Middle East Technical University | Department of Computer Engineering

CENG 213

Data Structures

Fall 2023

Programming Assignment 2

Due date: November 27, 2023, Monday, 23:55

1 Objectives

In this assignment, the objective is to implement a specialized dictionary utilizing a unique data structure for managing user-specific data on watched films. The primary structure of the dictionary will be a trie comprising binary trees. Specifically, the trie will be responsible for managing user IDs (keys), while the watched movies associated with each user will be stored as Binary Search Trees (BSTs) serving as the corresponding values. Each node denoting a user ID in the Trie will point to a uniquely associated Binary Search Tree as the set of movies watched by that user (value).

A Trie, short for retrieval tree or digital tree, is a tree-like data structure that is used to store a dynamic set of keys, typically strings. Each node in the trie represents a single character of a key, and the paths from the root to the leaf nodes represent the keys themselves. The key associated with a node is obtained by concatenating the characters along the path from the root to that node. Tries are particularly useful for efficient search operations, as the time complexity for searching, inserting, and deleting keys is often proportional to the length of the key, making them well-suited for tasks such as autocomplete features or dictionary implementations.

The implementation follows a dual-phase approach: first, the trie manages and organizes user IDs, and second, the binary search trees associated with each trie node store the watched movies for the respective users. This design facilitates the efficient organization and retrieval of watched films for each user.

2 Specialized Two-Phase structure Implementation

In the first phase of this assignment, the Trie data structure is implemented as a class template named 'Trie'. The template takes an argument 'T', representing the type of data stored in the Trie nodes. In this assignment, TrieNode data is user objects, meaning that the TrieNode class template is designed to hold and manage user-defined types. Each TrieNode contains an instance of the user-defined type, referred to as 'Data,' and serves as the foundational structure for organizing and efficiently locating specific users within the Trie.
The second phase involves the implementation of a Binary Search Tree (BST), which is realized as the class template 'BST'. Similar to the Trie, the template argument 'T' is used to specify the type of data stored in the nodes of the BST. This Binary Search Tree is associated with each Trie node,

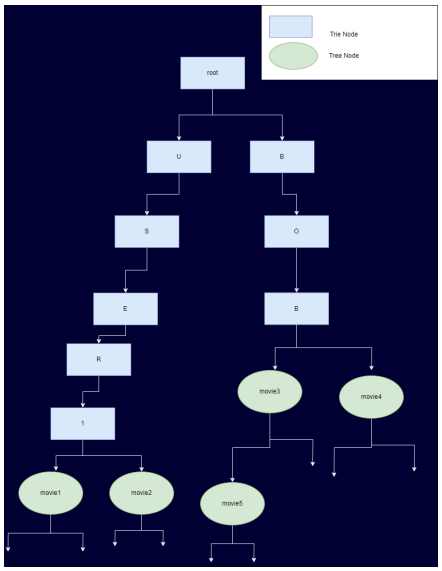

Figure 1: Specialized Two-Phase structure

and it facilitates the efficient organization and retrieval of watched films for individual users.

The combination of these two phases, where Trie handles user-specific organization and BST manages movie data for each user, provides a specialized two-phase structure. This structure optimizes the organization of the watched films database, enabling quick and efficient access to individual users' movie data based on their viewing history.

2.1 `Trie Class`

2.1.1 `TrieNode`

This is a nested structure representing a node in the trie. It contains an array of children, a character (`keyChar`), a flag to indicate the end of a key (`isEndOfKey`), and a pointer to associated data (`data`). Since userIDs will be consisting of alphanumeric characters, the node structure will allow for up to 128 children to accommodate the range of possible characters.

2.1.2 `Trie();`

The Trie class constructor initializes a new trie with a root node, using the null character ('\0') as the initial key character. This sets the foundation for the trie data structure, enabling the insertion of keys and the construction of the trie hierarchy.

2.1.3 `~Trie();`

The destructor for the Trie class is responsible for freeing the memory allocated for the trie nodes. It calls the private member function `deleteTrieNode(root)` to recursively traverse the trie and delete each node. This ensures proper memory cleanup, preventing memory leaks associated with the trie data structure.

2.1.4 `void findStartingWith(std::string prefix, std::vector<T*> &results);`

The findStartingWith method in the Trie class initiates a search for keys in the trie that start with a specified prefix. Starting from the root, it populates a vector, 'results', with the keys matching the

given prefix. This method provides an efficient way to retrieve keys that share a common starting sequence.

2.1.5 `void wildcardSearch(const std::string &wildcardKey, std::vector<T*> &results);`

The ‘wildcardSearch‘ method in the ‘Trie‘ class facilitates a search for keys in the trie that match a specified wildcard pattern. Starting from the root, it populates a vector ‘results‘ with keys that satisfy the wildcard condition specified by ‘wildcardKey‘. The wildcard pattern can contain two types of wildcards: ’?’ and ’*’. The ’?’ wildcard matches any single character in the input string, while the ’*’ wildcard matches zero or more characters. Some examples of wildcard search key are given below.

1. Wildcard Pattern: *abc*
   (a) This pattern matches any key that contains ”abc” as a substring with any characters before and after it.
   (b) Example Result: ”xabcz”, ”abc123”, ”defabcghi”
2. Wildcard Pattern: a*c?
   (a) This pattern matches any key that starts with ”a”, followed by any character, end with any single character .
   (b) Example Result: ”abc1”, ”axcf”, ”act”
3. Wildcard Pattern: a?c*
   (a) This pattern matches any key that starts with ”a”, followed by any single character, and has zero or more characters afterward.
   (b) Example Result: ”abc”, ”abcmnop”

2.1.6 `T* search(std::string username);`

The ‘search‘ method in the ‘Trie‘ class looks for a specific key, such as a username, in the trie. It traverses the trie based on the characters of the input key, returning a pointer to the associated data if the key is found. If the key is not present or the trie is empty, it returns a ‘NULL‘ pointer.

2.1.7 `Trie &insert(const string &username);`

The insert method in the Trie class allows for the insertion of a new key (e.g., a username) into the trie. The method returns a reference to the modified trie, enabling method chaining for consecutive operations on the same trie instance.

2.1.8 `void remove(std::string username);`

The remove method in the Trie class is designed to remove a specific key, such as a username, from the trie. It traverses the trie based on the characters of the input key and, if the key is found, marks the corresponding node as not the end of a key. This approach effectively removes the key from the trie without deleting the node itself. If the key is not present in the trie, the method does nothing.

2.1.9 `void print(const std::string &primaryKey);`

The ‘print‘ method in the ‘Trie‘ class serves to print either the entire trie or specific keys based on the provided primary key. If the primary key is empty, indicating a request to print the entire tree, it invokes the ‘printTrie‘ function to recursively print the trie starting from the root. If a specific primary key is provided, it traverses the trie to the corresponding node, printing the subtree rooted at that node. If the primary key is not found in the trie, it prints an empty set of braces.

2.1.10 `void deleteTrieNode(TrieNode* node);`

The deleteTrieNode method in the Trie class is a recursive function that ensures proper memory cleanup for a trie node and its descendants. It takes a pointer to a TrieNode and recursively deletes child nodes, associated data, and the node itself, preventing memory leaks during trie operations.

2.2 BST Class

2.2.1 `TreeNode`

The code defines a ‘TreeNode‘ structure for a binary search tree (BST). Each node in the tree holds a key of type ‘std::string‘, associated data of type ‘T‘, and pointers to its left and right child nodes. The constructor initializes the node with the provided key and data, setting the child pointers to ‘NULL‘. This structure is fundamental for organizing and managing data within a binary search tree.

2.2.2 `BST();`

The code declares a default constructor for the Binary Search Tree (BST) class, initializing a new instance of the class.

2.2.3 `~BST();`

The code declares a destructor for the Binary Search Tree (BST) class, responsible for resource cleanup when an instance is no longer needed.

2.2.4 `TreeNode* getRoot();`

The code declares a method getRoot() in the Binary Search Tree (BST) class, which returns a pointer to the root node of the tree.

2.2.5 `bool isEmpty();`

The code introduces a method isEmpty() in the Binary Search Tree (BST) class, evaluating whether the tree is empty by checking if the root node is NULL.

2.2.6 `BST& insert(const std::string key, const T& value);`

The code implements the ‘insert‘ method in the Binary Search Tree (BST) class. This method adds a new node and takes two parameters: key and value. The key represents the unique identifier associated with the movie name, and the value is the actual movie data. If the tree is empty, it becomes the root; otherwise, it traverses the tree to find the appropriate position for the new node based on the key. The method returns a reference to the modified BST.

2.2.7 `bool search(std::string key) const;`

The code presents a ‘search‘ method in the Binary Search Tree (BST) class. This method looks for a node with a specified key in the tree, starting from the root. The method returns true if a match is found and false otherwise.

2.2.8 `void remove(std::string key);`

The code introduces a ‘remove‘ method in the Binary Search Tree (BST) class. This method removes a node with a specified key from the tree. It updates the root by calling a recursive ‘remove‘ function, ensuring the binary search tree structure is maintained after the removal.

2.2.9 `BST<T>* merge(BST<T>* bst);`

The code introduces a ‘merge‘ method in the Binary Search Tree (BST) class. This method merges nodes from two BSTs into a new BST. It takes into account the possibility that either or both of the input BSTs can be empty or identical. It utilizes vectors obtained from the ‘tree2vector‘ method to compare node keys between the two BSTs. It creates a new BST, converts nodes from both input BSTs into vectors, and merges them in a sorted order. The merged nodes are inserted into the new BST, which is then returned.

2.2.10 `BST<T>* intersection(BST<T>* bst);`

The ‘intersection‘ method in the Binary Search Tree (BST) class creates a new BST representing the intersection of two existing BSTs. It takes into account the possibility that either or both of the input BSTs can be empty or identical. It utilizes vectors obtained from the ‘tree2vector‘ method to compare node keys between the two BSTs. Matching keys are added to the new BST, and the method continues by incrementing indices based on key comparisons. The resulting BST, containing nodes common to both input BSTs, is returned.

2.2.11 `std::vector<TreeNode> tree2vector(TreeNode* root);`

The ‘tree2vector‘ function in the Binary Search Tree (BST) class converts the nodes of a BST into a vector. It takes a pointer to the root of the BST, initializes an empty vector, and then recursively traverses the BST to populate the vector with the nodes. The resulting vector is returned, and this function is utilized in the ‘merge‘ and ‘intersection‘ methods for further processing.

2.2.12 `void BST<T>::print()`

Using recursion, the function displays the tree nodes hierarchically, indicating their values and relationships with appropriate symbols.

2.2.13 `void print(TreeNode* node, std::string indent, bool last, bool isLeftChild);`

This function, when given a TreeNode pointer and specific parameters, prints the key of the current node and recursively calls itself for the left and right children. This process creates a hierarchical representation of the BST structure, facilitating visualization of its organization.

2.2.14 `void tree2vector(TreeNode* node, vector<TreeNode>& result);`

The ‘tree2vector’ function in the Binary Search Tree (BST) class traverses the tree in an in-order fashion, adding each node to a vector. It ensures a deep copy by pushing back a copy of each node into the result vector, ultimately creating a linear representation of the tree.

2.3 Movie Class

2.3.1 Movie();

The 'Movie()' constructor is a default constructor for the 'Movie' class. It initializes a 'Movie' object with default values: an empty string for 'movieName', 0 for 'year', and 0.0 for 'rating'. This constructor is used when an object of the 'Movie' class is created without providing explicit values.

2.3.2 Movie(std::string movieName, int year, float rating);

The 'Movie(std::string movieName, int year, float rating)' constructor initializes a 'Movie' object with the provided values for 'movieName', 'year', and 'rating'.

2.3.3 Movie(const Movie& movie) : movieName(movie.getMovieName()), year(movie.getYear(rating(movie.getRating())

The copy constructor 'Movie(const Movie& movie)' initializes a new 'Movie' object by copying the values of 'movieName', 'year', and 'rating' from an existing 'Movie' object ('movie').

2.3.4 std::string getMovieName() const;

The 'getMovieName' function is a getter method in the 'Movie' class, and it returns the 'movieName' attribute of the 'Movie' object. The 'const' qualifier indicates that the function does not modify the state of the object.

2.3.5 int getYear() const;

This member function returns the year of the movie.

2.3.6 float getRating() const;

This member function returns the rating of the movie.

2.3.7 bool operator==(const Movie &obj);

The 'operator==' function in the 'Movie' class checks if two 'Movie' objects are equal by comparing their 'movieName', 'year', and 'rating' attributes. If all attributes are the same, the function returns 'true'; otherwise, it returns 'false'.

2.4 User Class

2.4.1 User();

The constructor initializes a 'User' object with an empty string for the username and an empty 'BST' for movies.

2.4.2 User(std::string username);

This constructor initializes a 'User' object with the provided 'username' and an empty 'BST' for movies.

2.4.3 std::string getUsername();

This member function returns the username of the 'User' object.

2.4.4 `BST<Movie> *getMovies();`

'getMovies' returns a pointer to the private member variable 'movies', which is a binary search tree (BST). It provides controlled access to the movie data within the class.

2.4.5 `void addMovie(std::string movieName, Movie movie);`

This member function adds a movie to the user's collection by inserting it into the binary search tree (`'BST<Movie>'`) using the 'insert' function.

2.4.6 `void removeMovie(Movie movie);`

This member function removes a movie from the user's collection by calling the 'remove' function on the binary search tree (`'BST<Movie>'`) and providing the movie name as the key.

2.4.7 `void searchMovie(Movie movie);`

This member function searches for a movie in the user's collection by calling the 'search' function on the binary search tree (`'BST<Movie>'`) and providing the movie name as the key.

2.4.8 `void printMovies();`

This member function prints the movies in the user's collection by calling the 'print' function on the binary search tree (`'BST<Movie>'`).

2.4.9 `BST<Movie>* merge(BST<Movie> bst);`

This member function returns the result of merging the movies of the current user with the movies from another binary search tree ('bst'). It calls the 'merge' function on the binary search tree (`'BST<Movie>'`) of the user's movies. The merged result is returned as a new binary search tree.

2.4.10 `BST<Movie>* intersection(BST<Movie> bst);`

The 'intersection' function in the 'User' class calculates the intersection of movies between the current user and another binary search tree ('bst'). It returns a new binary search tree containing the common movie data.

2.4.11 `friend std::ostream& operator<<(std::ostream& os, const User& user);`

The `'operator<<'` overload for the 'User' class enables the streaming of 'User' objects to an output stream, such as 'std::cout'. It prints the username of the user object.

2.5 UserInterface

2.5.1 `void addUser(std::string username);`

The 'addUser' method inserts a new user with the given username into the trie data structure in the 'UserInterface' class.

2.5.2 `void removeUser(std::string username);`

The removeUser function in a UserInterface class removes a user with a given username from a collection (users). It checks if the user exists, and if so, removes it. If the user object is dynamically allocated, it deletes it to avoid memory leaks.

2.5.3 `void addWatchedMovie(std::string username, Movie movie);`

The 'addWatchedMovie' method adds a watched movie to a user identified by the given 'username'. It first attempts to find the user using the 'findUser' method. If the user is found, it prints the username and then adds the movie using the 'addMovie' method of the user. If the user is not found (returns 'NULL'), it prints "NULL" to the console.

2.5.4 `void removeWatchedMovie(std::string username, Movie movie);`

The 'removeWatchedMovie' function in the 'UserInterface' class removes a specified movie from a user's watched list.

2.5.5 `User *findUser(std::string username);`

The 'findUser' function in the 'UserInterface' class locates and returns a 'User' object based on a specified username from the 'users' container. It utilizes the container's 'search' method, providing a clean and modular interface. The function returns a pointer to the user if found, or 'nullptr' if the user does not exist.

2.5.6 `BST<Movie> *getWatchedMovies(std::string username);`

The 'getWatchedMovies' function in the 'UserInterface' class returns a binary search tree (BST) containing the movies watched by a user identified by the provided 'username'. It uses the 'findUser' method to locate the user and then retrieves the watched movies using the 'getMovies' method.

2.5.7 `BST<Movie> *mergeWatchedMovies(std::string username1, std::string username2);`

The 'getWatchedMovies' function in the 'UserInterface' class returns a binary search tree (BST) containing the movies watched by a user identified by the provided 'username'. It uses the 'findUser' method to locate the user and then retrieves the watched movies using the 'getMovies' method.

2.5.8 `BST<Movie> *intersectionWatchedMovies(std::string username1, std::string username2);`

'intersectionWatchedMovies' in 'UserInterface' finds common watched movies between two users using the 'intersection' method. It retrieves user objects with 'findUser' and returns a BST representing shared movie preferences.

2.5.9 `std::vector<User> findUsersStartingWith(std::string prefix);`

'findUsersStartingWith' in 'UserInterface' searches for users with usernames starting with a specified prefix. It populates a vector of 'User' pointers ('foundUsers') and prints the count and details of the matches.The wildcard pattern can contain two types of wildcards: '?' and '*'. The '?' wildcard matches any single character in the input string, while the '*' wildcard matches zero or more characters.

2.5.10 `std::vector<User> findUsersContains(std::string infix);`

'findUsersContains' in 'UserInterface' locates users with usernames containing a specified infix. It uses the 'wildcardSearch' method, populates a vector ('foundUsersContains'), and prints count and details.The wildcard pattern can contain two types of wildcards: '?' and ''. The '?' wildcard matches any single character in the input string, while the '' wildcard matches zero or more characters.

2.5.11 `std::vector<User> findUsersEndingWith(std::string suffix);`

'findUsersEndingWith' in 'UserInterface' locates users with usernames ending in a specified suffix. It uses the 'wildcardSearch' method, populates a vector ('foundUsersEndingWith'), and prints count and details.The wildcard pattern can contain two types of wildcards: '?' and '*'. The '?' wildcard matches any single character in the input string, while the '*' wildcard matches zero or more characters.

2.5.12 `void printUsers();`

'printUsers' in 'UserInterface' prints the details of all users managed by the 'users' container.

2.5.13 `void printWatchedMovies(std::string username);`

'printWatchedMovies' in 'UserInterface' displays the watched movies of a specific user identified by their username. It locates the user with 'findUser' and prints the movie details using 'printMovies'. Useful for obtaining a quick overview of a user's movie-watching history within the system.

3 Regulations

1. **Programming Language:** You will use C++.
2. **S**tandard Template Library is allowed only for `list`,`queue`, and `stack`.
3. **E**xternal libraries other than those already included are not allowed.
4. **T**hose who do search, update, remove operations without utilizing the tree will receive 0 grade.
5. **T**hose who modify already implemented functions and those who insert other data variables or public functions and those who change the prototype of given functions will receive 0 grade.
6. **T**hose who use STL `vector` or compile-time arrays or variable-size arrays (not existing in ANSI C++) will receive 0 grade. Options used for `g++` are `-ansi -Wall -pedantic-errors -O0`.
7. **Y**ou can add private member functions whenever it is explicitly allowed.
8. **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 11:30 pm, you could use 2 late days to submit on Saturday by 11:30 pm 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

   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
9. **Cheating:** We have 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.

10. **R**emember that students of this course are bounded to code of honor and its violation is subject to severe punishment.

11. **Newsgroup:** You must follow ODTUClass for discussions and possible updates on daily basis.

4 Submission

- Submission will be done via Moodle.
- Do not write a *main* function in any of your source files.
- A test environment will be ready in CengClass.
  - You can submit your source files to CengClass and test your work with a subset of evaluation inputs and outputs.
  - Additional test cases will be used for evaluation of your final grade, and only the last submission before the deadline will be graded.
  - Only the last submission before the deadline will be graded.