Solutions for Homework 5

Problem 1 Consider a generalization of the binary heap structure. Every node has d children. It is an almost complete,d-ary tre, and a node must be less than or equal to all its children. Design an array representation of the heap. Design a Deletemin and Increasekey procedure here.

Solution: We generalize the representation of a 2-ary (binary) heap to a d-ary heap. Root is stored in array element 0. The children of root are stored in array elements 1 to d. The grandchildren of root are stored in array elements $d+1$ to d^2+d. In general, level j has d^j elements which are stored in locations $1+\sum_{i=1}^{i=j-1} d^i$ to $\sum_{i=0}^{i=j} d^i$. Parent of j is at position $\lfloor \frac{j-1}{d} \rfloor$, and children of j are at $dj+1$ to $dj+d$.

Deletemin removes the value in the root (which is the minimum value) and places the last element of the heap — name it a — to the side. Then the procedure finds the child of the root with the minimum value and compares this value to the value of a. If a is smaller then it is placed in the root, and the heap has been restored. Otherwise, the minimum child value moves up to the root, and the algorithm is applied recursively using this child as the new root.

Increasekey increases the value of the specified element. Then it compares its new value to the values of its children. If its value is greater than any of its children, then the minimum child value is interchanged with the value of the specified element. The algorithmn is applied recursively till it reaches a leaf.

Problem 2: Consider a binary heap. Print the keys as encountered in a preorder travel. Is the output sorted? Justify your answer Attempt the same question for inorder and postorder travel.

Solution: Consider first the preorder traversal. A counter-example is provided: The root of the heap keeps key-value 1. Its left child keeps key-value 4 and its right child keeps key-value 2. A preorder traversal would have the following output: $1, 4, 2$. Obviously the output is not sorted.

In the case of inorder, whatever heap we provide the output will not be sorted. The reason is that the heap property makes the left child being bigger than its parent. For example the above heap would give $4, 1, 2$ as output.

The same argument holds for postorder traversal. Checking this with the above heap, we get output $4, 2, 1$ which is not sorted.

Problem 3: Give an algorithm to find all nodes less than some value X in a binary heap. Analyze its complexity.

Solution: Here is a sketch of a recursive algorithm: start from the root of the heap. If the value of the root is smaller than X then print this value and call the procedure recursively once for its left child and once for its right child . If the value of a node is bigger or equal than X then the procedure stops without printing that value.

The complexity of this algorithm is $O(N)$, where N is the total number of nodes in the heap. This bound takes place in the worst case, where the value of every node in the heap will be smaller than X, so the procedure has to call each node of the heap.

Problem 4: Given any n, design an input of n elements such that the insertion sort takes $\Omega(n^2)$ operations.

Solution: Consider an input that is initially sorted in the reverse order. For example, the sequence $5, 4, 3, 2, 1$ satisfies the above property for $n = 5$.

In this case, let the first k elements be sorted using insertion sort. Then the $k + 1$th element will take k comparisons to be inserted properly in the first position, since it is smaller than the first k elements. Thus, the total number of comparisons is $1 + 2 + \ldots + (n - 1) = \frac{n(n-1)}{2}$ which is $\Omega(n^2)$.

Problem 5: Problem 6.15 a, b, c from Weiss.

Solution:

A. The minimum key in a min-max heap is found at the root. The maximum key is the largest child of the root.

B. A node is inserted by placing it into the first available leaf position and reestablishing the min-max heap property from the path to the root. Here is the procedure reestablishing the property:

```
/* A is the data array */

procedure PercolateUp(i) /* i is the position in the array */
{
```

```
  if (i is on min-level)

      if A[i]>A[parent(i)

    swap A[i] and A[parent(i)]
    PercolateUpMax(parent(i)) /* distinguish bw max and min levels */

      else PercolateUpMin(i)
  else /* i is on max level */

     if A[i]<A[parent(i)

    swap A[i] and A[parent(i)]
        PercolateUpMin(parent(i))

      else PercolateUpMax(i)
}

procedure PercolateUpMin(i) {
if A[i] has grandparent
 if A[i]<A[grandparent(i)]
   swap A[i] and A[parent(i)]
   PercolateUpMin(grandparent(i))
}

PercolateUpMax is similar to PercolateUpMin. The relational operators <,> are
reversed accordingly.
```

C. DeleteMin and DeleteMax are similar to the operations with normal heaps. The desired element (min or max) is extracted and the last element of the heap is inserted into the empty position. Then the algorithm must maintain the max-min heap property. Again the procedure differentiates between min-level and max-level.

```
procedure PercolateDownMin(i) /* i is the position in the array */
{
 if (i is on min level)
```

```
    PercolateDownMin(i)

 else PercolateDownMax(i)
}

procedure PercolateDownMin(i) {
 if A[i] has children
     m:= index of smallest of children
     and grandchildren (if any) of i

     if A[m] is grandchild of i

    if A[m]<A[i]

       swap A[m] and A[i]

       if A[m]>A[parent[m]

          swap A[m] and A[parent(m)]

       PercolateDownMin(m)

     else /* A[m] is a child of i */

    if A[m] < A[i]

        swap A[i] and A[m]
}
```

Again the PercolateDownMax(i) is similar to the one above.