1. Consider the following binary search tree.

\[
\begin{array}{c}
\text{24} \\
\text{8} \\
\text{4} & \text{18} & \text{28} \\
\text{12} & \text{21} \\
\end{array}
\]

Assuming it is an AVL tree, show the state of the tree after inserting 10 and then again after deleting 28.

**SOLUTION:**

AVL Tree
After inserting 10

\[
\begin{array}{c}
\text{24} \\
\text{12} \\
\text{8} & \text{18} & \text{28} \\
\text{4} & \text{21} \\
\end{array}
\]

After deleting 28

\[
\begin{array}{c}
\text{12} \\
\text{8} & \text{24} \\
\text{4} & \text{18} & \text{35} \\
\text{10} & \text{21} \\
\end{array}
\]

Splay tree:
After inserting 10

After deleting 28

Do the same assuming it is a splay tree.

2. Given an empty AVL tree of integers, show the structure of the tree after each of the values 5, 2, 4, 6, 7, 1, 8 is inserted and then show the change to the resulting tree when 4 is deleted. Indicate where rotations are done to rebalance the tree.

**SOLUTION:**

After inserting 5, 2, 4

Double rotate at 5

Insert 6, 7

Single rotate at 5

3. Given a binary search tree containing \( n \) floating point values and having a height that is \( O(\log n) \) (i.e., it is balanced), write an algorithm to count the
number of nodes storing values greater than x0 and less than x1. What is the running time of your algorithm? (More credit will be given for an efficient algorithm.)

You may assume the following declaration for tree nodes:

```cpp
class TreeNode {
public:
    float x;
    TreeNode * left;
    TreeNode * right;
}
```

Start from the following prototype and assume the function is initially passed a pointer to the root:

```cpp
int Count( TreeNode * T, float x0, float x1 )
```

**Solution:**

```cpp
{
    if ( T == NULL )
        return 0;
    else if ( T->x <= x0 )
        return Count( T->right, x0, x1);
    else if ( T->x >= x1 )
        return Count( T->left, x0, x1);
    else
        return 1 + Count( T->left, x0, x1 ) +
                Count( T->right, x0, x1 );
}
```

Showing that this is in fact $O(\log n + k)$ is a bit tricky. The basic idea is to count the recursive calls. For each value between x0 and x1 two recursive calls are made. These are calls made at the else statement, resulting in the k part of the $O(\log n + k)$. There are at most $O(\log n)$ recursive calls made following the else if statements. These correspond to walking down the left and right boundaries of the region of the tree containing the values between x0 and x1.

4. 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[ heapsize--];
    if ( i == 1 || heap[i] > heap[i/2] )
        Percolate_Down( heap, i );
    else
        Percolate_Up( heap, i );
}

5. 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 second largest value in the heap (assuming all values are distinct)?

Solution: 2 or 3. It must be one of the two children 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: ⌈n/2⌉ + 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: θ(n), because each of the n − ⌈n/2⌉ locations identified in the previous problem must be searched.

6. 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
class Leftist {
    public:
        float Value;
        Leftist *Parent, *Left, *Right;
        int Np1; // 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.

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->Np1 == q->Np1+1) ||


7. Here is a slightly different version of the Quick Select algorithm, which is much closer to the original version of Quick Sort (as opposed to the Median-of-Three version). At the end of the function (including all recursive calls), the desired value will be in the kth location of the array.

Q_Select_One ( int A[ ], const int k, const int Left, const int Right )
{
    if( Left < Right ) {  
        int Pivot = A[Left];
        unsigned int i = Left, j = Right + 1;

        for( ; ; ) {
            while( A[ ++i ] < Pivot && i<Right );
            while( A[ --j ] > Pivot );
            if( i < j )
                Swap( A[ i ], A[ j ] );
            else
                break;
        }

        if( k < j )        Q_Select_One( A, k, Left, j-1 );
        else if( k > j )   Q_Select_One( A, k, j+1, Right );
    }
}

(a) Assume the original call is made to this function with

Q_Select_One( A, 3, 0, 8)

with the following contents of A

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>24</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>71</td>
<td>26</td>
<td>25</td>
<td>12</td>
</tr>
</tbody>
</table>

Show the contents of A near the end of the code, just before entering the conditional to check if a recursive call should be made. What recursive call, if any, is made?
**Solution:** Here is the array.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>12</td>
<td>14</td>
<td>16</td>
<td>32</td>
<td>71</td>
<td>26</td>
<td>25</td>
<td>24</td>
</tr>
</tbody>
</table>

The recursive call is `Q_Select_One(A, 3, 3, 8)`. Note that this call is made, even though the 16 in position 3 is the correct value.

(b) What is the worst-case number of comparisons in `Q_Select_One` as a function of both \( n \) and \( k \), where \( n \) is the value of `Right-Left+1` in the first call? When does this occur?

**Solution:** The worst-case occurs when \( A \) is already sorted into increasing order. In this case, the first call will make \( n - 1 \) comparisons and recursively consider the interval from 1 to \( n - 1 \). Each of the \( k \) subsequent calls will each involve one fewer comparisons and, except when `Left`\(=k\), will make a recursive call involving an interval that is one location smaller. This gives \( k + 1 \) calls and \( O(n) \) comparisons per call, yielding \( O(nk) \) comparisons. More precisely, the number of comparisons is

\[
\sum_{i=0}^{k} (n - i - 1) = (k + 1) n - \sum_{i=0}^{k} i - (k + 1)
\]

\[
= (k + 1) n - k(k + 1)/2 - (k + 1)
\]

\[
= O(k n) \quad \text{since } k \leq n
\]

8. Solve the Quick Sort recursive function

\[ T(n) = T(k) + T(n - k - 1) + n \]

to yield a non-recursive form when \( k = 1 \). Give an “O” estimate of this form. Assume that \( T(0) = T(1) = 0 \) and \( T(2) = 2 \). You may assume that \( n \) is even.

**Solution:** In this special case,

\[ T(n) = T(1) + T(n - 2) + n = T(n - 2) + n. \]

Hence,

\[ T(n) = T(n - 2) + n \]

\[ = T(n - 4) + (n - 2) + n \]

\[ = T(n - 6) + (n - 4) + (n - 2) + n \]

\[ = \ldots \]

\[ = 2 + 4 + \ldots + (n - 4) + (n - 2) + n \]

\[ = \sum_{i=1}^{n/2} 2i < \sum_{i=1}^{n/2} 2n = (n/2)2n = O(n^2) \]