Assignment 2
John Abo
30088517

1. One way to implement a stack that allows for $MAX(S)$ to return the maximum key currently in stack S is to include a second parallel stack M that keeps track of max and is modified along with S. In the pseudocode, the pointer to the top of the stack are treated as the same for both stacks as they are the same size.

   $PUSH(S, k)$ will compare k with $M.top$ and will push k to S and the max of the two compared values to M, which will allow for the max to always be at top of stack.

   $Push(S, k)$
   $\quad S.top = S.top + 1;$
   $\quad S[S.top] = k;$

   $\quad if\ S.M[S.top - 1] < k;$
   $\quad\quad S.M[S.top] = k;$
   $\quad else$
   $\quad\quad$ //This line duplicates the top of M
   $\quad\quad S.M[S.top] = S.M[S.top - 1];$

   This is done in constant time

   $POP(S)$ will still pop the top of stack from S, but will now also decrement the top of stack from M.

   $Pop(S)$
   $\quad Input{:}S\ is\ a\ non-empty\ stack;$
   $\quad S.top = S.top - 1;$
   $\quad return\ S[S.top + 1];$

   This is also done in constant time

   $TOP(S)$ does not change, and still returns the key at the top of S.

   $Top(S)$
   $\quad return\ S[S.top];$

   $MAX(S)$ will function similar to $TOP(S)$, but instead of the top of stack S, it returns the top of stack M, which will contain the value of the max key in S at the top.

   $Max(S)$
   $\quad return\ S.M[S.top];$

2. Fong said making an array is okay, we're in the last stretch now boys

The conversion requires a separate subroutine that involves turning a given array into a Binary Search Tree. It will do this by setting the middle node as the root, then assigning children from subarrays to the left and right of the middle position ($A[p..midle - 1]$ and $A[middle + 1]$)

$ArrayToTree(A,p,r)$:

Input: A is a sorted array with middle element at a position between p and r

Output: Node containing key matching middle value of array A, and any child node from recursive calls

If $p > r$ then
    Return nil;
Else
    $mid = \lfloor (p + r)/2 \rfloor$;
    $node.key = A[mid]$
    $node.left = ArrayToTree(A,p,mid - 1)$;
    $node.right = ArrayToTree(A,mid + 1,r)$;
    Return $node$;

Using $T(n)$ as run time of $ArrayToTree$, we can see that

$$T(n) = 2T(n/2) + O(1)$$

As there are 2 recursive calls followed by the constant time needed to access the middle of the array.

Using the master theorem, we see that $O(1)$ is polynomially smaller than $n^{\log_2 2} = n$, so we get that

$$T(n) = \Theta\left(n^{\log_2 2}\right) = \Theta(n)$$

So this subroutine will be running in linear time

$Convert(D)$:

Input: D is a sorted doubly linked list that contains at least one element

//Create array A that will hold all the keys from linked list D while maintain order
$y = D.head$
$i = 1$

While $y \neq nil$ do
    $A[i] = y.key$;
    $y = y.next$;
    $i = i + 1$;
//This while loop runs at linear time, as it always runs through the entire linked list

Return $ArrayToTree(A,0,i - 1)$;

$Convert$ runs at $T(n) = n + \Theta(n)$, so this runs at constant time

3. Many subroutines will need to be used; some will be modified to account for the new specifications.

Search would go unchanged, as it does not require the node to know it's parent

$Search(x, k)$:
    If $x = nil$ or $k = x.key$ then
        Return x;
    If $k < x.key$ then
        Return $Search(x.left, k)$;
    Else
        Return $Search(x.right, k)$;

Delete requires a means to access the parent of the node z in order to change the child with transplant.
Using this subroutine, it'll now also give us access to the parent node of z.

$NodeParent(T, z)$:
    Input: z is a pointer to the node whose parent is being searched for in the subtree T. z must exist in the tree

    //Start with modified implementation of search that is iterative and keeps track of the previous node
    $y = nil$;
    $x = T.root$;

    While $x \neq nil$ and $x \neq z$ do   //x will equal nil only if the tree is empty
        If $z.key < x.key$ then
            $y = x$;
            $x = x.left$;
        Else
            $y = x$
            $x = x.right$;
    Return y;

$TreeMinimum$ is unchanged as given in the textbook and lecture slides, as it only needs to travel down left nodes starting from the root given until the node being looked at has no left child.

$TreeSucc$ is used often and is modified with:

$TreeSucc(T, x)$:
    Input: x is a node in the tree whose successor is being searched for in subtree T.

    If $x.right \neq nil$ then
        Return $TreeMinimum(x.right)$;

$y = NodeParent(T.root, x)$;

While $y \neq nil$ and $x = y.right$ do //While y is not root, and x is right child
$x = y$;
$y = NodeParent(T.root, y)$;

Return y;

It is assumed that the leaf nodes will hold a sentinel that contains a pointer to its successor, this is used in the following pseudocode for $insert$

$TreeInsertSucc(T, z)$:
Input: z is a node with children set to the sentinel nil, that will be inserted in tree T.

$parent = nil$; //Keeps track of the parent as the node will not keep track of it
$current = T.root$;

While $current \neq nil$ do
If $current.key < z.key$ then
$parent = current$;
$current = current.left$;
Else
$parent = current$;
$current = current.right$;
//At this point, current should be nil, and parent should point to the node that z will be attached to.
$z.succ = TreeSucc(T, z)$;

If $parent = nil$ then
$root = z$;
Else if $parent.key < z.key$ then
$parent.left = z$;
Else
$parent.right = z$;

Moving onto the delete pseudocode.

A modified version of $Transplant$ is also going to be used.

$Transplant(T, x, y)$:
Input: x is the node that will be replaced with y in subtree T.

//These are the parents of nodes x and y
$xParent = NodeParent(T.root, x)$;
$yParent = NodeParent(T.root, y)$;

//These change the child of x's parent based on which child x is

If $xParent = nil$ then //If x is a root node
    $T.root = y$;
Else if $x = xParent.left$ then //If x is a left child
    $xParent.left = y$;
Else
    $xParent.right = y$;

//Updates the successor of y
//If y has no right child, the successor must then come from the parent
If $y \neq nil$ and $y.right = nil$ then
    $y.succ = TreeSucc(T, y)$;

$TreeDeleteSucc(T, z)$:
Input: z is a pointer to a node in tree T that will be deleted.

If $z.left = nil$ then
    $Transplant(T.root, z, z.right)$;
Else if $z.right = nil$ then
    $Transplant(T.root, z, z.left)$;
Else
    $y = TreeMinimum(z.right)$;

    $yParent = NodeParent(T, y)$;
    If $yParent \neq z$ then
        $Transplant(T, y, y.right)$;
        $y.right = z.right$;
        $y.right.succ = TreeSucc(T, y.right)$;
    $Transplant(T, z, y)$;
    $y.left = z.left$;
    $y.left.succ = TreeSucc(T, y.right)$;

4. Code included with submission