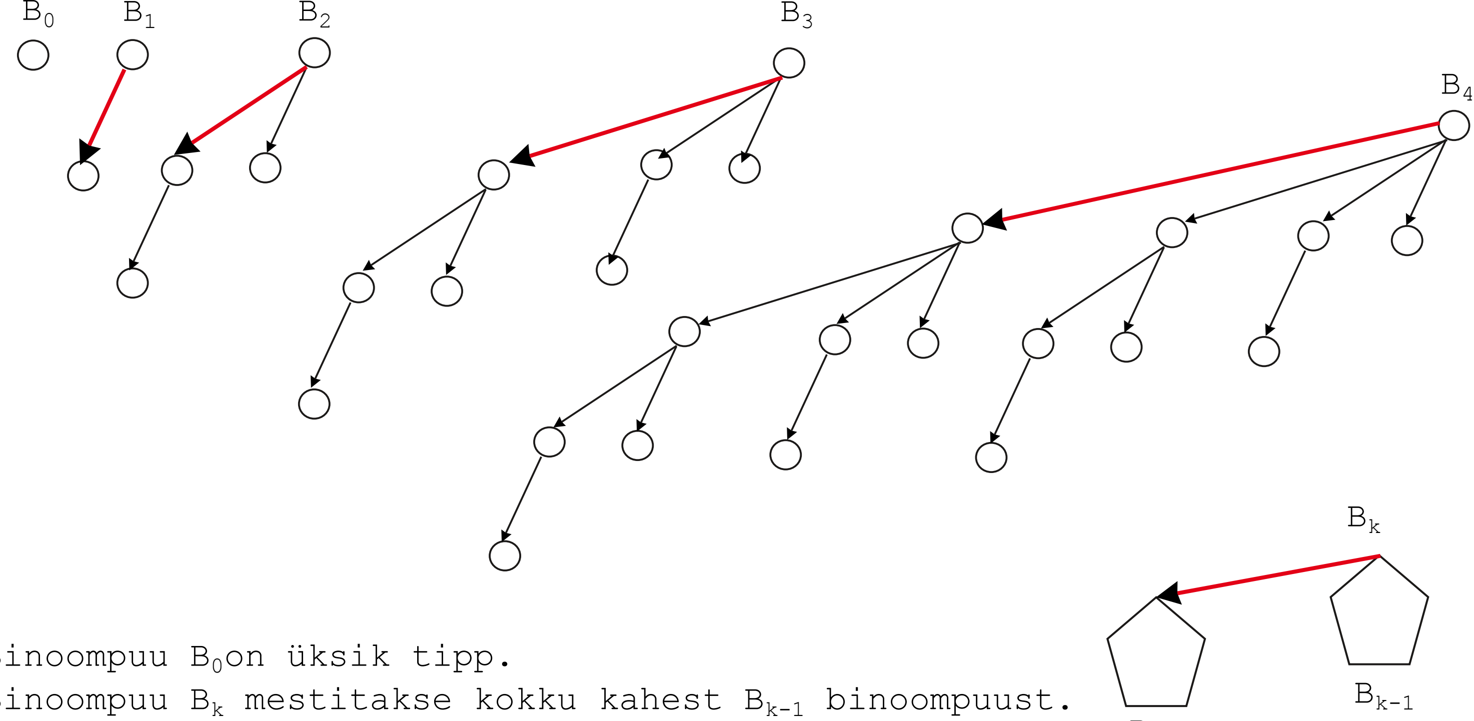


Vasakule kaldus puu (6)

Vasakule kaldus puul on kuhjaga võrreldes mitmed eelised:
- tegemist on kahendpuuga, millel on ainult viidad emalt tütardele
- kuna ta ei ole massiiv, siis uute kirjete lisamine ei tekita probleeme

Täiendavat infot leiab aadressilt:
https://www.geeksforgeeks.org/leftist-tree-leftist-heap/
Samas on toodud ka vajalike C-funktsioonide koodid.

Binoompuu (binomial tree)

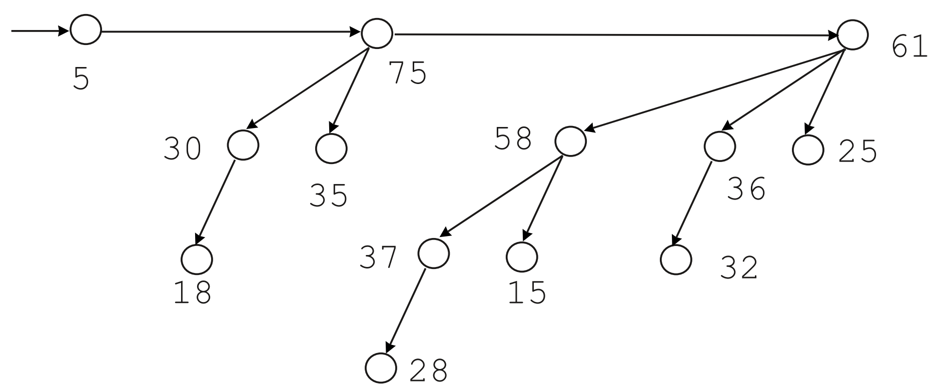


Binoompuu B_0 on üksik tipp.
Binoompuu B_k mestitakse kokku kahest B_{k-1} binoompuust.

Binoompuu B_k kõrgus on k+1 ja tippude arv 2^k.

Nivool i on $\binom{k}{i} = \frac{k!}{(k-i)!\ i!}$ tippu (binoomkoefitsient)

Binomiaalne kuhi (binomial heap) (1)



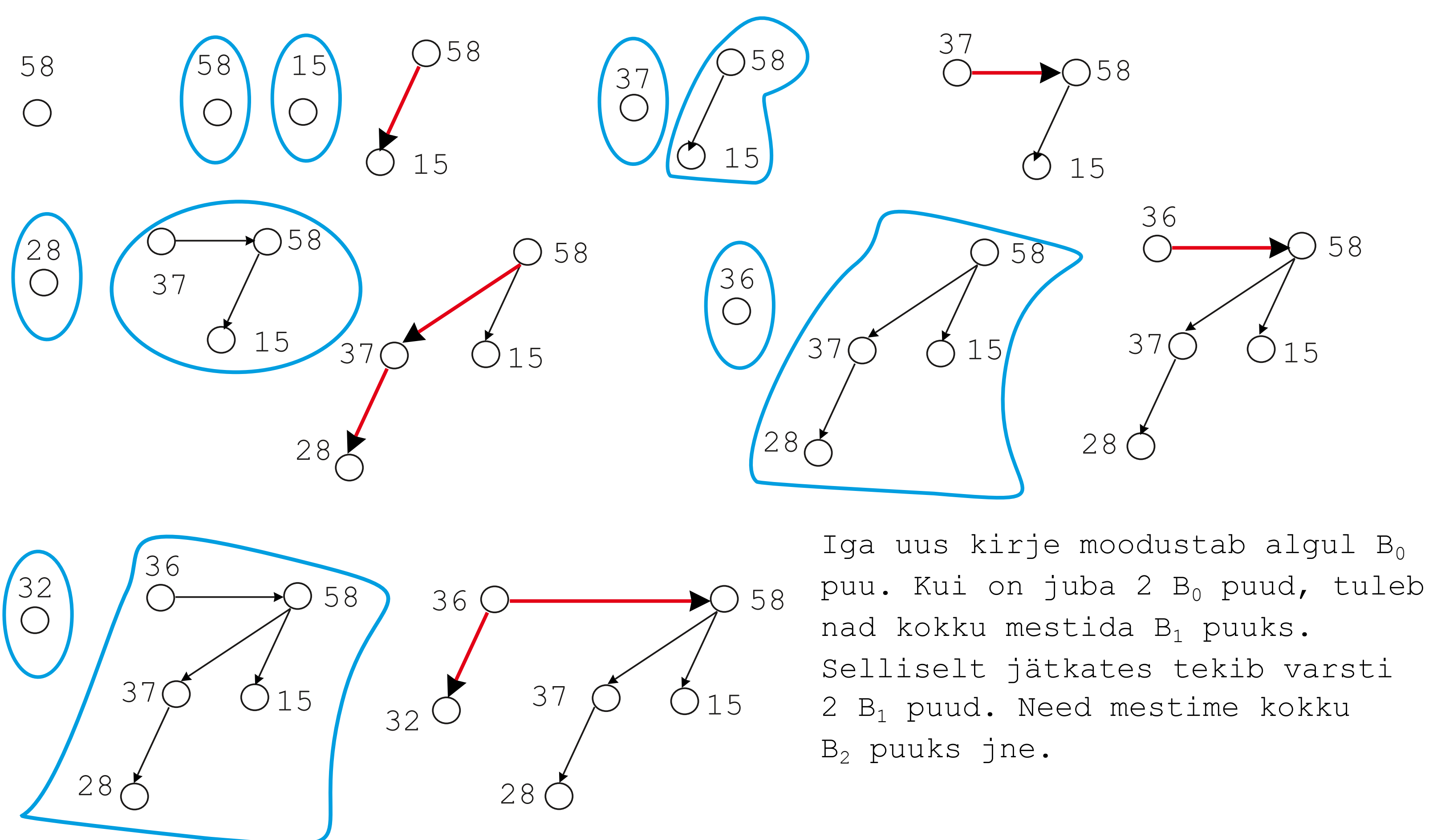
Binomiaalne kuhi on binoompuude hulk. Seejuures mistahes k puhul binoompuu B_k saab hulgas esineda kas ainult üks kord või siis hoopis puududa.
Hulgas olevad binoompuud peavad olema ülespoole järjestatud,s.t.mistahes ema-tütar paari puhul on ematipu võti suurem.

Koosnegu täisarvu n esitus mälus i bitist: $\{b_{i-1},b_{i-2},...b_0\}$, kus $i=\log_2 n+1$
Siis $n=b_0*2^0+b_1*2^1+...+b_{i-1}*2^{i-1}$
Näiteks $13=1101_2$ja $13=1*2^0+0*2^1+1*2^2+1*2^3$
Et binoompuu B_i omab 2^i tippu, siis n kirje esitamiseks läheb vaja binomiaalset kuhja, kus on maksimaalselt $\log_2 n+1$ binoompuud.
B_i esinemine või puudumine kuhjast sõltub sellest, kas vastav bitt n esitusest kahendsüsteemis on 1 või 0.

Kõrgeima prioriteediga kirje asub ühe binoompuu juures. Tema leidmiseks tuleb järjestikuse otsimise meetodil kõik juurtipud läbi käia.

Binomiaalne kuhi (2)

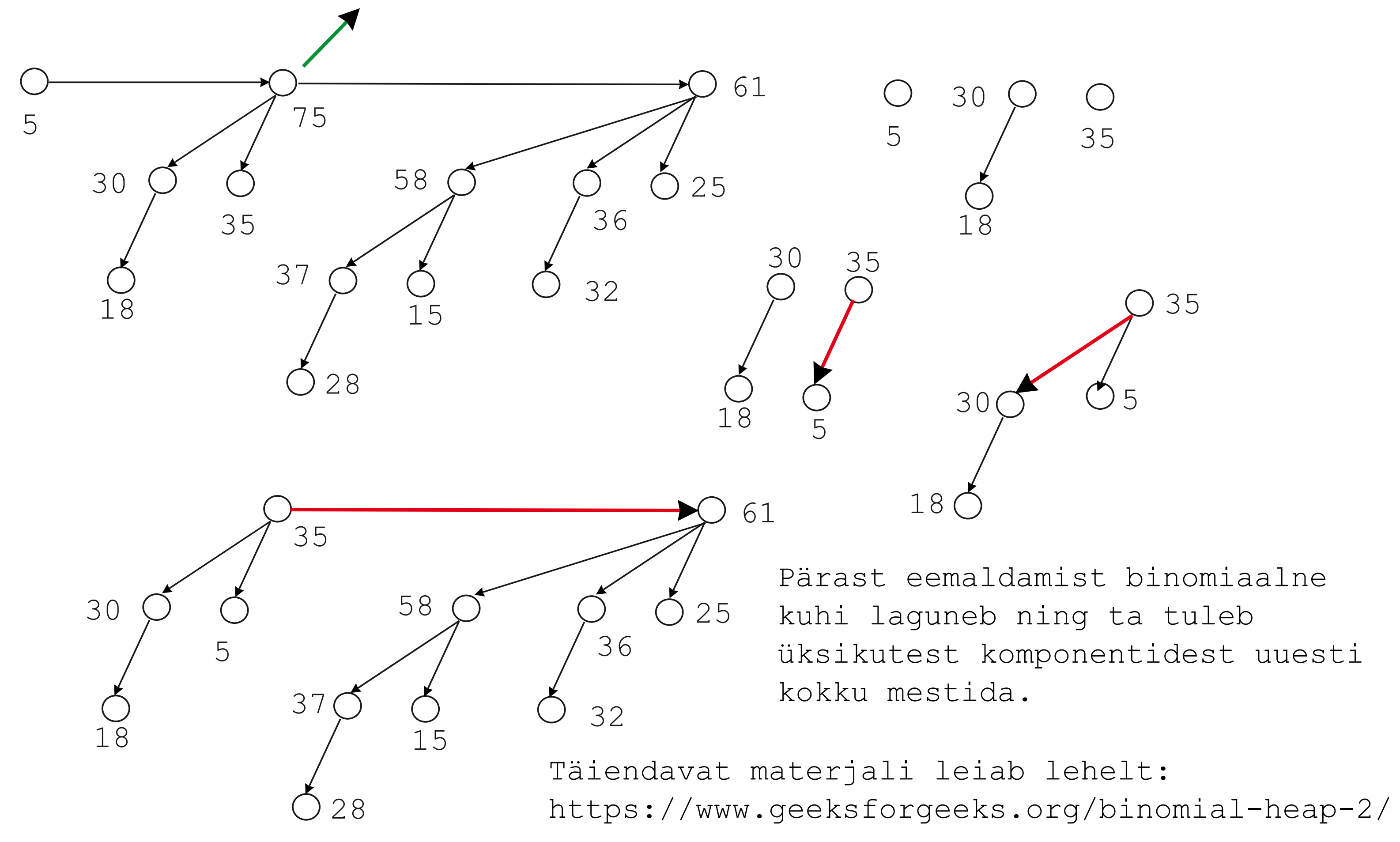
58 15 37 28 36 32 61 25 30 18 75 35 5



Iga uus kirje moodustab algul B_0 puu. Kui on juba 2 B_0 puud, tuleb nad kokku mestida B_1 puuks. Selliselt jätkates tekib varsti 2 B_1 puud. Need mestime kokku B_2 puuks jne.

Binomiaalne kuhi (3)

Binomiaalne kuhi (4)



Pärast eemaldamist binomiaalne kuhi laguneb ning ta tuleb üksikutest komponentidest uuesti kokku mestida.

Täiendavat materjali leiab lehelt: https://www.geeksforgeeks.org/binomial-heap-2/

Paisksalvestus (hashing)

Paisksalvestuse põhiidee on selline:
1. Olgu meil massiiv (paisktabel e. räsitabel) pikkusega m. Alguses on see tabel tühi.
2. Edasi olgu meil mingi funktsioon (paiskfunktsioon e. räsifunktsioon) h(k), mille argumendiks on kirjete võtmed ja väljundiks täisarv vahemikust 0...m-1.
3. Kirje paigutamiseks tabelisse leiame tema h(k) väärtuse. See annab meile tabeli indeksi.
4. Kirje otsimiseks arvutame h(k) abil tema asukoha. Kui see tabeli lahter on tühi, siis kirjet ei ole. Kui lahter ei ole tühi, tuleb ikkagi teha veel täielik võrdlus, sest h(k) võib erinevate võtmete puhul anda sama väljundi (nn. kokkupõrge, collision).

Kokkupõrke võimalus on äärmiselt häiriv uute kirjete paigutamisel tabelisse: võib juhtuda, et väljaarvutatud asukoht on juba hõivatud.

Paisksalvestuse 2 põhiprobleemi on:
1. Kuidas valida paiskfunktsiooni nii, et kokkupõrke tõenäosus oleks nii väike kui võimalik.
2. Kui kokkupõrge sellest hoolimata juhtus, siis kuhu paigatada kirje, mille arvutatud asukohal on juba teine kirje ees.

Paisksalvestuse nõrgaks kohaks on see, et me peame suutma ette ennustada kirjete arvu. Kui selgub, et tabel on liiga lühike, tuleb valida uus paisk-funktsioon s.t. alustama andmete paigutamist otsast peale.

Paiskfunktsioon jagamise meetodil(division method):

```
int k; // võti
int m; // tabeli pikkus
int h; // paiskfunktsiooni väärtus
h=k%m; // jagatise jääk
```

Halb valik: $m=2^n$

Olgu näiteks võti $k=2837=B15_{16}=1011\ 0001\ 0101_2$ ja n=10.

siis k%m annab tabeli indeksiks $789=315_{16}=0011\ 0001\ 0101_2$.

Sama indeksi saaksime kui võti on $k=1813=715_{16}=0111\ 0001\ 0101_2$.

Põhjus: kui m on kahe aste, siis jääk on võtme viimased n bitti.

Võtme kõrgemaid bitte ei kasutata üldse.

Soovitav lahendus: m olgu kirjete arvu mõnevõrra ületav algarv (prime number).

Neid leiab näiteks aadressilt:

https://www.mathsisfun.com/prime_numbers.html

Paiskfunktsioon ruutkeskmise meetodil(middle square method):

```
int k; // võti
int m; // tabeli pikkus on 2^m
int i,j1,j2; // abimuutujad
unsigned int kk,bit=0x80000000; // abimuutujad
unsigned int mask; // konstant, mille m madalamat bitti
                   // ühed, ülejäänud nullid,
                   // näiteks m=10 puhul on mask 0x3FF
int h; // paiskfunktsiooni väärtus
kk=k*k; // võtame ruutu
for (i=0; !(bit & kk); i++, bit>>=1); // leiame kõrgeima mittenullise biti
j1=(32-i-m)/2;
j2=32-i-m-j1;
h=((mask<<j2) & kk)>>j2;
```

k=2837, kk=8048569=$7ACFB9_{16}$

m=8

```
00000000 01111010 11001111 10111001 // kk

00000000 00000000 11111111 00000000 // mask<<j2
00000000 00000000 11001111 00000000 // (mask<<j2)&kk

00000000 00000000 00000000 11001111 // h, saame 207
```

tühi

32-i

j1

j2

m

Paiskfunktsioon voltimise meetodil(folding method)

```
int k; // võti
int m; // tabeli pikkus on 2^m, m peab olema suurem kui 8
unsigned char c[4]; // abimuutuja
int h; // paiskfunktsiooni väärtus
memcpy(&c[0],&k,4);
h=(c[0]^c[1])+((c[2]^c[3])<<(m-8));
```

8
m-8

XOR: samad 0, erinevad 1 (0^0=0, 1^1=0, 0^1=1, 1^0=1)

Aheldamine (chaining)

Kokkupõrganud kirjed paigutatakse ahelasse. Eelis: tabeli pikkuse valik ei ole kriitilise tähtsusega.

		0		0	

```
int compute_index(char *key)
{ // paigutus esitähe järgi
 return *key-'A';
}
struct hash_table // PAISKTABEL
{
 RECORD **ppHead;// viit viitade vektorile
 int nRecord; // viitade vektori pikkus
};
```

0		0	0	0	

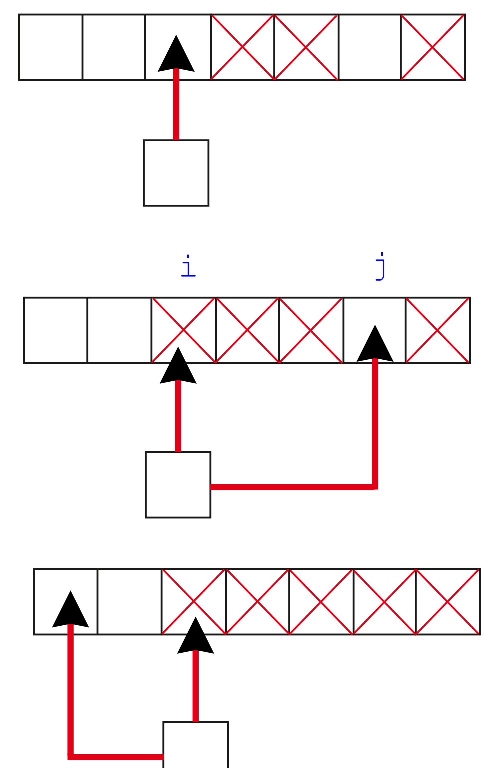
```
struct header // PÄIS
{
 void *pRecord;
 struct header *pNext;
};

struct hash_table // TABEL
{
 struct header **ppHead;
 int nPos;
};
```

Kelder (cellar)

Siin lähtume eeldusest, et kokkupõrgete oht on väike. Kelder on lihtsalt üks täiendav ahelloend, kuhu paigutatakse need kirjed, mida kokkupõrke tõttu tabelisse asetada ei saanud.
Otsimine keldri puhul toimuks selliselt:
- kui võtme alusel väljaarvutatud tabeli positsioon on tühi, siis kirjet pole
- kui positsioon ei ole tühi, siis teostatakse täiemahuline võtmete võrdlus. Kui see näitab võtmete kokkulangevust, on kirje leitud.
- kui võtmed on erinevad, siis on võimalik, et otsitav kirje on viidud keldrisse. Viimases tuleb rakendada järjestikulist otsimist.

Avatud adresseerimine (open addressing)(1)



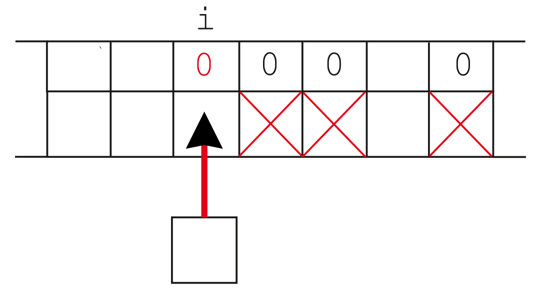
Avatud adresseerimine: kui koht on hõivatud, otsime mingite reeglite järgi mõne teise vaba koha.

Lineaarne proovimine: hakkame lihtsalt sammhaaval liikuma kuni leiame esimese tühja lahtri. Sinna panemegi oma kirje. Kui jõudsime lõppu ilma tühja lahtrit leidmata, jätkame algusest.

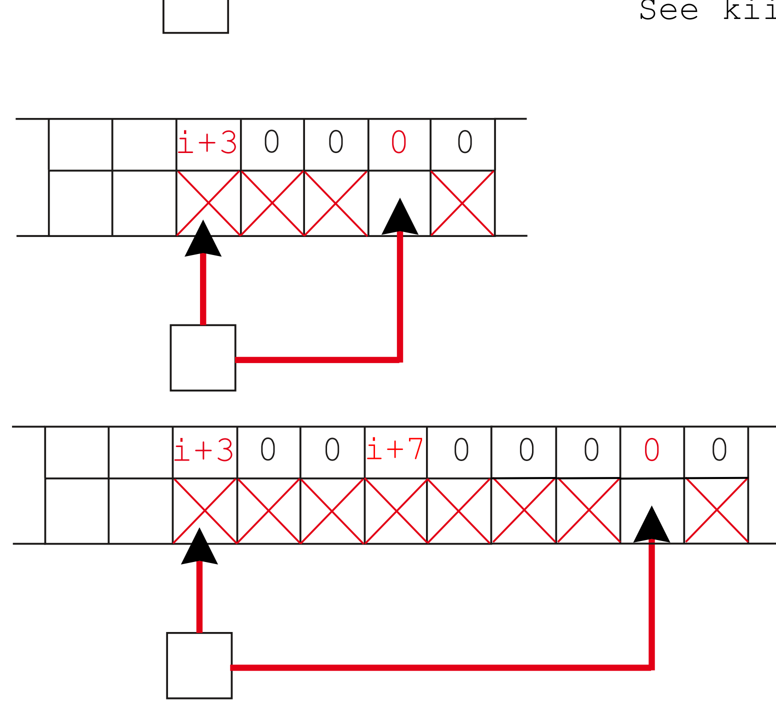
Probleem: olgu $h(k_1)=i$ ja $h(k_2)=i$. Kokkupõrke tõttu paigutame teise kirje kohale j. Kui aga $h(k_3)=j$, siis tuleb ka kolmandale kirjele otsida uus asukoht. Teoreetiliselt võib tekkida olukord, kus ainult esimene kirje on oma õigel kohal. Praktika näitab, et kui juba segadus tabelis on tekkinud, siis ta kasvab kiiresti.

Otsimine lihtsast lineaarse proovimise tabelist võib kergesti muutuda tavaliseks järjestikuliseks otsimiseks.

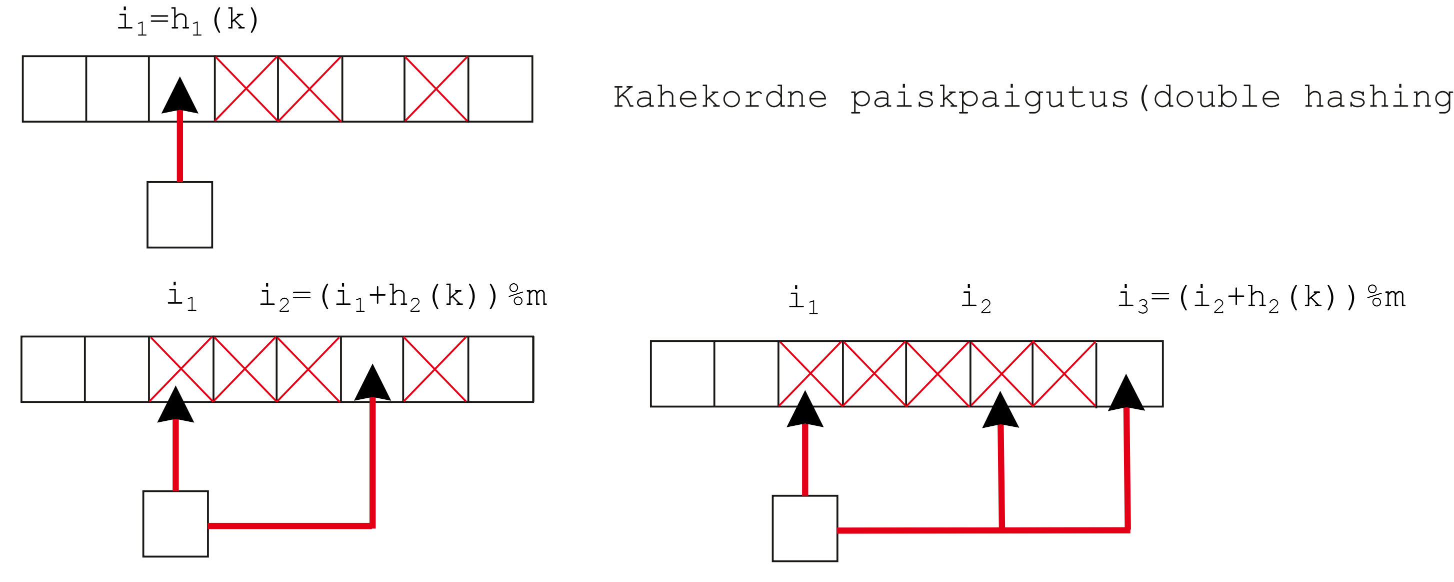
Avatud adresseerimine (open addressing)(2)



Koaleeruva paisktabeli (coalesced) puhul on tabeli igas lahtris veel indeks, mis alguses on null.
Kui toimub kokkupõrge ja kirjele otsitakse uus asukoht, asendatakse 0 uue asukoha indeksiga.
See kiirendab otsimist.



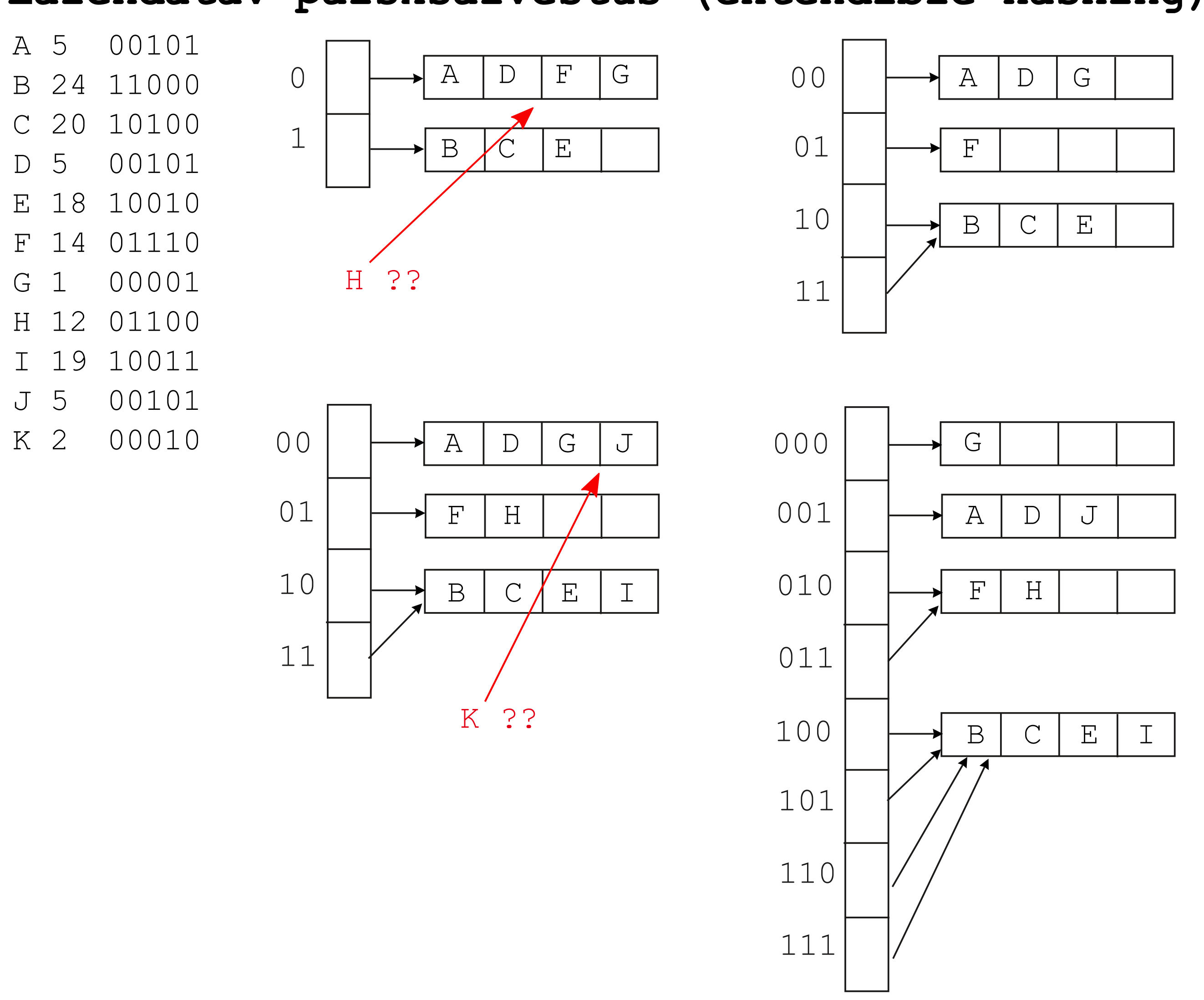
Avatud adresseerimine (3)




```
int k; // võti
int m; // tabeli pikkus
int h1, h2; // paiskfunktsiooni
            // väärtused
h1=hash_fun_1(k);
h2=hash_fun_2(k);
while (!empty(h1))
   h1=(h1+h2)%m;
```

h_2(k) ei tohi kunagi saada väärtusi, mis on võrdsed nulliga või jaguvad täpselt m-ga.
Põhjendus: (a+b)%c = (a%c+b%c)%c ja a%c=a kui a<c.
Soovitus: 0<h_2(k)<m, näiteks
h_2(k)=k%(m-1)+1
Kui m on algarv, siis parem on
h_2(k)=k%(m-2)+1

Laiendatav paisksalvestus (extendible hashing)



Valikuga sortimine (selection sort)

|35 40 16 79 48 (min: 16) — 16|40 35 79 48 (min: 35) — 16 35|40 79 48 (min: 40) — 16 35 40|79 48 (min: 48)

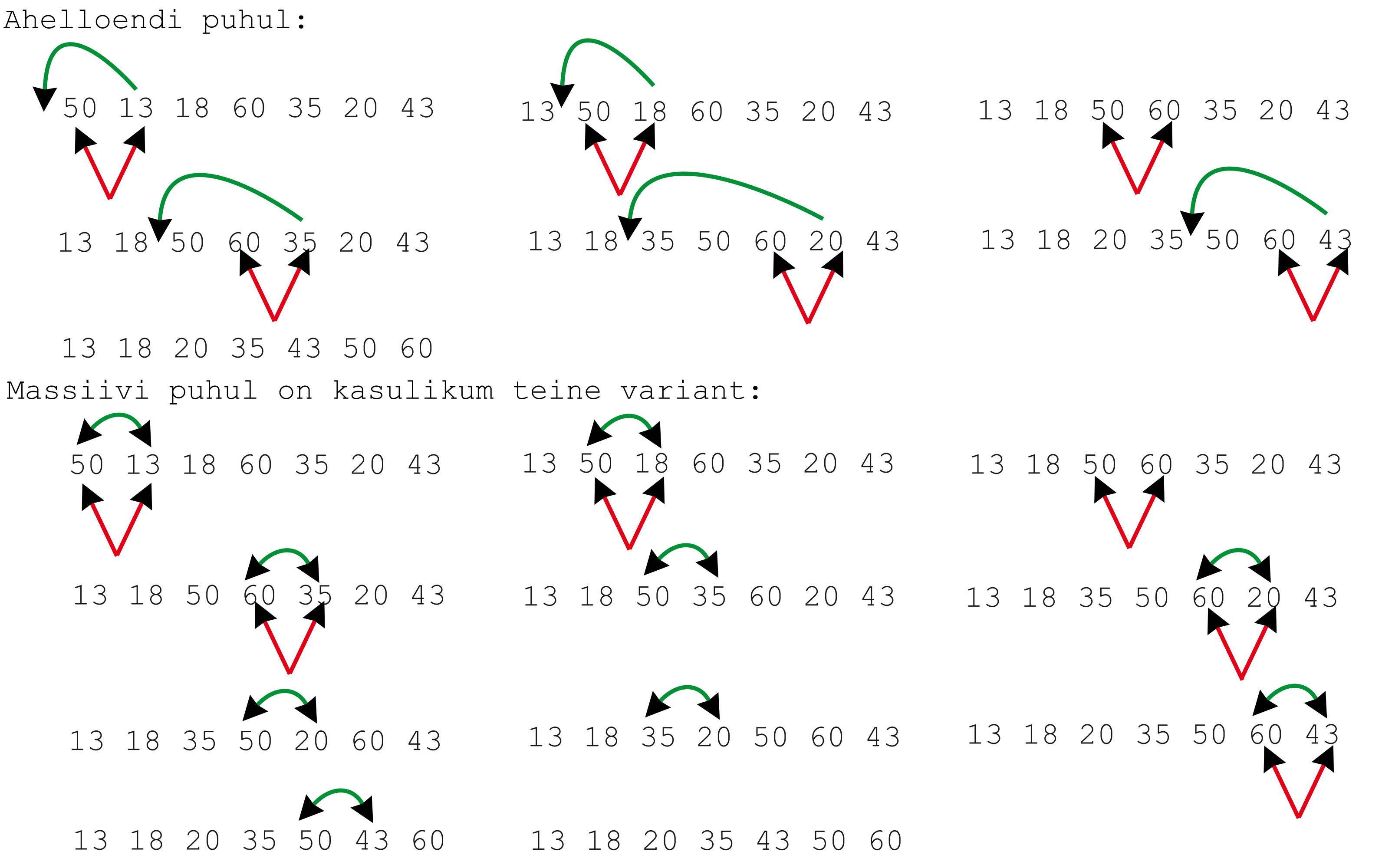
16 35 40 48|79

```
int SelectionSort(PERSON *pArray,int n) {
 int iStart, iMin, i;
 PERSON Temp;
 if (!pArray || !n)
   return 0;
 for (iStart = 0; iStart < n; iStart++) {
   for(i = iMin = iStart; i < n; i++)  {
      if (strcmp((pArray + i)->pName, (pArray + iMin)->pName) < 0)
         iMin = i;
     }
   if (iMin != iStart) {
      Temp = *(pArray + iMin);
      *(pArray + iMin) = *(pArray + iStart);
      *(pArray + iStart) = Temp;
      }
   }
   return 1;
}
```

Kvadraatse iseloomuga, äärmiselt ebaintelligentne. Isegi juba eelnevalt sorditud andmete korral tuleb ikka teha ära sama töö.

Vahelepanekuga sortimine (insertion sort)

Ahelloendi puhul:



Samuti kvadraatse iseloomuga, kuid kui andmed on eelnevalt mingil määral juba sorditud, kahaneb töö maht väga oluliselt.

Sortimine mullmeetodil (bubble sort)

25 57 48 37 12 92 86 33

25 48 57 37 12 92 86 33

25 48 37 57 12 92 86 33

25 48 37 12 57 92 86 33

25 48 37 12 57 86 92 33

25 48 37 12 57 86 33 92

---

25 48 37 12 57 86 33 92

25 37 48 12 57 86 33 92

25 37 12 48 57 86 33 92

25 37 12 48 57 33 86 92

---

25 37 12 48 57 33 86 92

25 12 37 48 57 33 86 92

25 12 37 48 33 57 86 92

Iga etapi lõpuks tõuseb üks kirje oma kohale loendi lõpus. Kui etapi jooksul mitte ühtegi ümberpaigutust ei toimu, on sortimine lõppenud.
Kvadraatse iseloomuga, kuid osaliselt juba sorditud lähte-andmete puhul tõhus.

Shell'i sortimine (Shellsort) (1)

Ühte gruppi kuuluvate kirjete indeksite vahe on n/2

25 57 48 37 12 92 86 33
1 2 3 4 1 2 3 4

25 57 48 37 12 92 86 33
1 2 3 4 1 2 3 4

12 57 48 37 25 92 86 33
1 2 3 4 1 2 3 4

12 57 48 37 25 92 86 33
1 2 3 4 1 2 3 4

12 57 48 37 25 92 86 33
1 2 3 4 1 2 3 4

12 57 48 33 25 92 86 37
1 2 3 4 1 2 3 4

Ühte gruppi kuuluvate kirjete indeksite vahe on n/4

12 57 48 33 25 92 86 37
1 2 1 2 1 2 1 2

12 57 25 33 48 92 86 37
1 2 1 2 1 2 1 2

Ühte gruppi kuuluvate kirjete indeksite vahe on n/8

12 33 25 37 48 57 86 92
1 1 1 1 1 1 1 1

12 25 33 37 48 57 86 92

Grupi sees kasutame vahelepanekuga sortimist

Shell'i sortimine (2)

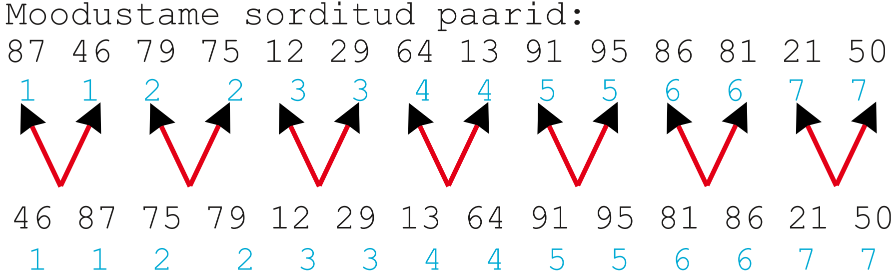
Põhiidee: jagame hulga segmentideks ja sordime need, siis mestime segmendid kokku suuremateks segmentideks ja sordime jne. Kuna tekkivad segmendid on juba eelmistel sammudel osaliselt sorditud, saame edukalt rakendada vahele-panekuga sortimist.
Segmendid moodustatakse kirjetest, mis asuvad teineteisest võimalikult kaugel.

Kui ütelda, et esimesel etapil moodustatakse k_1 segmenti, teisel mestitakse kokku k_2 segmenti jne. siis saame nn. inkrementide jada k_1, k_2,...,1. Eelmises näites $k_1= n/2$, $k_2= n/4$,...,1.

Meetodi efektiivsus sõltub kasutatud inkrementide jadast. Praktikas on läbi uuritud jadad $k_{i+1}=3k_i+1$ ja $k_i=2^i-1$. Nende puhul $\mathcal{O}(n^{3/2})$.

Mestimisega sortimine (merge sort) (1)

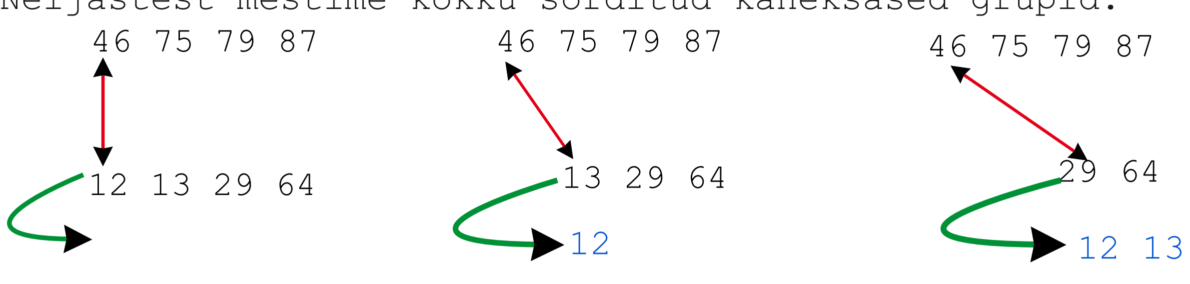
Moodustame sorditud paarid:



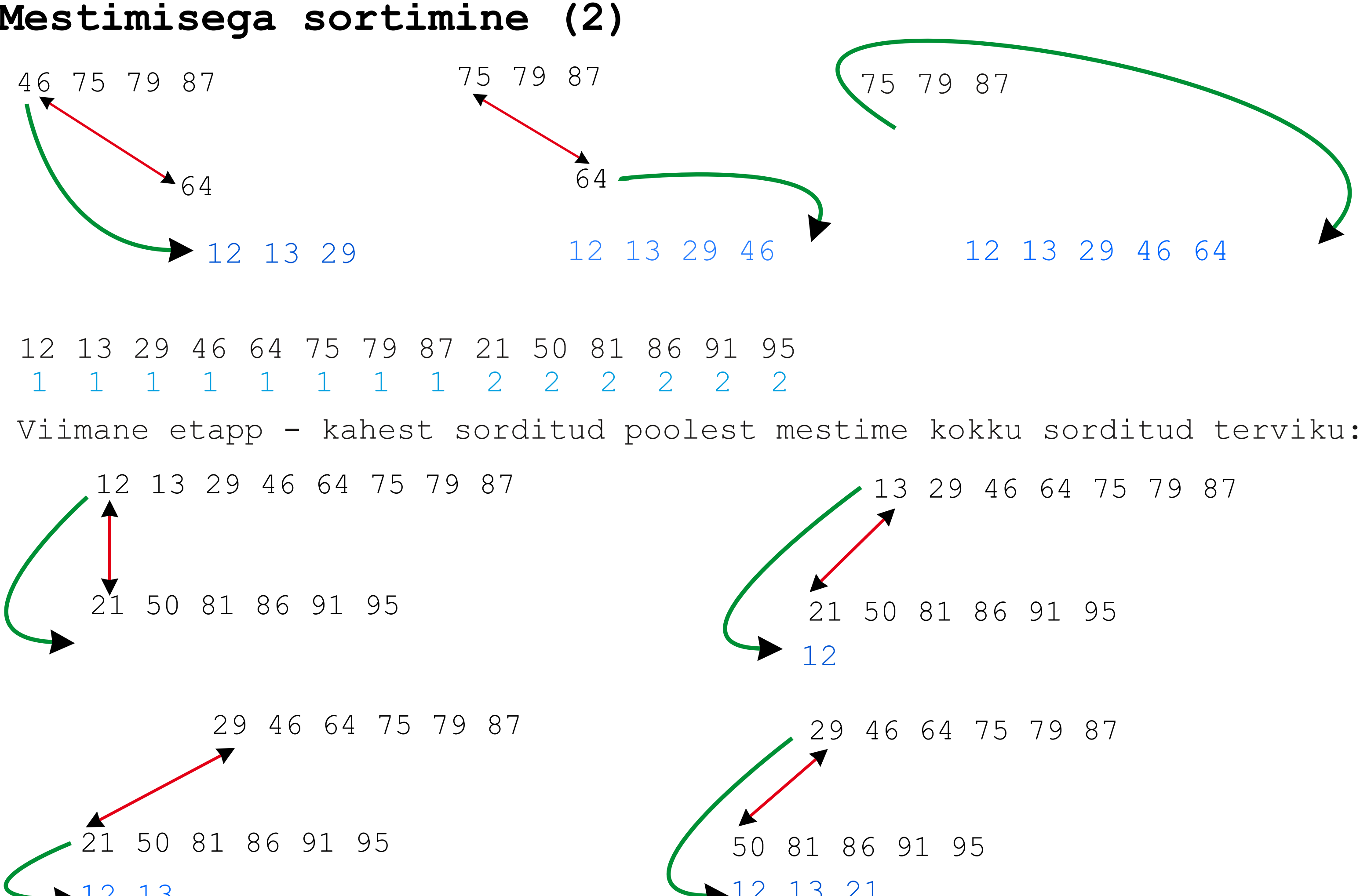
Paaridest mestime kokku sorditud neljased grupid:

46 75 79 87 12 13 29 64 81 86 91 95 21 50
1 1 1 1 2 2 2 2 3 3 3 3 4 4

Neljastest mestime kokku sorditud kaheksased grupid:



Mestimisega sortimine (2)



Mestimisega sortimine (3)

46 64 75 79 87
50 81 86 91 95
12 13 21 29

64 75 79 87
50 81 86 91 95
12 13 21 29 46

64 75 79 87
81 86 91 95
12 13 21 29 46 50

75 79 87
81 86 91 95
12 13 21 29 46 50 64

79 87
81 86 91 95
12 13 21 29 46 50 64 75

87
81 86 91 95
12 13 21 29 46 50 64 75 79

87
86 91 95
12 13 21 29 46
50 64 75 79 81

87
91 95
12 13 21 29 46
50 64 75 79 81
86

91 95
12 13 21 29 46
50 64 75 79 81
86 87

Lineaarlogaritmiline, sobib suurepäraselt ahelloendite sortimiseks. Ei vaja lisamälu, kõik toimub viitade ümbertõstmisega. Vt. ka https://www.geeksforgeeks.org/merge-sort/

Kiirsortimine (quick sort)(1)

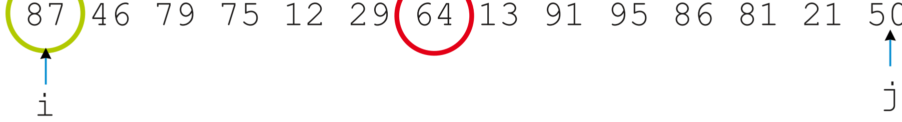
Valime peaelemendi:

87 46 79 75 12 29 64 13 91 95 86 81 21 50

i j

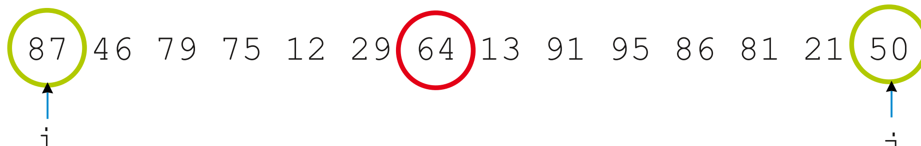
Alustame vasakpoolse indikaatori nihutamist paremale. Peatume, kui:

1. Jõudsime kohakuti parempoolse indikaatoriga.
2. Jõudsime võtmeni, mis on suurem kui peaelement.

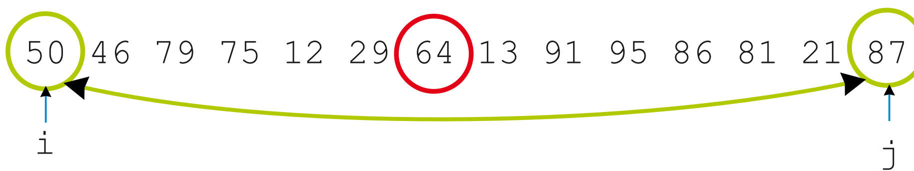


Alustame parempoolse indikaatori nihutamist vasakule. Peatume, kui:

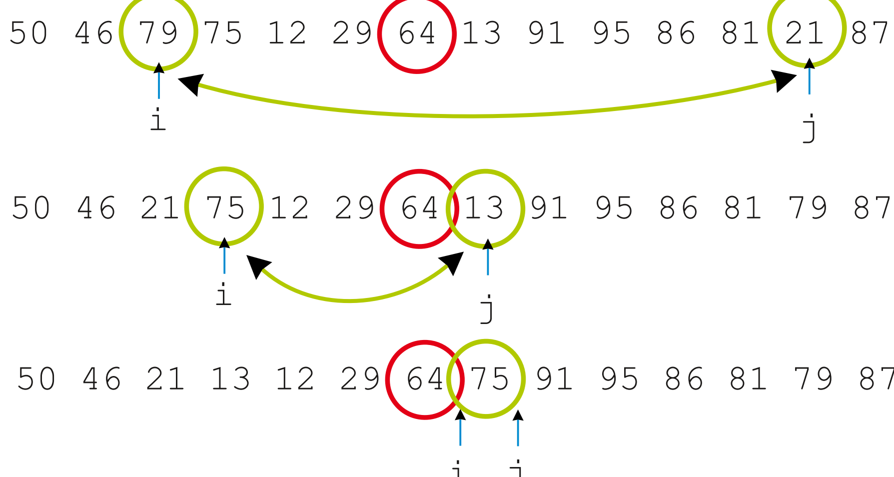
1. Jõudsime kohakuti vasakpoolse indikaatoriga.
2. Jõudsime võtmeni, mis on väiksem kui peaelement või võrdne temaga.



Kui nihutamise peatumisel indikaatorid ei olnud kohakuti, vahetame positsioonidel i ja j olevad kirjed ning jätkame liikumist.



Kiirsortimine (2)



Kui nihutamine peatus ja indikaatorid on kohakuti, tuleb massiiv poolitada. Kirje, millel indikaatorid peatusid, läheb parempoolse osa esimeseks kirjeks. Tema aadressi paneme magasini.

50 46 21 13 12 29 64 75 91 95 86 81 79 87

Jätkame samal moel vasakpoolse osaga:

50 46 21 13 12 29 64

i j

Kiirsortimine (3):

50 46 21 13 12 29 64 75 91 95 86 81 79 87

i j

50 46 21 13 12 29

i j

50 46 21 13 12 29

i j

29 46 21 13 12 50

i j

29 12 21 13 46 50 64 75 91 95 86 81 79 87

i j

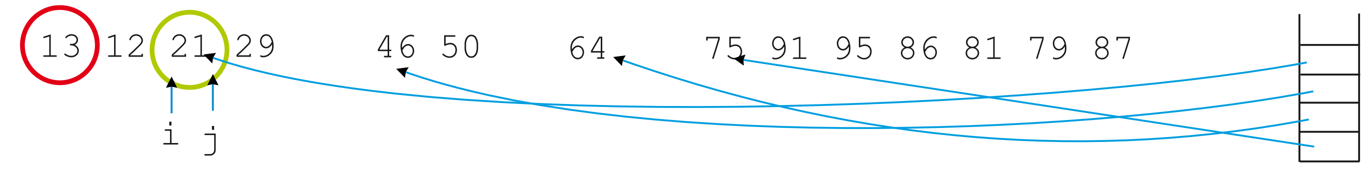
29 12 21 13

i j

29 12 21 13

i j

Kiirsortimine (4)

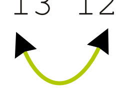


Kui vasakule jääv osa koosneb ainult ühest või kahest kirjest (viimasel juhul kontrollime järjestust ja vajaduse korral vahetame), siis on üks etapp läbi.

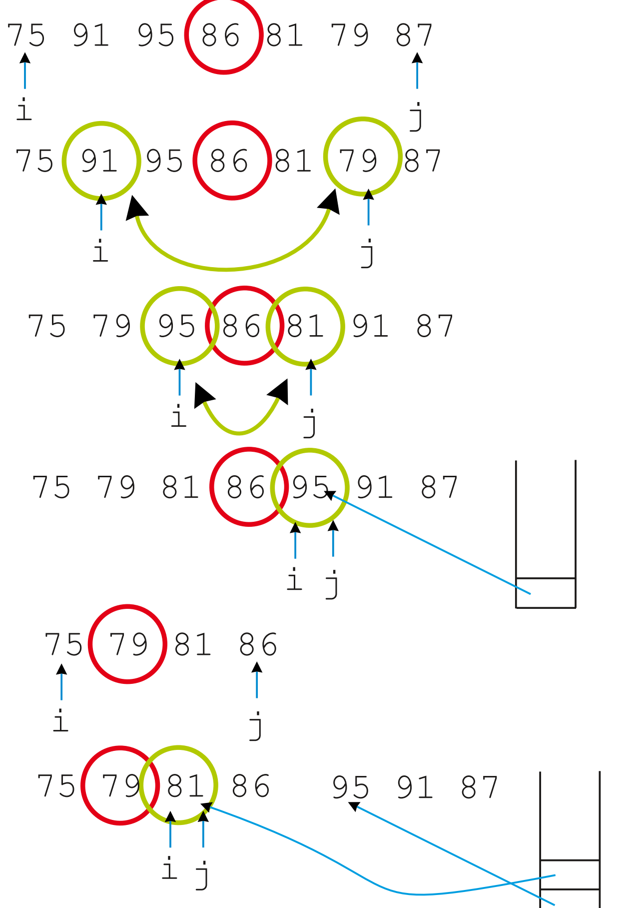
Magasinis olevad kaks ülemist viita annavad meile järjekordse sortimata lõigu alguse ja lõpu:

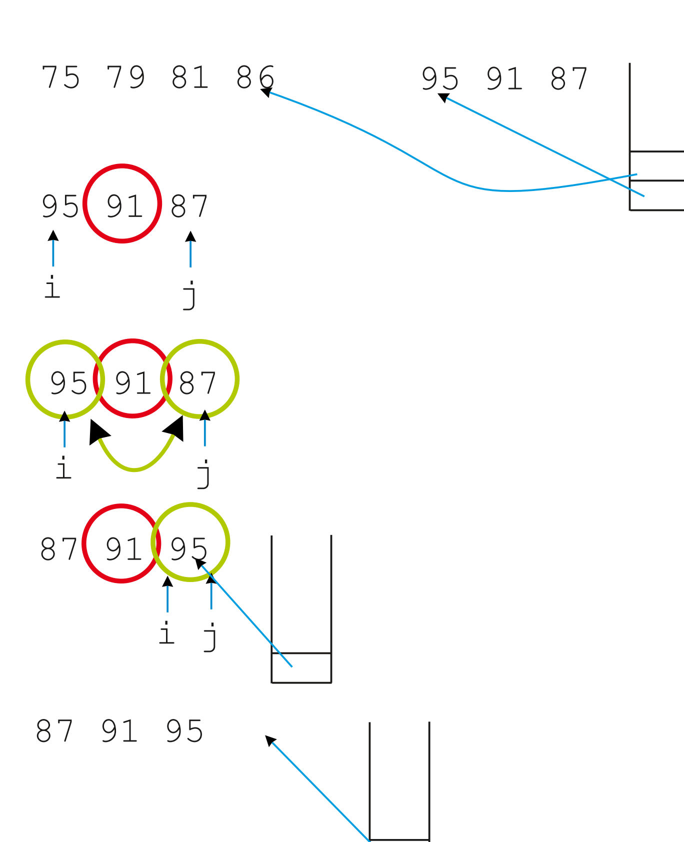
1. Kui see lõik koosneb vaid ühest kirjest, võtame järgmise lõigu.
2. Kui see lõik koosneb kahest kirjest, kontrollime järjestust, vajaduse korral vahetame ja võtame järgmise lõigu.
3. Kui kirjeid on rohkem, valime jälle peaelemendi jne.

13 12 21 29 46 50 64 75 91 95 86 81 79 87



Kiirsortimine (5)





Kiirsortimine (6)

Rekursiivse kiirsortimise põhiidee on selline:
1. Valime ühe kirjetest peaelemendiks (pivot).
2. Jaotame oma andmed nii, et peaelemendist paremale jäävatel kirjetel on suurem võti kui peaelemendil ja vasakule jäävatel kirjetel on väiksem võti kui peaelemendil.
3. Teeme kummagi osapoolega sedasama jne.

Peaelement ei tohi olla ei suurima ega ka mitte väikseima võtmega kirje. Tavaliselt valitakse ta nn. mediaan kolmest meetodil: võetakse üks kirje massiivi algusosast, üks kirje lõpuosast ja üks kirje keskelt. Sellest kolmikust saab peaelemendiks kirje, mille võti on keskmise väärtusega.

Väga lühikeste lõikudega jaotamist jatkata ei ole mõistlik. Tavaliselt jäetakse lõigud pikkusega alla 10 esialgu sortimata. Lõplik sortimine teostatakse vahelepanekuga sortimise teel.

Kiirsortimine on lineaarlogaritmilise iseloomuga. Kui aga andmed on algusest peale osaliselt juba sorditud, muutub see algoritm aeglasemaks. Halvim on olukord, kus andmed on juba sorditud kuid vastupidises järjekorras: siis on meil tegemist kvadraatse iseloomuga sortimisega.

Kiirsortimine sobib hästi massiividele kuid mitte ahelloenditele.

Microsoftil on C standardfunktsioon

```
void qsort(void *base, size_t num, size_t width,
          int (__cdecl *compare )(const void *, const void *) );
```

Vt. ka https://www.geeksforgeeks.org/quick-sort/

Sortimine kuhja abil (heapsort) (1)

Kui 2i<n ja 2i+1<n ja i/2>0, siis kehtib $a_i>a_{2i}$ ja $a_i>a_{2i+1}$ ja $a_i<a_{i/2}$

58 15 37 28 36 32 61 25 30 68 75 Oletame, et see on kuhi

1 2 3 4 5 6 7 8 9 10 11

$a_1>a_2$ ja $a_1>a_3$ kehtivad, kuid $a_2>a_4$ ja $a_2>a_5$ mitte.

58 15 37 28 36 32 61 25 30 68 75

58 28 37 15 36 32 61 25 30 68 75

$a_2>a_4$, $a_2>a_5$ ja $a_2<a_1$ kehtivad. Kehtib ka $a_3>a_6$ kuid mitte $a_3>a_7$.

58 36 37 15 28 32 61 25 30 68 75

58 36 61 15 28 32 37 25 30 68 75

$a_3>a_6$ ja $a_3>a_7$ kehtivad, kuid $a_3<a_1$ mitte.

58 36 61 15 28 32 37 25 30 68 75

61 36 58 15 28 32 37 25 30 68 75

i=4 puhul: 61 36 58 15 28 32 37 25 30 68 75

61 36 58 25 28 32 37 15 30 68 75

Sortimine kuhja abil (2)

Kui 2i<n ja 2i+1<n ja i/2>0, siis kehtib $a_i > a_{2i}$ ja $a_i > a_{2i+1}$ ja $a_i < a_{i/2}$

61 36 58 30 28 32 37 15 25 68 75

1 2 3 4 5 6 7 8 9 10 11

i=5 puhul: 61 36 58 30 28 32 37 15 25 68 75

61 36 58 30 68 32 37 15 25 28 75

61 68 58 30 36 32 37 15 25 28 75

---

75 68 58 30 61 32 37 15 25 28 36

1 2 3 4 5 6 7 8 9 10 11

Saadud korrapärases kuhjas on suurima võtmega kirje kõige esimene. Vahetame esimese ja viimase kirje:

75 68 58 30 61 32 37 15 25 28 36

36 68 58 30 61 32 37 15 25 28 75

Uueks kuhjaks loeme tekkinud massiivi ilma viimase kirjeta. Kuivõrd tema võti on suurim, on ta juba omal õigel kohal.

36 68 58 30 61 32 37 15 25 28 75

Järgnevalt korrastame tekkinud lühema kuhja jne. Tegemist on lineaarlogaritmilise protsessiga, vt. ka https://www.geeksforgeeks.org/heap-sort/

Jaotamisega sortimine (radix distribution sort)

1 19 15 18 20 9 14 7 5 24 2 13 16 12 6

Jaotus biti b_0 järgi:

0: 18 20 14 24 2 1 6 12 6

1: 1 19 15 9 7 5 13

18 20 14 24 2 16 12 6 1 19 15 9 7 5 13

Jaotus biti b_1 järgi:

0: 20 24 16 12 1 9 5 13

1: 18 14 2 6 19 15 7

20 24 16 12 1 9 5 13 18 14 2 6 19 15 7

Jaotus biti b_2 järgi:

0: 24 16 1 9 18 2 19

1: 20 12 5 13 14 6 15 7

24 16 1 9 18 2 19 20 12 5 13 14 6 15 7

Jaotus biti b_3 järgi:

0: 16 1 18 19 20 2 5 6 7

1: 24 9 12 13 14 15

16 1 18 19 20 2 5 6 7 24 9 12 13 14 15

Jaotus biti b_4 järgi:

0: 1 2 5 6 7 9 12 13 14 15

1: 16 18 19 20 24

1 2 5 6 7 9 12 13 14 15 16 18 19 20 24

1 00001 $\{ b_4, b_3, b_2, b_1, b_0 \}$

19 10011

15 01111

18 10010

20 10100

9 01001

14 01110

7 00111

15 00101

24 11000

2 00010

13 01101

16 10000

12 01100

6 00110

Sobib väga hästi ahelloenditele.
Lisamälu sel juhul ei vaja.

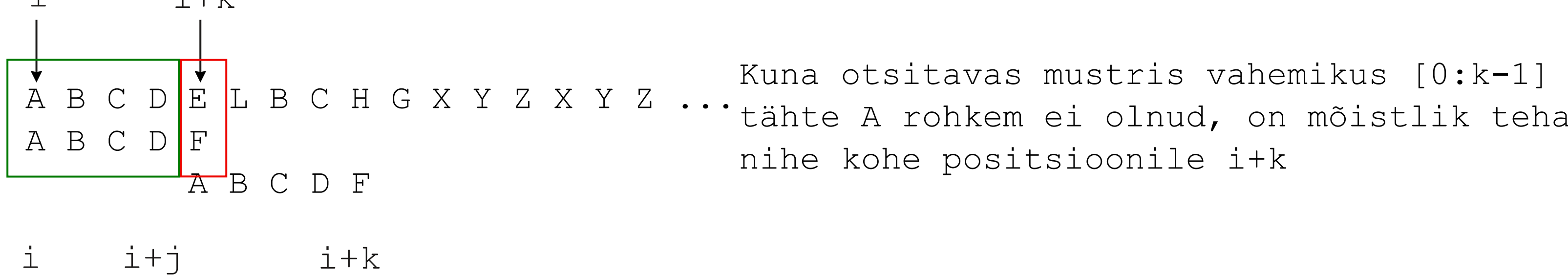
Vt. ka https://www.geeksforgeeks.org/radix-sort/

Tekstist otsimine jõumeetodil (brute force)

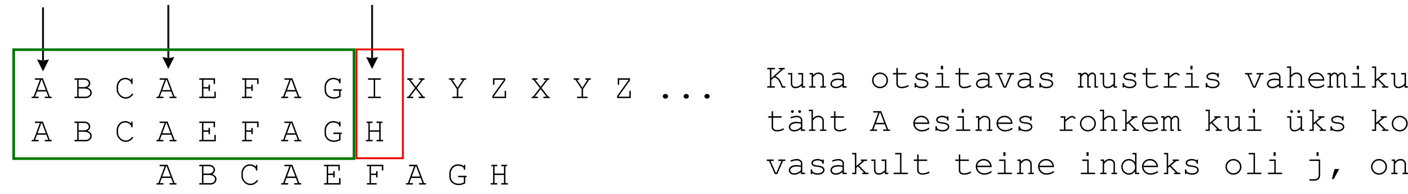
```
A B E B C E F G A
B C D
  B C D
    B C D
      B C D
        B C D
          B C D
            B C D
```

```
unsigned char *pattern_match(unsigned char *text,int n_text,
                             unsigned char *pattern,int n_pattern)
{
 unsigned char *p;
 for (p=text; p<=text+n_text-n_pattern; p++)
 {
   if (!memcmp(p,pattern,n_pattern))
      return p;
 }
 return 0;
}
```

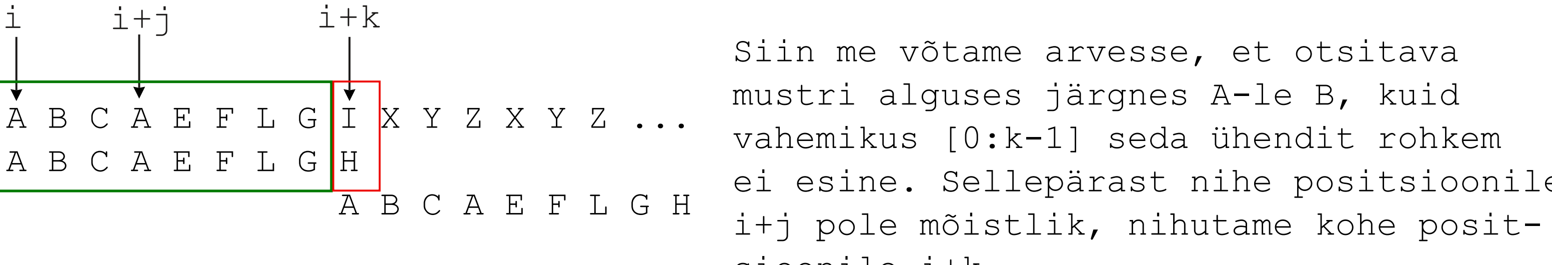
Tekstist otsimine: mõningaid lihtsaid trikke



Kuna otsitavas mustris vahemikus [0:k-1] tähte A rohkem ei olnud, on mõistlik teha nihe kohe positsioonile i+k

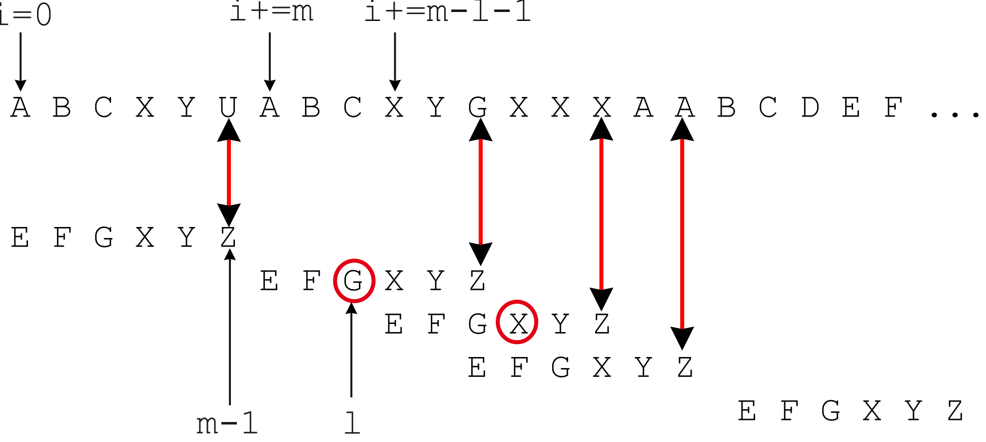


Kuna otsitavas mustris vahemikus [0:k-1] täht A esines rohkem kui üks kord ja tema vasakult teine indeks oli j, on mõistlik teha nihe kohe positsioonile i+j



Siin me võtame arvesse, et otsitava mustri alguses järgnes A-le B, kuid vahemikus [0:k-1] seda ühendit rohkem ei esine. Sellepärast nihe positsioonile i+j pole mõistlik, nihutame kohe positsioonile i+k.

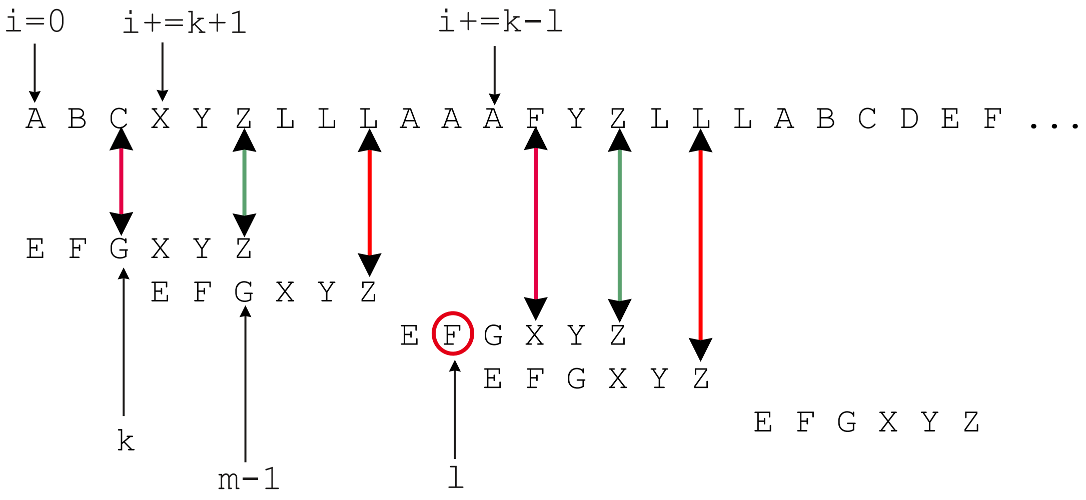
Boyer-Moore'i tekstist otsimise algoritm (1)



Oletame, et otsitav muster pikkusega m on paigutatud alates teksti positsioonist i ning et pattern[m-1]!=text[i+m-1]. Siis:

- kui text[i+m-1] ei esine mustris mitte kordagi, võib teda nihutada tema terve pikkuse võrra paremale, s.t. i+=m.
- kui text[i+m-1] esineb mustris üks või enam kordi ning l on tema äärmine parempoolne positsioon , siis mustri alguse võib nihutada positsioonile i+=m-l-1

Boyer-Moore'i tekstist otsimise algoritm (2)



Oletame, et muster pikkusega m on paigutatud alates teksti positsioonist i ning et pattern[m-1]==text[i+m-1]. Oletame edasi, et kui nüüd alustada liikumist paremalt vasakule, siis mustri positsioonil k tekib esimene ebakõla. Siis:

- kui text[i+k] ei esine mustri lõigul [0:k-1] mitte kordagi,võib tema alguse nihutada positsioonile i+=k+1.
- kui text[i+k] esineb mustri lõigul [0:k-1] üks või enam kordi ning l on sel lõigul tema äärmine parempoolne positsioon, siis mustri alguse võib nihutada positsioonile i+=k-l

Boyer-Moore meetod eeldab, et meil on tabel, kuhu on sisestatud kõik mustris esinevad märgid ja nende äärmised parempoolsed positsioonid.
Mida harvemini kohtab tekstis mustri viimast märki, seda suurema efekti Boyer-Moore'i meetod annab.

Rabin-Karp'i tekstist otsimise algoritm (1)

```
unsigned char *pattern_match(unsigned char *text,int n_text,
                             unsigned char *pattern,int n_pattern)
{
  int h_pattern,h_text=-1,i,j;
  h_pattern=hash_fun(pattern,n_pattern,0,-1);
  for (i=0; i<=n_text-n_pattern; i++)
  {
      h_text=hash_fun(text,n_pattern,i,h_text); // h(a_i:a_i+m-1)
      if (h_text==h_pattern)
      {
          for (j=0; j<n_pattern; j++)
          { // täiendav võrdlus
              if (*(pattern+j)!=*(text+i+j))
                  break; // kokkupõrge
          }
          if (i==n_pattern)
              return text+i; // leidis
      }
  }
  return 0; // ei ole
}
// hash_fun parameetrid: viit märkide massiivile; massiivi pikkus;
// indeks, kust alustada; eelmine väärtus või -1, kui see puudub
```

Rabin-Karp'i tekstist otsimise algoritm (2)

Probleem: leida paiskfunktsioon, mis annaks võimalikult vähe kokkupõrkeid ja võimaldaks hõlpsalt arvutada juba leitud $h(a_i:a_{m-i-1})$ põhjal järgmise paisk-funktsiooni $h(a_{i+1}:a_{i+m})$.

Lihtsaim lahendus:

$h(a_0:a_{m-1})=a_0+a_1+...+a_{m-1}$

$h(a_1:a_m)=a_1+a_2+...+a_m=h(a_0:a_{m-1})-a_0+a_m$

võib anda palju kokkupõrkeid.

Tuues sisse kaaluteguri c, saame kokkupõrgete ohtu vähendada:

$h(a_0:a_{m-1})=a_0c^{m-1}+a_1c^{m-2}+...+a_{m-2}c+a_{m-1}$

$h(a_1:a_m)=a_1c^{m-1}+a_2c^{m-2}+...+a_{m-1}c+a_m=h(a_0:a_{m-1})c-a_0c^m+a_m$

kuid praktikas võib siin c astendamisel tekkida väga suuri täisarve,mis ei mahu isegi kaheksale baidile.

Rabin-Karpi lahendus:

$h(a_0:a_{m-1})=(a_0*256^{m-1}+a_1*256^{m-2}+...+a_{m-2}*256+a_{m-1})\%q$,

kus algarv q>256 ja m on märkide arv mustris.

Tuleb tõestada, et

1. Paiskfunktsiooni arvutamisel suuri täisarve ei teki.
2. Juba leitud $h(a_i:a_{m-i-1})$ põhjal järgmise paiskfunktsiooni $h(a_{i+1}:a_{i+m})$ leidmine (nn. paiskfunktsiooni rullimine) on lihtne.

Rabin-Karp'i tekstist otsimise algoritm (3)

$h(a_0:a_{m-1})=(a_0*256^{m-1}+a_1*256^{m-2}+...+a_{m-2}*256+a_{m-1})\%q$, kus $q>256$

Moodularitmeetika teoreem 1: $(a*b)\%d=((a\%d)*b)\%d$
Moodularitmeetika teoreem 2: $(a+b)\%d=((a\%d)+(b\%d))\%d$

Kui m=1, siis $h(a0)=a0\%q=a0$, sest $a0<256$ ja seega $a0<q$ ning ka iga i puhul $a_i<q$

Kui m=2, siis $h(a_0:a_1)=(a_0*256+a_1)\%q=((a_0*256)\%q+a_1\%q)\%q=$
$=(((a_0\%q)*256)\%q+a_1\%q)\%q$
Et $a_1<q$, siis $h(a_0:a_1)=((h(a_0)*256)\%q+a_1)\%q$

Kui m=3, siis $h(a_0:a_2)=(a_0*256^2+a_1*256+a_0)\%q=((a_0*256)+a_1)*256+a_2)\%q$
(kasutame Horneri skeemi). Siit jätkates:

$h(a_0:a_2)=(((a_0*256+a_1)*256)\%q+a_2\%q)\%q=(((a_0*256+a_1)\%q*256)\%q+a_2)\%q$
ja seega
$h(a_0:a_2)=((h(a_0:a_1)*256)\%q+a_2)\%q$
....
$h(a_0:a_{m-1})=((h(a_0:a_{m-2})*256)\%q+a_{m-1})\%q$

Seega saame $h(a_0:a_{m-1})$ leida samm-sammult, arvutades algul $h(a_0)$,tema põhjal $h(a_0:a_1)$, viimasest omakorda $h(a_0:a_2)$ jne. kuni $h(a_0:a_{m-1})$.Et igal pool on tegemist jääkidega ja jääk ei saa ületada q-d,suuri arve ei teki.

Rabin-Karp'i tekstist otsimise algoritm (4)

Paigutusfunktsiooni rullimine:

$h(a1:am)=(a_1*256^{m-1}+a_2*256^{m-2}+...+a_{m-1}*256+a_m)\%q$

Kirjutame selle avaldise natuke teisel kujul:

$h(a1:am)=((a_0*256^{m-1}+a_1*256^{m-2}+...+a_{m-1})*256-a_0*256^m+a_m)\%q$

Olgu:

$A=(a_0*256^{m-1}+a_1*256^{m-2}+...+a_{m-1})*256$

$B=a_m-a_0*256^m$

Siis:

$h(a_1:a_m)=(A+B)\%q=(A\%q+B\%q)\%q$

Kumbagi liidetavat eraldi uurides saame:

$B\%q=(a_m\%q-(a_0*256^m)\%q)\%q=am-(a_0*256^m)\%q$

sest $a_m<q$ ja $(a_0*256^m)\%q<q$, seega on ka nende vahe q-st väiksem

$A\%q=((a_0*256_{m-1}+a_1*256^{m-2}+...+a_{m-1})\%q*256)\%q=(h(a_0:a_{m-1})*256)\%q$

Seega:

$h(a_1:a_m)=((h(a_0:a_{m-1})*256)\%q+a_m-(a_0*256^m)\%q)\%q$

Tähistame:

$K=(256^m)\%q$

Kokkuvõttes:

$h(a_1:a_m)=((h(a_0:a_{m-1})*256)\%q+a_m-(a_0*K)\%q)\%q$

....

$h(a_i:a_{i+m-1})=((h(a_{i-1}:a_{i+m-2})*256)\%q+a_{i+m-1}-(a_{i-1}*K)\%q)\%q$

Rabin-Karp'i tekstist otsimise algoritm (5)

$K=(256^m)\%q$ tuleb eelnevalt välja arvutada.

Et q>256, siis see ei too kaasa ületäitumisi:

$(256^2)\%q=(256\%q*256)\%q$

$(256^3)\%q=((256^2)\%q*256)\%q$ ja $(256^2)\%q$ leidsime juba eelmise sammuga

$(256^4)\%q=(((256)^3\%q)*256)\%q$ ja $(256^3)\%q$ leidsime juba eelmise sammuga.

Niimoodi leiame sammhaaval $(256^m)\%q$.

Vt. ka https://www.geeksforgeeks.org/searching-for-patterns-set-3-rabin-karp-algorithm/5

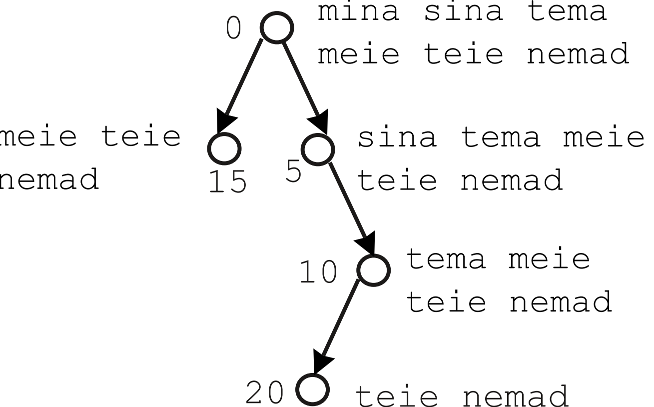
Teksti indekspuu

0 mina sina tema meie teie nemad

0 mina sina tema meie teie nemad

5 sina tema meie teie nemad

10 tema meie teie nemad



Eeldame, et tekst koosneb sõnadest. Juurtipu võtmeks on kogu tekst. Teisena paneme puusse tipu, mille võtmeks on tekst alates teisest sõnast jne. Igasse tippu salvestame sõna alguse indeksi.

mina sina tema meie teie nemad
0 5 10 15 20 25

0 mina sina tema meie teie nemad

5 sina tema meie teie nemad

0
meie teie nemad 15
5 sina tema meie teie nemad
10 tema meie teie nemad

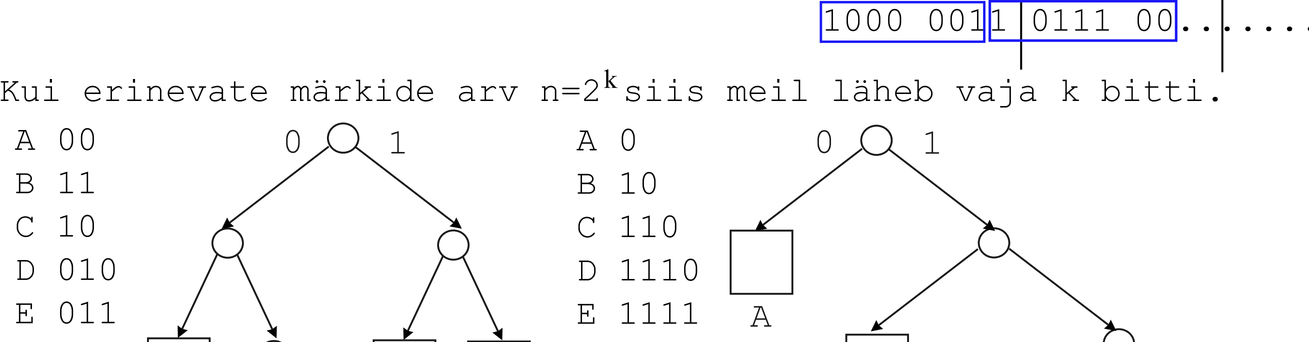
0 mina sina tema meie teie nemad
15 meie teie nemad
5 sina tema meie teie nemad
25 nemad
10 tema meie teie nemad
20 teie nemad

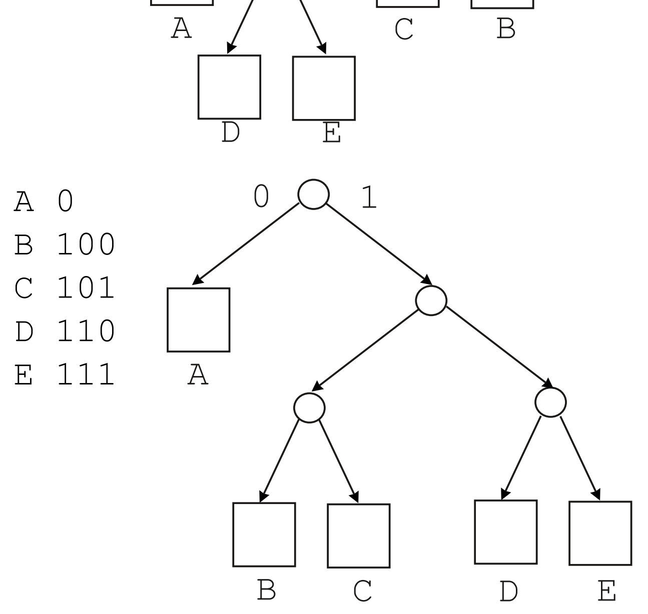
Huffmani kood (1)

8-bitise koodi asendamine 7-bitisega:

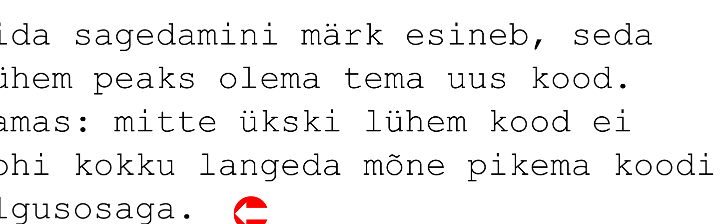
	A		Z	
4	1	5	A	
0100 0001		0101 1100		....
1000 0011		0111 00..		......

Kui erinevate märkide arv $n=2^k$ siis meil läheb vaja k bitti.



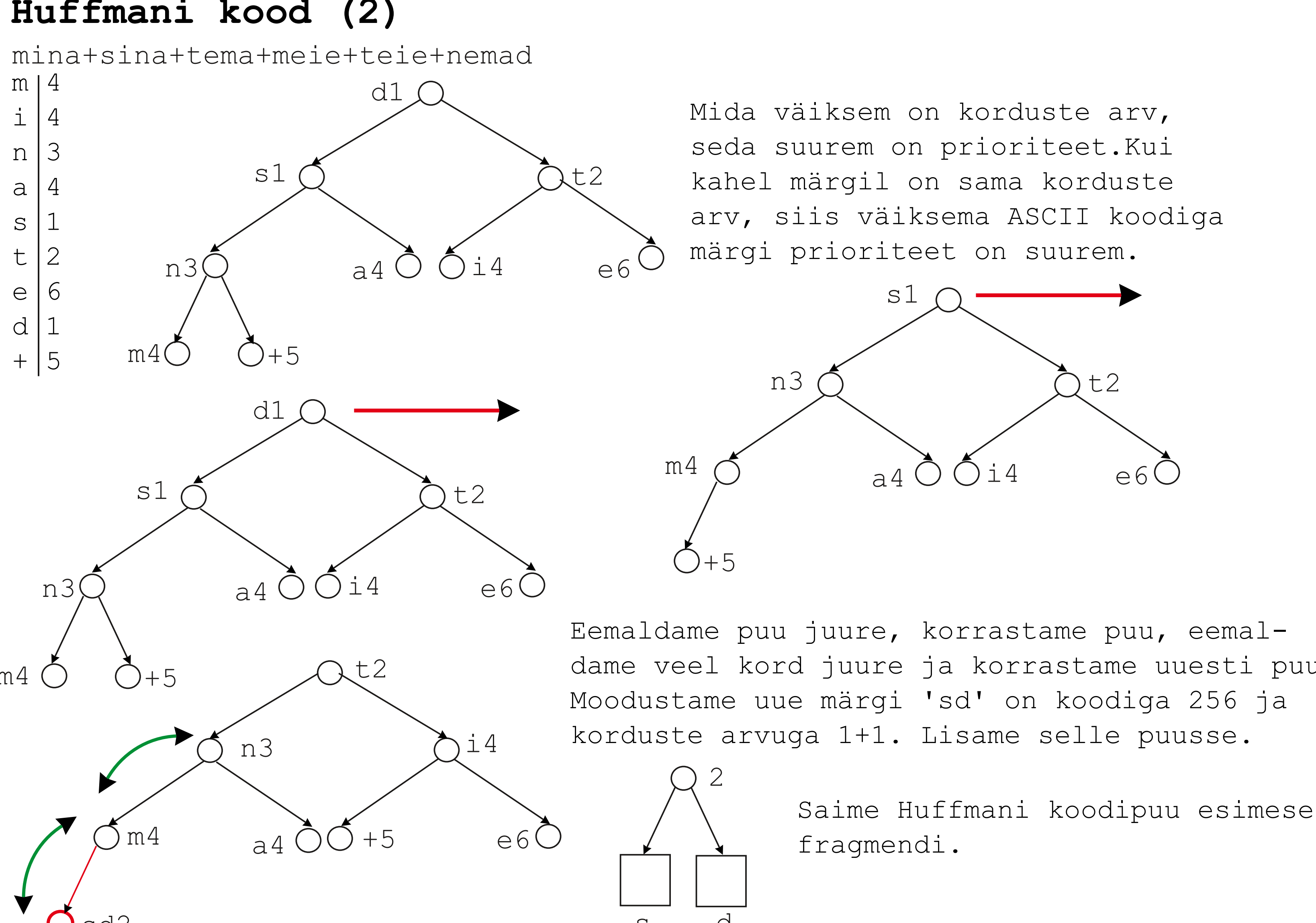


Mida sagedamini märk esineb, seda lühem peaks olema tema uus kood.
Samas: mitte ükski lühem kood ei tohi kokku langeda mõne pikema koodi algusosaga.

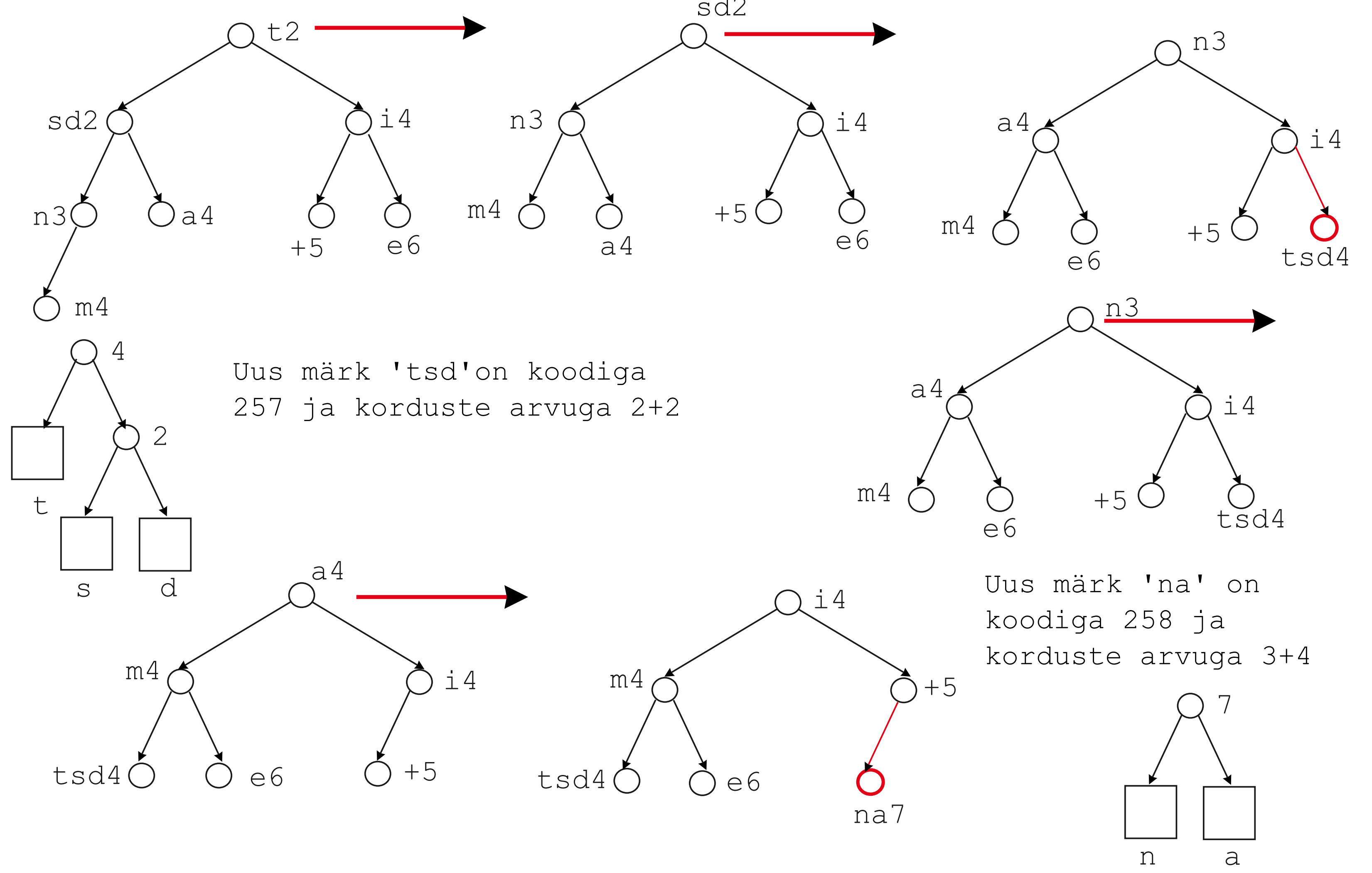


Huffmani kood annab kõige lühema ümberkodeeritud teksti.

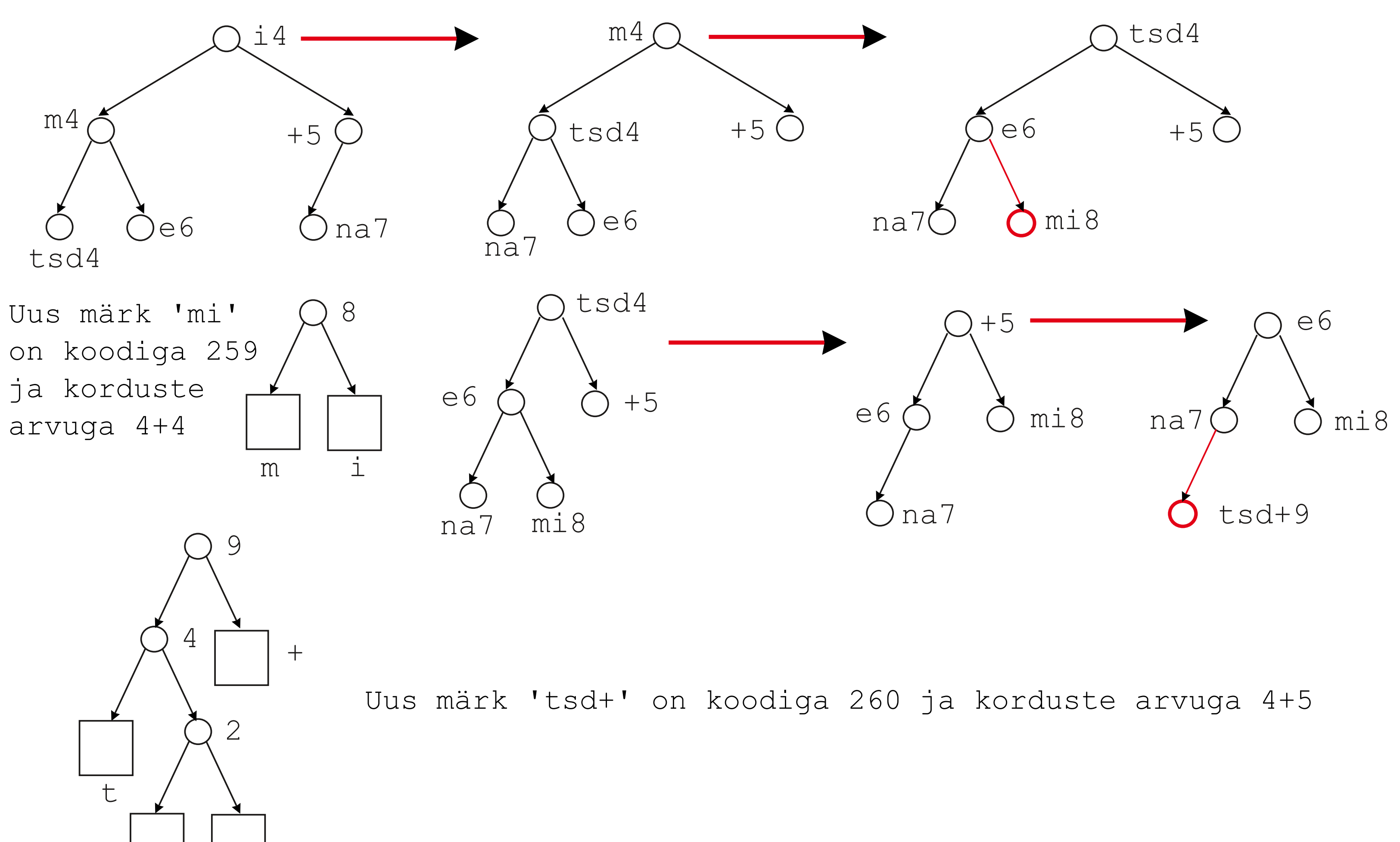
Huffmani kood (2)



Huffmani kood (3)

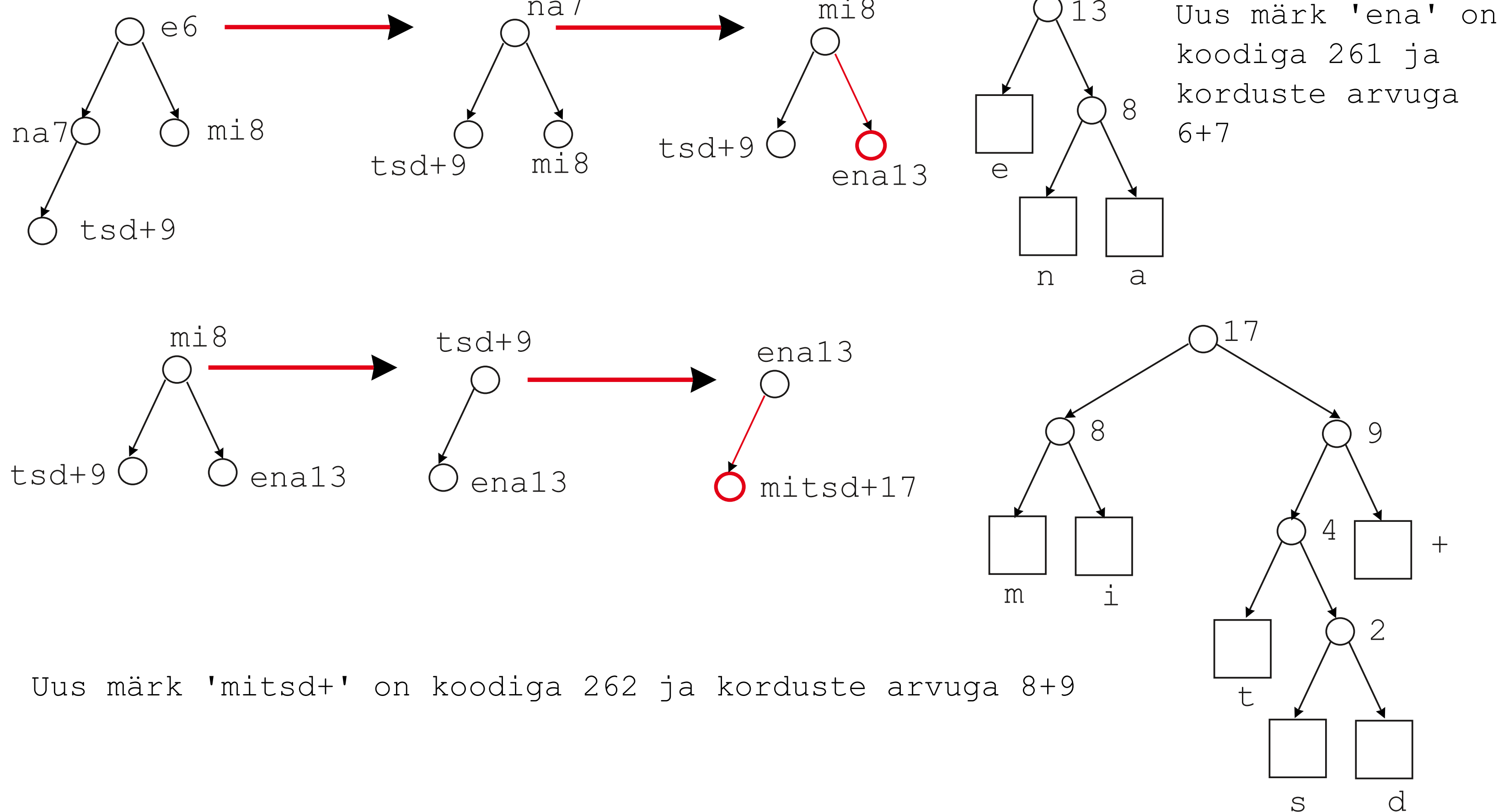


Huffmani kood (4)

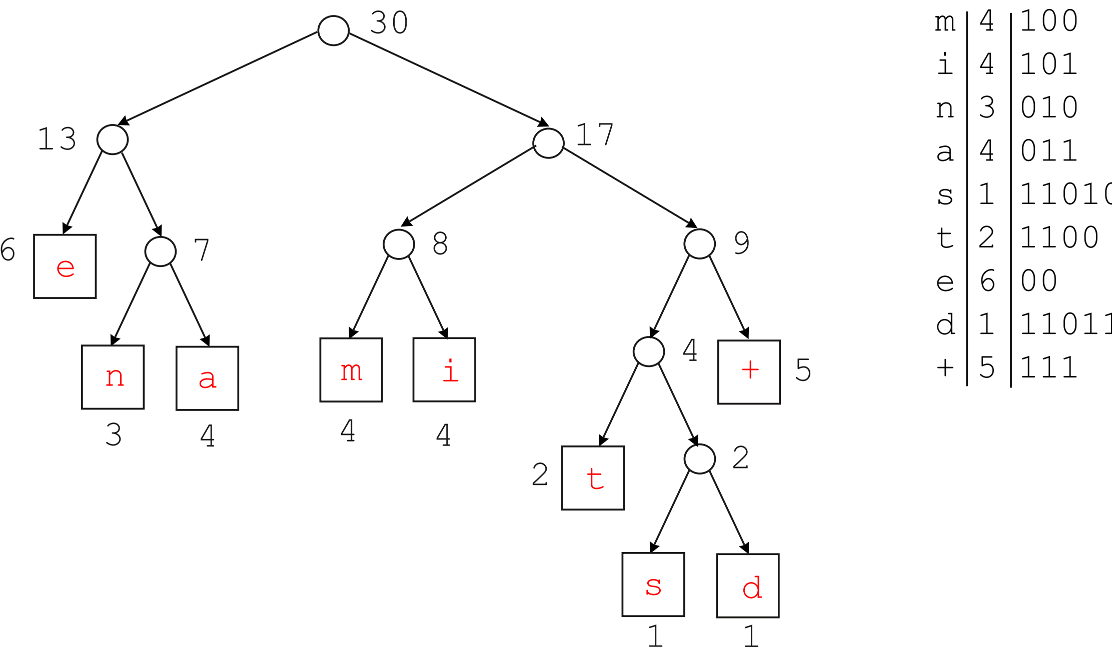


Uus märk 'tsd+' on koodiga 260 ja korduste arvuga 4+5

Huffmani kood (5)

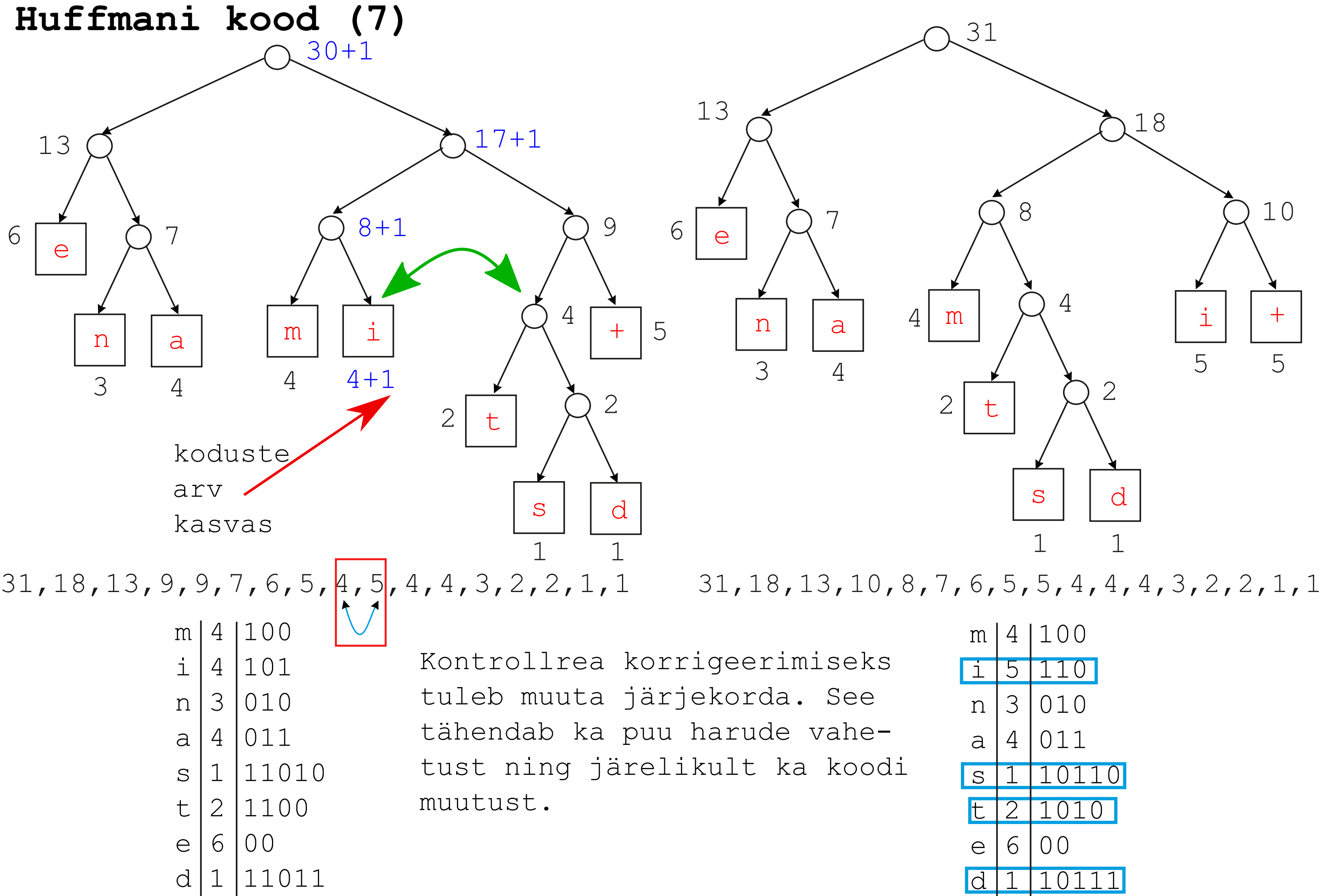


Huffmani kood (6)

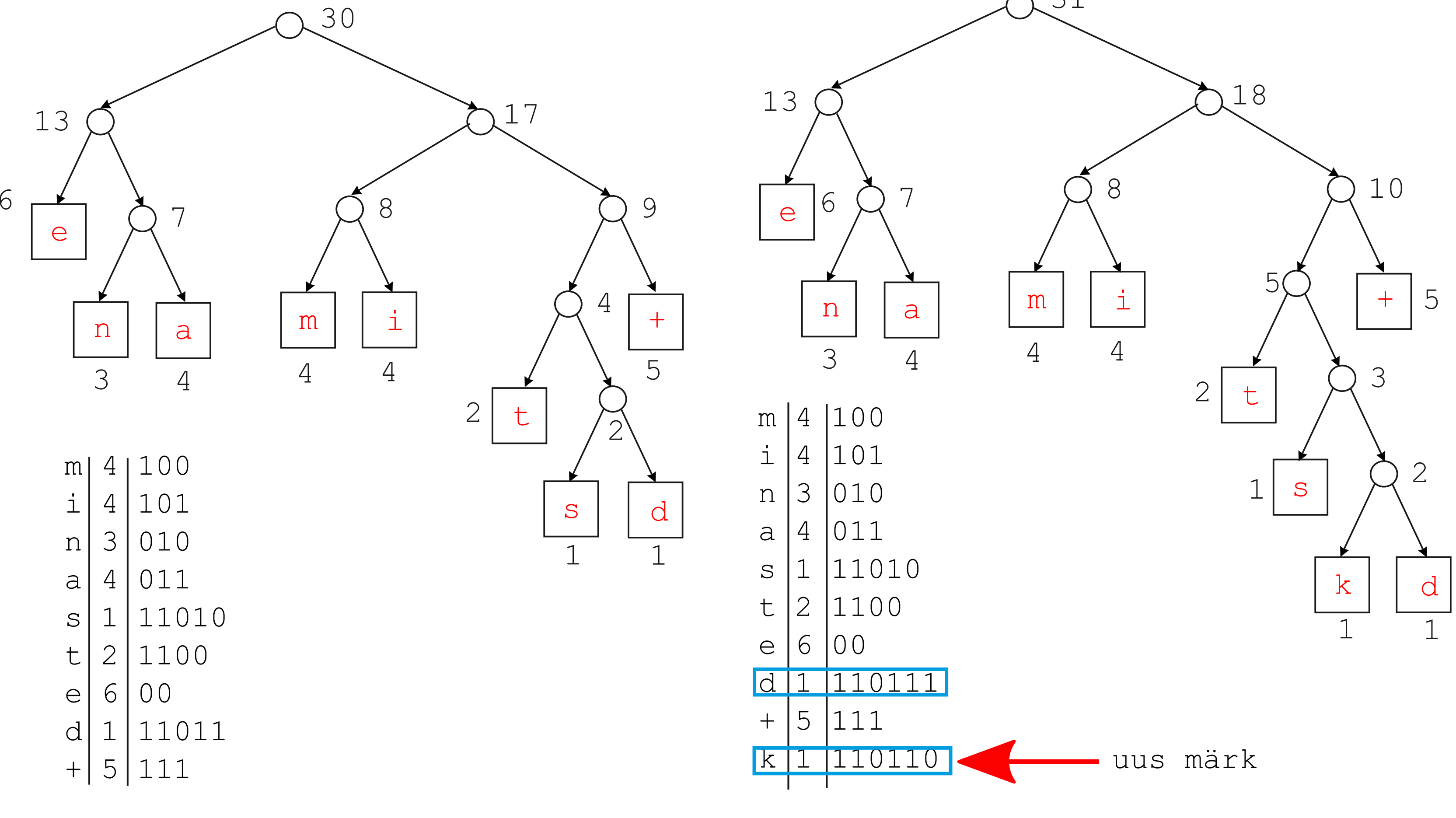


m	4	100
i	4	101
n	3	010
a	4	011
s	1	11010
t	2	1100
e	6	00
d	1	11011
+	5	111

30,17,13,9,8,7,6,5,4,4,4,4,3,2,2,1,1 - kontrollrida peab olema kahanev:
iga i puhul peab kehtima $a_i \geq a_{i+1}$


Huffmani kood (7)
30+1
13
17+1
6
e
7
8+1
9
n
a
m
i
4
+
5
3
4
4
4+1
2
t
2
koduste
arv
kasvas
s
d
1
1
31,18,13,9,9,7,6,5,4,5,4,4,3,2,2,1,1
m 4 100
i 4 101
n 3 010
a 4 011
s 1 11010
t 2 1100
e 6 00
d 1 11011
+ 5 111
Kontrollrea korrigeerimiseks
tuleb muuta järjekorda. See
tähendab ka puu harude vahe-
tust ning järelikult ka koodi
muutust.
31
13
18
6
e
7
8
10
n
a
4
m
4
i
+
3
4
5
5
2
t
2
s
d
1
1
31,18,13,10,8,7,6,5,5,4,4,4,3,2,2,1,1
m 4 100
i 5 110
n 3 010
a 4 011
s 1 10110
t 2 1010
e 6 00
d 1 10111
+ 5 111

Huffmani kood (8)



31,18,13,10,8,7,6,5,5,4,4,4,3,3,2,2,1,1,1

Lempel-Zivi pakkimise meetod (1)

Puhver pikkusega n, sisaldab märke $s_1, s_2, \ldots, s_n$

Fraasid: (s_1)

(s_1, s_2)

(s_1, s_2, s_3)

....

$(s_1, s_2, \ldots, s_n)$

Aken pikkusega m, sisaldab märke $s_1, s_2, \ldots, s_m$

Fraasid:

(s_1) (s_1, s_2) (s_1, s_2, s_3) .... $(s_1, s_2, \ldots, s_m)$

(s_2) (s_2, s_3) (s_2, s_3, s_4) .... $(s_2, s_3, \ldots, s_m)$

(s_3) (s_3, s_4) (s_3, s_4, s_5) .... $(s_3, s_4, \ldots, s_m)$

....

(s_{m-1}) (s_{m-1}, s_m)

(s_m)

[ ] [A B A B] C B A B A B C A D A

Aken, m=8 Puhver, n=4

Otsitakse fraase, mis esinevad nii aknas kui ka puhvris. Kui neid pole, võetakse puhvri esimene märk ja kirjutatakse pakitud faili. Aken ja puhver nihkuvad ühe koha võrra paremale.

[A] [B A B C] B A B A B C A D A

Lempel-Zivi pakkimise meetod (2)

A | B A B C | B A B A B C A D   A

Aknas fraas (A), puhvris (B), (BA), (BAB), (BABC).

Otsitakse fraase, mis esinevad nii aknas kui ka puhvris. Kui neid pole, võetakse puhvri esimene märk ja kirjutatakse pakitud faili. Aken ja puhver nihkuvad ühe koha võrra paremale.

A B | A B C B | A B A B C A D   A B

Aknas fraasid (A), (AB), (B).Puhvris (A), (AB), (ABC), (ABCB).

Võtame pikima kokkulangeva fraasi. Lõpptulemusse paneme kolmiku, kus:

1. Esimene number näitab kokkulangeva fraasi esimese märgi indeksit aknas.
2. Teine annab fraasi pikkuse.
3. Kolmas liige annab märgi, mis lähtetekstis järgneb meie fraasile.

Akent ja puhvrit nihutame fraasi pikkuse + 1 võrra.

A B A B C | B A B A | B C A D   A B (6,2,C)

Aknas fraasid (A), (AB), (ABA), (ABAB), (ABABC), (B), (BA), (BAB),(BABC), (ABC), (BC), (C).
Puhvris (B), (BA), (BAB), (BABA).
Pikim kokkulangev fraas (BAB).

A | B A B C B A B A | B C A D   A B (6,2,C) (4,3,A)

Lempel-Zivi pakkimise meetod (3)

A B A B C B A B A B C A D　　A B (6,2,C) (4,3,A)

Pikim kokkulangev fraas (BC).

A B A B C B A B A B C A D　　A B (6,2,C) (4,3,A) (2,2,A)

Kokkulangevaid fraase pole. Tulemus:　　A B (6,2,C) (4,3,A) (2,2,A) D

Lahtipakkimine:

A B (6,2,C) (4,3,A) (2,2,A) D

Nihutame akna kuni esimese kolmikuni:

A B (6,2,C) (4,3,A) (2,2,A) D

Kolmik ütleb,et lahtipakitud teksti tuleb panna fraas, mis

1. Asub aknas alates positsioonist 6
2. Koosneb kahest märgist.
3. Talle järgneb märk C.

A B A B C (4,3,A) (2,2,A) D

Lempel-Zivi pakkimise meetod (4)

A B A B C (4,3,A) (2,2,A) D

Järgmine kolmik ütleb,et lahtipakitud teksti tuleb panna fraas, mis
1. Asub aknas alates positsioonist 4.
2. Koosneb kolmest märgist.
3. Talle järgneb märk A.

A B A B C B A B A (2,2,A) D

Nüüd lisandub fraas BC, millele lisandub veel A.

A B A B C B A B A B C A D

Lahtipakitud tekst:

A B A B C B A B A B C A D