# *Andmestruktuurid*

Programmid töötlevad andmeid. Neid hoitakse programmi töö jooksul mälus. Andmed pole amorfne arvude ja stringide hulk, vaid neil on omavahel väga olulised struktuursed seosed, mis võivad omada lisaväärtust. Esialgu vaatleme andmestikke, mis on lineaarsed, edaspidi jõuame andmestikeni, kus valitsevad keerulisemad omavahelised seosed.

Informatsiooni andmeelementide omavaheliste seoste kohta saab, kui vastata järgmistele küsimustele:

Milline element on nimekirjas esimene ja milline viimane? Millised elemendid eelnevad või järgnevad antud elemendile? Mitut elementi nimekirjas hoitakse?

Lihtsaim viis (struktuur) andmete mälus hoidmiseks on massiiv(id). See on nn füüsiline struktuur. Loogiliseks struktuuriks on andmete loend – andmed on lineaarselt, nad on järjestatud millegi alusel, igale andmeelemendile eelneb ja järgneb alati üks element. On oluline, kes või mis on jadas esimene, viimane jne ehk elementide järjekord tähendab midagi.

> **Näit**. Ühemõõtmeline massiiv, kus on üliõpilaste nimekiri (loend): seoseks võib olla järjestatus tähestiku alusel, järjestus ainesse registreerumise järjekorra alusel jne.

Keerulisemad struktuurid on kahe ja enamamõõtmelised massiivid, omavahel loogiliselt seotud massiivid, aga ka näiteks hierarhilised puukujulised struktuurid jms.

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 andmestruktuuriga teha saab ja tohib. Mõistes olulisemate andmestruktuuride ideed, saab programmeerija ise, lähtudes töödeldavate andmete ja ülesande spetsiifikast, kasutada tüüpilisi struktuure või luua uusi.

Oluline on vahet teha andmestruktuuri kirjelduse kahel aspektil:

- struktuuri loogilisel tasemel;
- struktuuri realisatsiooni tasemel.

**Loogiline tase** kirjeldab struktuuri loogilist ülesehitust ja selle esitamiseks sobivad hästi mitmed graafilised võtted. Operatsioonide-funktsioonide selgitamiseks aga pseudokood või muud üldised vahendid algoritmi kirjeldamiseks. See on kirjeldus struktuuri käitumisest ja sellest, kuidas me teda tajume.

**Realisatsiooni tase** näitab, kuidas vastav struktuur tegelikult arvutis üles ehitatakse ja kuidas tegelikult toimuvad tema peal vajalikud operatsioonid. Realiseerida saab tavaliselt sama struktuuri mitmel erineval moel ning sõltub programmeerimiskeelest ja olukorrast, milline variant on otstarbekam. Samas esitavad erinevad realisatsioonid samasugust loogilist andmestruktuuri.

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.

# *Mõisted*

Andmestruktuuri elemendi kirjeldamisel kasutatakse tavaliselt järgmisi **mõisteid**:

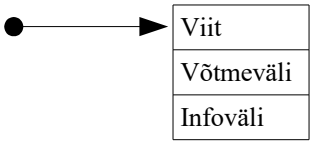
**Sõlm** *(node)* – üks andmeelement struktuuris (ka **kirje** – *record*; **objekt, element** – *entitie*)

**Väli** *(field)* – üks osa sõlmest, hõlmab mitut baiti mälus. Välju võib olla sõlmes olla mitu ja nendes hoitakse infot. Lisaks on vajalikud väljad, et luua seoseid struktuuri teiste sõlmedega.

**Võtmeväli** *(key)* - ühte või ka mitut välja vaadeldakse võtmeväljadena, mille järgi näiteks sorteerida ja otsida saab, kuid see väli ei pea olema erilises staatuses, vaid vastavalt eesmärgile saab olla võtmeväljaks kord üks ja kord teine infoväli.

Teisalt saab võtmevälja ka kirjeldada kui iga sõlme jaoks teda üheselt identifitseerivat välja, kus siis igal sõlmel on oma unikaalne väärtus.

**Sõlme aadress** *(link, pointer)* – (ka **link**, **viit sõlmele**) sõlme esimese baidi aadress arvuti mälus. Vt Joonis 1.




*Joonis 1. Sõlme kujutamiseks sobib kasutada ristkülikuid.*

Iga osa selles ristkülikus on väli, mummuga nool sümboliseerib sõlme aadressi, mumm peaks paiknema seal, kuhu aadress kirjutatud on.

Iga andmestruuriga kaasnevad **operatsioonid**, mida selle struktuuriga sobilik kasutada on.

Vastavalt oma ülesehitusele võivad andmestruktuurid olla **lineaarsed** või **mittelineaarsed**. Lineaarsed andmestruktuurid on loendd, kus elementide vahel on järgnevussuhe.

## Lineaarne loend

> **Lineaarne loend** *(linear list)*, ka järjend, 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 **tegevused** ehk **operatsioonid**, mida loendite juures vaja läheb:

- 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 tegevusi ühe 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.

Lineaarne loend võib olla **realiseeritud massiivi** või **ahela** (viitadega struktuuri) abil (staatiliselt või dünaamiliselt). Teatud juhtudel on ahelat parem kasutada kui massiivi (ehkki esialgu võib massiiv tunduda mõistetavam). Kuna ahela puhul on tegemist dünaamiliselt loodava ja muudetava andmestruktuuriga, siis on tema eelisteks staatilise massiivi ees:

- võimalus paremini lahendada mäluprobleeme;
- luua varieeruva pikkuse ja elementide arvuga struktuure;
- mõningaid algoritme (lisamine ja kustutamine loend keskel) saab palju loomulikumalt ja lihtsamalt realiseerida;
- dünaamiliselt saab tekitada palju huvitavama struktuuriga andmekogumeid, mis vastavad reaalsusele paremini.

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 see puhtalt staatiline või dünaamiline olla. Pigem kasutatakse mingeid hübriidvorme.

# Ahel ehk viitadega loend

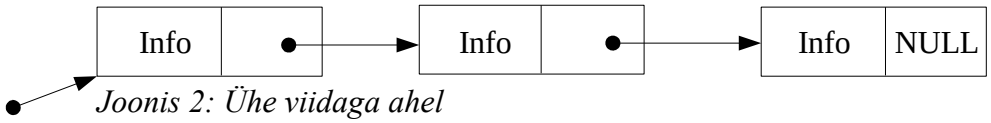
**Ahel** on lineaarne loend *(linked list)*, kus elementide vaheline järgnevussuhe luuakse viitasid ehk aadresse kasutades. Selleks on igas ahela elemendis viidaväli, milles on eelmise või järgmise elemendi aadress.

**Ahela element** (kasutatakse ka mõistet **sõlm**) koosneb kahte tüüpi väljadest - infoväljadest ja viidaväljadest. Ehkki infovälju on reeglina mitu, ei huvita nad meid üldiste algoritmide koostamisel. Piisab ühest võtmeväljast, 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 naaberelemendi aadress. Kui aga naaberelementi pole, on viidavälja kirjutatud tühi aadress või `NULL`-aadress, mida erinevates programmeerimiskeeltes erinevalt tähistatakse. (Järgnevalt on kasutatud tähist `Nil` või `NULL`.)

Ahelat ja seoseid tema elementide vahel on kõige parem ette kujutada ristkülikutena (ahela elemendid), mida ühendavad nooled (nool näitab, millise elemendi aadress on elemendi viidaväljas – seal, kuhu on joonistatud noole teises otsas olev mummuke) (vt Joonis 2).

Ahelal on **pea** *(head)*. Pead võib tõlgendada mitmeti – see on kas ahela esimene element, ka viit esimesele elemendile, ja **saba** *(tail)*, milleks on ahela viimane element või viit viimasele elemendile.




*Joonis 2: Ühe viidaga ahel*

## *Pseudokeel algoritmide üleskirjutamiseks*

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

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

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

`Node` – viit suvalisele ahela elemendile (aadress)

`Node.Next` – viidavälja väärtus (aadress), mis viitab järgmisele elemendile (`Next` on väli, mis kuulub sõlme koosseisu, sõlme aadressiks on `Node`)

`Node.Prior` – analoogiline eelmisega, kuid välja `Prior` kirjutatakse eelmise elemendi aadress

`Node.Info` – infovälja väärtus aadressil `Node` olevas sõlmes

`Node.Key` – võtmevälja väärtus aadressil `Node` olevas sõlmes

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

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

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

`x, link, ...` – tavaline muutuja, mis sõltuvalt olukorrast võib olla viidatüüpi või siis muud andmetüüpi

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

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

`for i = 1 to N` ... - for-tsükli üldkuju, `i` ja `N` on loomulikult muudetavad.

`=` - omistusmärk omistuslauses

**Näiteks**: `x = Node.Prior` tähendab, et `x` saab väärtuse aadressil `Node` paikneva elemendi `Prior` väljast (sisuliselt eelneva elemendi aadressi).

`x = Head` tähendab, et `x` saab väärtuseks ahela esimese elemendi aadressi.

## *Ühe viidaga ahel*

**Ühe viidaga ahel** *(singly-linked list)* koosneb viidast `Head`, mis sisaldab ahela esimese elemendi aadressi ja omavahel seotud ahela elementidest. Igas ahela elemendis on viidaväli, milles on järgmise elemendi aadress. Viimase elemendi viidaväljas on tühi aadress `(NIL)`, mille järgi on võimalik arusaada ahela lõppemisest. Tühja ahela `Head`-viida väärtuseks on samuti `NIL`. Kõigi ahelas tehtavate operatsioonide puhul tuleb jälgida, et ahela esimese elemendi aadress nö kaotsi ei läheks. Kui see juhtub, ei ole võimalik enam esimest elementi kätte saada ja see tähendab ühtlasi, et terve ahel on kaotatud. Teiseks peab arvestama sellega, et ükskõik mitmenda ahela elemendini jõudmiseks tuleb nö läbi käia kõik temast ettepoole jäävad elemendid.

Visuaalselt saame ahelat ettekujutada kastidena, mis omavahel nooltega ühendatud (vt Joonis 2). Järgnevalt mõned näited operatsioonidest ühe viidaga ahelas.

## Ahela läbimine

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

Ülesande täitmisel tuleb lähtuda ahela esimese elemendi aadressist (`Head`). Edasi liigutakse ühe sõlme haaval elemendilt elemendile kuni jõutakse ahela 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 ahela esimese elemendi aadress kaotsi ei läheks. Vastasel juhul kaotaksime kogu ahela (vt kood 1).

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

*Kood nr 1. Ahela läbimine*.

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

## Sõlme lisamine ahelasse

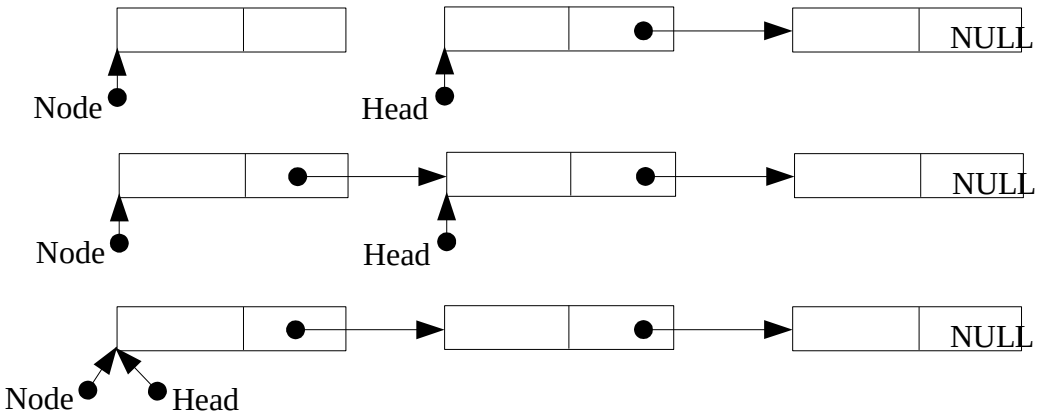
Sõlme lisamine võib toimuda ahela algusesse, keskele või lõppu. Kuna need tegevused on algoritmiliselt erinevad, tuleb iga olukorda käsitleda eraldi. Esimese elemendi lisamisel muutub ahela pea väärtus; viimase elemendi lisamisel tuleb uue sõlme viidavälja `NIL` kirjutada; elemendi lisamisel keskele aga jälgida, et ahel ei katkeks.

**Ülesanne: lisa ühe viidaga ahela algusesse uus element.**

Sõlme lisamisel ahle algusesse on vaja ennekõike hoolitseda selle eest, et kogu ahela algusaadress `Head` kaotsi ei läheks. Aga tehniliselt on see tegevus üpris lihtne. Elemente järjest algusesse lisades võib üles ehitada ka terve ahela. Seda loomulikult juhul, kui tekkiv elementide järjestus on sobiv. Nimelt on töö lõpuks elemendid ahelast (ja kättesaadavad) täpselt vastupidises järjekorras sellele kuidas neid lisati. Koodinäide 2 näitab vastavat algoritmi ja Joonis 3 iseloomustab tegevust visuaalselt.

```
// Uue elemendi lisamine ahela algusesse
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        // Ahela esimese (uue) elemendi aadressi salvestamine muutujasse Head – 3. rida
```

*Kood nr 2. Elemendi lisamine ahela algusese*




*Joonis 3: Elemendi lisamine ahela algusesse*

> **Ülesanne: lisa ühe viidaga ahelasse element nii, et tekiks infovälja järgi sorteeritud ahel. Infovälja väärtuse järgi otsitakse ahelas sobiv koht, kuhu element lisatakse. Sellist ahelat kutsutakse ka otsimisahelaks.**

Järgnevas näites liigutakse kõigepealt abiviidaga mööda ahelat 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 järgida, 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 viita: `Current` ja `Prev`. Nende viitadega liigutakse mööda ahelat (sarnaselt ahela läbimisele) seni kuni nad jõuavad nende elementide peale, millede vahele uus element lisatakse. St abiviitades on eelmise ja järgmise elemendi aadressid. Viit `Prev` on alati ühe elemendi võrra alguse pool kui `Current`. Lisatava elemendi infovälja väärtus on `x`. (vt kood 3).

```
// Elemendi lisamine ahelasse sorteeritult võtmevälja järgi.
// Kahe abiviidaga variant.
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 otsimisahela algusesse, keskele või lõppu

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

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

## Sõlme kustutamine ahelast

> **Ülesanne: kustuta ühe viidaga ahelast 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.

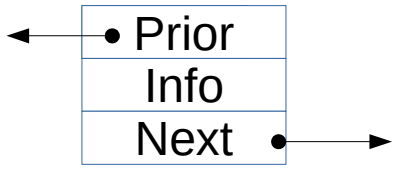
```
// Elemendi kustutamine, mille infovälja väärtus on X
Current = Head
Prev = Nil
while Current <> Nil and Current.Info <> X
        Prev = Current
        Current = Current.Next
if Cufrent.Info == X          // Leiti vastav element
      if Prev == Nil          // Kustutamine algusest
            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. Sobib kustutamiseks nii ahela algusest, keskelt kui ka lõpust

Mõnikord on hea kasutada ühe viidaga ahelaga seoses veel teist välist viita, mis näitaks ahela sabale `(Tail)` (viimasele elemendile). Sel juhul muutub lihtsamaks ja kiiremaks lisamine lõppu, kahe ahela ühendmine jms. Lisaks võib ahelaga siduda loenduri (päisele viitavas elemendis), kus pidevalt 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 ahel*

Kahe viidaga ahela *(doubly-linked lists)* funktsioonid on üldiselt samad, kui ühe viidaga ahelal. Kahe viidaga ahelate puhul on sõlmed omavahel seotud kahe viidaga. Selleks on igas sõlmes kaks aadressivälja – järgmisele sõlme aadress `Next[Node]` ja eelmise sõlme aadress `Prior[Node]`.




*Joonis 4: Kahe viidaga ahela element*

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

**Eelised**: saab teha kiiremini operatsioone ahela mõlema otsaga, teinekord mugavam ka keskmiste elementide töötlemiseks.

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

Järgnevalt kaks näidet kahe viidaga ahela algoritmidest – üks on sõlme lisamisest ja teine sõlme kustutamisest.

## Sõlme lisamine

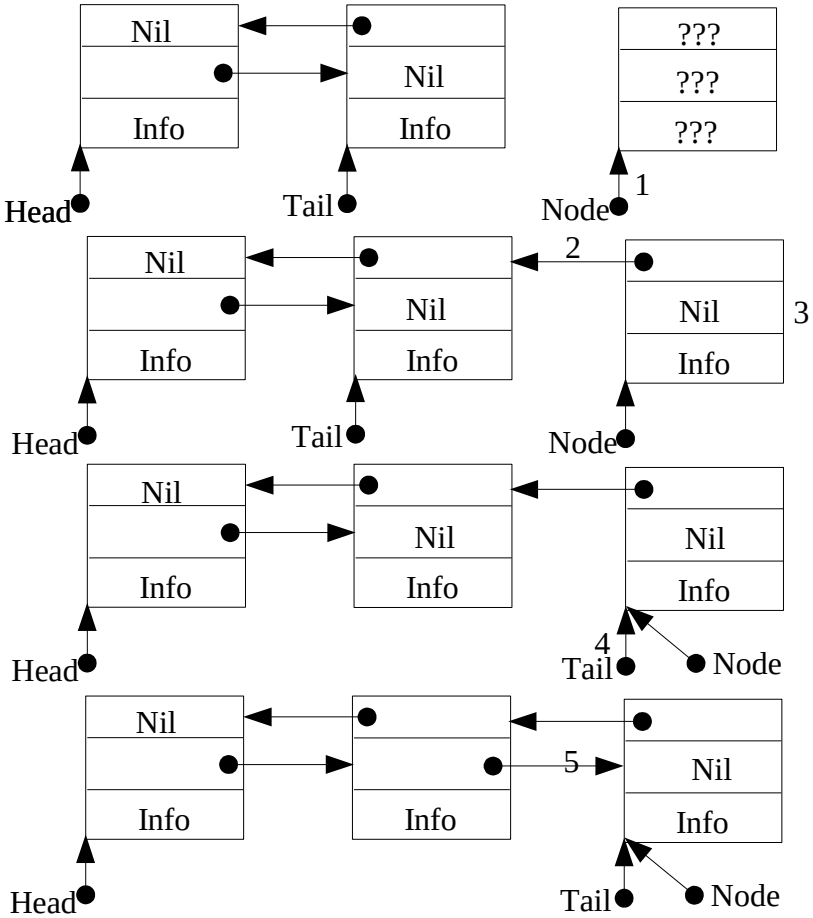
Ülesanne: lisa kahe viidaga ahela lõppu uus sõlm.

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

*// Ahela 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
// Ahel 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 ahela lõppu.




*Joonis 5: Elemendi lisamine kahe viidaga ahela lõppu*

## Sõlme kustutamine

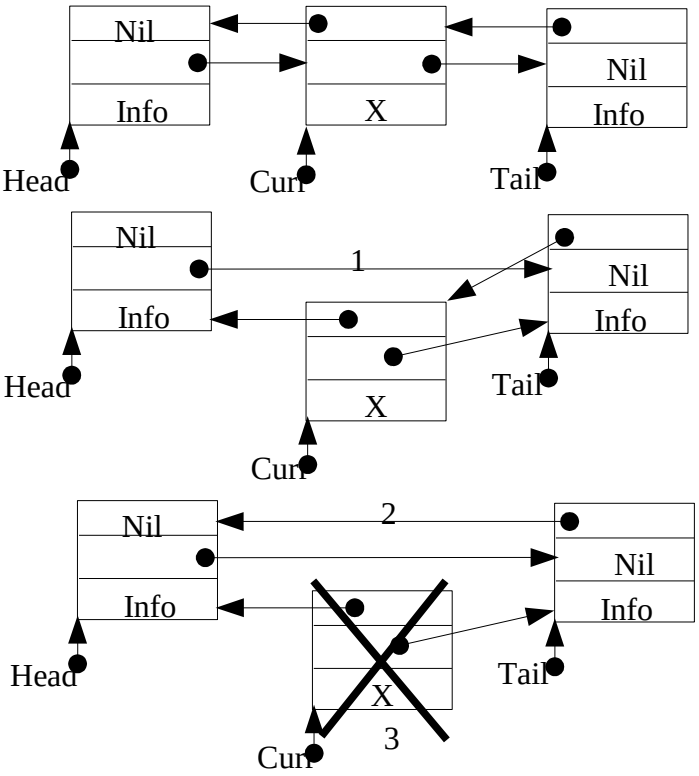
**Ülesanne: kustuta kahe viidaga ahelast sõlm, mis sisaldab informatsiooni X.**

Sõlme kustutamisel kahe viidaga ahelast ei ole vaja kasutada kahte abiviita, sest ahela keskel on nii eespool kui ka 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 6.

```
// Ahelast kustutatakse sõlm infoga X
Current = Head
// Liigume mööda ahelat õige 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
```

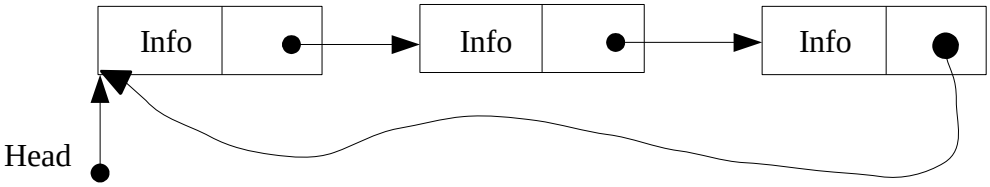
**Kood nr 6.** Sõlme kustutamine kahe viidaga ahelast.




*Joonis 6: Sõlme kustutamine kahe viidaga ahelast*

## Ringahel

**Ringahela** *(Circular linked list)* puhul on ahela viimane element seotud esimese elemendiga (viitab esimesele elemendile). Ahela 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 ahela elemendile. Ringahelat peetakse mõnikord ka mittelineaarseks struktuuriks. Reaalne struktuur, mille esitamiseks ringahelat kasutatakse, ei pea olema olemuselt tsükliline. Ahelat läbides on ilmselt vajalik hoida siiski üks viit paigal, et hiljem saaks tuvastada, kas kõigile elementidele on ring peale tehtud.




*Joonis 7: Ringahel*

## *NB! Üldised märkused*

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

# Ahelad C vahenditega

Ahela 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 siinkohal C-keele süntaksi ja põhiliste mõistete kohta.

## Viitmuutujad C-s

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 ahelaid moodustada.

Viitmuutujat saab C-s deklareerida järgmiselt (i on viit täisarvulisele muutujale ehk selle muutuja aadress):

```
int *i;
```

Samal ajal on `i` ka ise muutuja, kuid ei sisalda mitte täisarvu vaid aadressi. Täisarvu enda jaoks tuleb täiendavalt mäluruum eraldada

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

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 aadress on salvestatud muutujasse `i`. Täisarvu salvestamine eraldatud mälupesadesse toimub järgmiselt:

```
*i = 10;
```

Seda tegevust nimetatakse otsendamine (*dereferencing).*

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

```
int arv;
arv = 20;
i=&arv;
```

Tehtud lausete tulemusena on muutujas `i` on nüüd muutuja `arv` aadress 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 `&i`?

NB! Tuleb tähele panna, et viidad on reeglina tüpiseeritud ehk kasutada saab viita täiarvule, viita ujukomaarvule, viita struktuurile, massiivile jne. Ehkki aadressid on alati ühesugused, annab selline tüpiseerimine võimaluse harrastada viitadega aritmeetikat nagu näiteks `i++`. 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.

## Ahelad

Ahelate moodustamiseks on vaja kasutada keerulisemat struktuuri kui lihtsalt üks viitmuutuja. St on vaja moodustada "kahe- või rohkemaosalised kastid". Selleks sobib struktuurne andmetüüp, mille programmeerija ise kirjeldama peab - nn **struktuur** (`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 võib rekursiivselt kasutada struktuuri enda tüüpi. 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 viida tüüpi.

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

```
struct element node;          // deklareeritakse terve struktuur oma väljadega
```

```
struct element *nodepointer;   // deklareeritakse viit struktuurile, struktuuri enda jaoks tuleb veel mälu eraldada
```

Tüüpiliselt kaustatakse kirjetes ja samuti C struktuurides väljadele viitamiseks kirjaviisi, kus välja nimi eraldatakse muutujanimest punktiga (näit: `s6lm.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.

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

Järgnevalt mõned näited ahelaga toimetamisest. Näited kasutavad sama struktuuri, mis eespool kirjeldatud on.

## Ahela läbimine

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

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

Tsükkel töötab seni, kuni abimuutuja `current` on jõudnud ahela lõppu.

## Ahela moodustamine

Järgnev näide loob ahela, kusjuures uus element lisatakse alati ahela 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 ahelast

Järgnev näide kustutab ahelast elemendi, mille võtmeks on x. Näide on poolik. Eeldatakse, et ahel on enne 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 arv puudub
         printf("Selline element puudub\n");
    }
```

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