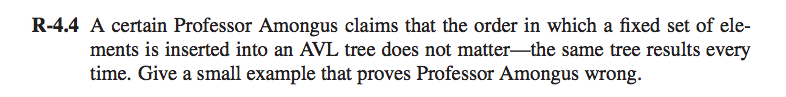
CS 600 Homework 3 | CWID 10430147 | Divyendra Patil | Username: dpatil3  
Date: 20/11/2017

Chapter 4:



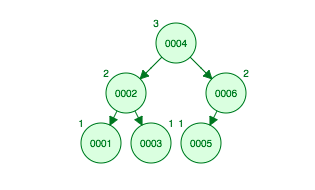
Professor Amongus claims can be proved wrong by assuming the following example:

The Insertion of {1,2,3,4,5,6} will not result into the same binary tree as that of {2,1,3,4,6,5}.

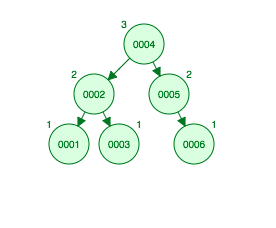
Basically the result will keep changing as per the sequence of inputs.

Thus, the assumption that the same tree results regardless the order of insertion of elements is wrong.

Result of Insertion: 1,2,3,4,5,6



Result of Insertion: 2,1,3,4,6,5



Screen%20Shot%202017-09-19%20at%201.22.17%20AM.png

If we consider a heap implementation, which is basically a binary tree sorted in an array.

It contains a red black tree which has comparatively few number of rotations compared to an AVL tree. Hence there is a possibility that the balance of a red black tree is not proper.

Now we know that the height is eight, each red node will not have red children and thus there can be a maximum of four red nodes in a single path of height eight (alternating red and black).

In this case, the last node of the path will be a red node with two null leaves.

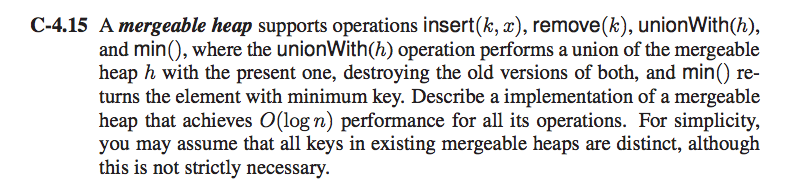
After we know that we have a maximum of four red nodes, we have at least four black nodes and the fifth black node will be beyond the node at height eight.

Since a black path is valid, **the minimum height of any branch in the RB tree is five, where the fifth node of each path is a null leaf.**

**We must note that four is the minimum height of any given branch. This can be deduced by keeping the depth of black (8/2)=4 for leaves and alternating black and red along the longest subtree.**

Children of branches having red nodes in each sub-tree will have a black child for each red child.

Keep in mind, the question is asked about the “**Minimum number of nodes**” and not just nodes since there can be many answers. If we draw a Red Black Tree with height 8 and calculate the nodes, the minimum will be **61** (If we take in consideration the leaf nodes).



Examples of mergeable heap are Binomial Heap, Leftist Trees & Fibonacci Heap. We will implement Binomial Heap.

A binomial heap H is a set of binomial trees that satisfies the following binomial heap properties:

• Each binomial tree in H is heap-ordered: the key of a node is greater than or equal to the key of its parent.

• There is at most one binomial tree in H whose root has a given degree.

**Inserting a node**:

Assumptions: Node x has already been allocated, and key

x->key has already been filled in.

BINOMIALHEAPINSERT(H, x)

1 H0 := MAKEBINOMIALHEAP()

2 x->p := NIL

3 x->child := NIL

4 x->sibling := NIL

5 x->degee := 0

6 H0:head = x

7 H := BINOMIALHEAPUNION(H,H0)

Inserting a new element to a heap can be done by simply creating a new heap containing only this element and then merging it with the original heap. Due to the merge, insert takes O(log n) time.

**Finding min**:

BINOMIALHEAPMIN(H)

1 y := NIL

2 x := H:head

3 min := 1

4 while x 6= NIL

5 if x->key < min

6 min = x->key

7 y := x

8 x := x->sibling

9 return y

BINOMIALHEAPMIN(H) returns a pointer to the node with the minimum key in an n-node heap. It is assumed that there are no keys with value 1 in H.

• Binomial heaps are heap-ordered : the minimum key is in a node in the root list of the heap.

• The root list had length <= log n + 1 : the running time of this operation is O(log n).

**Removing a node**:

Deleting a node x from a binomial heap H is trivial:

• First, decrease the key of x to a value smaller than any key in H, e.g., -∞.

• Next, extract from H the node with minimal key, which is x with key -∞.

If heap has n nodes, then REMOVE(k) takes O(log n) time.

**Union**:

It works in 2 phases:

• 1 First, BINOMIALHEAPMERGE(H1, H2) merges the root lists of H1 and H2 into a single list H that is sorted by degree into monotonically increasing order.

• 2 There might be at most 2 roots of each degree, the second phase links roots of equal degree until at most one root remains of each degree.

The meaning of the local pointers used in the algorithm is:

• x points to the root currently being examined.

• prev\_x points to the root preceding x on the root list, thus prev\_x->sibling = x.

• next\_x points to the root following x on the root list, thus x->sibling = next\_x.

There can be 4 cases when we traverse the list of roots:

• Case 1: Orders of x and next-x are not same, we simply move ahead.

• Case 2 : If order of next-next-x is also same, move ahead. x = next\_x = next\_x->sibling

• Case 3 : If key of x is smaller than or equal to key of next-x, then make next-x a child of x by linking it with x. x = next\_x != next\_x->sibling and x->key <= next\_x->key.

• Case 4 : If key of x is greater, then make x as child of next

Unionwith(H)

1 H := MAKEBINOMIALHEAP()

2 H. head = BinomialHeapMerge(H1, H2)

3 free the objects H1 and H2, but not the lists they point to

4 if H. head = NIL

5 return H

6 prev\_x := NIL

7 x := H:head

8 next\_x := x->sibling

9 while next\_x != NIL

10 if x != next\_x or

(next\_x->sibling != NIL and next\_x->sibling =x)

11 prev\_x := x // Cases 1 and 2

12 x := next\_x // Cases 1 and 2

13 else if x->key \_ next\_x->key

14 x->sibling := next\_x->sibling // Case 3

15 BINOMIALLINK(next\_x; x) // Case 3

16 else if prev\_x = NIL // Case 4

17 H:head := next\_x // Case 4

18 else prev\_x->sibling := next\_x // Case 4

19 BINOMIALLINK(x; next\_x) // Case 4

20 x := next\_x // Case 4

21 next\_x := x->sibling

22 return H

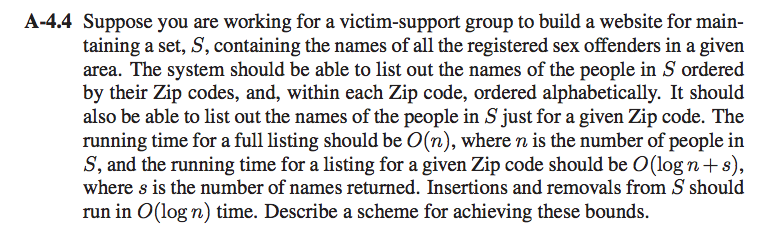
Let H1 contains n1 nodes, H2 contains n2 nodes and let n = n1 + n2.

Then, H1 contains <= log n1 + 1 roots, and H2 contains <= log n2 + 1 roots

H contains at most log n1 + log n2 + 2 <= 2 log n + 2 = O(log n) roots immediately after the call of BinomialHeapMerge.

**BinomialHeapMerge** takes O(log n ) time. Each iteration of the while loop takes O(1) time and there are at most log n1 + log n2 + 2 iterations because each iteration either advances the pointers one position down the root list or removes a root from the list.

Total run time is O(log n)



We consider a binary search tree where each node points to a child binary search tree.

The parent binary search tree will have keys containing zip code and all the child binary search trees will have keys containing names.

**Full list retrieval**:

If we have to retrieve a full list of people in S, we need to traverse the binary search tree, and report all the values stored at each node by fully traversing the values stored in the node child’s binary search tree. This will execute once for every stored node and will have a runtime of O(n).

**List of names within a zip code retrieval**:

To retrieve a list of all names within a zip code, we need to first locate the zip which we are intending to target. Searching in a binary search tree is an O(log n) operation.

Once found, we traverse the node’s child binary search tree and report every name under it in alphabetical order. The second traversal will be O(s) where s is the number of nodes containing respective returned. Thus the time complexity for this operation will be O(log n + s).

**Insertion and Removal**:

**INSERTION**

To insert node N

1) Perform standard BST insert for N.

2) Starting from n, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from N to z and x be the grandchild of z that comes on the path from N to z.

3) Re-balance the tree by performing appropriate rotations on the subtree rooted with N. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:

=> y is left child of z and x is left child of y (Left-Left Case)

=> y is left child of z and x is right child of y (Left-Right Case)

=> y is right child of z and x is right child of y (Right-Right Case)

=> y is right child of z and x is left child of y (Right-Left Case)

Time Complexity: O(log n)

**REMOVAL**

Let N be the node to be deleted

1) Perform standard BST delete for N.

2) Starting from N, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from insertion here.

3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:

=> y is left child of z and x is left child of y (Left-Left Case)

=> y is left child of z and x is right child of y (Left-Right Case)

=> y is right child of z and x is right child of y (Right-Right Case)

=> y is right child of z and x is left child of y (Right Left Case)

Time Complexity: O(log n)

Insertion and removal operations in the binary search tree will be O(log n) since we take into account the height of the parent and child trees.

The sum of them both is bound by the log of the number of elements in the tree, assuming we have a balance tree.

Thus, the time complexity of insertion and removal **are O(log n).**

Chapter 5:

Screen%20Shot%202017-09-19%20at%201.24.33%20AM.png

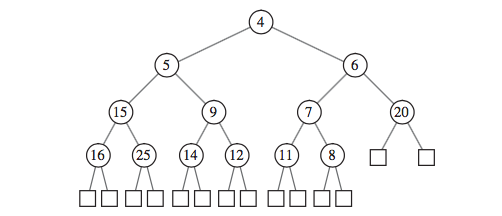
The working of insertion sort is that it compares the current element to the largest element in the sorted part of the list. If largest element in the sorted list is larger than the current element nothing is done. Else the element e is inserted in place of it. And the array/list is reshuffled accordingly.

The worst case for this algorithm would be when the given list is in the reverse order of it to be sorted.

For example: 54, 43, 38, 29, 20, 19, 15, 9, 8, 7.

In this list, every element will be moved ﬁrst to the front and then then back in the list, as every remaining is processed. Thus, each element will be moved n times. For n elements, this means at a total of n2 times. Therefore, we can conclude that worst case time complexity of insertion sort is Ω(n2).

Screen%20Shot%202017-09-19%20at%201.24.42%20AM.png



As a first step, we need to replace 5 with 18.

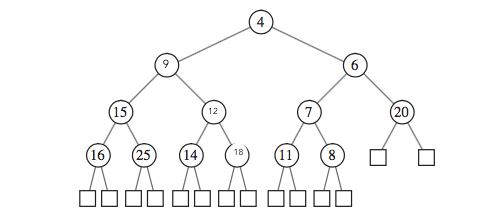
Step 1: Compare the nodes with its children.

Step 2: Check whether the parent is the smallest of them.

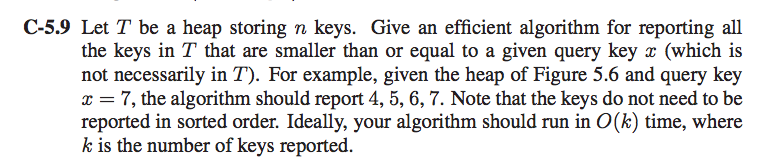
Step 3: If not, replace the smallest child with the parent. In this case, we compare 18 with it’s children. Since 9 is the smallest child, we swap 9 with 18 to make it the new parent.

Step 4: Perform step 2 and step 3 to find the smallest node as the parent. We compare 18 to its child nodes again. We find that 12 is smaller than 18 so we swap 18 and 12 node.

Thus, now the tree is a heap tree.



**Final Answer**



We know that a node with key k will have subtrees which contains key values larger than k.

We only consider values which are smaller than x.

We ignore all the nodes in such a subtree whose root does not meet our criteria.

This algorithm will run recursively:

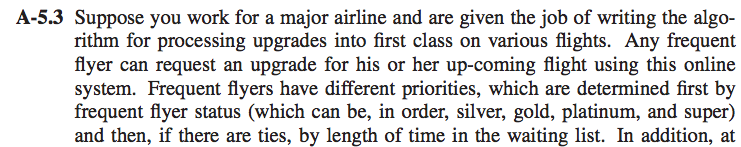
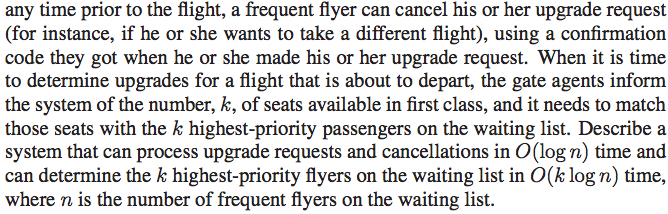
• Check whether the key is greater than equal to x; return

• Output the value of this node

• Search the left leaf of the tree

• Search the right leaf of the tree

The algorithm justifies only relevant nodes. It results into k keys number of searches and therefore an O(k). The algorithm takes time O(k), because none of node which is in T has a key bigger than x (which has a descendant have any key less than x)

We should use Priority Queue is the data structure which can upgrade requests and cancellations in O(log n) time and can determine the k highest-priority flyers on the waiting list in O(k log n) time, where n is the number of frequent flyers on the waiting list.

Priority queues can be implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time.

**UPGRADE\_REQUESTS**() : this can be implemented using **Bottom-up** **reheapify**

upgrade(k) {

while (k > 1 && less(k/2, k)) {

exch(k, k/2);

k = k/2;

}

}

Worst-case runtime of the algorithm is O(log n).

**CANCELLATIONS**: this can be implemented using **Top-down heapify**

cancel(k) {

while (2\*k <= N) {

int j = 2\*k;

if (j < N && less(j, j+1)) j++;

if (!less(k, j)) break;

exch(k, j);

k = j;

}

}

Worst-case runtime of the algorithm is O(log n).

**SEARCH**: Since binomial heap takes O(log n) time where n is the number of frequent flyers on the waiting list.. K passengers with high priority can be located using reheapify in O(K log n )

worst-case runtime of the algorithm is O(k log n)

**Note**: We don’t need to write a separate remove method since we wont be removing a passenger from economy class and putting him in the Business Class, rather we would just be moving him up the priority queue.