CS 112: Data Structures

Sesh Venugopal

Heap - Implementation

Implementation: Structure for storage of heap items

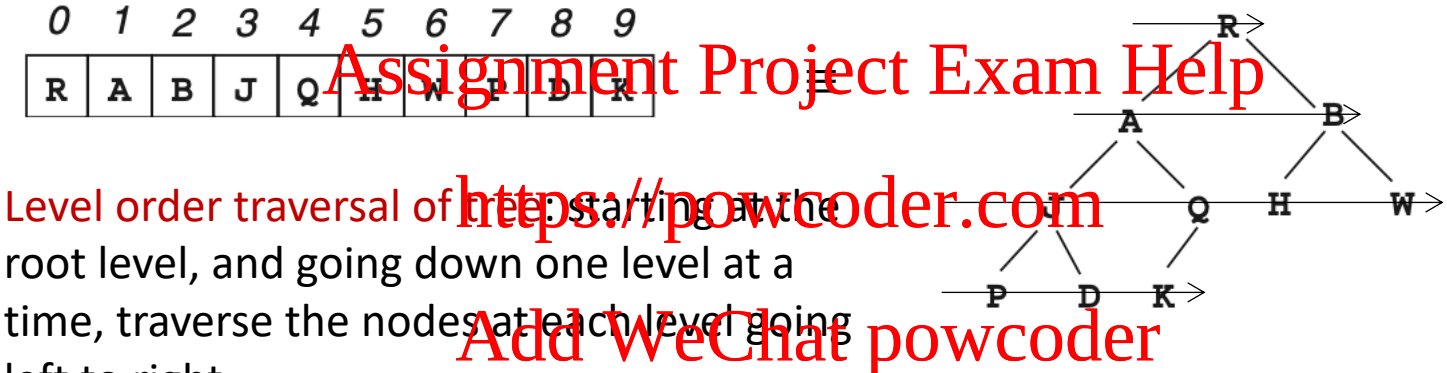
As far as the *conceptual* structure goes, the heap is a binary tree.

Therefore, one would expect that a heap could be implemented using a linked binary tree structure, as we did with BSTs

However, the fact that the heap is a *complete* binary tree has a very interesting consequence, and it is that *heap entries can be stored in an array.* This is quite a surprising twist!

Array Storage: Level Order Equivalence

The entries of a complete binary tree can be stored in an array in such a way that stepping through the array from beginning to end is equivalent to the level-order traversal of the tree.

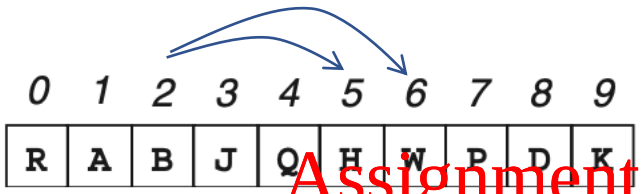


Level order traversal of tree: starting at the root level, and going down one level at a time, traverse the nodes at each level going left to right

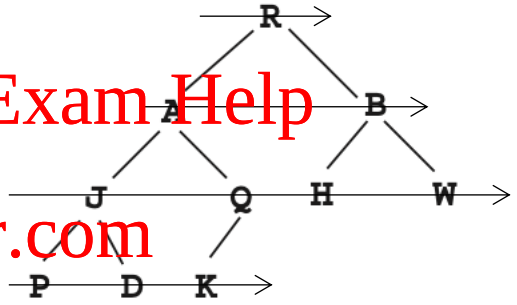
(It's not as if we actually build a tree then traverse and store in an array. This is just the conceptual equivalence – the array is the only storage structure in a heap implementation.)

Array Storage: Formula for Children and Parent Index

Since insertion/sift up and deletion/sift down work with the tree structure, and we have items stored in an array, how do we tell where a node's children are stored in the array, and where its parent is stored?



≡



B is at *k*=2, its children are at 2*2+1=5 (H), and 2*2+2=6 (W)

H is at *k*=5, its parent (B) is at (5-1)/2=2.
W is at k=6, its parent (B) is at (6-1)/2=2.5, truncated to 2

If a node is at index *k* of the array:

- Children are at indices 2*k*+1 and 2*k*+2
- Parent is at (*k*-1)/2 [integer division with truncation, as usual in Java]

If 2*k*+1 is outside the array bound, then the node at *k* is a leaf node
If 2*k*+1 is the last index in the array, then the node at *k* has a left child, but not a right child. This can only happen, if at all, for one node in the heap (e.g. Q)

In Resources, see `Heap.java` for a `Heap` class implementation, and `HeapApp.java` for a sample usage

Sorting using a heap

Back when we studied BST, we saw that we could sort a set of items by first inserting all of them one at a time in a BST, and then performing an inorder traversal of the BST. The inorder traversal would visit the items in the BST in sorted order

This sorting process (Treesort) runs in worst case $O(n^2)$ time

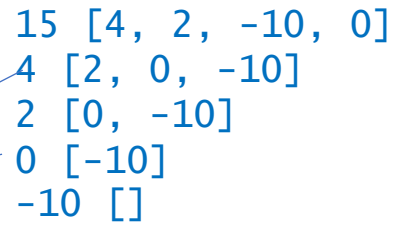
We can do a similar thing with the heap. To sort a set of items, insert them one at a time in a heap. When all inserts are done, perform a sequence of deletes. This gives back all the items in descending order.

Input: `4,2,15,0,-10` inserted one at a time into a heap:

`[4] , [4, 2], [15, 2, 4], [15, 2, 4, 0], [15, 2, 4, 0, -10]`

Delete from the heap one at a time:

The deleted items can be stored in an output array, back to front



Sorted Output:

`[-10, 0, 2, 4, 15]`

Sorting using a heap: worst case big O running time

What we know:

- Worst case time to insert in a heap: O(log n)
- Worst case time to delete from a heap: O(log n)

Inserting n items one at a time into a heap, running time:

log(1) + log(2) + log(3) + ... + log(n-1) + log(n)

(we'll put the big O back in at the end)

=> log(1*2*3...*(n-1)*n)

=> log(n!)

There is a result called **Stirling's formula** that approximates n! like this:

$$n! \approx (n/e)^n \sqrt{2\pi n}$$

Which essentially implies that

log(n!) = O(nlogn)

So the running time to insert all n items is O(nlogn)

Sorting using a heap: worst case big O running time

Deleting n items one at a time into a heap, running time:

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

This is exactly the same series as the inserts, just written in reverse order.
So the running time to delete all n items is $O(n \log n)$

So the total running time for the sort is

$$O(n \log n) + O(n \log n) = O(n \log n)$$


```
int[] arr = {42,15,20,10};

Heap<Integer> sortHeap = new Heap<Integer>(n);

// insert one at a time
for (int val: arr) {
     sortHeap.insert(val);
}

// delete one at a time, and store in result
for (int i=arr.length-1; i >= 0; i--) {
     arr[i] = sortHeap.delete();
}
```

Heapsort, max heap, min heap

We have a sorting technique that is as good as mergesort, with a worst case running time of O(nlogn)!

Later we will study a sorting algorithm called heapsort, which is a variation of this, and runs even faster in real time (although its big O running time is still nlogn)

By default, every heap is a MAX heap unless otherwise specified.

However, for certain applications, we want to invert the heap order so that the key at any node is less than or equal to the keys at the node's children. This is a MIN heap, and consequently, the least valued key would be at the top. (The Heap class code in resources has comments on how to set up a min heap instead of a max heap.)

If a min heap class is not handy, a max heap object can be effectively used as a min heap, by inverting every key before inserting, either as 1/key or -key