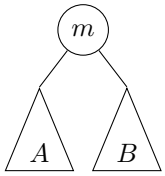
Kortfattade lösningsförslag för dugga i Datastrukturer (DAT036/DAT037) från 2015-11-27

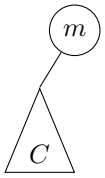
Nils Anders Danielsson

1. Notera att `q` hela tiden innehåller $\mathtt{n}^2$ eller $\mathtt{n}^2 - 1$ heltal. Tidskomplexiteten för varje leftistheapoperation är $O(\log(\mathtt{n}^2)) = O(\log \mathtt{n})$, så den totala tidskomplexiteten blir $O(\mathtt{n}^2 \log \mathtt{n})$.

   Vi kan dock göra en mer noggrann analys. Notera att `q` till att börja med har följande form:

   Om vi kör `delete-min` (tidskomplexitet: $O(\log \mathtt{n})$) får vi en ny kö genom att slå ihop A och B, låt oss kalla den C. (Den här kön är eventuellt tom.) Om vi sedan sätter in m får vi, på konstant tid, en ny kö D med följande form:

   Om vi kör `delete-min` igen får vi på konstant tid C. Efter första anropet till `delete-min` så kommer `q` alltså att växla mellan C och D, och varje köoperation (efter den första) tar konstant tid. Den totala tidskomplexiteten blir alltså $\Theta(\mathtt{n}^2)$.

2. *Haskellösning:* Anta att vi har tillgång till en FIFO-ködatastruktur med följande operationer, som alla tar amorterat konstant tid:

```
empty   :: Queue a
enqueue :: a -> Queue a -> Queue a
dequeue :: Queue a -> Maybe (a, Queue a)
```

I så fall kan uppgiften lösas på följande sätt:

```
-- Noderna i träden, i nivåordning: först alla rötter (med
-- början på första trädet i kön), sedan rötternas barn,
-- och så vidare.

levelOrder' :: Queue (Tree a) -> [a]
levelOrder' q = case dequeue q of
  Nothing     -> []
  Just (t, q) -> case t of
    Empty      -> levelOrder' q
    Node l x r ->
      x : levelOrder' (enqueue r (enqueue l q))

-- Noderna i trädet, i nivåordning.

levelOrder :: Tree a -> [a]
levelOrder t = levelOrder' (enqueue t empty)
```

Algoritmen utför amorterat konstant arbete per trädnod, och amorterat konstant övrigt arbete, så implementationen är linjär i trädets storlek.

Javalösning:

```
// Noderna i trädet, i nivåordning.
public List<A> levelOrder() {
    // En kö som kan hantera null-värden.
    Deque<Node> q  = new LinkedList<>();
    List<A>     ns = new ArrayList<>();

    q.addLast(root);

    while (! q.isEmpty()) {
        Node n = q.pollFirst();

        if (n != null) {
            ns.add(n.contents);
            q.addLast(n.left);
            q.addLast(n.right);
        }
    }

    return ns;
}
```

3. Tidskomplexiteterna som anges nedan är (ibland) amorterade.

   Använd två binära heapar, en minheap och en maxheap. I båda fallen ska heapen kombineras med en hashtabell som mappar värden till heappo-

sitioner, och tabellen ska uppdateras varje gång heapen ändras. Låt oss anta att hashfunktionerna är tillräckligt bra, så att de relevanta hashtabelloperationernas tidskomplexiteter är $O(1)$.

Prioritetsköoperationerna kan då implementeras på följande sätt:

- `empty`: Skapa två tomma heapar ($O(1)$).
- `insert`: Sätt in värdet och prioriteten i båda heaparna ($O(\log n)$).
- `delete-min`: Ta bort ett av värdena med lägst prioritet från minheapen ($O(\log n)$). Låt oss kalla värdet v. Använd ena hashtabellen för att hitta positionen för v i maxheapen ($O(1)$), och ta bort v: bubbla upp ($O(\log n)$), kör `delete-max` ($O(\log n)$). Total tidskomplexitet: $O(\log n)$.
- `delete-max`: På motsvarande sätt.

Alternativ lösning: Använd ett AVL-träd som mappar prioriteter till icketomma länkade listor med värden.