Kortfattade lösningsförslag för tentamen i Datastrukturer (DAT036) från 2014-04-25

Nils Anders Danielsson

1. Proceduren utför konstant arbete för varje nod, och konstant arbete för varje kant, så tidskomplexiteten blir $\Theta(|V| + |E|)$.

2. Haskellösning:

```
deleteAll :: Int -> [Int] -> [Int]
deleteAll i []       = []
deleteAll i (j : js) = if i == j then js' else j : js'
  where js' = deleteAll i js
```

   Rekursiv Javalösning (varning för ”stack overflow”):

```
public void deleteAll(int i) {
    first = deleteAll(i, first);
}

private Node deleteAll(int i, Node n) {
    if (n == null) {
        return n;
    }

    n.next = deleteAll(i, n.next);

    if (n.contents == i) {
        return n.next;
    } else {
        return n;
    }
}
```

Iterativ Javalösning:

```
public void deleteAll(int i) {
    Node current  = first;
    Node previous = null;

    while (current != null) {

        if (current.contents == i) {
            if (previous == null) {
                // Specialfall för borttagande av
                // listans första element.
                first = current.next;
            } else {
                previous.next = current.next;
            }
        } else {
            previous = current;
        }

        current = current.next;
    }
}
```

Alla implementationerna utför konstant arbete per nod, och konstant övrigt arbete, och är därför linjära.

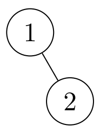
3. *För fyra:* Använd AVL-träd, och implementera `new`, `insert`, `delete` och `member` på vanligt sätt. Implementera `delete-min-max` genom att ta bort elementet längst till vänster (om det finns något) och sedan elementet längst till höger (om det finns något).

   För femma: Använd dessutom en hashtabell (med en tillräckligt bra hashfunktion). Använd hashtabellen för att implementera `member`. Då trädet uppdateras, uppdatera hashtabellen på motsvarande sätt; för `delete-min-max`, använd information från trädet för att avgöra vilket eller vilka element som ska tas bort.

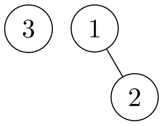
4. 1:

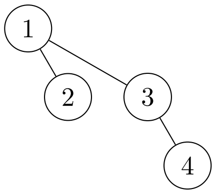
   2:

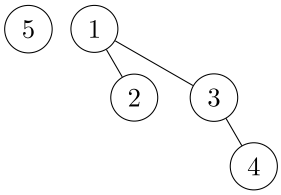
3:



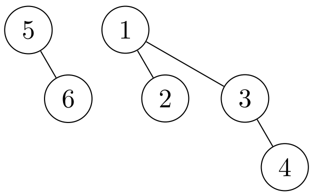
4:



5:



6:



5. (a) 4 stycken: $\{0, 1, 4\}$, $\{2, 3\}$, $\{5\}$, $\{6\}$.

   (b) Representera grafen på följande sätt:

   - Ett heltal `n`, antalet noder i grafen. (Låt oss för enkelhets skull anta att noderna är numrerade från 0 till $\texttt{n} - 1$.)
   - En array `component` av storlek `n`, innehållandes komponentnummer: för noder u och v ska vi ha $\texttt{component}[u] = \texttt{component}[v]$ omm u och v hör till samma komponent.
   - En grannmatris av storlek $\texttt{n} \times \texttt{n}$.

   Arrayen `component` gör det lätt att uppfylla krav i, och grannmatrisen gör det lätt att uppfylla krav ii.

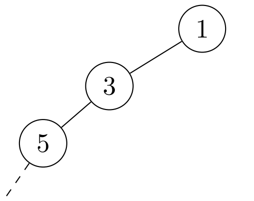
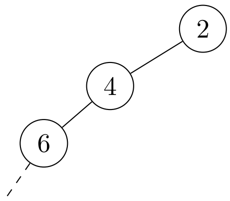
   För att lägga till en (ny) kant från u till v kan man använda följande (ickeoptimerade) algoritm:

   - Uppdatera grannmatrisen ($O(1)$).
   - Ersätt `component`-arrayen med en ny, beräknad med (en variant av) SCC-algoritmen som gicks igenom i kursen. Notera att i och med att en grannmatris används så blir tidskomplexiteten för djupet först-sökningarna och SCC-algoritmen $\Theta(\texttt{n}^2)$.

   Total tidskomplexitet: $\Theta(\texttt{n}^2)$.

6. Summan av de två trädens höjder minskar med minst ett i varje rekursivt `merge`-anrop. Operationen `merge` anropas alltså $O(h_{\mathtt{l}} + h_{\mathtt{r}})$ gånger, där h_t är höjden hos trädet t. Varje `merge`-anrop tar, bortsett från rekursiva `merge`-anrop, konstant tid, så tidskomplexiteten är $O(h_{\mathtt{l}} + h_{\mathtt{r}})$.

   Notera att för par av träd med följande struktur och nodinnehåll är tidskomplexiteten $\Theta(h_{\mathtt{l}} + h_{\mathtt{r}})$, givet att den vanliga ordningen för naturliga tal används:

   `l`:  `r`: 

   Tidskomplexiteten för den här sortens trädpar är också $\Theta(s_{\mathtt{l}} + s_{\mathtt{r}})$, där s_t är antalet noder hos trädet t. Trots detta kan man ge `merge l r` den *amorterade* tidskomplexiteten $O(\log s_{\mathtt{l}} + \log s_{\mathtt{r}})$; se t ex Okasakis ”Fun with binary heap trees”.