Heaps and Priority Queues

CS 240 Spring 2019

Problem

Assume you are looking for a job, and you have been given job offers from 50 different companies. Each company wants you to come and do an onsite visit, whenever you would like, but you must decide if you will take the job immediately upon concluding the visit. How would you sort which companies you would visit first?

Priority Queue

- What if we want to access data similar to a stack or queue, but organized by importance or priority
  - we call this a priority queue.
- Instead of being a "First In First Out" or “Last In First Out”, values come out in order by priority.
  - Like a Stack or Queue, you only ever access one element at a time

Problem

- Let's put our job offers into data structures we've seen so far
    - Using an Array
        - *sorted order makes insert slow, removePriority fast*
        - *arbitrary order makes insert fast, removePriority slow*
    - Using a Linked List
        - *Sorted makes insert O(n)= cn (linear)*
        - *Unsorted makes removePriority O(n)= cn (linear)*
    - Using a BST
        - *removePriority is O(logn) IF the tree stays balanced*
        - *If the tree is not balanced, insert and removePriority can be O(N).*

Balanced and Complete Trees

- Full Tree
  - A full binary tree is a tree in which every node other than the leaves has two children. There are no single node parents.
- Balanced Tree
  - The tree’s height is such that no leaf is more than one level away from any other leaf in the tree
- Complete Tree
  - A tree in which every level, except the last, is filled, and all nodes are as far left as possible.

Solution

- A balanced BST will give the best performance for a priority queue
- More Problems:
  - Read Priority operation still has to traverse the left branch
    - *Make read constant time by making the priority value the root*
  - BST structure uses strict ordering
    - *All values can only go one place which makes keeping the tree balanced difficult*

A tree that isn't a BST

- We need a data structure that allows:
    - partial ordering
    - always has the priority value as the root
- A **Heap** is:
    - a complete binary tree
        - *nearly always implemented using the array representation*
    - The values in the tree maintain a parent/child relationship only.
        - *No defined relationship with the tree as a whole*
        - *This is called partial order*

Array Based Tree

- Mapping elements of a tree into an array
  - if a node is stored at index k
    - *the left child is at index 2k+1*
    - *The right child is at index 2k+2*
    - *The parent is (k-1)//2 #integer division*
- **Assertion:** Maintaining a balanced tree is easier with an array based implementation.
  - Agree?
    - *Don’t confuse the logical representation of a heap (tree) with its implementation (array).*

Heap Variants

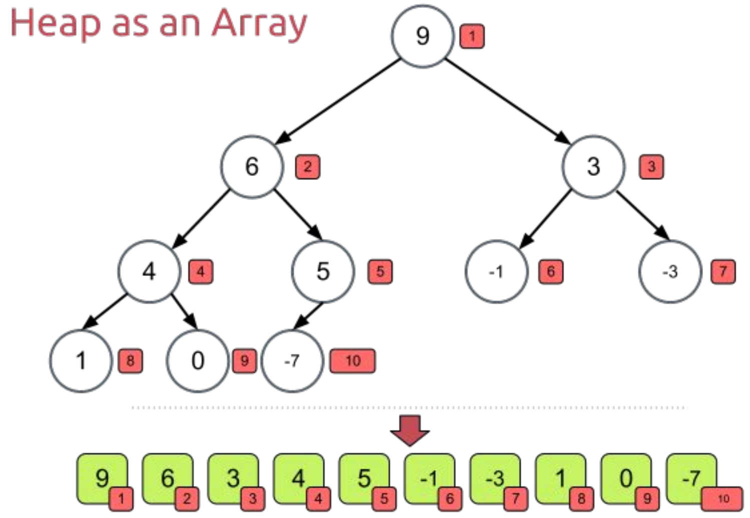
- There are two variants of the heap
  - In a **max heap**, every node stores a value that is greater than or equal to the value its children.
    - *Because the root has a value greater than or equal to its children, which have values greater than or equal to their children, the root stores the maximum of all values in the tree.*
  - In a **min heap**, every node stores a value that is less than or equal to that of its children.

Node Relationships

- There is no relationship between the value of a node and that of its sibling in a heap
  - For example, the values for all nodes in the left subtree of the root could be greater than the values for every node of the right subtree.
    - *This is a feature, not a bug*
- Contrast with a BST which has a strict ordering relationship

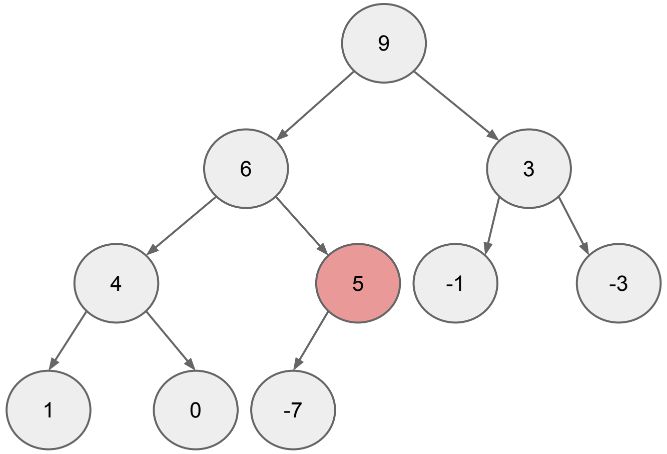
Creating A Heap

- Given an array of N values, a heap can be built by “sifting” each node down to its proper place
  - Any array can be made into a heap using the ‘Heapify’ sorting algorithm



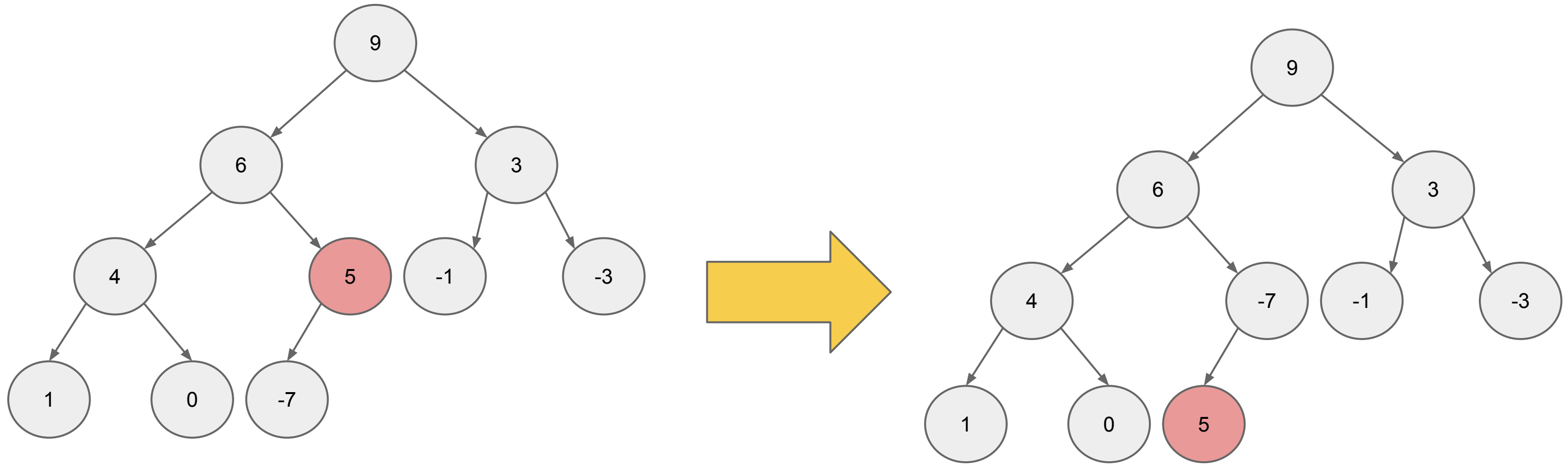
Where to Start?

- Start with the last internal node
  - How do we find the last internal node (non-leaf)?
    - *Take the last index, then find parent: (i-1)//2*
- Swap the current internal node with its smaller child, if necessary



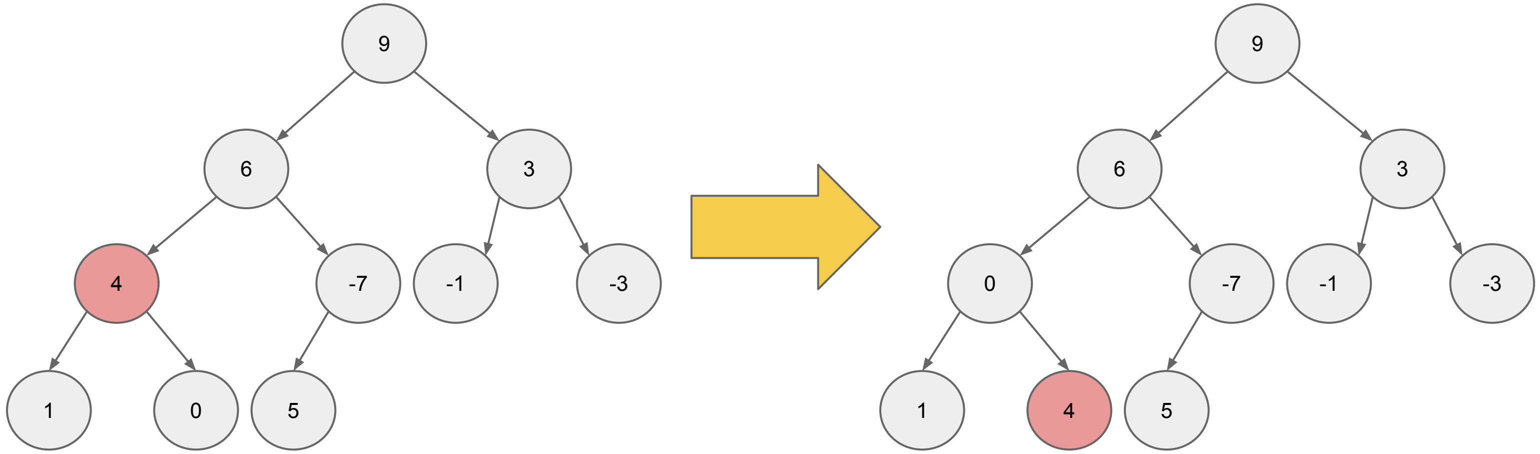
Sift Down

- Follow the swapped node down the tree until both children are larger



Sift Down

- Go to the next internal node. Repeat until all internal nodes are done
  - Check both children for the smaller value, swap with the smaller node



Classwork

From BSTs to Heaps

the Heap Instance Variables

```
// Min-heap implementation
        class MinHeap{
            private:
                Comparable  * heap; //Pointer to an array of comparables
                int n;          // Number of things now in heap
                int max; //maximum size of the heap
                …
```

If you use an STL container, such as a vector, that would be all you need for private instance variables

Constructing the Heap

- //Constructor supporting preloading of heap contents

  -
    ```
    MinHeap(Comparable * h, int num, int max){
        Heap = h;
        n = num;
        size = max;
        buildheap(); //creates the heap data structure
    }
    ```

Constructing an Array Based heap

- Though not required, you should have methods that return pointers to or indexes of parents and children

    -
      ```
      int left(i){
          if(2i + 1 > n)
              return -1;
          else
              return 2i + 1;
      }
      ```

Heapify

- //Heapify the array elements

  -
    ```
    void buildheap(){
        for (int i=(n-2)/2; i>=0; i--)
            siftdown(i);
    }
    ```
    - *Why is 'i' initalized like this?*
      - Because n is 1 more than the last index of the array
- Buildheap will run through (almost) every element of the array
  - O(n) to heapify

SiftDown

- // Put element in its correct place

```
void siftdown(int pos) {
    if ((pos < 0) || (pos >= n)) return; // Illegal position
    while (!isLeaf(pos)){ //Keep swapping until you get to a leaf
        int j = left(pos); //Get left child
        if ((j+1 < n && (heap[j] > heap[j+1]))
            j++; // j is now index of child with greater priority
        if (heap[pos] < heap[j]) return; //pos is in correct position
            swap(heap[pos], heap[j]);
        pos = j;  // Move down
    }
}
```

Cost of SiftDown

- In a complete binary tree of N nodes, the number of levels is at most 1 + log(N).
    - Each non-terminating iteration of the loop moves the target value a distance of 1 level, the loop will perform no more than log(N) iterations.
- Thus, the worst case cost of SiftDown() is O(logn)

Total Cost of Building a Heap

- Suppose we start with a complete binary tree with N nodes
  - The max number of steps required for sifting values is N = 2d-1 for some integer d = log N.
- In the worst case, the number of swaps BuildHeap() will require in building a heap of N nodes is N-logN
  - So we can say building a heap costs T(n) = cn

Removing the Priority Value

- What happens when we remove the priority value?
  - The priority value is stored at the root
- Choose the last leaf to replace the root, then sift down
  - Why choose the last leaf?

```
Comparable removePriority(){
    //Check for empty heap
    if (numVertices == 0)
        return ;
    //Swap the root with last leaf
    Comparable priority = heap[0];
    heap[0] = heap[n - 1];
    heap[n - 1] = priority;
    //shrink heap by one node
    n--;
    //sift new root down
    siftDown(0);
    return priority;
}
```

Classwork

`isHeap()`

Classwork

Balanced vs Complete