Lösningsförslag till tentamen

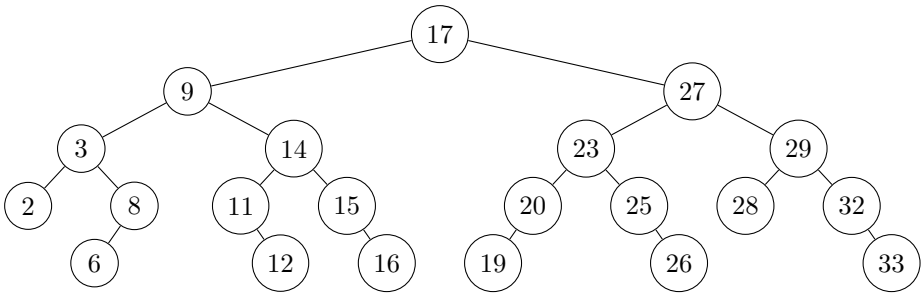
Datastrukturer, DAT037, 2018-04-05

1.
   - `q.dequeue()` tar $O(1)$ (eventuellt amorterat)
   - `s.contains(x)` tar $O(1)$
   - `pq.add(x)` tar $O(\log i)$

   I värsta fall exekveras innehållet i if-satsen. En iteration tar $O(1+1+\log i)$. Loopen itereras för $i = 0, 1, \ldots, \mathtt{n} - 1$.

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

2. Efter att ha satt in 26 får man följande träd:



3. För trea kan man spara elementen sist i en lista och söka linjärt efter minsta elementet enligt komparatorn.

   För högre betyg kan man använda en binär heap där elementens inbördes ordning förutom den generiska typen också avspeglar insättningsordningen.

   Låt representationen vara

```
Ett heltal t.
En prioritetskö q implementerad med en binär heap.
  Kön innehåller par av typen <E, int>.
  Ordningen, komparatorn, som används för jämförelser i kön jämför först det första
   komponenten i paren.
  Om de är olika är det denna ordning som gäller.
  Om de är lika är det omvända ordningen mellan andra komponenten i paren som gäller.
   (Större heltal innebär högre prioritet.)
```

   Konstruktorn och operationerna implementeras så här:

```
empty:
  t = 0  O(1)
  q = tom binär heap med komparator enligt beskr. ovan  O(1)

add(x):
  q.add(<x, t>)  O(log n)
  t = t + 1  O(1)

deleteNewestMin():
  p = q.deleteMin()  O(log n)
  returnera första komponenten i paret p (värdet av typen E)  O(1)
```

   Genom att definiera ordningen för kö-elementen på det sättet som beskrivs ovan och öka `t` vid varje insättning kommer det nyaste element tas bort när det finns fler än ett minsta element.

   Konstruktorn har tidskomplexiteten $O(1)$ och operationerna $O(\log n)$.

4. För att beräkna uppspännande skog kan man utföra en dfs varje besökt nod skapar en trädnod och varje omstart skapar ett nytt träd.

```
public List<TreeNode> spanningForest() {
    List<TreeNode> sf = new ArrayList<>();  // O(1)
    Set<Integer> visited = new HashSet<>();  // O(1)
    Iterator<Integer> i = adj.keySet().iterator();  // O(1)
    while (i.hasNext()) {  // O(1) (villkoret)
        int v = i.next();  // O(1)
        TreeNode t = dfs(visited, v);
        if (t != null) {
            sf.add(t);  // O(1) amorterat
        }
    }
    return sf;
}
private TreeNode dfs(Set<Integer> visited, int v) {
    if (visited.contains(v)) return null;  // O(1)
    visited.add(v);  // O(1)
    TreeNode t = new TreeNode();  // O(1)
    t.data = v;
    List<Integer> a = adj.get(v);  // O(1)
    for (int i = 0; i < a.size(); i++) {
        TreeNode st = dfs(visited, a.get(i));
        if (st != null) {
            t.children.add(st);  // O(1) amorterat
        }
    }
    return t;
}
```

Loopen i `spanningForest` upprepas V gånger. `dfs` anropas max $V + E$ gånger. Koden fr o m rad två i `dfs` utförs V gånger. Loopen i `dfs` utförs totalt E gånger eftersom varje gången exekveringen når loopen så är det för en ny nod. Tidskomplexiteten blir $O(V + E)$.

För fyra eller femma kan man implementera en preorder höger-till-vänster traversering av skogen. Detta ger en topologisk sortering för en uppspännande skog av en acyklisk graf.

```
static List<Integer> preorderRL(List<TreeNode> f) {
    List<Integer> l = new ArrayList<>();
    preorderRL(f, l);
    return l;
}
private static void preorderRL(List<TreeNode> f, List<Integer> l) {
    for (int i = f.size() - 1; i >= 0; i--) {
        l.add(f.get(i).data);
        preorderRL(f.get(i).children);
    }
}
```

5.
```
private static void msort(int[] x, int[] tmp, int l, int r) {
    if (r <= l) return;
    int m = l + (r - l) / 2;
    msort(x, tmp, l, m);
    msort(x, tmp, m + 1, r);
    merge(x, tmp, l, m + 1, r);
}

private static void merge(int[] x, int[] tmp, int l1, int l2, int r2) {
    int r1 = l2 - 1, i = l1, i1 = l1, i2 = l2;
    while (i1 <= r1 && i2 <= r2) {
     // repeated at most (r1 - l1 + 1) + (r2 - l2 + 1) = r2 - l1 + 1 times
        if (x[i1] <= x[i2]) tmp[i++] = x[i1++];
        else tmp[i++] = x[i2++];
    }
    while (i1 <= r1)  // repeated at most r1 - l1 + 1 = l2 - l1 times
        tmp[i++] = x[i1++];
    while (i2 <= r2)  // repeated at most r2 - l2 + 1 times
        tmp[i++] = x[i2++];
    for (int j = l1; j <= r2; j++)  // repeated r2 - l1 + 1 times
        x[j] = tmp[j];
}
```

Alla satser förutom looparna i `merge` tar konstant tid. De första tre looparna upprepas tillsammans $\texttt{r2}-\texttt{l1}+1$ gånger. Den sista loopen upprepas lika många gånger. Tidskomplexiteten är $O(\texttt{r2}-\texttt{l1}+1) = O(\texttt{r2}-\texttt{l1})$.

6. För trea kan man spara heltalen sorterat i en lista.

   För fyra eller femma kan man använda ett balanserat sökträd men inte lagra heltal i noderna utan intervall (två heltal som representerar det minsta och största talet i intervallet). Ett intervall är mindre än ett annat om talen som det innehåller är mindre än talen som det andra intervallet innehåller. Inget tal kommer att finnas med i mer än ett intervall och inget intervall kommer vara tomt.

   Använd en hjälpklass för intervall:

```
class Interval {
    int first, last;
}
```

   Definiera Konstruktorn och operationerna så här.

```
empty() =
 t = tomt balanserat sökträd med talpar som element ordnade enligt ovan  O(1)

add(x) =
 Interval i = find(x);  // O(log n)
 if (i != null) return;
 Interval il = find(x - 1);  // O(log n)
 Interval ir = find(x + 1);  // O(log n)
 if (il != null) t.remove(il);  // O(log n)
 if (ir != null) t.remove(ir);  // O(log n)
 Interval newi = new Interval();
 newi.first = il != null ? il.first : x;
 newi.last = ir != null ? ir.last : x;
 t.add(newi);  // O(log n)

contains(x) =
 return find(x) != null;  // O(log n)

nextNonMember(x) =
 Interval i = find(x);  // O(log n)
 if (i == null) return x;
 return i.last + 1;
```

   `t.remove` och `t.add` är standardborttagning och -insättning för det balanserade sökträdet.

`find(x)` är en hjälpmetod som avgör om heltalet `x` finns i någon av intervallen i `t`. Finns talet med så returneras det intervall som det ingår i, annars returneras `null`.

```
find(x) =  O(log n)
 n = t.root;  // trädets rotnod
 while (n != null) {  // O(log n)
     if (n.data.first <= x && n.data.last >= x) {
         return n.data;
     } else if (n.data.first > x) {
         n = n.left;
     } else {
         n = n.right;
     }
 }
 return null;
```

Koden ovan utgår från att `t.root` utgör rotnoden i trädet och att noderna tillhör följande klass:

```
class Node {
    Interval data;
    Node left, right;
}
```

Varje iteration av loopen i `find` tar $O(1)$. Loopen upprepas $O(\log n)$ gånger eftersom trädet är balanserat.