Tree review

**complete** tree

* if it has height **h** the number of nodes is \_\_\_\_\_\_\_\_
* if it has **N** nodes, the number of leaves (nodes on the last level) is \_\_\_\_\_\_\_\_\_
* if it has N nodes, if each leaf keeps swapping with its parent, until it gets to the root, then one leaf will do \_\_\_\_\_\_\_\_\_\_\_ swaps and all leaves together will do \_\_\_\_\_\_\_\_\_\_\_\_\_\_ swaps.

**nearly complete** tree

* if it has height **h** the number of node N is: \_\_\_\_ ≤ N ≤ \_\_\_\_\_\_\_\_
* if it has **N** nodes, the height is \_\_\_\_\_\_
* if it has **N** nodes, the longest path has length \_\_\_\_\_\_
* if it has N nodes, the number of leaves is \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Array traversal using

* idx = idx/2

for(idx = N; idx >= 1; idx = idx/2) // repeats \_\_\_\_\_

* left = idx\*2, right = idx\*2+1

for(idx = 1; idx <= N; idx = idx\*2) // repeats \_\_\_\_\_

for(idx = 1; idx <= N; idx = idx\*2+1) // repeats \_\_\_\_\_

Priority Queues/Heaps:

What is a priority queue or a max-heap?

Uses:

Necessary operations:

Visualizing Heaps in Tree Format:

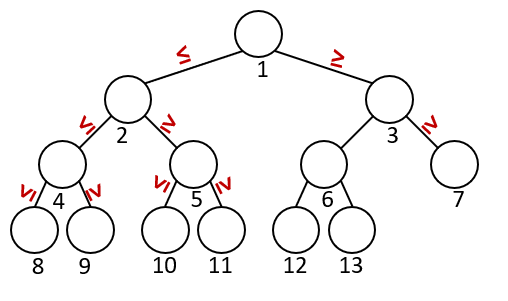
Drawing out heaps as a tree can make it much easier to understand the relationship certain elements have with each other based on the indexes they are stored at. However, note that a heap is NOT a \_\_\_\_\_\_\_\_. It is an \_\_\_\_\_\_\_\_\_\_\_\_ .

Required Tree Properties for a Max-Heap:

* Order of the Elements
* Values at children indexes/nodes are \_\_\_\_\_\_ than values at their parent indexes/nodes
* Items must respect order only along \_\_\_\_\_\_\_\_\_\_\_. Nodes in 2 subtrees have \_\_\_\_\_\_\_\_
* Shape of the Tree
* The tree is a \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
* There are no \_\_\_\_\_\_\_\_\_
* All levels are complete but possibly the \_\_\_\_\_\_\_\_\_\_\_\_\_\_
* If not complete, all nodes are to the \_\_\_\_\_\_\_\_ on the last level

Ex: Heap array

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| value | **-** | **9** | **7** | **5** | **3** | **5** | **4** | **3** | **2** | **1** | **1** | **3** | **4** | **1** |
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |



Index computation when 1st item is at index 1 (root is at index 1)

int left(int idx) {return idx\*2 ;}

int right(int idx) {return (idx\*2)+1 ;}

int parent(int idx) {return idx/2 ;}

E.g.:

left(4) -> \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

right(4) -> \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

parent(4)-> \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

left(5) -> \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

right(5) -> \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

parent(5)-> \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Inserting New Item:

1. Increase size of array and add a new element to end
2. Continually compare and swap new element with elements at the parent indexes until heap order is correct again (Perform swimUp())

swimUp(int\* A, int idx){

}

TC:

SC:

A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z

|  |  |
| --- | --- |
| Original Heap: | Add \_\_\_\_\_\_\_\_. Show the new heap below and affected/swapped elements on the original heap to the left. |

Remove() - will remove the top item

1. Swap the first element in the array with the last element in the array
2. Reduce the size of the array being dealt with (This is done not by physically altering the array but by altering the size input provided to the function so the last element in the array is ignored)
3. Continually compare and swap the new element at the beginning of the array with the element at one of the children indexes which has the largest value until heap order is correct again (Perform sinkDown())
4. Return the element stored at the end of the array

|  |  |
| --- | --- |
| TC: \_\_\_\_\_\_\_\_\_ SC: \_\_\_\_\_\_\_\_\_\_  int idxOfMaxValue(int\* A,int p,int le,int ri,int N){  } | TC: \_\_\_\_\_\_\_\_\_ SC: \_\_\_\_\_\_\_\_\_\_  sinkDown(int\* A, int p, int N){  } |

A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z

|  |  |
| --- | --- |
| Original Heap: | Remove \_\_\_\_\_\_\_\_. Show the new heap below and affected/swapped elements on the original heap t the left. |

Batch Initialization of Heap:

* Bottom Up Batch: Turns an array into a heap by reordering its elements to fit

requirements for heap order

buildMaxHeap(int\* A, int N) {

}

TC:

SC:

* Top Down Batch: Builds a heap by **repeated insertions** in an originally empty heap

TC:

SC:

Heapsort:

Implementation:

Code:

Heapsort(int\* A,int N){

}

TC:

SC:

Stable? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Adaptive? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
| Original Array |  |  |  |  |  |  |
| Heap |  |  |  |  |  |  |
| Removal 1 |  |  |  |  |  |  |
| Removal 2 |  |  |  |  |  |  |
| Removal 3 |  |  |  |  |  |  |
| Removal 4 |  |  |  |  |  |  |
| Removal 5 |  |  |  |  |  |  |

Variations:

What are the required adjustments to create a min heap instead of a max heap?

When would it be better to use a min-heap?

