Heaps and Priority Queues

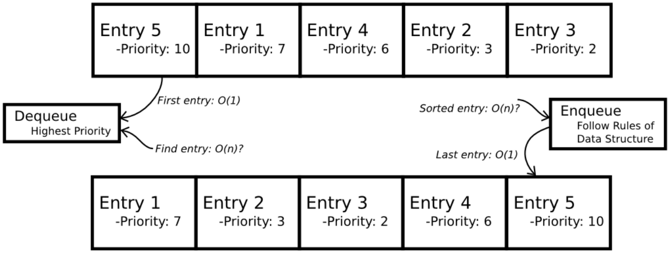
Cody Narber, M.S.
November 4, 2017

Data Structures - Priority Queue

Priority Queue: is a data structure that is like a regular queue in that can enqueue and dequeue elements, but those elements are accessed/stored by priority.

- **enqueue** - adds an element to the "end" of the queue
- **dequeue** - takes an element off the "front" of the queue

Example: At the implementation level, the data can be stored in multiple ways (Entries 1-5 presented in order):



Data Structures - Priority Queue

Given the data structures we have seen so far we could implement the priority queue is 1 of 4 ways:

1. **An unsorted list:** - whether that be array-based or reference-based is irrelevant
   - *enqueue* - add to the end of the list O(1)
   - *dequeue* - search through the entire list to find highest priority O(n)
2. **A sorted array-based list:**
   - *enqueue* - find where to place the new element, O(log_2n) using binary search, and place it shifting all subsequent elements O(n).
   - *dequeue* - As it is sorted, just take the first element

Data Structures - Priority Queue

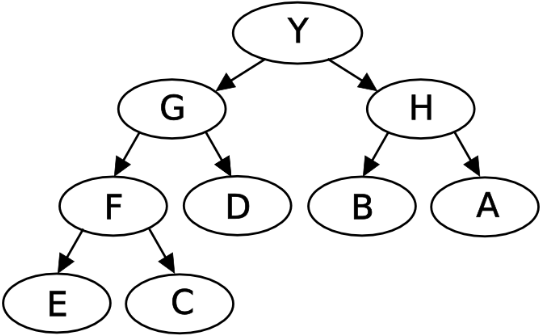
Given the data structures we have seen so far we could implement the priority queue is 1 of 4 ways (continued):

3. **A sorted reference-based list:**
   - *enqueue* - find where to place the new element which takes O(n), and then adjust links as necessary O(1)
   - *dequeue* - As it is sorted, just take the first element
4. **A binary search tree:**
   - *enqueue* - would be the standard `insert`, which can be O(log_2n) assuming it is a balanced tree, but could take as long as O(n).
   - *dequeue* - is the same as finding the maximum node (i.e. right-most), which again could be as little as O(log_2n) or as bad as O(n) if unbalanced. Then a `remove` operation is done which takes O(1), since we already found the node to be removed.

Data Structures - Heap

Heap: is a specialized binary tree that has the following properties:

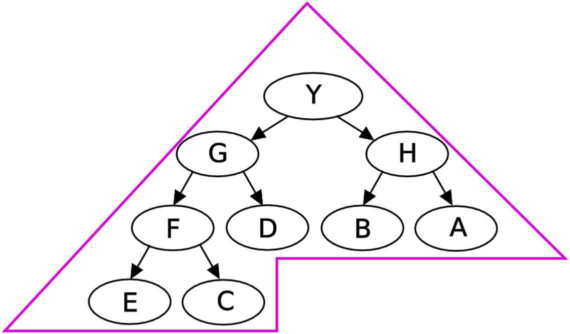
- **order property** - *that for every node in the tree, it is greater than or equal to all it's children.*
- **shape property** - must be a complete binary tree.



Data Structures - Heap

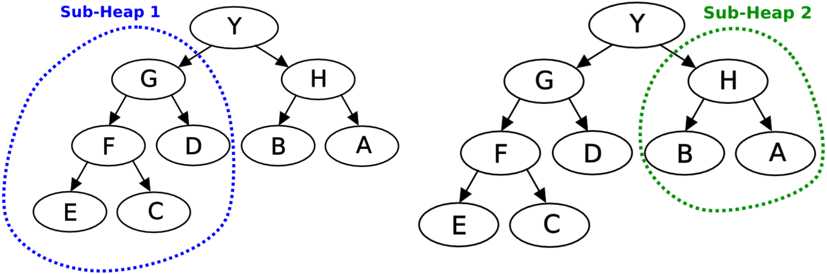
Heap: is a specialized binary tree that has the following properties:

- **order property** - that for every node in the tree, it is greater than or equal to all it's children.
- **shape property** - *must be a complete binary tree.*



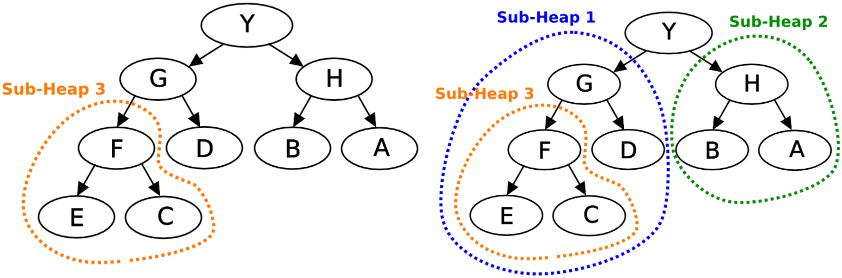
Data Structures - Heap

Provided the **order** and **shape** properties of the heap we can notice that all sub-trees follow the same properties such that each sub-tree is also a heap.



Data Structures - Heap

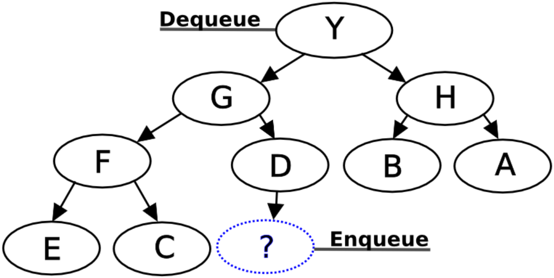
Provided the **order** and **shape** properties of the heap we can notice that all sub-trees follow the same properties such that each sub-tree is also a heap.



Priority Queue - Heap

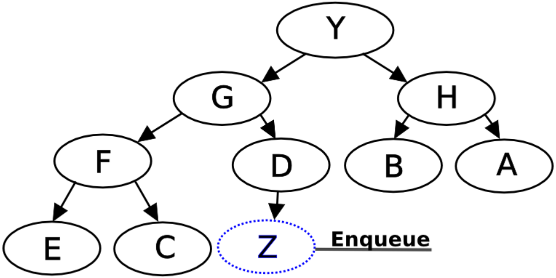
Maximum Heaps (those where the root is max, can easily be inverted to a Minimum Heap) are designed specifically to work as a priority queue. Where:

- Enqueing will place the next element at the next available leaf location to maintain order.
- Dequeueing will remove the root as it is the largest element (largest based on sorting criteria).



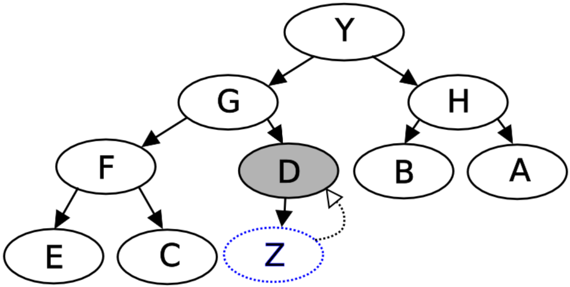
Priority Queue - Heap (Enqueue 1)

Enqueuing onto the heap adds a new element to the next available spot. In the case of a heap it is the next location that would preserve completeness...*However this may not perserve the order property*



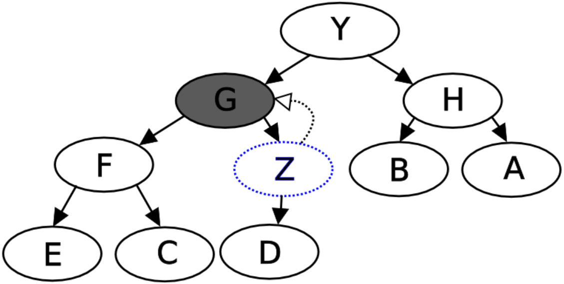
Priority Queue - Heap (Enqueue 2)

The order property states that the parent must be larger than it's children therefore if the child is larger we can swap the parent and child to perserve order in the subtree (we know this will hold the order property for all other children due to the transitive property e.g. *right* $>$ *parent* $>$ *left*).



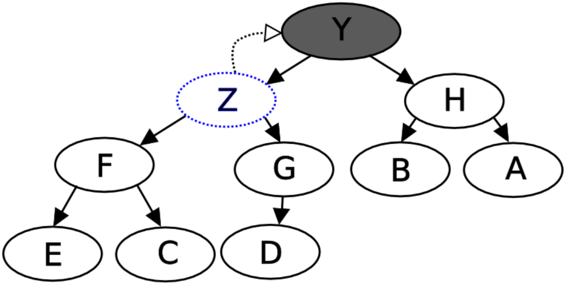
Priority Queue - Heap (Enqueue 3)

Repeat: The order property states that the parent must be larger than it's children therefore if the child is larger we can swap the parent and child to perserve order in the subtree (we know this will hold the order property for all other children due to the transitive property e.g. *right* $>$ *parent* $>$ *left*).



Priority Queue - Heap (Enqueue 4)

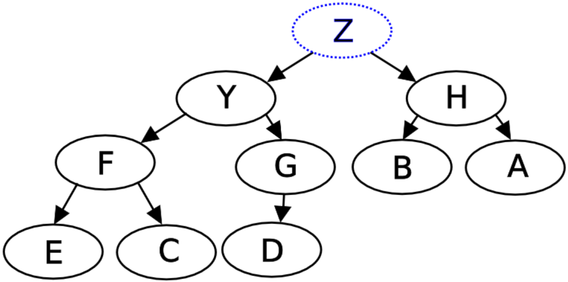
Repeat: The order property states that the parent must be larger than it's children therefore if the child is larger we can swap the parent and child to perserve order in the subtree (we know this will hold the order property for all other children due to the transitive property e.g. *right* $>$ *parent* $>$ *left*).



Priority Queue - Heap (Enqueue 5)

We stop swapping due to one of two cases:

- There is no parent (as in this case)
- The parent has higher priority than the node added (order property now holds)



Heap Enqueue Implementation

Provided that we have the BTNode as noted in the UML:

- Update Last Available spot to be new node.
- Swap added node with parent if greater.
- Repeat swapping added node as needed.

BTNode<T>
- data : T - parent : BTNode<T> - left : BTNode<T> - right : BTNode<T>
- set...(...) - get...() : ... - rmLink(n : BTNode<T>) - cpyChildren(n : BTNode<T>) - swapLinks(n : BTNode<T>)

Heap Enqueue Implementation

BTNode<T> Link Management Functions:

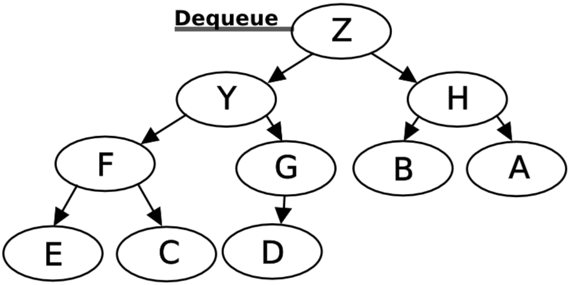
```java
public void rmLink(BTNode<T> n){
        if(left == n) left = null;
        if(right == n) right = null;
        if(parent == n) parent = null;
}
public void cpyChildren(BTNode<T> n){
        n.setLeft(left);
        n.setRight(right);
}
public void swapLinks(BTNode<T> n){
        BTNode<T> l = left;
        BTNode<T> r = right;
        BTNode<T> p = parent;
        left = n.getLeft();
        right = n.getRight();
        parent = n.getParent();
        n.setLeft(l);
        n.setRight(r);
        n.setParent(p);
}
```

Heap Enqueue Implementation

```
// Inside Our Heap - root, lastNode, lastAvail
public void enqueue(T datum){
        BTNode<T> node = new BTNode(datum);
        //Method that can just use in order traversal to find
        lastAvail = findLastParent();
        if(lastAvail.getLeft()==null){
                lastAvail.setLeft(node);
                node.setParent(lastAvail);
        } else if(lastAvail.getRight()==null){
                lastAvail.setRight(node);
                node.setParent(lastAvail);
        }
        swapUp(node);
}
public void swapUp(BTNode<T> node){
        if(node.getParent()==null) return;
        BTNode<T> parent = node.getParent();
        if(node.compareTo(parent)>0){
                node.swapLinks(parent);
                swapUp(node);
        }
}
```

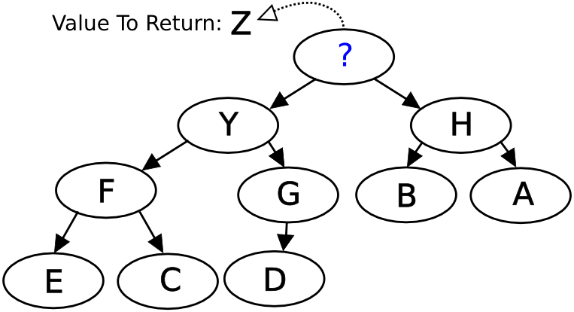
Priority Queue - Heap (Dequeue 1)

Dequeueing the root will remove it so that we can return the value stored. *Both order and shape are not held...we have a hole!*



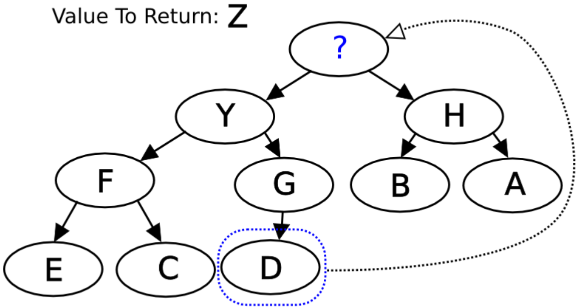
Priority Queue - Heap (Dequeue 2)

Dequeueing the root will remove it so that we can return the value stored. *Both order and shape are not held...we have a hole!*..Our tree now has one less node so once we reorganize the current far right leaf will not be there in the end, therefore...



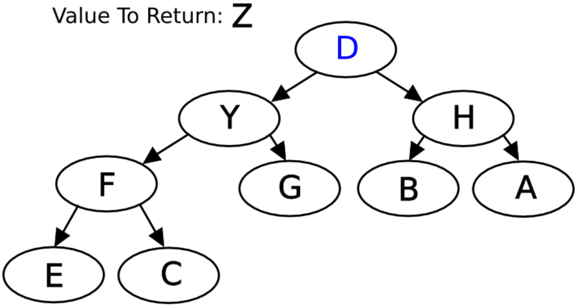
Priority Queue - Heap (Dequeue 3)

We move the far right leaf node into the root's spot, now we have satisfied the **shape** property! *However, our order property does not hold...we can swap again!*



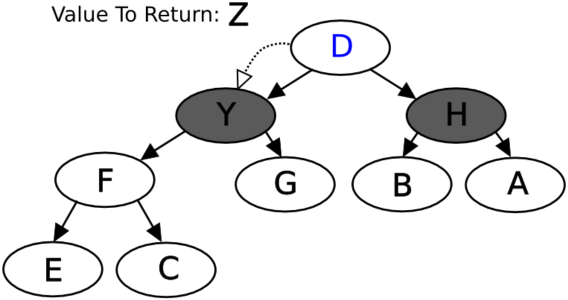
Priority Queue - Heap (Dequeue 4)

We move the far right leaf node into the root's spot, now we have satisfied the **shape** property! *However, our order property does not hold...we can swap again!*



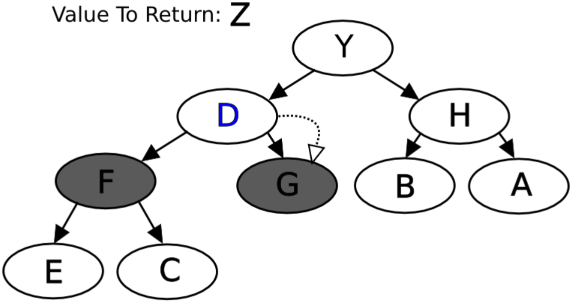
Priority Queue - Heap (Dequeue 5)

Swap the added node with the largest child. By using the larger of the two children we will be preserving the order when swapped.



Priority Queue - Heap (Dequeue 6)

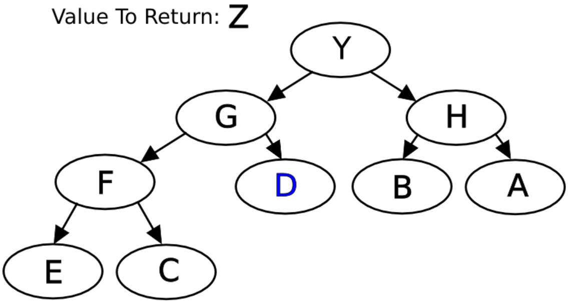
Repeat: Swap the added node with the largest child. By using the larger of the two children we will be preserving the order when swapped.



Priority Queue - Heap (Dequeue 7)

We stop swapping due to one of two cases:

- There is more children (as in this case)
- Both children have lesser priority than the node added (order property now holds)



Heap Dequeue Implementation

Provided that we have the BTNode as noted in the UML

- Store the root value to be returned when done.
- Update Last Element to be root
- Repeat swapping added node as needed.

BTNode<T>
- data : T
- parent : BTNode<T>
- left : BTNode<T>
- right : BTNode<T>
- set...(...)
- get...() : ...
- rmLink(n : BTNode<T>)
- cpyChildren(n : BTNode<T>)
- swapLinks(n : BTNode<T>)

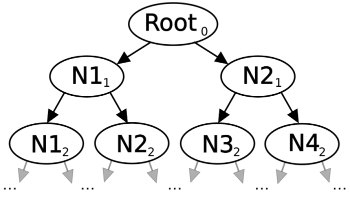
Heap Dequeue Implementation

```
// Inside Our Heap - root, lastNode, lastAvail
public T dequeue(){
        T datum = root.getData();
        lastNode.getParent().rmLink(lastNode);
        lastNode.cpyChildren(root);
        root = lastNode;
        swapDown(root);
        return datum;
}
public void swapDown(BTNode<T> node){
        BTNode<T> left = node.getLeft();
        BTNode<T> right = node.getRight();
        if(node.compareTo(right)<0 && left.compareTo(right)<0){
                node.swapLinks(right);
                swapDown(node);
        }
        if(node.compareTo(left)<0 && right.compareTo(left)<0){
                node.swapLinks(left);
                swapDown(node);
        }
}
```

Array Representation

In order to store a binary tree into an array we know how many nodes are possible on each level:

- Level 0: 1
- Level 1: 2
- Level 2: 4
- Level 3: 8
- Level n: 2^n

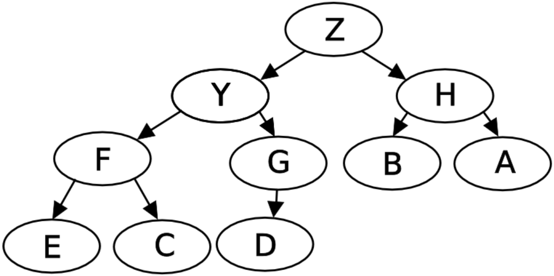


We can unwrap a tree such that every possible node is concatenated into an array:

Lvl 0	Lvl 1		Lvl 2				Lvl 3...		
$Root_0$	$N1_1$	$N2_1$	$N1_2$	$N2_2$	$N3_2$	$N4_2$	$N1_3$	$N2_3$	...

Array Representation

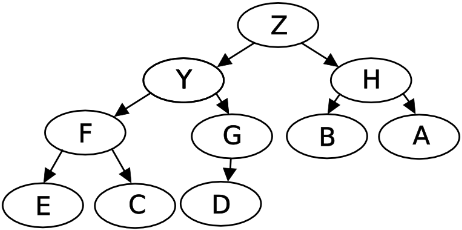
One of the nice things with Heaps, is that they are guarenteed to be complete; therefore there will be no holes! and we can easily maintain the last element and next available spots!



0	1	2	3	4	5	6	7	8	9	10
Z	Y	H	F	G	B	A	E	C	D	

Array Representation

We can also see a relationship between array locations and children:



0	1	2	3	4	5	6	7	8	9	10
Z	Y	H	F	G	B	A	E	C	D	

- $Parent(n) = (n - 1)/2$
- $Left(n) = 2 * n + 1$
- $Right(n) = 2 * n + 2$

Big-Oh Comparisons

Priority Queue Representations: (dequeue gets largest)

Methods	Heap	BST*	Array LL**	Single LL
Constructor	O(1)	O(1)	O(n)	O(1)
enqueue				
–find	O(1)	$O(log_2(n))$	O(n)	O(1)
–process	$O(log_2(n))$	O(1)	O(n)	O(1)
–total	$O(log_2(n))$	$O(log_2(n))$	O(n)	O(1)
dequeue				
–find	O(1)	$O(log_2(n))$	O(1)	O(n)
–process	$O(log_2(n))$	O(1)	O(1)	O(1)
–total	$O(log_2(n))$	$O(log_2(n))$	O(1)	O(n)

**Assuming Always Balanced*

***Maintain Priority Sort on Enqueue*