Lineaarsed andmestruktuurid

Järgnevas tekstis on loendina silmas peetud jadamisi paikevaid andmeid, sõltumata realisatsioonist.

Ühes rakenduses on andmete töötlemisel harva vaja kasutada kõiki loendite jaoks kirjeldatud operatsioone, nagu neid mainitud oli loendite materjalis. Lisaks võivad teatud loenditüübid andmetüüpidena ja koos vajalike operatsioonide-funktsioonidega olla juba programmeerimiskeeles olemas ja vajadust neid ise programmeerima hakata tegelikult ei ole. Samas ei saa kasutada mingit spetsiifilist loenditüüpi, kui ei tea, millisel moel ta käitub ja milliseid võimalusi omab.

Kõige tihedamini on vaja elemente lisada ja eemaldada loendi otstes. Ajalooliselt ja vajadustest lähtudes on kujunenud kolm enim kasutatavat lineaarset andmestruktuuri koos iseloomulike omaduste ja operatsioonidega: pinu, järjekord ja dekk. Just sellised lineaarsed struktuurid on kõige efektiivsemad ja kõige vajalikumad.

Järgnevates peatükkides kirjeldatakse nimetatud andmestruktuuride ülesehitust, realiseerimise võimalusi ning kasutusvõimalusi.

Pinu ehk magasin

> **Pinu** ehk **magasin** *(stack)* on lineaarloend, kuhu elemente lisatakse ja kust elemente kustutatakse ühest ja samast otsast, **pinu tipust** *(top)*.

Nimi magasin tuleb sarnasusest automaadi magasiniga. Kes pole automaadi magasini näinud, siis selgituseks kasutatakse ka taldrikute kuhja, kus targem on ükshaaval taldrikud üksteise peale laduda ja neid siis pealtpoolt ka ükshaaval võtta. Samuti võib näiteks toole pinutada, sõltuvalt tooli konfiguratsioonist ei pruugi see alati õnnestuda. Võib-olla oli üheks lapsepõlve mänguasjaks varras, mille otsa sai ükshaaval puust või plastist kettaid asetada? Kõigil toodud näidetel on ühisteks omadusteks – viimasena paika pandud asja saab kätte esimesena ja analoogiliselt esimesena paika pandud asja viimasena. See ongi pinu tööpõhimõte. Sellist tüüpi struktuuri kutsutakse inglise keeles ka *last-in-first-out (LIFO)* struktuuriks. Mõiste ja tööpõhimõte pole kasutusel ainult tarkvaraarenduses, vaid ka riistvaras.

Pinu tööpõhimõtet on esmakordselt maininud Alan Turing 1946 ning Konrad Zuse 1945 (seoses alamprogrammide väljakutsumistega). Põhjalikumalt kirjeldasid pinu tööpõhimõtet 1955. aastal Klaus Samelson ja Friedrich Bauer. Viimane sai pinu põhimõtte leiutamise eest 1988. aastal auhinna Computer Pioneer Award.

Pinu operatsioonid

Lähtudes eelnevast kirjeldusest on pinu-tüüpi andmestruktuuri jaoks kõige olulisemad kaks elementidega töötamise operatsiooni:

1. Elemendi lisamine pinusse, enamasti nimetatud `push()`
2. Elemendi kustutamine pinust, enamasti nimetatud `pop()`

Lisaks võivad osutuda vajalikuks järgmised operatsioonid:

3. Uue pinu loomine.
4. Kontroll, kas pinu on tühi (pinust ei saa midagi kustutada, kui seal ei ole enam ühtegi elementi).
5. Kontroll, kas pinu on täis (ruum uute elementide lisamiseks on otsa saanud).

Lisaks loetletutele võivad lisanduda veel mõned operatsioonid, mis aga kindlasti pinu lõhkuma hakata ei tohi (näiteks on teinekord olemas võimalus nö pealmise elemendi vaatamiseks teda kustutamata). Igasugused muud variandid nagu näiteks pinu keskele elemendi lisamine või keskelt vaatamine ja kustutamine on keelatud. Lubatud ei ole ka pinu läbimine või andmete ümberjärjestamine. Kui nimetatud operatsioonid vajalikud on, siis tuleb otsida mõni teine andmestruktuur. Pinu on seega pigem ajutine andmete hoidla, mille toimimise loogika annab teatud viisil kätte sinna eelnevalt pandud andmeelemendid.

Väljakujunenud terminoloogia kohaselt räägitakse pinu **tipust** *(top)* ja **põhjast** *(bottom)* ning teda kujutatakse visuaalselt pigem vertikaalse struktuurina.

Pinu kasutamine

Kuna pinu on üsna spetsiifiline struktuur ja tavaelus tihti ei esine, siis leiab ta kasutamist peamiselt erinevates arvutiteaduslikes rakendustes. Samuti rakendatakse pinu põhimõtet riistvaraliselt.

Näiteks alamprogrammide väljakutsete organiseerimine programmis. Uue alamprogrammi väljakutse tähendab seda, et programmi täitmine jääb teatud kohas pooleli ja peab peale alamprogrammi töö lõppemist jätkuma samalt kohalt. Selleks pannakse pooleli jäänud alamprogramm pinusse koos oma muutujate komplektiga. Kui momendil töötav alamprogramm oma töö lõpetab, võetakse pinust välja kõige peal olev alamprogramm ja kogu väljakutsete loogika on just selline, et see on see õige, millega edasi minna. Selliselt paigutuvad kõik pooleli jäänud alamprogrammid üksteise otsa. Kui nüüd ühe alamprogrammi täitmine lõpeb, siis võetakse tema asemel pinust eelmine alamprogramm, mis lõppenu välja kutsus jne kuni pinu tühjaks saab (vahepeal võidakse vajaduse korral ka uusi alamprogramme juurde lisada).

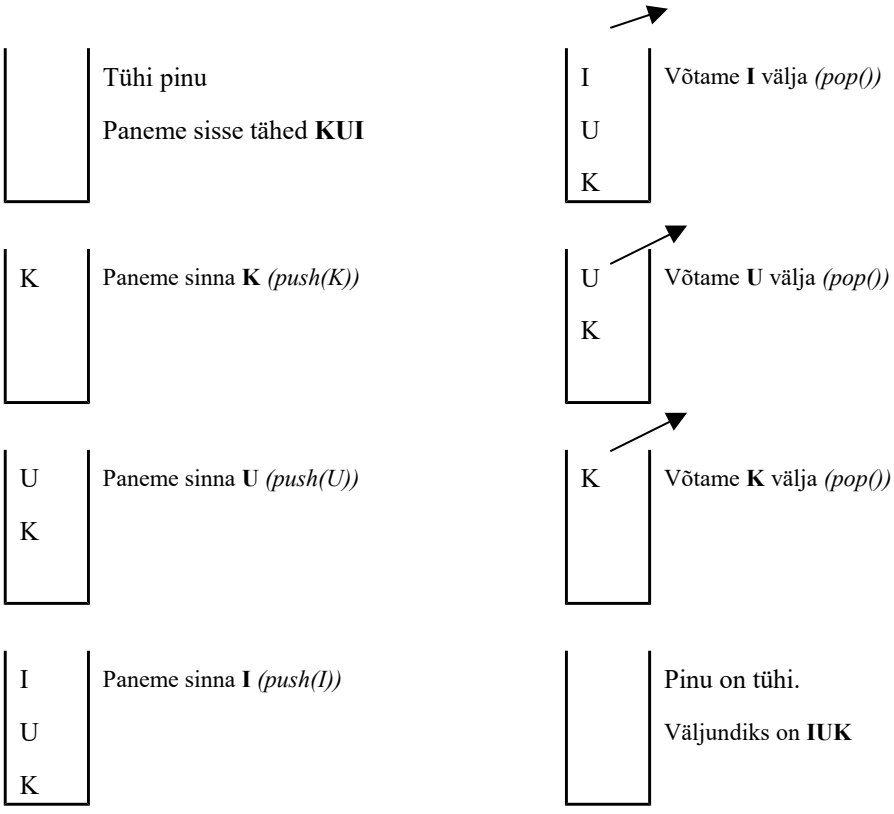
Osade programmeerimiskeelte kompilaatorid kasutavad pinu avaldiste või ka suuremate plokkide süntaksi läbivaatamiseks (parsimiseks).

Pinu saab kasutada teatud tüüpi algoritmide puhul, mille põhimõtet kutsutakse inglise keeles *backtracking* (tagurdus) ja millega me ka veidi edaspidi tutvust teeme.

Arvuti arhitektuuris on pinu piirkond arvuti mälus, kus andmete lisamine ja kustutamine toimuvad pinu ehk LIFO põhimõttel.

Näide:

Kasutame pinu mingite väärtuste rea tagurpidi pööramiseks:



Pööratud poola kuju

1951 a pakkus Poola loogik Jan Lukasiewicz välja, kuidas panna kirja loogikaavaldisi sulge kasutamata. Kirjaviis levis ka algebrasse ja mujale, kus operaatoreid-operande kasutatakse. Süsteem sai nimeks **Poola kuju** *(Polish notation)*.

Me oleme harjunud aritmeetikaavaldisi kirjutama nii, et operaator (tehtemärk) asetatakse operandide (näiteks arvud) vahele. Aga see ei pea olema ainus võimalus tehete esitamiseks.

Teistsuguse kirjapildi idee on selles, et kui operaatorid panna operandide ette või järele ja kui operaatori poolt töödeldavate operandide hulk on üheselt määratud, saab avaldise üheselt mõistetavalt kirja panna sulge kasutamata. Algselt pandi operaatorid operandide ette ja see kirjaviis sai nimeks **Poola kuju** *(Polish notation)*.. Edasi viidi operaatorid operandide järgi ja seda kutsutakse **pööratud Poola kujuks** *(reverse Polish notation - RPN)*. Kokkuvõttes võib öelda, et nii loogika kui ka aritmeetikaavaldisi saab kirja panna kolmel erineval kujul: **infiks** `(a+b)`, **prefiks** `(+ab)` ja **postfks** `(ab+)` kujul. Viimased kaks on alternatiivsed mõisted Poola ja pööratud Poola kujule.

Pööratud Poola kujul olevat avaldist kasutavad nii mõned taskukalkulaatorid kui ka programmeerimiskeeled. Meile pakub ta hetkel huvi kui üks näide, mille raames saame katsetada pinu kasutamist.

Näide: Infiks kuju on `(a*b)+c`, sama avaldis postfiks kujul oleks `ab*c+`

Avaldise postfiks kujule teisendamine

Järgnev teisendusalgoritm eeldab, et tavalises avaldises on maksimaalselt sulge, st kõigi tehete järjekord on määratud sulgudega. On ka alternatiivseid algoritme, mis meie jaoks loomuliku tehete järjekorraga arvestavad.

Tööta märkhaaval

- Arv kirjuta väljundisse.
- Vasakut (avanevat) sulgu ignoreeri.
- Operaator (tehtemärk) pane pinusse.
- Parempoolse (sulgeva) sulu puhul võta üks operaator pinust ja kirjuta väljundisse.

Postfiks-kujul oleva aritmeetika-avaldiste arvutamine

Aritmeetika-avaldiste arvutamisel on üheks teeks muuta nad kompilaatorite poolt sulgudeta postfiks-kujule. Seejärel kasutatakse pinu avaldise väärtuse leidmiseks. Pinus hoitakse operande. Arvutusreeglid on väga lihtsad:

- Töötle avaldises olevaid operaatoreid ja operande vasakult paremale
- Kui tegemist on operandiga, lisa ta operandide pinusse.
- Kui tegemist on operaatoriga, võta pinust välja kaks operandi ja rakenda neile vastavat operaatorit ning pane tulemus pinusse tagasi.

Ainus koht, kus aritmeetikaavaldiste puhul probleem tekib on miinus-märgi kasutamine. Tavaliselt lahendatakse see nii, et miinust loetakse unaarseks operaatoriks ja käsitletakse koos arvuga. Lahutamine pannakse aga kirja `a+(-b)`, ehk postfikskujul: `a-b+` (kus miinus kuulub b-ga kokku)

Pinu realisatsioon

Pinu realisatsiooni tehnika ehk see, kuidas pinu arvutis programmeerides üles ehitada, sõltub kasutuseesmärgist ja programmeerimiskeelest. On tüüpiliselt kaks võimalust:

1. **Staatiline meetod** kasutades **massiivi**. Sel juhul on pinu maht piiratud massiivi elementide arvuga (kui massiiv ise on staatiline). Pinusse lisatavad andmed kirjutatakse järjest massiivi. Viimati lisatud elemendi asukohta (pinu tippu) saab meeles pidada massiivi indeksi abil.

2. **Dünaamiline meetod** kasutades **ühe viidaga ahelat**. Uue andmeelemendi lisamisel pinusse lisatakse ahelasse uus element ja andmeelemendi kustutamisel pinust kustutatakse element ahelast.

Teostus massiiviga

Iga massiivi element (lahter) sisaldab infot pinu ühe elemendi kohta. Pinu tipuks on indeks, mis määrab, kui kaugele massiivi algusest täidetud pinu ulatub. Tühja pinu tähistab pinu tipu (indeksi) võrdumine 0 (nulliga). Kui uus element lisatakse pinusse, suurendatakse tipu (indeksi) väärtust ühe võrra ja uus element salvestatakse massiivi vastavasse lahtrisse. Elemendi kustutamiseks tuleb elemendi väärtus tipulahtrist välja võtta ja tippu nihutada koha võrra ettepoole (ehk lahutada indeksist 1). Tipu väärtuse järgi saab ka jälgida, et pinu ei täituks üle.

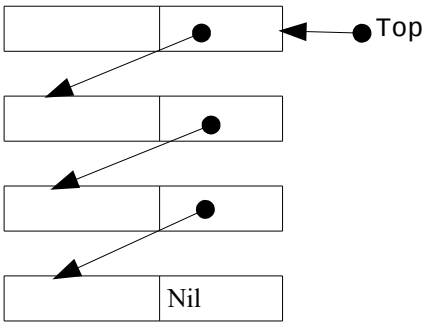
Selles kirjelduses on loomulikult võimalikud väikesed muutused – näiteks tühja pinu tähistab indeks -1. Või pinu tipu indeks on alati massiivi esimese tühja lahtri indeks.

Ehkki tehniliselt on kõik massiivi lahtrid suvalisel hetkel indeksite kaudu ligipääsetavad, tuleb seada nö loogiline tõke ja töötada pinuga ainult reeglitele alludes. Ainus element, mida pinust võtta võib on see, mis paikneb tipu indeksiga määratud lahtris (või selle ees, sõltuvalt konkreetsest realisatsioonist).

Usun, et ka üsna vähese "massiiviteadlikkusega" on selline pinu võimalik ära programmeerida, soovitavalt tehes funktsioonid kõigi oluliste tegevuste, ennekõike aga lisamise ja kustutamise jaoks.

Teostus ahelaga

Dünaamilise realisatsiooni korral ei pea elemendid paiknema füüsiliselt järjestikku, vaid järgnevusseose määravad viidad. Ahela esimese elemendi mäluaadress sobib ka pinu tipuks. Pinu põhioperatsioone (elemendi lisamine ja kustutamine) peab saama teha mugavalt ja kiirelt. Teostuse seisukohast on mugavam, kui elemendid viitavad pinu põhja poole. Miks see nii on, mõtle ise välja.



Joonis 1 Pinu ühe viidaga ahelana

Algoritmiliselt on lisamine ja kustutamine (loodetavasti) selged üldise ahelate jutu juurest. Sisuliselt on tegemist elemendi lisamisega ahela algusesse ja elemendi kustutamisega ahela algusest

Järjekord e. saba

> **Järjekord ehk saba** *(queue)* on lineaarloend, kus elementide lisamine toimub alati loendi ühes otsas ja elementide eemaldamine teisest otsast.

Järjekord on igapäevases elus üsna tavaline "andmestruktuur". Näiteks eksisteerib järjekord poes kassas, arsti ukse taga või TLÜ kohvikus. Järjekorra puhul on vaja opereerida järjekorra andmetega – jälgida, millised andmed järjekorda lisanduvad ning neid ka lisandumise järgi teenindada.

Informaatikas kasutatav järjekord töötab tavalise järjekorra põhimõttel. Põhilised operatsioonid on järjekorda lisamine ja järjekorrast kustutamine. Esimesena kustutatakse järjekorrast see element, mis esimeena järjekorda lisati. Seega on järjekorral kaks töötavat otsa – ühte lisatakse elemente juurde ning teisest kustutatakse. Järjekorda kutsutakse inglise keeles ka *first in first out (FIFO)* struktuuriks: kes esimesena järjekorda lisatakse *(first in)*, see teenindatakse samuti esimesena *(first out)*. Lisaks eksisteerivad erilised nn eelistusjärjekorrad.

Järjekorra operatsioonid

Järjekorra põhioperatsioonid on analoogilised pinu operatsioonidega. Samuti võib leida järjekorra kirjeldusi, kus lisatud näiteks järjekorras olevate elementide arvu teadmine või järgmise teenindatava vaatamine (aga mitte teenindamine) jne. Peamised järjekorra operatsioonid on järgmised:

1. Elemendi lisamine järjekorra lõppu (tihti nimetatud `enqueue()`)
2. Elemendi kustutamine järjekorra algusest (nimetatud `dequeue()`)

Lisaks võivad vajalikud olla ka:

3. Uue tühja järjekorra loomine.
4. Kontroll, kas järjekord on tühi.
5. Kontroll, kas järjekord on täis.

Nagu näha, on operatsioonid sarnased pinuga. Põhimõtteline vahe ongi selles, kuhu elemente juurde panna ja kust ära võtta. Terminoloogias räägitakse tavaliselt järjekorras olemisest, järjekorra algusest ja lõpust ning visuaalselt kujutatakse järjekorda horisontaalse struktuurina.

Järjekorra kasutamine

Järjekorda kasutatakse siis, kui andmed saabuvad kindlas ajalises järgnevuses ja saabumise järgnevus on oluline ka andmete töötlemisel. Näiteks operatsioonisüsteem paneb saabunud käsud / päringud järjekorra lõppu ja võtab neid järjekorra algusest täitmiseks (ajajaotussüsteemide puhul). Paljud tegeliku elu probleemid taanduvad järjekorra kasutamisele. Näiteks süsteem, mis lubab lennukile pileteid broneerida. Broneerimissoovid saabuvad erinevaist paigust erinevatel ajahetkedel, samas järjekorras tuleks need soovid ka rahuldada ja broneeringud teha.

Järjekorral võib olla ka veidi teistsugune iseloom. Näiteks tuuakse välja järgmised “erilised” järjekorrad:

- Mitme teenindajaga järjekord *(Multiple server queue);*
- Prioriteetidega järjekord ehk eelistusjärjekord *(Priority queue)*;
- Piiratud pikkusega järjekord *(Bounded length queue)*.

Nime järgi on võimalik neile reaalelust mingeid vasteid leida. Pikemalt nendest selles kursuses juttu ei tule (välja arvatud prioriteetidega ehk eelistusjärjekorrast, millest võib lugeda eraldi materjalist)

Järjekorra realisatsioon

Järjekorra realisatsiooni tehnika sõltub kasutatavast programmeerimiskeelest ning eesmärgist. On tüüpiliselt kaks võimalust, täpselt nagu oli pinu puhulgi:

1. Staatiline ehk kasutades **massiivi**.
2. Dünaamiline ehk kasutades **ahelat**.

Oma olemuselt vastab järjekord rohkem teisele variandile, kuid indeksit mõistlikult kasutades saab tagada järjekorrale omase käitumise ka massiivis. Samas võib sarnaselt pinuga ka järjekorra jaoks olla keeles juba andmetüüp olemas ja seda on vaja vaid õiges kohas ja õigel viisil kasutada.

Teostus massiiviga

Massiivi abil järjekorda teostades peab meeles pidama kahte indeksit (`head` ja `tail`). Andmeid lisatakse indeksi `tail` järgi ning kustutatakse indeksi `head` järgi. Seega sarnaselt pinule muudetakse nii elemendi lisamisel kui ka kustutamisel vastavaid indekseid. Et massiiv kustutamiset ja lisamiste käigus liiga kiiresti otsa ei saaks (järjekord liigub justkui mööda massiivi edasi lisamise ja eemaldamise käigus), tuleb massiivi lõpu täitumisel jätkata elementide lisamist massiivi algusesse, kus loodetavasti selleks ajaks järjekorras olnud elemendid juba kustutatud on. Idekseid omavahel võrreldes on võimalik aru saada, kas järjekord on tühi või on ta hoopiski täis saanud.

Teostus ahelaga

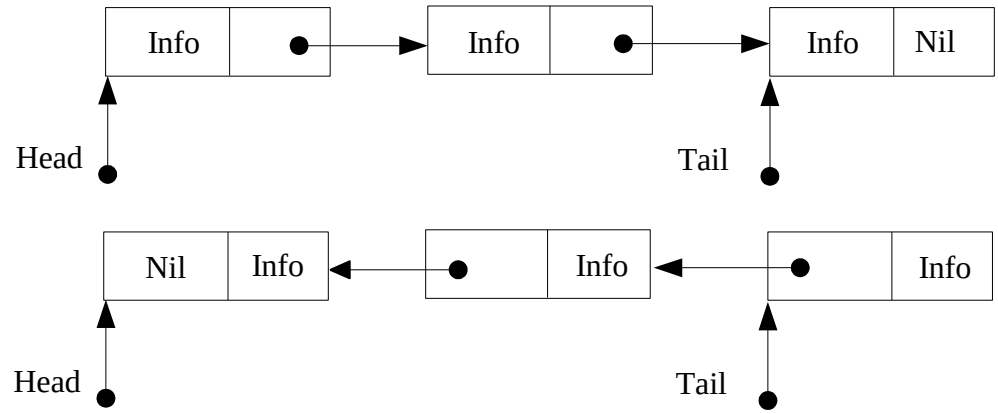
Enne ahela abil realiseerimisele asumist tuleb teha mõned põhimõttelised otsused:

- Mitu viita järjekorraga siduda?

Ilmselt on kasulik, et viitasid oleks kaks – üks algusele (`head`) ja teine lõpule (`tail`). Põhjus on järjekorra eripäras, sest ligi on vaja pääseda ahela mõlemale otsale.

- Mis pidi panna jooksma viidad?

Selleks tuleks joonistada ühe viidaga ahel ja proovida, kuidas lisamine ja eemaldamine mugavamalt õnnestuvad.



Joonis 2 Järjekord, kaks varianti

Arvestame, et lisamine toimub järjekorra lõppu (`tail`) ja kustutamine ehk teenindamine järjekorra algusest (`head`). Sel juhul on mugava kasutada esimest joonisel 2 olevat varianti. Et selles veenduda tuleb mõlemal juhul proovida ette kujutada, mida tähendab elemendi lisamine ja kustutamine ahela mõlemas otsas.

Dekk e. kaheotsaline järjekord

> **Dekk** *(deque, double-ended queue)* on lineaarloend, kus elementide lisamine ja kustutamine on lubatud mõlemast otsast. Samuti on mõlemast otsast lubatud juurdepääs andmetele.

Dekki nimetatakse magasini ja järjekorra üldistuseks. Üldistus tähendab seda, et eemaldamine ja lisamine on lubatud dekki mõlemast otsast. Terminoloogias räägitakse deki vasakust ja paremast otsast. Eristatakse veel kahte piiratud varianti: piiratud väljundiga *(output-restricted deque)* ja piiratud sisendiga *(input-restricted deque)* dekki. St, et vastav tegevus piiratakse ühe deki otsaga.

Deki operatsioonid:

1. Lisada element deki algusesse.
2. Lisada element deki lõppu.
3. Eemaldada element deki algusest
4. Eemaldada element deki lõpust.

Lisaks:

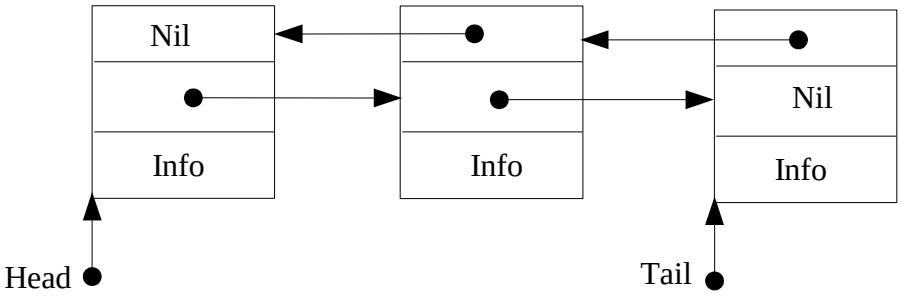
5. Uue tühja deki loomine.
6. Kontroll, kas dekk on tühi.
7. Kontroll, kas dekk on täis.

Kasutamine

Sobib siis, kui pinu või järjekorra võimalused piiratuks jäävad.

Realisatsioon

Võimalik on kasutada nii massiivi kui ka ahelat, kuid massiivi puhul muudab olukorra ebamugavaks see, et lisada peab saama mõlemasse otsa. Seetõttu on ahela kasutamine loomulikum. Kuid juba järjekorra realiseerimisel sai selgeks, et ühe viidaga ahel hästi ei sobi viitade ühtpidi suunatuse tõttu. Kõigi operatsioonide mugavamaks korraldamiseks tuleb kasutusele võtta hoopis kahe viidaga ahel. Kui meil on selged operatsioonid kahe viidaga ahela peal, siis rohkem probleeme ei tohiks ka deki haldamine tekitada. Kindlasti peab dekiga seotud olema kaks viita – vasakule ja paremale otsale, et lisamisi ja eemaldamisi mõlemas otsas efektiivsemalt teha.

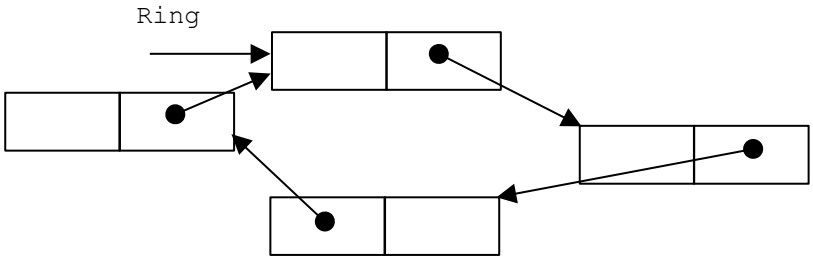


Joonis 3. Kaheotsaline järjekord, mis moodustatakse kahe viidaga ahela abil.

Ringjärjekord

> **Ringjärjekord** *(Circular queue)* on andmestruktuur, mille erisuseks on see, et viimane ja esimene element on omavahel seotud.

Mõne rakenduse seisukohalt on just selline järjekord parim. Samal ajal saab ringjärjekorra abil imiteerida ka tavalist järjekorda.



Joonis 4. Ühe viidaga ringjärjekord. Algust (ja lõppu) tähistab viit `Ring`.

Realisatsioon

Realiseerimiseks ahela abil piisab ühe viidaga ahelast, kus viimase elemendi viidaväljas on `null`-viida asemel esimese elemendi aadress. Kindlasti on vaja ühte välist viita mingile elemendile.

Realistasioon C keelt kasutades

Abstraktne andmetüüp

Seoses pinu ja järjekorraga tuleb üle vaadata mõiste **abstraktne andmetüüp** *(Abstract Data Type – ADT)*. ADT kirjeldab operatsioonide hulga, mida antud andmetüübiga teha saab ja operatsioonide tähenduse (semantika), kuid ei kirjelda ei andmetüübi enda ega operatsioonide teostust. Selline abstraktsus (üldistus) on kasulik järgmistel põhjustel:

- Lihtsustub vastavat andmetüüpi kasutavate algoritmide üleskirjutamine ning ka programmeerimine, sest ei pea pidevalt mõtlema sellele, kuidas täpselt üks või teine operatsioon realiseeritakse (vt näiteks pööratud Poola kuju kirjelduses esitatud algoritme).
- Kuna ADT-d saab tavaliselt realiseerida mitmel viisil, siis tekib võimalus vajaduse korral realisatsiooni muuta samal ajal muutmata ADT-d kasutava algoritmi / programmi loogikat.
- Tuntumad ja vajalikud ADT-d on tihti keeltes realiseeritud ja teinekord ei olegi vaja hädasti mõtelda sellele, kuidas ADT-d ennast üles ehitada (ehkki kasulik on ikka teada, mis "karul kõhus", et otsustada sobivama realisatsiooni üle).
- ADT-de operatsioonid annavad võimaluse neist algoritmidest rääkida kergemini ka inimkeeles.

Pinu

Pinu tuuakse tihti näiteks, kui räägitakse ADT-st ja just sellise abstraktse andmetüübina teda järgnevalt vaatame. Pinu **liides** *(interface)* (operatsioonide kirjeldused) on tüüpilisi operatsioonide nimetusi kasutades järgmine:

`init()`

Uue tühja pinu algväärtustamine

`push()`

Lisa uus element pinu tippu.

`pop()`

Kustuta ning tagasta element pinu tipust.

`isEmpty()`

Kontroll, kas pinu on tühi.

Üks võimalus pinu realiseerimiseks on kasutada massiivi. C-s võiksid vastavad funktsioonid olla järgmised (näide on kursuse veebis samuti failina olemas):

```
static char *stack;   // Viida deklareerimine pinu alguse jaoks, hiljem eraldatakse mälu massiivi jaoks
static int N;

// Pinu loomine, mälus reserveeritakse soovitud arv mälupesi massiivi jaoks
void init(int maxN)
  { stack = malloc(maxN*sizeof(int)); N = 0; }

// Tagastab true või false vastavalt sellele, kas pinu on tühi või mitte
int isEmpty()
  { return N == 0; }

// Lisab väärtuse pinusse indeksile N
void push(char item)
  { stack[N] = item; N++;}

// Tagastab pinust viimase väärtuse
char pop()
  { N--; return stack[N];}
```

Sellised väikesed funktsioonid võivad esmapilgul tunduda mõttetud. Kuid päris nii see siiski pole. Kaks peamist põhjust on:

1. Omades sama liidesega, kuid ahelat kasutavat pinu realisatsiooni, võib neid rahulikult üksteise vastu väljavahetada.
2. Viies olulised operatsioonid üheselt mõistetavate nimede alla pääseb programmi kirjutaja mõtlemisest pinu tehniliste üksikasjade üle.

Järjekord

Järjekorra ADT liides on järgmine (sisluliselt sarnased operatsioonid pinuga, kui traditsiooniliselt teised nimed):

`init()`

Tühja järjekorra loomine.

`enqueue()`

Uue elemendi lisamine

`dequeue()`

Viimati lisatud elemendi kõrvaldamine ja väärtuse väljastamine

`isEmpty()`

Kontroll, kas järjekord on tühi.

Edasi on vaja meetodid realiseerida ja järjekord ehk saba on kasutatav.

Võrdleme üldiste lineaarloendite realiseerimist massiivi ja viitadega

1. Viida jaoks tuleb elemendis eraldada lisaks üks või kaks välja, mis mälu võtab. Kui aga ahelaid on mitu, siis saavad nad kasutada vaheldumisi sama mälupiirkonda. Viitadega realisatsioon on tavaliselt efektiivsem, sest pole vaja igaks juhuks mälu eraldada.
2. Elemendi lisamine vahele on lihtsam – selleks tuleb vaid uus element tekitada ja viidad ringi tõsta. Massiivi puhul tuleks aga nö paremal paiknevad elemendid enne edasi nihutada.
3. Elemendi eemaldamiseks on samuti vaja ainult viidad ringi tõsta, massiivi puhul aga mälupesade sisud ringi kirjutada.
4. Suvalise k-nda elemendi poole pöördumine on järjestikuse paiknemise puhul lihtsam ja kiirem – aeg on konstantne, viitadega struktuuri korral sõltub pöördumise aeg k suurusest, st tuleb teha k kordust, et õiget elementi kätte saada.
5. Viitadega realisatsiooni korral on kergem organiseerida nimistute ühendamist ja nimistute lõhkumist mitmesse ossa, mis hiljem iseseisvalt kasvada/kahaneda saavad.
6. Viitadega realisatsiooni korral saab luua palju keerulisemaid struktuure: näiteks linkides iga ahela elemendiga terve uue ahela.
7. Loendi läbimine toimub tavaliselt kiiremini järjestikulise paiknemise (massiivi) korral.