Föreläsning
Datastrukturer (DAT036)

Nils Anders Danielsson

2013-11-04

Repetition

Förra gången:

- Amorterad tidskomplexitet.
- Listor, stackar, köer.
- Länkade listor, pekarjonglering.
- Testning.
- Persistenta datastrukturer.

Idag

- Mer om persistenta datastrukturer.
- Listor: sammanfattning.
- Cirkulära arrayer.
- Köer m h a två listor.
- Träd.
- Prioritetsköer. (Om vi hinner.)

Persistens

- (Vanliga) Haskellistor kan inte uppdateras.
- När man lägger till ett element till en lista så finns den gamla listan kvar (tills den skräpsamlas).
- Svansar kan delas.

Persistens

Exempel:

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

Här delas `tail xs` mellan två listor.

Persistens

Exempel:

```
xs = [2, 3, 4]
zs = xs ++ [5]
```

Här delas bara listelementen (2, 3 och 4).

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)`	$O(1)$ 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;
}
```

Vilka datastrukturer ger $O(1)$ `enqueue` och `dequeue` för FIFO-köer?

- Dynamiska arrayer (linjära).
- Cirkulära, dynamiska arrayer.
- Enkellänkade listor utan pekare till sista noden.
- Enkellänkade listor med pekare till sista noden.
- Enkellänkade listor med pekare till två vaktposter (först och sist).
- Dubbellänkade listor med pekare till två vaktposter (först och sist).

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.

Träd

- Ett träd är tomt eller icketomt.
- Ett icketomt träd består av:
  - En rot (ev med innehåll).
  - Noll eller flera icke-tomma delträd (ofta ordnade).
- Terminologi:
  - Förälder, barn, syskon, barnbarn o s v.
  - Löv.

Träd

En representation:

```
data Tree a              = Empty
                         | NonEmpty (NonEmptyTree a)

data NonEmptyTree a = Node a [NonEmptyTree a]
```

Exempel:

```
tree :: Tree Integer
tree = NonEmpty (Node 1 [ Node 2 []
                        , Node 3 [Node 5 []]
                        , Node 4 []
                        ])
```

Träd

En annan representation:

```
class TreeNode<A> {
  A           contents;
  TreeNode<A> firstChild;   // null om löv.
  TreeNode<A> nextSibling;  // null om sista barnet.
}

class Tree<A> {
  TreeNode<A> root;  // null om trädet är tomt.
}
```

Träd

```
Tree<Integer> tree =
  new Tree<>
    ( new TreeNode<>
        ( 1
        , new TreeNode<>
            ( 2
            , null
            , new TreeNode<>
                ( 3
                , new TreeNode<>(5, null, null)
                , new TreeNode<>(4, null, null)
                )
            )
        , null
        )
    );
```

Träd

Höjd:

- Tomma träd har höjd -1.
- Annars:
  Antalet steg från roten till djupaste lövet.

Träd

Höjd:

- Tomma träd har höjd -1.
- Annars:
  Antalet steg från roten till djupaste lövet.

```
height :: Tree a -> Integer
height Empty          = -1
height (NonEmpty t) = heightNE t

heightNE :: NonEmptyTree a -> Integer
heightNE (Node _ [])       = 0
heightNE (Node _ children) =
  1 + maximum (map heightNE children)
```

Träd

```
1  int height(TreeNode<A> n) {
2    if (n == null) {
3      return -1;
4    } else {
5      int largestHeight   = 0;
6      TreeNode<A> current = n.firstChild;
7      while (current != null) {
8        largestHeight = max(largestHeight,
9                            height(current));
10       current       = current.nextSibling;
11     }
12     return 1 + largestHeight;
13   }
14 }
```

Träd

```
1  int height(TreeNode<A> n) {
2    if (n == null) {
3      return -1;
4    } else {
5      int largestHeight  = 0;
6      TreeNode<A> current = n.firstChild;
7      while (current != null) {
8        largestHeight = max(largestHeight,
9                            height(current));
10       current       = current.nextSibling;
11     }
12     return 1 + largestHeight;
13   }
14 }
```

Hitta felet!

Testa.

Binära träd

Binära träd:

- Max två barn.
- Skiljer på vänster och höger barn även om det bara finns ett.

Binära träd

En representation:

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)
```

Exempel:

```
tree :: Tree Integer
tree = Node (Node Leaf 2 Leaf)
            1
            (Node Leaf 3 (Node Leaf 5 Leaf))
```

Binära träd

En annan representation:

```
class Tree<A> {
  class TreeNode {
    A        contents;
    TreeNode left;   // null om vänster barn saknas.
    TreeNode right;  // null om höger barn saknas.
  }

  TreeNode root;  // null om trädet är tomt.
}
```

Prioritetsköer

Köer där varje element har viss prioritet.

Gränssnitt (exempel):

- Konstruerare för tom kö.
- `insert`: Lägger till element.
- `delete-min`:
  Tar bort och ger tillbaka *minsta* elementet.
- `find-min`: Ger tillbaka minsta elementet.
- `decrease-key`: Ändrar ett elements prioritet.
- `merge`: Slår ihop två köer.

Prioritetsköer

Några tillämpningar:

- Schemaläggning av processer.
- Sortering.
- Dijkstras algoritm (labb 3).

Labb 2: Implementera prioritetskö.

Om man implementerar
prioritetskö-ADTn med listor,
vad blir värstafallstidskomplexiteten för
`insert` och `delete-min`?

- `insert`: $\Theta(1)$, `delete-min`: $\Theta(1)$.
- `insert`: $\Theta(1)$, `delete-min`: $\Theta(n)$.
- `insert`: $\Theta(n)$, `delete-min`: $\Theta(1)$.
- `insert`: $\Theta(n)$, `delete-min`: $\Theta(n)$.

Binära heapar

Kompletta binära träd med heapordningsegenskapen.

Heapordningsegenskapen

Varje nod är mindre än eller lika med alla sina barn.

Komplett binärt träd

Så lågt som möjligt, alla nivåer helt fyllda utom möjligtvis den sista, som är fylld från vänster.

En binär heap med n noder har höjden $\Theta(\log n)$.

Implementation av binära heapar

- Tom kö: Tomt träd.
- `find-min`: Ge tillbaka roten.
- `insert`: Stoppa in sist. Bubbla upp.
- `delete-min`: Ta bort roten.
  Stoppa in sista elementet överst. Bubbla ned.
  Ge tillbaka gamla roten.
- `decrease-key`: Ändra prioritet,
  bubbla åt rätt håll.

Bubbla upp/ned tills heapordningsegenskapen återställts.

Tidskomplexitet

Om man kan hitta noderna snabbt:

- Tom kö: $O(1)$.
- `find-min`: $O(1)$.
- `insert`: $O(\log n)$.
- `delete-min`: $O(\log n)$.
- `decrease-key`: $O(\log n)$.

(Givet att jämförelser tar konstant tid.)

De flesta noderna är långt ned i trädet.
Medelkomplexitet för `insert`: $O(1)$.

Implementation av binära heapar

Kan representera trädet med array (labb 2).

- Rot på position 1.
- Nod is vänstra barn: $2i$.
- Nod is högra barn: $2i + 1$.
- Nod is förälder ($i > 1$): $\lfloor i/2 \rfloor$.

`decrease-key`

När man implementerar `decrease-key`,
hur hittar man noden snabbt?

Kan använda extra datastruktur.

Exempel: Hashtabell.

Sammanfattning

- Cirkulära arrayer.
- Köer m h a två listor.
- Träd.
- Prioritetsköer.

Nästa gång:

- Mer om prioritetsköer.
- Kanske lite om sortering.