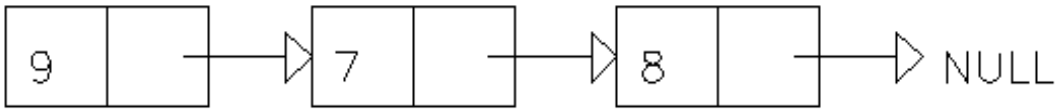
7 Tietorakenteita

Luvussa käsitellään tietorakenteita, joita voidaan kasvattaa dynaamisesti ajon aikana. Tällöin tilaa ei varata etukäteen, staattisesti, vaan tarpeen mukaan.

7.1 Listat

Yhteen suuntaan ketjutetussa listassa on kaksi osaa: osoitin seuraavaan alkioon ja itse tieto. Listarakenteen etuna on se, että listan kokoa voidaan dynaamisesti muuttaa. Tilaa ei tarvitse varata etukäteen (jolloin sitä saattaa tuhlaantua) eikä myöskään rakenteeseen tarvitse tehdä ohjelmallisia muutoksia, vaikka alkioiden määrä muuttuisi. Linkitetyn listan rakenteen käyttömahdollisuus sisältyy useimpiin ohjelmointikieliin.

Yhteen suuntaan linkitetyn listan rakenne on siis seuraavan kuvan kaltainen:



Listan alkion rakenne voisi olla esimerkin valossa seuraavanlainen:

Lista:

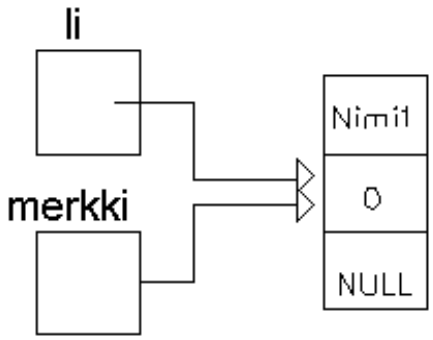
```
typedef struct AUTOK
{   char nimi[15];
    int ika ;
    struct AUTOK *seur;
}jasen;

jasen *li;
```

Ensimmainen listan alkio luodaan seuraavasti:

Listan ensimmäinen alkio:

```
jasen *merkki;
merkki = new (jasen);
merkki->seur = NULL;
*li = merkki;
cout << "Anna auton nimi ";
cin >> merkki->nimi;
merkki->ika = 5;
```

Rakenne näyttää aluksi tältä:

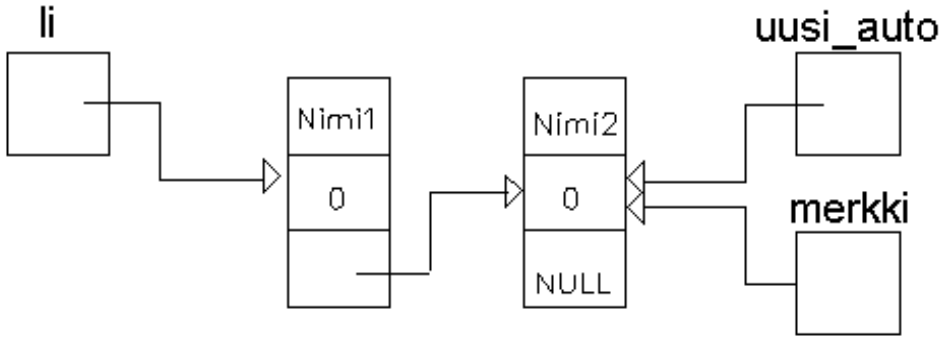


Lisäämme nyt 5 uutta autoa listaan.

Listan kasvatus:

```
jasen *uusiauto;
for (int i = 1; i < 5; i++)
{
uusiauto = new (jasen);
uusiauto->seur = merkki->seur;  // NULL
merkki->seur = uusiauto;
merkki = uusiauto;
uusiauto->ika = 10;

cout << " Anna uuden auton nimi \n";
cin >> uusiauto->nimi;

}
```

Ensimmäisen kierroksen jälkeen rakenne näyttää tältä:

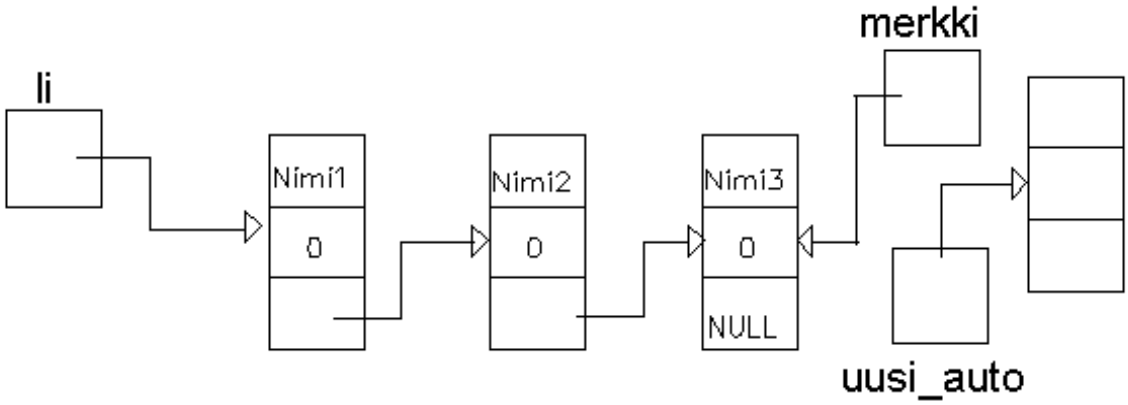


Näin jatketaan, kunnes kaikki 5 autoa on lisätty listaan.

Tarkastelemme seuraavassa hieman tarkemmin koko proseduuria.

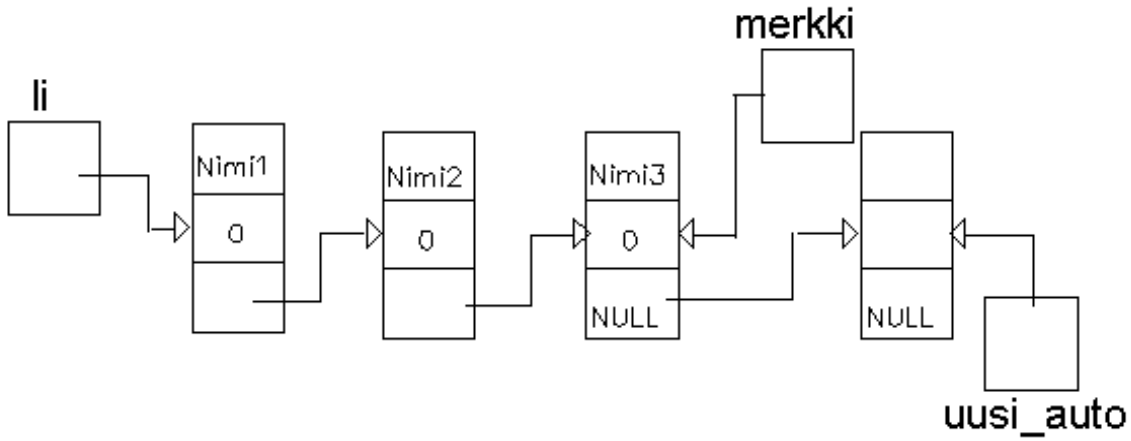
Ensin varasimme muistia uudelle autolle:

```
uusiauto = new jasen;
```



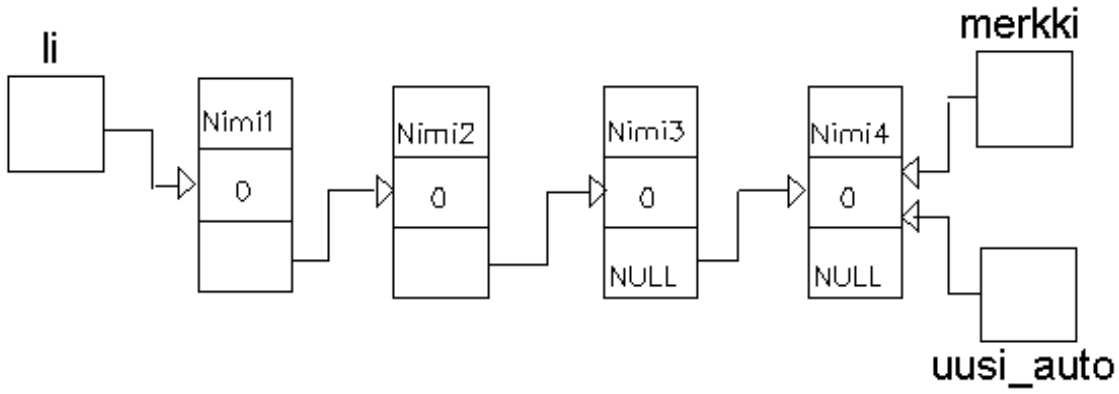
Sitten asetimme uudelle autolle osoittimen NULL-alkioon:

```
uusi_auto->seur = merkki->seur;
```



Seuraavaksi sijoitimme uuden auton listan loppuun:

```
merkki->seur = uusi_auto;
```



Annoimme uudelle autolle arvoja:

```
uusiauto->ika = 10;
cout << " Anna uuden auton nimi \n";
cin >> uusiauto->nimi;
```

Lopuksi voimme tulostaa autojen nimet.

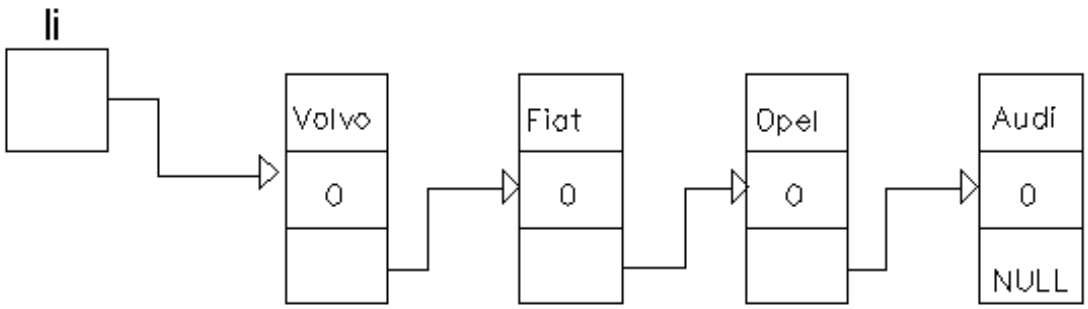
Listan tulostus:

```
void tulostalista(jasen *li)
{
    jasen *apu;
    cout << "\nListan tulostus: \n";
```

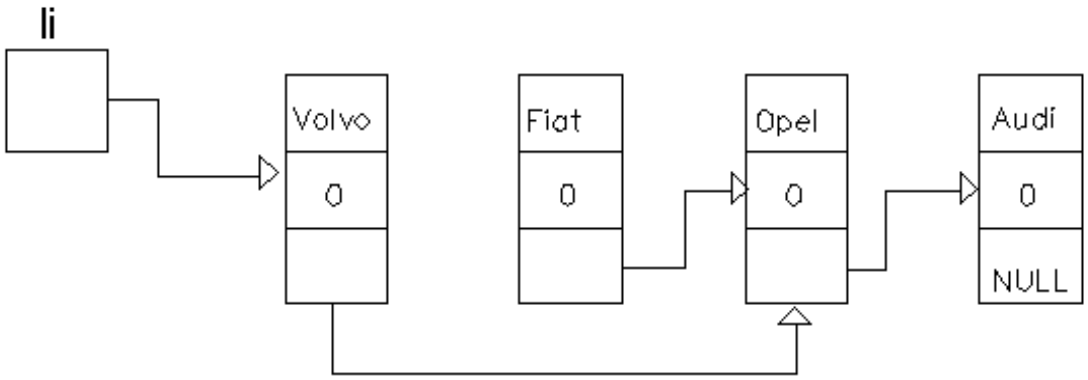
```
    for (apu=li;apu !=NULL;apu=apu->seur)
    cout << apu->ika << " " << apu->nimi << "\n";
}
```

Seuraavaksi tarkastelemme hieman *alkion poistamista* listasta.

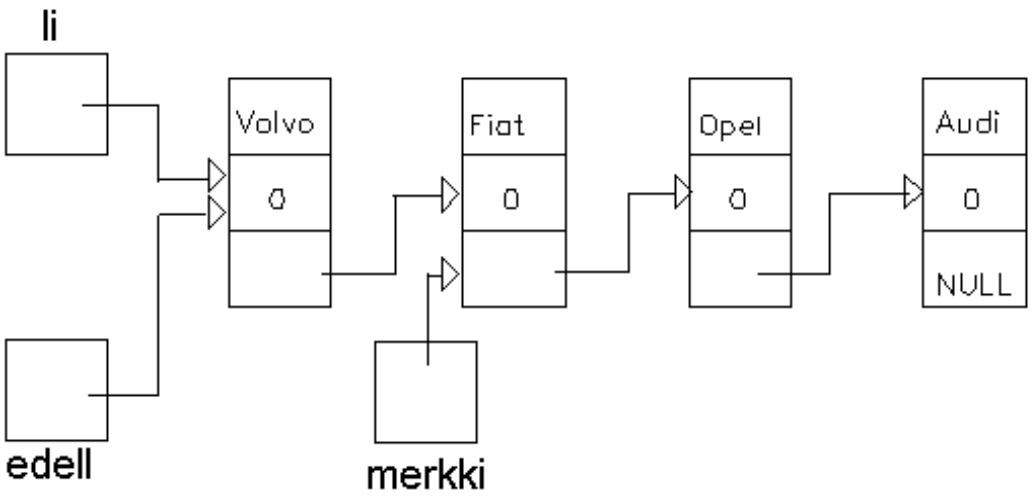
Oletetaan nyt, että listamme näyttäisi vaikka tältä:



Haluamme poistaa merkin Fiat listasta, jolloin on ensiksi järjestettävä niin, että Fiatin edellinen alkio (Volvo) osoittaa Fiatin jälkeen tulevaan alkioon (Opeliin). Tämän lisäksi meidän on vapautettava aiemmin Fiatille varattu muistitila.



Auton poistamiseksi on osoittimen *merkki* osoitettava poistettavaan autoon ja uuden osoittimen nimeltä *edell* poistettavaa alkiota edeltävään alkioon, kuten alla oleva kuva esittää. Alkio on kuitenkin ensimmäiseksi löydettävä listasta, joten sitä tutkitaan alusta alkaen, kunnes oikea alkio on löydetty.



Alkion poisto listasta:

Poistettaessa alkiota Fiat tarkistetaan ensin, onko alkio listan ensimmäinen alkio, eli alku->nimi. Jos ensimmäinen alkio on oikea, poistaminen tapahtuu:
alku = alku->seur; ellei kyseessä ole oikea alkio, siirrytään eteenpäin:

```
jasen * poista_alkio(jasen *li)
{
    jasen *merkki, *edell, *seur;
    int ika;
    cout << "\n\nMikä arvo poistetaan? ";
    cin >> ika;
    merkki = li;
    if (merkki ->ika == ika)
        { li = li->seur; delete merkki;       return li;}
    edell = li;

    for (merkki = li; merkki != NULL; merkki = merkki ->seur)
    {
    if (merkki ->ika == ika)
    {
    seur = merkki ->seur;
    edell->seur = seur;
    delete merkki;
    break;
    }
    edell = merkki;
    }
    return li;

     }
```

Seuraavana on menettely, jolla voidaan poistaa alkioita listan alusta:

```
jasen * poista_alusta(jasen *li)
{
    jasen *apu;
    int i,j=1;
    cout << "\n\nMontako alkiota poistetaan alusta? ";
    cin  >> i;
    if (i<1) cout << "\n Ei onnistu ! \n";
    else
    for (j=0; j < i; j++)
    {
apu = li;
if (apu == NULL)
        {
    cout << "\n LOPPUIVAT JO! \n";
            break;
        }
        li = li->seur;
delete apu;
    }
    return li;
 }
```

Lista voidaan tietenkin esittää myös luokkana, johon liittyy tietue-muotoisia alkioita. Seuraavana on osoitin tietueeseen listaluokan jäsenmuuttujana.

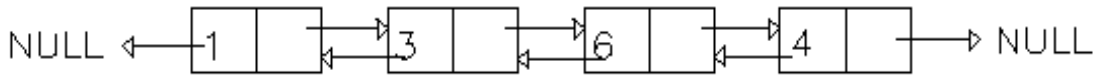
Oliolista:

```
#include <iostream.h>

typedef struct solmu
{
int tieto;
solmu * seuraava;
}solmu;

class Lista
{
solmu *li;
public:
void luolista( solmu **li);
void tulostalista(solmu *li);
};
```

```
void Lista::luolista(solmu **li)
{
    int i;
    solmu *apu;  /* esitellään apumuuttuja listatyyppiä */
    *li = NULL;   /* Maadoitetaan */
    for (i=0;i<10;i++)
     {
         apu = new(solmu);  /* luodaan dynaaminen muuttuja */
     apu->tieto=i*i;     /* Tietosisältö" */
     apu->seuraava = *li; /* apu osoittaa NULLiin  */
     *li=apu;
     }
}

void Lista:: tulostalista(solmu *li)
{
    solmu *apu;
    cout << "Listan tulostus: \n";
    for (apu=li;apu !=NULL;apu=apu->seuraava)
    cout << apu->tieto << "\n";
}

main()
{
    Lista LinkList;
    solmu *alku;              // alkuosoite solmu-tyypille
    LinkList.luolista(&alku);// &alku on osoittimen osoite
// (vastaa **li) */
    LinkList. tulostalista(alku);
    return(0);
}
```

Kahteen suuntaan linkitetty lista

Huomaamme, että kahteen suuntaan linkitetyllä listalla on etuja esimerkiksi poistettaessa alkioita, koska kukin alkio jo alunperin osoittaa sekä edelliseen että seuraavaan listan alkioon. Myös silloin, kun alkiot on pystyttävä lukemaan käänteisessä järjestyksessä, tulevat kaksisuuntaisen listan edut esille. Kaksisuuntaista listaa kutsutaan myös pakaksi. Seuraava kuva esittää kahteen suuntaan ketjutettua listaa:



Seuraavana on ohjelma, jolla voidaan poistaa alkio listan keskeltä. Tällöin on tärkeää, että sekä edelliseen että seuraavaan alkioon osoittavat osoittimet asetetaan osoittamaan oikein poiston jälkeen.

Kahteen suuntaan linkitetty lista:

```
#include <iostream.h>

typedef struct solmu
{  int tieto;
    struct solmu *seuraava;
    struct solmu *edellinen;
} lista;

lista *li;

void luolista(lista **li);
void tulostalista(lista *li);
lista *poista_alkio(lista *li);

main()
{
    lista *alku;     // alkuosoite lista-tyypille
    luolista(&alku); // &alku on osoittimen osoite
// (vastaa **li) */
    tulostalista(alku);
alku = poista_alkio(alku);
tulostalista(alku);
    return(0);
}

void luolista(lista **li)
{
    int i;
    lista *apu, *temp;  /* esitellään apumuuttuja listatyyppiä */
    *li = NULL;          /* Maadoitetaan */
    apu = new(lista);
    apu->tieto=100;
    apu->edellinen = NULL;
    apu->seuraava = NULL;
    *li = apu;

    for (i=1;i<5;i++)
     {
     apu = new(lista);
```

```
      apu->tieto=i*i;
      (*li)->edellinen = apu;
      apu->edellinen = NULL;
      apu->seuraava = *li;
      *li=apu;
      }
}

lista *poista_alkio(lista *li)
{
     lista *apu, *poista, *edell, *seur;
     int arvo,j=1;
     cout << "\n\nMikä arvo poistetaan? ";
     cin >> arvo;

     for (apu=li;apu !=NULL;apu=apu->seuraava)
     {
     edell = apu->edellinen;
     if (apu->tieto == arvo)
     {
     seur = apu->seuraava;
     edell->seuraava = seur;
     seur->edellinen = edell;
     delete (apu);
     }
     }
         return li;
 }



void tulostalista(lista *li)
{
// Listan tulostus havainnollisesti
     lista *apu;
     cout << "\nListan tulostus: \n";
     apu = li;
     cout << apu->tieto << "\t";
     cout << apu->seuraava->tieto << "\t";
     cout << apu->seuraava->seuraava->tieto << "\t";
     cout << apu->seuraava->seuraava->seuraava->tieto << "\t";
     cout << "\n";
apu =    apu->seuraava->seuraava->seuraava;
cout << apu->tieto << "\t";
     cout << apu->edellinen->tieto << "\t";
cout << apu->edellinen->edellinen->tieto << "\t";
cout << apu->edellinen->edellinen->edellinen->tieto << "\t";
     cout << "\n";
     for (apu=li;apu !=NULL;apu=apu->seuraava)
     cout << apu->tieto << "\t";
}
```

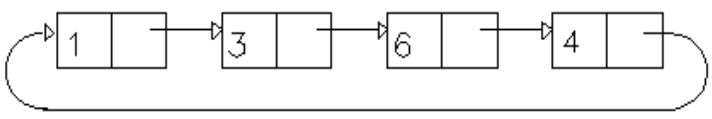
Lista voidaan tulostaa nyt myös käänteisessä järjestyksessä. Ensin on tiedettävä viimeisen alkion sijainti.

Listan tulostaminen käänteisessä järjestyksessä:

```
void tulostalista(lista *li)
{
     lista *apu;
     cout << "\nListan tulostus: \n";
     apu = li;
     lista * loppu;
     for (apu=li;apu !=NULL;apu=apu->seuraava)
     cout << apu->tieto << "\t";
     cout << "\n";

     // Haetaan viimeisen alkion sijainti
     for (apu=li;apu !=NULL;apu=apu->seuraava)
     loppu = apu;

     // Tulostetaan kääntäen lopusta alkuun
     for (apu=loppu;apu != li->edellinen;apu=apu->edellinen)
     cout << apu->tieto << "\t";

}
```

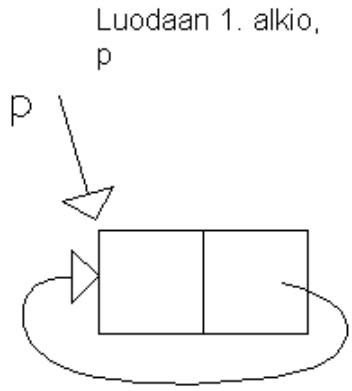
Muita yleisimpiä listarakenteita:

Yksisuuntainen rengas:

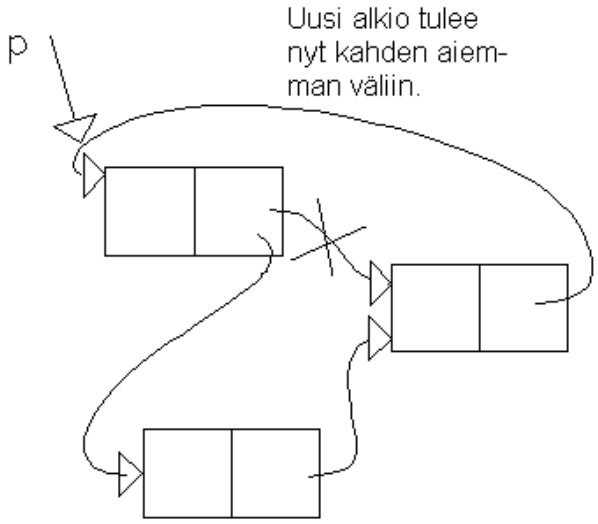


Rengas on hyvin käyttökelpoinen rakenne. Katsommekin seuraavaksi, kuinka renkaaseen lisätään alkioita.

Luodaan ensin yksi alkio:



Muut alkiot tulevat väliin:



Seuraavana on ohjelma, joka käyttää tunnussolmuja head ja tail. Jos haluat tehdä siitä renkaan, yhdistä tail ja head esimerkiksi lauseella tail->seur = head.

head-tail-lista:

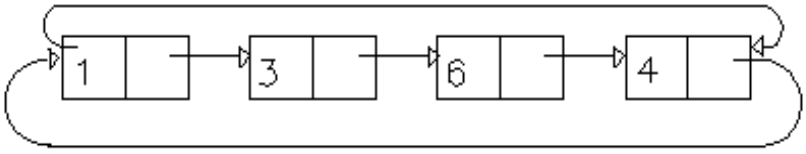
```
#include <iostream.h>
#include <conio.h>

typedef struct solmu
{  int tieto;
    struct solmu *seuraava;
} lista;    /* kyseistä rakennetta vastaavan muuttujan  nimi */

lista *li;
lista *head, *tail;
```

```
main()
{

     int i;
     lista *apu;  /* esitellään apumuuttuja listatyyppiä */
     li = NULL;            /* Maadoitetaan */
     apu = new(lista);
     apu->tieto=100;
     tail = apu;
//   cout << tail->tieto << "\n";
     apu->seuraava = NULL;
     li = apu;
     for (i=1;i<5;i++)
          {
      apu = new(lista);
      apu->tieto=i*i;
      cout << apu->tieto << "\n";
      apu->seuraava = li;
      li=apu;
      head = li;
//    tail->seuraava = head;
          }


// TULOSTUS
     cout << "\nListan tulostus: \n";
     for (apu=head;apu !=tail->seuraava;apu=apu->seuraava)
     cout << apu->tieto << "\t";



// POISTO KESKELTÄ
     lista *edell, *seur;
     int arvo;
     cout << "\n\nMikä arvo poistetaan? ";
     cin >> arvo;
     apu = li;
     edell = apu;
     do
     {
     cout << apu->tieto << "\t"  << edell->tieto << "\n";
      if (apu->tieto == arvo)
       {
          seur = apu->seuraava;
          edell->seuraava = seur;
          delete (apu);
          break;
       }
          edell = apu;
          apu = apu->seuraava;
     }
```

```
        while ( apu != NULL);


    // TULOSTUS POISTON JÄLKEEN
        cout << "\nListan tulostus: \n";
        for (apu=head;apu !=tail->seuraava;apu=apu->seuraava)
        cout << apu->tieto << "\t";


        return(0);
    }
```

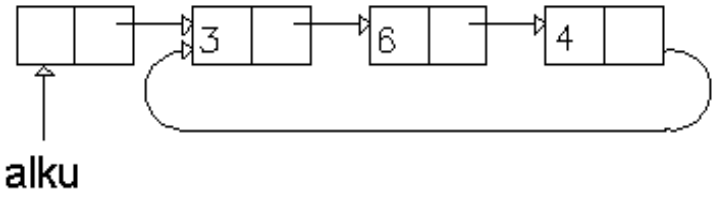
Jos listasta ei tehdä rengasta, tulisi head ja tail maadoittaa.

Kaksisuuntainen rengas



Kaksisuuntaista rengasta kutsutaan joskus pakaksi (vrt. korttipakka). Sen etuna on mm. helppous käsitellä alkioita eri järjestyksessä. Myös alkioiden lisäyksen ja poiston suorittaminen helpottuvat.

Tunnussolmulla varustettu yksisuuntainen rengas



7.2 Pino ja jono

Pino on hyvin hyödyllinen ja usein käytetty tietorakenne. Tietokoneessa esimerkiksi keskeytykset ja ympäristön vaihdot hyödyntävät pinoja.

7.2.1 Pino

Pino muistuttaa huomattavasti lineaarista listaa. Pinorakenteessa kaikki arvoihin kohdistuvat operaatiot voidaan toteuttaa vain listan jommankumman pään viimeisen alkion suhteen. Se listan pää, jonka alkioita voidaan käsitellä, on pinon huippu ja vastapää taas pinon pohja. Pino on yleisesti käytetty rakenne esimerkiksi tietokoneiden perusoperaatioissa kuten keskeytysten käsittelyssä.
Pino voidaan esittää taulukkona tai listana. Pinon kaksi tärkeää operaatiota ovat POP ja PUSH. POP-operaatio poistaa päällimmäisen pinon alkion ja PUSH taas lisää pinoon uuden alkion.
Listamuotoinen pino ja edellä mainitut operaatiot voidaan esittää esimerkiksi seuraavasti:

Pinoalgoritmeja (rakenne, pop, push):

```
struct PINO
{
    int tieto;
    struct PINO *seur;
};
struct PINO *p;


void pop (struct PINO **p)
{
    struct PINO *apu;
    apu = *p:
    *p = *p->seur;
    delete apu;
}


void push (int x, struct PINO **p)
{
    struct PINO *apu;
    apu = new PINO;
    apu->tieto = x;
    apu->seur = *p;
    *p = apu;
}
```

Seuraavana on luokan STACK jäsenmuuttujana osoitin pinon alkioon. Itse alkio on tietue-tyyppinen.

Oliopino:

```
#include <iostream.h>

typedef struct solmu
{
    int tieto;
    solmu *seur;
} solmu;


class STACK
{
solmu *top;
public:
STACK(): top(0) {}
~STACK() { Clear(); }
void Clear();
void Push(int i);
int Pop();
int Koko();
};

void STACK::Clear()
{
        solmu *apu=top;
        while (apu != NULL)
        {
            top = top->seur;
            delete apu;
            apu = top;
        }
}

void STACK:: Push(int i)
{
        solmu *apu = new solmu;
        apu->tieto = i;
        apu->seur = top;
        top = apu;
}

int STACK:: Pop()
{
  if (top != 0)
  {
        int temp = top->tieto;
        cout << temp << "\n";
```

```
        solmu *apu=top;
        top = top->seur;
        delete apu;
        return temp;
    }
        else return 0;
}

int STACK:: Koko()
{
    int i=0;
    solmu *apu=top;
    while (apu != NULL)
    {
        i++;
        apu = apu->seur;
    }
    return i;
}


int main()
{
STACK pino;
for (int i = 0; i < 3; i++)
pino.Push(10*i);
for (i = 0; i < 3; i++)
pino.Pop();
}
```

7.2.2 Jono

Jono eroaa pinosta siinä, että jonossa lisäysoperaatiot tehdään loppupäähän ja poistot taas jonon alkupäähän. Jonorakenteessa noudatetaan ns FIFO-periaatetta, eli kullakin hetkellä olevista alkioista voidaan poistaa vain ensiksi sinne lisätty alkio. Jonoja (QUEUEs) käytetään mm käyttöjärjestelmän operaatioissa.

Jonoja voidaan muodostaa ja käsitellä STL-kirjaston avulla. Seuraavana on esimerkki jonon käsittelystä.

Prototyypit ja esittelyt: queue-luokka toimii jonon malliluokkana:

```
#include <queue>
template <class T, class Container = deque<T>>

class queue {
```

```
    public:
       typedef typename Container::value_type value_type;
       typedef typename Container::size_type size_type;
    protected:
       Container c;
    public:
       bool empty () const { return c.empty; }
       size_type size () const { return c.size; }
       value_type& front () { return c.front; }
       const value_type& front () const { return c.front(); }
       value_type& back () { return c.back; }
       const value_type& back () const { return c.back(); }
       void push (const value_type& x) { c.push_back(x); }
       void pop () { c.pop_back(); }
    };
```

Jono STL-kirjastoa käyttäen:

```
#include <queue>
#include <string>
    #include <deque>
    #include <list>
    #include <iostream.h>
    #include <conio.h>

    using namespace std;

    main()
    {
    // int-jono

       queue<int, list<int> > jono;

       for (int k = 100; k < 400; k = k + 100)
       jono.push(k);

       cout << "Etumaisin jonossa " << endl;
       cout << jono.front() << endl;

       cout << "Jonon hännillä on " << endl;
       cout << jono.back() << endl;

       cout << "Otetaan pois edestä pari jäsentä " << endl;
       jono.pop();    jono.pop();

       cout << "Nyt etumaisin jonossa " << endl;
       cout << jono.front() << endl;
```

```
   cout << "Poistetaan yksi edestä " << endl;
   jono.pop();

   cout << "Onko jo tyhjä " << endl;
   if (jono.empty()) cout << "On tyhjä" << endl;



// float-jono

   queue<float,deque<float> > desijono;
   for (float i = 0; i < 20; i = i + 2.5)
   {
     desijono.push(i);
   }

    cout << "Desijonon hännillä on " << endl;
   cout << desijono.back() << endl;

   cout << "Desijonon pituus on " << endl;
   cout << desijono.size() << endl;

   cout << "Poistetaan yksi edestä " << endl;
   desijono.pop();

   cout << "Desijonon pituus on nyt" << endl;
   cout << desijono.size();


   getch();
 }
```

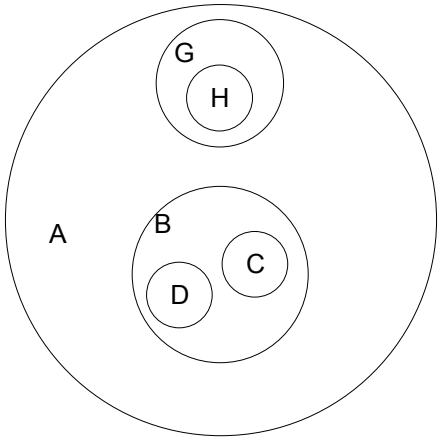
7.3 Puurakenteet

Useissa sovelluksissa on tiedot voitava järjestää usealle eri tasolle. Monien eri alojen ongelmia voidaan kuvata puurakenteina. Ongelman määrittäminen puun muodossa tukee myös ratkaisun etsimistä ja päätöksentekoa. Esimerkkinä tällaisesta päätöksentekoa tukevasta puurakenteesta on päätöspuumenetelmä, jota käytetään esimerkiksi monivaiheisissa investoinneissa: kukin päätöksentekoa vaativa kohta voidaan kuvata puun solmuna. Useimmiten tällöin muodostuu aina kaksi uutta haarautumaa, koska päätös on joko myönteinen tai kielteinen. Myös prosesseja tai mitä tahansa kokonaisuuksia, joissa on keskinäisiä riippuvuuksia eri tasoilla, on järkevää kuvata puurakenteina. Tietotekniikassa puurakenteet tukevat ohjelmien ja tietokantojen määrittelyä.

Puun määritelmän mukaan solmujoukko T muodostaa puun, jos

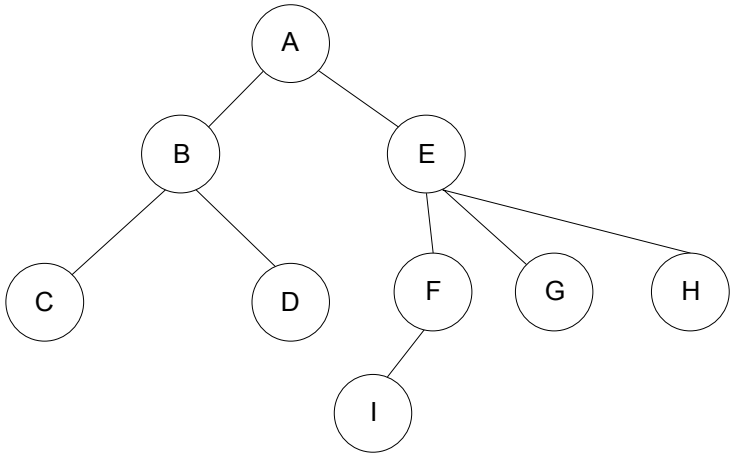
1. T:ssä on erikoisasemassa oleva solmu, jota nimitetään juureksi.
2. T:n muut solmut on jaettu n:ään erilliseen joukkoon T_1, T_2, ..., T_n eli kukin joukoista T_i (1 = 1, ..., n) on puu (T:n alipuu).

Joukko-opin merkinnöin puun joukot näkyvät seuraavasti:



Edellä oleva puu voitaisiin kuvata myös listana seuraavasti: A(G(H), B(C,D))

Käsite solmun *aste* tarkoittaa alipuiden lukumäärää.
Esimerkki kuvannee tapausta parhaiten:



Esimerkkipuussamme A:n aste on 2, E:n aste on 3 ja H:n aste on 0.

Solmua, jonka aste on 0, nimitetään *lehtisolmuksi*, kun taas muut solmut ovat *haarautumasolmuja*. Lisäksi esimerkkipuussamme solmut C ja D ovat B:n *poikia*, ja B taas on C:n ja D:n *isä*.

Solmun *taso* kuvaa tasoa, jolla solmu sijaitsee suhteessa juureen, jonka taso on 1. Esimerkkipuussamme B:n ja E:n taso on 2 ja I:n taso 4.

Solmun *syvyys* määritellään seuraavasti: lehtisolmun syvyys on yksi ja haarautumasolmun syvyys on yhtä yksikköä suurempi kuin suurempi sen poikien syvyydestä.

Puun *korkeus* on sen juuren syvyys tai puuhun kuuluvien solmujen suurin taso.

Puut voivat olla joko *järjestettyjä* tai *järjestetämättömiä*. Tietotekniikassa käytetään pelkästään järjestettyjä puita eli puita, joissa alipuiden järjestys on määritetty.

Puun solmut muodostavat *polun*, jossa peräkkäin ovat aina isä- ja poikasolmu.

Puut muodostavat yhdessä *metsän*.

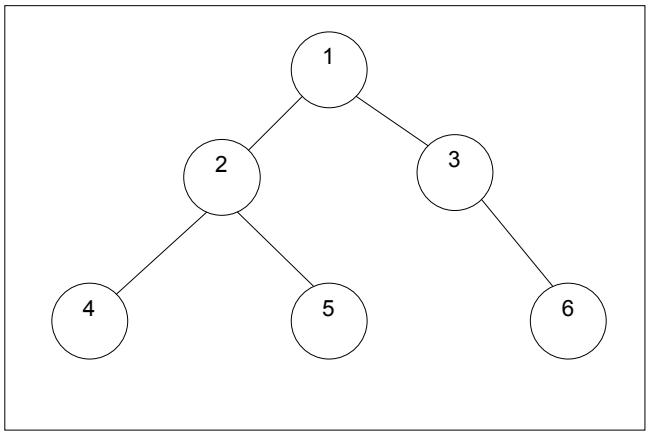
7.3.1 Binääripuu

Binääripuu muistuttaa hyvin paljon linkitettyä listaa. Kyseessä on järjestetty puu, jonka jokaiseen solmuun liittyy aina tarkalleen kaksi alipuuta. Binääripuu voi olla myös tyhjä, jolloin soluun liittyy NULL-arvo.

Binääripuun solmuja voidaan käsitellä ainoastaan siirtymällä polkua pitkin solmusta toiseen eli suoraan tiettyyn solmuun on mahdotonta viitata. Yleensä solmu haetaan esille kulkemalla polkua alkaen juuresta tai mahdollisesti käymällä läpi kaikki solmut ennalta määritetyn järjestyksen mukaan. Viimeksimainitussa tapauksessa käsittelyjärjestykset ovat *esijärjestys*, *sisäjärjestys* ja *jälkijärjestys*.

Esijärjestys etenee seuraavasti: ensin käsitellään juuri, sen jälkeen vasen alipuu ja oikea alipuu. Sisäjärjestyksessä käsitellään ensin vasen alipuu, sitten juuri ja lopuksi oikea alipuu. Jälkijärjestyksessä käsitellään ensin vasen alipuu, sitten oikea alipuu ja lopuksi juuri.

Esimerkki kuvannee parhaiten käsittelyjärjestyksiä. Olettakaamme, että puumme on seuraavanlainen:

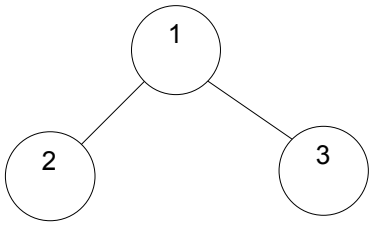


Solmujen käsittelyjärjestys olisi seuraava eri järjestysmenettelyillä:

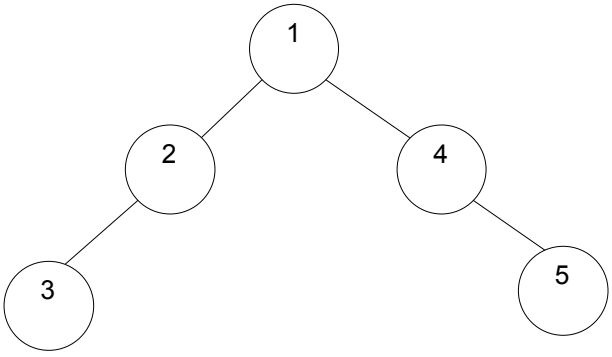
1. Esijärjestys: 1 2 4 5 3 6
2. Sisäjärjestys: 4 2 5 1 3 6
3. Jälkijärjestys: 4 5 2 6 3 1

Järjestys ei vielä määritä puun rakennetta täsmällisesti.

Binääripuun sanotaan olevan *aito*, jos jokaisessa solmussa on 0 tai 2 ei-tyhjää poikaa. Seuraava puu on aito:

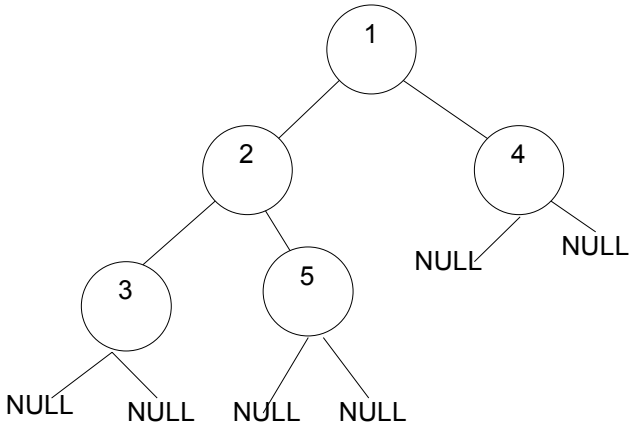


Seuraava puu ei ole aito



Binääripuu on *melkein täydellinen*, jos puun kaikki lehdet ovat kahdella peräkkäisellä tasolla ja alemman tason lehdet ovat mahdollisimman vasemmalla ja jos ylemmillä tasoilla olevilla tasoilla on kaksi ei-tyhjää poikaa.

Seuraavana eräs binääripuu, joka täyttää melkein täydellisen ehdot:

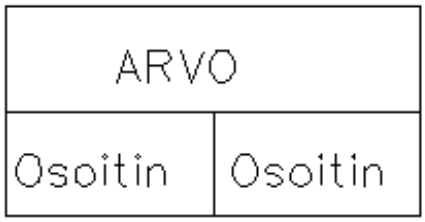


Binääripuu voidaan muodostaa seuraavalla rakenteella:

Binääripuun rakenne:

```
struct binpuu
{
    int luku;
    struct binpuu *oikeasolmu;
    struct binpuu *vasensolmu;
};
```

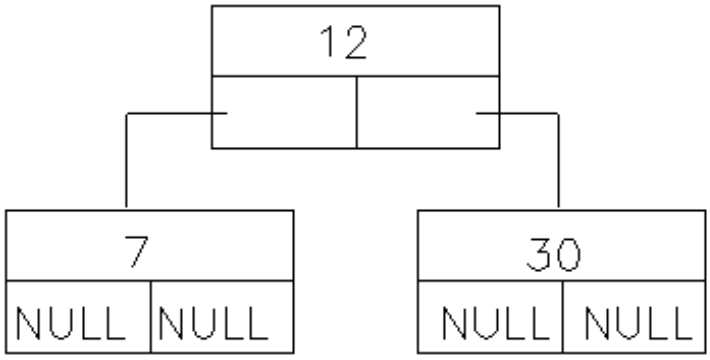
Binääripuun solmu näyttäisi seuraavalta:



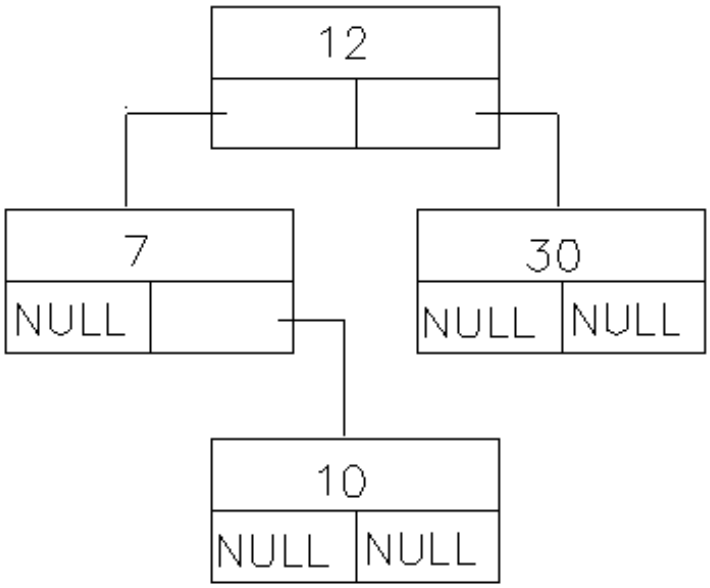
Binääripuuta käytetään usein tiedon lajitteluun. Solmun vasemmalle puolelle laitetaan yleensä solmua pienemmät arvot ja oikealle vastaavasti suuremmat arvot.

Simuloimme seuraavaksi manuaalisesti uusien arvojen asettamista binääripuuhun, jossa solmua pienemmät arvot asetetaan solmun vasempaan ja suuremmat solmun oikeaan alipuuhun.

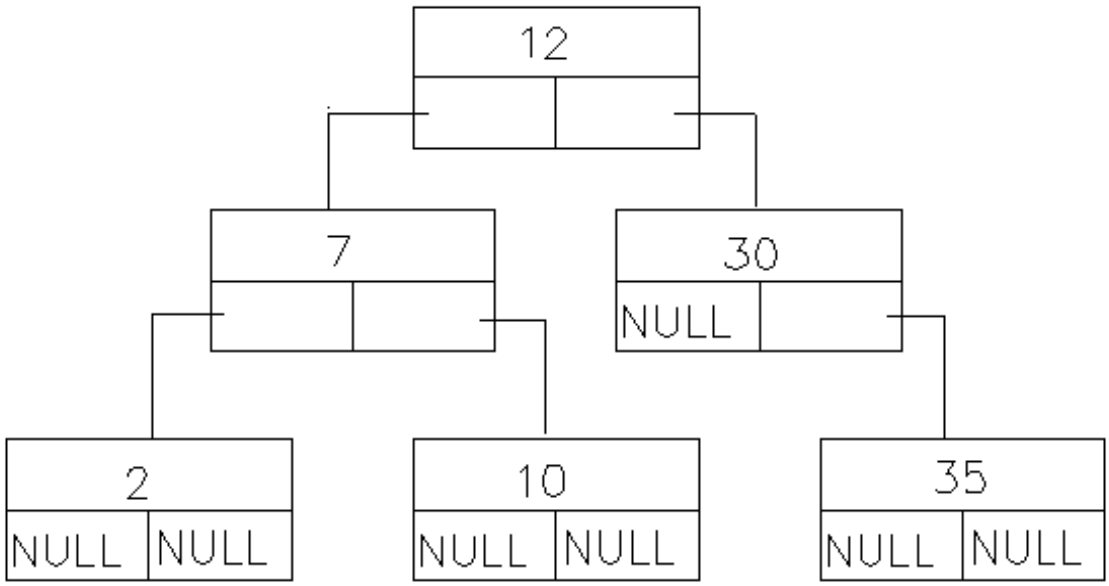
Meillä on esimerkiksi seuraavanlainen puu:



Haluaisimme nyt sijoittaa luvun 10 puuhun. Sijoituksen jälkeen puu näyttää seuraavalta:



Jos sijoitamme luvut 2 ja 35, saamme puun:



Luomme seuraavaksi puun ja sijoitamme siihen arvoja.

Arvojen sijoitus binääripuuhun:

```
#include <iostream.h>
#define TOSI 1
#define EITOSI 0
#define TYHJA '\0'
//Sijoitetaan arvot 12, 7, 30, 2, 10,35

typedef struct binpuu
{
    int luku;
    struct binpuu *oikeasolmu;
    struct binpuu *vasensolmu;
}puu;

puu *juuri, *nyk_solmu, *uusi_solmu;
void tulostapuu(puu *solmu);
int ota_luku();

void main()
{
    int antoluku;            /*syötetty luku*/
    int lkm;              /* annettujen lukujen lukumäärä*/
```

```
    int i;                /*luettujen lukujen määrä*/
    int ei_paikkaa;       /*arvon sijainti; on tosi, jos
                paikkaa ei löydetä*/
    lkm = ota_luku();

    if (lkm > 0)       /* syötetään ensimmäinen eli juuren arvo
ensimmäiseksi */
    {
        nyk_solmu = juuri;
        nyk_solmu->luku = ota_luku();
        nyk_solmu->oikeasolmu = TYHJA;
        nyk_solmu->vasensolmu = TYHJA;
    }

/* syötetään nyt muut arvot */

    for (i=2; i<= lkm; i++)
    {
        antoluku = ota_luku();
        nyk_solmu = juuri;        /* aloitetaan alusta*/
        ei_paikkaa = TOSI;        /* paikka ei vielä tiedossa*/

        uusi_solmu = new (puu);

        uusi_solmu->luku = antoluku;
        uusi_solmu->oikeasolmu = TYHJA;
        uusi_solmu->vasensolmu = TYHJA;

/* haetaan arvon sijainti puussa*/

        while (ei_paikkaa)
        {
            if (antoluku < nyk_solmu->luku)
              if (nyk_solmu->vasensolmu == TYHJA)
                    ei_paikkaa = EITOSI;
              else nyk_solmu = nyk_solmu->vasensolmu;

              else if (nyk_solmu->oikeasolmu == TYHJA)
                    ei_paikkaa = EITOSI;
                else nyk_solmu = nyk_solmu->oikeasolmu;
        }
/* asetetaan uusi solmu omalle paikalleen*/
        if (antoluku < nyk_solmu->luku)
            nyk_solmu->vasensolmu = uusi_solmu;
        else    nyk_solmu->oikeasolmu = uusi_solmu;
}

tulostapuu(juuri);

}
```

```
int ota_luku()
{
int apu;
cout << "anna luku puuhun \n";
cin >> apu;
return apu;
}
```

Binääripuu voidaan tulostaa suurimmasta arvosta pienimpään seuraavalla algoritmilla.

Binääripuun tulostaminen:

```
void tulostapuu(puu *solmu)
{
        if (solmu->vasensolmu != TYHJA)
            tulostapuu(solmu->vasensolmu);
        cout << solmu->luku << "\n";
        if (solmu->oikeasolmu != TYHJA)
            tulostapuu(solmu->oikeasolmu);
}
```