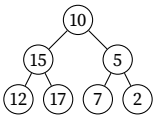
Level-order 12 poeng

En *level-order*-traversering besøker nodene i et tre etter *dybde*. Det vil si at roten besøkes først, så alle barna til roten, så alle barna til rotens barn, og så videre.

En *level-order*-traversering av følgende tre kan, for eksempel, besøke nodene i rekkefølgen $10, 15, 5, 12, 17, 7, 2$.



Merk at output for algoritmene under er en *liste*, men det er en liten detalj som ikke er vesentlig for *algoritmene*. Vi trekker ikke poeng for dette alene, men det kan være med på å trekke i samspill med andre unøyaktigheter.

(a) Anta at treet er en *binær heap*, representert ved et array A med n elementer. Gi en algoritme som returnerer en liste med elementene fra A i *level-order*.

```
Input: En binær heap, representert ved et array A med n elementer
Output: En liste med elementene i heapen i level-order
1 Procedure LevelOrderHeap(A)
  | // ···
```

En heap er allerede i *level-order*, så det er tilstrekkelig å returnere heapen som en liste.

```
1 Procedure LevelOrderHeap(A)
2 |  list ← an empty list
3 |  for i ← 0 to n − 1 do
4 |  |  append A[i] to list
5 |  return list
```

- Inntil 2 poeng.

(b) Anta at treet er et *binært søketre* med n noder, der hver node v har følgende felter:

- v.`element` — verdien som er lagret i noden
- v.`left` — venstre barn av v
- v.`right` — høyre barn av v

Gi en algoritme som returnerer en liste med elementene fra treet i *level-order*.

```
Input: Roten v av et binært søketre med n noder
Output: En liste med elementene i søketreet i level-order
1 Procedure LevelOrderBST(v)
  | // ···
```

Dette gjøres enklest ved hjelp av et bredde-først søk fra rotnoden. Siden input er et tre er det ikke nødvendig å bruke et `visited`-set.

```
1  Procedure LevelOrderBST(v)
2      list ← an empty list
3      queue ← an empty queue
4      if v ≠ null then
5          queue.enqueue(v)
6      while queue is not empty do
7          u ← queue.dequeue()
8          append u.element to list
9          if u.left ≠ null then
10             queue.enqueue(u.left)
11         if u.right ≠ null then
12             queue.enqueue(u.right)
13     return list
```

- Inntil 6 poeng.
- Inntil 2 poeng trekk for manglende **`null`**-sjekker (altså, ikke ta høyde for at en node mangler venstre/høyre barn).
- Trekk 3 poeng for bruk av stack i stedet for kø (som gir dybde først, i stedet for bredde først).
- Vi trekker ikke for overflødig bruk av `visited`-set (men det trekker litt ned på helhetsinntrykket).

(c) Oppgi kjøretidskompleksiteten for algoritmene dine i (a) og (b).

Begge er $\mathcal{O}(n)$, siden de flytter elementene over i en liste.

- Inntil 1 poeng for riktig kjørekompleksitet på begge algoritmer.

(d) Se på følgende algoritme:

```
Input: Roten v av et binært søketre
Output: Et nytt binært søketre
1  Procedure ConstructBST(v)
2      B ← empty binary search tree
3      nodes ← LevelOrderBST(v)
4      for u in nodes do
5          B ← Insert(B, u)
6      return B
```

Hva kan du si om relasjonen mellom input-treet og output-treet?

Input-treet og output-treet er *like*. Roten vil plasseres først, så vil dens to barn plasseres, og deretter deres barn og så videre.

- Inntil 3 poeng for riktig svar.
- Oppgaven krever ikke en begrunnelse.