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 loend, kuhu elemente lisatakse ja kust elemente kustutatakse ühest ja samast otsast, **pinu tipust** *(top)*.

Pinu on loend, mille ehitusviis ja hooldus on sellised, et järgmisena võetakse kõige hilisemana salvestatud andmeelement. Seda meetodit iseloomustatakse väljendiga "viimasena sisse, esimesena välja" (LIFO-printsiip).

(IT-Terministandardi sõnastik EVS-ISO 2382-4:1999)

Nimetus magasin tuleb sarnasusest automaadi magasiniga. 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, ehkki 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 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. a ning Konrad Zuse 1945. a seoses alamprogrammide väljakutsete korraldamisega. 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 ühtegi elementi).
5. Kontroll, kas pinu on täis (kas ruum uute elementide lisamiseks on otsa saanud).

Lisaks loetletutele võivad lisanduda veel mõned operatsioonid, mis aga pinu lõhkuda ei tohi (näiteks võib olla 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 riistvaras.

Näiteks toimub pinu põhimõtte kohaselt funktisoonide väljakutsete organiseerimine programmis. Uue funktsiooni väljakutse tähendab seda, et programmi täitmine jääb sel kohal pooleli ja peab peale funktsiooni töö lõppemist jätkuma samalt kohalt. Selleks pannakse pooleli jäänud funktsioon pinusse koos oma muutujate komplektiga. Kui momendil töötav funktsioon oma töö lõpetab, võetakse pinust välja kõige peal olev funktsioon ja kogu väljakutsete loogika on just selline, et see on see õige, millega edasi minna. Selliselt paigutuvad kõik pooleli jäänud funktsioonid üksteise otsa. Kui nüüd ühe funktsiooni täitmine lõpeb, siis võetakse tema asemel pinust eelmine funktsioon, mis lõppenu välja kutsus jne kuni pinu tühjaks saab (vahepeal võidakse vajaduse korral ka uusi funktsioone 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.

Pööratud poola kuju

Oleme harjunud aritmeetikaavaldisi kirjutama nii, et operaator (tehtemärk) asetatakse operandide (näiteks arvud) vahele. Aga see ei ole ainus võimalus tehete esitamiseks. 1951 a pakkus Poola loogik Jan Lukasiewicz välja, kuidas panna kirja loogikaavaldisi tehetejärjekorda muutvaid sulge kasutamata. Kirjaviis levis ka algebrasse ja mujale, kus operaatoreid-operande kasutatakse.

Teistsuguse kirjapildi idee (operaatorid operandide ees või järel) on anda võimalus avaldise üheselt mõistetavalt kirja panemiseks sulge kasutamata. Algselt pandi operaatorid operandide ette ja see kirjaviis sai nimeks **prefiksesitus** (ka **Poola kuju** *(Polish notation))*. Edasi viidi operaatorid operandide järgi ja seda kutsutakse postfiksesituseks (ka **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.

IT terministandardi sõnastikust (EVS-ISO 2382-2:1999):

Prefiksesitus on matemaatiliste avaldiste esitusviis, kus iga tehtemärk eelneb oma operandidele ning tähistab tehet, mis tuleb sooritada talle järgnevate operandide või vahe tulemitega.

Postfiksesitus on matemaatiliste avaldiste esitusviis, kus igale tehtemärgile eelnevad ta operandid ning tehtemärk tähistab tehet, mis tuleb sooritada talle eelnevate operandide või vahe tulemitega.

Avaldise postfiksesitust kasutavad nii mõned taskukalkulaatorid kui ka programmeerimiskeeled. Meile pakub ta hetkel huvi kui üks näide, mille raames saame katsetada pinu kasutamist.

Näide: Infiksesitus on `(a*b)+c`, sama avaldis postfiksesituses on `ab*c+`

Avaldise teisendamine postfiksesitusse

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.
- Lahtisulgu (vasakut) ignoreeri.
- Operaator (tehtemärk) pane pinusse.
- Kinnisulu (parema) leidmisel võta üks operaator pinust ja kirjuta väljundisse.

Postfiksesituses oleva aritmeetikaavaldiste arvutamine

Aritmeetikaavaldiste arvutamisel on üheks teeks muuta nad kompilaatorite poolt sulgudeta postfiksesitusse. Seejärel kasutatakse pinu avaldise väärtuse leidmiseks. Pinus hoitakse operande. Arvutusreeglid on väga lihtsad:

Töötle järjest avaldises olevaid operaatoreid ja operande vasakult paremale:

- Operand (arv) lisa pinusse.
- Operaatori (tehtemärk) leidmisel võta pinust kaks operandi, tee operaatorile vastav tehe ja pane tulemus pinusse.

Aritmeetikaavaldise töötlemisel tekib probleem jagamisel ja lahutamisel. Lahutamise saab lahendada nii, et miinust loetakse unaarseks operaatoriks ja käsitletakse koos arvuga. Lahutamine pannakse kirja `a+(-b)`, ehk postfiksesituses: `a-b+` (kus miinus kuulub b-ga kokku). Jagamisel tuleb esimesena pinust välja tulev arv panna jagajaks.

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 indeksi abil.
2. **Dünaamiline meetod** kasutades **ühe viidaga ahelloendit**. Uue väärtuse lisamisel pinusse lisatakse ahelloendisse uus element ja väärtuse kustutamisel kustutatakse element ahelloendist.

Teostus massiiviga

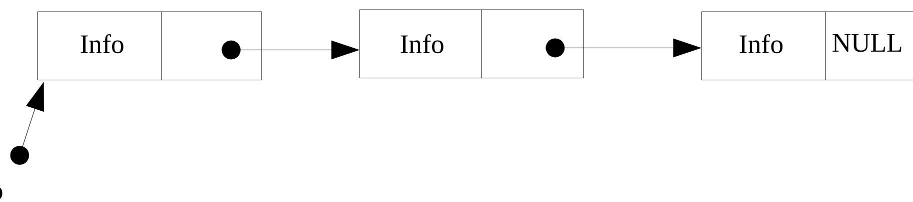
Iga massiivi element sisaldab infot pinu ühe elemendi kohta. Pinu tipuks on indeks, mis määrab, kui kaugele massiivi algusest täidetud pinu ulatub. Pinu on tühi, kui tipu indeks võrdumine nulliga. Kui uus element lisatakse pinusse, salvestatakse see massiivi pinu tipu indeksiga määratud lahtrisse ja seejärel suurendatakse piku väärtut ühe võrra. Elemendi kustutamiseks tuleb tippu nihutada ühe koha võrra ettepoole ja seejärel elemendi väärtus tipulahtrist välja võtta. Tipu väärtuse järgi saab ka jälgida, et pinu ei täituks üle. Kirjeldatud variandis on pinu tipp alati seotud esimese vaba kohaga.

Loomulikult on võimalikud väikesed muutused – näiteks tühja pinu tähistab indeks -1. Pinu tipu indeks on alati pinusse viimati lisatud elemendi indeks. Ehkki tehniliselt on kõik massiivi lahtrid suvalisel hetkel indeksite kaudu ligipääsetavad, tuleb seada nö loogiline tõke ja töötada pinuga ainult reeglitele alludes.

Usun, et ka üsna vähese "massiiviteadlikkusega" on sellise pinu programmeerimine jõukohane.

Teostus ahelloendiga

Dünaamilise realisatsiooni korral ei pea elemendid paiknema füüsiliselt järjestikku, vaid järgnevusseose määravad viidad. Ahelloendi esimese elemendi mäluaadress sobib ka pinu tipuks. Pinu põhioperatsioone (elemendi lisamine ja kustutamine) tehakse eelistatult ahelloendi alguses (vt Joonis 1). Mõtle välja, miks lisamine ja kustutamine ahelloendi lõpus on ebamugavam tegevus. Lisamine algusesse ja kustutamine algusest on juba läbi tehtud operatsioonid.



Joonis 1: Pinu ühe viidaga ahelloenina.

Järjekord e. saba

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

IT terministandardi sõnastik ütleb: **Järjekord** on loend, mille ehitusviis ja hooldus on sellised, et järgmisena võetakse kõige varasemana salvestatud andmeelement. Seda meetodit iseloomustatakse väljendiga „esimesena sisse, esimesena välja„ (FIFO-printsiip).

Järjekord on igapäevases elus üsna tavaline "andmestruktuur". Näiteks eksisteerib järjekord poes kassas, arsti ukse taga või TLÜ kohvikus.

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 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)*.

Järjekorra operatsioonid

Järjekorra põhioperatsioonid on analoogilised pinu operatsioonidega. Võib leida järjekorra kirjeldusi, kus lisatud näiteks järjekorras olevate elementide arvu jälgimine 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:

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 element lisatakse ja kust kustutatakse. 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 (ajajaotussüsteemid). Paljud tegeliku elu probleemid taanduvad järjekorra kasutamisele, näiteks lennukipiletite broneerimine. 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 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 pinuga:

1. **Staatiline** meetod kasutades **massiivi**.
2. **Dünaamiline** meetod kasutades **ahelloenditt**.

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

Teostus massiiviga

Järjekord massiivis eeldab kahe indeksit (`head` ja `tail`) meeles pidamist. Andmeid lisatakse indeksi `tail` järgi ning kustutatakse indeksi `head` järgi. Pinuga sarnaselt muudetakse elemendi lisamisel ja kustutamisel vastavaid indekseid. Massiivi lõpu täitumisel tuleb jätkata elementide lisamist massiivi algusesse, kus loodetavasti selleks ajaks järjekorras olnud elemendid juba kustutatud on. Indekseid omavahel võrreldes on võimalik aru saada, kas järjekord on tühi või on ta hoopiski täis saanud.

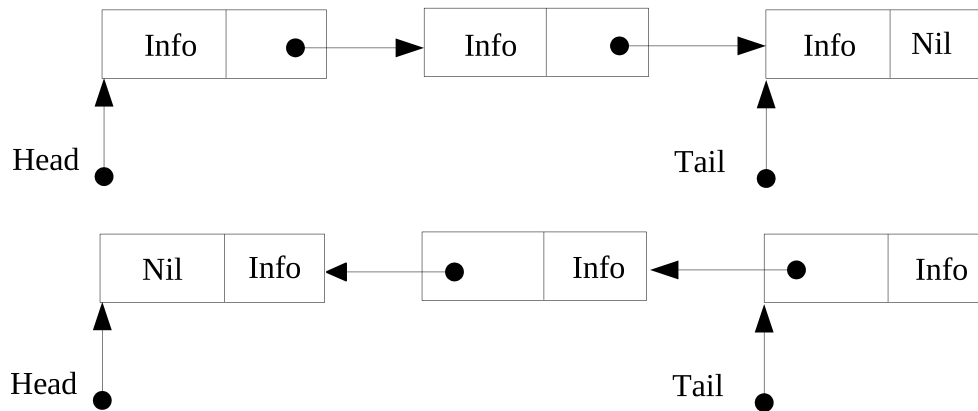
Teostus ahelloendiga

Enne ahelloendi abil realiseerimist tuleb teha mõned põhimõttelised otsused:

- Mitu välist 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 järjekorra mõlemale otsale.

- Mis pidi panna jooksma viidad ehk kas `head` on seotud ahelloendi esimese või viimase elemendiga?



Joonis 2: Järjekord, kaks varianti.

Selleks tuleks joonistada ühe viidaga loend ja proovida, kuidas lisamine ja eemaldamine mugavamalt õnnestuvad (vt joonis 2).

Arvestame, et lisamine toimub järjekorra lõppu (`tail`) ja kustutamine ehk teenindamine järjekorra algusest (`head`). Sel juhul on mugavam kasutada esimest joonisel 2 olevat varianti. Et selles veenduda tuleb läbi mõelda elemendi lisamine ja kustutamine ahelloendi mõlemas otsas.

Dekk e. kaheotsaline järjekord

Dekk *(deque, double-ended queue)* on loend, 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 deki 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:

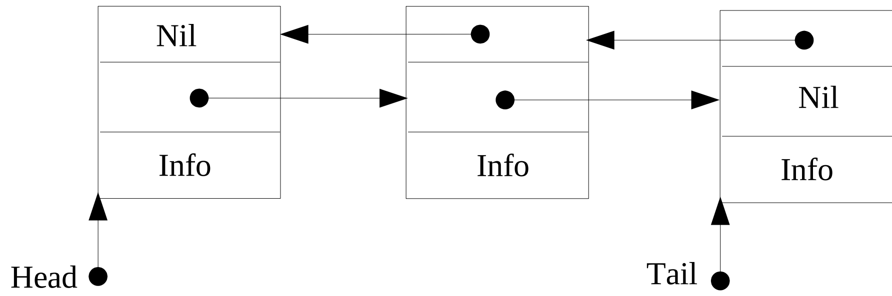
1. Lisada element deki algusesse.
2. Lisada element deki lõppu.
3. Eemaldada element deki algusest
4. Eemaldada element deki lõpust.

Lisaks:

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

Realisatsioon

Võimalik on kasutada nii massiivi kui ahelloendit, kuid massiivi kasutamise muudab ebamugavaks vajadus lisada mõlemasse otsa. Seetõttu on ahelloendi kasutamine loomulikum. Kuid juba järjekorra realiseerimisel sai selgeks, et ühe viidaga ahelloend hästi ei sobi viitade ühtpidi suunatuse tõttu. Kõigi operatsioonide mugavamaks korraldamiseks tuleb kasutusele võtta kahe viidaga ahelloend. Vaata joonis 3.

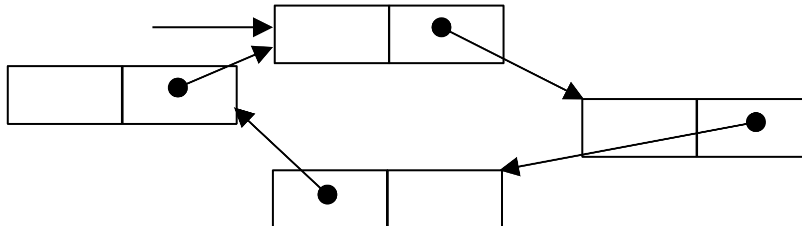


Joonis 3: Dekk kahe viidaga ahelloendina.

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.

Realiseerimiseks piisab ühe viidaga ahelloendist, kus viimase elemendi viidaväljas on `null`-viida asemel esimese elemendi aadress. Vajalik on üks väline viit mingile elemendile.

Realistasioon programmeerimiskeeles C

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 avaldise postfiksesitusse teisendamise 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 top;

// Pinu loomine, mälus reserveeritakse soovitud arv mälupesi massiivi jaoks
void init(int size) {
    stack = malloc(size * sizeof int); top = 0;
}
// Tagastab true või false vastavalt sellele, kas pinu on tühi või mitte
int isEmpty() {
    return top == 0; }

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

// Tagastab pinust viimase väärtuse
char pop() { 7
    top--; return stack[top];
}
```

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

1. Omades sama liidesega, kuid ahelloendit 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 loendite realiseerimist massiivi ja ahelloendiga

1. Ahelloendis tuleb igas elemendis eraldada lisaks üks või kaks välja, mis mälu võtab. Kui aga ahelloendeid 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. Massiivis tuleb aga lõpu pool 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. Ahelloendi korral on kergem organiseerida loendite ühendamist ja lõhkumist mitmesse ossa, mis hiljem iseseisvalt kasvada/kahaneda saavad.
6. Viitadega realisatsiooni korral saab luua palju keerulisemaid struktuure: näiteks linkides iga ahelloendi elemendiga terve uue ahela.
7. Loendi läbimine toimub tavaliselt kiiremini järjestikulise paiknemise (massiivi) korral.