HEAP

Lesson-2

Print all nodes less than a value x in a Min Heap

Approach

The idea is to do a preorder traversal of the given Binary heap. While doing preorder traversal, if the value of a node is greater than the given value x, we return to the previous recursive call. Because all children nodes in a min heap are greater than the parent node. Otherwise we print current nodes and recur for its children.

```
        void printSmallerThan(int x, int pos)
1.      {
2.          /* Make sure item exists */
3.          if (pos >= heap_size)
4.              return;
5.
6.          if (harr[pos] >= x) {
7.              /* Skip this node and its descendants,
8.               as they are all >= x . */
9.              return;
10.         }
11.
12.         System.out.print(harr[pos] + " ");
13.
14.         printSmallerThan(x, left(pos));
15.         printSmallerThan(x, right(pos));
16.     }
```

K largest(or smallest) elements in an array | added Min Heap method

Approach

1) Build a Min Heap MH of the first k elements (arr[0] to arr[k-1]) of the given array. O(k)

2) For each element, after the kth element (arr[k] to arr[n-1]), compare it with the root of MH.
   (a) If the element is greater than the root then make it root and call heapify for MH
   (b) Else ignore it.

// The step 2 is O((n-k)*logk)

3) Finally, MH has k largest elements and root of MH is the kth largest element.

```
1.     void FirstKelements(int arr[],int size,int k){
2.        // Creating Min Heap for given
3.        // array with only k elements
4.        MinHeap* m = new MinHeap(k, arr);
5.
6.        // Loop For each element in array
7.        // after the kth element
8.        for (int i = k; i < size; i++) {
9.
10.           // if current element is smaller
11.           // than minimum element, do nothing
12.           // and continue to next element
13.           if (arr[0] > arr[i])
14.               continue;
15.
16.           // Otherwise Change minimum element to
17.           // current element, and call heapify to
18.           // restore the heap property
19.           else {
20.               arr[0] = arr[i];
21.               m->heapify(0);
22.           }
23.       }
24.       // Now min heap contains k maximum
25.       // elements, Iterate and print
26.       for (int i = 0; i < k; i++) {
27.           cout << arr[i] << " ";
28.       }
```

Time Complexity:

O(k + (n-k)Logk) without sorted output. If sorted output is needed then O(k + (n-k)Logk + kLogk)

Sort a nearly sorted (or K sorted) array

Approach

1) Create a Min Heap of size k+1 with first k+1 elements. This will take O(k) time
2) One by one remove min element from heap, put it in result array, and add a new element to heap from remaining elements.

```
int sortK(int arr[], int n, int k)
{
    // Insert first k+1 items in a priority queue (or min
    // heap)
    //(A O(k) operation). We assume, k < n.
    priority_queue<int, vector<int>, greater<int> > pq(arr, arr + k + 1);

    // i is index for remaining elements in arr[] and index
    // is target index of for current minimum element in
    // Min Heap 'pq'.
    int index = 0;
    for (int i = k + 1; i < n; i++) {
        arr[index++] = pq.top();
        pq.pop();
        pq.push(arr[i]);
    }

    while (pq.empty() == false) {
        arr[index++] = pq.top();
        pq.pop();
    }
}
```

Time Complexity:

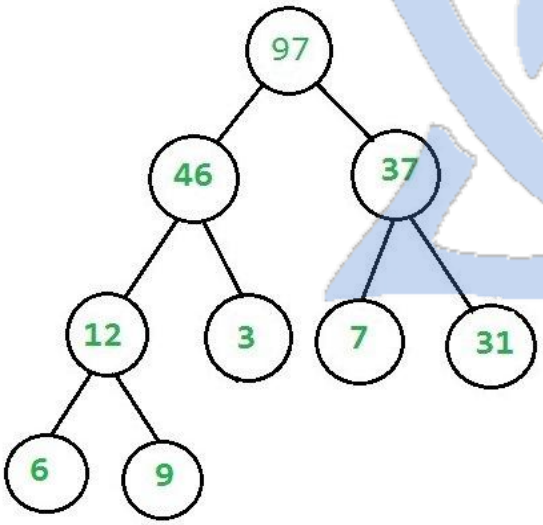
Removing an element and adding a new element to min heap will take log k time. So overall complexity will be O(k) + O((n-k) * log(k)).

Check if a given Binary Tree is Heap

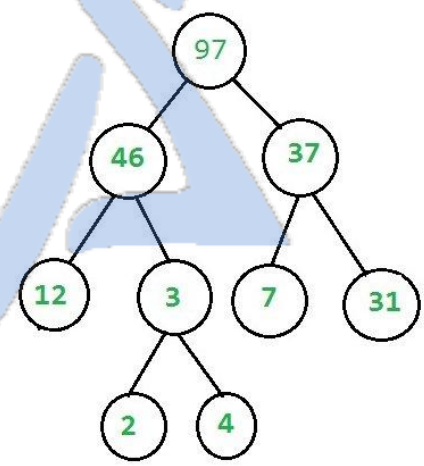
Conditions for an Heap:

1. It should be a complete tree (i.e. all levels except last should be full).
2. Every node's value should be greater than or equal to its child node (considering max-heap).

For example this tree contains heap property –



While this doesn't



Approach

1. Every Node can have 2 children, 0 child (last level nodes) or 1 child (there can be at most one such node).
2. If Node has No child then it's a leaf node and returns true (Base case)
3. If Node has one child (it must be left child because it is a complete tree) then we need to compare this node with its single child only.

4. If the Node has both child then check heap property at Node at recur for both subtrees .

```
1.  unsigned int countNodes(struct Node* root)
2.  {
3.     if (root == NULL)
4.         return (0);
5.     return (1 + countNodes(root->left)
6.             + countNodes(root->right));
7.  }
8.
9.  /* This function checks if the
10. binary tree is complete or not */
11. bool isCompleteUtil (struct Node* root,
12.                      unsigned int index,
13.                      unsigned int number_nodes)
14. {
15.    // An empty tree is complete
16.    if (root == NULL)
17.        return (true);
18.
19.    // If index assigned to
20.    // current node is more than
21.    // number of nodes in tree,
22.    // then tree is not complete
23.    if (index >= number_nodes)
24.        return (false);
25.
26.    // Recur for left and right subtrees
27.    return (isCompleteUtil(root->left, 2*index + 1,
28.                           number_nodes) &&
29.            isCompleteUtil(root->right, 2*index + 2,
30.                           number_nodes));
31. }
32.
33. // This Function checks the
34. // heap property in the tree.
35. bool isHeapUtil(struct Node* root)
36. {
37.    //  Base case : single
38.    // node satisfies property
39.    if (root->left == NULL && root->right == NULL)
40.        return (true);
41.
42.    //  node will be in
```

```
    // second last level
    if (root->right == NULL)
    {
        //  check heap property at Node
        //  No recursive call ,
        // because no need to check last level
        return (root->key >= root->left->key);
    }
    else
    {
        //  Check heap property at Node and
        //  Recursive check heap
        // property at left and right subtree
        if (root->key >= root->left->key &&
            root->key >= root->right->key)
            return ((isHeapUtil(root->left)) &&
                    (isHeapUtil(root->right)));
        else
            return (false);
    }
}
//  Function to check binary
// tree is a Heap or Not.
bool isHeap(struct Node* root)
{
    // These two are used
    // in isCompleteUtil()
    unsigned int node_count = countNodes(root);
    unsigned int index = 0;
    if (isCompleteUtil(root, index,
                       node_count)
        && isHeapUtil(root))
        return true;
    return false;
}
```

How to check if a given array represents a Binary Heap?

Approach:

A **Simple Solution** is to first check root if it's greater than all of its descendants. Then check for children of the root. Time complexity of this solution is $O(n^2)$

An **Efficient Solution** is to compare root only with its children (not all descendants), if root is greater than its children and the same is true for all nodes, then tree is max-heap (This conclusion is based on transitive property of > operator, i.e., if x > y and y > z, then x > z). The last internal node is present at index (n-2)/2 assuming that indexing begins with 0.

```
1.   bool isHeap(int arr[], int i, int n)
2.   {
3.       // If a leaf node
4.       if (i >= (n - 2) / 2)
5.           return true;
6.
7.       // If an internal node and is
8.       // greater than its children,
9.       // and same is recursively
10.      // true for the children
11.      if (arr[i] >= arr[2 * i + 1] &&
12.          arr[i] >= arr[2 * i + 2]
13.          && isHeap(arr, 2 * i + 1, n)
14.          && isHeap(arr, 2 * i + 2, n))
15.          return true;
16.
17.      return false;
18.  }
```

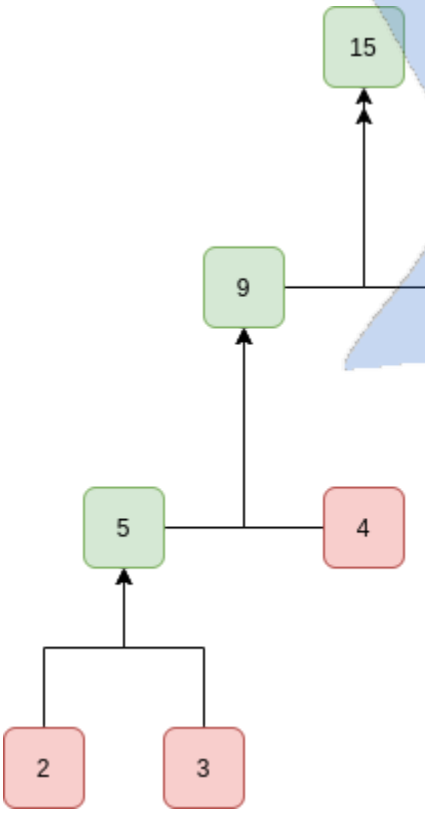
Time Complexity

Time complexity of this solution is O(n).

Connect n ropes with minimum cost

Approach

1. Create a min-heap and insert all lengths into the min-heap.
2. Do the following while the number of elements in min-heap is not one.
1. Extract the minimum and second minimum from min-heap
2. Add the above two extracted values and insert the added value to the min-heap.
3. Maintain a variable for total cost and keep incrementing it by the sum of extracted values.
3. Return the value of this total cost.



```
int minCost(int len[], int n)
{
    int cost = 0; // Initialize result

    // Create a min heap of capacity
    // equal to n and put all ropes in it
    struct MinHeap* minHeap = createAndBuildMinHeap(len, n);

    // Iterate while size of heap doesn't become 1
    while (!isSizeOne(minHeap)) {
        // Extract two minimum length
        // ropes from min heap
        int min = extractMin(minHeap);
        int sec_min = extractMin(minHeap);

        cost += (min + sec_min); // Update total cost

        // Insert a new rope in min heap
        // with length equal to sum
        // of two extracted minimum lengths
        insertMinHeap(minHeap, min + sec_min);
    }

    // Finally return total minimum
    // cost for connecting all ropes
    return cost;
}
```

Complexity Analysis:

- **Time Complexity:** O(nLogn), assuming that we use a O(nLogn) sorting algorithm. Note that heap operations like insert and extract take O(Logn) time.
- **Auxiliary Complexity:**O(n), The space required to store the values in min heap

Merge k sorted arrays

Approach:

1. Create a min Heap and insert the first element of all k arrays.
2. Run a loop until the size of MinHeap is greater than zero.
3. Remove the top element of the MinHeap and print the element.
4. Now insert the next element from the same array in which the removed element belonged.
5. If the array doesn't have any more elements, then replace root with infinite.After replacing the root, heapify the tree.

```
1.    nt *mergeKArrays(int arr[][n], int k)
2.    {
3.
4.    // To store output array
5.       int *output = new int[n*k];
6.
7.       // Create a min heap with k heap nodes.
8.       // Every heap node has first element of an array
9.       MinHeapNode *harr = new MinHeapNode[k];
10.      for (int i = 0; i < k; i++)
11.      {
12.
13.   // Store the first element
14.          harr[i].element = arr[i][0];
15.
16.   // index of array
17.          harr[i].i = i;
18.
19.   // Index of next element to be stored from the array
20.          harr[i].j = 1;
21.      }
22.
23.   // Create the heap
24.      MinHeap hp(harr, k);
```

```
25.
26.      // Now one by one get the minimum element from min
27.      // heap and replace it with next element of its array
28.      for (int count = 0; count < n*k; count++)
29.      {
30.          // Get the minimum element and store it in output
31.          MinHeapNode root = hp.getMin();
32.          output[count] = root.element;
33.
34.          // Find the next elelement that will replace current
35.          // root of heap. The next element belongs to same
36.          // array as the current root.
37.          if (root.j < n)
38.          {
39.              root.element = arr[root.i][root.j];
40.              root.j += 1;
41.          }
42.          // If root was the last element of its array
43.   // INT_MAX is for infinite
44.   else root.element =  INT_MAX;
45.
46.          // Replace root with next element of array
47.          hp.replaceMin(root);
48.      }
49.
50.      return output;
51.   }
```

Complexity Analysis:

- **Time Complexity :**O(n * k * log k), Insertion and deletion in a Min Heap requires log k time. So the Overall time complexity is O(n * k * log k)
- **Space Complexity :**O(k), If Output is not stored then the only space required is the Min-Heap of k elements. So space Complexity is O(k).