Lab : Heaps

Information

In-class labs are meant to introduce you to a new topic and provide some practice with that new topic.

Topics: Heaps

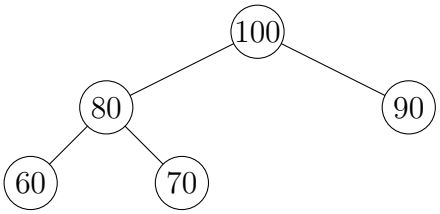
Solo work: Labs should be worked on by each individual student, though asking others for help is permitted. Do not copy code from other sources, and do not give your code to other students. **Students who commit or aid in plagiarism will receive a 0% on the assignment and be reported.**

Turn in: Print out the worksheet and fill it out. Upload pictures of your work to Canvas once done.

Starter files: None

Contents

1.1 Intro to heaps



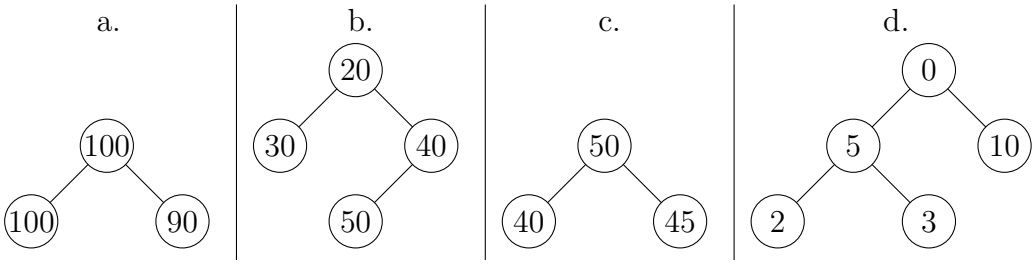
Heaps are a type of tree structure, though they are commonly implemented as arrays. Conceptually, we look at them as trees.

A heap must be a **complete binary tree**, and if the heap is not empty, then the **subtrees of the root note's children must also be heaps**.

If a heap is a **maxheap**, then the root has the greatest value. If a heap is a **minheap**, then the root has the smallest value. Note that a child node may also share this greatest/smallest value, which is fine, as long as the root contains the greatest/smallest value as well.

1.1.1 Identifying heaps

Question 1
For each of the following, identify whether the graph is a minheap, a maxheap, or not a heap at all. If it is not a heap, give a reason why not.



1.1.2 Indices of heap nodes

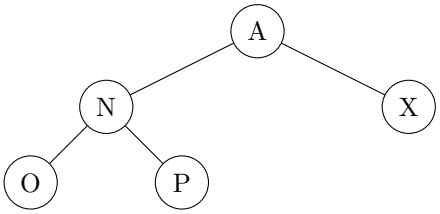
When a heap is implemented in a array structure, node indices are set based on these rules:

- For a node i, if it has a left child, the child is at position $[2i + 1]$.
- For a node i, if it has a right child, the child is at position $[2i + 2]$.
- For a node i, its parent will be at position $[(i - 1)/2]$.
- Only the root (at index 0) doesn't have a parent.

Question 2
Given the array, draw the corresponding heap.

0	1	2	3	4	5	6
BR	GA	KL	PB	TS	UK	TN

Question 3
Given the heap, fill out the array.



0	1	2	3	4

1.2 Heap functionality

1.2.1 Retrieving an item from a heap

As we retrieve items from a heap, we will be returning the max/min value, which will be at the root - at index 0.

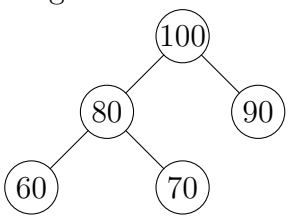
```
Top() { return item[0] }
```

1.2.2 Removing an item from a heap

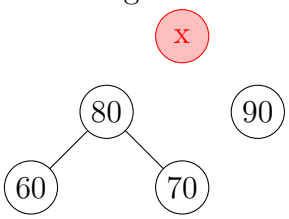
When we remove an item from the heap, we only remove the item at position 0: the min/max value.

However, if you do this, a hole in the heap is created, so this is not actually the strategy we use to remove the min/max item...

Original



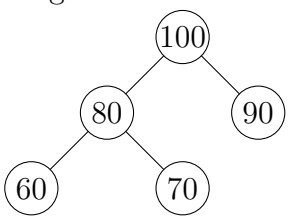
Removing root



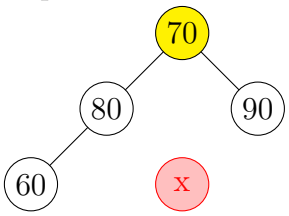
Instead, we want to keep the heap in a valid **complete binary tree** state. To do this, we remove the last node (the last index of the array), copying its value to overwrite the root.

0	1	2	3	4
100	80	90	60	70

Original



Replace root

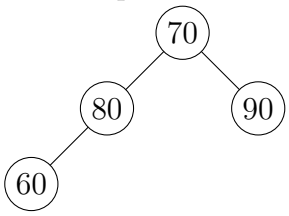



```
Pop() {
    item[0] = item[size-1];
    size--;
    BubbleDown( 0 );
}
```

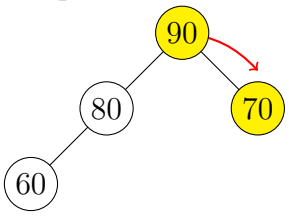
After this operation, the heap is not in a valid heap state because the root is no longer the min/max value. This state is called a **semiheap**.

At this state, we will begin swapping values between the root and a child until we're at a valid heap state. If the root value is smaller than both its children, then you swap the root and the larger item (for a maxheap). You stop swapping once the "old-root" is greater than or equal to both its children (or less than/equal to for a minheap.)

Semiheap



Swap



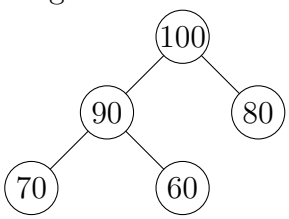

```
BubbleDown( index ) {
    if root isn't only node...
        find larger child: Left or right?
        swap( root, larger child )
        BubbleDown( larger child index )
}
```

1.2.3 Adding items to a heap

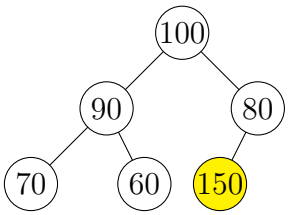
When we add a new item to the heap, we place the new item at the bottom of the tree (filling from left-to-right, because it's a complete binary tree). After that, we begin bubbling up to find its proper place in the heap.

0	1	2	3	4
100	90	80	70	60

Original



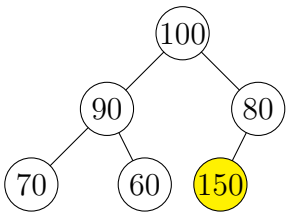
New item



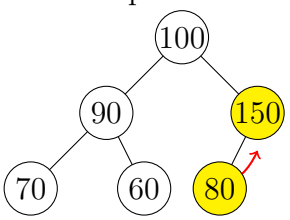

```
Push( data ) {
    items[size] = data
    BubbleUp( size );
    size++;
}
```

We look at the parent at each step, and if it's less than (for a maxheap), we swap the new data with that node.

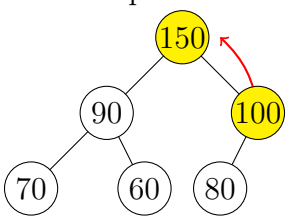
New item



Bubble up once



Bubble up twice

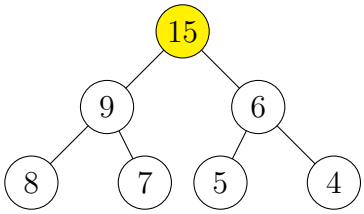



```
BubbleUp( index ) {
    parent = (index - 1) / 2

    if ( heap[index] > heap[parent] )...
        swap( heap[index], heap[parent] )
        BubbleUp( parent )
}
```

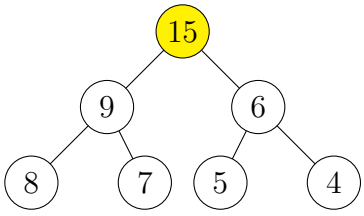
Question 4

Draw the tree at each step of the remove / bubble down process until the heap is restored to a valid state. Remember that Pop() will remove the value at the root.



Question 5

Draw the tree at each step of the add / bubble up process until the heap is restored to a valid state.




```
Add( 10 )
```