Delvis kortfattade lösningsförslag för tentamen i Datastrukturer (DAT036) från 2013-12-16

Nils Anders Danielsson

1. Notera att `t` kommer att innehålla som mest `n` olika värden, så operationen `t.insert(j)` har amorterade tidskomplexiteten $O(\log \mathtt{n})$. Den här operationen utförs $\Theta(\mathtt{n}^2)$ gånger, och därför är värstafallstidskomplexiteten $O(\mathtt{n}^2 \log \mathtt{n})$.

   Man kan dock göra en noggrannare analys. Notera först att den inre loopen har samma asymptotiska tidskomplexitet som, och transformerar trädet på samma sätt som, följande kod, givet att `t` till att börja med innehåller elementen 0, 1, ..., $\mathtt{i} - 2$:

   ```
   for (int j = 0; j < i - 1; j++) {
       t.lookup(j);
   }
   t.insert(i-1);
   ```

   Tarjan visar i "Sequential access in splay trees takes linear time" att det tar linjär tid ($\Theta(\mathtt{i})$) att köra loopen ovan, givet antagandet om vilka element trädet innehåller. Den efterföljande insättningen tar också linjär tid ($O(\mathtt{i})$). Hela programmet har alltså tidskomplexiteten $\Theta(\mathtt{n}^2)$.

2. Haskellösning:

   ```
   -- reverseAppend xs ys är en lista som innehåller
   -- elementen i xs i omvänd ordning, och därefter
   -- elementen i ys. Tidskomplexitet: Θ(|xs|).

   reverseAppend :: [a] -> [a] -> [a]
   reverseAppend []       ys = ys
   reverseAppend (x : xs) ys = reverseAppend xs (x : ys)

   -- reverse xs är en lista som innehåller elementen i xs
   -- i omvänd ordning. Tidskomplexitet: Θ(|xs|).

   reverse :: [a] -> [a]
   reverse xs = reverseAppend xs []
   ```

Javalösning som utför konstant arbete per nod, och konstant övrigt arbete, och därför är linjär:

```
public void reverse() {
    Node current = first;
    while (current != null) {
        Node next    = current.next;
        current.next = current.prev;
        current.prev = next;
        current      = next;
    }
    Node temp = first;
    first     = last;
    last      = temp;
}
```

3. Låt datastrukturen bestå av tre komponenter:
   (a) Talet N.
   (b) Köns storlek n.
   (c) Ett AVL-träd `tree` som mappar prioriteter till *länkade listor* av värden. Trädet ska uppfylla följande invarianter:
   - Alla listor är icketomma.
   - Antalet listelement är n.
   - $n \leq N$.

   Pseudokod:

```
new Priority-queue(N):
  this.N = N
  n      = 0
  tree   = new empty AVL tree

insert(v, p):
  if the tree contains p then
    add v to the front of the list for p
  else
    insert (p, [v]) into the tree

  if n < N then
    increase n
  else
    if the tree node with maximum priority
         contains a list of length 1 then
      remove this node
    else
      remove the first element from this list
```

```
delete-min():
  assert n > 0

  decrease n

  n-min = the tree node with minimum priority
  vs    = the list in n-min
  v     = the head of vs
  if vs has length 1 then
    remove n-min from the tree
  else
    remove the first element from vs

  return v
```

Eftersom trädet innehåller p noder så kommer tidskomplexiteten för `insert` och `delete-min` att vara $O(\log p)$ (givet antagandet att prioriteter är heltal). Notera att analysen ovan inte gäller om noder tillåts innehålla tomma listor, eller om trädet innehåller en nod per element snarare än en nod per unik prioritet.

4. Uppgiftstexten specificerar inte om det rör sig om en minheap eller en maxheap. För en minheap med roten på position 1 (andra platsen) får vi följande arrayer:

   - | | 3 | | | | | |
   - | | 2 | 3 | | | | |
   - | | 2 | 3 | 5 | | | |
   - | | 2 | 3 | 5 | 4 | | |
   - | | 1 | 2 | 5 | 4 | 3 | |
   - | | 1 | 2 | 5 | 4 | 3 | 6 |

5. (a) Ett möjligt svar: 0, 1, 3, 2, 4, 5.

   (b) En möjlighet är att använda både en grannmatris (för att uppfylla krav i) och grannlistor (för krav ii). En annan möjlighet är att använda ”grannhashtabeller”, implementerade på ett sådant sätt att hashtabellernas lastfaktorer är större än en global konstant (så att alla direkta efterföljare kan räknas upp på linjär tid). Krav i uppfylls i så fall om hashfunktionerna är tillräckligt bra.

6. Både `insert bs t` och `member bs t` utför (som mest) konstant arbete för varje element i `bs`, och konstant övrigt arbete, och har därför tidskomplexiteten $O(|\texttt{bs}|)$, där $|\texttt{bs}|$ är längden av `bs`.