Föreläsning 2

Stackar, köer och listor

TDDC91,TDDE22,725G97: DALG

Utskriftsversion av föreläsning i *Datastrukturer och algoritmer* 6 september 2018

Magnus Nielsen, IDA, Linköpings universitet

x

2.1

Introduktion

- Sekvenser av data förekommer i många applikationer;
  - Hur representera dem i minnet?
  - Vad är typiska och specifika operationer? (*definiera ADTer*)
  - Hur implementera dem?

2.2

Innehåll

2.3

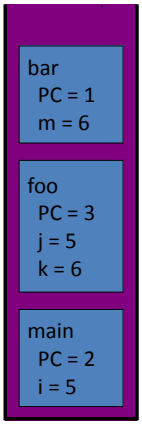
[1] eller ett felmeddelande om S är tom

2.5

Användning av stack: funktionsanrop i JVM

- Javatolken (Java virtual machine (JVM)) håller reda på en anropskedja med hjälp av en stack.
- När en metod anropas *push*:ar JVM argumenten, lokala variabler, returvärde och programräknaren (dessa bildar funktionens *aktiveringspost*).
- När metoden körts klart *pop*:as aktiveringsposten och kontrollen lämnas över till metoden överst på stacken.
- Detta möjliggör rekursion.

```
main() {
   int i = 5;
   foo(i);
   }
foo(int j) {
   int k;
   k = j+1;
   bar(k);
   }
bar(int m) {
   …
   }
```



2.6

Svansrekursion

Ett rekursivt anrop är *svansrekursivt* omm första instruktionen efter att kontrollflödet kommit tillbaka efter anropet är **return**.

- stacken behövs inte: allting på stacken kan kastas direkt
- svansrekursiva funktioner kan skrivas om till iterativa funktioner

Det rekursiva anropet i FACT är *inte* svansrekursivt:

function FACT(n)
  if $n = 0$ **then return** 1
  else return $n \cdot$FACT($n-1$)

Första instruktionen efter retur från det rekursiva anropet är *multiplikation* $\Rightarrow n$ *måste behållas på stacken*

2.7

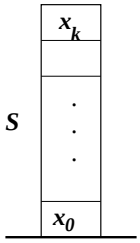
En svansrekursiv funktion

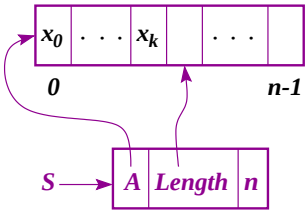
function BINSEARCH($v[a,\ldots,b],x$)
  if $a < b$ **then**
    $m \leftarrow \lfloor \frac{a+b}{2} \rfloor$
    if $v[m].key < x$ **then**
      return BINSEARCH($v[m+1,\ldots,b],x$)
    else return BINSEARCH($v[a,\ldots,m],x$)
  if $v[a].key = x$ **then return** a
  else return 'not found'

De två rekursiva anropen är *svansrekursiva*. 2.8

1.2 Representation i angränsande minne

Implementation av stack i angränsande minne





Length(S) = k + 1 <= n

function MAKEEMPTYSTACK
  $S \leftarrow$ NEW($StackTableHead$)
  $S.A \leftarrow$ NEW(**table**$[0,\ldots,n-1]$)
  $S.Length \leftarrow 0$
  return S

function PUSH(S,x)
  if $S.length = n$ **then error**
  else
    $S.A[S.Length] \leftarrow x$
    $S.Length \leftarrow S.Length + 1$

function POP(S)
  if $S.Length = 0$ **then error**
  else
    $S.Length \leftarrow S.Length - 1$
    return S.A[S.Length]

2.9

Implementation av stack i angränsande minne

function TOP(S)
  if $S.Length = 0$ **then error**
  else return S.A[S.Length-1]

function ISEMPTYSTACK(S)
  return S.Length = 0

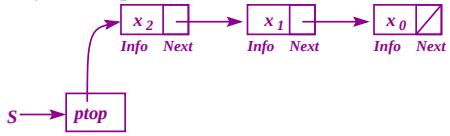
Alla operationer tar $O(1)$ tid. 2.10

1.3 Representation i länkat minne

Implementation av stack i länkat minne

function MAKEEMPTYSTACK
  $S \leftarrow$ NEW($StackListHead$)
  $S.ptop \leftarrow$ **null**
  return S
function PUSH(S,x)
  $y \leftarrow$ NEW($StackListItem$)
  $y.Info \leftarrow x$
  $y.Next \leftarrow S.ptop$
  $S.ptop \leftarrow y$

+ maxstorlek behöver inte vara känd i förväg - anrop till NEW vid varje PUSH-operation och till FREE vid varje POP-operation



function POP(S)
  if $S.ptop =$ **null then error**
  else
    $y \leftarrow S.ptop$
    $x \leftarrow y.Info$
    $S.ptop \leftarrow y.Next$
    FREE(y)
    return x

2.11

2 ADT kö

ADT kö (först in först ut)

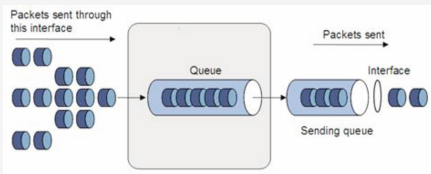
Operationer:

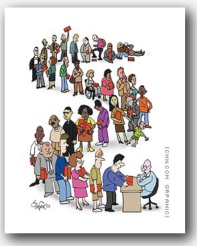
- *Front*(Q) returnerar det första elementet i kön Q
- *Dequeue*(Q) tar bort och returnerar det första elementet i kön Q
- *Enqueue*(Q,x) lägger x sist i kön Q
- *MakeEmptyQueue*() skapar en ny tom kö
- *IsEmptyQueue*(Q) returnerar *true* omm Q är tom

2.12

Typiska användningsområden för ADT kö

- Bekanta applikationer
  - Spellista i Spotify
  - Databuffert (iPod, VOD)
  - Asynkron dataöverföring (fil-I/O, pipes, sockets)
  - Ta hand om förfrågningar till delad resurs (skrivare, processor)
  - Labbar som ska rättas
- Simulering
  - Trafikanalys
  - Väntetider hos en kundtjänst
  - Bestämma hur många kassörer/kassörskor som behövs på en stormarknad

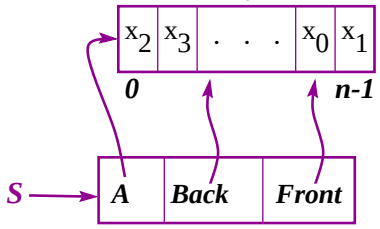




2.13

2.1 Representation i angränsande minne

Implementation av kö i ringbuffer/cirkulär array



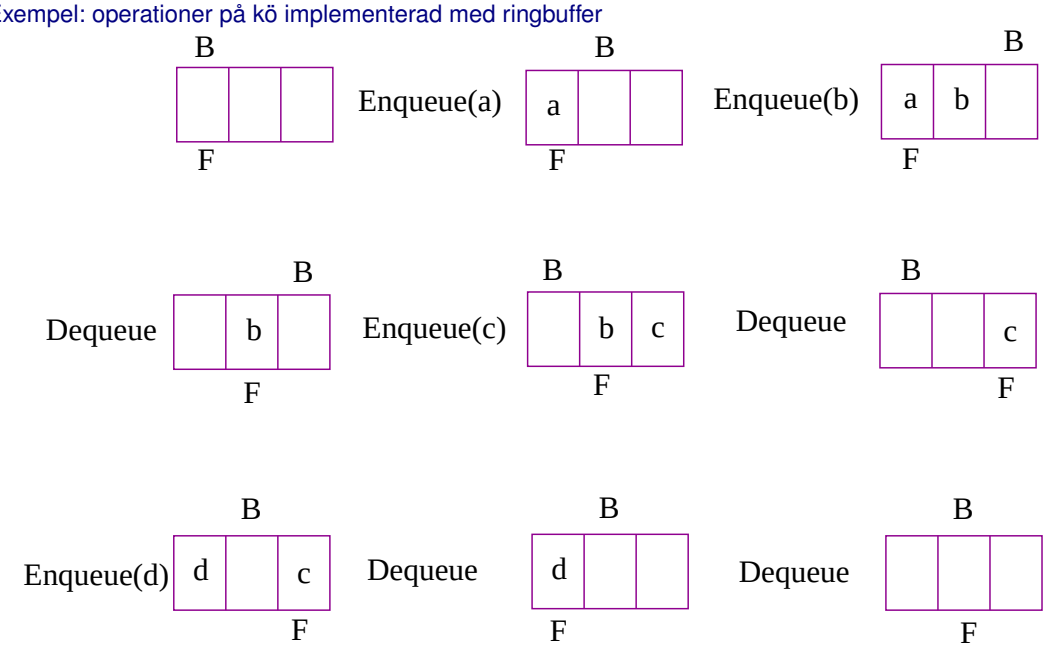
Length = k < n+1

Back = index för första lediga cell *Front* = index för första elementet *Length* eller *Size* för att kontrollera overflow:

$$Size = (n - F + B) \mod n$$

2.14

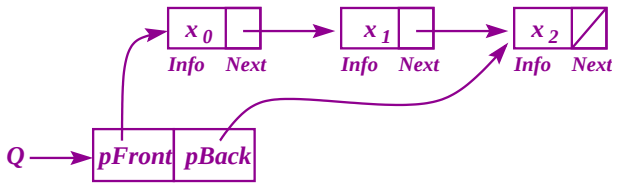
Exempel: operationer på kö implementerad med ringbuffer



2.15

2.2 Representation i länkat minne

Implementation av kö i länkat minne



Javakod för köoperationerna: se kursboken, kapitel 5.2.3

+ maximal storlek behöver inte vara känd i förväg + pekaren *pBack* gör att *EnQueue* går i tid $O(1)$(utan *pBack*: hela listan måste traverseras $\Rightarrow$ behöver $\Theta(k)$ tid) - extra minneshantering i *EnQueue* och *DeQueue*

2.16

3 Listor

Listor

En *lista* L är en sekvens av element $\langle x_0, \ldots, x_{n-1} \rangle$

- *size* eller *length* $|L| = n$

- *tom* lista $\langle\rangle$ med längd 0
- Urval med *index* i (ibland med *rank*): väljer det i:te elementet, x_i, där $0 \leq i \leq n-1$
- Urval med aktuell *position*, t.ex. *första* elementet i L, eller *sista*, *föregående*, *nästa*, ... *position* abstraherar bort från indexering

  → ADT arraylista: använder *index*

  → ADT nodlista: använder *position*

2.17

3.1 ADT arraylista

ADT arraylista

Domän: listor

Operationer på en vektor S

- *size*() returnerar $|S|$
- *isempty*() returnerar *true* omm $|S| = 0$
- *elemAtIndex*(i) returnerar $S[i]$; fel om $i < 0$ eller $i >$ *size*() -1
- *setAtIndex*(i,x) returnera elementet med index i och ersätt det med x som nytt element med index i; fel om $i < 0$ eller $i >$ *size*() -1
- *insertAtIndex*(i,x) sätter in x som nytt element med index i: ökar storleken; fel om $i < 0$ eller $i >$ *size*()
- *removeAtIndex*(i) tar bort och returnerar det i:te elementet i S: minskar storleken; fel om $i < 0$ eller $i >$ *size*() -1

2.18

Exempel: några operationer på en initialt tom arraylista S

operation	utdata	S
insertAtIndex(0,7)	-	(7)
insertAtIndex(0,4)	-	(4,7)
elemAtIndex(1)	7	(4,7)
insertAtIndex(2,2)	-	(4,7,2)
elemAtIndex(3)	"error"	(4,7,2)
remove(1)	7	(4,2)
insertAtIndex(1,5)	-	(4,5,2)
insertAtIndex(1,3)	-	(4,3,5,2)
insertAtIndex(4,9)	-	(4,3,5,2,9)
elemAtIndex(2)	5	(4,3,5,2,9)
setAtIndex(3,8)	2	(4,3,5,8,9)

2.19

3.2 ADT nodlista

ADT nodlista

Domän: listor Operationer på en lista L, förutom *size*() och *isempty*()

- *first*() returnerar *positionen* för första elementet i L; fel om L är tom
- *last*() returnerar *positionen* för sista elementet i L; fel om L är tom
- *prev*(p) returnerar *positionen* för elementet som föregår p i L; fel p är första positionen
- *next*(p) returnerar *positionen* för elementet som följer på p i L; fel om p är sista positionen
- *set*(p,x) ersätt elementet i position p med x, returnera elementet som förut fanns i position p
- *insertFirst*(x) sätt in nytt element x som första elementet i L, returnera positionen för x
- *insertLast*(x) sätt in nytt element x som sista elementet i L, returnera positionen för x
- *insertBefore*(p,x) sätt in nytt element x före position p i L, returnera positionen för x
- *insertAfter*(p,x) sätt in nytt element x efter position p i L, returnera positionen för x
- *remove*(p) ta bort och returnera el. i position p från L

2.20

Exempel: några operationer på en initialt tom nodlista L

operation	utdata	L
insertFirst(8)	-	(8)
first()	p_1(8)	(8)
insertAfter(p_1,5)	-	(8,5)
next(p_1)	p_2(5)	(8,5)
insertBefore(p_2,3)	-	(8,3,5)
prev(p_2)	p_3(3)	(8,3,5)
insertFirst(9)	-	(9,8,3,5)
last()	p_2(5)	(9,8,3,5)
remove(first())	9	(8,3,5)
set(p_3,7)	3	(8,7,5)
insertAfter(first(),2)	-	(8,2,7,5)

2.21

Implementation av ADT arraylista, ADT nodlista

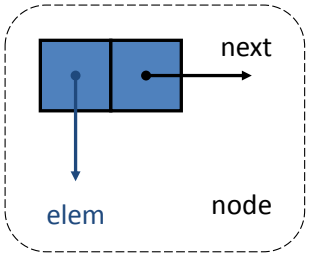
- *Representation i angränsande minne för arraylistor:* elementen lagras i tabell/array (t.ex. den utökningsbara arrayen från föreläsning 2).
  - *elemAtIndex* går i $O(1)$ tid;
  - hur blir det med andra operationer?
- *Enkellänkad lista för nodlistor:* se nästa slide.
  - *positioner* implementerade som pekare.
  - Analysera tidskomplexiteten för operationerna.
  - *prev*, *insertBefore* kräver listtraversering.
- *Dubbelllänkad lista för nodlistor:* se nedan.

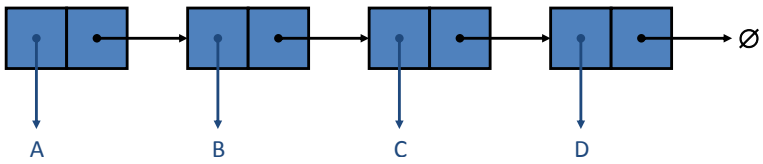
2.22

3.3 Enkel- och dubbellänkade listor

Enkellänkade listor

- En enkellänkad lista är en konkret datastruktur bestående av en sekvens av noder
- Varje nod lagrar
  - element
  - pekare till nästa nod

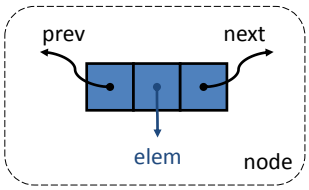


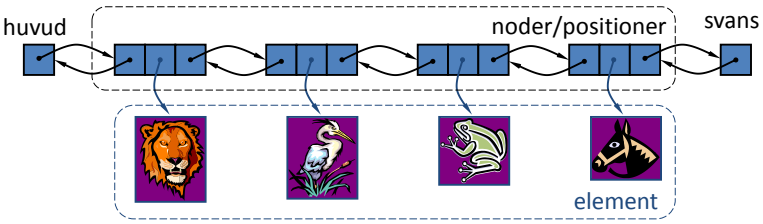


2.23

Dubbellänkade listor

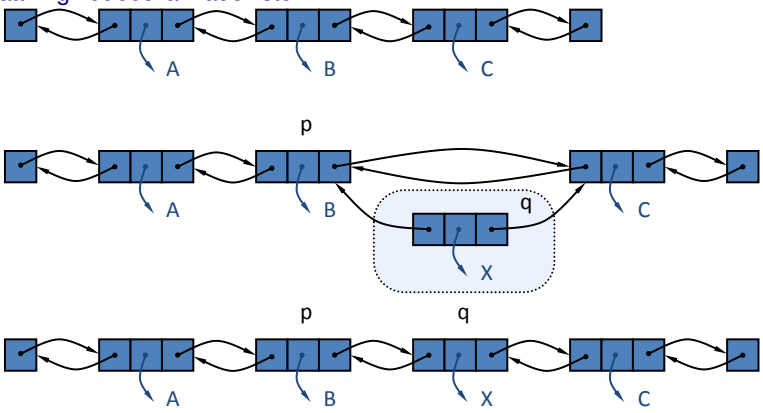
- En dubbelänkad lista ger en naturlig implementation av ADTn nodlista
- Varje nod lagrar
  - element
  - pekare till föregående nod
  - pekare till nästa nod
- Särskilda huvud- och svansnoder





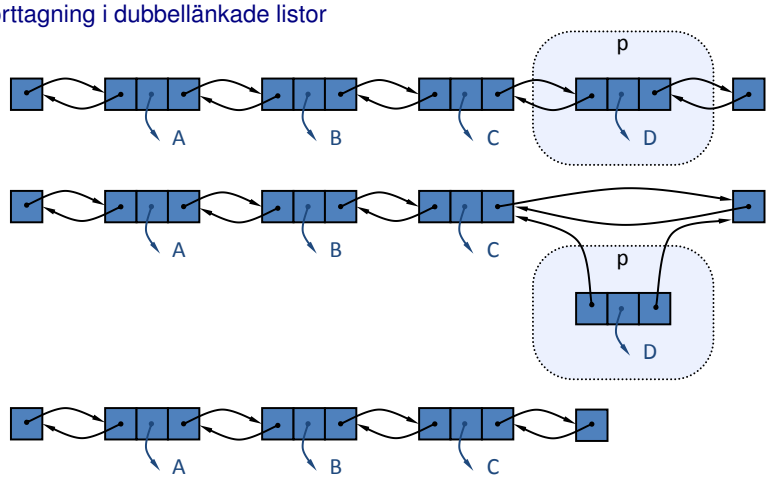
2.24

Insättning i dubbellänkade listor



2.25

Borttagning i dubbellänkade listor



2.26