Föreläsning
Datastrukturer (DAT036)

Nils Anders Danielsson

2013-10-30

Repetition

- Analys av tidskomplexitet.
- Kostnadsmodeller.
- Asymptotisk komplexitet/notation.
- Dynamiska arrayer.

Amorterad tidskomplexitet

- Tidskomplexitet för att lägga till n element till en tom dynamisk array: $\Theta(n)$.
- Tidskomplexitet för att lägga till ett element: $O(n)$, där n är arrayens storlek.
- *Amorterad* tidskomplexitet för att lägga till ett element: $O(1)$.

Amorterad tidskomplexitet

- Man kan låta tidiga, billiga operationer "betala" för dyra, sena.
- Efter dubblering:
  n snabba insättningar,
  1 dubblering.
- Insättning: Lägg ett mynt på cellen,
  och ett på en "gammal" cell.
- Kopiering: Har ett mynt på varje cell,
  kopieringen betald.

Amorterad tidskomplexitet

- Vissa operationer långsamma:
  kan vara problematiskt i realtidssammanhang.
- Gamla tentor m m: potentialmetoden.

Dynamiska arrayer med `remove`

Har `add` och `remove` amorterade tidskomplexiteten $O(1)$ om man halverar arrayen när den blir...

- ...halvfull?
- ...kvartsfull?

Varför?

`http://pingo.upb.de/`

Problem/algoritm

Problem: sortera en lista.

Algoritmer:

- Insertion sort.
- Mergesort.
- Quicksort.
- ...

Abstrakt datatyp/datastruktur

Abstrakt datatyp (matematisk abstraktion): lista.

Datastrukturer (implementation):

- Array (dynamisk?).
- Enkellänkad lista.
- ...

Listor, stackar och köer: ADT-er

ADT	Operationer (exkl konstruerare)
Stack	push, pop
Kö	enqueue, dequeue
Lista	add(x), add(x, i), remove(x), remove(i), get(i), set(i, x), contains(x), size, iterator
Iterator	hasNext, next, remove

(Inte exakta definitioner.)

Listor, stackar och köer: datastrukturer

ADT	Implementationer
Lista	dynamisk array, enkellänkad lista, dubbellänkad lista
Stack	lista
Kö	lista, cirkulär array, två listor

ADT-er

- Kan använda en ADT för att implementera en annan.
- Även på tentan!

Collections i Java

- Java Collections Framework.
- ADT $\sim$ interface, datastruktur $\sim$ klass.

Listor, stackar och köer: tillämpningar

Diskussionsuppgift

- Vad kan man ha de här ADT-erna till?
- Svara helst med ett ord per fält,
  gärna substantiv i obestämd form, singular.

Listor, stackar och köer: tillämpningar

Stackar
- Implementera rekursion.
- Evaluera postfix-uttryck.
- Topologisk sortering (grafalgoritm).
- ...

Köer
- Skrivarköer.
- Simulering av ”riktiga” köer.
- Topologisk sortering (grafalgoritm).
- Bredden först-sökning (grafalgoritm).
- ...

Listor ...

Länkade listor

Många varianter:

- Objekt med pekare till första noden, kanske storlek.
- Pekare till sista noden?
- Enkellänkad, dubbellänkad?
- Vaktposter (sentinels)? Först/sist/både och?

Dubbellänkad lista med dubbla vaktposter

```
public class DoublyLinkedList<A> {
    private class ListNode {
        A contents;
        ListNode next;  // null omm sista vakten.
        ListNode prev;  // null omm första vakten.
    }

    ...
}
```

Dubbellänkad lista med dubbla vaktposter

```
public class DoublyLinkedList<A> {
    ...

    private ListNode first, last;  // Ej null.
    private int size;

    // Konstruerar tom lista.
    public DoublyLinkedList() {
        first      = new ListNode();
        last       = new ListNode();
        first.next = last;
        last.prev  = first;
        size       = 0;
    }
}
```

Övning

- Implementera metoden `addFirst`.
  (Tips: Rita först upp vad som ska hända.)
- Hitta sedan felet i följande kod.

```
1 void addFirst(A x) {
2   ListNode node = new ListNode();
3   node.contents = x;
4   node.next     = first.next;
5   node.prev     = first;
6   last.prev     = node;
7   first.next    = node;
8   size++;
9 }
```

Invariant

- Egenskap som ”alltid” gäller för programmets tillstånd.
- Exempel:
  1. `first != null`.
  2. `last.next = null`.
  3. `n.next.prev = n` (om `n` ej är vaktpost).
  4. $\texttt{first.next}^{\texttt{size}+1} = \texttt{last}$.
- Invarianter bryts ibland temporärt när objekt uppdateras.

Precondition

- Krav som förväntas vara uppfyllt
  då metod anropas.
- Exempel: `pop` kräver att stacken inte är tom.

Postcondition

- Egenskap som är uppfylld efter anrop, givet preconditions och invarianter.
- Exempel: Efter `push` är stacken inte tom.

Assertion

Man kan testa vissa egenskaper med "assertions".
Kan vara smidigt för att hitta/undvika fel i labbar.

Fail.java

```
public class Fail {
  public static void main (String[] args) {
    assert(args.length == 2);
  }
}
```

```
$ javac Fail.java
$ java Fail
$ java -ea Fail ett två
$ java -ea Fail
Exception in thread "main" java.lang.AssertionError
        at Fail.main(Fail.java:3)
```

Assertion

I Haskell (`assert :: Bool -> a -> a`):

Fail.hs

```
module Fail where

import Control.Exception

foo :: Integer -> Integer
foo n = assert (n > 0) (7 `div` n)
```

```
*Fail> foo 3
2
*Fail> foo 0
*** Exception: Fail.hs:6:9-14: Assertion failed
```

Assertion

```
// Skapar ny /intern/ listnod.
// Precondition: prev != null && next != null.
ListNode(A contents, ListNode prev, ListNode next) {
  assert prev != null && next != null;

  this.contents = contents;
  this.prev     = prev;
  this.next     = next;
}
```

Nu leder

```
new ListNode(x,first.prev,first.next)
```

till ett `AssertionError` (med `-ea`),
inte ett kryptiskt fel senare.

Assertion

```
// Lägger till x efter n.
// Precondition: n != null && n != last.
void addAfter(ListNode n, A x) {
    assert n != null && n != last;

    ListNode next = n.next;
    n.next        = new ListNode(x, n, next);
    next.prev     = n.next;
    size++;
}

void addFirst(A x) {
    addAfter(first, x);
}
```

Korrekthet

Hur kan man förvissa sig om att man löst en uppgift korrekt?

- Bevis. Kan vara svårt, ta mycket tid.
- Tester. Kan gå snabbare.

Tester

När ni jonglerar pekare (på papper eller i dator):

- Testa gärna lösningen.
- Några representativa fall:
  - Tom lista.
  - Lista med några element.
  - ...

```
// Implementera en operation som lägger till en lista i slutet av en
// annan lista. I värsta fallet ska operationen ta konstant tid.
// Exempel: Om man lägger till [3, 4, 5] i slutet av [1, 2] ska
// resultatet bli [1, 2, 3, 4, 5].

// Enkellänkade listor med en "vaktpost" i början samt pekare till
// vaktposten och sista noden:
class List<A> {
    class ListNode {
        A        contents;
        ListNode next;
    }

    ListNode head;  // Pekar på vaktposten.
    ListNode tail;  // Pekar på vaktposten om listan är tom,
                    // annars på den sista riktiga noden.

    // Lägger till ys sist i listan, utan att förstöra ys.
    void append(List<A> ys) {
        tail.next = ys.head.next;
        tail      = ys.tail;
    }

    // Fler operationer...
}
```

Haskell

- Enkellänkade listor.
- Två slags noder: tom lista och conscell.
- Endast pekare till första noden,
  inget objekt innehållandes storlek e d.
- Inga vaktposter.
- Svansar kan delas.
- Exempel:

```
xs = [3, 4, 5]
ys = 1 : 2 : xs
```

Persistens

- (Vanliga) Haskellistor kan inte uppdateras.
- När man lägger till ett element till en lista så finns den gamla listan kvar (tills den skräpsamlas).
- Fördel: Behöver inte kopiera listor.
- Fördel: Parallellprogrammering kan bli enklare.
- Nackdel: Vissa operationer mindre effektiva.
- Kan implementera persistenta datastrukturer i Java också.

Hur många consceller innehåller `tails [1, 2, 3]`?

```
tails :: [a] -> [[a]]
tails [] = [[]]
tails xs = xs : tails (tail xs)

tails [1, 2, 3] =
  [[1, 2, 3], [2, 3], [3], []]
```

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista	Haskellista
`add(x)`			
`add(x, i)`			
`remove(x)`			
`remove(i)`			
`get(i)`			
`set(i, x)`			
`contains(x)`			
`size`			
`iterator`			
`hasNext/next`			
`remove`			

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista	Haskellista
`add(x)`	$O(1)$ am	$O(1)$	$O(1)$
`add(x, i)`			
`remove(x)`			
`remove(i)`			
`get(i)`			
`set(i, x)`			
`contains(x)`			
`size`			
`iterator`			
`hasNext/next`			
`remove`			

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista	Haskellista
`add(x)`	$O(1)$ am	$O(1)$	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$	$O(n)$
`remove(x)`			
`remove(i)`			
`get(i)`			
`set(i, x)`			
`contains(x)`			
`size`			
`iterator`			
`hasNext/next`			
`remove`			

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista	Haskellista
`add(x)`	$O(1)$ am	$O(1)$	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$	$O(n)$
`remove(i)`			
`get(i)`			
`set(i, x)`			
`contains(x)`			
`size`			
`iterator`			
`hasNext/next`			
`remove`			

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista	Haskellista
`add(x)`	$O(1)$ am	$O(1)$	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$	$O(n)$
`remove(i)`	$O(n)$	$O(n)$	$O(n)$
`get(i)`			
`set(i, x)`			
`contains(x)`			
`size`			
`iterator`			
`hasNext/next`			
`remove`			

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista	Haskellista
`add(x)`	$O(1)$ am	$O(1)$	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$	$O(n)$
`remove(i)`	$O(n)$	$O(n)$	$O(n)$
`get(i)`	$O(1)$	$O(n)$	$O(n)$
`set(i, x)`			
`contains(x)`			
`size`			
`iterator`			
`hasNext/next`			
`remove`			

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista	Haskellista
`add(x)`	$O(1)$ am	$O(1)$	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$	$O(n)$
`remove(i)`	$O(n)$	$O(n)$	$O(n)$
`get(i)`	$O(1)$	$O(n)$	$O(n)$
`set(i, x)`	$O(1)$	$O(n)$	$O(n)$
`contains(x)`			
`size`			
`iterator`			
`hasNext/next`			
`remove`			

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista	Haskellista
`add(x)`	$O(1)$ am	$O(1)$	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$	$O(n)$
`remove(i)`	$O(n)$	$O(n)$	$O(n)$
`get(i)`	$O(1)$	$O(n)$	$O(n)$
`set(i, x)`	$O(1)$	$O(n)$	$O(n)$
`contains(x)`	$O(n)$	$O(n)$	$O(n)$
`size`			
`iterator`			
`hasNext/next`			
`remove`			

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista	Haskellista
`add(x)`	$O(1)$ am	$O(1)$	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$	$O(n)$
`remove(i)`	$O(n)$	$O(n)$	$O(n)$
`get(i)`	$O(1)$	$O(n)$	$O(n)$
`set(i, x)`	$O(1)$	$O(n)$	$O(n)$
`contains(x)`	$O(n)$	$O(n)$	$O(n)$
`size`	$O(1)$	$O(1)$	$O(n)$
`iterator`			
`hasNext/next`			
`remove`			

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista	Haskellista
`add(x)`	$O(1)$ am	$O(1)$	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$	$O(n)$
`remove(i)`	$O(n)$	$O(n)$	$O(n)$
`get(i)`	$O(1)$	$O(n)$	$O(n)$
`set(i, x)`	$O(1)$	$O(n)$	$O(n)$
`contains(x)`	$O(n)$	$O(n)$	$O(n)$
`size`	$O(1)$	$O(1)$	$O(n)$
`iterator`	$O(1)$	$O(1)$	
`hasNext/next`			
`remove`			

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista	Haskellista
`add(x)`	$O(1)$ am	$O(1)$	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$	$O(n)$
`remove(i)`	$O(n)$	$O(n)$	$O(n)$
`get(i)`	$O(1)$	$O(n)$	$O(n)$
`set(i, x)`	$O(1)$	$O(n)$	$O(n)$
`contains(x)`	$O(n)$	$O(n)$	$O(n)$
`size`	$O(1)$	$O(1)$	$O(n)$
`iterator`	$O(1)$	$O(1)$	
`hasNext/next`	$O(1)$	$O(1)$	
`remove`			

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista	Haskellista
`add(x)`	$O(1)$ am	$O(1)$	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$	$O(n)$
`remove(i)`	$O(n)$	$O(n)$	$O(n)$
`get(i)`	$O(1)$	$O(n)$	$O(n)$
`set(i, x)`	$O(1)$	$O(n)$	$O(n)$
`contains(x)`	$O(n)$	$O(n)$	$O(n)$
`size`	$O(1)$	$O(1)$	$O(n)$
`iterator`	$O(1)$	$O(1)$	
`hasNext/next`	$O(1)$	$O(1)$	
`remove`	$O(n)$	$O(1)$	

Listor: tidskomplexitet

Några kommentarer:

- `remove`, `contains`:
  Givet att jämförelser tar konstant tid.
- Indexbaserade operationer för länkade listor:
  Bättre än $\Theta(n)$ om man vet att `i` pekar
  ”nära” början (eller i vissa fall slutet) av listan.

Cirkulära arrayer

Implementationsteknik för köer.

Cirkulära arrayer

```
public class CircularArrayQueue<A> {
    private A[] queue; // Invariant: queue.length > 0.
    private int size;
    private int front; // Nästa element (eller rear).
    private int rear;  // Nästa lediga (eller front).

    // Precondition: capacity > 0.
    public CircularArrayQueue(int capacity) {
        if (capacity <= 0) {
            throw new NonPositiveCapacityException();
        }
        queue = (A[]) new Object[capacity];
        size = front = rear = 0;
    }

    ...
```

Cirkulära arrayer

```
// Precondition: Kön får inte vara full.
public void enqueue(A a) {
    if (size == queue.length) {
        throw new FullQueueException();
    }

    size++;
    queue[rear] = a;
    rear = (rear + 1) % queue.length;
}
```

(Fungerar ej om `rear` = `Integer.MAX_VALUE`...)

Cirkulära arrayer

```
// Precondition: Kön får inte vara tom.
public A dequeue() {
    if (size == 0) {
        throw new EmptyQueueException();
    }

    size--;
    A a = queue[front];
    queue[front] = null;  // Undvik minnesläckor.
    front = (front + 1) % queue.length;

    return a;
}
```

Köer i Haskell

- Implementera kö-ADTn m h a lista?
- Nja, dyrt att sätta in element sist.
- Ett alternativ: två listor.

Köer i Haskell

```
module Queue
  (Queue, empty, isEmpty, enqueue, dequeue) where

-- Invariant: front är tom omm kön är tom.
data Queue a = Q { front :: [a], rear :: [a] }

empty :: Queue a
empty = Q [] []

isEmpty :: Queue a -> Bool
isEmpty (Q [] _) = True
isEmpty _        = False
```

Köer i Haskell

```
-- Invariant: front är tom omm kön är tom.
data Queue a = Q { front :: [a], rear :: [a] }

enqueue :: a -> Queue a -> Queue a
enqueue x (Q [] _) = Q [x] []
enqueue x (Q f r)  = Q f (x : r)

first :: Queue a -> Maybe a
first (Q []      _) = Nothing
first (Q (x : _) _) = Just x

dequeue :: Queue a -> Maybe (Queue a)
dequeue (Q []      _) = Nothing
dequeue (Q [_]     r) = Just (Q (reverse r) [])
dequeue (Q (_ : f) r) = Just (Q f r)
```

Köer i Haskell

Tidskomplexitet (jag ignorerar lathet):

- `empty`, `isEmpty`, `enqueue`, `first`: $\Theta(1)$.
- `dequeue`: $O(1)$ (amorterat).
  Betala ett mynt extra vid insättning,
  använd mynten för att betala för `reverse`.

Skapa en kö innehållandes $1, 2, ..., n$:

```
xs = enqueue n (… (enqueue 2 (enqueue 1 empty)) …)
```

Kör sedan följande kod n ggr:

```
dequeue xs
```

Vad är tidskomplexiteten?

- $\Theta(1)$.
- $\Theta(n)$.
- $\Theta(n^2)$.
- $\Theta(n^3)$.

Persistens, igen

- Analysen fungerar inte eftersom vi betalar med samma mynt många gånger.
- Den amorterade tidskomplexiteten för `dequeue` är $O(1)$ om kön används *enkeltrådat*.
- Man kan konstruera persistenta köer med bra "flertrådad" tidskomplexitet, se t ex Okasakis avhandling.

Sammanfattning

- Amorterad tidskomplexitet.
- ADT-er/datastrukturer.
- Persistenta datastrukturer.
- Listor, stackar, köer.
- Invarianter, assertions m m.
- Testning.
- Länkade listor, pekarjonglering.
- Cirkulära arrayer.
- Köer m h a två listor.