EC 504
Spring, 2021
HW 1 Software

Due Monday, 2/15/21, 8 pm

1. (30 pts) In this exercise, you are to implement four sorting algorithms to sort arrays of integers and compare them on two sorting inputs. The sorting algorithms to be implemented are:

   - Insertion Sort
   - Merge Sort
   - Merge Sort Blend, where, when in the recursion you find you have 32 or less items to sort, you do insertion sort and don't recur further.
   - A simple version of Timsort, to be described below.

   All of the sorting routines are to be implemented in the header file `sorting.h`, which is enclosed as parrt of the HW. The main file can be used for debugging, but should not be changed.

   The codes will sort two lists of integers. The first list is randomly generated in the main program. The second list is read from file. You cannot change the original sortScaling.cpp file, but you can write your own main program for debugging with smaller list lengths if you wish. Once you have completed this, you can make the executable using `make all` . You can then run `./sort >> sort.out` . Turn in your `sorting.h` file and the `sort.out` file in HW1 folder in your /projectnb/ec504/ directory.

   Now, here is a brief description of simpleTimsort:

```
Set minrunsize to 32. Initialize runstack as array, with runstack[0] = 0, and num_in_stack = 0;
input array a[], with n elements
for i in 0 to n-1,
   find a run of nondecreasing entries in i.
   If run length is less than minrunsize, insert element of a[i] into the run and continue;
   If run length is greater than or equal to minrunsize, increment num_in_stack, store the index of t
   current value of i) into runstack[num_in_stack].
if i gets to n-1, increment num_in_stack and set runstack[num_in_stack] to n.

Now, collapse the stack of runs as follows: While num_in_stack > 1, do
Merge the last two runs (starting at runstack[num_in_stack-2] and runstack[num_in_stack - 1]);
decrement num_in_stack by 1 and set runstack[num_in_stack] = n;
```

   This last step is inefficient, and does not merge in the same clever way Timsort does, maintaining an invariant. This will be easier to implement, and even with the inefficiencies, will show how Timsort works when the data is nearly sorted.

   Extra credit (4 pts): Implement a better simpleTimsort by managing the stack dynamically as follows: Whenever you insert a run into the stack, if its length is greater than or equal to the run below in the stack, merge it right away. Continue this merging recursively into the stack before starting the next run.

   Hence, if we have a stack with runs of length 128, 64, 32 and you insert a run of length 32 on top, you first merge it with the top of 32, yielding a size 64 run. Then, you merge that with the 64, yielding a 128 run, which you merge with the 128 run, leaving a single run of length 256 in the stack. This algorithm will be faster than both versions of mergesort on random data.

2. (30 pts) The purpose of this exercise is to learn to implement top-down splay trees. Top-down splay trees are the preferred way to implement splay trees, for many reasons. First, we don't need to point to parent nodes, so the nodes are smaller. Second, by going top-down, double rotations are treated as single rotations, and many operations are easily implemented.

For instance, consider the Insert operation. Instead of inserting the key into the splay tree and splaying up, you simply splay the key into the tree, and the node that becomes the new root is the node which would be the parent of where the key was to be inserted. It is then easy to insert the new node at the root, and make the root one of its children.

Similarly, for Delete, one splays the key top down, and if the key is in the the tree, it is the new root of the tree, so we can delete it, and be left with a right subtree and a left subtree. To find the largest value in the left subtree, we splay the deleted key into that left subtree, and the new root will be the largest value! This makes it trivial to connect the right subtree to the new root of the left subtree and finish delete.

The Search operation becomes a one-line operation: splay the key to be searched and it is at the root of the returned tree.

In this problem, we provide you with a main function `topdownsplay.cpp`, which contains a Class called SplayTree that implements a top-down splay tree, but is missing three key member functions:

```
Tree_node* Splay(int key, Tree_node* root), Tree_node* Right_Rotate(Tree_node* k2), Tree_node* Left_
```

, SplayTree takes in a key value, and a pointer to the root of the splay tree as input, and performs a top-down splay operation on this key value, returning the new root of the resulting splay tree. The other two functions take in the root of a a tree, and do a single rotation to the right or left, returning the new root. We also provide you with a makefile for the executable. These files will be in the directorry

```
/projectnb/ec504/HW1_codes
```

Look up on the internet (or on Weiss' book) how to implement top-down splaying, and write the missing function. Make sure you understand how the function works. A brief pseudocode for this function is:

```
function Splay(int key, Tree_node* root)
    if root is null, return null;
    Create a Tree_node Tree to grow the right, left subtrees as we go down.
    Initialize right and left children of Tree as null
    Initialize the pointers Rpoint, Lpoint as to where to insert left and right subtrees: the address
    Do until break:
        if key < root key,
            if root has no left_child, break out of Do;
            if (key <  left_child's key,
                rotate root to the right, using Right_Rotate.
                if the new root has no left child, break out of Do;
            Split current root and all of its right descendants into right subtree by:
                Make Rpoint's left child point to root
                Define new Rpoint equal to root
                Define new root to be root's left child
                Make Rpoint's left child null
        else if (key > root->key)
            if root has no right child, break out of do.
            if (key > right_child's key,
                rotate root to the left, using Left_Rotate
                if  the new root has no right child, break out of Do
            Split current root and all its left descendants to left subtree by:
                Make Lpoint's right child point to root
                Define new Lpoint to point to root
                Define new root to be root's right child
                Make Lpoint's right child to be null
        else break out of Do: the root has the key you are looking for.
    Assemble the final splay tree by merging center, left and right, by:
```

```
        Make Lpoint's right child point to root's left child
        Make Rpoint's left child point to root's right child
        Make root's left child point to Tree's right child
        Make root's right child point to Tree's left child
    return root
```

Once you have implemented your function correctly, run the command `./topdownsplay >> hw1splay.out`. Include your completed `topdownsplay.cpp` code and the output file `hw1splay.out` in your directory in a subdirectory titled HW1. /projectnb/ec504/, in HW1 subdirectory.