Andmestruktuurid

Programmid töötlevad andmeid. Andemeid hoitakse programmi töö jooksul mälus. Andmed pole amorfne arvude ja stringide hulk, vaid neid seovad olulised struktuursed seosed, mis võivad omada lisaväärtust. Esialgu vaatleme lineaarseid andmestikke, edaspidi jõuame keerulisema struktuuriga mittelineaarsete andmestikeni.

Informatsiooni saamiseks andmeelementide omavaheliste seoste kohta proovi vastata järgmistele küsimustele:

Milline element on nimekirjas esimene ja milline viimane? Ja miks? Millised elemendid eelnevad või järgnevad antud elemendile? Ja miks? Mitut elementi nimekirjas hoitakse? Nendele küsimustele vastates saame aru andmete loogikast, tähendusest ja seoste tähendusest.

Lihtsaim (ja üks vanim) viis andmete programmi töö ajal mälus hoidmiseks on massiiv. See on nn füüsiline struktuur, kus andmed salvestatakse järjest mälupesadesse.

Eelnevast lähtudes on oluline on vahet teha andmestruktuuri kirjelduse kahel aspektil:

- struktuuri loogilisel kirjeldusel (vaatel);
- struktuuri realisatsiooni kirjeldusel (vaatel).

Loogiline vaade kirjeldab andmestiku loogilist ülesehitust ja selle esitamiseks sobivad hästi graafilised võtted (nooled, ristkülikud, ...). Operatsioonide-funktsioonide selgitamiseks aga pseudokood või muud üldised algoritmi kirjeldamise vahendid. See on kirjeldus struktuuri käitumisest ja sellest, kuidas me teda tajume.

Realisatsioonivaade näitab, kuidas mingi loogiline struktuur arvutis üles ehitatakse ja kuidas tegelikult toimuvad tema peal vajalikud operatsioonid. Sama loogilist andmestruktuuri saab realiseerida mitmel erineval moel ning sõltub programmeerimiskeelest ja olukorrast, milline variant on otstarbekam.

Arvutiteaduse ajaloo jooksul on välja kujunenud teatud **andmestruktuurid**, mis on osutunud sobilikeks erinevate ülesannete lahendamisel. Nende kasutamisega kaasnevad kindlad mängureeglid: milline on struktuuri ülesehitus ning mida ja kuidas konkreetse andmestruktuuri ja tema elementidega teha saab ja tohib. Mõistes olulisemate andmestruktuuride ideed, saab programmeerija ise, lähtudes töödeldavate andmete ja ülesande spetsiifikast, kasutada tüüpilisi struktuure või luua uusi.

Üldistatult võib öelda, et andmestruktuure saab realiseerida nii **staatiliselt** kui **dünaamiliselt** ja kummalgi võimalusel on omad plussid ja miinused, mida juba konkreetsest struktuurist rääkides kirjeldada saab. Ka siin on suur osatähtsus kasutataval programmeerimiskeelel. Lisaks on nii mõneski programmeerimiskeeles edaspidi kirjeldatavad struktuurid realiseeritud. Alati ei ole peale vaadates võimalik üheselt tuvastada, milline realisatsiooni põhimõte kasutusel on.

Enamasti räägime nüüd ja edaspidi kursuse käigus andmestruktuuri dünaamilisest ülesehitusest, sest see annab tavaliselt loomulikuma pildi ning rikkalikumalt võimalusi (ning võimaluse end viitadega kurssi viia). Kus aga staatilisest realisatsioonist kasu võib olla, saab ka viimasest juttu tehtud.

Loend

Loend, ka järjend, nimistu *(list)* on lõplik järgnevus, mis sisaldab 0 või enam elementi. Loendi elementide vahel on järgnevussuhe. Võib rääkida esimesest ja viimasest, eelmisest ja järgmisest elemendist.

Omadused:

- Ideaaljuhul on loendi **elementide arv tõkestamata**. Tegelikult tuleb ette piir, mis on seotud arvuti mälu mahuga.
- Kõik loendi elemendid on **ühesuguse struktuuriga**.

On väljakujunenud peamised **toimingud** ehk **operatsioonid** loenditega:

- Uue elemendi **lisamine** loendi algusesse, keskele ja lõppu.
- Elemendi **kustutamine** loendi algusest, keskelt ja lõpust.
- Loendi elementide külastamine ükshaaval ehk loendi **läbimine**.
- Kahe loendi **ühendamine**.
- Ühe loendi **poolitamine**
- Loendi **pööramine tagurpidi**
- Loendist **koopia** tegemine

- Loendielementide **arvu leidmine**
- Loendis **otsimine** ja loendi **sorteerimine**.

Tavaliselt kõiki nimetatud toiminguid korraga vaja ei lähe. Seetõttu on rohkem kasutatavad piiratud omadustega ja operatsioonidega ning kindlate nimedega loendi erijuhud, nagu näiteks pinu ja järjekord. Nendest tuleb juttu edaspidi.

Loend võib olla realiseeritud **lineaarloendina** või **ahelloendina** (viitadega struktuuri).

Lineaarloend *(linear list)* on lineaarselt järjestatud andmeelementide kogum, mille elementide järjestus on mälus järjestikpaigutusega säilitatud. (EVS-ISO 2382-4:1999)

Ahelloend *(linked list)* on loend, mille andmeelemendid võivad olla mälus hajutatud, kuid iga andmeelement sisaldab informatsiooni järgmise elemendi asukoha kohta. (EVS-ISO 2382-4:1999)

Teatud juhtudel on ahelloendit parem kasutada kui lineaarloendit (enamasti on see massiiv *(array)*). Kuna ahelloend ondünaamiliselt loodav ja muudetav andmestruktuur, siis on tema eelisteks staatilise lineaarloendi ees on:

- võimalus paremini lahendada mäluprobleeme;
- luua varieeruva pikkuse ja elementide arvuga struktuure;
- loomulikumalt ja lihtsamalt realiseerida mõningaid algoritme (lisamine ja kustutamine loend keskel);
- dünaamiliselt saab luua huvitavama struktuuriga ja reaalsusele paremini vastavaid andmestikke.

Kirjeldatud realisatsiooniprobleemid jäävad nö madalale tasemele ja reaalselt mõnes kaasaegses keeles programmi kirjutades ei pruugi enam aru saada, milline tegelikult konkreetse andmetüübi realisatsioon on. Ka ei pruugi realisatsioon puhtalt staatiline või dünaamiline olla. Pigem kasutatakse mingeid hübriidvorme.

Ahelloend ehk viitadega loend

Ahelloend *(linked list)* on loend, mille andmeelemendid võivad olla mälus hajutatud, kuid iga andmeelement sisaldab informatsiooni järgmise elemendi asukoha kohta. (EVS-ISO 2382-4:1999)

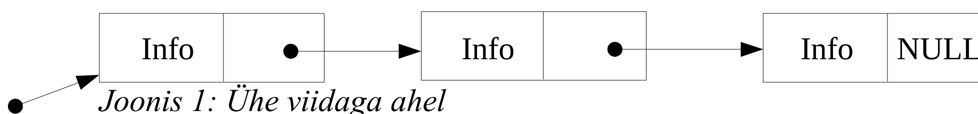
Ahelloendis luuakse elementidevahelised seosed viitasid ehk mäluaadresse kasutades. Selleks on igas ahelloendi sõlmes viidaväli, milles on eelmise ja/või järgmise sõlme aadress.

Ahelloendi sõlm (ka element, kirje) koosneb kahte tüüpi **väljadest** *(field)* - infoväljadest ja viidaväljadest. Ehkki infovälju on reeglina mitu, ei huvita nad meid eriti üldiste algoritmide koostamisel. Piisab ühest, nn **võtmeväljast** *(key)*, milles olevat väärtust algoritmis kasutada saab. Suurem huvi info vastu tekib alles siis, kui struktuuri konkreetse probleemi lahendamiseks kasutama tuleb hakata.

Struktuurses mõttes pakub rohkem huvi **viidaväli**, kus paikneb teise, naabersõlme **mäluaadress** ehk link *(link, pointer, reference)*. Kui aga naabersõlme pole, on viidavälja kirjutatud tühi aadress või `NULL`-aadress, mida erinevates programmeerimiskeeltes erinevalt tähistatakse. (Järgnevalt on kasutatud tähiseid `Nil` või `NULL`.)

Ahelloendit ja seoseid tema elementide vahel on kõige parem ette kujutada ristkülikutena (ahelloendi elemendid), mida ühendavad nooled (nool näitab, millise elemendi aadress on elemendi viidaväljas – seal, kuhu on joonistatud noole teises otsas olev mummuke) (vt Joonis 1).

Ahelloendil on **pea** *(head*). Pead võib tõlgendada mitmeti – see on kas ahelloendi esimene element, samuti viit esimesele elemendile ehk esimese elemendi aadress. **Saba** *(tail)*, milleks on loendi viimane element või viimase elemendi aadress.



Joonis 1: Ühe viidaga ahel

Pseudokeel algoritmide üleskirjutamiseks

Ahelloendite ja teiste struktuuridega seotud algoritme tuleb mingil viisil üleskirjutada. Lepime kokku algoritmide üleskirjutamiseks järgnevas **tähistuses** (kasutatavates nimedes) ja **pseudokeeles**. Kirjanduses võib leida erinevaid stiile, käeolev on sarnane MIT õpikus *Introduction to Algorithms* kasutatava keelega.

`Head` – viit ahelloendi esimesele elemendile ehk esimese elemendi aadress

`Tail` – viit ahelloendi viimasele elemendile ehk viimase elemendi aadress

`Node` – viit suvalisele ahelloendi elemendile (aadress)

`Node.Next` – viidavälja väärtus (järgmisele elemendi aadress). `Next` on väli sõlme koosseisus, sõlme aadressiks on `Node`)

Veel väljade nimesid: `Prior` – eelmise elemendi aadress, `Info, Key` – infoväli, võtmeväli

`New(Node)` – uue elemendi tegemine (mälu eraldamine), aadress kirjutatakse `Node'`i sisse

`Delete(Node)` – aadressil `Node` asuva elemendi kustutamine (mälu vabastamine)

`Nil (NIL)` või `NULL` – tühja viidavälja väärtus ehk null-aadress

`if tingimus ... else ...` - valikulause üldkuju, ploki ulatuse määrab taane

`while tingimus ....` – tsüklilause üldkuju, ploki ulatuse määrab taane

`=` - omistusmärk

Näiteks: `x = Node.Next` - `x` saab väärtuse aadressil `Node` oleva elemendi `Next` väljast

`x = Head` – et `x` saab väärtuseks ahelloendi esimese elemendi aadressi.

Ühe viidaga ahelloend

Ühe viidaga ahelloend *(singly-linked list)* koosneb viidast `Head`, mis sisaldab ahelloendi esimese elemendi aadressi ja omavahel seotud ahelloendi elementidest. Igas ahelloendi elemendis on viidaväli, milles on järgmise elemendi aadress. Viimase elemendi viidaväljas on tühi aadress `(NIL)`, mille järgi on võimalik arusaada loendi lõppemisest. Tühja ahelloendi `Head`-viida väärtuseks on samuti `NIL`. Kõigi operatsioonide puhul tuleb jälgida, et esimese elemendi aadress kaotsi ei läheks. Teiseks peab arvestama, et ükskõik mitmenda ahelloendi elemendini jõudmiseks tuleb läbida kõik temast ettepoole jäävad elemendid.

Visuaalselt saab ahelloendit ettekujutada ristkülikutena, mis omavahel nooltega ühendatud (vt Joonis 1). Järgnevalt mõned näited operatsioonidest ühe viidaga ahelloendis.

Ahelloendi läbimine

Ülesanne: vaata järjest läbi ühe viidaga ahelloendis olevad elemendid alustades esimesest ja lõpetades viimasega.

Ülesande täitmisel tuleb lähtuda ahelloendi esimese elemendi aadressist (`Head`). Edasi liigutakse ühe sõlme haaval elemendilt elemendile kuni jõutakse lõppu. Vastavalt konkreetsemale eesmärgile saab iga sõlmega midagi teha (uurida võtmevälja väärtust vms). Abimuutuja `current` on samuti viit, mille sisse salvestatakse järjest elementide aadresse. Ta tuleb kasutusele võtta selleks, et esimese elemendi aadress kaotsi ei läheks (vt kood 1).

```
current = Head
while current <> Nil
        // Tegevus jooksva elemendiga; liigume edasi mööda loendit
        current = current.next
```

Kood nr 1. Ahelloendi läbimine.

Tegevuse lõppedes on `Head'i` väärtuseks ikka esimese elemendi aadress ja `current'i` väärtuseks on `NIL`.

Sõlme lisamine

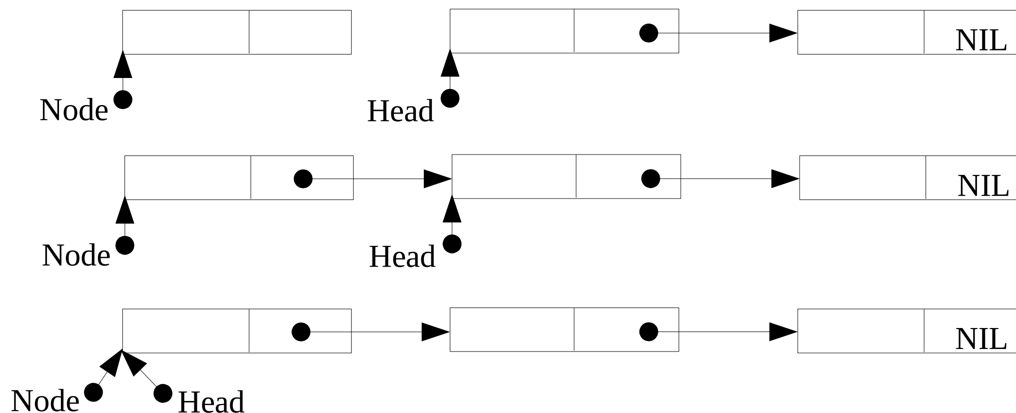
Sõlme saab lisada ahelloendi algusesse, keskele või lõppu. Kuna need tegevused on algoritmiliselt erinevad, tuleb iga olukorda käsitleda eraldi. Esimese elemendi lisamisel muutub `Head` väärtus; viimase elemendi lisamisel tuleb uue sõlme viidavälja `NIL` kirjutada; elemendi lisamisel keskele aga jälgida, et ahelloend ei katkeks.

Ülesanne: lisa ühe viidaga ahelloendi algusesse uus element.

Sõlme lisamisel algusesse on vaja ennekõike hoolitseda selle eest, et algusaadress `Head` kaotsi ei läheks. Aga tehniliselt on see tegevus üpris lihtne. Elemente järjest algusesse lisades võib üles ehitada ka terve ahelloendi. Seda loomulikult juhul, kui tekkiv elementide järjestus on sobiv. Nimelt on töö lõpuks elemendid ahelloendis (ja kättesaadavad) vastupidises järjekorras. Koodinäide 2 näitab vastavat algoritmi ja Joonis 2 iseloomustab tegevust visuaalselt.

```
New(Node)           // Uue sõlme tegemine – joonise 1. rida
Node.Info = X       // Info kirjutamine uude sõlme
Node.Next = Head    // Viida lisamine uue ja vana sõlme vahele – 2. rida
Head = Node         // Ahelloendi esimese (uue) elemendi aadressi salvestamine muutujasse Head – 3. rida
```

Kood nr 2. Elemendi lisamine ahelloendi algusese



Joonis 2: Elemendi lisamine loendi algusesse

Ülesanne: lisa ühe viidaga ahelloendisse element nii, et tekiks infovälja järgi sorteeritud loend. Infovälja väärtuse järgi otsitakse sobiv koht, kuhu element lisada.

Järgnevas näites liigutakse ahelloendis abiviidaga õigesse kohta, ehk siina, kus vastavalt infovälja väärtusele uus element lisada tuleb. Seejärel sõltuvalt kohast toimub lisamine kahel erineval viisil: algusesse lisamine ühtemoodi ja keskele ning lõppu lisamine teistmoodi. Algusesse lisamisel tuleb sarnaselt eelmisele näitele jälgida, et `Head` uue väärtuse saaks. Keskele lisamisel tuleb aga jälgida, et tekiks ühendus nii eelmise kui ka järgmise elemendiga. Õnneks sobib keskele lisamine ka lõppu lisamiseks, mille käigus `NULL`-viit oma õigele kohale viimase elemendi viidavälja satub.

Järgnevas algoritmis võetakse abiks kaks abiviita: `Current` ja `Prev`. Nende viitadega liigutakse mööda ahelloendit, kuni nad jõuavad nende elementide peale, millede vahele uus element lisatakse. St abiviitades on eelmise ja järgmise elemendi aadressid. Viit `Prev` on `Current'ist` alati ühe elemendi võrra alguse pool. Lisatava elemendi infovälja väärtus on `x`. (vt kood 3).

```
New(Node)                  // Uus sõlm tehakse valmis, Node on tema aadress
Node.Info = x
Current= Head
Prev = Nil
// Liigume õige kohani, kus Current on sõlme aadress, mille ette lisatakse
// ning Prev sõlme aadress, mille järele lisatakse
while Current <> Nil and Current.Info < X
      Prev = Current
      Current = Current.Next
// Algab lisamine, sõltuvalt kohast toimub see erinevalt
if Prev == Nil then        // Lisamine algusesse
      Node.Next = Head
      Head = Node
else                       // Lisamine keskele Current ja Prev vahele või lõppu
      Prev.Next = Node
      Node.Next = Current
```

Kood nr 3 Elemendi lisamine otsimisahelloendi algusesse, keskele või lõppu

Miks ei ole vaja eraldi algoritmi osa loendi lõppu lisamiseks, vaid sobib keskele lisamise variant?

Joonista läbi keskele ja lõppu lisamine, et algoritmist paremini aru saada.

Sõlme kustutamine aheloendist

Ülesanne: kustuta ühe viidaga ahelloendist element, mis sisaldab informatsiooni x.

Kustutav sõlm otsitakse tavaliselt üles infovälja või võtmevälja väärtuse järgi. Mugavam on kustutada kasutades kahte abiviita (sarnaselt lisamisele), ehkki hakkama saab ka ühega. Järgnevas algoritmis kasutatakse kahte abiviita. Kustutatava elemendini liikumine toimub samuti sarnaselt eelmisele näitele (vt kood 4). Esimese elemendi kustutamine erineb ülejäänud elementide kustutamisest, sest on vaja hoolitseda, et muutuja `Head` uue väärtuse saaks.

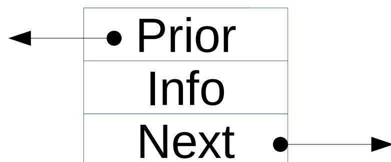
```
// Elemendi kustutamine, mille infovälja väärtus on X
Current = Head
Prev = Nil
while Current <> Nil and Current.Info <> X
        Prev = Current
        Current = Current.Next
if Cufrent.Info == X            // Leiti vastav element
      if Prev == Nil            // Kustutamine algusest
            Head = Head.Next
      else                      // Kustutamine keskelt või lõpust
            Prev.Next = Current.Next
      Delete(Current)
else
      // Element infovälja väärtusega X puudub
```

Kood nr 4 Elemendi kustutamine.

Mõnikord on hea kasutada ühe viidaga ahelloednis veel teist välist viita, mis näitaks loendi viimasele elemendile (sabale) `(Tail)`. Sel juhul muutub lihtsamaks ja kiiremaks lisamine lõppu, kahe ahelloendi ühendmine jms. Ahelloendiga võib siduda loenduri (päisele viitavas elemendis), kus peetakse arvet sõlmede arvu kohta ning sel juhul pole vaja eraldi protseduuri elementide loendamiseks. Parima variandi kasuks saab otsustada vaid konkreetsest olukorrast lähtudes.

Kahe viidaga ahelloend

Kahe viidaga ahelloendi *(doubly-linked lists)* funktsioonid on üldiselt samad, kui ühe viidaga ahelloendil. Kahe viidaga ahelloendites on sõlmed omavahel seotud kahe viidaga. Selleks on igas sõlmes kaks aadressivälja – järgmisele sõlme aadress `Next` ja eelmise sõlme aadress `Prior`.



Joonis 3: Kahe viidaga ahelloendi element

Kahe viidaga ahelloendiga on alati seotud ka kaks välist viita: `Head` (esimese sõlme aadress) ja `Tail` (viimase sõlme aadress).

Eelised: saab teha kiiremini operatsioone ahelloendi mõlema otsaga, teinekord mugavam ka keskmiste elementide töötlemiseks. Liikuda saab elementide vahel mõlemas suunas.

Puudused: võtab rohkem mälu, on keerukam hallata, sest lahti võtta ja ühendada on alati vaja poole rohkem viitasid.

Sõlme lisamine

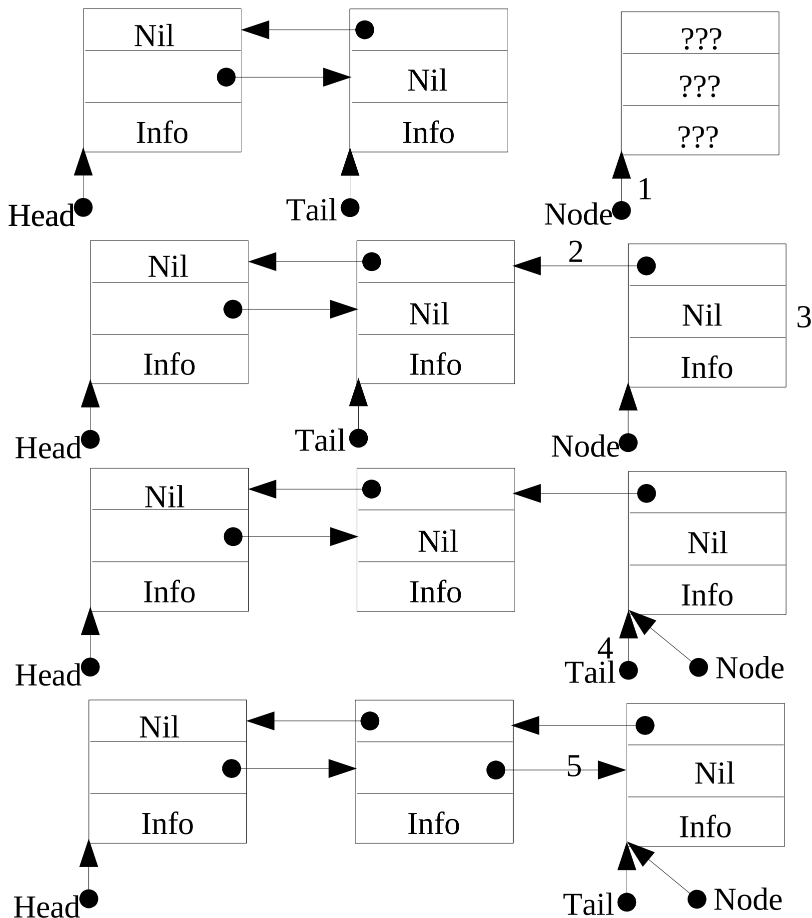
Ülesanne: lisa kahe viidaga ahelloendi lõppu uus sõlm.

Vaatame näitena sõlme lisamist ahelloendi lõppu. Selleks on olemas viimase elemendi aadress `Tail` ja tuleb jälgida, et kõik viidad lisatud saaksid (vt Joonis 4 ja kood 5). Eraldi olukorrana on välja toodud tühja ahelloendisse uue (esimese) elemendi lisamine. Lisatavasse sõlme kirjutatakse info X. Numbrid kommentaaridena koodi juures ja joonisel tähistavad samu operatsioone.

```
// Ahelloendi lõppu lisatakse uus element, mis sisaldab infot X
New(Node)          // 1
Node.Info = X
Node.Prior = Tail  // 2
Node.Next = NIL    // 3
```

```
Tail = Node         // 4
If Node.Prior == Nil
// Ahelloend oli tühi, uus sõlm saab ühtlasi peaks, st Tail ja Head viitavad samale elemendile
      Head = Node
else
      Node.Next.Prior = Node // 5 Ühendatakse viimase sõlme külge
```

Kood nr 5. Elemendi lisamine kahe viidaga ahelloendi lõppu.



Joonis 4: Elemendi lisamine kahe viidaga ahelloendi lõppu

Sõlme kustutamine

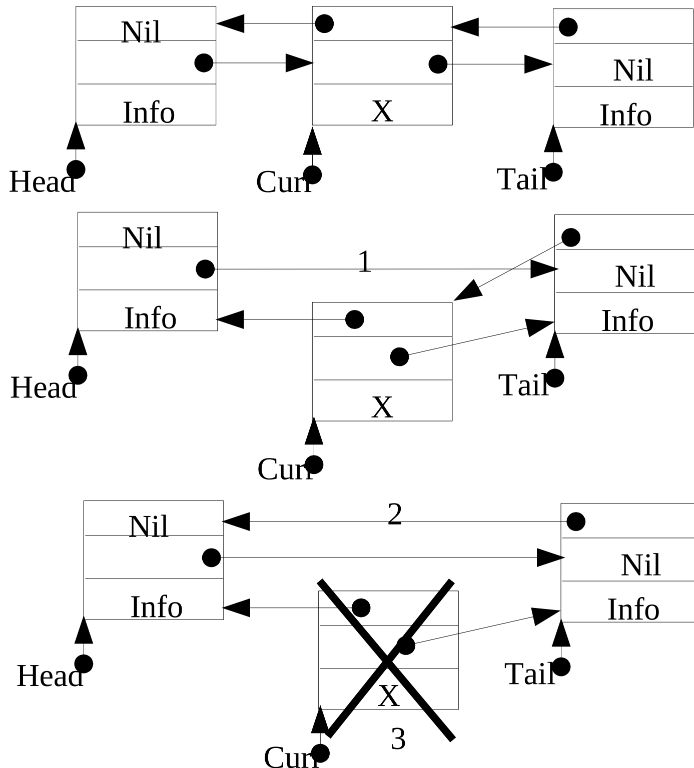
Ülesanne: kustuta kahe viidaga ahelast sõlm, mis sisaldab informatsiooni X.

Sõlme kustutamisel kahe viidaga ahelast ei ole vaja kasutada kahte abiviita, sest ahelloendi keskel on nii ees- kui tagapool olevad elemendid kättesaadavad tänu mõlemat pidi jooksvatele viitadele. Kõigepealt on aga vaja üles otsida element, mille infoväljas on informatsioon X. Vaata kood nr 6 ja Joonis 5.

```
// Ahelloendist kustutatakse sõlm infogaX
Current = Head
// Liigume kustutatava sõlmeni
while Current <> Nil and Current.Info <> X
        Current = Current.Next
// Kui leiti vastava infoga sõlm (tema aadressiks on Current),
// siis kustutame, aga enne tuleb kontrollida, kus kustutatav sõlm paikneb.
if Current.Info == X
      if Current.Prior == Nil   // Eespool midagi ei ole, seega kustutame esimese
            Head = Current.Next
```

```
      Head.Prior = Nil
   else if Current.Next == Nil // Taga midagi ei ole, seega kustutame viimase
         Tail = Current.Prior
         Tail.Next = Nil
      else                    //Kustutame keskelt
         Current.Prior.Next = Current.Next  // 1
         Current.Next.Prior = Current.Prior // 2
   Delete(Current)   // 3
else
   // Sellist elementi ei leitud
```

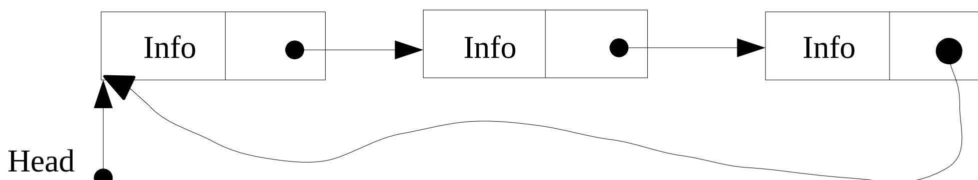
Kood nr 6. Sõlme kustutamine kahe viidaga ahelloendist.



Joonis 5: Sõlme kustutamine kahe viidaga ahelloendist

Ringahelloend

Ringahelloendis *(Circular linked list)* on ahelloendi viimane element seotud esimese elemendiga (viitab esimesele elemendile). Haldamiseks on vajalik üks viit, mis viitab suvalisele elemendile (juhul kui mingit formaalset algust või lõppu fikseerida vaja ei ole). Olenemata nimetatud viida asukohast, on alati olemas juurdepääs suvalisele ahelloendi elemendile. Ringahelloendit peetakse mõnikord ka mittelineaarseks struktuuriks. Reaalne struktuur, mille esitamiseks ringahelloendit kasutatakse, ei pea olema olemuselt tsükliline. Loendit läbides on siiski vajalik hoida üks viit paigal, et saaks tuvastada, kas kõigile elementidele on ring peale tehtud.



Joonis 6: Ringahelloend

NB! Üldised märkused

1. Kõigi ahelloendite ja kõigi algoritmide juures tuleb jälgida, et nad töötaksid ka tühja ahelloendi korral `(Head==Nil, Tail==Nil).`
2. Ei tohi unustada tühjade viitade omistamist, et ahelloendi lõpu saaks üles leida.
3. Ei tohi unustada esimese elemendi (`Head`) ega viimase elemendi (`Tail`) aadresse, sest siis läheb suur osa infost kaotsi.
4. Sõlmede keskele lisamisel tuleb jälgida, et viidad õiges järjekorras ümberomistatud saavad, et struktuur katki ei läheks.

Ahelloendid C vahenditega

Ahelloendi tegemiseks C keeles on vaja kõigepealt tuttavaks saada viidatüüpi ehk aadressitüüpi muutujatega. Selle kohta sobib pikemalt lugeda kursuse veebis olevatelt viidetelt ja vaadata näidete kaustas olevaid näiteid. Et aga järgnevat paremini (ja kiiremini) mõista, siis paar märkust siiski C-keele süntaksi ja põhiliste mõistete kohta.

Viitmuutujad keeles C

Viit on tegelikult muutuja aadress. Kõik muutujad, mida tavapärasel viisil programmis kasutame, on programmi töö vältel paigutatud mingitele mäluaadressidele, milledega majandamise meelsasti arvuti hooleks jätame. Kuid mõnel juhul osutub vajalikuks neid aadresse otse ja teadlikult kasutada. Muuhulgas ka selleks, et eelnevates peatükkides kirjeldatud struktuure moodustada.

Viitmuutujat saab C-s deklareerida järgmiselt (`ptr` on viit täisarvulisele väärtusele):

```
int *ptr;
```

Tähis `ptr` on muutuja, kuid ei sisalda mitte täisarvu ennast vaid tema mäluaadressi. Täisarvu enda jaoks tuleb täiendavalt mäluruum eraldada

```
ptr = malloc(sizeof *ptr);
```

Mälu antakse täpselt nii mitu baiti, kui täisarv ruumi võtab (seega nelja baidi jagu). Peale funktsiooni `malloc()` tööd on mälus täiendavalt reserveeritud neli baiti mälupesi. Viit nendele mälupesadele ehk mäluaadress on salvestatud muutujasse `ptr`. Täisarvu salvestamine eraldatud mälupesadesse toimub järgmiselt:

```
* ptr = 10;
```

Seda tegevust nimetatakse otsendamine või otsendus (*dereferencing).*

Võimalik on küsida muutuja aadressi, selleks on &-märk (kasutades ära eelnevalt deklareeritud muutujat `ptr`):

```
int number;
number = 20;
ptr = &number;
```

Tehtud lausete tulemusena on muutujas `ptr` on nüüd muutuja `number` mäluaadress ning eelnevalt omistatud 10 on jäädavalt koos oma mälupesadega kadunud. St tegelikult on ta mälus küll alles, aga me ei tea enam kus – aadress kirjutati üle.

Edasi võib küsida, et mida tähendab `&ptr`?

NB! Pane tähele, et viidad on reeglina tüpiseeritud ehk kasutada saab viita täiarvule, viita ujukomaarvule, viita struktuurile (kirjele), massiivile jne. Ehkki aadressid on alati ühesugused, annab selline tüpiseerimine võimaluse harrastada viitadega aritmeetikat nagu näiteks `ptr++`. Viimane tähendab seda, uueks väärtuseks saab nelja baidi kaugusel oleva mälupesa aadress (sest `int` on neli baiti suur). Kuid veelgi olulisemana annab ta kompilaatorile mingigi võimaluse kontrollida mälukasutust, et näiteks reserveeritud nelja baiti mälupesadesse ja proovitaks omistada kaheksat baiti infot.

Ahelloendid

Ahelloendi moodustamiseks on vaja kasutada keerulisemat struktuuri kui lihtsalt üks viitmuutuja. St on vaja moodustada "kahe- või rohkemaosalised ristkülikud". Selleks sobib struktuurne andmetüüp, mille programmeerija ise kirjeldama peab - nn **struktuur** (keelesõna `struct`). Struktuur võib koosneda mitmest erinevat tüüpi väljast. Mitmetes keeltes on sellise andmetüübi vasteks **kirje** *(record)*. Väljadele antakse nimed, mida kasutatakse hiljem ka programmis, ja määratakse andmetüübid. Välja kirjeldamisel saab rekursiivselt kasutada struktuuri enda tüüpi, mida on vaja viida- ehk aadressivälja deklareerimiseks. Viimast silmas pidades kirjeldame ahela elemendi, mis sisaldab infovälja ja viidavälja:

```
struct element {
          int key;
          struct node *next;
   };
```

`element` – uus andmetüüp, mitte aga veel muutuja!

`key` – täisarvuline väli võtme / väärtuste jaoks

`next` – tüpiseeritud viit järgmisele elemendile, mis on selle sama struktuuri aadressi tüüpi.

Eelnevalt kirjeldatud `struct element` andmetüüpi kasutades deklareerime kaks muutujat:

```
struct element node;          // deklareeritakse terve struktuur oma väljadega
struct element *nodepointer;  // deklareeritakse viit elemendile, elemendi enda jaoks tuleb veel mälu eraldada
```

Tüüpiliselt kasutatakse kirjetes ja samuti C struktuurides väljadega töötamiseks kirjaviisi, kus välja nimi eraldatakse muutujanimest punktiga (näit: `node.key`). Seoses viitade kasutamisega muutuks aga kirjaviis üsna tülikaks:

```
(*nodepointer).key
```

Nii on olemas spetsiaalne süntaks kasutamiseks juhul, kui kogu struktuurile viitab aadress:

```
nodepointer->key
```

Antud juhul on `nodepointer` struktuurielemendi aadress ning `key` on selle struktuuri üks väli. Mõlemad kirjaviisid töötavad edukalt.

Elemendi lisamiseks ahelloendisse eraldatakse mälu `element`-jagu ning seejärel ühendatakse see ahelloendisse soovitud kohta.

Järgnevalt mõned näited ahelloendiga toimetamisest. Näited kasutavad sama struktuuri `element`, mis eespool kirjeldatud on.

Ahelloendi läbimine

Näide ahelloendist info (võtmeväljade) väljatrükkimise kohta. Esitatud eraldi funktsioonina (`printList`), mille argumendiks on ahelloendi esimese elemendi aadress):

```
printList(struct element *head){
   struct element *current;    //Deklareeritakse lokaalne muutuja ahelloendis liikumiseks
   current = head;
   while (current != NULL) {
       printf("%d\n",current->key);
       current = current->next;
   }
 }
```

Tsükkel töötab seni, kuni muutuja `current` on jõudnud ahelloendi lõppu.

Ahelloendi moodustamine

Järgnev näide loob ahelloendi, kusjuures uus element lisatakse alati ahelloendi algusesse (vt ka koodinäidet 2). Kood ei ole töötav tervik. Terviklik näiteprogramm on failis `list_loomine.c`.

```
int number;
struct element *node, *head;
head = NULL;
printf("Sisesta arv! (Lõpetamiseks 0)");
scanf("%d",number);
while (number != 0){
      node = malloc(sizeof *node);   // Uus element
      node->next = head;             // Uus element seotakse esimese elemendiga
      node->key = number;
      head = node;                   // Head saab uueks väärtuseks uue elemendi aadressi
      printf("Sisesta arv! (Lõpetamiseks 0)");
      scanf("%d",&number);
}
```

Elemendi kustutamine ahelloendist

Järgnev näide kustutab ahelloendist elemendi, mille võtmeks on x. Näide on poolik. Eeldatakse, et loend on eelnevalt moodustatud ning esimese elemendi aadressiks on `head`. Vt ka koodinäidet 4, terviklik programm on failis `kustuta_element.c`

```
printf("Mida otsime >");
scanf("%d",&number);
```

```
current = head;
prev = NULL;
while (current != NULL && current->key != number) {
    prev = current;
    current = current->next;
}
if (current != NULL) {            //Leiti vastav element
    if (prev == NULL) {           //Kustutamine algusest
        head = head->next;
    }
    else {                        //Kustutamine keskelt või lõpust
        prev->next = current->next;
    }
    free(current);               //Mälu vabastatakse
}
else {                           //Element infovälja väärtusega number puudub
     printf("Selline element puudub\n");
}
```

While-tsükli tingimust sobib sellisel viisil kirja panna vaid juhul, kui loogikaavaldiste väljaarvutamine toimub osaliselt, st kui avaldise esimese poole järgi on tulemus selge, siis teist poolt ei arvutata. Miks?