Middle East Technical University | Department of Computer Engineering

CENG 213

Data Structures

Fall 2024

Programming Assignment 2

Due date: December 08, 2024, Sunday, 23:55

1 Objectives

In this programming assignment, you will first implement a self-balancing binary search tree which uses the Day–Stout–Warren (DSW) algorithm for balancing. Each node of the tree will contain the data element and two pointers to the root nodes of its left and right subtrees. The binary tree data structure will include a single pointer that points to the root node of the tree. The details of the structure are explained further in the following sections. Then, you will use this specialized binary search tree structure to implement a tree map abstract data type (i.e., a balanced binary tree that keeps its entries (key-value pairs) sorted according to the natural ordering of its keys). And finally you will use the tree map in a text retrieval application.

Keywords: C++, Data Structures, Binary Search Trees, Self-Balancing,Day-Stout-Warren (DSW) Algorithm, TreeMap Implementation

2 Overview of the Day-Stout-Warren (DSW) Algorithm

The Day-Stout-Warren (DSW) algorithm is a technique for balancing a binary search tree (BST) to transform it into a complete binary tree, minimizing its height to approximately $\lfloor \log_2(n) \rfloor$, where n is the number of nodes. Unlike an AVL tree, it does not do this incrementally during each insertion or deletion, but periodically, or as needed, making it suitable in scenarios where global balancing is sufficient.

The building block for tree transformation in this algorithm is single rotation. There are two types of rotation, left and right, which are symmetrical to one another, just like in AVL trees. Basically the DSW algorithm transforms 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.

In the first phase, a backbone is created using the following algorithm:

```
createBackbone(root, n)
   tmp = root:
   while (tmp!=null)
      if tmp has a left child
         rotate tmp right with this child (hence left child becomes parent of tmp)
         set tmp to the child which just became parent;
      else set tmp to its right child;
```

This algorithm is illustrated in Section 8.1.

In the second phase, the backbone is transformed into a complete tree. In each pass down the backbone, every second node down to a certain point is rotated left with its parent. The initial pass is used to account for the difference between the number n of nodes in the curent tree and the number $2^{\lfloor \log_2(n+1) \rfloor} - 1$ of nodes in the closest complete binary tree.

```
createCompleteTree(n)
```

m $= 2^{\lfloor \log_2(n+1) \rfloor}$;

```
make n-m left rotations starting from the top of backbone;
while (m > 1)
   m = m/2;
   make m rotations starting from the top of backbone;
```

Note that in this algorithm, in each iteration of the while loop, the backbone corresponds to the rightmost path on the tree starting from the root to the node containing the maximum element. The backbone is halved in size after every iteration of the while loop.

There will be $n - m$ number of nodes on the last level of the complete tree. The initial $n - m$ left rotations outside the while loop is a preprocessing step to prepare the bottommost left of the complete tree. Once this initial step is finished, the while loop is used to left rotate every second node in the backbone. The number of rotations to be performed is calculated by m which is halved in each iteration.

See references for further information on DSW algorithm:

```
https://en.wikipedia.org/wiki/Day%E2%80%93Stout%E2%80%93Warren_algorithm
https://liacs.leidenuniv.nl/~rijnjnvan/ds2013/assets/opdrachten/dsw.pdf
```

3 Binary Search Tree with DSW Balancing (50 pts)

The binary search tree data structure used in this assignment is implemented as the class template DSWTree with the template argument T, which is used as the type of the element stored in the nodes. The node of the tree is implemented as the class template Node with the template argument T, which is the type of the element stored in nodes. Node class is the basic building block of the DSWTree class. DSWTree class has a Node pointer (namely root) which points to the root node of the binary search tree, and an integer (namely height) which keeps the current height of the node in its private data field.

The DSWTree class has its definition and implementation in DSWTree.h file and the Node class has its in Node.h file.

3.1 Node Class

Node class represents nodes that constitute DSW trees. A Node keeps two pointers (namely left and right) to the root nodes of its left and right subtrees, a data variable of type T (namely element) to hold the data and an integer which represents the height of the node. The class has two constructors and the overloaded output operator. They are already implemented for you. You should not change anything in file Node.h.

3.2 DSWTree Class

DSWTree class implements a binary search tree data structure which uses DSW algorithm for balancing. Previously, data members of DSWTree class have been briefly described. Their use will be elaborated in the context of utility functions discussed in the following subsections. You must provide implementations for the following public interface methods that have been declared under indicated portions of DSWTree.h file.

3.2.1 `DSWTree();`

This is the default constructor. You should make necessary initializations in this function.

3.2.2 `DSWTree(const DSWTree<T> &obj);`

This is the copy constructor. You should make necessary initializations, create new nodes by copying the nodes of the given obj, and insert those new nodes into this DSW tree. The structure among the nodes of the given obj should also be copied to this DSW tree.

3.2.3 `~DSWTree();`

This is the destructor. You should deallocate all the memory that were allocated before.

3.2.4 `bool isEmpty() const;`

This function should return true if this DSW tree is empty (i.e., there are no nodes in this DSW tree). If this DSW tree is not empty, it should return **false**.

3.2.5 `int getHeight() const;`

This function should return the height of the root node.

3.2.6 `const int getNodeHeight(const T& element) const;`

This function returns the height of the node containing the specified `element`. The height is defined as the length of the longest path from the node to a leaf node. If the element is not found in the tree, the function returns `-1`.

3.2.7 `int getSize() const;`

This function should return an integer that is the number of nodes in this DSW tree.

3.2.8 `bool insert(const T &element);`

The `insert` function adds a new node with the given `element` to the binary search tree while maintaining the BST properties. Do not forget to make necessary pointer modifications in the tree. Since each `Node` contains a `height` attribute, the function updates the `height` of each node along the insertion path, ensuring that all heights reflect the correct subtree depths after insertion. It is guaranteed that you will not be asked to insert duplicated data to the binary search tree.

3.2.9 `bool remove(const T &element);`

You should remove the node with the given element from this DSW tree and return true. Don't forget to make necessary pointer and height modifications in the tree. If there is no such node in the binary search tree this DSW tree, this function should return false. There will be no duplicated elements in the DSW tree.

3.2.10 `void removeAllNodes();`

You should remove all nodes of the DSW tree so that the tree becomes empty. Don't forget to make necessary pointer modifications in the tree.

3.2.11 `const T &get(const T &element) const;`

You should search this DSW tree for the node that has the same element with the given element and return the element of that node. You can use the operator== to compare two T objects. If there exists no such node in this DSW tree, this function should throw NoSuchItemException. There will be no duplicated elements in the DSW tree.

3.2.12 `void print(TraversalMethod tp=inorder) const;`

Given a traversal method (inorder, preorder or postorder) as parameter (namely tp), this function prints this DSW tree by traversing its nodes accordingly. The code for inorder printing is already given. You should complete this function for preorder and postorder printing. You should follow the printing format used for the given inorder printing to implement the others.

3.2.13 `DSWTree<T> &operator=(const DSWTree<T> &rhs);`

This is the overloaded assignment operator. You should remove all nodes of this DSW tree and then, you should create new nodes by copying the nodes of the given rhs DSW tree and insert those new nodes into this DSW tree. The structure among the nodes of the given rhs should also be copied to this DSW tree.

3.2.14 `bool isBalanced()`

The `isBalanced` function checks if this DSW tree is balanced. Here, a balanced tree means that for every node, the height difference between its left and right subtrees is no more than one. Since each `Node` contains its height information, this function can use that data directly to verify balance without recalculating heights.

3.2.15 `void balance()`

The `balance()` function initiates the DSW algorithm. This algorithm ensures that the BST is a complete binary tree with its height minimized to $\log_2(\texttt{size})$, which optimizes search, insertion, and deletion operations. This function calls two functions `createBackbone` and `createCompleteTree`.

3.2.16 `void createBackbone(Node<T> *&root)`

This function performs a right rotation on every node with a left child in the given BST, ensuring that all nodes form a single right-skewed linked list like tree. It starts at the root of the BST. While traversing the tree, if the current node has a left child, perform a right rotation around it to eliminate the left child. If there is no left child, it moves to the right child. This process is repeated until the tree is straightened into a "linked-list-like" structure. Make sure that the height information in each node is updated correctly.

3.2.17 `void createCompleteTree(Node<T> *&root, int n)`

The `createCompleteTree` function performs a series of left rotations to convert the backbone into a complete binary tree as described in Section 2.

3.2.18 `const T &getCeiling(const T &element) const;`

You should search this DSW tree for the node associated with the least element greater than or equal to the given element and return the element of that node. You can use the operator== to compare two T objects. If there exists no such node in this DSW tree (i.e., all nodes have elements less than the given element), this function should throw NoSuchItemException. There will be no duplicated elements in the DSW tree.

3.2.19 `const T &getFloor(const T &element) const;`

You should search this DSW tree for the node associated with the greatest element less than or equal to the given element and return the element of that node. You can use the operator== to compare two T objects. If there exists no such node in this DSW tree (i.e., all nodes have elements greater than the given element), this function should throw NoSuchItemException. There will be no duplicated elements in the DSW tree.

3.2.20 `const T &getMin() const;`

You should search this DSW tree for the node associated with the smallest element and return the element of that node. You can use the operator== to compare two T objects. If there exists no such node in this DSW tree (i.e., the DSW tree is empty), this function should throw NoSuchItemException. There will be no duplicated elements in the DSW tree.

3.2.21 `const T &getMax() const;`

You should search this DSW tree for the node associated with the greatest element and return the element of that node. You can use the operator== to compare two T objects. If there exists no such node in this DSW tree (i.e., the DSW tree is empty), this function should throw NoSuchItemException. There will be no duplicated elements in the DSW tree.

3.2.22 `const T &getNext(const T &element) const;`

You should search this DSW tree for the node associated with the smallest element greater than the given element and return the element of that node. You can use the operator== to compare two T objects. If there exists no such node in this DSW tree (i.e., all nodes haxve elements less than or equal to the given element), this function should throw NoSuchItemException. There will be no duplicated elements in the DSW tree.

4 Tree Map Implementation (30 pts)

Tree map is a DSW tree based map implementation that keeps its entries sorted according to the natural ordering of their keys. Tree maps are used to store data values in key:value pairs. A tree map is a collection which is ordered (i.e., the entries have a defined order and that order will not change), changeable (i.e., we can update, add or remove entries after the tree map has been created) and do not allow duplicates (i.e., tree maps cannot have two entries with the same key). Tree map entries are presented in key:value pairs, and can be referred to by using the keys.

The tree map in this assignment is implemented as the class template `TreeMap` with the template argument `K`, which is used as the type of keys stored in the map, and `V`, which is used as the type of values stored in the map. `TreeMap` class has a `DSWTree` object in its private data field (namely `stree`) with the type `KeyValuePair<K, V>`. This `DSWTree` object keeps the key-value mappings (or pairs) in the tree map. `KeyValuePair` class represents the key-value mappings in the tree map.

The `TreeMap` and `KeyValuePair` classes has their definitions in *TreeMap.h* and *KeyValuePair.h* files, respectively.

4.1 KeyValuePair Class

`KeyValuePair` class represents key-value mappings that constitute tree maps. A `KeyValuepair` object keeps a variable of type `K` (namely key) to hold the key, and a variable of type `V` (namely value) to hold the value. The class has three constructors, overloaded relational operators, getters, setters, and the overloaded output operator. They are already implemented for you. Note that the overloaded relational operators compare only keys of the objects to decide. You should not change anything in file *KeyValuePair.h*.

4.2 TreeMap Class

In `TreeMap` class, all member functions should utilize `stree` member variable to operate as described in the following subsections. In *TreeMap.h* file, you need to provide implementations for following functions declared under *TreeMap.h* header to complete the assignment.

4.2.1 `void clear();`

This function removes all key-value mappings from this map so that it becomes empty.

4.2.2 `const V &get(const K &key) const;`

This function returns the value of the key-value mapping specified with the given `key` from this map. If there exists no such key-value mapping in this tree map, this function should throw `NoSuchItemException`.

4.2.3 `V& operator[] (const K &key) const;`

This functions uses `get` function above to overload `operator[]`.

4.2.4 `void put(const K &key, const V &value);`

This function adds a new key-value mapping specified with the given `key` and value to this map. It takes the mapping information (`key` and `value`) as parameter and inserts a new `KeyValuePair` object to the `stree` DSW tree. If there is already a mapping with the given `key` in this tree map, this function should update the mapping of the key with `value`.

4.2.5 `bool containsKey(const K &key);`

This function returns true if the `key` is associated with a value mapping. If there is no such key-value mapping in this tree map, this function should return false.

4.2.6 `bool deletekey(const K &key);`

This function removes the key-value mapping specified with the given `key` from this map and returns true. If there is no such key-value mapping in this tree map, this function should return false.

4.2.7 `const KeyValuePair<K, V> &ceilingEntry(const K &key);`

This function returns a key-value mapping from this tree map associated with the smallest key greater than or equal to the given key. If there exists no such key-value mapping in this tree map, this function should throw `NoSuchItemException`.

4.2.8 `const KeyValuePair<K, V> &firstEntry();`

This function returns a key-value mapping from this tree map associated with the least key. If there exists no such key-value mapping in this tree map, this function should throw `NoSuchItemException`.

4.2.9 `const KeyValuePair<K, V> &lastEntry();`

This function returns a key-value mapping from this tree map associated with the greatest key. If there is no such key-value mapping in this tree map, this function should throw `NoSuchItemException`.

4.2.10 `std::vector<KeyValuePair<K, V> > subMap(K fromKey, K toKey) const;`

This function returns a portion of this map whose keys range from `fromKey`, inclusive, to `toKey`, inclusive as `std::vector` of key-value mappings. For this function, you may assume that there are key-value mappings in this map with keys `fromKey` and `toKey`, and you may also assume that `fromKey` is less than or equal to `toKey`.

4.2.11 `void balance();`

Rebalances the underlying `stree` using its `balance` method, optimizing the tree's structure for efficient operations.

5 Inverted Index Implementation(20)

Many search engines incorporate an inverted index when evaluating a search query to quickly find documents containing the words in a query. Because the inverted index stores a list of the documents containing each word, the search engine can use direct access to find the documents associated with each word in the query in order to retrieve the matching documents quickly. Figure 1 illustrates a simple inverted index where for each word the list of document ids are stored in ascending order.

Such an index determines which documents match a query word. If the query includes more than one word then the docid list of each word must be retrieved and the docids which occur in all lists (i.e. intersection of all lists) will be returned as the query result.

For this assignment, you are going to implement such an inverted index by using your `TreeMap` class. The inverted index in this assignment is implemented as the class `InvertedIndex`. `InvertedIndex` class has a `TreeMap` object (namely `invertedIndex`) with the type

Figure 1: A simplified illustration of a sample inverted index

Word	Documents
the	Document 1, Document 3, Document 4, Document 5, Document 7
cow	Document 2, Document 3, Document 4
says	Document 5
moo	Document 7

`TreeMap<std::string, std::vector<int>>`. The `InvertedIndex` class has its definition in *InvertedIndex.h* file and its implementation in *InvertedIndex.cpp* file.

5.1 `InvertedIndex Class`

In `InvertedIndex` class, there is a single member variable named as `invertedIndex`, which is a `TreeMap` object. The mapping of a word (key) to the list of document ids (value) will be stored in this map. All member functions should utilize this `TreeMap` to operate as described above. Some public member functions have already been implemented for you. Do not change those implementations. In *InvertedIndex.cpp* file, you need to provide implementations for following functions declared under *InvertedIndex.hpp* header to complete the assignment. You should not change anything in file *InvertedIndex.hpp*.

5.1.1 `void addDocument(const std::string &documentName, int docid);`

This is the member function that takes the name of a document as string, reads that document word by word, and populates the inverted index with (word, [docid]) pairs. It is guaranteed that documents will be text files with extension ".txt" and their content will be list of words separated by single space characters. No characters other than [a-z] and the space character will be given in documents. If it is a new word the `invertedindex` should be updated with this new key and a value which is a vector including the docid. If the word already exists in the `invertedindex`, then the value vector is updated. Note that the vector must be kept sorted and there should not be any duplicates. Some parts of this function are already implemented for you. You should complete the implementation of this function.

5.1.2 `std::vector<int> search(const std::string &query);`

The `search` function takes a query string, which can be a single word or a sequence of words separated by spaces. It processes the query as follows:

1. **Split Query:** The function splits the `query` string into individual words based on whitespace.
2. **Retrieve Vectors:** For each word, it retrieves the corresponding vector from the inverted index, which contains documents where the word occurs.
3. **Combine Results:** It combines these vectors into a single result vector, ensuring that all relevant documents for each word in the query are included.

This function returns a combined vector of document ids that match any word in the query, enabling a search across multiple words in the inverted index.

6 Regulations

1. **Programming Language:** You will use C++.

2. **Standard Template Library is allowed only for `vector`. Usage of `std::find` and `std::sort` is not permitted.**

3. **External libraries other than those already included are not allowed.**

4. **Operations must utilize the tree structures; failure to do so will result in a zero grade.**

5. **Modifying provided functions or adding unauthorized variables will result in a zero grade.**

6. **Late Submission:** Each student receives 5 late days for the entire semester. You may use late days on programming assignments, and each allows you to submit up to 24 hours late without penalty. For example, if an assignment is due on Thursday at 11:30pm, you could use 2 late days to submit it on Saturday by 11:30pm with no penalty. Once a student has used up all their late days, each successive day that an assignment is late will result in a loss of 5% on that assignment.

   No assignment may be submitted more than 3 days (72 hours) late without permission from the course instructor. In other words, this means there is a practical upper limit of 3 late days usable per assignment. If unusual circumstances truly beyond your control prevent you from submitting an assignment, you should discuss this with the course staff as soon as possible. If you contact us well in advance of the deadline, we may be able to show more flexibility in some cases.

7. **Cheating:** We have a zero tolerance policy for cheating. In case of cheating, all parts involved (source(s) and receiver(s)) get zero. People involved in cheating will be punished according to the university regulations. Remember that students of this course are bound to the code of honor and its violation is subject to severe punishment.

8. **Newsgroup:** You must follow the Forum (odtuclass.metu.edu.tr) for discussions and possible updates on a daily basis.

7 Submission

- Submission will be done via Class, (class.ceng.metu.edu.tr).
- Do not write a *main* function in any of your source files.
- A test environment will be ready in CengClass.
  - Submit your source files to CengClass for testing with evaluation inputs and outputs.
  - Additional test cases will be used for the evaluation of your final grade. So, your actual grades may be different than the ones you get in Class.
  - Only the last submission before the deadline will be graded.

8 Example

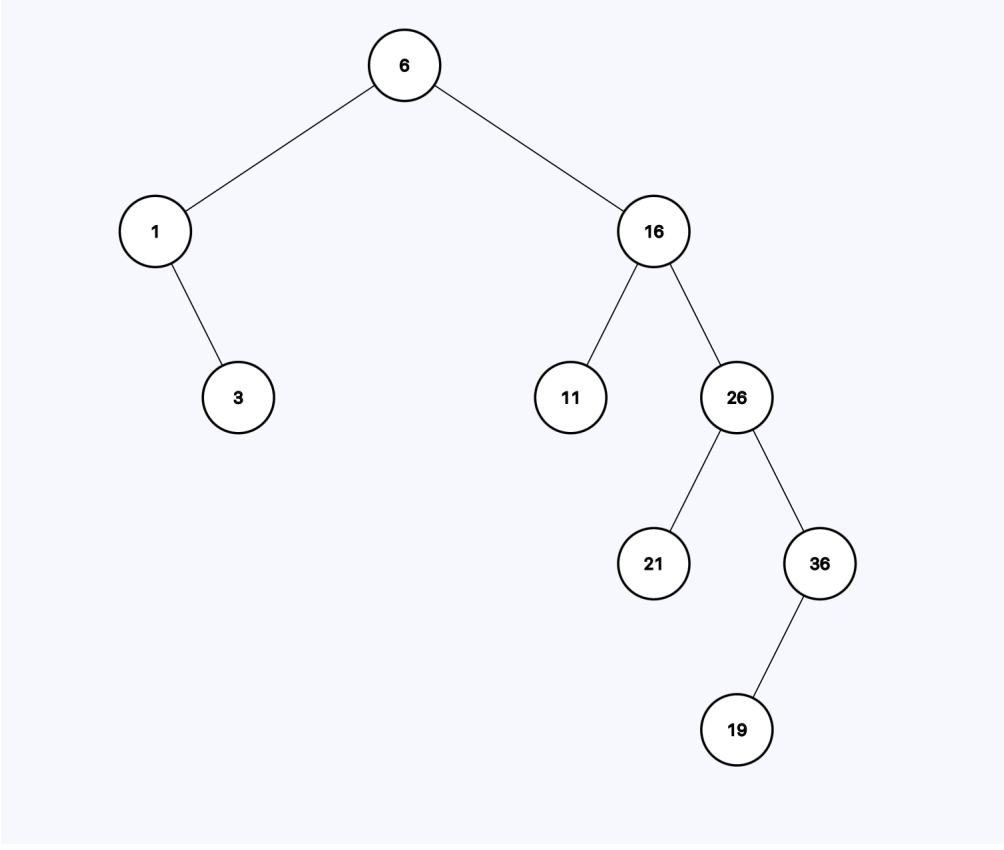
To demonstrate the **Day-Stout-Warren (DSW) balancing algorithm**, we begin by inserting nodes into a binary search tree (BST) in the following order:

```
1  DSWTree<std::int32_t> tree;
2
3  tree.insert(6);
4  tree.insert(1);
5  tree.insert(3);
6  tree.insert(16);
7  tree.insert(11);
8  tree.insert(26);
9  tree.insert(21);
10 tree.insert(36);
11 tree.insert(19);
```

Step 1: DSWTree Example

After inserting the nodes, the resulting BST is as shown below:

Figure 2: Initial Binary Search Tree



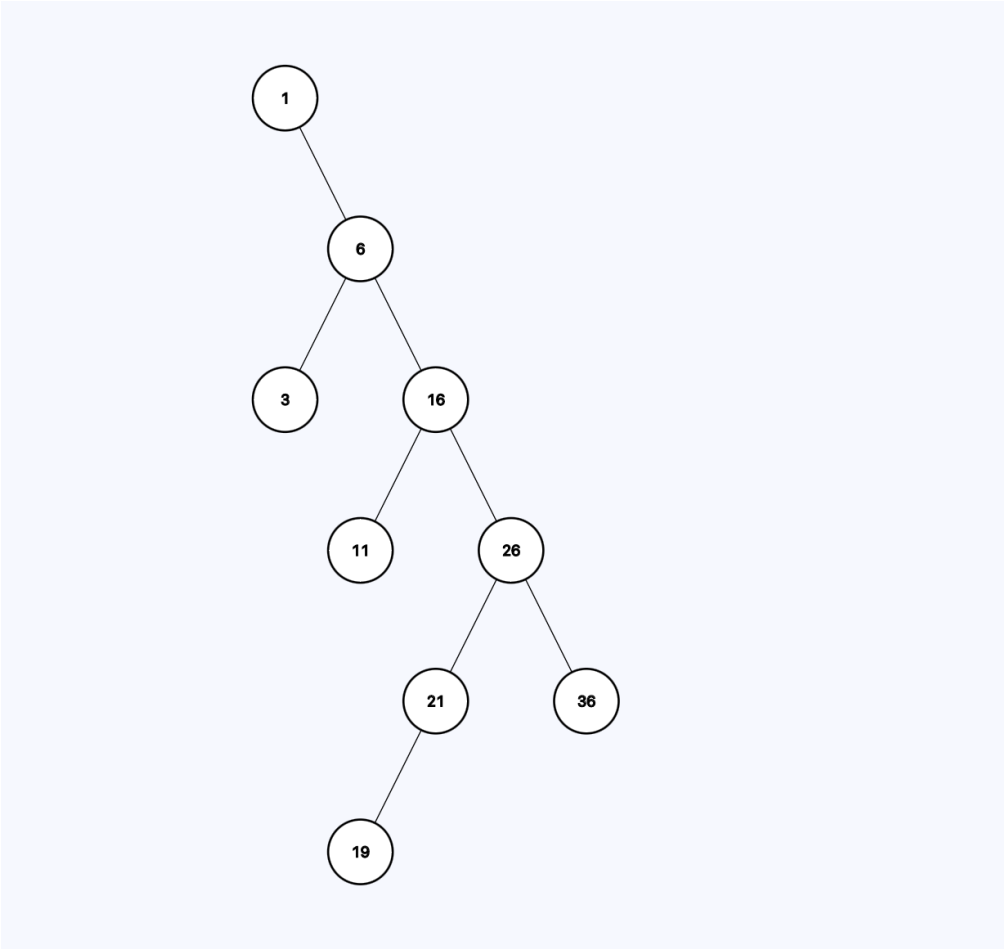
8.1 Step 1: Creating the Backbone (Vine)

The first step in the DSW algorithm is to transform the tree into a **backbone** (or **vine**) structure. A backbone is a completely right-skewed tree where every node has only a right child. This can be achieved by performing **right rotations** on every node that has a left child.

Example of Backbone Transformation:

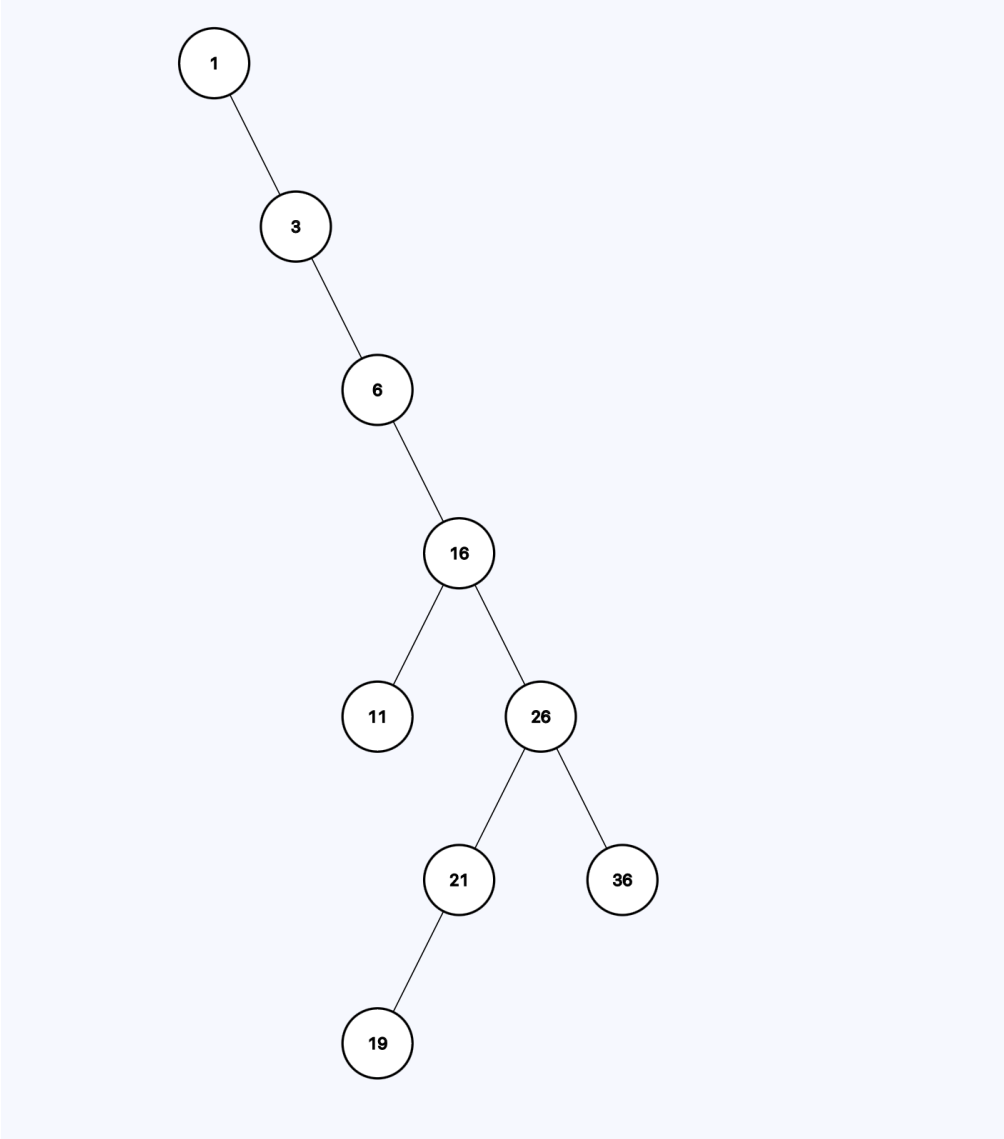
- Starting at the root (6), we observe it has a left child (1). We perform a **right rotation** on 6, resulting in:

Figure 3: After Right Rotate on Node 6



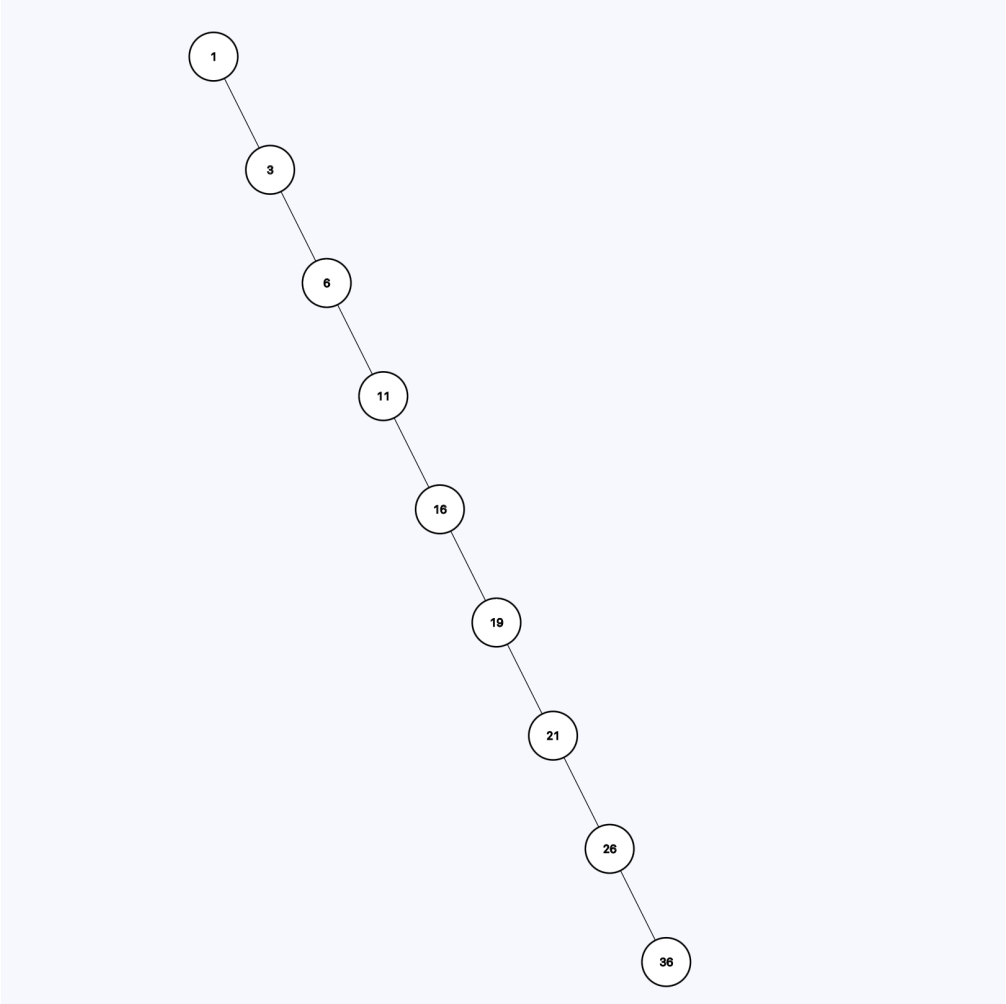
- The new root is now 1. Since 1 has no left child, we move to its right child (6). 6 has a left child (3), so we perform a **right rotation** on 6, resulting in:

Figure 4: After Right Rotate on Node 6 (Second Time)



- We continue this process, rotating each node with a left child, until the tree becomes a fully right-skewed backbone:

Figure 5: Final Backbone Structure



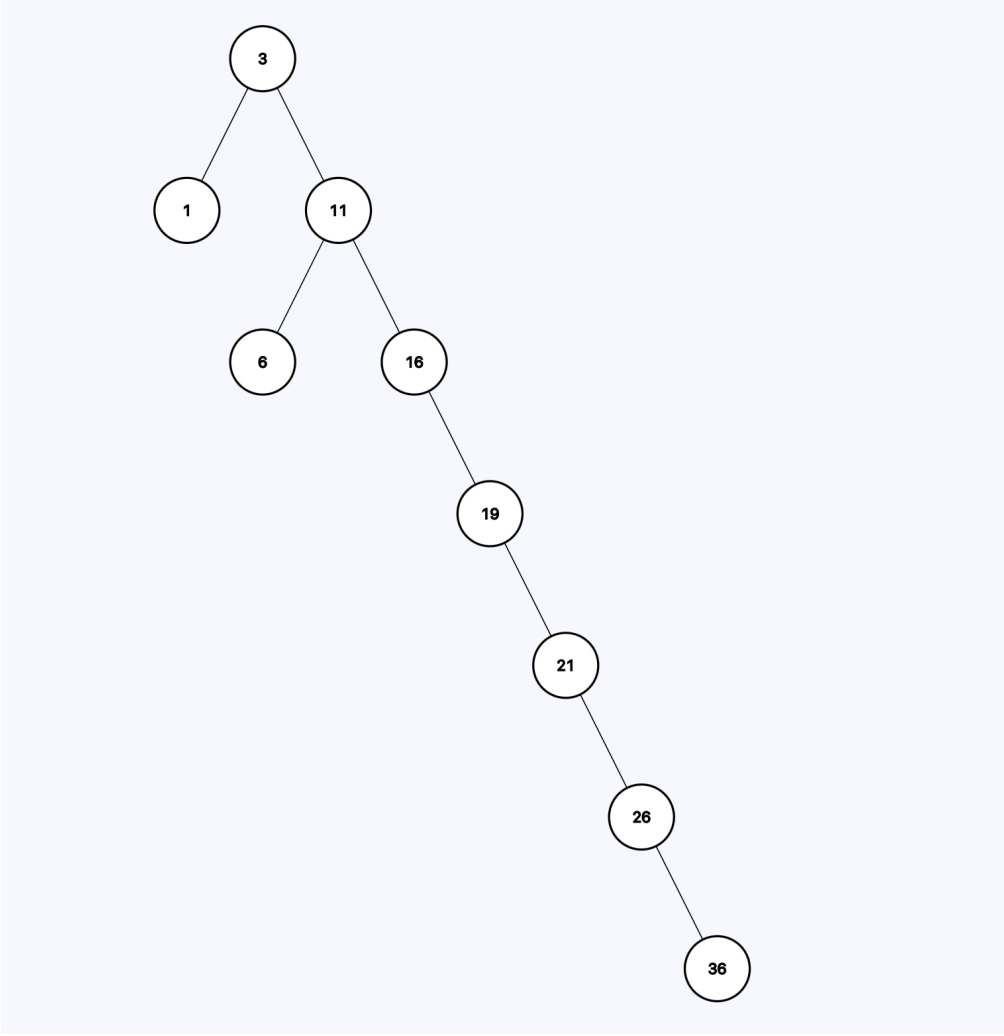
8.2 Step 2: Balancing the Tree

With the backbone constructed, the next step is to perform a series of **left rotations** to transform the backbone into a complete binary search tree.

Initial Left Rotations for the Bottommost Level:

- Calculate the number of nodes expected at the bottommost level. For 9 nodes, the bottom level should contain 2 nodes.
- Starting from the root, we rotate the first 2 odd-indexed nodes left with their right child in the backbone. After the initial rotations, the two nodes on the left of bottom level (namely 1 and 6) are placed and the tree becomes:

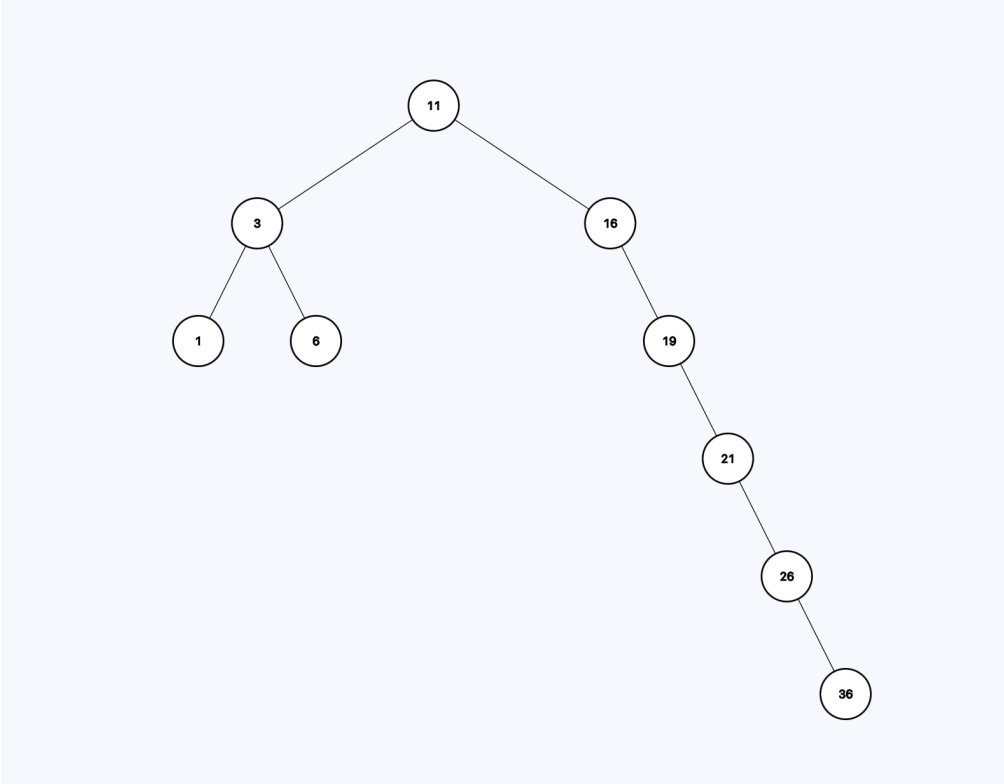
Figure 6: After Initial Left Rotations



Iterative Left Rotations: After the initial rotations, we continue performing **left rotations** on all odd-indexed nodes in the backbone (the righ-most path on the tree). The number of iterations required depends on the height of the tree, calculated as $\lfloor \log_2(n+1) \rfloor$. For 9 nodes, we need a total of 3 rotations in this iteration:

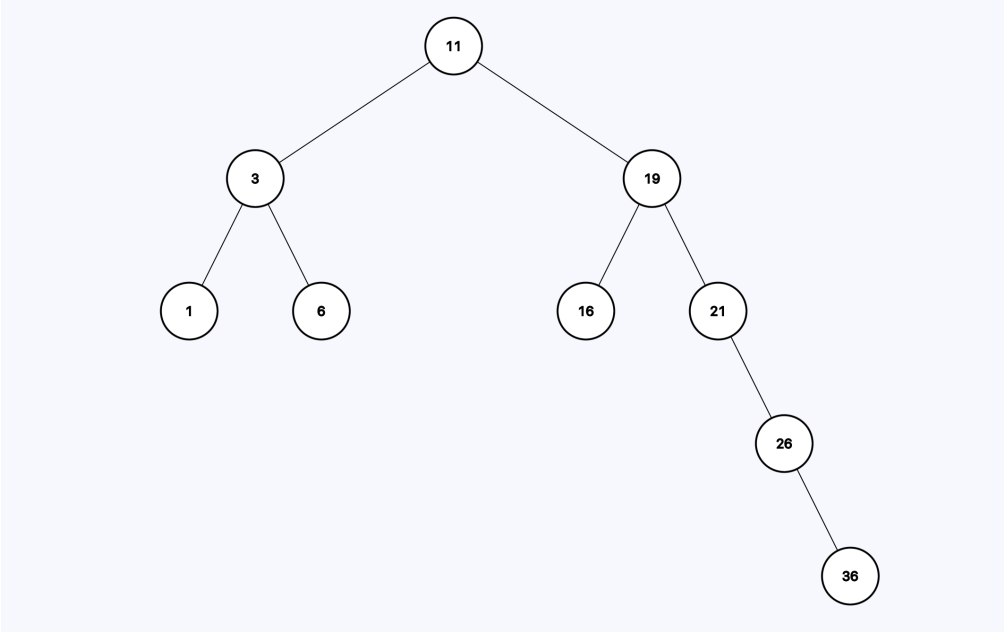
- **First Left Rotation:**

Figure 7: First of Three Left Rotations



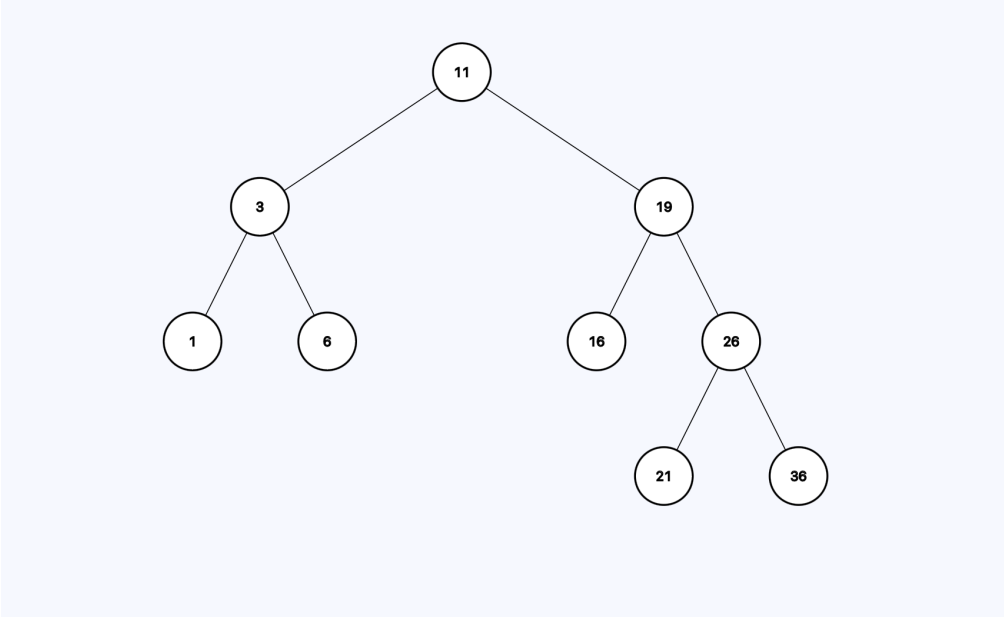
- **Second Left Rotation:**

Figure 8: Second of Three Left Rotations



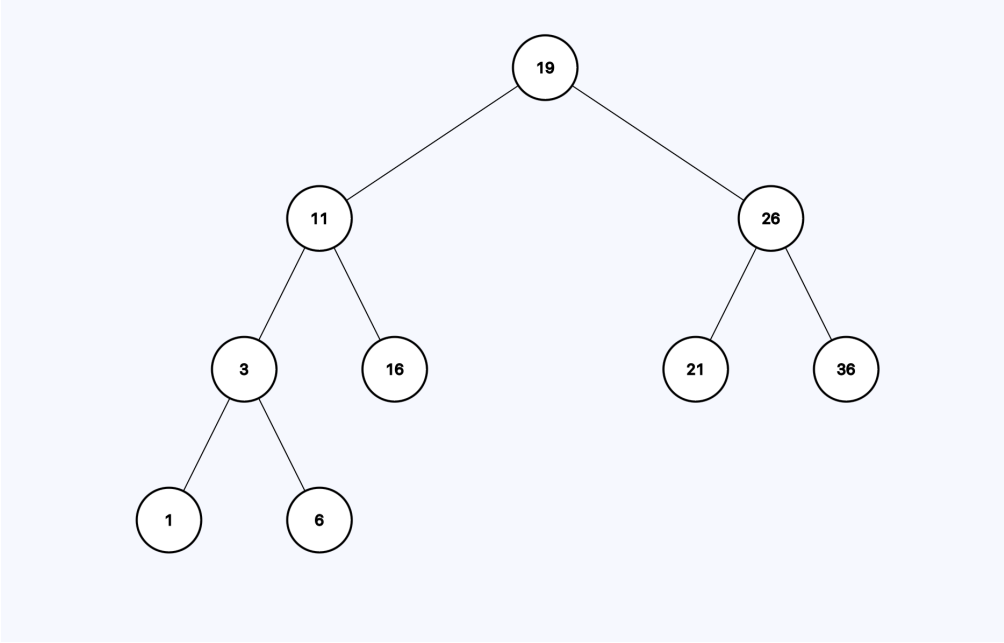
- **Third Left Rotation:**

Figure 9: Third of Three Left Rotations



Final Adjustment: To complete the balancing process, we perform one last **left rotation** (as calculated by $3 \div 2 = 1$):

Figure 10: Final Balanced Tree



The tree is now fully balanced, achieving a height of 4. This concludes the DSW balancing process.