Föreläsning
Datastrukturer (DAT037)

Nils Anders Danielsson

2015-11-06

Repetition

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

Amorterad tids-komplexitet

Amorterad tidskomplexitet

- Lägga till ett element till en dynamisk array: $O(\ell)$, där ℓ är antalet element i arrayen.
- I avsaknad av annan information:
  Lägga till n element till en tom dynamisk array: $O(n^2)$.
- Vår analys från förra föreläsningen: $\Theta(n)$.
- $O(\ell)$ ger inte tillräckligt mycket information.

Amorterad tidskomplexitet

- Lägga till ett element till en dynamisk array: $O(\ell)$, där ℓ är antalet element i arrayen.
- I avsaknad av annan information:
  Lägga till n element till en tom dynamisk array: $O(n^2)$.
- Vår analys från förra föreläsningen: $\Theta(n)$.
- $O(\ell)$ ger inte tillräckligt mycket information.
- Vi kommer att använda

  amorterad tidskomplexitet

  för att hantera den här situationen på ett smidigt sätt.

Bokföringsmetoden

- 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.

Bokföringsmetoden

Amorterad tidskomplexitet =
faktisk tidskomplexitet
+ värdet av nya mynt
− värdet av sparade mynt som används.

Bokföringsmetoden

- Amorterad tidskomplexitet för insättning utan kopiering:

$$O(1) + 2m = O(1).$$

  Här är m värdet av ett mynt (en konstant > 0).
- Amorterad tidskomplexitet för kopiering:

$$O(n) - nm.$$

- Om m är tillräckligt stor:

$$O(n) - nm = O(1).$$

Bokföringsmetoden

- Amorterad tidskomplexitet för att lägga till ett element till en dynamisk array: $O(1)$.
- I avsaknad av andra operationer.
- Kan använda klass- eller modulsystem för att förhindra operationer som förstör analysen.

Bokföringsmetoden

- Amorterad tidskomplexitet för att lägga till ett element till en dynamisk array: $O(1)$.
- I avsaknad av andra operationer.
- Kan använda klass- eller modulsystem för att förhindra operationer som förstör analysen.
- En operation kan ges olika amorterad tidskomplexitet i olika analyser, beroende på hur mycket vi betalar, och hur många sparade mynt vi använder.

Bokföringsmetoden

- Börja med en datastruktur ”utan mynt”.
- Utför n operationer som var och en har den amorterade tidskomplexiteten $O(1)$.
- N är antalet nya mynt som betalats, S är antalet sparade mynt som använts:

$$N - S \geq 0.$$

- Den *riktiga* tidskomplexiteten för den här sekvensen av operationer är

$$T \leq T + (N - S)m = O(n).$$

Amorterad tidskomplexitet

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

Dynamiska arrayer med `remove`

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

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

Varför?

Listor, stackar, köer

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 datastruktur för en viss ADT för att implementera en annan ADT.
- Även på tentan!

Collections i Java

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

Stackar och köer: tillämpningar

Stackar

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

Köer

- Skrivarköer.
- Topologisk sortering (grafalgoritm).
- Bredden först-sökning (grafalgoritm).
- ...

Länkade 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 first;  // Pekar på vaktposten.
    ListNode last;   // Pekar på vaktposten om listan är tom,
                     // annars på den sista riktiga noden.

    // Lägger till ys sist i listan, ys blir tom.
    void append(List<A> ys) {
        last.next     = ys.first.next;
        last          = ys.last;
        ys.first.next = null;
        ys.last       = ys.first;
    }

    // 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.
- Exempel:

```
xs = [2, 3] = 2 : 3 : []
```

Iteratorer

Listor används ofta som iteratorer:

- Istället för `hasNext`/`next` används mönstermatchning.
- `remove` finns inte.

Persistens

- (Vanliga) Haskellistor kan inte uppdateras.
- När man ”ändrar” en lista
  så finns den gamla listan kvar
  (tills den skräpsamlas).
- Svansar kan delas.

Persistens

Exempel:

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

Här delas `[2, 3]` mellan två listor.

Persistens

Append:

```
(++) :: [a] -> [a] -> [a]
[]       ++ bs = bs
(a : as) ++ bs = a : (as ++ bs)
```

Exempel:

```
xs = [2, 3]
zs = xs ++ [4]
```

Här delas bara listelementen (2 och 3).

Persistens

- Fördel: Kanske enklare att förstå, undvika buggar.
- Nackdel: Vissa operationer mindre effektiva.
- Kan implementera persistenta datastrukturer i Java också.

Hur många consceller innehåller
`tails [1, 2, 3]`
(när uttrycket är fullt evaluerat)?

```
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`	$O(1)$	$O(1)$	$O(1)$
`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`	$O(1)$	$O(1)$	$O(1)$
`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.

Sammanfattning

- Amorterad tidskomplexitet.
- ADT-er/datastrukturer.
- Persistenta datastrukturer.
- Listor, stackar, köer.
- Invarianter, assertions m m.
- Testning.
- Länkade listor, pekarjonglering.