**CS311 Yoshii- Week 9 B (Notes-9B) Heap Trees and Priority Queues**

**🡺 HW5 balanced tree and heap**

**Data Structure Agenda:**

* **Heap (a tree for sorting and priority queues)**

**Binary Trees and Sorting**

Last week, we looked at binary trees and their properties.

We also looked at a binary search tree and its implementation.

**Recall that once you built a binary search tree, if you traverse it in IN-order, you will see a sorted list.**

**Today, we will look at another way to sort elements using a binary tree.**

**Binary Heaps (Chapter 6.9 page 269)**

**A heap tree**

--> a nearly complete binary tree

--> no parent has a value bigger than the value of its children

**(i.e. the values get larger as you go down the tree)**

--> right most leaves at level B (the bottom level) may be missing.

/ \

/ --- Some right side leaves are at an earlier level

---- This is level B

**NOTE: the book has the values increase as you go UP the tree.**

**However, the sorting algorithm works better if we use**

**the version presented here.**

The following are all heaps.

**e.g 1**

**2 3**

**4 5 6 7**

e.g.

1

3 2

6 7

e.g.

1

3 6

4 5 9

**What Is Heap Sort?**

**If we use a heap tree to sort elements, it is called the Heap Sort.**

If **we remove the root**, we know that it is the minimum element.

**\*Inter1\* Finding the minimum is thus O(??)**

**Re-Heapify** --> make the tree have the heap property again after removing the root.

If we re-heapify the tree, then the root will be the minimum of the

remaining elements.

**\*Inter2\* So, if we do**

**- remove root**

**- re-heapify**

**repeatedly, what we will accomplish????**

**Heap Sort ==**

**Building the initial heap tree +**

**Re-heapifying repeatedly**

***How do we re-heapify a tree?***

e.g 1 remove 1

2 3

4 5 6 7

**1) Move the very bottom level right most leaf to the root**

**(this is one the of the largest values found in the tree)**

7 <--X move 7 to the root

2 3

4 5 6

**X points to the temporary root; (7 in the above example)**

**2) We will make X trickle down to the right place by comparing**

**with the smaller of its children. Stop if children are not smaller.**

i.e.

* **Compare X's value with the smaller of Left-Child and Right-Child.**
* **If X's value is smaller than that,**
  + - **X can stay where it is, so stop. You are done.**
* **If X's value is larger than that,**
  + - **X cannot stay where it is.**
    - **Exchange values with that smaller child.**
* **X should now point to the smaller child node and repeat this process if X is not a leaf yet.**

e.g.

2 2 has been exchanged with 7



X--> **7**  3



4 5 6



e.g.

2

4 3 4 has been exchanged with 7

**7** 5 6

X has reached the bottom.

The tree has been re-heapified.

**This tree had B = 2.**

**Thus, any X has the chance of trickling down to level 2.**

**At each level, we need to do two comparisons.**

**In the worst case, it requires 4 comparisons to re-heapify the tree.**

***Analyzing Re-heapify***

**\*Inter3\* For each re-heapify,**

**how many comparisons are possible in the worst case??**

**Hint:**

**we start at the root and compare with X's children (2 comparisons)**

**at level 1, at level 2, at level 3...... at level B.**

**Express the answer in terms of B. W(N) =**

**Note that B is Theta(logN) where N is the number of nodes in the tree.**

**Therefore, express the answer in terms of N. W(N) = Theta(**

***Analyzing Heap Sort***

If you had to sort N elements, then the Heap tree starts



out with N nodes.



You remove a node and re-healify.

You remove a node and re-heapify.

Again and Again.

Intuitively, Theta(log N) is done N times to remove N elements => **W(N) = Theta(N log N).**

**But you need to realize that the tree is**

**getting smaller with each removal.**



Let K be the number of elements left in the tree.



1st time, the tree has N-1 elements to re-heapify K=N-1



2nd time, the tree has N-2 elements to re-heapify K=N-2

..

last time, the tree has 1 element left K=1



For each re-heapify, we do Theta(log K) comparisons

**#TotalComparisons = Sum of Theta(log K) as K varies from N-1 down to 1**

**This turns out to be still Theta(NlogN).**

NOTE: This is the same as **the lower bound** we had established for comparison based sorting.

**But we did not take care of the initial construction of the Heap.**

How long will that take?? If it takes more than Theta(N log N)

then we cannot not achieve the lower bound anyway.

**Two alternatives follow for creating the initial heap tree.**

***Method 1: Construct the initial tree via Inserts:***

**This method starts out with no tree and inserts each value by making it trickle up to the right place.**

**There is no heap tree to begin with.**

**X = new node with new data;**

**Make X the last leaf of the very bottom level.**

**While X has parent with a larger key do**

**{ // trickle X up the tree as follows**

**P is the parent of X;**

**Exchange the values of X and P;**

**X now points to the parent;**

}

e.g.

8 added 8

e.g.

8 added 7

7

--> 7 7 trickles up

8

e.g.

7 added 6

8 6

--> 6 6 trickles up

8 7

e.g.

6 added 5

8 7

5

--> 6 5 trickles up

5 7

8

--> 5 5 trickles up again

6 7

8

***Analysis of Each Insert:***

Each Insertion can end up exchanging with every ancestor of the new node. This is the worst case.

How many ancestors?

Same as the bottom level number of the current tree.

**Worst case # of comparisons is B for each insertion. W(n) = B = Theta(logN)**

***Analysis of Construction via Inserts:***

But we do N insertions and each time the tree gets bigger.

Do bottom up exchanges for each node inserted at the current bottom level CB, where CB = 1 to B.

The number of nodes inserted at level CB is at most 2^CB

**Thus, The total number of comparisons in the worst case is:**

Sum of #nodes-added-at-CB \* CB for CB = 1 to B

**Sum of 2^CB \* CB for CB = 1 to B**

**<= Theta(NlogN)**

**Thus, the initial tree can be built within Theta(NlogN)**

***Method 2: Construct the initial tree via Heapify:***

**Start with a non-Heap FULL binary tree and apply heapify starting**

**at node X where X is found in the reverse level order.**

Recall that heapify == trickle down the node to the right place.

Thus, every sub-tree from the bottom is heapified before

their roots are.

e.g.

**Depth = 1 subtrees**

8

[7] [6]

4 5 3 2

Heapify subtree at 6 and get..

8

[7] 2

4 5 3 [6]

Heapify subtree at 7 and get ..

8

4 2

[7] 5 3 6

**Now Depth = 0 subtree**

[8] heapify the tree at [8]

4 2

7 5 3 6

This results in

2

4 3

7 5 [8] 6

***Analysis of Each Heapify:***

Each heapify on a sub-tree requires 2\*(B-D) comparisons in the worst case where D is the depth level of the sub-tree’s root.

B is the bottom level of the tree.

***Analysis of Construction via Heapifying:***

The number of sub-trees at depth D is 2^D where

D varies from B-1 to 0

(2^(B-1), 2^(B-2), 2^(B-3) etc.)

**The total number of comparisons in the worst case is:**

**Sum of (#subtrees) \* (#comps) for D = B-1 to 0**

**Sum of (2^D) \* 2(B-D) for D = B-1 to 0**

**<= Theta(NlogN)**

**Again, the initial tree can be built within Theta(NlogN)**

**Summary of Algorithms:҉҉**

* **Re-heapify as the root is removed: trickle down the parent to the smaller child**
* **Heapify an existing tree: for each sub-stree, trickle down the parent to the smaller child**
* **Insertion to create a tree: trickle up the new element up if the parent is larger.**

***Conclusion on Heap Sort:***

**W(n) = time to construct + time to remove+reheapify**

**<= Theta(nlogn) + Theta(nlogn) -- method 1**

**<= Theta(nlogn) + Theta(nlogn) -- method 2**

**҉҉**

**And Heap Sort sorts in place (within the tree)!!!!!!**

**Therefore, it is a very fast algorithm. OPTIMAL.**

**\*Inter5\* But is it as good as Merge Sort?**

**Time =**

**Space =**

***Application of Heap Sort – Priority Queue***

Recall an earlier exercise for a printer queue.

**Assume there is a priority printer "queue"**

such that you **maintain a sorted list**

**(i.e. always search for the right place to insert a new job).**

The best job is always at the front of the list.

**Let's say N jobs are in the list right now.**

**\*Inter1\* To find the best job, the time complexity is O(?)**



**\*Inter2\* In what format should we store the data to make insertion fast?**



**\*Inter3\* If we stored it this way, then how many comparisons**



**does it take to insert a new job? (worst case)**

**\*Inter4\* To which phase and operation of Heap Sort does this insertion**

**of a job correspond?**

**\*Inter5\* After the best job is printed, what needs to be done???**



**How long will this take????**

***Array Implementation of Heap (see page 274-275)***

One of the problems with Heap Sort is the dynamic storage allocation.

As you recall, a tree is a very space expensive structure with pointers.

We do not have to use pointers. We can use an array to represent a heap tree.

In the array, elements are stored in the **Breadth First** order.



e.g.



1



2 3



4 5 6 7



Is stored as 1 2 3 4 5 6 7



How will we find the children of a vertex??

Notice that the root is in slot 0.



Its children are in slot 1 and 2.



Slot 1’s children are in slot 3 and 4.

Slot 2’s children are in slot 5 and 6.

Slot’s 3’s children will be in slot 7 and 8.



**\*Inter6\* Thus slot I’s children are in slots ?? and ??**



**\*Inter7\* How do we remove the root from the array?**



**\*\*** [**Demonstrate the Priority Queue Visualizer**](http://cgi.csusm.edu/ryoshii/MyVisualizers/pqVis.html) **\*\***



**!!! Email me if you have any concern about the content so far!!!**

**Next page for the in-class EX!!**



**In-Class Exercise Week9B - Heap Sort Array - (MAX 2 people per team) 5 pts**

**PRINT names first:**

**Answer in the context of using an array. Q has Count elements.**

**1) With Count elements, the very last element is in which slot?**



**2) After the Q[0] is removed from the array, the item in which slot (in terms of Count) should replace it?**



**3) How do we then “trickle down” a value to its right place?**

* **Describe what is swapped with what.**



* **What is the for formula for finding the left child and the right child slots.**



* **When to stop (2 cases).**



**4) Where in the array do we add a new job? Which slot (in terms of Count)?**



**5) And how do we “trickle up” to the right place?**



* **Describe what is swapped with what.**
* **What are the formulas for finding the parent slot.**



* **When to stop (2 cases).**



**-------------------------------- break -----------------------------------------------------**

**\*\* GO OVER HW5 PROGRAM \*\* Pay attention to the Preparation Questions!**

**The End of Part 4. Trees:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Make sure you have written down the answers to all Inter questions.**

***®Summarize here what you have learned in your own words and also write down your own thoughts/reactions/questions.***

***Email me now if you have any questions about what you read in this file.***