Föreläsning
Datastrukturer (DAT036)

Nils Anders Danielsson

2012-11-26

Administrativt

Man kan nu boka plats i labbsalarna.

Idag

- Repetition av ordonotation.
- Duggaresultat.
- Sökträd.

Ordo

Ordonotation (en variant)

$T(n) = O(f(n))$

om och endast om

det finns ett naturligt tal n_0

och ett reellt tal $c > 0$

så att $T(n) \leq cf(n)$ för alla $n \geq n_0$.

Ordonotation: regler

$T(n)$ är ett polynom av grad k:

$$T(n) = O(n^k).$$

- $7n^2 + 3n + 2 = O(n^2)$.
- $0.1n^3 + 1000 = O(n^3)$.

Ordonotation: regler

Om $k > 0$ är en *konstant*:

$$(\log_2 n)^k = O(n) \quad \text{(ej } \Theta(n)\text{)},$$
$$\log_2(n^k) = k \log_2 n = O(\log n).$$

För konstant $a > 1$:

$$\log_a n = \log_2 n / \log_2 a = O(\log_2 n) = O(\log n).$$

- $(\log_2 n)^{10000} = O(n)$.
- $\log_2(n^{10000}) = O(\log n)$.

Ordonotation: regler

Om $T(n) = O(f(n))$ och $U(n) = O(g(n))$:

$$\begin{aligned} T(n) + U(n) &= O(f(n) + g(n)) \\ &= O(\max(f(n), g(n))), \\ T(n)U(n) &= O(f(n)g(n)). \end{aligned}$$

- $7n^2 + 3n^2 = O(n^2 + n^2) = O(n^2)$.
- $2n^3 + (\log_2 n)^{1000} = O(n^3 + n) = O(n^3)$.
- $2n^3 \log_7 5n^2 = O(n^3 \log n)$.

Övning

Ge ett lämpligt ordouttryck för

$$\left(2n^7 + 3n \log_{12} \sqrt{n}\right) \left(n^{-7} + 1\right).$$

Övning

Ge ett lämpligt ordouttryck för

$$\left(2n^7 + 3n \log_{12} \sqrt{n}\right) \left(n^{-7} + 1\right).$$

$$\begin{aligned}
&\left(2n^7 + 3n \log_{12} \sqrt{n}\right) \left(n^{-7} + 1\right) = \\
&O\left(\left(n^7 + n \log n\right) (1)\right) = \\
&O\left(n^7 + n^2\right) = \\
&O\left(n^7\right).
\end{aligned}$$

Dugga

Duggaresultat

Uppgift 1 24/65.

Uppgift 2 13/65.

Uppgift 3 13/65.

Uppgift 1

```
for(int i = 0; i < n; i++) {
  xs.add-first(pq.delete-min());
}
```

- `add-first`: $O(|\texttt{xs}|)$.
  Totalt: $O(\texttt{n}^2)$.
- `delete-min`: $O(\log |\texttt{pq}|)$.
  Totalt: $O(\texttt{n} \log \texttt{n}^3) = O(\texttt{n} \log \texttt{n})$.
- Totalt: $O(\texttt{n}^2 + \texttt{n} \log \texttt{n}) = O(\texttt{n}^2)$.

Uppgift 1

```
for(int i = 0; i < n; i++) {
  xs.add-first(pq.delete-min());
}
```

$$O\left(\sum_{i=0}^{\mathtt{n}-1}(i+\log(\mathtt{n}^3-i))\right) =$$

$$O\left(\sum_{i=0}^{\mathtt{n}-1} i + \sum_{i=0}^{\mathtt{n}-1}\log(\mathtt{n}^3-i)\right) =$$

$$O\left(\mathtt{n}^2 + \sum_{i=0}^{\mathtt{n}-1}\log \mathtt{n}\right) = O(\mathtt{n}^2+\mathtt{n}\log\mathtt{n}) = O(\mathtt{n}^2).$$

Uppgift 1

```
for(int i = 0; i < n; i++) {
  xs.add-first(pq.delete-min());
}
```

Vanliga misstag:

- `add-first`: $O(1)$.
- Multiplikation istället för addition: $n \cdot n \cdot \log n^3$.
- `delete-min`: linjär eller konstant.

Uppgift 2

Implementera en algoritm som, givet ett binärt träd utan föräldrapekare, skapar ett motsvarande träd med föräldrapekare.

```
public TreeWith(TreeWithout<A> t) {
    root = addParents(t.root, null);
}

private TreeNode addParents
    (TreeWithout<A>.TreeNode without,
     TreeNode parent) {
    if (without == null) return null;

    TreeNode with = new TreeNode(without.contents,
                                 null, null, parent);

    with.left  = addParents(without.left,  with);
    with.right = addParents(without.right, with);

    return with;
}
```

Uppgift 2

Vanliga misstag:

- Cyklisk kod:

```
with  = new TreeNode(..., left, right, ...);
left  = addParents(..., with);
right = addParents(..., with);
```

- Cyklisk kod:

```
with = new TreeNode(...,
                    addParents(..., with),
                    addParents(..., with),
                    ...);
```

- Ihopblandning av träd och trädnoder.
- Referenser till det gamla trädet i det nya.

Uppgift 2

Vanliga misstag:

- Cyklisk kod:

```
with  = new TreeNode(..., left, right, ...);
left  = addParents(..., with);
right = addParents(..., with);
```

- Cyklisk kod:

```
with = new TreeNode(...,
                    addParents(..., with),
                    addParents(..., with),
                    ...);
```

- Ihopblandning av träd och trädnoder.
- Referenser till det gamla trädet i det nya.

Testa!

Uppgift 3

Beskriv en *linjär* algoritm som avgör om två listor innehållandes heltal är permutationer av varandra.

- Hashtabell som håller reda på antalet förekomster av varje tal.
- Beräkna förekomster för första listan.
- Subtrahera förekomster för andra listan; `false` om något element saknas i första.
- Kontrollera om summan är noll för varje element i första listan.

Uppgift 3

Beskriv en *linjär* algoritm som avgör om två listor innehållandes heltal är permutationer av varandra.

```
occurrences = new hash table

for i in the first list do
  n = occurrences.get(i)
  if n == null then n = 0
  occurrences.set(i, n + 1)
```

Uppgift 3

Beskriv en *linjär* algoritm som avgör om två listor innehållandes heltal är permutationer av varandra.

```
for i in the second list do
  n = occurrences.get(i)
  if n == null then
    return false
  else
    occurrences.set(i, n - 1)

for i in the first list do
  if not (occurrences.get(i) == 0) then
    return false

return true
```

Uppgift 3

Beskriv en *linjär* algoritm som avgör om två listor innehållandes heltal är permutationer av varandra.

Vanliga misstag:

- Sortera båda listorna, testa likhet: $\Theta(n \log n)$ (med t ex mergesort).
- Felaktig hantering av listor med repeterade element. Testa!

Tips

- Läs kursboken, gärna *innan* föreläsningarna.
- Lös många uppgifter.
- Många gamla tentauppgifter har lösningar.
  Spara inte alla sådana uppgifter
  till tentaplugget.

Tips

Några potentiellt lämpliga *gamla* tentauppgifter för de som misslyckades med duggauppgift...

1: 2009-12/2, 2008-12/3, 2007-12/5, 2005-12/3, 2004-12/2, 2003-12/2, 2002-12/3, 2001-12/1.

2: 2006-12/4, 2001-12/2.

3: 2006-12/7, 2005-12/4, 2003-12/3, 2002-12/4.

Sökträd

Binära sökträd

Binära träd med sökträdsegenskapen:

- Tomma binära träd är sökträd.
- Ett icketomt binärt träd är ett sökträd om:
  Vänster och höger delträd är sökträd och
  alla element i vänstra delträdet $<$
  elementet i roten $<$
  alla element i högra delträdet.

Binära sökträd

- Behöver kunna jämföra element, t ex med komparator.
- Komparatorn antas (oftast?) implementera en *total ordning*.

Binära sökträd

Sökträd kan användas för att implementera utökad ”set”-/”map”-ADT:

- Konstruerare: Tom mängd/avbildning.
- `member(k)`/`lookup(k)`.
- `insert(k)`/`insert(k,v)`.
- `delete(k)`.
- `find-min()`.
- `delete-min()`.
- `iterator()`:
  Går igenom elementen i sorterad ordning.

Obalanserade binära sökträd

En möjlig implementation:

```
public class BinaryTree
  <A extends Comparable<? super A>> {

  private class TreeNode {
      A        contents;
      TreeNode left;   // null om vänster barn saknas.
      TreeNode right;  // null om höger barn saknas.

      TreeNode(A contents) {
          this.contents = contents;
      }
  }

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

Obalanserade binära sökträd

Iterativ kod:

```
public boolean member(A a) {
    TreeNode here = root;

    while (here != null) {
        int cmp = a.compareTo(here.contents);
        if      (cmp < 0) { here = here.left; }
        else if (cmp > 0) { here = here.right; }
        else return true;
    }

    return false;
}
```

Obalanserade binära sökträd

Rekursiv kod (varning för stack overflow):

```
public void insert(A a) {
    root = insert(a, root);
}

private TreeNode insert(A a, TreeNode n) {
    if (n == null) return new TreeNode(a);

    int cmp = a.compareTo(n.contents);
    if      (cmp < 0) n.left  = insert(a, n.left);
    else if (cmp > 0) n.right = insert(a, n.right);
    else n.contents = a;  // Skriver över.
    return n;
}
```

Obalanserade binära sökträd

Implementera

```
// En ordnad lista med trädets element.
public List<A> elements() {
   ...
}
```

Obalanserade binära sökträd

Inordertraversering:

```
public List<A> elements() {
    List<A> xs = new ArrayList<A>();
    copyElements(root, xs);
    return xs;
}

private void copyElements(TreeNode n, List<A> xs) {
    if (n == null) return;

    copyElements(n.left, xs);
    xs.add(n.contents);
    copyElements(n.right, xs);
}
```

Obalanserade binära sökträd

Hur tar man bort ett element? Förslag:

- Hitta nod som ska tas bort (om någon).
- Lätt om noden har max ett barn.
- Annars:
  Ta bort högra delträdets minsta elementet,
  använd som nodens innehåll.

Alternativ: Lat borttagning.

Obalanserade binära sökträd

```
public void delete(A a) { root = delete(a, root); }

private TreeNode delete(A a, TreeNode n) {
    if (n == null) return null;

    int cmp = a.compareTo(n.contents);
    if      (cmp < 0) n.left  = delete(a, n.left);
    else if (cmp > 0) n.right = delete(a, n.right);
    else if (n.right == null) return n.left;
    else if (n.left  == null) return n.right;
    else {
        n.contents = findMin(n.right);
        n.right    = deleteMin(n.right);
    }
    return n;
}
```

Obalanserade binära sökträd

Lätt att hitta minsta elementet:

```
// Precondition: Delträdet måste vara icketomt.
private A findMin(TreeNode n) {
    assert n != null;

    TreeNode here = n;

    while (here.left != null) {
        here = here.left;
    }

    return here.contents;
}
```

`deleteMin`: Övning.

Obalanserade binära sökträd

Värstafallstidskomplexitet:

- `elements`: $\Theta(\mathit{storlek})$.
- `findMin`, `deleteMin`: $\Theta(\mathit{höjd})$.
- `member`, `insert`, `delete`:
  $\Theta(\mathit{höjd})$, givet att jämförelser tar konstant tid.

Höjd:

- Värsta fallet: $\Theta(\mathit{storlek})$.
- Värsta fallet uppstår t ex om man
  sätter in elementen 1, 2, 3, 4, ….

Kan vi se till att värstafallshöjden är $\Theta(\log \mathit{storlek})$?

Balans

Balanserade sökträd

Träd som är balanserade,
med höjden $\Theta(\log \textit{storlek})$:

- AVL-träd (Adelson-Velskii & Landis).
- Röd-svarta träd.
- B^+-träd.

AVL-träd

- Invariant (för varje nod):
  Vänster och höger delträd har samma höjd, ± 1.
- Operationer som ändrar trädets struktur
  använder (ibland) *rotationer* för att
  återställa invarianten.

Sammanfattning

Idag:

- Repetition.
- Sökträd.

Nästa gång: Mer om sökträd.