Heaps

Priority Queues

A special type of queue that allows us to keep a prioritized list of items

In a priority queue, each item you insert into the queue has a "priority rating"

Dequeue: dequeues the item with the highest priority

Operations

Insert new item

Get value of highest priority item

Remove highest priority item

Heap

The **Heap** data structure is one of the most efficient ones we can use to implement a Priority Queue

Uses a special type of **binary tree** to hold its data

- It is NOT a binary tree

Heap

A heap is a tree-based data structure used to implement priority queues and do efficient sorting

Two types of heaps:

1. minheaps: the smallest item is always at the tree's root
    - everytime you extract or add an item, you update the tree to maintain this property
    1. Quickly insert a new item into the heap
    2. Quickly retrieve the smallest item from the heap
2. maxheaps: the largest item is at the root
    1. Quickly insert a new item into heap
    2. Quickly retrieve the largest item from the heap

Efficient

- Fast to keep the proper item at the top during extraction/insertion: **O(log2n)**

All Heaps use a "Complete" binary tree

Complete Binary Tree

1. The top N-1 levels of the tree are completely filled with nodes
2. All nodes on the bottom-most level must be as far left as possible
    - no empty slots between nodes

The Maxheap

A maxheap is a binary tree which follows these rules:

1. The value contained by a node is ALWAYS GREATER THAN OR EQUAL TO the values of the node's children
2. The tree is a COMPLETE binary tree
3. The biggest item is always at the root of the tree

Extract Biggest Item

Steps

1. If the tree is empty, return error
2. Otherwise, top item in the tree is the biggest value.
    - Remember it for later
3. If heap has only one node, then delete it and return saved value
4. Copy the value from the right-most node in the bottom-most row to the root node
5. Delete the right-mostnode in the bottom-most row
6. Repeatedly swap the just-mosed value with the larger of its two children until the value is greater than or equal to both of its children
    - sifting down
7. Return saved value to user

Adding a Node to Maxheap

1. If the tree is empty, create a new root node & return
2. Otherwise, insert the new node in the bottom-most, left-most position of the tree (soit's still a complete tree)
3. Compare the new value with its parent's value
4. If the new value is greater than its parent's value, then swap them
5. Repeat steps 3-4 until the new values rises to its proper place

---

Heap Implementation

Classic Binary Tree Node

- Node easy to locate the bottom-most right-most node during extraction, bottom-most left-most spot to insert, locate a node's parent during swaps

Use an array

Copy Our Node a Level at a Time Into Array

```
[12, 7, 10, 3, 2, 8, 4, 2]
```

Use a `int` variable to track how many items are in our heap

Properties

1. Find the root value in `heap[0]`
2. Find the bottom-most, right-most node in `heap[count - 1]`
3. Find the bottom-most, left-most empty spot (to add a new value) in `heap[count]`
4. Add or remove node by using array `heap[count] = value`

Locating left and right child of a node

leftChild(parent) = 2*parent + 1

rightChild(parent) = 2*parent + 2

- To find the left child of node 7: leftChild(1) = 2 * 1 + 1 = slot 3
- Use indexes

Find parent

(child - 1) / 2 = parent

Summary

1. The root of the heap goes in array[0]
2. If data for a ndoe appears in array[i], its children, if they exist, are in these locations

   Left child: array[2i + 1]

   Right child: array[2i + 2]
3. If the data for a non-root node is in array[i], then its parent is always at array[(i-1)/2]

Heap Helper Class

```
class HeapHelper
{
    HeapHelper() { num = 0; }
    int GetRootIndex() { return (0); }
    int LeftChildLoc(int i ) {return (2*i + 1); }
    int RightChildLoc(int i) {return (2*i + 2); }
    int ParentLoc(int i ) {return ((i-1)/2); }
    int PrintVal(int i ) { cout << a[i]; }
    void AddNode(int v) { a[num] = v; ++num; }
private:
    int a[MAX_ITEMS];
    int num;
};
```

```
main()
{
    HeapHelper a;
    a.AddNode(123);
    a.AddNode(42);
    a.AddNode(-7);
    a.AddNode(999);
    a.AddNode(314);

    int i = GetRootIndex();
    PrintVal(i);
    i = LeftChildLoc(i);
    PrintVal(i);
    i = RightChildLoc(i);
    PrintVal(i);
}
```

Implement a MaxHeap

Extraction

1. If the `count==0` (empty tree), return error
2. Otherwise, `heap[0]` holds the biggest value. Remember it for later.
3. If the `count==1` (that was the only node) then `set count=0` and return the saved value
4. Copy the value from the right-most, bottom-most node to the root node: `heap[0] = heap[count-1]` (the last element)
5. Delete the right-most node in the bottom-most row: `count=count-1`
6. Repeatedly swap the just-moved value with the larger of its two children:

    - Starting with `i=0`, compare and swap: `heap[i]` with `heap[2*i+1]` and `heap[2*i+2]`

7. Return the saved value to the user

Insertion

1. Insert a new node in the bottom-most, left-most open slot:

```
heap[count] = value;
count = count + 1;
```

2. Compare the new value `heap[i]` with its parent's value: `heap[(i-1)/2]`

3. If the new value is greater than its parent's value, then swap them

4. Repeat steps 2-3 until the new value rises to its proper place or we reach the top of the array

Complexity of Heap: `log2(N)`

Insertion

- Every time we insert a new item, we need to keep comparing it with its parent until it reaches the right spot
- Since tree is a complete binary tree, if it has N entries, guaranteed to be exactly log2N levels deep
    - Hence worst case, log2N comparisons and swaps of new value

Extraction

Just as with heap insertion when we extract a value we need to bubble an item from the root down the tree

Maximum number of levels in our tree is `log2N` -> `log2N` swaps

Heapsort

Heapsort is $O(n*log2n)$ sort that uses a **maxheap** to sort a bunch of values

Idea:

- Start by taking the array that you want to sort and converting it into a maxheap
- Repeatedly remove the largest item from the maxheap and store it back into the array
- The first item removed from the maxheap goes into the last slot of the array, the next item removed goes into the second-to-last slot etc.
- The final item (the smallest one) goes in slot 0
- Once you've removed all N items from the heap, and put them back into the array -> sorted

Naive Way to Sort Using heap

Given an array of N numbers:

1. Insert all N numbers into a separate new maxheap
    - insertion one by one and reordering
2. While there are numbers left in the heap:
    - Remove the biggest value from the heap
    - Place it in the last open slot of the array

Complexity: **$O(Nlog2N)$**

- Cost of inserting an item into a maxheap
    - N items * log2N steps per item
- Cost of extracting an item from maxheap
    - N items * log2N steps per item

Efficient Heapsort

Avoid creating a new maxheap and moving numbers back and forth

Algorithm:

1. Convert our input array into a maxheap
2. While there are numbers left in the heap
    - Remove the biggest from heap
    - Place it in the last open slot of array

Step 1: Convert randomly-arranged input array into a maxheap

```
startNode = N/2 -1
for (curNode = startNode thru rootNode):
    focus on the subtree rooted at curNode

    think of this subtree as a maxheap

    keep shifting the top value down until your subtree becomes a valid maxheap
(using normal heapification algorithm)

        keep swapping our root value down with its larger child until it's bigger
than both of its children, or hits a leaf
```

- `startNode` locates the lowest, right-most node in the tree that has **at least one child**

- allows us to skip all of the single-element trees
- As we heapify higher sub-trees, they rely upon the lower sub-trees that were heapified earlier

Step 2

While there are numbers left in the heap:

1. Extract the biggest value from the maxheap and re-heapify
    - Copy right most bottom most node to root node
    - Delete the right-most node in the bottom-most row
    - Repeatedly swap the just-moved value with larger of two children until value is greater than or equal to both children
    - When we finish reheapifying, first N -1 slots hold a valid maxheap AND teh least slot of the array is empty
2. This frees up the last slot in the array
3. Put the extracted value into the freed slot of the array

Complexity of Heapsort: **$O(Nlog2N)$**

1. First take N-item array and convert it into a maxheap
    - Convert successively larger subtrees into maxheaps by visiting each "node" or element in the array once
    - O(N)
2. Repeatedly extract the jth largest item from the maxheap and place that item back into the array, j slots from the end
    - O(Nlog2N)
    - Each time we extract: log2N steps, we perform this N times

- Hence O(N + Nlog2N) and ignore the N