INF 4130 Exercise set 8, 2017
w/solutions

Exercise 1

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

6.19

Merge the two leftist heaps in Figure 6.58

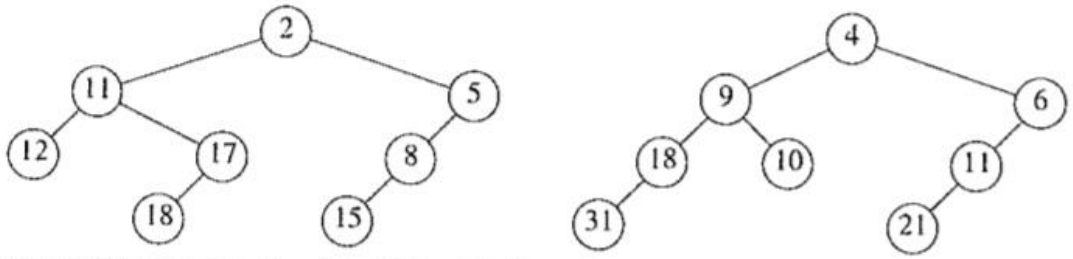
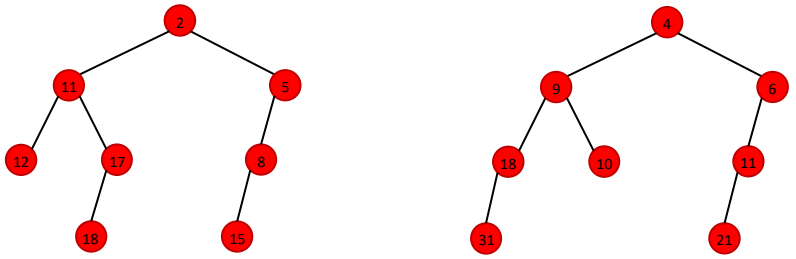
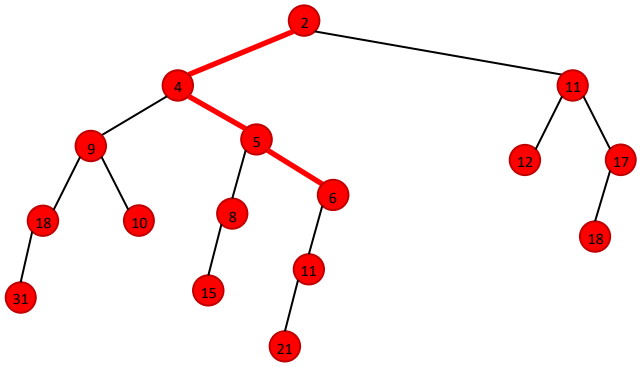


Figure 6.58 Input for Exercises 6.19 and 6.26

The following trees are merged



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



Exercise 2

Solve exercise 6.25 in MAW.

6.25:

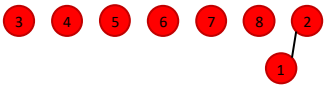
We can perform buildHeap in linear time for leftist heaps by considering each element as a one-node leftist heap, placing all these heaps on a queue. and performing the following step: Until only one heap is on the queue, dequeue two heaps, merge them and enqueue the result.

1. *Prove that this algorithm is O(N) in the worst case.*
2. *Why might this algorithm be preferable to the algorithm described in the text?*

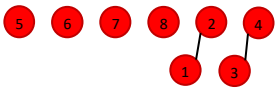
We are technically allowed to construct a normal binary heap (using the normal `buildHeap()`-method that percolatesDown all subtree roots, starting at the bottom.) Convince yourself that this is the case. The following method, however, constructs a tree that is more leftist:

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

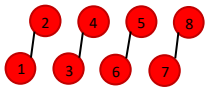
Merge 1 and 2 (leftist manner, maintain heap property!) and insert at end of queue.



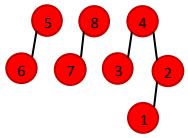
Merge 3 and 4 and insert.



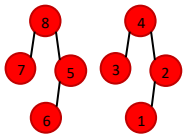
5, 6 and 7, 8 (5.key < 6.key)



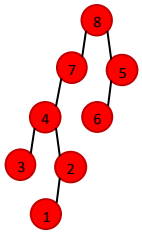
(1,2) and (3,4)



(5,6) and (7,8)



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



The running time is given by the following expression. It describes the time it takes to merge two and two single nodes into 2-node trees, plus the time it takes to merge two and two 2-node trees into 4-node trees, and so on. The number of trees will halve in each step, and the running time for each merge follows from the height of the trees, which increases by one in each step. (See the figure above)

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

We omit the *O*'s and write

$\frac{n}{2}\cdot 1+\frac{n}{4}\cdot 2+\frac{n}{8}\cdot 3+\cdots$	
⇳	The worst case is when the number of elements, *n*, is a power of 2. If this is the case, all trees will be as large as possible in each step, so let Let $n=2^k$, for some *k*.
$\frac{2^k}{2}\cdot 1+\frac{2^k}{4}\cdot 2+\frac{2^k}{8}\cdot 3+\cdots$	
⇳	Since $2^k/2 = 2^{k-1}$ and $2^k/4 = 2^k/2^2 = 2^{k-2}$, and so on, we rewrite
$2^{k-1}\cdot 1+\ 2^{k-2}\cdot 2+2^{k-3}\cdot 3+\cdots$	
⇳	We rewrite it using summation form
$\sum_{i=1}^{k-1}2^{k-i}\cdot i$	
⇳	A trick we can use when we work with sums where we feel that the terms will cancel each other out in a telescoping manner, is to write the sum as $\Sigma=2\Sigma-\Sigma$. (2*x* minus *x* is *x*, no matter what x is.) [This is a trick one just needs know about.] It is probably easier to see which terms cancel each other out if we set it up like this on two lines: $\Sigma = 2\Sigma$ $\quad\ -\Sigma$ and write out the terms, but first we just multiply our original sum by 2 (we multiply each term by 2) $2\cdot 2^{k-1}\cdot 1+\ 2\cdot 2^{k-2}\cdot 2+\ 2\cdot 2^{k-3}\cdot 3+\cdots+2\cdot 2^{k-(k-1)}$ ·(k-1) Which is the same as $2^k\cdot 1\ +\ 2^{k-1}\cdot 2\ +\ 2^{k-2}\cdot 3\ +\cdots+2^2$ ·(k-1)

We then subtract, like this

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

$- 2^{k-1} \cdot 1 - 2^{k-2} \cdot 2 - \cdots - 2^{k-(k-1)} \cdot (k-1)$

We get (the last term we subtract does not cancel with anything)

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

$- 2^{k-(k-1)} \cdot (k-1)$

Writing it out in the opposite direction

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

$- 2^{k-(k-1)} \cdot (k-1)$

we see that this is the same as

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

$\Updownarrow$ Since $2^{k-(k-1)} = 2$, we get

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

We use the $\Sigma = 2\Sigma - \Sigma$ trick, again... (on the sum)

$\Updownarrow$ This time we are left only with the last term multiplied by 2 and minus the first term: $2 \cdot 2^k = 2^{k+1}$ and $= -2^2 = -4$

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

$\Updownarrow$ Factoring out 2

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

$\Updownarrow$ Recall that we chose the number of elements as a power of 2. Since $n = 2^k$ we have $k = \log n$

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

$\Updownarrow$

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

$\Updownarrow$ We multply out, and since $n > \log n$, the expression for the running time ($O(n)$) follows

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

Exercise 3

Solve exercise 6.30 in MAW.

> **6.30**
>
> *Prove that a binomial tree B_k has binomial trees $B_1, B_2, \ldots, B_{k-1}$ as children of the root.*

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.)

In induction proofs we show that something holds for a fixed basis, for instance that something (a formula) is true for $x = 0$. We then assume that the same something (the formula) holds for $x = i$ and show that this implies that it also holds for $x = i + 1$, this is called the induction step. Together the basis and the step show that what holds for the basis ($x = 0$) also holds for the next value ($x = 1$), and so on, and so on.

Basis: B_1 has B_0 as a child (subtree) from the root. (B_1 is simply constructed by connecting a B_0 to another B_0, so this is obviously true.)

Step: Assume B_i has $B_0,\ldots,B_{(i-1)}$ as separate subtrees of the root.
We must show that this implies that $B_{(i+1)}$ has $B_0,\ldots,B_i$ as subtrees of the root.

Our $B_{(i+1)}$ 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 just connected to the root of the other B_i, plus the subtrees that already were connected to the root of the other B_i (the root one), these are (by the assumption): $B_0,\ldots,B_{(i-1)}$. Therefore $B_{(i+1)}$ must have the subtrees $B_0,\ldots,B_i$. We already have $B_0,\ldots,B_{(i-1)}$ (making up the first B_i) and just connected a whole new 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.

Try using the applet on

http://www.cs.yorku.ca/~aaw/Jason/FibonacciHeapAnimation.html

to construct this chain, and to get a feel for Fibonacci heaps.

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 the 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.