Lab 12
Data Structures
BS DS Fall 2024 Morning/Afternoon

Instructions:

- **Attempt the following tasks exactly in the given order.**
- Make sure that there are no ***dangling pointers*** or ***memory leaks*** in your programs.
- Indent your code properly.
- Use meaningful variable and function names. Follow the naming conventions.
- Use meaningful prompt lines/labels for all input/output that is done by your programs.

Task 1

Add the following method in BST ADT which you implemented in the Last Lab.

void construct_from_traversals(vector<int> in_order, vector<int> pre_order)

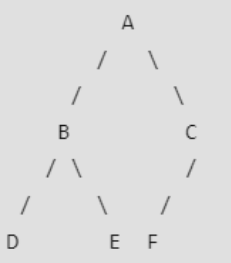
The method **construct_from_traversals** in the BST class is used to construct a binary search tree from its in-order and pre-order traversals.

Let us consider the below traversals:

- Inorder sequence: D B E A F C
- Preorder sequence: A B D E C F

In a Preorder sequence, the leftmost element is the root of the tree. So we know 'A' is the root for given sequences. By searching 'A' in the Inorder sequence, we can find out all elements on the left side of 'A' is in the left subtree and elements on right in the right subtree. So we know the below structure now.

We recursively follow the above steps and get the following tree.



Driver

```
int main() {
    BST bst;
    vector<int> in1 = {1, 2, 3, 4, 5, 6};
    vector<int> pre1 = {3, 1, 2, 5, 4, 6};

    bst.construct_from_traversals(in1, pre1);

    cout << "In-order traversal (Example 1): ";
    bst.display_in_order();

    cout << "Post-order traversal (Example 1): ";
    bst.display_post_order();

    BST bst2;
    vector<int> in2 = {5,10,15,25,27,30,35,40,45,50,52,55,60,65,70,75,80,85,90,100};
    vector<int> pre2 = {50,25,10,5,15,40,30,27,35,45,75,60,55,52,65,70,90,80,85,100};

    bst2.construct_from_traversals(in2, pre2);

    cout << "\nIn-order traversal (Example 2): ";
    bst2.display_in_order();

    cout << "Post-order traversal (Example 2): ";
    bst2.display_post_order();
    return 0;
}
```

The output of the following driver should be:

```
In-order traversal (Example 1): 1 2 3 4 5 6
Post-order traversal (Example 1): 2 1 4 6 5 3

In-order traversal (Example 2): 5 10 15 25 27 30 35 40 45 50 52 55 60 65 70 75 80 85 90 100
Post-order traversal (Example 2): 5 15 10 27 35 30 45 40 25 52 55 70 65 60 85 80 100 90 75 50
```

Task # 2

Add the following **public member function** to the **`BST`** class (for storing integers) which you implemented last week in (**Lab # 11 – Task # 2**):

`void BST::createBalancedTree (int* arr, int start, int end)`

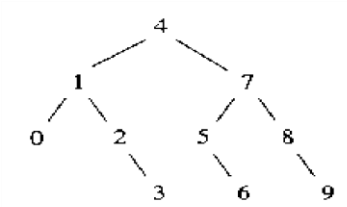
This function will take an array of integers (**`arr`**) which is sorted in increasing order, and its starting (**`start`**) and ending (**`end`**) indices. You can assume that all the integers present in the array are distinct (i.e. there is no repetition).

When called on an empty BST, the above function should insert the values of the given array to create a balanced BST. If the BST is non-empty, this function should firstly make the tree empty (using the recursive **`destroy(…)`** function), and then insert the given values to create a balanced BST.

Note that:

- The function **`createBalancedTree(…)`** will be implemented **recursively**.
- This function will use the **`insert(…)`** function to insert a value into the BST.

For example, if the array contains the following elements **{0 1 2 3 4 5 6 7 8 9}**, the following BST should be created by the above function:



In your driver program, after you have created the balanced tree, you should display the **preorder**, **in-order**, and **post-order** traversals of the tree to convince yourself (and the TA's) that you have properly implemented the algorithm.

Driver

```
int main() {
    BST bst;

    int arr[] = {0,1,2,3,4,5,6,7,8,9};
    int n = sizeof(arr) / sizeof(arr[0]);

    bst.createBalancedTree(arr, 0, n - 1);

    cout << "Pre-order: ";
    bst.display_pre_order();

    cout << "In-order: ";
    bst.display_in_order();

    cout << "Post-order: ";
    bst.display_post_order();
    return 0;
}
```

The output of the following driver should be:

```
Pre-order: 4 1 0 2 3 7 5 6 8 9
In-order: 0 1 2 3 4 5 6 7 8 9
Post-order: 0 3 2 1 6 5 9 8 7 4
```

Task # 3

*Here is the description of the **DSW algorithm** for creating a balanced BST as described in Section 6.7.1 of the book "Data Structures and Algorithms in C++" (4th Edition) by Adam Drozdek:*

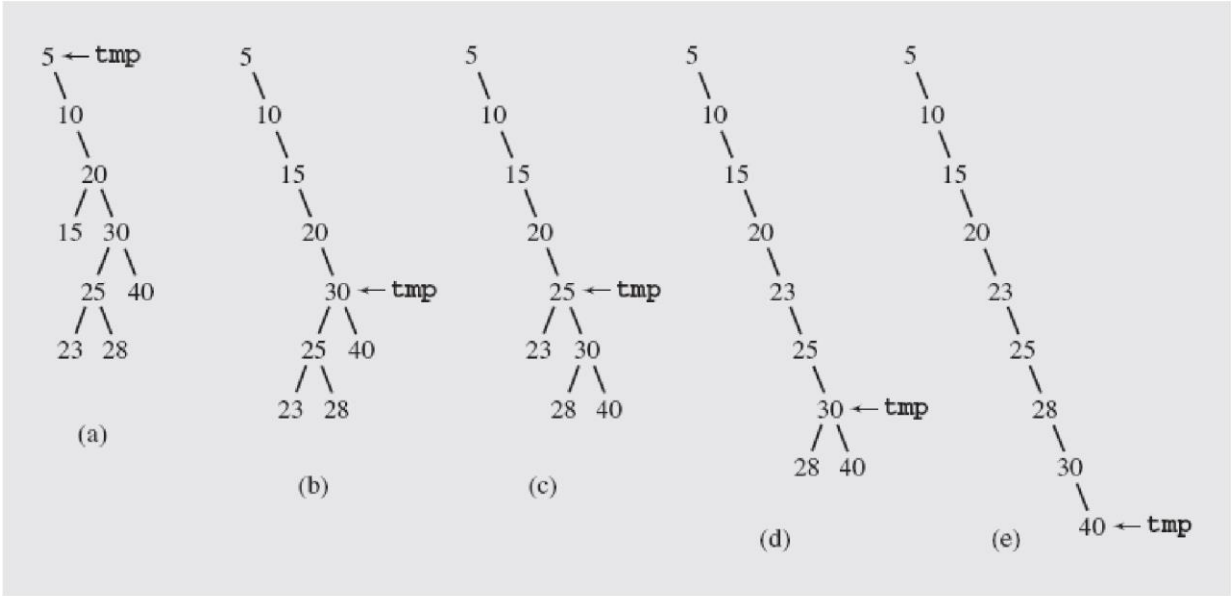
The DSW algorithm transfigures an arbitrary binary search tree into a linked-list like tree called a *backbone* or *vine*. Then this elongated tree is transformed in a series of passes into a perfectly balanced tree by repeatedly rotating every second node of the backbone about its parent.

The algorithm for creating a backbone out of a BST is as follows:

```
createBackbone(root)
      tmp = root;
      while (tmp != 0)
            if tmp has a left child
                  right-rotate this child about tmp;
                                          // left child becomes parent of tmp
                  set tmp to the child that just became parent;
            else set tmp to its right child;
```

Note that the above algorithm uses the **Right-Rotation** which we have discussed in class. See the following figure for an example showing the working of the above algorithm:

FIGURE 6.38 Transforming a binary search tree into a backbone.



You are required to implement a member function of the **BST** class which converts the BST into a backbone using the above-mentioned algorithm. The prototype of your function should be:

void BST::convertToBackbone ()

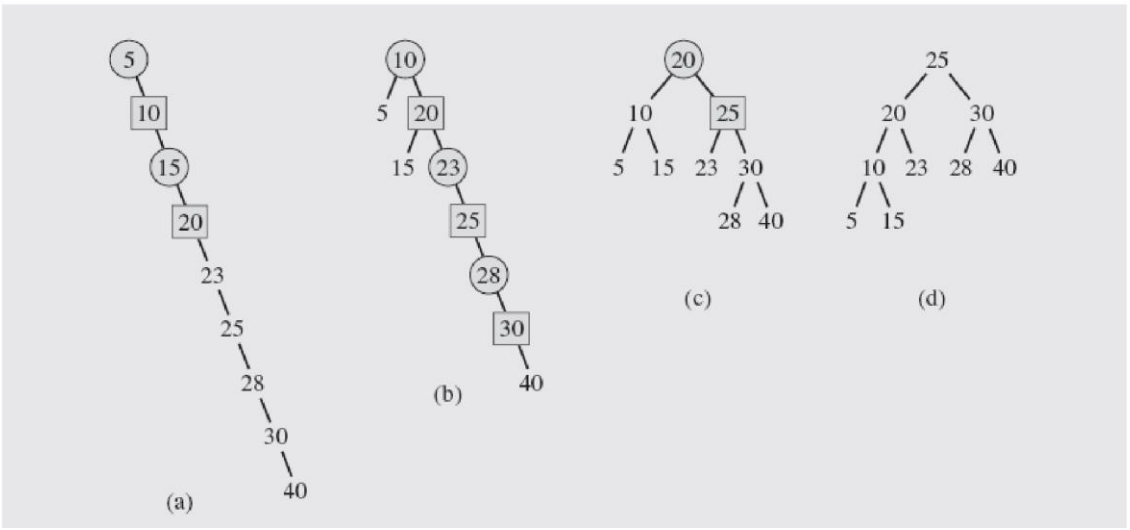
In the second phase of the DSW algorithm, the backbone is transformed into a tree, but this time, the tree is perfectly balanced by having leaves only on two adjacent levels. In each pass down the backbone, **every second node** down to a certain point is rotated about its parent.

```
createPerfectTree()
  1. n = number of nodes;
  2. m = 2^⌊lg(n+1)⌋ - 1 ;
  3. make n-m left rotations starting from the top of backbone;
  4. while (m > 1)
        a. m = m/2;
        b. make m left rotations starting from the top of backbone;
```

The following figure (Fig. 6.39) shows the conversion of the backbone created previously (in Fig. 6.38) into a perfectly Balanced BST.

- Fig. 6.39 (a) shows the initial backbone tree.
- Fig. 6.39 (b) shows the tree after performing the **n-m** (9 – 7 = **2**) left rotations as specified in Line # 3 of the above algorithm.
- Fig. 6.39 (c) shows the tree after the 3 left rotations performed in the first iteration of the while loop of Line # 4 of the above algorithm.
- Fig. 6.39 (d) shows the tree after the 1 left rotation performed in the second (and also the last) iteration of the while loop of Line # 4. This is the final balanced tree.

FIGURE 6.39 Transforming a backbone into a perfectly balanced tree.



You are required to implement a member function of the **BST** class which converts any BST into a perfectly balanced BST using the DSW algorithm described above. The prototype of your function should be:

void BST::createPerfectBST ()

First of all, this function will use the **`convertToBackbone()`** function to convert the BST into a backbone. Then, it will convert this backbone tree into a perfectly balanced BST.

Driver

```
int main() {
    BST bst;
    bst.insert(1);
    bst.insert(5);
    bst.insert(10);
    bst.insert(20);
    bst.insert(25);
    bst.insert(30);
    bst.insert(35);
    bst.insert(40);
    bst.insert(45);

    cout<<"Original BST  Height: "<<bst.getHeight()<<endl;

    cout << "\nOriginal BST (Inorder): ";
          bst.display_in_order();
    cout << "\nOriginal BST Pre-order: ";
    bst.display_pre_order();

    cout << "\nOriginal BST Post-order: ";
    bst.display_post_order();

    bst.createPerfectBST();

    cout<<"\nHeight After DSW: "<<bst.getHeight()<<endl;

    cout << "\nBalanced BST using DSW (Inorder): ";
    bst.display_in_order();

    cout << "\nBalanced BST using (DSWPre-order:) ";
    bst.display_pre_order();

    cout << "\nBalanced BST using DSW (Post-order:) ";
    bst.display_post_order();

    return 0;
}
```

The output of the following driver should be:

```
Original BST  Height: 8

Original BST (Inorder): 1 5 10 20 25 30 35 40 45

Original BST Pre-order: 1 5 10 20 25 30 35 40 45

Original BST Post-order: 45 40 35 30 25 20 10 5 1

Height After DSW: 3

Balanced BST using DSW (Inorder): 1 5 10 20 25 30 35 40 45

Balanced BST using (DSWPre-order:) 30 20 5 1 10 25 40 35 45

Balanced BST using DSW (Post-order:) 1 10 5 25 20 35 45 40 30
```