Middle East Technical University   Department of Computer Engineering

CENG 213

Data Structures

Fall 2024

Programming Assignment 3

Due date: December 29, 2024, Sunday, 23:55

1 Objectives

In this programming assignment, you will implement a basic framework for a social media application and simulate information diffusion on such an application through Breadth First Search (BFS) and Depth First Search (DFS). To achieve this, you will implement a hash table with linear probing to keep user information and keep track of who follows whom to get a directed graph. You are also expected to implement several utility functions related to the social media application such as creating an adjacency matrix corresponding to the current state of the application. In addition to everything, you are expected to manage exceptions for invalid inputs.

Keywords: C++, Data Structures, Hash Table, Linear Probing, Directed Graph, Information Diffusion

2 Overview of Data Structures and Algorithms

2.1 Hash Table with Linear Probing

A **hash table** is a data structure that maps keys to indices using a **hash function**. For our purposes, usernames of users are the keys. The hash function computes an initial index for each key, ensuring the index lies within the valid range of the table.

When inserting an element:

- The hash function calculates the index of the key.
- If the calculated index is already occupied, **linear probing** is used: the table is searched sequentially (wrapping around if necessary) until an empty slot is found.

As the table becomes fuller, the **linear probing** causes longer search times for both insertions and lookups. To maintain efficiency, the **load factor** (the ratio of occupied slots to the table's size) is kept below a threshold (e.g., 0.7). Before adding a new element, the load factor is checked. If it exceeds the threshold:

- The table is resized to a new size, typically computed as $\texttt{new\_size} = 2 \cdot \texttt{old\_size} + 1$.
- All existing elements are **rehashed** and reinserted into the resized table, as the hash function depends on the table's size.

When removing an element, it is replaced with a **dummy element** instead of being deleted completely. This ensures that subsequent searches or insertions can proceed correctly, as probing may rely on elements further along in the table. Dummy elements indicate that the slot is available for new insertions but may not mark the end of a search sequence.

In summary, while hash tables with linear probing are efficient for many use cases, their performance degrades as the table becomes full, making careful load factor management and periodic resizing essential.

2.2 Directed Graph

A **directed graph**, or **digraph**, is a collection of **vertices** (or nodes) connected by **directed edges** (or arcs). Each directed edge has an associated direction, represented as an ordered pair of vertices (u, v), indicating that the edge points from vertex u (the **source**) to vertex v (the **destination**).

A directed graph can be represented using an **adjacency list** or an **adjacency matrix**.

- **Adjacency List:** Stores a list of neighbors for each vertex, indicating all outgoing edges.
- **Adjacency Matrix:** Uses a $V \times V$ matrix where $A[i][j] = 1$ if there is a directed edge from vertex i to vertex j, and 0 otherwise.

We will use the adjacency list representation, each outgoing edge being stored within each user's data. But you will expected to produce an adjacency matrix representation.

2.2.1 Breadth-First Search (BFS)

- BFS explores the graph layer by layer, starting from a given **source vertex**.
- It uses a **queue** data structure to keep track of vertices to visit.

Algorithm 1 Breadth-First Search (BFS)

```
Require: Graph $G$, Source vertex $s$
Ensure: Distances from $s$ to all other vertices
1: Initialize a queue $Q$ and mark $s$ as visited
2: Enqueue $s$ into $Q$
3: while $Q$ is not empty do
4:     Dequeue a vertex $v$ from $Q$
5:     for each unvisited neighbor $u$ of $v$ do
6:         Mark $u$ as visited
7:         Enqueue $u$ into $Q$
8:     end for
9: end while
```

Important Note: You must always add vertices in the order they are stored in the adjacency list (their order in the follows field of the `UserNode` class).

2.2.2 Depth-First Search (DFS)

- DFS explores as far as possible along each branch before backtracking.
- It uses a **stack** (explicit or via recursion) to manage vertices.

As previously mentioned, you can also call an helper function recursively instead of using a stack.

Important Note: You must always visit vertices in the order they are stored in the adjacency list (their order in the follows field of the `UserNode` class).

Algorithm 2 Depth-First Search (DFS)

```
Require: Graph $G$, Source vertex $s$
Ensure: Distances from $s$ to all other vertices
 1: Create an empty stack $S$, and mark $s$ as visited
 2: Push $s$ onto stack $S$
 3: while stack $S$ is not empty do
 4:     Pop a vertex $v$ from $S$
 5:     if $v$ is not visited then
 6:         Mark $v$ as visited
 7:         for each unvisited neighbor $u$ of $v$ do
 8:             Push $u$ onto stack $S$
 9:         end for
10:     end if
11: end while
12: return visited
```

3 Social Media Application

3.1 Exceptions

Exceptions are defined in `Exceptions.h`.

3.1.1 `UsernameTakenException(const string& username);`

Already Implemented.
This exception should be thrown when a new user tries to get a username that is already assigned to an existing user.

3.1.2 `UserDoesNotExistException(const string& username);`

Already implemented.
This exception should be thrown whenever an action that includes a user that does not exist is attempted to be performed.

3.1.3 `RedundantActionException(const string& username1, const string& username2, const bool& follow);`

Already implemented.
This exception should be thrown whenever a redundant follow or unfollow action is attempted to be performed, `const bool& follow` signifying whether it is a follow action or not.

3.2 `UserNode Class`

Already implemented.
`UserNode Class` is located in `UserNode.h` and it contains the details of a user, its username (`string username`) and whichever other users it already follows (`vector<string> follows`).

3.3 Utility Functions

Utility functions are located in `Utility.h`. `Utility.h` also contains necessary `#include` statements and `using` statements.

3.3.1 `inline string printStyle(const string& s, size_t max_length)`

Already implemented.
This function is used in `printDatabase` function, which is also already implemented, thus you have no reason to change or use this function.

3.3.2 `inline size_t hashFunction(const string& username, size_t capacity)`

Already implemented.
Hash function you should be using when you are implementing the hash table.

3.4 `UserDatabase Class`

Hold the rest of the variables required for the social media application. Its declerations are located in `UserDatabase.h` and it should be implemented in `UserDatabase.cpp`.

3.4.1 Variables

This variables are already exist inside the class, but you are responsible with initializing and clearing them.

- `vector<UserNode*> _userTable`: The hash table in which the user information is held.
- `size_t _size`: How many users exist in the application.
- `double _load_factor`: Load factor of the hash table.
- `UserNode* _dummy`: A dummy `UserNode` pointer to replace removed entries in the hash table.

3.4.2 `inline void printDatabase() const;`

Already implemented.
Prints the hash table to standard output.

3.4.3 `inline void printUserList(bool withCount) const;`

Already implemented.
Prints the list of user in the order of their location in the hash table to the standard output. Requires `getUserList` function to be implemented when `withCount` is `false`, requires `getUserList`, `userFollowerCount`, and `UserFollowsCount` to be implemented when `withCount` is `true`.

3.4.4 `UserDatabase();`

Default constructor with a load factor of 0.75 and hash table size of 11.

3.4.5 `UserDatabase(size_t s, double load);`

Constructor with a load factor of `load` and hash table size of `s`.

3.4.6 `UserDatabase();`

Destructor of the class, it should deallocate all of the memory that is previously allocated.

3.4.7 `size_t size() const;`

Returns the number of users in the application.

3.4.8 `size_t capacity() const;`

Return the current capacity of the hash table.

3.4.9 `bool isEmpty() const;`

Returns `true` is there is no users, returns `false` otherwise.

3.4.10 `void addUser(string username);`

Checks the current load of the hash table against the `_load_factor`, and resize the table to its current capacity times 2 plus 1 if required. Then add the user to the hash table. Should throw `UsernameTakenException` instead if the `username` was already taken by another user.

3.4.11 `void removeUser(string username);`

Removes the user with the given username. You should also remove the user from the follows list of all other users. Throws a `UserDoesNotExistException` if no such user exists.

3.4.12 `UserNode* getUser(string username) const;`

Returns the pointer to the user with the given username. Throws a `UserDoesNotExistException` if no such user exists.

3.4.13 `void resize(size_t size);`

Resizes the hash table to `size`, and rehashes the contents of the hash table.

3.4.14 `void follow(string username1, string username2);`

Adds `username2` to the bottom of the follow list of the user with the username `username1`. Throws a `UserDoesNotExistException` instead if either of the users does not exists. Throws a `RedundantActionException` with `follow=true` instead it user with the username `username1` already follows the user with the username `username2`.

3.4.15 `void unfollow(string username1, string username2);`

Removes `username2` from the follow list of the user with the username `username1`. Throws a `UserDoesNotExistException` instead if either of the users does not exist. Throws a `RedundantActionException` with `follow=false` instead it user with the username `username1` already does not follow the user with the username `username2`.

3.4.16 `bool userExists(string username) const;`

Returns `true` is the user with the given username exists in the hash table, returns `false` otherwise.

3.4.17 `bool userFollows(string username1, string username2) const;`

Return `true` if the user with the username `username1` follows the user with the username `username2`, returns `false` otherwise. Throws a `UserDoesNotExistException` instead if either of the users does not exist.

3.4.18 `size_t userFollowerCount(string username) const;`

Returns the number of users that follows the user with the given username. Throws a `UserDoesNotExistException` if no such user exists.

3.4.19 `size_t userFollowsCount(string username) const;`

Returns the number of users the user with the given username follows. Throws a `UserDoesNotExistException` if no such user exists.

3.4.20 `vector<string> getUserList() const;`

Returns the list of user in the order of their location in the hash table.

3.4.21 `void printAdjacencyMatrix() const;`

Prints the adjacency matrix of the users. Users are ordered by their location in the hash table.

3.4.22 `int BFS(string username1, string username2, bool printPath);`

Returns the distance between the user with the usernames `username1` and `username2`, starting from the first one, using the BFS algorithm. You must stop searching as soon as the user with `username2` is found. You must traverse users in the order they are added to the follow list of a user. If the user with the username `username2` is not accessable when you start searching from the user with the username `username1`, you must return -1. You should print each user you visit if `printPath=true`. Throws a `UserDoesNotExistException` instead if either of the users does not exist.

3.4.23 `int DFS(string username1, string username2, bool printPath);`

Returns the distance between the user with the usernames `username1` and `username2`, starting from the first one, using the DFS algorithm. You must stop searching as soon as the user with `username2` is found. You must traverse the users in the order in which they are added to the following list of a user. If user with the username `username2` is not accessable when you start searching from the user with the username `username1`, you must return -1. You should print each user you visit if `printPath=true`. Throws a `UserDoesNotExistException` instead if either of the users does not exist.

3.4.24 `double averageBFS();`

Return the average distance between two users when using the BFS algorithm. To obtain this, you should calculate the BFS distance between all users (not including the distance from a user to itself) and return their average, excluding the cases where there is no path between two users.

3.4.25 `double averageDFS();`

Return the average distance between two users when using the DFS algorithm. To obtain this, you should calculate the DFS distance between all users (not including the distance from a user to itself) and return their average, excluding the cases where there is no path between two users.

3.4.26 `vector<string> getSharedNeighbourhood(string username1, string username2, size_t k);`

Returns the list of users whose distance to both users with usernames `username1` and `username2` is less than `k`, in the order of their location in the hash table. Throws a `UserDoesNotExistException` instead if either of the users does not exist.

4 Examples

4.1 Example 1: Adding Users

Steps	Step 1	Step 2

Code	`UserDatabase database(5, 0.50);`
	`database.printDatabase();`
		`database.addUser("firat");`
		`database.printDatabase();`

Results	NULL NULL NULL NULL NULL	NULL NULL NULL firat NULL
Explanation	Created a `UserDatabase` with capacity 5 and maximum load factor of 50%.	Hash function returned the value 3 for the username "firat" and capacity 5. Because 3rd slot was empty, user is placed there.

Steps	Step 3	Step 4

Code	`database.addUser("cihad");`
	`database.printDatabase();`
		`database.addUser("bora");`
		`database.printDatabase();`

Results	NULL NULL NULL firat cihad	bora NULL NULL firat cihad
Explanation	Hash function returned the value 4 for the username "cihad" and capacity 5. Because 4th slot was empty, user is placed there.	Hash function returned the value 3 for the username "bora" and capacity 5. Because 3rd slot was occupied, it was attempted to be placed in the 4th slot. Because 4th slot is also occupied, it is placed in the 0th slot.

Steps	Step 5	

Code	`database.addUser("burak");`
	`database.printDatabase();`	

Results	NULL NULL NULL burak firat NULL cihad NULL NULL NULL bora	
Explanation	Before adding a new user, because current load factor is grater than 50%, the table is resized to have the capacity of 11. Then, in order, "bora", "firat", and "cihad are placed back into the hash table, preserving their follow information. Finally, the new user "burak" is placed into the table.	

4.2 Example 2: Removing Users

Steps	Step 1	Step 2

Code	`UserDatabase database(5, 0.50);`
	`database.addUser("firat");`
	`database.addUser("cihad");`
	`database.printDatabase();`
		`database.removeUser("cihad");`
		`database.printDatabase();`

Results	NULL NULL NULL firat cihad	NULL NULL NULL firat DUMMY
Explanation	Same as events as the Steps 1-3 of Example 1 occurs.	User "cihad" is replaced with a DUMMY user.

Steps	Step 3	Step 4

Code	`database.addUser("bora");`
	`database.printDatabase();`
		`database.addUser("burak");`
		`database.printDatabase();`

Results	NULL NULL NULL firat bora	NULL burak NULL firat bora
Explanation	Hash function returned the value 3 for the username "bora" and capacity 5. This time, there is a DUMMY node in 4th slot, which is overwritten with the user "bora".	Hash function returned the value 1 for the username "burak" and capacity 5. Because 1st slot is empty, it is placed there. Unlike Step 5 of Example 1, there is no need to resize the table.

4.3 Example 3: Following Other Users, BFS and DFS

Steps	Step 1	Step 2

Code	`UserDatabase database(11, 0.75);`
	`database.addUser("firat");`
	`database.addUser("cihad");`
	`database.addUser("bora");`
	`database.addUser("burak");`
	`database.addUser("nihan");`
	`database.addUser("cagri");`
	`database.printDatabase();`
		`database.follow("cihad", "firat");`
		`database.follow("firat", "cihad");`
		`database.follow("bora", "firat");`
		`database.follow("burak", "firat");`
		`database.follow("bora", "cihad");`
		`database.follow("burak", "cihad");`
		`database.follow("bora", "nihan");`
		`database.follow("burak", "nihan");`
		`database.follow("cihad", "nihan");`
		`database.follow("firat", "cagri");`
		`database.follow("nihan", "cagri");`
		`database.follow("cagri", "nihan");`
		`database.follow("nihan", "bora");`
		`database.follow("cagri", "firat");`
		`database.follow("cihad", "burak");`
		`database.printUserList(true);`

Results	NULL NULL cagri burak firat nihan cihad NULL NULL NULL bora	cagri 2 2 burak 1 3 firat 4 2 nihan 4 2 cihad 3 3 bora 1 3
Explanation	Created a `UserDatabase` with capacity 11 and maximum load factor of 75%. Then added the users ”firat”, ”cihad”, ”bora”, ”burak”, ”nihan”, and ”cagri”, in that order.	Users are printed in the order they appear in the hash table (utilizing the `UserDatabase::getUserList()` function) with the number of users that follows them (utilizing the `UserDatabase::userFollowerCount()` function) and the number of users they follow (utilizing the `UserDatabase::userFollowsCount()` function).
Steps	Step 3	Step 4

Code	`vector<string> users =`
	`  database.getUserList();`
	`size_t i = 0, j = 0;`
	`size_t size = users.size();`
	`size_t follows;`
	`for (;i < size; i++){`
	`  UserNode* user =`
	`    database.getUser(users[i]);`
	`  follows = user->follows.size()`
	`  cout << user->username << endl;`
	`  for (;j < follows; j++){`
	`    cout << "- " <<`
	`      user->follows[j] << endl;`
	`  }`
	`}`
		`database.printAdjacencyMatrix();`

Results	cagri
	- nihan
	- firat
	burak
	- firat
	- cihad
	- nihan
	firat
	- cihad
	- cagri
	nihan
	- cagri
	- bora
	cihad
	- firat
	- nihan
	- burak
	bora
	- firat
	- cihad
	- nihan
		0 0 1 1 0 0
		0 0 1 1 1 0
		1 0 0 0 1 0
		1 0 0 0 0 1
		0 1 1 1 0 0
		0 0 1 1 1 0

Explanation	Printed who follows who.	Printed the adjacency matrix. The order of rows and columns are the same as the user list in Step 2.

Steps	Step 4	Step 5

Code	`int bfs =`
	`  database.BFS("firat", "cagri");`
	`cout << "firat -> cagri: " <<`
	`  bfs << endl;`
		`int dfs =`
		`  database.DFS("firat", "cagri");`
		`cout << "firat -> cagri: " <<`
		`  dfs << endl;`

Results	firat cihad cagri firat − > cagri: 1	firat cihad nihan cagri firat − > cagri: 3

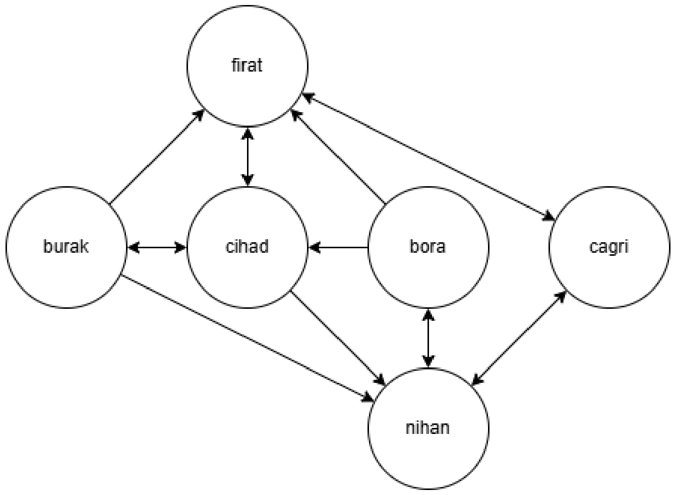

Figure 1: Graph of the social network in Example 3

Explanation	In BFS from ”firat”, we first visit all of its neighbours in the order they appear in its follows. Which means we first visit ”cihad”, then ”cagri”, arriving at our destination. Because we are visiting a neighbour of ”firat”, we get the distance 1.	In DFS from ”firat”, we the first unvisited neighbour of each user (in the order they appear in each user’s follows) until we reach our destination or arrive at a dead-end. Which means we first visit ”cihad”. Because we already visited ”firat”, we visit ”nihan”. Finally from ”nihan” we visit ”cagri”, arriving at our destination. Because we are visiting a neighbour’s neighbour’s neighbour, we get the distance 3.

Steps	Step 6	Step 7
Code	`cout << "Average BFS: " << database.averageBFS() << endl;`	`cout << "Average DFS: " << database.averageDFS() << endl;`
Results	Average BFS: 1.6	Average DFS: 2.5
Explanation	If we calculate BFS distance from each user to each user, excluding unreachable ones (none in this case) and self distances, we achieve an average distance of 1.6.	If we calculate DFS distance from each user to each user, excluding unreachable ones (none in this case) and self distances, we achieve an average distance of 2.5.

5 Regulations

1. **Programming Language:** You will use C++.
2. **Standard Template Library is allowed only for `vector` and `queue`. 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.

6 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.