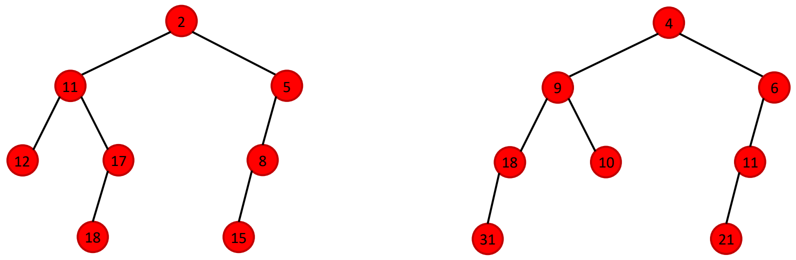
IN3130 Exercise set 11a

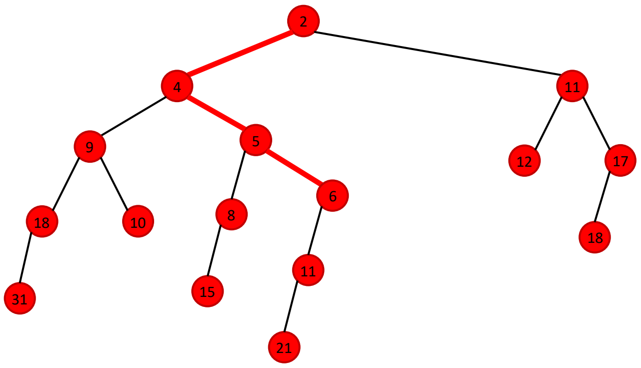
Exercise 1

Solve exercise 6.19 in Mark Allen Weiss *Algorithms and Datastructures in Java* (the book previously used in INF 2220 (now IN2010)).

The following trees are merged



The result is as follows, after merging and swapping, the original right path marked

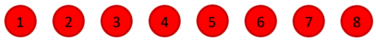


Exercise 2

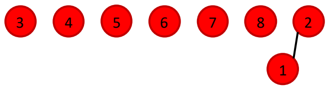
Solve exercise 6.25 in MAW.

We are technically allowed to construct a normal binary heap (using the normal `buildHeap()`-method that percolates down all subtree roots, starting at the bottom.) Convince yourself that this is the case. The following method, however, constructs a tree that is more leftist:

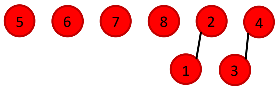
Insert the nodes into a queue. (Numbers indicate initial place in queue, not priority



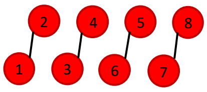
Merge 1 and 2 (leftist manner, maintain heap property!) and insert at end



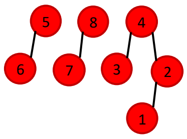
Merge 3 and 4 and insert.



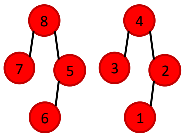
5, 6 and 7, 8 (5.key < 6.key)



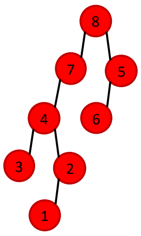
(1,2) and (3,4)



(5,6) and (7,8)



(1,2,3,4) and (5,6,7,8)



The time complexity is:

$$\frac{n}{2}\cdot O(1)+\frac{n}{4}\cdot O(2)+\frac{n}{8}\cdot O(3)+\cdots=O(n)\,.$$

We omit the O's and write

$$\frac{n}{2}\cdot 1+\frac{n}{4}\cdot 2+\frac{n}{8}\cdot 3+\cdots$$

$\Updownarrow$ Let $n=2^k$, this is worst case – full trees

$$\frac{2^k}{2}\cdot 1+\frac{2^k}{4}\cdot 2+\frac{2^k}{8}\cdot 3+\cdots$$

$$\Updownarrow$$

$$2^{k-1}\cdot 1+\ 2^{k-2}\cdot 2+2^{k-3}\cdot 3+\cdots$$

$\Updownarrow$ Writtten in summation form

$$\sum_{i=1}^{k-1}2^{k-i}\cdot i$$

$\Updownarrow$ Use the old $\Sigma=2\Sigma-\Sigma$ ploy…

$$\left(\sum_{i=2}^{k}2^{i}\right)-2^{k-(k-1)}(k-1)$$

$$\Updownarrow$$

$$\left(\sum_{i=2}^{k} 2^i\right) - 2(k-1)$$

$$\Updownarrow \qquad \text{The old } \Sigma = 2\Sigma - \Sigma \text{ ploy, again...}$$

$$(2^{k+1} - 4) - 2(k-1)$$

$$\Updownarrow$$

$$2(2^k - 2) - 2(k-1)$$

$$\Updownarrow \qquad n = 2^k, k = \log n$$

$$2(2^{\log n} - 2) - 2(\log n - 1)$$

$$\Updownarrow$$

$$2(n-2) - 2(\log n - 1)$$

$$\Updownarrow$$

$$2n - 4 - 2\log n + 2 = 2n - 2\log n - 2 = O(n)\,.$$

Exercise 3

Solve exercise 6.30 in MAW.

This should be obvious ("one can easily see..."), but we give a short induction proof. (The trees are constructed in an inductive manner that lends itself well to this proof technique.)

Basis: B_1 has B_0 as a child (subtree) from the root.

Step: Assume B_i has $B_0,\ldots,B_{(i\text{-}1)}$ subtrees of the root.
Must show that $B_{(i+1)}$ has $B_0,\ldots,B_i$ as subtrees of the root.

$B_{(i+1)}$ is constructed by connecting a B_i to the root of another B_i, therefore $B_{(i+1)}$ will consist of one B_i that we connected to the root of the other B_i, plus the subtrees that already are connected to the root of the other B_i (the root one), these are (by the assumption): $B_0,\ldots,B_{(i\text{-}1)}$. Therefore $B_{(i+1)}$ must have the subtrees $B_0,\ldots,B_i$.

Exercise 4

Write a non-recursive implementation of `merge()` for leftist heaps.

We do this kind of merge with a two pass method.

1) The nodes in the right paths of the heaps can be viewed as lists. the root is the head, the .right pointers in the nodes is next.

   The lists are merged (elements in lexicographic order). Always choose the smallest and copy into a new tree (a new list).

2) Traverse the new path (list), from the end towards the root (we need a pointer this way – doubly linked lists). Check that the leftist-property holds (null path lengths of children), swap left and right children if property is violated.

Rough pseudo code can be something like this:

```
function merge(h1,h2)
   var list result
   while h1 <> nil and h2 <> nil
      if h1.key <= h2.key
         append h1.first to result     // assuming .first works
         h1 = h1.right
      else
         append h2.first to result
         h2 = h1.right
      if h1 <> nil
         append h1 to result
      if h2 <> nil
         append h2 to result

   var elem node
   elem = result.last
   while elem <> result.first
      if elem.left.npl < elem.right.npl
         swapChildren(elem);
      elem = elem.parent               // assuming a parent pointer

   return result
end
```

Exercise 5

Professor Pinocchio claims that the height of an *N*-node Fibonacci heap is *O*(log *N*). Prove the professor wrong by showing that for every positive integer *N*, there is a sequence of Fibonacci heap operations constructing a heap that is one long chain of *N* nodes.

(Some applets exists on the internet that visualize Fibonacci heaps, most require javascript.)

A kind of induction is also at the basis of this construction. We build our chain by using the structure of binomial trees as model.

Our basis is a tree consisting of two nodes. We can construct this tree by inserting three nodes in an empty heap, and then run `deleteMin()`.

The step in our construction (induction) consists on inserting three nodes with a lower key than the nodes already in the heap, name them *a, b, c* (sorted by key, increasing order), and run `deleteMin()`, this results in a tree with two branches, the root is *b*, one branch is the tree we started with, the other branch is *c*. Now erase *c*. Repeat as many times as necessary.

Exercise 6

Discuss the notions of average and amortized time briefly.

Left for the group to discuss. Look for instance at series of operations on an imaginary data structure with the following running times:

Series 1: 1, 1, 1, 3, 2, 1

Series 2: 1, 2, 3, 1, 1, 1

Series 3: 100, 100, 100, 1, 1, 1

Assume the operations are three inserts and three deletes, and look at possible subsets of the series, for instance the first four operations.

[end]