15
Arrays og Lister samt Stakke og Køer.

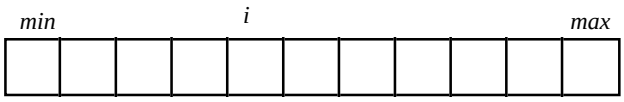
- Introduktion til arrays.
- Algebraisk specifikation af arrays.
- Arrays i Eiffel.
- Introduktion til lister og kædede lister.
- Simpel algebraisk specifikation af lister.
- Dobbeltkædede og cirkulære lister.
- Fælles egenskaber ved stakke og køer.
- Algebraisk specifikation af stakke.
- Algebraisk specifikation af køer.
- Prioritetskøer.

PS1 -- Arrays, lister, stakke og køer

Noter

Dette er den første af tre forelæsninger om elementære men centrale abstrakte datatyper. Foruden arrays, lister, stakke og køer (som er emnet i dette kapitel) vil vi i dette kursus, og senere i disse noter, se nærmere på strenge, filer, træer og hobe.

Introduktion til arrays.



- Et array er en abstrakt datatype som repræsenterer en homogen tabel.
- Faste øvre og nedre grænser, og dermed fast størrelse.
- Vigtige array-operationer:
  - **hent** i-te element.
  - **gem** nyt objekt i i-te element.
- Hent og gem operationerne har konstant tidskompleksitet.
- Underliggende lagres elementerne i et array konsekutivt.

Noter

Denne slide opremser traditionelle egenskaber af arrays.

Et array kan implementeres som en konsekutiv mængde af lagerceller, hvor positionen af et element i array'et kan beregnes ud fra indekset. Det befordrer den konstante tidskompleksitet af de to fundamentale operationer: hent og gem på arrays. Med andre ord: hent og gem operationerne beregner positionen af et element i arrayet, og henter/gemmer dette element direkte, uden nogen som helst form for søgning eller gennemløb af arrayet.

På dette kursus vil vi konsekvent opfatte arrays som objekter af en abstrakt datatype. På denne slide ser vi dels på den traditionelle repræsentation af denne datatype, dels på vigtige egenskaber af typens operationer.

Homogenitetsegenskaben af arrays er relativ til programmeringssprogets regler for typesammenlignelighed. Homogenitetsegenskaben kan også ses som en kontrast til records, hvori felterne jo kan være af forskellige typer.

Kravet om fast størrelse af arrays kan forstås på to måder:

1. Det er statisk bestemmeligt, hvor stort et array skal være. Dvs. at inden programmet kører kan man bestemme, hvor grænserne for programmets arrays.

2. Det er dynamisk bestemmeligt, hvor stort et array skal være. Størrelsen kan f.eks. afhænge af bruger-input. Men når et array først er allokeret, kan det ikke ændre størrelse.

Det er selvfølgelig fristende at hæve disse restriktioner, og i visse sprog er det da også muligt at have arrays, som kan ændre størrelse under programmets udførelse. I Eiffel klassen ARRAY kan man anvende en operation *resize*, som ændrer størrelsen af array-et.

Algebraisk specifikation af arrays.

Type Array[X]
Functions
create: Integer x Integer -> Array[X]
put: X x Integer x Array[X] -> Array[X]
item : Integer x Array [X] -> X
count: Array [X] -> Integer
lower: Array [X] -> Integer
upper: Array[X] -> Integer

Axioms
for all x in X, l, u, i, j in Integer, A in Array [X]
item(i, create(l,u)) =undefined
item (i, put (x, j, A)) = if i = j
then x
else item (i, A)

count(create(l,u)) = if u • l then u - l + 1else undefined
count(put (x, j, A)) = count(A)
lower(create(l,u)) = l
lower(put (x, j, A)) = lower(A)
upper(create(l,u)) = u
upper(put (x, j, A)) = upper(A)

PS1 -- Arrays, lister, stakke og køer

Noter

Det er ikke helt oplagt, hvordan man håndterer indekstypen i en algebraisk specifikation af et array.

Man kan vælge at angive to generiske typer til array specifikationen: indekstype og elementtype.

Vi har dog i denne specifikation valgt at lade grænserne for arrayet være parameter til "basis konstruktoren" create. Da der jo ikke er ligninger for konstruktorerne, kan vi ikke direkte angive, at den nedre grænse skal være mindre end den øvre grænse. Dette må afspejles i specifikationen af andre operationer, eller, om muligt, i en prebetingelse til Create. Vi har ikke indbygget sådanne fejlchecks konsekvent i ovenstående specifikation. Vi er interesseret i at angive de vigtige egenskaber ved arrays, *axiomerne*. De sekundære, og mere pragmatisk betonede egenskaber har vi stort set udeladt, da de blot forvirrer billedet.

En af de vigtige ting at fæstne sig ved i denne specifikation, er hvilken værdi item(i, A) returnerer, hvis vi har tilskrevet elment nummer i værdien a1, og dernæst tilskrevet det samme element værdien a2. Specifikationen af item operationen angiver, at den sidst tilskrevne værdi til det i-te element (a2) overskriver den første. Overvej dette!

Bemærk, at denne specifikation specificerer et array, men operationelt er den "forfærdelig" fra et kompleksitetsmæssigt synspunkt. Item operationen, som tager et element ud af array-et er foreskrevet som et rekursivt gennemløb af strukturen. Dette skal på ingen måde tages som et forbillede for en implementation, i hvilken vi jo kræver, at arrayopslag skal have konstant tidskompleksitet.

Arrays i Eiffel.

I Eiffel er array typen defineret af en generisk klasse.

Eiffel

```
class ARRAY[T]
export lower, count, upper, item, put, ...

  make(minindex, maxindex: INTEGER) is
  do ... end

  item(i: INTEGER): T is
  do ... end
  ...
  end
  ...
end; --class ARRAY
```

Pascal

```
(* intet modstykke *)
(* sprogdefineret *)
```

```
a: ARRAY[BOOLEAN];          a: array[1..n] of boolean;
...
!!a.make(1, n);
a.put(i,  false);           a[i] := false;
x := a.item(i)              x := a[i];
```

PS1 -- Arrays, lister, stakke og køer

Noter

I forhold til bogen OOSC er der i vores implementation af Eiffel indført andre navne på operationerne på arrays. Denne slide bruger de nye og gældende navne. Lad være med at programmere med ARRAYs efter OOSC. "Eiffel the Language" kapitel 28 beskriver Array's i Eiffel3.

Der er flere operationer i klassen ARRAY end dem, vi har specificeret på forrige slide. Se kap. 28.

Det skal bemærkes at der findes et infix alias for operationen item. I stedet for at skrive

```
a.item(i)
```

som ovenfor kan man i Eiffel3 skrive

```
a @ i
```

Det er generelt muligt at definere infix (og prefix) notation for operationer med to parametre (én parameter), incl. modtager objektet. Se detaljer i 25.10 i "Eiffel the Language".

Vær også opmærksom på klassen ARRAY2---to-dimensionelle arrays, som er i datastrukturbiblioteke.

Manifeste arrays i Eiffel.

I Eiffel er der en manifest notation for arrays, hvis nedre grænse er 1.

- << e_1, e_2, ... e_n >> er et udtryk, hvis værdi er af typen ARRAY[T], forudsat at alle deludtryk e_1, e_2, ... e_n er af typer, som er typesammenlignelige med T.
- << e_1, e_2, ... e_n >> både instantierer og initialiserer et array objekt.

Skabelse af et array ved brug af manifest array-udtryk.

```
A: ARRAY[INTEGER]
...
A := << 8, 9, 10 >>
```

Tilsvarende skabelse af et array uden brug af manifest array-udtryk.

```
A: ARRAY[INTEGER]
...
A.make(1, 3);
A.put (8, 1);
A.put(9, 2);
A.put(10, 3)
```


Noter

Man kan i "Eiffel the Language" læse om manifeste arrays i afsnit 23.20.

Introduktion til lister.

$(e_1\ e_2\ e_3\ ...\ e_n)$

- En liste er en abstrakt datatype som tillader os at registrere en bestemt *rækkefølge* af et antal *homogene elementer*.

- Vigtige liste-operationer:
  - **–Indsættelse** af nyt element i listen.
  - **–Sletning** af eksisterende element fra listen.
  - **–Navigering** til et naboelement i listen.
  - **–Aflæsning** af et nærmere angivet element i listen.

- Der er mange mulige *repræsentationer* af lister:
  - baseret på arrays.
  - baseret på sammenkædede strukturer.
  - andre...

- Repræsentationen påvirker
  - operationsrepertoire.
  - tidskompleksitet.
  - andre basale egenskaber (f.eks. persistens).

Noter

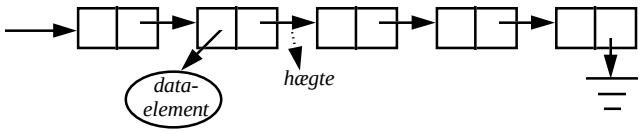
På sammen måde som vi opfatter et array som en abstrakt datatype, vil vi også opfatte lister som en abstrakt datatype. Lister er på flere måder et mere abstrakt begreb end arrays. Det basale i listebegrebet er *rækkefølgen mellem et antal elementer*. Dette træk genfinder vi også fra array-typen, idet der jo er en ganske bestemt rækkefølge af tabellens pladser.

Vi har forsøgt at fastslå de vigtige og karakteristiske operationer på lister. Indsættelse og sletning tillader os manipulere rækkefølgen af elementer. Hvis vi sammenligner med array-typen kan vi notere os, at der er fundamentale forskelle, idet rækkefølgen af array-elementer i bund og grund er fastlagt på det tidspunkt, array'et skabes. (At vi kan simulere ændringer af rækkefølgen af array-elementer ved at skubbe og ombytte *indholdet* er tabellens elementer er en lidt anden sag, som vi ser bort fra her). Navigering til naboelementer er også fundamentalt for lister. På dette niveau af diskussionen vil vi ikke fastlægge nøjagtigt hvilken navigering der er tale om, og ej heller hvordan resultatet af navigeringen manifesterer sig. Men i næsten alle lister vil vi kunne navigere til *næste elemente* i listen. (I mange liste typer kan vi også komme afsted med at navigere til forrige element i listen, til første element, til sidste element og til *i-te* element). Endelig må vi naturligvis have en eller anden måde at aflæse elementer fra listen. Aflæsning hænger naturligvis ofte sammen med navigering, således at vi kan aflæse det element, hvortil vi har bevæget os.

Lister kan repræsenteres på mange forskellige måder; hver repræsentation giver operationerne bestemte egenskaber hvad angår tidskompleksitet. Der findes to meget anvendte måder at repræsentere lister på: (1) som sammenkædede strukturer og (2) ved arrays. Dette vender vi tilbage til på næste side.

Kædede lister.

Enkeltkædet liste



- Sammenkædet (hægtet) struktur af homogene dataelementer.
- Ingen grænser på strukturen.
- Indsættelse og sletning af elementer: operationer med køretid pÅ O(n), hvor n er længden af listen.
- Navigering kan kun foregå ved at følge hægterne mellem listeelementer.

En liste er mere end samlingen og rækkefølgen af elementer.

Det er "farligt" at referere til en liste gennem en pointer til dets første element.

En liste er *et objekt, som har selvstændig identitet.*

Noter

På denne slide studerer vi lister repræsenteret som (enkelt) kædede strukturer, og vi sammenligner med en array-baseret løsning.

Den "udefra kommende" reference til listen øverst til venstre på denne slide peger på det første element i listen. (Se dog diskussionen af denne reference herunder).

I et array er det ikke muligt direkte at indsætte et element mellem to eksisterende elementer. Vi kan, f.eks., ikke indsætte et element mellem a[2] og a[3] i array-et a ved at sige a[2.5] := e. Det nærmeste, vi kan komme dette, er at skubbe alle elementerne mod højre, i eksemplet alle elementer fra a[3] og fremad, således der bliver plads til det nye element. Derved ændres elementernes indeks. Denne operation er endvidere kostbar, i størrelsesordenen O(længden af array-et).

Kædede lister er velegnede hvad angår indsættelse og sletning af elementer imellem (eller før eller efter) eksisterende elementer. Indsættelse består i at skabe en ny "kasse" og hægte den ind i strukturen. Sletning består i at afskaffe en eksiterende kasse, og afhægte den fra strukturen. Det er naturligvis vigtigt at man behersker de enkelte trin i disse to handlinger.

Det er vigtigt at få fat i, hvor tidsforbruget for indsættelse og sletning af elementer i en kædet lister stammer fra. Hvis vi ønsker at slette det i-te element i en kædet liste må vi opsøge dette element. Hvis "opsøgningen" skal foretages ved at gennemrejse alle de forudgående elementer, koster det i gennemsnit n/2 trin.

Selve indsættelses- og slette operationerne af et led i kæden er af konstant kompleksitet, *forudsat* at man også har en reference til det forrige liste element. I praktiske implementationer er det således attraktivt at introducere en listeposition, som kan flyttes af bestemte operatione. Eiffel gør et stort nummer ud af dette.

Hvis man refererer til en liste gennem en pointer til dets første element kommer man i alvorlige vanskeligheder hvis man vil indsætte eller slette det første element i listen. Derimod er det ikke et problem at indsætte/slette de øvrige elementer. Det er klart, at dette er noget rod! En løsning (lappeløsning) er at introducere et listehovede eller et dummy element (se Weiss afsnit 3.2.3). Den pæne løsning består i at indse, at en liste som så er et objekt med identitet (udgjort af mere end blot dets elementer) og med egen tilstand. En del af tilstanden er naturligvis referencen til listens første element.

Algebraisk specifikation af lister.

Type List[X]
Functions
empty-list: -> List[X]
insert: X x List[X] -> List[X]
delete: List[X] -> List[X]
item : Integer x List [X] -> X
count: List [X] -> Integer
append: List [X] x List [X] -> List [X]

Axioms
for all x in X, L, in List[X], i in Integer
item(i, empty-list ()) = error
item (i, insert (x, L)) = if i = 1
then x
else item (i-1, L)
delete(empty-list ()) = error
delete(insert (x, L)) = L
count(empty-list ()) = 0
count(insert(x, L)) = 1 + count(L)

PS1 -- Arrays, lister, stakke og køer

Noter

Vi indskrænker os til at specificere lister i det specialtilfælde, hvor elementer altid indsættes og slettes i begyndelsen af listen. Bemærk, at vi i realiteten derved kommer til at specificere noget, der ligner en stak. Vi kan dog gennem operationen item bede om det i-te element i listen.

Dette er mere realistisk, end det måske lyder. Lisp, som er et listehåndteringssprog, arbejder med lister på denne måde. Empty-list svarer da til nil i Lisp, insert til cons, delete til cdr, item til nth (med lidt anden nummereringskonvention af elementer i listen). Vi har ikke i denne specifikation et eksplicit modstykke til car fra Lisp. Den kan fåes ved item(1, ...).

I Lisp er det dog meget forkert at tænke på cdr som delete. (Cdr l) returnerer halen af listen l, men l er stadigvæk en reference til hele listen l.

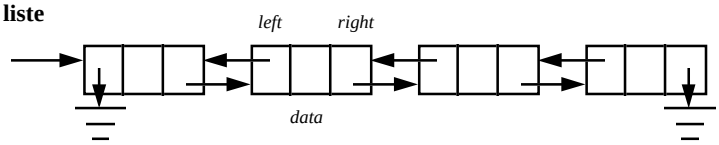
Når vi har brug for en manifest notation for lister vil vi bruge Lisp notation. Ifølge denne er (e1 e2 ... en) listen bestående af elementerne e1, e2, ..., en.

Opgave 1. I specifikationen på denne slide er der ikke vist ligninger for append. Append skal sammensætte to lister, således at elementerne fra den første liste kommer forud for elementerne i den anden liste. Skriv ligningerne for append.

Opgave 2. Lav en algebraisk specifikation af lister, hvor elementer kan indsættes og slettes fra vilkårlige positioner. Create laver en tom liste, insert indsætter et element, således at det bliver det i-te element i listen, delete sletter det i-te element, og item udtager det i-te element. Count betegner antallet af elementer i listen.

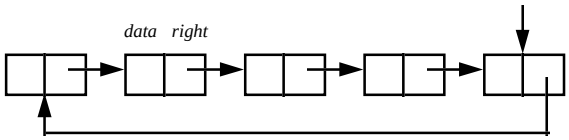
Dobbeltkædede lister og cirkulære lister.

Dobbeltkædet liste



Effektivt at navigere både mod højre og venstre.

Cirkulær liste



Effektivt at indsætte først såvel som sidst i listen.

PS1 -- Arrays, lister, stakke og køer

Noter

I en cirkulær liste peger vores "udefra kommende reference" på det sidste element. Idet listens sidste element peger på det første element, er det effektivt at indsætte elementer både først og sidst i listen.

I Eiffel er den mest benyttede kædede liste LINKED_LIST, som er enkeltkædet. Som et alternativ har vi på dette kursus adgang til en lidt simplere klasse (og knap så "rig" klasse for enkeltkædede lister) kaldet SIMPLE_LIST. Denne klasse introduceres i et appendex til nærværende del af noterne.

Gennemløb af lister.

Forskellige former for gennemløb af lister:

- udførelse af en handling på alle elementer i en liste.
- filtrering af alle elementerne i en liste, som opfylder en betingelse.
- reduktion af en liste til ét element, som afspejler en beregning på listens elementer.

Det er attraktivt at være i stand til at programmere generelle abstraktioner, som afspejler forskellige former for gennemløb af lister.

Noter

Gennemløb af lister er så hyppige, at det kan betale sig at programmere dem een gang for alle, og derefter parametrisere gennemløbene med passende handlinger.

Denne slide postulerer, at der er forskellige former for gennemløb, som vi kan vælge at programmere hver for sig.

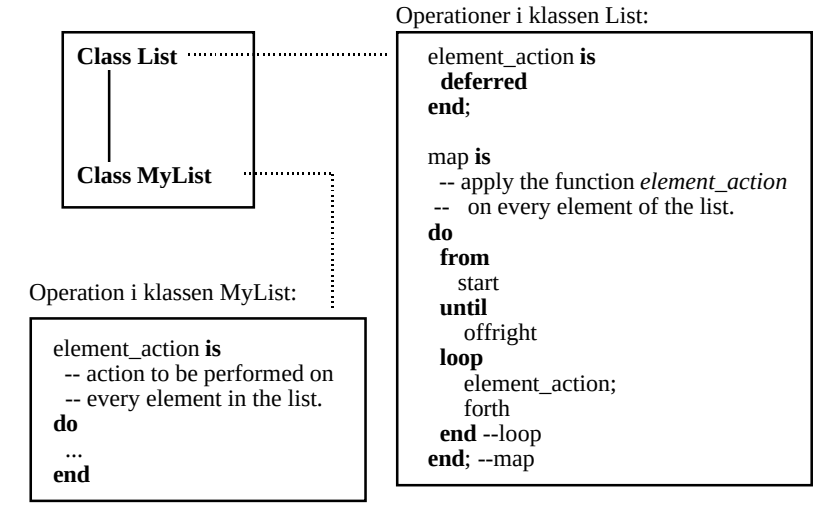
I Lisp (hvor lister, som omtalt, spiller en helt afgørende rolle) findes der (blandt andre) funktioner, der afspejler de tre nævnte former for listegemmenløb. Disse funktioner hedder traditionelt map, filter og reduce. Disse funktioner kaldes højere ordens funktioner, fordi de tager en funktion som parameter, der udtrykker handlingen på hvert enkelt liste element.

Eksempel: Vi ønsker at udskrive de lige tal i listen L af tal. Dette kan let og elegant gøres med

```
(map print (filter even L))
```

hvor even er en funktion (et prædikat), der returnerer, hvorvidt et tal er lige.

Gennemløb af lister i Eiffel.



PS1 -- Arrays, lister, stakke og køer

Noter

Denne slide illustrerer, hvordan tilsvarende gennemløb af lister kan finde sted i Eiffel.

Vi tager udgangspunkt i en klasse List, som i princippet kunne være (men ikke er) linked_list. I List kan vi programmere en procedure, som gennemløber listen, og som på hvert element kalder en anden funktion, element_action.

Element_action kan enten være forudannonceret (deferred) eller blot tom. (Sidstnævnte er sikkert at foretrække i praksis, idet det herved bliver muligt at instantiere klassen List).

Når man vil programmere et bestemt gennemløb, laver man i en subklasse, hvori proceduren element_action er redefineret.

Dette kan forekomme som en tung fremgangsmåde, i det mindste i forhold til Lisp løsningen, som vi omtalte i tilknytning til forrige slide. Hvis vi ønsker at anvende to forskelligt parametriserede gennemløb kommer vi i problemer.

Eiffel håndterer ikke iteration over liste på helt sammen måde som skitseret. Men Eiffel teknikken er tæt beslægtet med den viste fremgangsmåde. Eiffel benytter en særlig klasse, linear_iter, til at beskrive iteration. Linear_iter kan arves (sammen med andre klasser i multipel nedarvning) hvis man ønsker at iterere over lister.

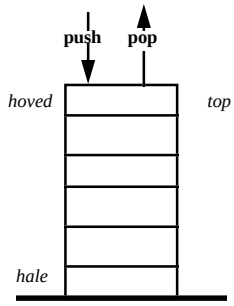
Interesserede læsere kan konsultere Eiffel biblioteket og tilknyttede eksempler på dette.

Fælles egenskaber af stakke og køer.

- I stakke og køer er højst ét element i strukturen tilgængelig for aflæsning og udtræk.
- Stakke og køer er datatyper, hvor aflæsning og modifikation finder sted i strukturernes yderpunkter.
- Stakke og køer kan opfattes som specialiseringer af enten arrays eller kædede lister.

Noter

Introduktion til stakke.



En stak er en liste, hvor elementer indsættes, aflæses og slettes i hovedet af listen.

En stak følger en LIFO (Last In First Out) disciplin.

Traditionelle operationer:

- Indsættelse af element i stak: push
- Sletning af element fra stak: pop
- Aflæsning af element fra stak: top

Noter

Vi har adskillige gange tidligere på kurset anvendt datatypen stak som illustration af forskellige aspekter ved specifikation af ADT-er og objekt-orienteret programmering.

Af en eller anden grund er stakken mange forfatteres foretrukne eksempel i en bred vifte af sammenhænge.

I denne forelæsning vil vi derfor kun kort opsummere stakke's egenskaber og algebraiske specifikation, først og fremmest som en kontrast til køer.

Tidskompleksiteten af de omtalte operationer er konstant.

Algebraisk specifikation af stakke.

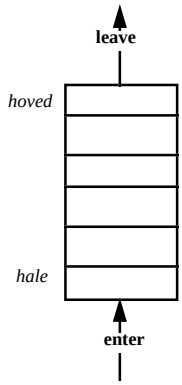
Type Stack[X]
Functions
create: -> Stack[X]
push: X x Stack[X] -> Stack[X]
pop: Stack[X] -> Stack[X]
top: Stack[X] -> X
empty: Stack[X] -> Boolean

Axioms
for all x in X, S in Stack[X]
pop(create()) = Undefined
pop(push (x, S)) = S
top(create()) = Undefined
top(push (x, S)) = x
empty(create()) = true
empty(push (x, S)) = false

Noter

Opgave. Vi har tidligere lavet en simpel specifikation af lister (se slide med "algebraisk specifikation af lister"). Sammenlign denne specifikation af lister med specifikationen af stakke, som givet ovenfor. Hvad er forskelle og ligheder?

Introduktion til køer.



En kø er en liste, hvor elementer indsættes i halen og udtages i hovedet af listen.

En sådan kø følger en FIFO (First In First Out) disciplin.

Traditionelle operationer:

- Indsættelse af element i køen: enter.
- Udtagning af element fra køen: leave.
- Aflæsning (uden udtagning) af element fra køen: front.

Noter

Man kunne naturligvis også vedtage, at elementer indsættes i køens hoved og "trækkes ud af halen". Det vigtige er, at man følger en fast konvention.

Hvis vi med lister mener en bestemt klasse LIST, er det måske ikke helt ligegyldigt hvilken konvention vi følger. Vi ønsker naturligvis at arrangere tingene sådan, at operationerne bliver hurtigst mulige, med så lille pladsbehov som mulig, og at vi som programmører slipper lettest muligt om ved programmeringen af køen givet LIST.

Der findes alternative navne på de karakteristiske kø-operationer. I Weiss kaldes enter for *enqueue* og leave for *dequeue.*

Algebraisk specifikation af køer.

Type Queue[X]
Functions
create: -> Queue [X]
enter: X x Queue [X] -> Queue [X]
leave: Queue [X] -> Queue [X]
front: Queue[X] -> X
size: Queue[X] -> Integer
isempty: Queue[X] -> boolean

Axioms
for all x in X, Q in Queue[X]
leave(create()) = Undefined
leave(enter (x, Q)) = if isempty(Q) then create()
else enter(x, leave (Q))
front(create()) = Undefined
front(enter (x, Q)) = if isempty(Q) then x
else front(Q)
size(create()) = 0
size(enter (x, Q)) = 1 + size(Q)
isempty(create()) = true
isempty(enter (x, Q)) = false

PS1 -- Arrays, lister, stakke og køer

Noter

Konstruktorerne i denne algebraiske specifikation er create og enter.

Forskellen i indsættelses- og slettedisciplin mellem stakke og køer fremgår tydeligt, hvis man sammenligner ligninger for leave i denne specifikation med ligningerne for pop i stack.

Prioritetskøer.

En prioritetskø er en datatype, hvor hvert kø-element har tilknyttet en prioritet.

Karakteristiske operationer:

- *Enter(el, pr)*
  Indsæt elementet *el* med prioritet *pr*.
- *Leave*
  Udtag elementet med den højeste prioritet.

To naive implementationsstrategier:

1. *Enter* ordner elementerne efter prioritet. O(n).
   Leave udtager element. O(1)
2. *Leave* søger efter element med max prioritet. O(n).
   Enter indsætter element. O(1)

Der findes en meget bedre implementationsstrategi, som kaldes en hob.

Noter

Vi vender tilbage til prioritetskøer og hobe senere i kurset, hvor vi vil bruge et helt kapitel på emnet.