Middle East Technical University | Department of Computer Engineering

CENG 213

Data Structures

Fall 2025

Programming Assignment 2

1 Objectives

In this programming assignment, you will implement a fully generic and self-balancing **Binary Search Tree (BST)** that supports efficient ordered storage, retrieval, and modification of elements. The BST is templated over:

- a value type `T` (commonly `KeyValuePair<K,V>`),
- a **Comparator** defining the ordering between elements,
- a **Balance Condition Functor** that specifies when a subtree should be rebuilt.

Each node maintains structural metadata—**height** and **subtree size**—to support fast balance checking. Whenever the balance condition indicates that a subtree is no longer well-formed, it must be reconstructed into a **minimum-height complete BST**, which is by definition the unique BST shape that achieves the minimum possible height.

This BST forms the underlying data structure for the **TreeMap** Abstract Data Type. The TreeMap uses your BST to provide ordered key–value storage and efficient navigation through the mapping.

Keywords: C++, Templates, Binary Search Trees, Height Balancing, Complete BSTs, Ordered Maps, TreeMap

2 Binary Search Tree with Balancing

The binary search tree used in this assignment is implemented as the class template `BinarySearchTree<T, BalanceCondition, Comparator>` in *BinarySearchTree.h*. It is parameterized by:

- `T` – the element type stored inside each node,
- `BalanceCondition` – a function object that determines whether a subtree is considered balanced,
- `Comparator` – a function object defining the ordering between elements.

The BST stores elements inside dynamically allocated nodes of type `Node<T>`, declared in `Node.h`. Each node stores:

- `T element` – the stored key–value object,
- `Node *left`, `Node *right` – child pointers,
- `size_t height` – height of the subtree rooted at this node,
- `size_t subsize` – total number of nodes in the subtree including itself.

2.1 BinarySearchTree Class

The tree maintains:

- `Node<T>* root` – pointer to the root node,
- `int numNodes` – number of nodes in the BST,
- `Comparator isLessThan` – comparator functor,
- `BalanceCondition isBalancedFunctor` – balance functor.

You must implement the following public interface methods in `BinarySearchTree.h`.

2.1.1 BinarySearchTree();

This is the default constructor. You should initialize:

- `root = nullptr`,
- `numNodes = 0`,
- comparator and balance functor using their default constructors.

This sets up an empty BST ready for insertions.

2.1.2 BinarySearchTree(const std::list<T> &sortedList);

This constructor builds a **minimum-height complete BST** from a sorted list of elements. The input list is already sorted, and **must not be sorted again**.
The resulting BST must:

- be **complete**, with nodes filled from left to right in each level,
- have heights correctly computed for all nodes,
- have subtree sizes correctly computed for all nodes.

2.1.3 ∼BinarySearchTree();

The destructor must deallocate all nodes in the BST. You should recursively delete the entire tree and set:

`root = nullptr`, `numNodes = 0`.

2.1.4 `bool isEmpty() const;`

Returns `true` if the BST is empty, i.e. `root == nullptr`. Otherwise, returns `false`.

2.1.5 `int getSize() const;`

Returns the total number of nodes in the BST. This value is stored in `numNodes`.

2.1.6 `int getHeight() const;`

Returns the height of the root node. If the tree is empty, returns `-1`. Uses the stored `height` field; no recomputation is necessary.

2.1.7 `void toCompleteBST();`

This function restructures the current binary search tree into a **minimum-height complete BST** while keeping the **memory addresses** of all existing nodes unchanged. The transformation must run in linear time with respect to the number of nodes, and it must be performed using *only pointer rewiring*. Therefore, the implementation must **not**

- allocate new nodes,
- delete existing nodes,
- copy or assign stored element values into other nodes.

The objective is to rebuild the tree so that it satisfies the inorder ordering of a BST and exhibits the shape of a complete binary tree: all levels are filled from left to right, except possibly the last level, which is also populated from left to right.

The transformation proceeds in four stages:

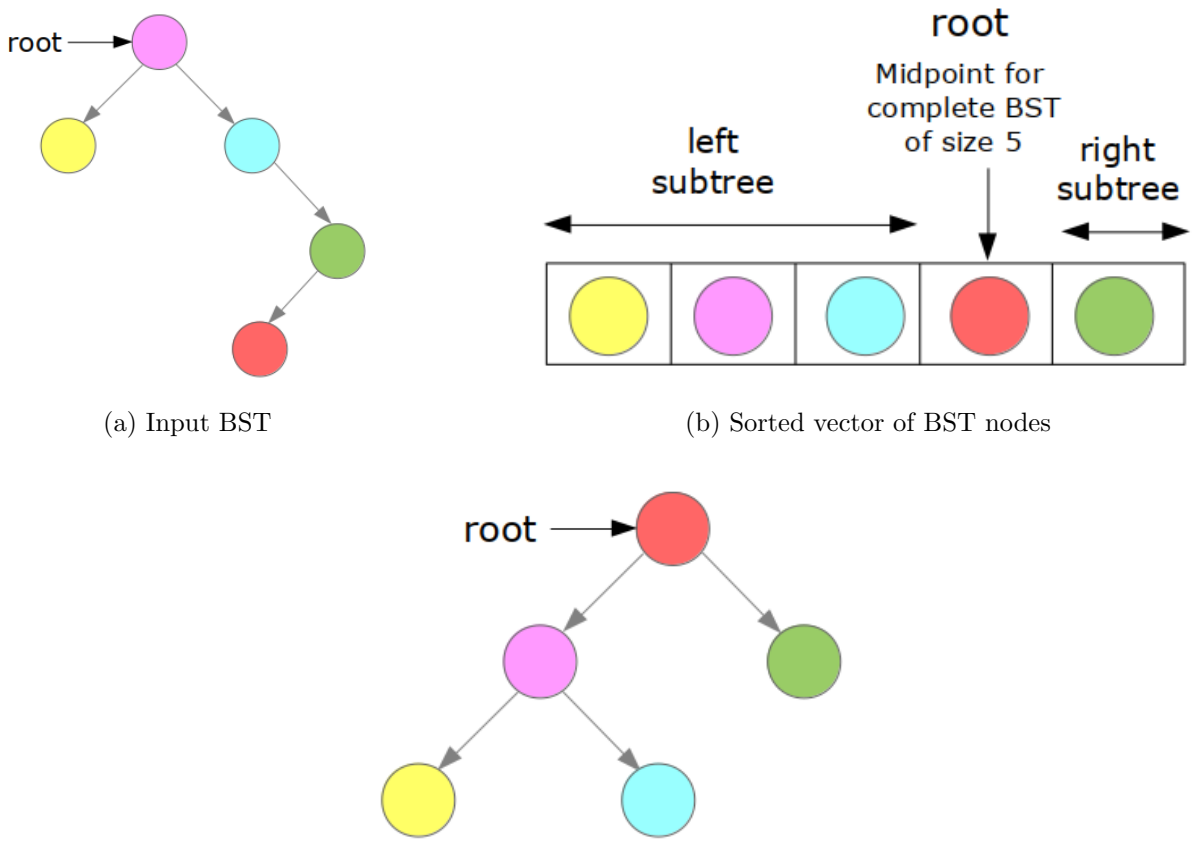
1. **Inorder collection of nodes.** An inorder traversal is performed on the current BST to gather *pointers* to all nodes in strictly sorted order. No new nodes are created; only existing node addresses are recorded.
2. **Determination of the complete-tree root index.** A complete binary tree of size N does not necessarily have its root at the median index. Instead, the function computes the number of nodes that fully populate the left subtree of a complete tree of size N, and selects that index as the root. This ensures that the output tree is populated level-by-level **from left to right**, avoiding skewing.
3. **Recursive reconstruction using pointer rewiring.** Using the computed root index, the algorithm:
   - designates the node at that index as the root,
   - recursively builds the left subtree from the segment $[0, \text{rootIndex} - 1]$,
   - recursively builds the right subtree from the segment $[\text{rootIndex} + 1, N - 1]$.

   During this step, only child pointers are reassigned. The memory addresses and stored elements of all nodes remain unchanged.
4. **Metadata recomputation.** After reconstruction, the algorithm updates all nodes' `height` and `subsize` fields with a postorder traversal. This metadata is essential for balancing decisions and lookup operations.

Upon completion, the tree becomes a canonical complete BST of size N whose inorder traversal matches that of the original tree. The resulting height is minimal, satisfying

$$h = \lfloor \log_2 N \rfloor,$$

and the structure matches the complete-tree form illustrated in Figure 1.
You can implement other algorithms that do not violate the outlined restrictions.



(a) Input BST

(b) Sorted vector of BST nodes

(c) Output complete BST

Figure 1: Conversion of a BST of size 5 into a complete BST

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

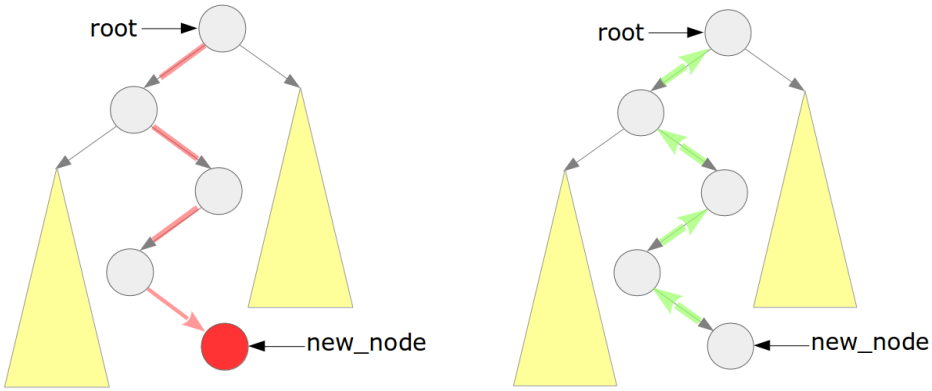
This function inserts a new element into the binary search tree while preserving the BST ordering defined by the `Comparator`. If a node containing an equal key already exists, its stored value is **updated in place**, and the function returns `false` to indicate that no new node was created.
If insertion succeeds (i.e., a new node is allocated), the function returns `true`.

If the key does not already exist, the function performs the following steps:

1. **BST Search.** Starting from the root, the algorithm recursively follows left or right child pointers according to the comparator until a `nullptr` is reached.

2. **Node creation.** A new `Node<T>` object is dynamically allocated at the identified location, and the total node count `numNodes` is incremented.

3. **Backtracking and metadata update.** As recursion unwinds, each ancestor node updates its:
   - stored `height`,
   - stored `subsize`.
4. **Balance checking.** For each visited subtree of size N, its ideal height is computed as
$$H_{\text{ideal}} = \lfloor \log_2(N) \rfloor .$$
   If the `BalanceCondition` functor evaluates to `false` for the pair (actual height, ideal height), the subtree is considered imbalanced.
5. **Complete BST reconstruction.** When rebalancing is required, the subtree is rebuilt into a **minimum-height complete BST** by calling `toCompleteBST()`, which performs inorder collection followed by pointer rewiring. No node allocations, deletions, or value-copies occur during this step.



(a) Search path for insertion (b) Before rebalancing

Figure 2: Insertion procedure when the key does not already exist.

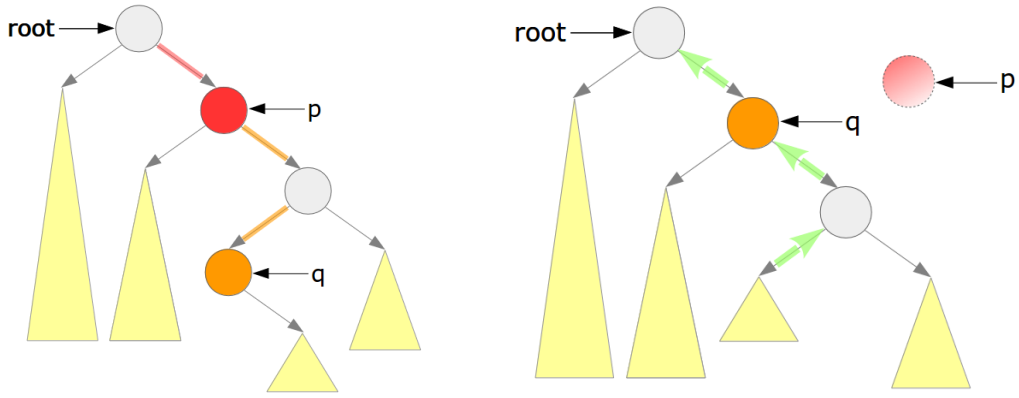
When balancing is not triggered, insertion runs in logarithmic expected time. If subtree reconstruction is needed, the cost becomes linear in the size of the affected subtree.

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

This function removes the node whose key matches the given element. If no such key exists, the function returns `false` and the tree is not modified. If removal succeeds, the function returns `true`. Deletion follows the standard BST cases:

- **Case 1 — Leaf node.** The node is deleted and its parent's pointer becomes `nullptr`.
- **Case 2 — Node with one child.** The node is replaced by its single child and then deleted.
- **Case 3 — Node with two children.** The inorder successor `q` is identified within the right subtree. Instead of copying keys or values, the algorithm **moves the successor node itself** into the removed node's position using pointer rewiring. This approach ensures that all elements remain stored at their original memory addresses.

After removing the node:



(a) Finding the inorder successor (b) After rewiring

Figure 3: Deletion of a node with two children.

- **numNodes is decremented,**
- all ancestor nodes update their `height` and `subsize` values,
- **Balance checking** is performed using

$$H_{\text{ideal}} = \lfloor \log_2(N) \rfloor .$$

  If imbalance is detected by the balance functor, the subtree is rebuilt into a complete BST via `toCompleteBST()`, using inorder collection and pointer rewiring.

As in insertion, removal is logarithmic when no rebalancing occurs, and linear when subtree reconstruction is required.

2.1.10 `std::list<Node<T>*> find(const T &low, const T &high) const;`

This function returns all nodes whose keys lie in the closed interval [`low`, `high`].
The search must use BST ordering to prune irrelevant subtrees.

2.1.11 `const T& get(const T &key) const;`

Searches the BST for a node whose key matches the given `key`. If found, returns a **const reference** to the stored element. If no such node exists, throws `NoSuchItemException`.

2.1.12 `const T& getMin() const;`

Returns the element associated with the **smallest key** in the BST. Traversal proceeds by repeatedly following left child pointers. If the tree is empty, throws `NoSuchItemException`.

2.1.13 `const T& getMax() const;`

Returns the element associated with the **largest key**. Traversal proceeds by repeatedly following right child pointers. Throws `NoSuchItemException` if the BST is empty.

2.1.14 `const T& getNext(const T &key) const;`

Returns the element associated with the **smallest key strictly greater than** `key`. If no such key exists, throws `NoSuchItemException`. Must use BST successor-search logic.

2.1.15 `const T& getCeiling(const T &key) const;`

Returns the **smallest key ≥ given key**. If all keys are smaller than `key`, this function throws `NoSuchItemException`.

2.1.16 `const T& getFloor(const T &key) const;`

Returns the **largest key ≤ given key**. If all keys are larger than `key`, throws `NoSuchItemException`.

2.1.17 `void removeAllNodes();`

Deletes all nodes in the BST so that the structure becomes completely empty. Resets:

`root = nullptr`, `numNodes = 0`.

This operation must recursively deallocate all dynamically allocated nodes.

2.1.18 `BinarySearchTree<T>& operator=(const BinarySearchTree<T> &rhs);`

This overloaded assignment operator must:

1. remove all nodes of the current BST,
2. deep-copy every node of `rhs`,
3. recreate the exact same structure in `*this`.

With these components, you will implement a complete and efficient height-balanced `BinarySearchTree`. It supports ordered key–value storage, successor/predecessor retrieval, and dynamic subtree rebalancing using minimum-height complete BST reconstruction.

3 Tree Map Implementation

Tree map is a `BinarySearchTree`-based map implementation that keeps its entries sorted according to the natural ordering of their keys. Tree maps 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 **does not allow duplicates** (i.e., a tree map cannot have two entries with the same key). Tree map entries are stored as key–value pairs and are always accessed by using the keys.

The tree map in this assignment is implemented as the class template `TreeMap<K, V>`, where template argument `K` is the type of the keys stored in the map and template argument `V` is the type of the values associated with those keys. The `TreeMap` class contains a `BinarySearchTree<KeyValuePair<K, V>>` object in its private data field (namely `stree`). This `BinarySearchTree` object maintains all key–value mappings in sorted order, relying on the key comparisons defined in `KeyValuePair<K, V>`. The `KeyValuePair` class represents the key–value mappings stored in the tree map.

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

3.1 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*.

3.2 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.

3.2.1 `void clear();`

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

3.2.2 `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`.

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

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

3.2.4 `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` BinarySearchTree. 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`.

3.2.5 `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.

3.2.6 `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.

3.2.7 `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`.

3.2.8 `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`.

3.2.9 `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 Contact List Implementation

In this part of the assignment, you will implement a **contact management application** using your `TreeMap` class. Each contact is stored using a **full name** ("Name Surname") as the key, and a **ContactInfo** structure as the associated value. Your task is to support insertion, deletion, update, lookup, and alphabetical traversal of contact entries using the same ordered BST implementation developed in earlier sections.
A contact list behaves as a dictionary in which each key (full name) is unique, and the information associated with that name is stored in a structured value containing phone numbers, an email address, a company field, and optional notes.

4.1 ContactInfo Struct

Each entry in the contact list stores a value of type `ContactInfo`. The structure is defined as follows:

```
1 struct ContactInfo {
2     std::vector<std::string> phoneNumbers; // can store multiple phones
3     std::string email;
4     std::string company;
5     std::string notes; // optional description field
6 };
```

- A contact may have **multiple phone numbers**. These must be stored in the exact order provided.
- Email, company name, and a free-form notes field are stored as plain strings.
- No STL algorithms such as `std::sort` or `std::find` may be used.

4.2 ContactList Class

The contact list is implemented across the files *ContactList.h* and *ContactList.cpp*. The class contains a single `TreeMap` member defined as:

```
TreeMap<std::string, ContactInfo> contacts;
```

Here,

- the key type is `std::string`, representing a person's full name,
- the value type is `ContactInfo`.

All operations of `ContactList` must be implemented using this underlying `TreeMap` instance.

4.3 `ContactList Class | Member Functions`

4.3.1 `void addContact(const std::string& fullName, const ContactInfo& info);`

Adds a new contact or updates an existing one.

- If `fullName` does not exist in the map, a new entry is inserted.
- If the contact already exists, its stored `ContactInfo` is fully replaced by the new value.

Internally, this function must call the `put` method of the `TreeMap`.

4.3.2 `bool deleteContact(const std::string& fullName);`

Removes the contact with the given name.

- Returns `true` if the contact existed and was removed.
- Returns `false` if no such key exists.

This operation must use the underlying `deletekey` function of the map.

4.3.3 `const ContactInfo& getContact(const std::string& fullName) const;`

Looks up and returns the stored `ContactInfo` for the given name.

- If the key exists, a **const reference** to the stored value is returned.
- If the key does not exist, `NoSuchItemException` must be thrown.

4.3.4 `bool contains(const std::string& fullName) const;`

Returns `true` if the name exists in the contact list; otherwise returns `false`.

4.3.5 `void updateContact(const std::string& fullName, const ContactInfo& newInfo);`

If the contact exists, replaces its stored `ContactInfo` with `newInfo`. If no such key exists, this function does nothing (or may throw, depending on your design — state it explicitly).

4.3.6 `void iterateAlphabetically() const;`

Prints all contacts in **ascending alphabetical order of full names**. This function traverses the entries of the underlying `TreeMap` from lowest key to highest key and outputs each contact. The traversal is **read-only** and must not modify the contact list.

4.4 Notes and Restrictions

- Keys must be stored exactly as provided (case-sensitive).
- Phone numbers must remain in the order inserted; no sorting is allowed.
- No STL algorithms other than `std::vector` operations may be used.
- You must not change the definition of `TreeMap` or `KeyValuePair`.
- All structural modifications (insert, delete, update) must go through the `TreeMap` object.

This completes the contact list component of your assignment. The structure serves as an example of applying your self-balancing BST to a real-world data management scenario.