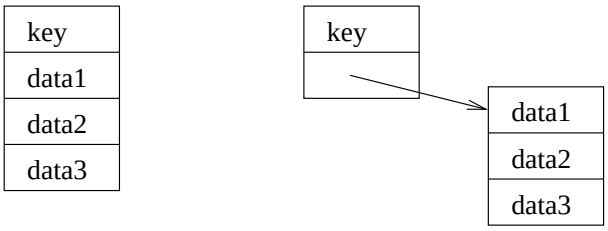
Abstrakti tietotyyppi joukko

- usein ohjelma ylläpitää suoritusaikanaan jotain joukkoa tietoalkioita, esim: puhelinluettelo nimi-numero -pareja
- joukon tietoalkiot muodostuvat *tietueista* missä yksi kenttä on ns. *avain* jonka perusteella tietueita voidaan hakea, ja johon perustuen tietueet voidaan järjestää, esim: puhelinluettelossa nimi on avain
- muu tieto voi olla tietueen muissa kentissä tai talletettuna viitteen takana "satelliitti-tiedoksi"



- joukon käsittelyyn tarvitaan seuraavia operaatioita:

- **search(S,k)**	jos joukosta löytyy avaimella k varustettu alkio, operaatio palauttaa viitteen alkioon, muuten operaatio palauttaa viitteen NIL
- **insert(S,x)**	lisää joukkoon alkion johon x viittaa
- **delete(S,x)**	poistaa tietueen johon x viittaa
- **min(S)**	palauttaa viitteen joukon pienimpään alkioon
- **max(S)**	palauttaa viitteen joukon suurimpaan alkioon
- **succ(S,x)**	palauttaa viitteen alkiota x seur. suurimpaan alkioon, jos x oli suurin palauttaa NIL
- **pred(S,x)**	palauttaa viitteen alkiota x seur. pienempään alkioon, jos x oli pienin palauttaa NIL

- huomaamme siis että puhelinluettelon operaatiot (paitsi **list**) muodostavat osajoukon joukkojen operaatioista
- näin määriteltynä joukko on *abstrakti tietotyyppi*, joka toteuttamiseen on useita vaihtoehtoisia tietorakenteita:
  - taulukko (edellyttäen että tiedämme joukon maksimikoon)
  - linkitetty lista
  - hakupuu
  - tasapainoitettu hakupuu
  - hajautusrakenne
  - keko
- joukon operaatioiden tehokkuus riippuu suuresti siitä minkä tietorakenteen avulla joukko on toteutettu, esim:

	lista1	lista2	tasap.puu	hajautusrak
search	O(n)	O(n)	O(log n)	O(1)
insert	O(1)	O(n)	O(log n)	O(1)
delete	O(1)	O(1)	O(log n)	O(1)
min	O(n)	O(1)	O(log n)	O(n)
max	O(n)	O(1)	O(log n)	O(n)
succ	O(n)	O(1)	O(log n)	O(n)
pred	O(n)	O(1)	O(log n)	O(n)

- toteutustavan valinnassa siis on ratkaisevaa se mitä operaatioita joukkoa käyttävä ohjelma tarvitsee eniten
- kurssin seuraavien viikkojen ohjelmassa on käsitellä joukon toteutukseen sopivia tietorakenteita

2. Lista, pino ja jono

- ennen kuin menemme varsinaiseen lista-tietorakenteeseen, opiskelemme kahta joukon "erikoistapausta", pinoa ja jonoa
- pino ja jono eivät toteuta kalvon 22 joukko-operaatioita, vaan toimivat niinkuin pino ja jono toimivat normaalielämässä
- pino ja jono ovat tärkeitä rakenneosasia monissa algoritmeissa, kuten tulemme myöhemmin näkemään ...

2.1 Pino

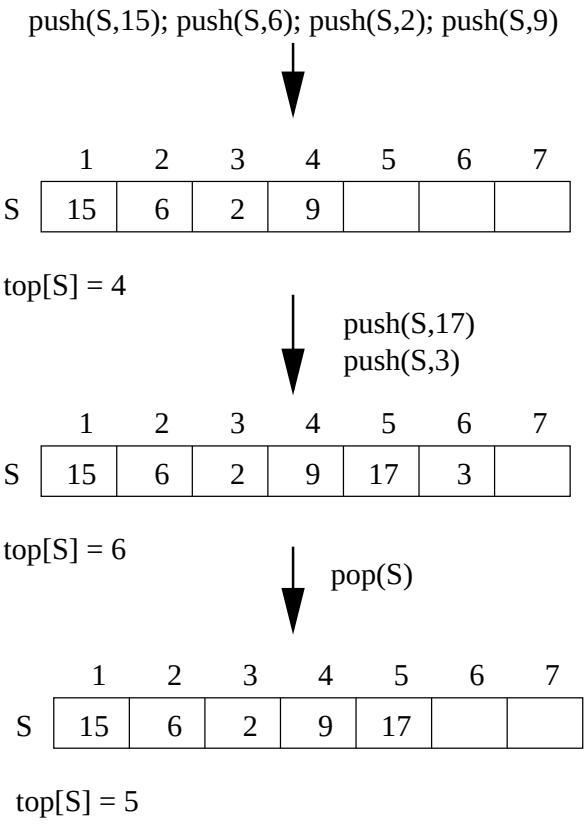
- last-in-fist-out -periaate
- pino-operaatiot:

  - **push(S,x)** laittaa tietoalkion (mihin x viittaa) pinon päälle
  - **pop(S)** palauttaa viitteen pinon päällimmäiseen tietoalkion ja poistaa sen pinosta
  - **empty(S)** palauttaa *true* jos pino tyhjä, muuten *false*

- oletetaan seuraavassa yksinkertaisuuden vuoksi että pinoon talletettavat alkiot sisältävät ainoastaan kentän key
- eli operaation parametrit yksinkertaistetussa tapauksessa:

  - **push(S,k)** laittaa tietoalkion k pinon päälle
  - **pop(S)** palauttaa pinon päällimmäisen tietoalkion

- jos yläraja pinon kokoon tiedetään ennakolta, voidaan pino toteuttaa helposti taulukon avulla
  - talletetaan pinon alkiot taulukkoon $S[1, \ldots, n]$
  - siis pinossa olevien alkioiden maksimimäärä on n
  - pinolla attribuutti *top* joka kertoo huipulla olevan alkion paikan taulukossa
  - alussa $top[S] = 0$ ja pino tyhjä
  - ensimmäinen alkio laitetaan paikkaan $S[1]$, toinen paikkaan $S[2]$, jne
- pinon toimintaa:



- operaatioiden toteutus

```
empty(S)
    if top[S]=0
        then return true
        else return false

push(S,k)
    top[S] ←top[S]+1
    S[top[S]] ←k

pop(S)
    top[S] ←top[S] − 1
    return S[top[S]+1]
```

- toteutus olettaa että ennen pop-operaation käyttämistä ohjelma testaa että pino ei ole tyhjä
- entä jos pino on täysi push-operaatiota yritettäessä?
- operaatio jolla testataan onko pino täynnä:

```
full(S)
    if top[S] = n
        then return true
        else return false
```

- normaalisti full-operaatio ei kuitenkaan kuulu pinon operaatiovalikoimaan
- kaikki pino-operaatiot sisältävät vain saman vakiomäärän komentoja riippumatta siitä kuinka monta alkiota pinossa on, operaatiot ovat siis vaativuudeltaan $\mathcal{O}(1)$

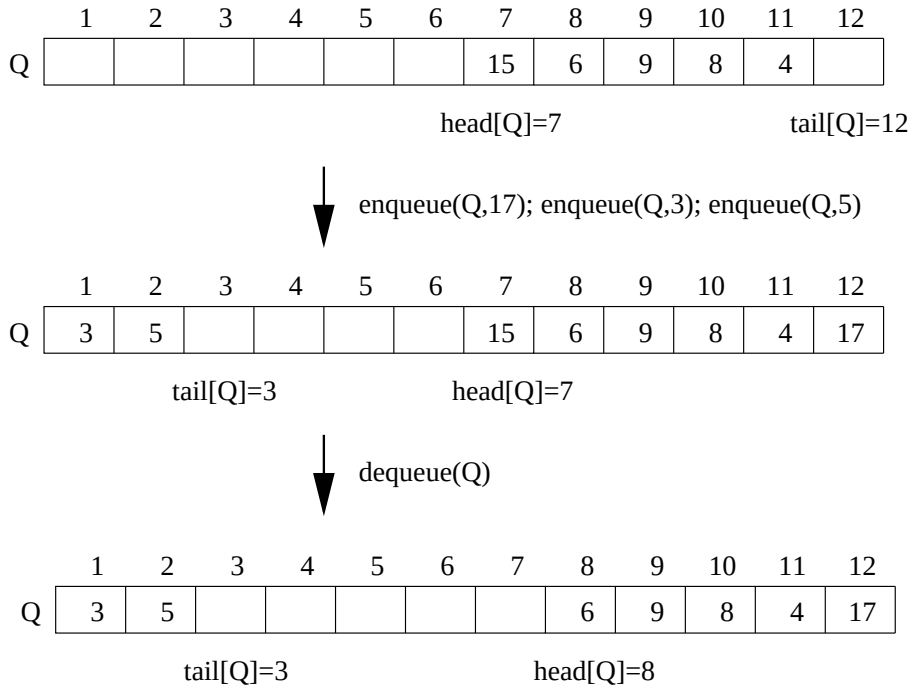
2.2 Jono

- first-in-first-out
- jono-operaatiot:

  - **enqueue(Q,k)** laittaa tietoalkion k jonon perälle
  - **dequeue(Q)** palauttaa jonon ensimmäisenä olevan tietoalkion ja poistaa sen jonosta
  - **empty(Q)** palauttaa *true* jos jono tyhjä, muuten *false*

- jälleen, jos jonon koon yläraja tiedetään ennakolta, toteutus onnistuu taulukon avulla
  - talletetaan jonon alkiot taulukkoon $Q[1, \ldots, n]$
  - jonossa olevien alkioiden maksimimäärä on $n - 1$
  - attribuutti *head* kertoo jonossa ensimmäisenä olevan olevan alkion paikan taulukossa
  - attribuutti *tail* kertoo seuraavaksi jonoon laitettavan alkion paikan
  - alussa $tail[Q] = tail[Q] = 1$ ja jono tyhjä
- seuraavalla sivulla esimerkki jonon toiminnasta sekä operaatioiden toteutus

- huom:
  - miksi jonon maksimikoko on $n - 1$?
  - mikä on operaatioiden aikavaativuus?



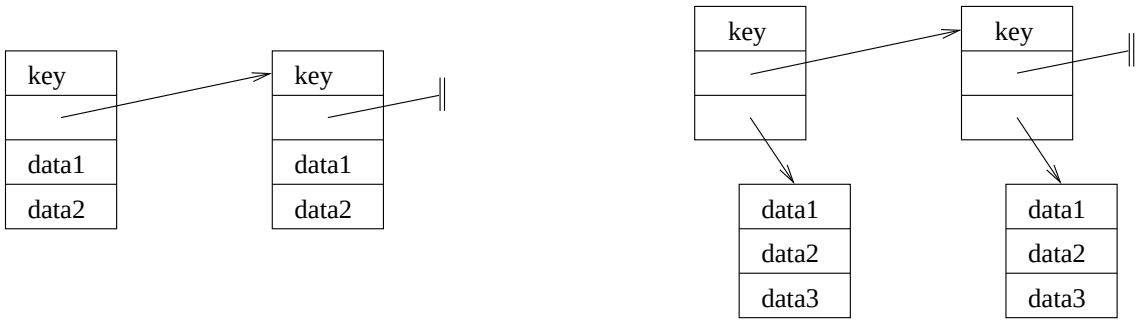

```
empty(Q)
    if head[Q] = tail[Q] then return true else return false

enqueue(Q,k)
    Q[tail[Q]] ←k
    if tail[Q]=n
        then tail[Q] ←1
        else tail[Q] ←tail[Q]+1

dequeue(Q)
    x ←Q[head[Q]]
    if head[Q]=n
        then head[Q] ←1
        else head[Q] ←head[Q]+1
    return x
```

2.3 Linkitetyt rakenteet

- Jos emme tiedä mikä on pinoon tai jonoon talletettavien alkioiden maksimimäärä, ei taulukkototeutusta voida käyttää
- ratkaisu saadaan lisäämällä tietoalkiot tallentaviin tietueisiin kenttä joka sisältää *viitteen* johonkin toiseen tietoalkioon
- esim. tietue joka sisältää avaimen, viitteen toiseen datatietueeseen sekä muuta dataa ja tietue joka sisältää avaimen, viiteen toiseen datatietueeseen sekä viitteen "satelliittidataan"



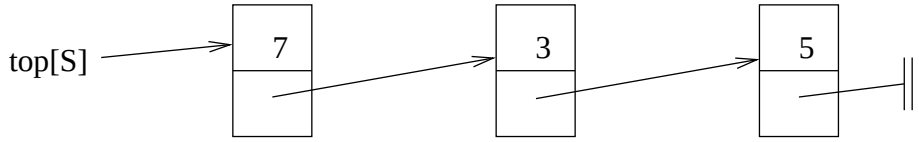
- huom: NIL on viite mikä ei viittaa mihinkään, kuvassa ⊩

- pino voidaan toteuttaa linkitettynä rakenteena seuraavasti:
  - pinossa oleva tieto talletetaan *pinosolmu*-tyyppiä oleviin tietueisiin, joilla kentät:

key	pinoon talletettu tietoalkio
next	viite toiseen pinosolmu-tietueeseen

  - Pinolla S attribuutti $top[S]$ joka on viite *pinosolmu*-tyyppiä olevaan tietueeseen
  - Jos $top[S] = NIL$ on pino tyhjä

- esim: pino missä huipulla 7, jonka alla 3 ja 5



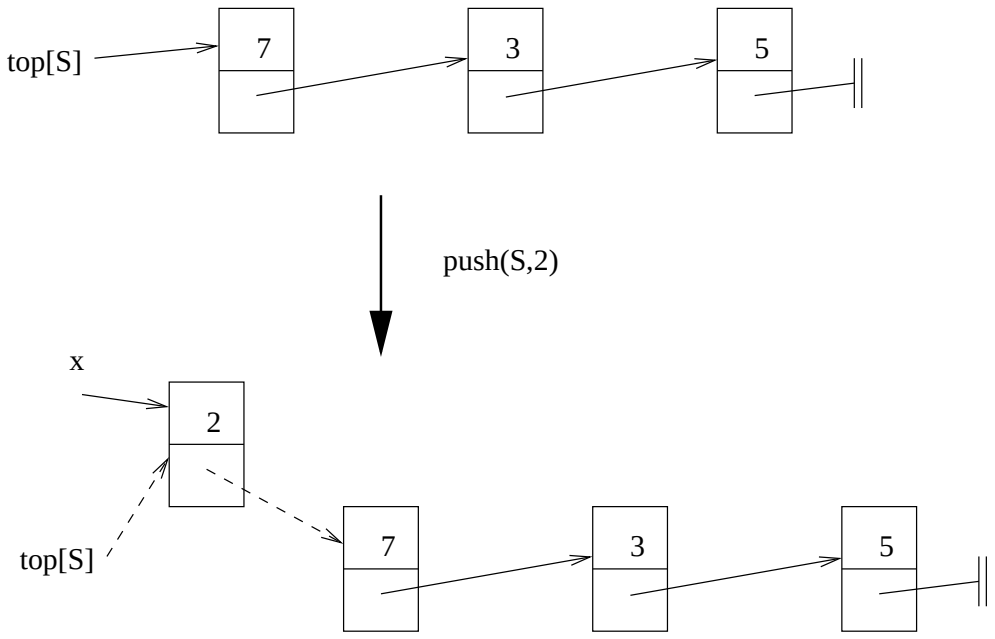
- operaatioiden toteutus ja toiminta

Empty(S)

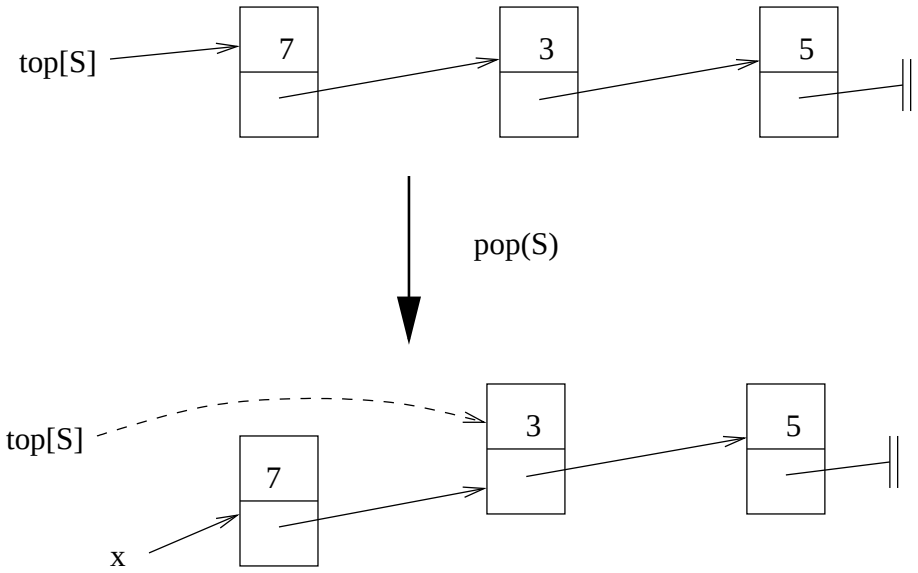
if top[S]=NIL **then return** true **else return** false

Push(S,k)

x ←**new** pinosolmu
key[x] ←k
next[x] ←top[S]
top[S] ←x



Pop(S)
    x ←top[S]
    top[S] ←next[x]
    return key[x]



- huom: popin jälkeen x:n osoittama pinon vanha huippu jää "orvoksi", esim. Javassa roskankeruumekanismi huolehtisi sen varaaman muistin vapauttamisesta
- esim c-kielessä ohjelmoijan on vapautettava itse x:n osoittama muistialue

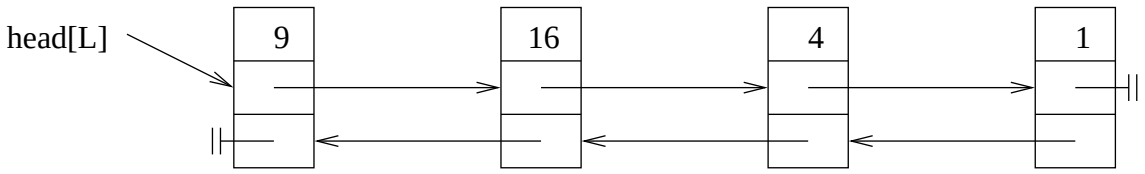
- selvästikin pino-operaatiot ovat jälleen vaativuudeltaan $\mathcal{O}(1)$
- myös jono voidaan toteuttaa linkitettynä rakenteena siten että jono-operaatiot vaativat vakioajan. Miten?

2.4 Lista

- linkitetyn listan avulla voimme helposti toteuttaa sivulla 22 esitellyn abstraktin tietotyypin joukko
- linkitetyistä listoista on monia eri variaatioita joista ensin tarkastelemme *kahteen suuntaan linkitettyä listaa*
- listalla oleva tieto talletetaan *listasolmu*-tyyppiä oleviin tietueisiin, joilla kentät:

key	talletettu tietoalkio
next	viite seuraavana olevaan listasolmu-tietueeseen
prev	viite edellisenä olevaan listasolmu-tietueeseen

- Listalla L attribuutti $head[L]$ joka on viite *listasolmu*-tyyppiä olevaan tietueeseen
- Jos $head[L] = NIL$ on lista tyhjä
- esim: lista missä missä luvut 9, 16, 4 ja 1



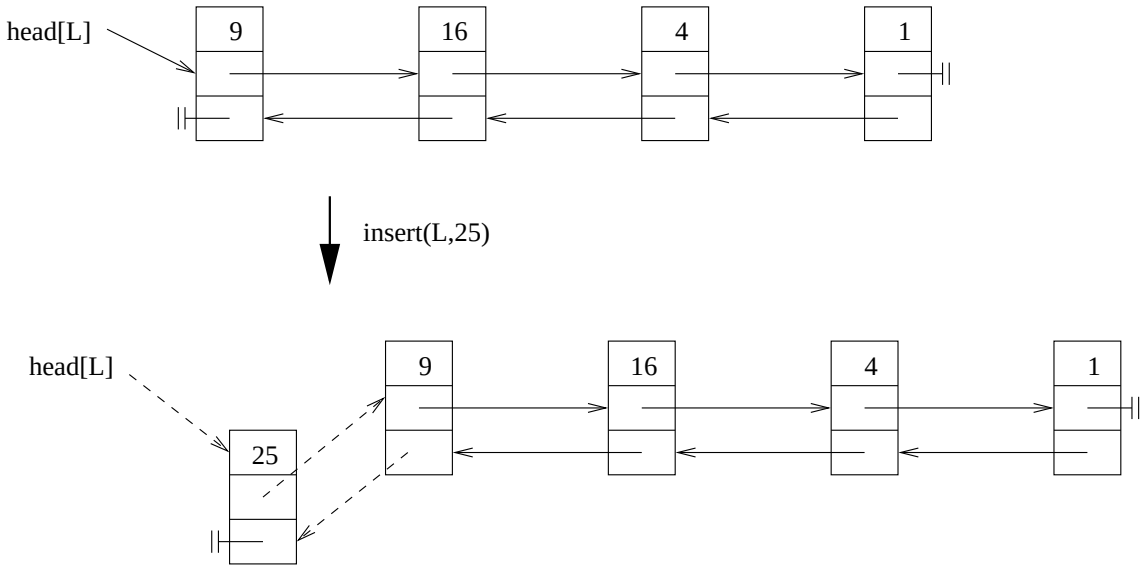
- alkioiden etsintä listalta

```
search(L,k)
    x ←head[L]
    while x ≠ NIL and k ≠ key[x] do
        x ←next[x]
    return x
```

- search-operaatiossa verrataan listan alkioita alusta lähtien etsittävään, jos etsittävää ei löydy, käydään kaikki listan alkiot läpi
- pahimmassa tapauksessa operaation aikavaativuus $\mathcal{O}(n)$, missä n listalla olevien alkioiden määrä

- alkioiden lisääminen

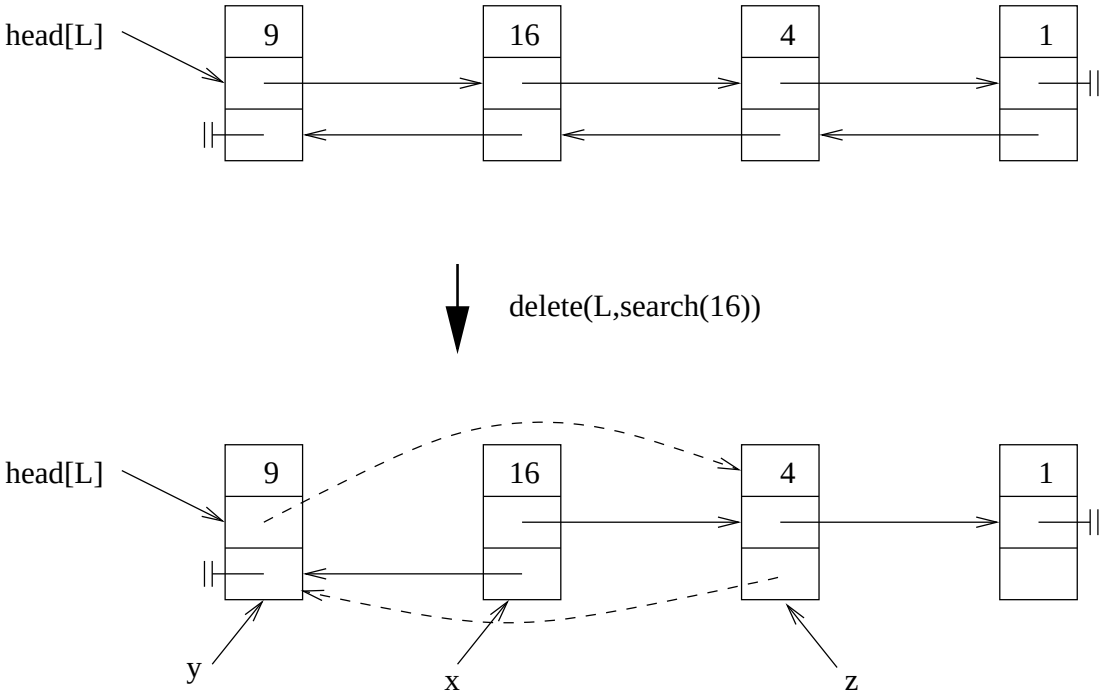
insert(L,k)
    x ←**new** jonosolmu
    key[x] ←k
    next[x] ←head[L]
    prev[x] ←NIL
    if head[L] ≠ NIL **then** prev[head[L]] = x
    head[L] ←x



- riippumatta jonossa olevien alkioiden määrästä, vain muutama viitteen manipulaatio, eli operaation vaativuus $\mathcal{O}(1)$

- alkioiden poistaminen

```
delete(L,x)
    y ←prev[x]
    z ←next[x]
    if y ≠ NIL
        then next[y] ←z
        else head[L] ←z
    if z ≠ NIL
        then prev[z] ←y
```



- vakiomäärä viitteiden manipulaatioita, eli vaativuus $\mathcal{O}(1)$
- huom: delete-operaatio saa parametrinaan viitteen poistettavaan solmuun eli jos halutaan poistaa listalta esim. luku 4 on viite poistettavaan solmuun selvitettävä search-operaatiolla: delete(L,search(4))

- entä loput sivun 22 operaatioista, min, max, succ ja pred?
- min voidaan toteuttaa seuraavasti:

```
min(L)
    x ←head[L]
    p ←head[L]
    while p ≠ NIL
        if key[p]<key[x] then x ←p
        p ←next[p]
    return x
```

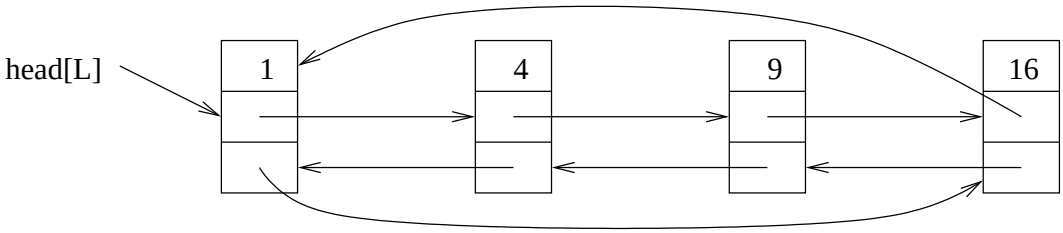
- operaatio käy läpi listan kaikki alkiot, eli vaatii ajan $\mathcal{O}(n)$
- operaatio succ:

```
succ(L,x)
    y ←NIL
    p ←head[L]
    while p ≠ NIL
        if key[p]>key[x] then
            if y = NIL or key[p]<key[y] then y ←p
        p ←next[p]
    return y
```

- operaatio käy koko listan läpi, eli aikavaativuus $\mathcal{O}(n)$
- max ja pred toteutetaan samaan tyyliin kuin min ja succ
- toteutuksessamme operaatioiden vaativuudet ovat itse asiassa samat kuin sivun 23 taulukon lista1:llä

Rengaslista

- jos talletamme luvut listalle *suuruusjärjestyksessä*, insertin vaativuus huononee luokkaan $\mathcal{O}(n)$ mutta operaatiot min, succ sekä pienillä modifikaatioilla myös operaatiot max ja pred saadaan vakioaikaisiksi, eli päädytään sivun 23 taulukon lista2:n
- insert siis ei lisää uusia alkioita aina listan ensimmäiseksi alkioksi vaan etsii niille oikean paikan siten että lisäyksen jälkeenkin listan alkiot ovat suuruusjärjestyksessä
- yksi tapa operaatioiden max ja pred saamiseksi vakioaikaiseksi on käyttää *rengaslistaa* missä alkiot on talletettu suuruusjärjestykseen



- jonon ensimmäisen alkion prev osoittaa jonon viimeiseen alkioon, jonka next osoittaa jonon ensimmäiseen alkioon

- operaatioiden min, max, succ ja pred toteuttaminen järjestetylle rengaslistalle on triviaalia:

```
min(L)
    return head[L]

max(L)
    if head[L] = NIL then return NIL
        else return prev[head[L]]

succ(L,x)
    if next[x] = head[L] then return NIL
        else return next[x]

pred(L,x)
    if prev[x] = prev[head[L]] then return NIL
        else return prev[x]
```

- edelliset neljä operaatiota ovat selvästi vakioaikaisia, olipa lista miten pitkä tahansa, aina suoritetaan sama määrä käskyjä
- listan alkioiden tulostus suuruusjärjestyksessä voidaan nyt toteuttaa seuraavasti

```
x ←min[L]
while x ≠ NIL
    print(key[x])
    x ←succ(L,x)
```

- n alkiota sisältävän listan läpikäynti onnistuu siis ajassa $O(n)$

- alkioiden poistaminen järjestetyltä rengaslistalta:

```
delete(L,x)
    y ←prev[x]
    z ←next[x]
    if x = z
        then
            ▷poistettava on listan ainoa alkio
            head[L] ←NIL
        else
            next[y] ←z
            prev[z] ←y
    if x = head[L] then head[L] ←z
```

- alkioiden etsintä järjestetyltä rengaslistalta listalta:

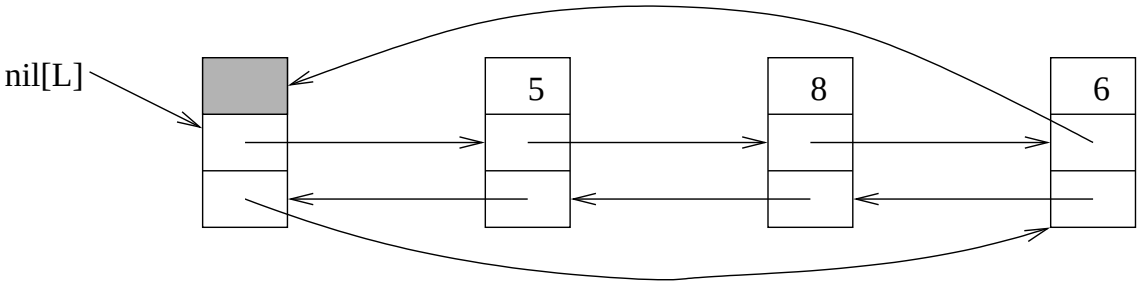
```
search(L,k)
    if head[L] = NIL return NIL
    x ←head[L]
    while next[x] ≠ head[x] and k > key[x] do
        x ←next[x]
    if key[x]= k then return x else NIL
```

- voimme lopettaa etsinnän kun listalla tulee vastaan alkio joka on suurempi kuin etsittävä avain, etsinnän pahimman tapauksen aikavaativuus on kuitenkin $O(n)$
- järjestetyn rengaslistan operaatioista hankalin on alkioiden lisääminen, koska alka on vietävä oikealle paikalle, kuluu aikaa pahimmassa tapauksessa $O(n)$
- operaatio on hiukan hankala myös monien erikseen huomioitavien tapaustensa takia

```
insert(L,k)
    x ←new jonosolmu
    key[x] ←k
    p ←head[L]
    if p = NIL then                ▷lisäys tyhjään listaan
        next[x] ←x
        prev[x] ←x
        head[L] ←x
        return
    if k<key[p] then               ▷lisäys alkuun
        next[x] ←p
        prev[x] ←prev[p]
        prev[p] ←x
        next[prev[x]] ←x
        head[L] ←x
        return
    while key[p] < k and next[p] ≠ head[L] do
        p ←next[p]
    if k>key[p] then               ▷lisäys loppuun
        next[x] ←head[L]
        prev[x] ←p
        next[p] ←x
        prev[head[L]] ←x
    else                           ▷lisäys väliin
        next[x] ←p
        prev[x] ←prev[p]
        next[prev[p]] ←x
        prev[p] ←x
```

Tunnussolmullinen lista

- joitakin operaatiota mutkistaa hieman erikoistapauksen *onko poistettava alkio listan alussa* huomioiminen
- yksi variaatio linkitetyistä listoista on *tunnussolmullinen rengaslista*
  - nyt listan alussa tunnussolmu, eli solmu missä ei säilytetä tietoa
  - attribuutti nil[L] osoittaa listan tunnussolmuun
  - listan ensimmäinen alkio löytyy viitteen next[nil[L]] päästä
  - listan viimeinen alkio löytyy viitteen prev[nil[L]] päästä
  - tyhjällä listalla next[nil[L]] = prev[nil[L]] = nil[L]
- käsitellään seuraavassa tunnussolmullisen listan variaatiota joka ei edellytä alkioille suuruusjärjestystä
- esim: tunnussolmullinen rengaslista missä luvut 5, 8 ja 6



- alkion poisto tunnussolmullisesta rengaslistalta hoituu erittäin helposti

```
delete(L,x)
    next[prev[x]] ←next[x]
    prev[next[x]] ←prev[x]
```

- alkion etsintä tunnussolmullisesta rengaslistasta

```
search(L,k)
    x ←next[nil[L]]
    while x ≠ nil[L] and k ≠ key[x] do
        x ←next[x]
    if x = nil[L]
        then return NIL
        else return x
```

- alkion lisääminen

```
insert(L,k)
    x ←new jonosolmu
    key[x] ←k
    next[x] ←next[nil[L]]
    prev[x] ←nil[L]
    prev[next[x]] ←x
    next[nil[L]] ←x
```

- listoista on muitakin variaatioita: tunnussolmuton rengaslista, yhteensuuntaan linkitetyt listat sekä tunnussolmulla että ilman, tavallisena tai rengaslistana ...

- huom: määrittelemämme lista-tietorakenne ei tarkalleen ottaen toteuta joukkoa niin kuin se käsitetään matemaattisesti, miksi?

2.4 Pinon toteutus Javalla

- vaikka kurssi on kieliriippumaton, toteutamme esimerkkinä Javalla abstraktin tietotyypin Pino
- operaatiot:

  public void push(int k)
  asettaa pinon hupulle luvun k
  public int pop()
  palauttaa ja poistaa pinosta päällimmäisen alkion
  public boolean empty()
  palauttaa true jos pino tyhjä, muuten false

- Pinon solmut toteutetaan Pino-luokan yksityisen sisäluokan PinoSolmu ilmentyminä ja pinon päällimmäiseen alkioon osoittaa yksityinen PinoSolmu-tyyppinen kenttä top:

```
public class Pino {
    private class PinoSolmu {
        int key;
        PinoSolmu next;

        private PinoSolmu(int k, PinoSolmu seur) {
            key = k; next = seur;
        }
    }

    private PinoSolmu top;

    Pino() { top = null; }
```

- huom: yksityisen sisäluokan PinoSolmu ilmentymät ovat käytettävissä ja viitattavissa vain luokan Pino sisällä
- Näin saamme aikaan oikean abstraktin tietotyypin sillä pinon sisältöä ei pääse katsomaan tai muuttelemaan suoran, vaan ainoastaan operaatioita push, pop ja empty käyttämällä
- operaatioiden toteutus

```
public void push(int k) {
    PinoSolmu uusi = new PinoSolmu(k,top);
    top = uusi;
}

public int pop() {
    PinoSolmu pois = top;
    top = pois.next;
    return pois.key;
}

public boolean empty() { return ( top == null ); }
```

- pop-operaation yhteydessä entinen pinon huippu-solmu jää ilman viitettä, ja Javan roskankeräysmekanismi tulee aikanaan poistamaan vanhalle huipulle varatun muistialueen
- Pinon käyttö sovelluksissa:

```
Pino luvut = New Pino() ;
luvut.push(10);
luvut.push(15);
System.out.println("pinon huipulla: "+ luvut.pop());
```

2.5 Esimerkki pinoa käyttävästä algoritmista

- Ohjelmoinnissa tulee helposti virheitä sulkumerkkien kirjoittamisessa:
  - sulkuja on joko liikaa tai liian vähän
  - jollekin alkusululle ei löydy loppusulkua tai päinvastoin
  - sulkeet voivat "mennä ristiin".

- Sulkujen tasapaino syötteenä olevasta merkkijonosta voidaan testata esim. seuraavasti:
  - luetaan syötemerkkijonon jokainen merkki, yksi kerrallaan, alusta alkaen
  - viedään jokainen vasen sulkumerkki pinoon
  - kun luetaan jokin oikea sulkumerkki, sitä vastaavan vasemman sulkumerkin pitää löytyä pinon päältä, se poistetaan
  - kun tiedosto on luettu, pinon pitää olla tyhjä

- Oletetaan että $readnext(stream)$ on funktio mikä poistaa ja palauttaa kirjaimen merkkijonon $stream$ alusta

- Pinoa S käyttävä algoritmi sulkujen tasapainon tarkastamiseksi

```
while ( stream not empty ) do
    c1 ←readnext(stream)
    if c1 = '(' or '[' or '{' then push(S,c1)
    if c1 = ')' then
        c2 ←pop(S)
        if c2 ≠ '(' then ERROR
    if c1 = '}' then
        c2 ←pop(S)
        if c2 ≠ '{' then ERROR
    if c1 = ']' then
        c2 ←pop(S)
        if c2 ≠ '[' then ERROR
if not empty(S) then ERROR
```