Huffman-trær

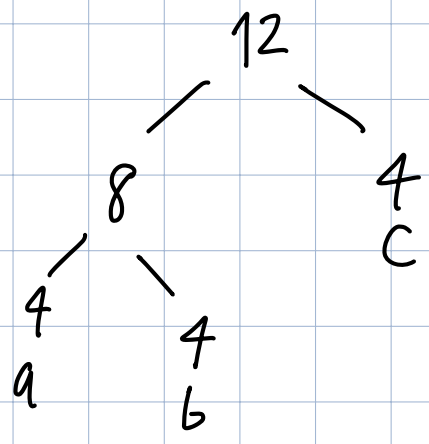
Du har nå sett algoritmen som tar en mengde `C` med par `(s, f)` som input (der `s` er et symbol og `f` er den tilhørende frekvensen) og gir et Huffman-tre som output. I denne oppgaven skal du skrive en algoritme som reverserer denne prosessen; altså én som tar et Huffman-tre som input og gir en mengde `C` med par `(s, f)` som output.

(a) La hver node `v` i et Huffman-tre ha følgende felt:

- `v.symbol` – nodens symbol (eller `null`)
- `v.freq` – nodens frekvens
- `v.left` – nodens venstre barn (eller `null`)
- `v.right` – nodens høyre barn (eller `null`)

Skriv en algoritme med signaturen som er vist nedenfor. Et prosedyrekall `HuffmanLeaves(root, {})`, altså med treets rot og en tom mengde som argumenter, skal returnere en mengde med par `(s, f)`, der `s` er et symbol og `f` er den tilhørende frekvensen.

```
proc HuffmanLeaves(v, C)
    // ...
```



{(a,4),(b,4),(c,4)}

HuffmanLeaves(, {(a,4)})

{(a,4),(b,4)}

```
proc HuffmanLeaves(tree, tuples):
    if tree.symbol = null:
        HuffmanLeaves(tree.left, tuples)
        HuffmanLeaves(tree.right, tuples)
    else:
        add (tree.symbol, tree.freq) to tuples
    return tuples
```

Kompleksitet: O(n)

fordi vi går gjennom hver node én gang

Binære heaps

Du har nå sett hvordan binære heaps kan representeres som arrayer. Et annet alternativ er å representere dem som trær. Utfordringen er å finne en effektiv måte å få tilgang til den siste foreldernoden i treet, og det er dette problemet du nå skal få bryne deg på.

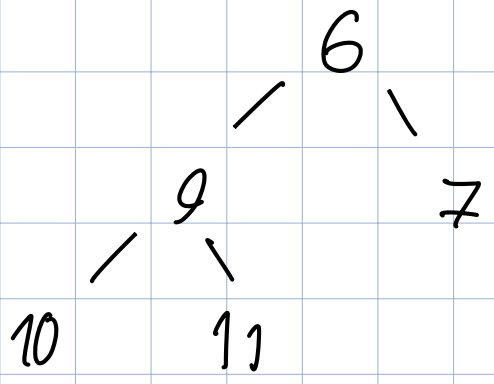
Implementér en binær heap som et tre med noder og kanter i det programmeringsspråket du foretrekker. Algoritmene for innsetting og sletting skal ha logaritmisk kjøretidskompleksitet. Begge fordrer tilgang til den siste foreldernoden i treet. Det finnes flere mulige tilnærminger, og du står fritt til å velge den som faller mest naturlig for deg.

```
class Heap:
  Heap(x):
    this.head ← Node(x)
    this.size ← 1

  class Node:
    Node(x):
      this.elem ← x
      this.left ← null
      this.right ← null
      this.parent ← null

    method bubbleUp():
      if this.parent = null:
        return
      if this.elem < this.parent.elem:
        this.elem, this.parent.elem
            ← this.parent.elem, this.elem
        this.parent.bubbleUp()

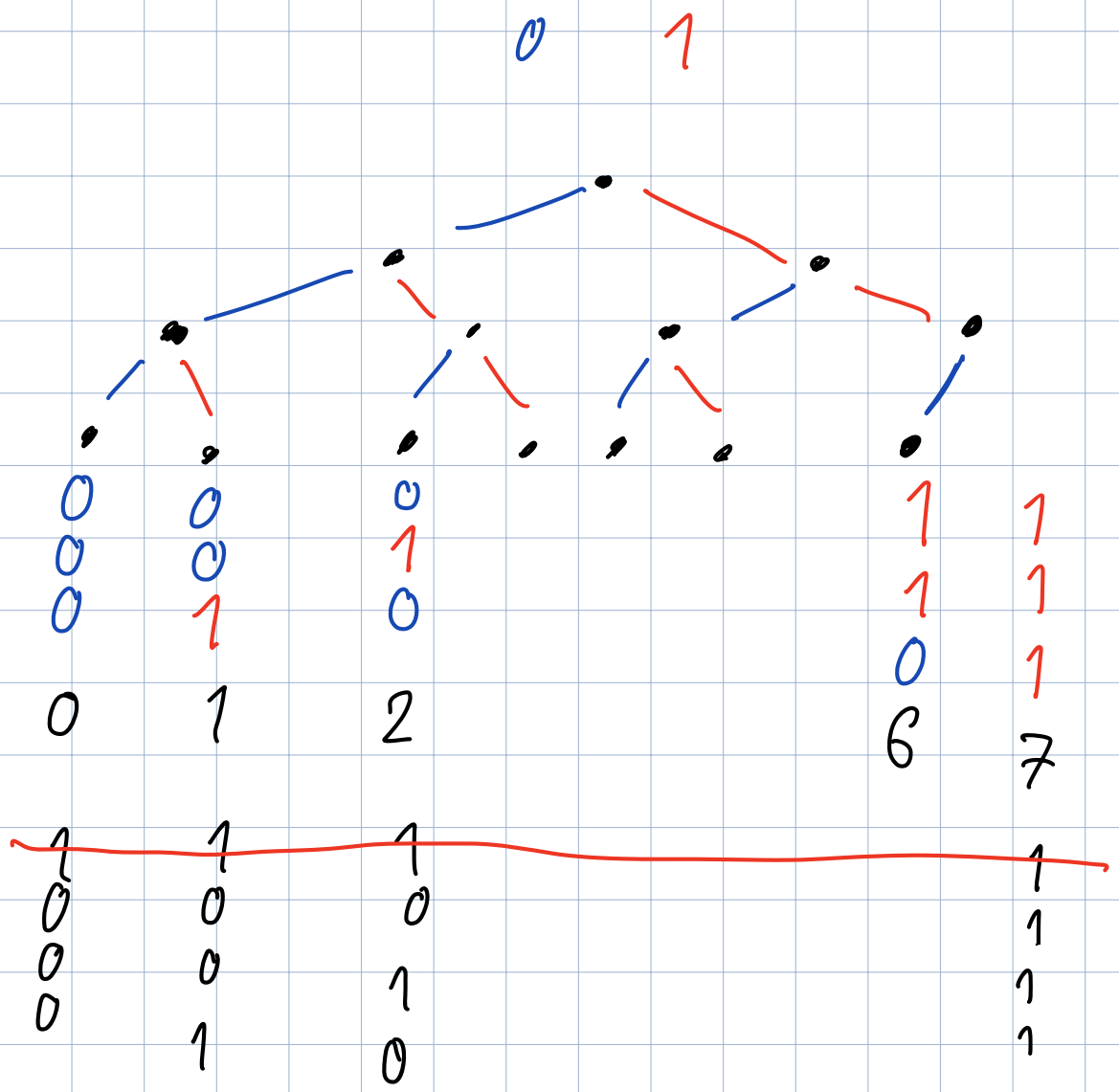
    method bubbleDown():
      if this.left = null: return
      minChild ← this.left
      if this.right ≠ null and this.right.elem < minChild.elem
        minChild ← this.right
      if this.elem > minChild.elem:
        this.elem, minChild.elem ← minChild.elem,
                                   this.elem
        minChild.bubbleDown()
```



```
method insert(x):
    node ← Node(x)
    this.addLast(node)
    node.bubbleUp()


method removeMin():
    min ← this.head.elem
    oldLast ← this.removeLast()
    this.head.elem ← oldLast.elem
    this.head.bubbleDown()

method addLast(newNode):
    this.size ← this.size + 1
    path ← this.size in binary      1110
    node ← this.head
    for digit in path[1..]:
      if digit = 0:
        if node.left ≠ null:
          node ← node.left
        else:
          node.left ← newNode
      else
        if node.right ≠ null:
          node ← node.left
        else:
          node.right ← newNode
```


0 1
0 0 0 0
0 0 0 1
0 1 0 0
1 1 0 6
1 1 1 7
0
1
2
6
7
1 0 0 0
1 0 0 1
1 0 1 0
1 1 1 1

```
direction = if digit = 0 then "left"
                         else "right"
if node[direction] ≠ null:
    node ← node[direction]
else
    node[direction] ← newNode
```