ECE 242
Data Structures and Algorithms

http://www.ecs.umass.edu/~polizzi/Teaching/ECE242/

Heaps I
Lecture 22

Prof. Eric Polizzi
Motivations

- **Review of priority queue**

Data structure that offers convenient access to the data item with the largest (or smallest) priority key.

Useful for task scheduling in computers, etc.

It is a Abstract Data Type (ADT) that can be implemented using various underlying data structures. We have used an 'ordered array' in Lecture 9.

Problem with ordered array: dequeue (removal) is $O(1)$ but enqueue (insertion) is $O(N)$

**Solution**: use a heap implementation
Motivations: Insertion and Removal for existing data structures

- Priority Queue (ordered array)
  
  <+> Removal of largest (smallest) key: O(1)
  
  <-> Insertion O(N)

- Binary Search Tree
  
  <+> Insertion and Removal O(logN)
  
  <-> degenerate to O(N) for unbalanced trees

- Red-Black Tree
  
  <+> Insertion and Removal O(logN)
  
  <-> Complex rebalancing necessary

- Heap
  
  <+> Insertion and Removal of largest (smallest) key: O(logN)
  
  <+> Easy to implement using an array
  
  <+> Heapsort algorithm
  
  <-> No easy searching, No sorted traversal
Introduction to Heap

- It is a binary tree with three characteristics
  - 1- It is a CBT (Complete Binary Tree)
  - 2- It is usually implemented as an array

Remark: Heap (ADT) → CBT → Array implementation

- 3- Each node satisfies the heap condition: every node's key is larger than (or equal to) the keys of its children.

- Heap or not Heap?

![Binary Tree Diagram]

- Diagram of a complete binary tree with nodes labeled 7, 13, 14, 23, 17, 27.
Heap: Array-based representation

- Since a Heap is a CBT, there is no 'holes' in the array

![](heap_array.png)

- Given the node with index \( i \)
  - Parent’s index is \((i-1)/2\)
  - Left child’s index is \((2*i+1)\)
  - Right child’s index is \((2*i+2)\)

- One needs to keep track of the current size of the array
- If current size exceeds max\(Size\), one needs to copy the data into a bigger array (for example \(x2\) the size)
Heap: Comments

- CBT for HeapArray[27,23,17,14,19,6,10,7,3]

- A heap is weakly ordered compared to a BST
- Paths are independent of each others
- Searching or traversing the nodes is not practical
- At first, the ordering of nodes seems pretty awkward, but it is just sufficient to allow for fast removal of the maximum node and fast insertion of new nodes
Heap: Removal

- Removal means here removing the node with the maximum key (node with maximum priority)
- This node is always the root
- The problem is that if the root is removed, there is an empty cell and we do not have a CBT anymore. The hole must be filled in.
- 3 steps to follow:
  - 1- remove the root:
  - 2- move the last node into the root:
  - 3- trickle the last node down until it is below a larger node and above a smaller one

```java
public Node remove() {
    Node root = heapArray[0];
    heapArray[0] = heapArray[--N];
    trickleDown(0);
    return root;
}
```
Heap: Removal - Example 1

Step 1 - Delete root 35 from heap

Step 2 - Last node 19 in root

Step 3 - Trickle down 19 (successive swaps with largest child)
Heap: Removal - Example 2

How many swaps needed?
At most #levels-1 (height-1)

For N nodes, the height is \( \log_2(N+1) \)

So cost for deletion is \( O(\log N) \)
A traditional swap requires 3 copies
Example: 3 swaps usually mean 9 copies

We can optimize the procedure of trickle algorithm by substituting swaps by copies
Example: 3 successive 'swaps' can be done using 5 copies

At the limit of large number of levels, the saving comes close to a factor 3
Heap: Insertion

- A bit easier than removal (less comparisons)
- Initially the node to be inserted is placed in the first open position of the CBT
- Insertion uses then trickle up, the node moves up until it is below a node with larger key

```java
public boolean insert(int key) {
    if (N==maxSize) return false;
    Node newNode=new Node(key);
    heapArray[N] = newNode;
    trickleUp(N++);
    return true;
}
```
Add new node 35 in this heap

Trickle up 35 using successive swaps with parent until heap property is satisfied
Cost for insertion is $O(\log N)$

**Remark:**
For the same set of nodes, one can generate many valid Heaps depending on the order of insertion (it is not unique).

**To do:** Test Java applet Heap.html
private void trickleUp(int i) {
    // reach top of heap - done
    if (i==0) return;
    // check if parent is smaller
    int parent=(i-1)/2;
    if (heapArray[i]>heapArray[parent]) {
        swap(i,parent); //swap
        trickleUp(parent); //recursion
    }
}

private void trickleDown(int i) {
    int leftChild=2*i+1;
    int rightChild=leftChild+1;
    // i is a leaf node - done
    if (leftChild>=n) return;
    // i has only a left child
    if (rightChild>=n) {
        if (heapArray[i]<heapArray[leftChild]) {
            swap(i,leftChild);
        }
        return;
    }
    // n has two children
    if (heapArray[i]<heapArray[leftChild] ||
        heapArray[i]<heapArray[rightChild]) {
        // need to continue trickling down
        if (heapArray[leftChild]> heapArray[rightChild]) {
            // swap with left child
            swap(i,leftChild);
            trickleDown(leftChild); //recursion
        }
        else{
            // swap with right child
            swap(i,rightChild(i));
            trickleDown(rightChild); //recursion
        }
    }
}