Tentamen
Datastrukturer (DAT036)

- Datum och tid för tentamen: 2013-12-16, 14:00–18:00.
- Ansvarig: Nils Anders Danielsson. Nås på 0700 620 602 eller anknytning 1680. Besöker tentamenssalarna ca 15:00 och ca 17:00.
- Godkända hjälpmedel: Ett A4-blad med handskrivna anteckningar.
- För att få betyget n (3, 4 eller 5) måste du lösa minst n uppgifter. Om en viss uppgift har deluppgifter med olika gradering så räknas uppgiften som löst om du har löst *alla* deluppgifter med gradering n eller mindre.
- För att en uppgift ska räknas som ”löst” så måste en i princip helt korrekt lösning lämnas in. Enstaka mindre allvarliga misstag kan *eventuellt* godkännas (kanske efter en helhetsbedömning av tentan; för högre betyg kan kraven vara hårdare).
- Lämna inte in lösningar för flera uppgifter på samma blad.
- Skriv din tentakod på varje blad.
- Lösningar kan underkännas om de är svårlästa, ostrukturerade eller dåligt motiverade. Pseudokod får gärna användas, men den får inte utelämna för många detaljer.
- Om inget annat anges så kan du använda kursens uniforma kostnadsmodell när du analyserar tidskomplexitet (så länge resultaten inte blir uppenbart orimliga).
- Om inget annat anges behöver du inte förklara standarddatastrukturer och -algoritmer från kursen, men däremot motivera deras användning.

1. Analysera nedanstående kods värstafallstidskomplexitet, uttryckt i `n`:

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

   Använd kursens uniforma kostnadsmodell, och gör följande antaganden:

   - Att `n` är ett ickenegativt heltal, och att typen `int` kan representera alla heltal.
   - Att `t` är ett splayträd som till att börja med är tomt.
   - Att den vanliga ordningen för heltal ($... < -1 < 0 < 1 < 2 < ...$) används vid insättning i trädet.
   - Att om man sätter in ett värde som redan finns i trädet så byts det gamla värdet ut mot det nya.

   Svara med ett enkelt uttryck (inte en summa, rekursiv definition, eller dylikt). Onödigt oprecisa analyser kan underkännas.

2. Implementera en metod/funktion `reverse` som reverserar en lista. Operationen får (men måste inte) förstöra den ursprungliga listan.

   Exempel: Resultatet av att reversera $[1, 2, 3]$ ska vara $[3, 2, 1]$, och resultatet av att reversera $[\,]$ ska vara $[\,]$.

   Du kan representera listor på ett av följande sätt:

   - Som vanliga Haskellistor.
   - Med följande Javaklass:

```
public class DoublyLinkedList<A> {
    private class Node {
        public A contents;  // Nodens innehåll.
        public Node next;   // Nästa nod; null omm noden
                            // är sista vaktposten.
        public Node prev;   // Föregående nod; null omm
                            // noden är första
                            // vaktposten.

        public Node() {}
    }

    private Node first;  // Första vaktposten.
    private Node last;   // Sista vaktposten.

    public DoublyLinkedList() {
        first      = new Node();
        last       = new Node();
        first.next = last;
        last.prev  = first;
    }

    public void reverse() {
        // Din uppgift.
    }
}
```

   Endast detaljerad kod (inte nödvändigtvis Haskell eller Java) godkänns. Bortsett från listkonstruerarna får du inte anropa andra procedurer e d (inkl standardfunktioner som `(++)`), om du inte implementerar dem själv som en del av din lösning.

   Operationen måste uppfylla följande tidskomplexitetskrav, där n är listans längd:

   - *För trea:* $O(n^2)$.
   - *För fyra:* $O(n)$.

   Tips: Testa din kod, så kanske du undviker onödiga fel.

3. Uppgiften är att konstruera en datastruktur för en prioritetskö-ADT med följande operationer:

   `empty(`N`)` eller `new Priority-queue(`N`)` Konstruerar en tom kö. Kön får inte innehålla fler än N värden. *Precondition*: $N \geq 0$.

   `insert(`v`, `p`)` Sätter in värdet v, med tillhörande prioritet p. Om kön då innehåller fler än N värden så tas värdet med *högst* prioritet bort (eller, om det finns flera värden med högst prioritet, ett av dem).

   `delete-min()` Tar bort och ger tillbaka värdet med lägst prioritet (eller, om det finns flera värden med lägst prioritet, ett av dem). *Precondition*: Kön får inte vara tom.

   Du kan anta att prioriteter är heltal.

   Exempel: Följande kod, skriven i ett Javaliknande språk, ska ge `true` som svar:

```
Priority-queue q = new Priority-queue(2);
q.insert('a', 2);
q.insert('b', 1);
q.insert('c', 2);
char c1 = q.delete-min();
char c2 = q.delete-min();
q.insert('d', 3);
q.insert('d', 3);
char c3 = q.delete-min();
char c4 = q.delete-min();
return c1 == 'b' && (c2 == 'a' || c2 == 'c') &&
       c3 == 'd' && c4 == 'd';
```

   Du måste visa att operationerna uppfyller följande tidskomplexitetskrav:

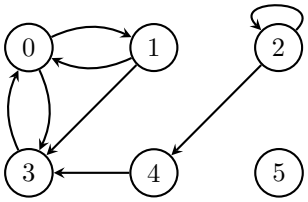
   - *För trea:* `empty`: $O(1)$, `insert`: $O(n)$, `delete-min`: $O(\log n)$, där n är köns storlek.
   - *För fyra:* `empty`: $O(1)$, `insert`, `delete-min`: $O(\log n)$, där n är köns storlek.
   - *För femma:* `empty`: $O(1)$, `insert`, `delete-min`: $O(\log p)$, där p är antalet *olika* prioriteter i kön.

   Implementationen av datastrukturen behöver inte beskrivas med detaljerad kod, men lösningen måste innehålla så mycket detaljer att tidskomplexiteten kan analyseras.

   Tips: Konstruera om möjligt inte datastrukturen från grunden, bygg den hellre m h a standarddatastrukturer. Testa dina algoritmer, så kanske du undviker onödiga fel.

4. Sätt in värdena 3, 2, 5, 4, 1, 6, i den givna ordningen, i en från början tom binär heap. Heapen ska vara implementerad med en array av längd 7. Visa hur arrayen ser ut efter varje insättning.

5. (a) *För trea:* Betrakta följande graf:



Börja med en tom lista. Besök sedan alla grafens noder i djupet först-ordning, och lägg dem sist i listan när de besöks första gången. Svara med den slutgiltiga listan.

(b) *För fyra:* Uppgiften är att konstruera en datastruktur som representerar riktade, oviktade grafer, och uppfyller följande krav:

i. Givet två noder u, v ska det gå att avgöra, på konstant tid, om det finns en kant från u till v.

ii. Givet en nod v ska det gå att räkna upp nodens alla direkta efterföljare på linjär tid ($O(n)$, där n är antalet direkta efterföljare).

Visa att kraven ovan uppfylls.

Implementationen av datastrukturen behöver inte beskrivas med detaljerad kod, men lösningen måste innehålla så mycket detaljer att tidskomplexiteten kan analyseras.

Tips: Konstruera om möjligt inte datastrukturen från grunden, bygg den hellre m h a standarddatastrukturer.

6. Betrakta följande Haskellimplementation av binära prefixträd (en datastruktur som vi inte gått igenom tidigare i kursen):

```
-- Binära prefixträd (tries).
data Trie = Node Bool Trie Trie
          | Empty

-- Ett tomt prefixträd.
empty :: Trie
empty = Empty

-- Finns den tomma listan i trädet?
root :: Trie -> Bool
root Empty        = False
root (Node b _ _) = b

-- Vänster delträd.
left :: Trie -> Trie
left Empty        = Empty
left (Node _ l _) = l

-- Höger delträd.
right :: Trie -> Trie
right Empty        = Empty
right (Node _ _ r) = r

-- Sätter in listan i trädet.
insert :: [Bool] -> Trie -> Trie
insert []           t = Node True (left t) (right t)
insert (True  : bs) t = Node (root t) (insert bs (left t))
                                      (right t)
insert (False : bs) t = Node (root t) (left t)
                                      (insert bs (right t))

-- Avgör om listan finns i trädet.
member :: [Bool] -> Trie -> Bool
member _            Empty        = False
member []           (Node b _ _) = b
member (True  : bs) (Node _ l _) = member bs l
member (False : bs) (Node _ _ r) = member bs r
```

Analysera följande uttrycks tidskomplexitet: `insert bs t`, `member bs t`.

Svara med ett enkelt uttryck (inte en summa, rekursiv definition, eller dylikt). Onödigt oprecisa analyser kan underkännas. Använd inte variabeln n i ditt svar utan att definiera vad n står för.

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`.