# CS1102: Data Structure and Algorithms

**Tutorial 9 Heaps (Solution)**
Week of 29 March 2010

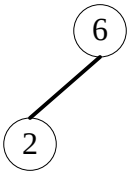
**Question 1: Heap Operations**

a. Create a max-heap by inserting the following integers into an initially empty heap (one by one):
   6, 2, 4, 1, 8, 5, 3, 7, 9
   Draw some diagrams to show the evolution of this heap.

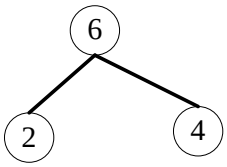
**Answer:**

Insert 6

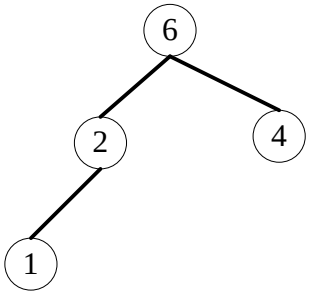

Insert 2



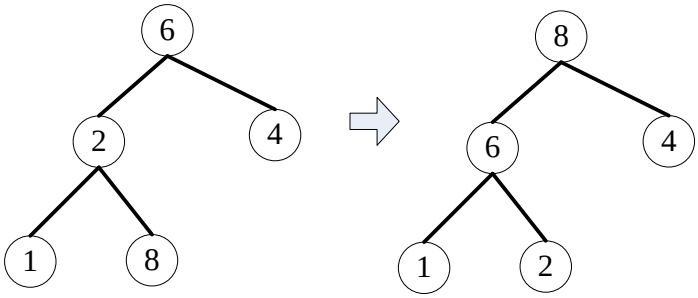

Insert 4



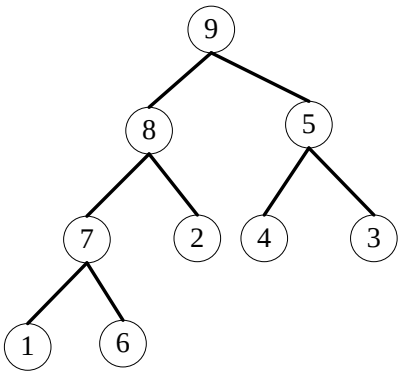

Insert 1




Insert 8:



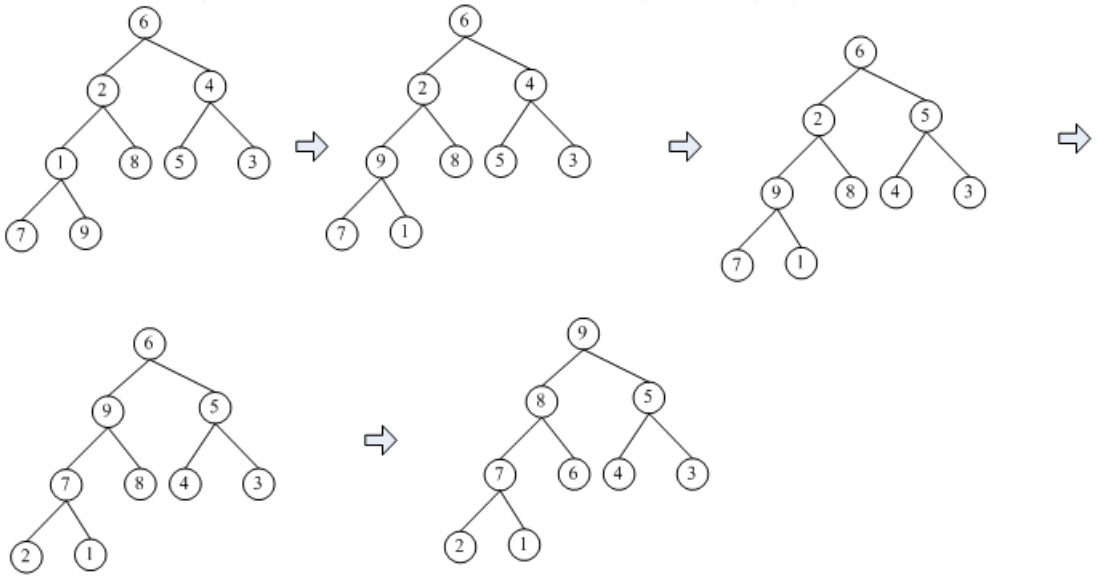
… Finally:




b. Next, assume array {6, 2, 4, 1, 8, 5, 3, 7, 9} represents a complete binary tree, draw the tree and some other diagrams to show the "heapification" of this tree into a max-heap.

**Answer:**

Bubble down, from the last internal node 1, then 4, 3, 6.



c. What are the differences between the above two approaches in constructing a heap from a given set key values?

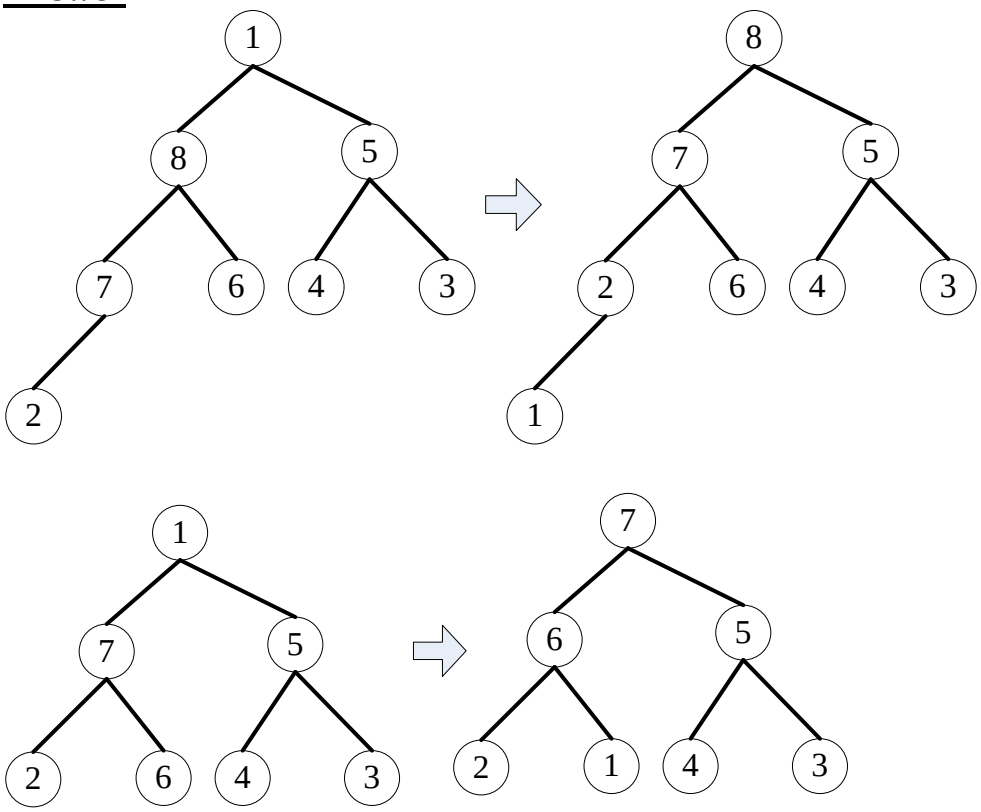
**Answer:**

1) Given the same set of integers, the two algorithms can result in different max-heap.
2) As a note, the time complexity of "heapification" is O(n) whereas "sequential insertions" is O(n log n).

d. Next, perform two deleteMax() operations from the heap obtained in 1.b. Show the final results.

Note that performing deleteMax() n times is simply the heap sort algorithm.

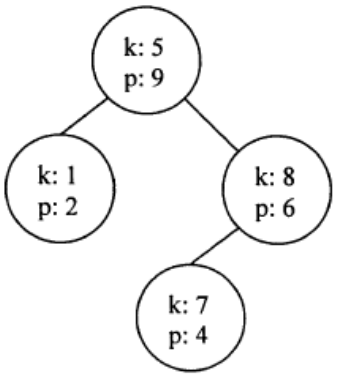
**Answer**



# CS1102: Data Structure and Algorithms

**Question 2: Treap**

A **treap** is a **binary tree** where each node has a **key** and a **priority**. The **keys** follow a **binary search tree property** and the **priorities** follow the **heap property**. See the example below




a. Is a treap a heap?

**Answer:**
No! Because the other requirement for a heap is that it is a complete binary tree!

b. Is a treap a binary search tree?

**Answer:**
Yes! Treap is actually a BST but with an added constraint of maintaining the heap priorities.

c. You are given a set of n pairs (k = keys, p = priority) and you are asked to find a corresponding treap. For instance, if n = 4 and the pairs are (5, 9), (7, 4), (8, 6), (1, 2) then a correct answer is given in the picture above. How many such treaps are possible in general? Justify your answer briefly. Assume that there is no duplicate key or priority. Namely, you won't see any input containing (1,2) and (1,3); neither will you see any input containing (2,1) and (3,1).

**Answer:**
There can only be a unique treap. The reasons are as follows:

1. Given a set of n pairs, the pair for the root is unique because the root should have the pair with the highest priority based on heap property.
2. The pair of the root divides the remaining pairs into two distinct sets: the set with keys smaller than root.key and should be the left subtree of the root; and the set with keys greater than root.key and should be the right subtree.

3. Both the left and right subtrees are treaps. Following the reasoning in 1, the roots of them are unique too.

   By recursively applying the reasoning for each of the subtrees, we can see that the treap is indeed unique.

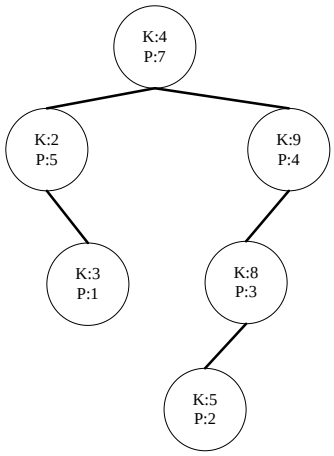
d. Design an algorithm (no programming required) to transform a treap from an array representation to a tree structure. You can use the following treap as an instance:
   The treap contains 6 nodes with labels (k = 2, p = 5), (k = 5, p = 2), (k = 3, p = 1), (k = 4, p = 7), (k = 9, p = 4), (k = 8, p = 3).

**Answer:**

One of the easiest way to do this is to first sort the 6 nodes based on decreasing priority, which will give us: (k = 4, p = 7), (k = 2, p = 5), (k = 9, p = 4), (k = 8, p = 3), (k = 5, p = 2), (k = 3, p = 1).

Then, simply insert the nodes one by one from (k = 4, p = 7) to (k = 3, p = 1) as in normal BST. Both BST and Heap property will be maintained if we do the insertion like this: BST property is maintained because insertions are performed exactly like a normal BST; and Heap property is maintained because, given any two nodes n and m such that n is an ancestor of m, n must be inserted before m. As a result, n.priority is greater than m.priority and Heap property is correctly maintained.

This is O(n log n) for sorting plus O(n) * O(h) BST insertion. There will be only one unique treap, drawn as follow:

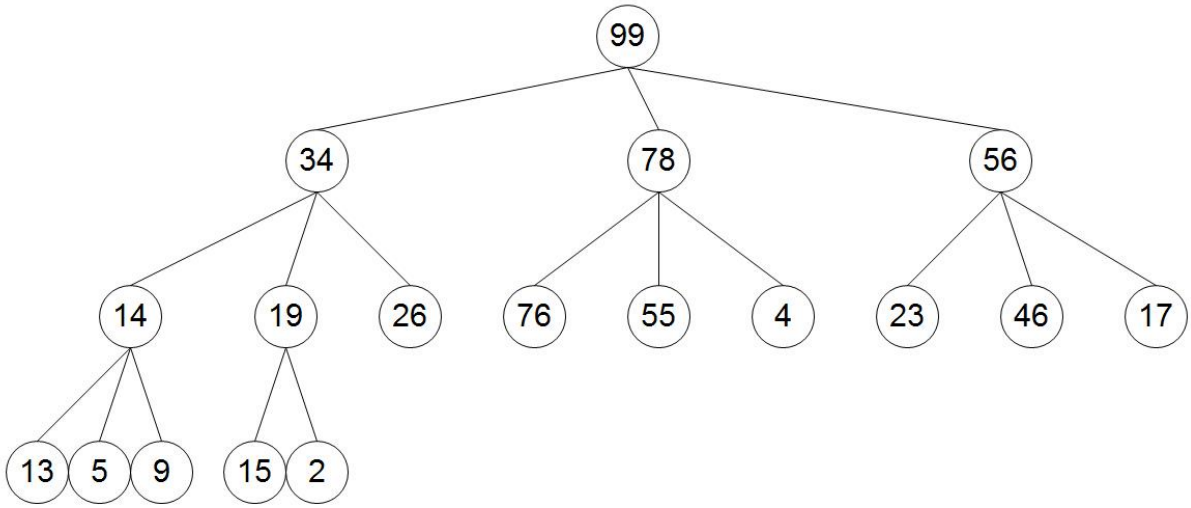



A better answer is to use rotations during insertions to get a height balanced binary tree. However, this is beyond the scope of current CS1102 syllabus.

# CS1102: Data Structure and Algorithms

**Question 3: d-ary Heap**

A d-ary heap is like a binary heap, with all but the last non-leaf node having d children instead of only 2 children. A ternary (d=3) max-heap is shown below to give you an idea.




a. How would you represent a d-ary heap in an array? Given a child node with index childId, how to locate the index of its parent? Given a parent node, how to locate the indices of its children?
b. What is the height of a d-ary heap of n elements in terms of n and d?
c. Give an implementation of a d-ary max-heap class. We mainly focus on the method add() and removeMax(). Analyze the running time of these two methods in terms of n and d.
d. Analyze the running time of heap sort using your d-ary heap in terms of n and d.

**Answer:**

**a.**
Recall that in binary heap case, we have root at index 0 and parentId = (childId-1) / 2. It can be straightforwardly generalized as

parentId = (childId-1) / d.

On the other hand, for any parent located at parentId, the indexes of it children from left to right are:

parentId*d+1, parentId*d+2, …, parentId*d+d

But for the sake of completeness, a proof is shown below. You may omit the proof.

- The relationship between parentId and childId in a full d-ary heap?
  - For each rightmost child in each level: Suppose root is the 1st level, recall that it is located at index 0; so the rightmost node in the 2nd level is located at index d,…, the rightmost node in the $i^{th}$ level is located at $d+d^2+\ldots+d^{i-1}$.
    Thus, the rightmost node in each $i^{th}$ level has its parent located at index $d+d^2+\ldots+d^{i-2}$; notice that $(d+d^2+\ldots+d^{i-1}-1)/d=d+d^2+\ldots+d^{i-2}$ (integer division). Thus, the equation holds.
  - The equation also holds for all siblings of the rightmost child in each $i^{th}$ level, because all these siblings locate at indices $j+d^2+\ldots+d^{i-1}$, j=1,2,…,d, and when deduct by 1 and then divided by d, they all return the same value, which comes along the fact that they share the same parent. Since we just showed the rightmost child gets its parent correctly, the equation holds for all its siblings. In other words, the equation holds for all children of the rightmost parent in any level.
  - Suppose the equation holds for all d children of a parent, so we know its children locate at indices parentId*d+j, j=1,2,…,d. Let's consider the parent located at index (parentId-1): we know its children locate at indices parentId*d+j-d=(parentId-1)*d+j,j=1,2,…,d. Thus the equation holds for this new parent.
  - Apply mathematical induction using the two facts above. For each level, first we can prove the equation holds for all children of the rightmost parent, then those of the second-to-the-rightmost one, …, until those of the leftmost parent.

- We have the equations for full d-ary heaps. This obviously implies the equation holds for general d-ary heaps. To conclude, root locates at index 0; given a child at index childId, parentId = (childId-1)/d where parentId is the index its parent. On the other hand, for any parent located at parentId, parentId*d+1, parentId*d+2, …, parentId*d+d, identify all its children.

**b.**

If n=0 then 0. If n>0:
If we have 1 node only, then we have 1-level d-ary heap.
If we have d more nodes, then we fill up another level.
If we have $d^2$ more nodes, then we fill up another level.
…
To conclude, with $1+d+d^2+\ldots+d^{i-1}$ nodes, we can have a full i-level d-ary heap.
Notice that $1+d+d^2+\ldots+d^{i-1} = (d^i-1)/(d-1) = n$, which means

$i=\log_d(n*(d-1)+1)=O(\log_d n) = O(\log n)$, since d is a constant.

For general case where the heap is not necessarily full, it still holds.

**c.**

The implementation is straightforward given the binary heap implementation. The underlying mechanism is the same; we still perform the same heap operations, bubbling up for insertion and bubbling down for deletion, only a couple of places need to be taken care of: In order to perform heapRebuild() correctly, one needs to find out the maximum among a parent node and its d, instead of just 2 children. This is the place we need to completely rewrite. Other than that, only several places need modifying by applying our new equations in 3.a. Here we only show the implementation of heapInsert() and heapDelete(), together with heapRebuild() of course (notice that //~ denotes where we apply the new equations):

```
public class Heap{
    private int d;
    …
    public void heapInsert (int newItem) throws HeapException {
        if (size < MAX_HEAP) {
                items[size] = newItem; // append the new item to the array
                int place = size; // place is the index of the new item
                int parent = (place - 1)/d;    //~
                while ( (parent >= 0) && (items[place] > items[parent]) {
                        // heap property violated, need to bubble up
                        int temp = items[place];
                        items[place] = items[parent];
                        items[parent] = temp;
                        place = parent;
                        parent = (place - 1)/d; //~
                }
                ++size;
        }
        else
                throw new HeapException("HeapException: Heap full");
    }


    public int heapDelete() {
        int rootItem = 0;
        if (!heapIsEmpty()) {
                rootItem = items[0]; // rootItem is set to the max key value
                items[0] = items[--size]; //replace the root by the last item
                heapRebuild (0); // to rebuild the help – bubble down
        }
```

```
        return rootItem; // return the maximum key value
    }

    protected void heapRebuild (int root) {
        int largestChild = d * root + 1; //~ initialize the largestChild
        if (largestChild < size) {// it exists
                for (int i = largestChild + 1; i < largestChild + d; i++)
                        if ( (i < size) && items[i] > items[largestChild] )
                                largestChild = i; // identify child with the largest key
                if ( items[root] < items[largestChild] ) {
                        // bubble down - swap root with the largest child
                        int temp = items[root];
                        items[root] = items[largestChild];
                        items[largestChild] = temp;
                        heapRebuild(largestChild); // recursive call
                }
        }
    }…
}
```

Analysis of heapInsert ():
Now we have an additional argument d. Note that the number of swaps we need is no more than the height of the heap. Thus, the time complexity is $O(\log_d n) = O(\log n)$ since d is constant.

Analysis of heapDelete ():
Note that the number of swaps we need is once again no more than the height of the heap. But every time before we swap, we need O(d) time to find the maximum among d children, while in heapInsert(), each time we only compare one child to its parent. Thus, the time complexity for both heapDelete () and heapRebuild () is $O(d*\log_d n)=O(\log n)$, since d is a constant.

**d.**
We adopt exactly the same strategy as in 1.b. to do sorting. Thus we have 2 phases.
First we construct the d-ary heap using a strategy similar to that in 1.b. But keep in mind that each time before we swap, we need O(d) time to find the maximum among d children. Thus, we can obtain the d-ary heap in time $O(d*n)=O(n)$, since d is a contant.
Then we apply removeMax() n times, introducing $O(d*n* \log_d n)$ time complexity. Therefore, the answer is $O(d* n *\log_d n) = O(n*\log n)$, since d is a constant.