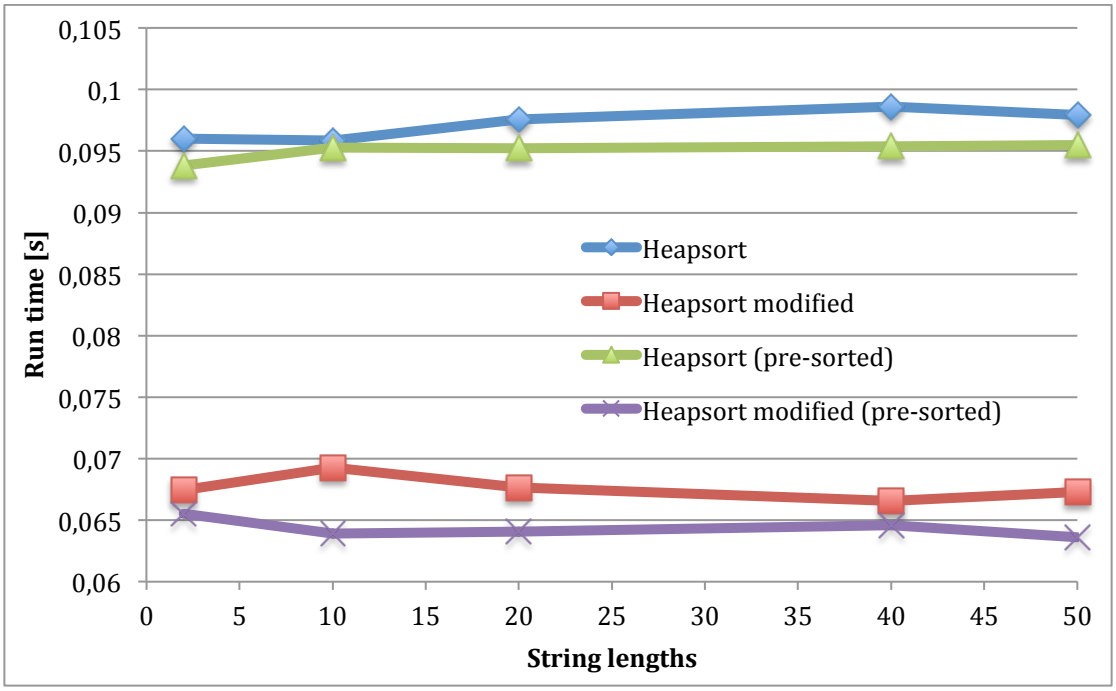
1. Sink-and-swim heapsort in maximum priority queues

I modified the sorting part of heapsort in *Heap.java* by adding a method *sinkToBottomAndSwimUp.* This method does not check whether the sinking element reaches it correct position along the sinking operation. Instead in sinks the element to the bottom, and then it swims the element up. I added the standard *swim* method (p. 316) for this.

Next I generated arrays of size 10 000 consisting of strings concatenated by characters from the lower case alphabet (abcd...). I then ran the program 600 times per string length, disregarding the 100 first runs, and took the average runtime. The reason I disregard runs is to get a good average. The first runs may take longer time due to processing power and memory allocation.

I did all runs for different string lengths, and modified/normal heapsort in one run. I found this to give me most stable results. I also turned of compiler optimization. I also did the same runs where I did not shuffle between runs (i.e. pre-sorted arrays)



The results show that the modified Heapsort algorithm runs significantly faster. Normal Heapsort takes about 50% more time to complete. Secondly (but not surprisingly,) we see that the pre-sorted arrays are always sorted faster. Lastly we notice that the length of the strings seems to be insignificant to runtime. This is because to compare strings we only need to compare the first character(s).

2. Blooming E-coli

To compress the *n* strings I used *FNV* and *Murmur* hash functions found on Github. I had a total of four different available hash functions (k) and varied the length of the Hash Symbol table and *k*, according to a pre-determined tolerance of percentage of false positive *p*. Using the following formula I determined the optimal number of bits *m,* in the Hash Symbol Table:

$$m = -\frac{n \cdot ln(p)}{ln(2)^2}$$

where *n* is the number of strings to be compressed (10^6). Then we find the optimal number of hash functions *k,* using:

$$k = \frac{m}{n} \cdot n(2)$$

Based on these formulas I constructed the following table for given tolerances *p*:

p	m	k
0,01	9 585 059	6,6
0,02	8 142 364	5,6
0,03	7 298 441	5,1
0,04	6 699 668	4,6
0,05	6 235 224	4,3
0,06	5 855 746	4,1
0,07	5 534 902	3,8
0,08	5 256 973	3,6
0,09	5 011 823	3,5
0,1	**4 792 529**	**3,3**
0,11	4 594 154	3,2
0,12	4 413 051	3,1
0,13	4 246 452	2,9

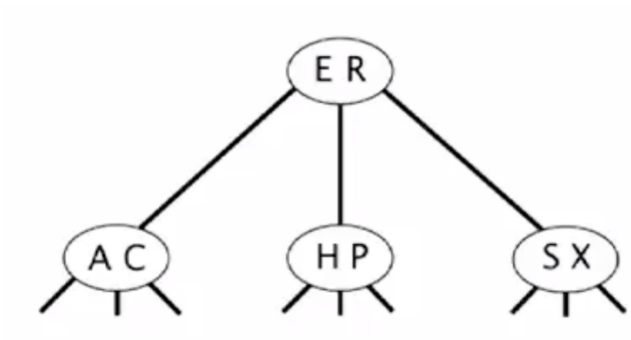
I found m=4 792 528 and k=3 to give the best score according to the score criterion:

$\frac{S_i}{S_c}(1 - (S_{ratio} \cdot FPR))$, with FPR < 10%

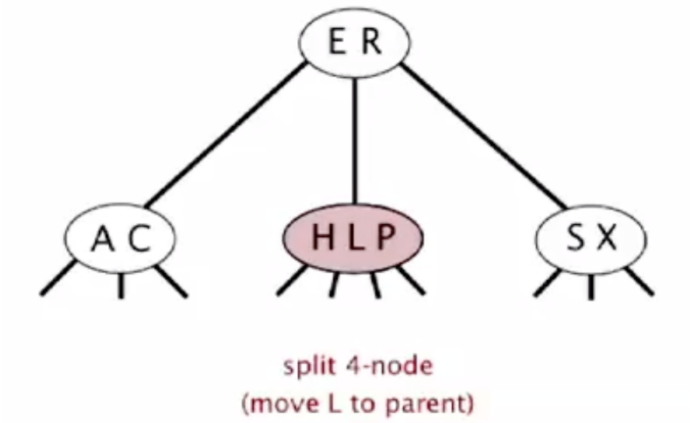
I got a score of 6.42 and FPR of 9.85% which was just within the requirement.

3. A)

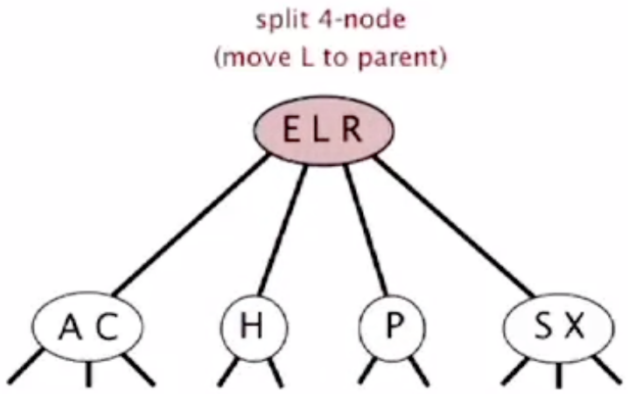
Show stepwise what happens when you perform *insert(L)* to the following 2-3 tree:



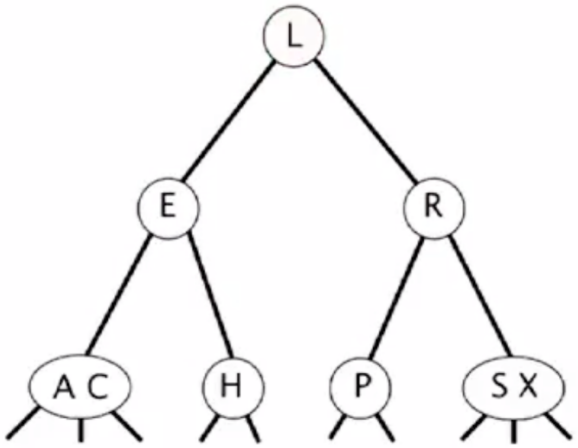
Solution:
1. First we search for L's correct position. It's between E and R so it goes down the middle. It is between H and P, so it goes in the middle, creating a temporary 4 node.



2. Then the 4 node (HLP) is split and L travels up to the parent (it's the middle element in the 4 node that travels up).



3. This creates another temporary four node, which is split, causing L to travel up in the tree creating a new root.



3. B)

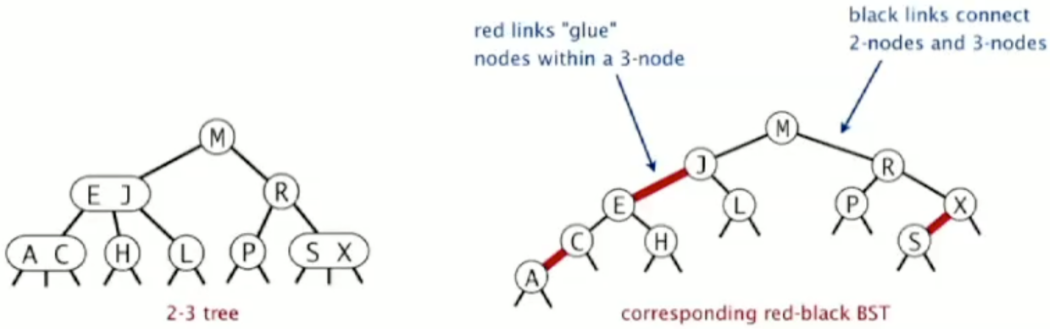
What is the worst and best case scenarios regarding tree height for a 2-3 tree?

Solution:
Worst case would be to have only 2-nodes. With N elements this would results in a height $lg_2(N)$.

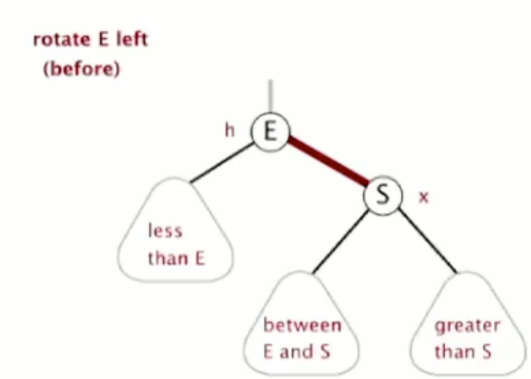
Best case would be if all nodes were 3-nodes and would result in $lg_3(N)$ height.

3. C)

One way to represent a 2-3 binary search tree is with left-leaning red-black (LLRB) search tree. In such representation the red links indicate that the node below is a 2-3 node. The red links always lean the the left.



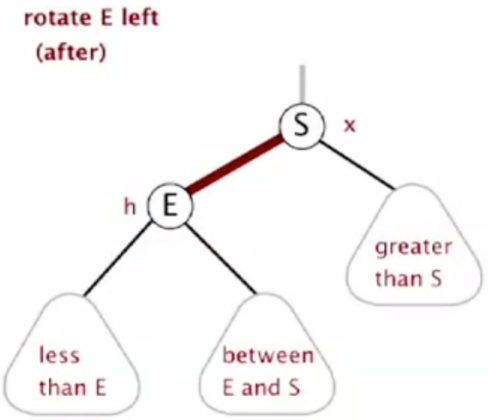
One of the operations in such a tree is rotateLeft, which is used if a red link leans to the right:



1. Show how the result after performing rotateLeft on the above LLRB search tree.

2. Another propery of the LLRB is that is can never have two sonsecutive red links. Explain why.

Solution:
1. One property of the left-learning Red-Black search tree is that the red links always lean to the left. For the above example, rotateLeft would maintain this property by reorganizing the three like this:



2. Two consecutive red links would represent a 4 node which is not possible in a 2-3 tree. The LLRB search tree represents a 2-3 tree where each red link represents a 3 node.