Data Structures and Algorithms — CSCI 230
Chapter 6 — Review Solutions

1. Consider a binary heap, implemented, of course, as an array. Write a function to delete the element stored at location `i` of the heap. Assume that functions `Percolate_Down` and `Percolate_Up` exist, with prototypes

   ```
   void Percolate_Down( ElementType * heap, int size, int i);
   void Percolate_Up( ElementType * heap, int size, int i);
   ```

   where `heap` is the array of elements, and `size` is the current number of elements. Assume that `i` is correctly given to you, i.e., assume `1 <= i <= size`.

   Start from the following prototype

   ```
   void Delete( ElementType * heap, int & size, int i);
   ```

   Solution:
   The trick here is that a percolate up operation may be required in some instances! Otherwise, it is very similar to the delete min operation.

   ```
   {
      heap[i] = heap[size--];
      if ( i == 1 || heap[i] > heap[i/2] )
         Percolate_Down( heap, size, i );
      else
         Percolate_Up( heap, size, i );
   }
   ```

2. Consider a priority queue implemented as a heap. The heap is implemented using a vector, and it contains n values stored in subscript locations 1 through n. Assume it is a "max heap", so that the largest value is stored in subscript location 1.

   (a) What are the possible subscript locations for the third largest value in the heap (assuming all values are distinct)?

   Solution:
   2 through 7. It could be one of the two children of the root or it could be one of the grandchildren of the root.

   (b) What is the range of possible subscript locations for the smallest value in the heap (again assuming all values are distinct)?

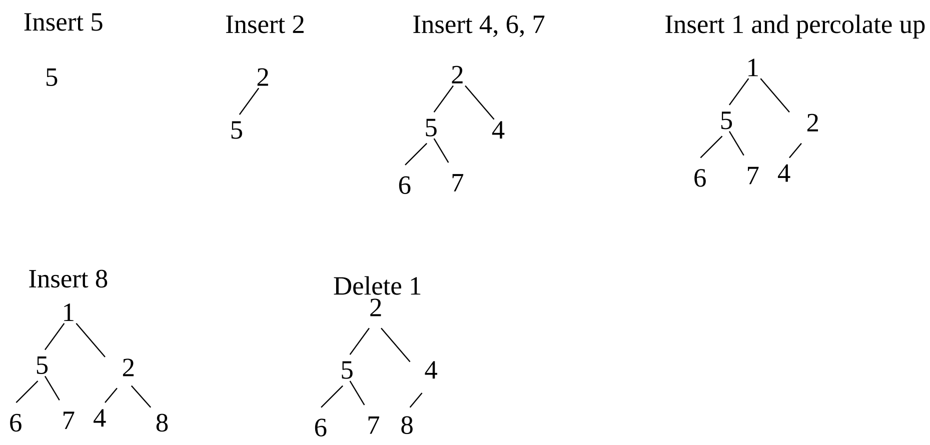
   Solution:
   $\lfloor n/2 \rfloor + 1$ through n. This is any location that has no children.

(c) What is the worst-case time complexity required to find the minimum value in the heap?

Solution:
$\theta(n)$, because each of the $n - \lfloor n/2 \rfloor$ locations identified in the previous problem must be searched.

3. Given an empty binary heap of integers, show the structure of the binary heap after the values 5, 2, 4, 6, 7, 1, 8 are inserted into it. (Note, you are not being asked to simulate the `Build_Heap` function.) Then show the heap after the minimum value is removed. You may show the heap as a tree rather than as an array. The heap is ordered so that the minimum value is at the root.

Solution:



4. Suppose you are given a pointer, `Root`, to the root of a leftist heap of floating point values. Suppose you are also given a pointer, `P`, to a particular node in the leftist heap. Assume each `Leftist` node has the structure

```
struct Leftist {
  float Value;
  Leftist *Parent, *Left, *Right;
  int Npl;   // the null path length
};
```

You may assume the existence of a merge function with the following prototype:

```
Leftist* Merge( Leftist* t1_root, Leftist* t2_root )
```

which merges two leftist heaps, updates the null path lengths as necessary, and returns a pointer to the root of the resulting leftist tree.

(a) Write a function to remove the node pointed to by `P`. The function should be as efficient as possible. Here is the prototype:

```
void LeftistRemove( Leftist* & Root, Leftist* & P )
```

Solution:
This was a challenging problem, and was intended as such! There are two parts to the solution. First, is removing `P` from the tree, which involved both fixing pointers and a merge operation. The second, it isn't nearly enough to merge the left and right subtrees and merge the result with the root.

Anyway, here is the code.

```
void LeftistRemove( Leftist* & Root, Leftist* & P )
{
    //  Do the merge and fix pointers.
    Leftist * q = Merge( P->Left, R->Right );
    if ( P == Root ) {
        Root = q;
        return;
    }
    if ( P->Parent->Left == P )
        P->Parent->Left = q;
    else
        P->Parent->Right = q;
    q->Parent = P->Parent;
    delete P;
```

```
    //  Go up the tree only until the NPL doesn't change!
    while ( q->Parent != NULL ) {
        Leftist * parent = q->Parent;
        if ( (parent->Right == q && parent->Npl == q->Npl+1) ||
             (parent->Left == q && q->Npl >= parent->Right->Npl) )
            break;
        else {
            parent->Npl = q->Npl+1;
            if (q->Npl < parent->Right->Npl) swap( q, parent->Right );
            q = parent;
        }
    }
}
```

(b) What is the worst-case time required by the function? Assume N is the number of nodes in the entire tree. Explain briefly, but carefully.

Solution:
This one is subtle. Certainly, the merge is $O(\log N)$. Walking back up the tree is also $O(\log N)$, despite the fact that the longest path back to the route is $O(N)$. The reason is that propagation back up to the route can only continue up the right path or when the right path and left path are balanced. Since the right path from the route is $O(\log N)$, the result follows. Combining the times additively yields $O(\log N)$ overall.