

Løsningsforslag eksamen INF1020 høsten 2005

Merk at dette er et løsningsforslag på selve oppgavene, og ikke slik vi forventer at en besvarelse skal se ut. Dette gjelder spesielt oppgave 3.

Oppgave 1: Flervalgsoppgaver

1. A, B, C
2. C
3. A, B, D, E
4. B, E
5. B, C, D
6. D
7. A, B
8. A, B, C
9. E (20)
10. C

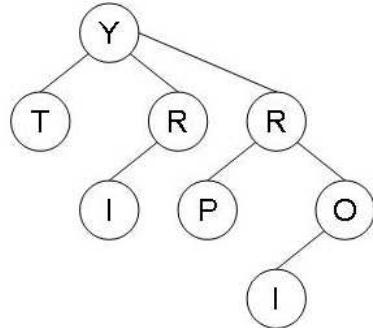
Oppgave 2: Heap

Oppgave 2a

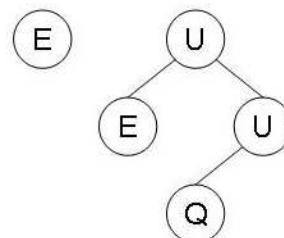
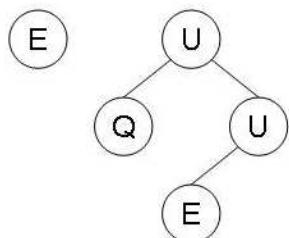
R R P O T Y I I U Q E U

Oppgave 2b

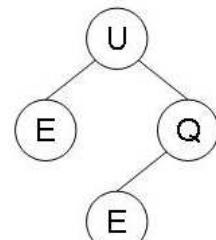
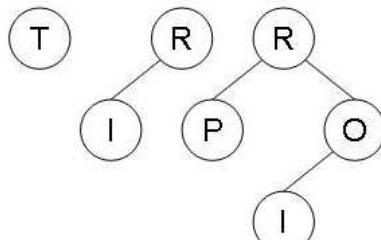
1. For PRIORITY:



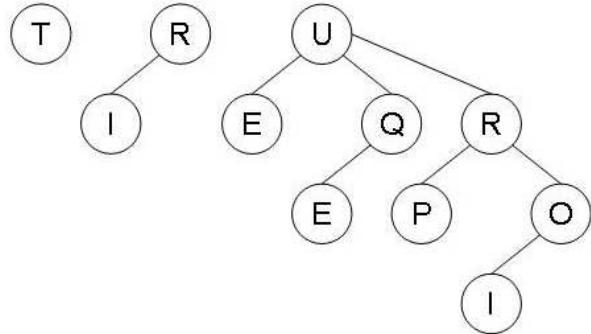
For QUEUE er det to muligheter, der den til venstre vel er den som tilsvarer implementasjonen i læreboken (bruker denne videre i forslaget):



2.



3.

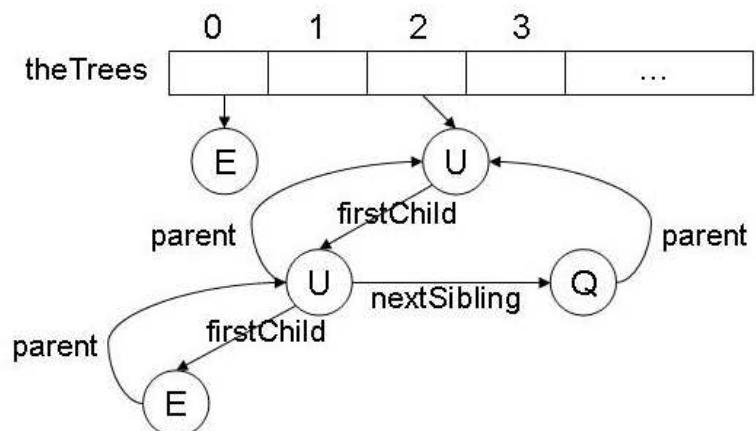


Oppgave 2c

Bruker lærebokens variant, med tillegg av foreldre-peker (forenkler oppgave 2e):

```
class BinomialQueue {  
    BinomialNode[] theTrees;  
}  
  
class BinomialNode {  
    char elem;  
    BinomialNode parent;  
    BinomialNode firstChild;  
    BinomialNode nextSibling;  
}
```

Den første av QUEUE-køene fra oppgave 2b.1, vil da se slik ut (tegner ikke null-peker):



Oppgave 2d

Med nøyaktig $N = 2^k$ elementer, har vi bare ett tre i binomial-køen. Finner indeksen til dette og bruker koden i figur 6.54 (utvidet med oppdatering av foreldre-peker) for å slå sammen de to trærne.

```
BinomialQueue merge(BinomialQueue q1, BinomialQueue q2) {
    int index = 0;
    while (q1.theTrees[index] == null) {
        index++;
    }

    // Antar at k1-arrayen er stor nok til den nye køen.
    q1.theTrees[index+1] = merge2(q1.theTrees[index], q2.theTrees[index];
    q1.theTrees[index] = null;
    return q1;
}

BinomialNode merge2(BinomialNode t1, BinomialNode t2) {
    if (t1.elem < t2.elem) {
        return merge(t2, t1);
    } else {
        t2.nextSibling = t1.firstChild;
        t1.firstChild = t2;
        t2.parent = t1;
        return t1;
    }
}
```

Oppgave 2e

1. Ide: Hvis den nye prioriteten er større, bytter vi plass med forelder så lenge denne er mindre. Hvis derimot den nye prioriteten er mindre, bytter vi plass med største barn så lenge denne er større.

```
void changePriority(BinomialNode bn, char newPri) {
    if (newPri > bn.elem) {
        bn.elem = newPri;
        moveUp(bn);
    } else {
        bn.elem = newPri;
        moveDown(bn);
    }
}
```

```

void moveUp(BinomialNode bn) {
    BinomialNode n = bn;
    while (n.parent != null && n.parent.elem < n.elem) {
        changeElem(n, n.parent);
        n = n.parent;
    }
}

void moveDown(BinomialNode bn) {
    BinomialNode n = bn;
    BinomialNode largestChild = findLargest(n.firstChild);
    while (largestChild != null && largestChild.elem > n.elem) {
        changeElem(largestChild,n);
        n = largestChild;
        largestChild = findLargest(n.firstChild);
    }
}

BinomialNode findLargest(BinomialNode bn) {
    if (bn == null) return null;

    BinomialNode largest = bn;
    BinomialNode n = bn.nextSibling;

    while (n != null) {
        if (n.elem > largest.elem)
            largest = n;
        n = n.nextSibling;
    }

    return largest;
}

void changeElem(BinomialNode b1, BinomialNode b2) {
    char tmp = b1.elem;
    b1.elem = b2.elem;
    b2.elem = tmp;
}

```

2. Ide: Endrer prioriteten til det elementet som skal fjernes slik at det blir det første som fjernes av deleteMax.

```

void remove(BinomialNode bn) {
    changePriority(bn, Character.MAX_VALUE);
    deleteMax();
}

```

Oppgave 2f

Ide: Løper gjennom alle nodene i binomial-køen og setter disse løende inn i heaparrayen. Dette tar $O(n)$ tid. Kaller så buildheap på binærheopen for å ordne disse elementene. Dette tar $O(n)$ tid. Total tar dermed hele konverteringen $O(n)$ tid.

Antar at binærheopen er definert som i seksjon 6.3 i læreboken, men at arrayen kan aksesseres utenfra.

```
BinaryHeap convert(BinomialQueue bq) {
    BinaryHeap bh = new BinaryHeap();
    int nextIndex = 0;

    for (int i = 0; i < bq.theTrees.length; i++) {
        nextIndex = convert(bq.theTrees[i], bh, nextIndex);
    }

    bh.buildheap();
    return bh;
}

int convert(BinomialNode bn, BinaryHeap bh, int index) {
    int i = index;

    if (bn != null) {
        bh.array[i++] = bn.elem;
        i = convert(bn.nextSibling, bh, i);
        i = convert(bn.firstChild, bh, i);
    }

    return i;
}
```

Oppgave 3: Grafer

```
Node[] graf;

int teller; // Til bruk i oppgave 3c

class Node {
    String navn;
    LinkedList utkanter; // Liste med Kant-objekter

    boolean kjent; // Til bruk i oppgave 3b
    int avstand, rute; // Til bruk i oppgave 3b
    Node vei; // Til bruk i oppgave 3b
}
```

```

class Kant {
    Node fra, til;
    int rutenummer, reisetid;

    boolean kjent; // Til bruk i oppgave 3c
    int lav, nummer; // Til bruk i oppgave 3c
    Kant forelder; // Til bruk i oppgave 3c
}

```

Oppgave 3b

Som eksempelet i oppgaveteksten viser er det ikke mulig å bruke Dijkstras algoritme rett frem. Først når kanten D-E velges, finner man hvilken vei som bør velges frem til D (som IKKE er den samme som om man skulle stoppet på D). Følgende kode legger derfor alle mulige veier (som bare bruker kjente noder) frem til D inn i prioritetskøen, mens avstanden til D oppdateres som i vanlig Dijkstra.

Bruker BinaryHeap og LinkedList fra læreboken (kapittel 6 og 3). Elementene som puttes inn i køen er av typen QueueNode:

```

class QueueNode implements Comparable {
    Node node;
    int avstand;
    Node vei;
    int rute;

    public int compareTo(Object n) {
        return avstand - ((QueueNode) n).avstand;
    }
}

```

For hver runde i while-løkken under er det to muligheter:

1. qn.node var ukjent, men blir nå satt kjent, med avstand og rute satt til korteste reiserute fra fra-noden. Sjekker ukjente nabonoder som for vanlig Dijkstra.
2. qn.node var allerede kjent, vi har en alternativ reiserute til denne, og må sjekke om denne gir en forbedring i forhold til de ukjente nabonodene. Siden kønodene settes i en prioritetskø, kan vi ikke ha noen forbedring for kjente nabonoder.

```

void finnRute(Node fra, Node til) {
    BinaryHeap bh = new BinaryHeap();
    Node tilNode, v;
    Kant k;
    LinkedListItr it;
    int avstand;
}

```

```

// Lager en kønode tilsvarende fra-noden.
QueueNode qn = new QueueNode();
qn.node = fra;
qn.avstand = 0;
fra.avstand = 0;

while (qn != null) {
    avstand = qn.avstand;
    v = qn.node;
    v.kjent = true;
    it = v.utkanter.first();
    k = (Kant) it.retrieve();
    while (k != null) {
        tilNode = (k.til == v) ? k.fra : k.til;
        if (!tilNode.kjent) {
            // Legger til 5 hvis rutebytte.
            if (v != fra && k.rutenummer != qn.rute) {
                avstand += 5;
            }
        }

        // Har nytt forslag til reiserute til tilNode,
        // legger denne inn i prioritetskøen.
        qn = new QueueNode();
        qn.node = tilNode;
        qn.avstand = avstand + k.reisetid;
        qn.vei = v;
        qn.rute = k.rutenummer;
        bh.insert(qn);

        // Vanlig Dijkstra, oppdaterer node-informasjonen
        // hvis vi har funnet en forbedring.
        if (avstand + k.reisetid < tilNode.avstand) {
            tilNode.avstand = avstand + k.reisetid;
            tilNode.vei = v;
            tilNode.rute = k.rutenummer;
        }
    }
    it.advance();
    k = (Kant) it.retrieve();
}
qn = (QueueNode) bh.deleteMin();
}

skrivRute(til, til.rute);
System.out.println(".");
System.out.println("Antall total reisetid: " + til.avstand + " minutter.");
}

```

```

void skrivRute(Node n, int rute) {
    if (n.vei == null) {
        System.out.print("Ta linje " + rute + " fra " + n.navn);
    }
    if (n.vei != null) {
        skrivRute(n.vei, n.rute);
        System.out.print(" til ");
        System.out.print(n.navn);
        if (rute != n.rute) {
            System.out.println(".");
            System.out.println("Bytt til linje " + rute + ".");
            System.out.print("Ta linje " + rute + " fra " + n.navn);
        }
    }
}

```

Oppgave 3c

Dette kan gjøres på flere måter. Et alternativ er å behandle kanter som noder og noder som kanter, slik at man kan bruke algoritmen for å finne artikulasjonspunkter (kapittel 9.6.2 i læreboken):

```

void finnStrekninger(Kant k) {
    k.kjent = true;
    k.lav = k.nummer = teller++;
    sjekk(k, k.fra);
    sjekk(k, k.til);
}

void sjekk(Kant k, Node v) {
    LinkedListItr it = v.utkanter.first();
    Kant l;

    l = (Kant) it.retrieve();

    while (l != null) {
        if (l != k) {
            if (!l.kjent) {
                l.forelder = k;
                finnStrekninger(l);
                if (l.lav >= k.nummer) {
                    System.out.println("Kanten mellom " + k.fra.navn +
                        " og " + k.til.navn +
                        " er en sårbar strekning.");
                }
                if (l.lav < k.lav) {
                    k.lav = l.lav;
                }
            }
        } else {

```

```
        if (k.forelder != l) {
            if (l.nummer < k.lav) {
                k.lav = l.nummer;
            }
        }
    }
    it.advance();
    l = (Kant) it.retrieve();
}
}
```