

VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING



DATA STRUCTURES AND ALGORITHMS - CO2003

ASSIGNMENT 1

**IMPLEMENT A BotKify Application
USING LIST**

HO CHI MINH CITY, 01/2026

ASSIGNMENT'S SPECIFICATION

Version 1.0

1 Assignment's outcome

After completing this major assignment, students will be able to:

- Apply object-oriented programming (OOP) principles proficiently.
- Design and implement list-based data structures.
- Use list-based data structures to implement a Playlist for the BotKify application.

2 Introduction

In Major Assignment 1, students are required to implement a music streaming application named *BotKify*. The application supports adding, removing, updating, and querying songs based on individual listening behaviors, all built upon list-based data structures.

Through this assignment, students will:

- Reinforce object-oriented programming skills through class design and implementation.
- Understand and implement list, stack, and queue data structures.
- Apply lists, stacks, and queues to solve problems while reducing computational complexity.

3 Description

This major assignment requires students to implement a simple music streaming application named *BotKify*. The core focus of the assignment is the design and implementation of list-based data structures, which are then used to manage and process song data within a **Playlist**.

The entire system is built upon three main components:

- A linked list data structure **BotkifyLinkedList**
- The **Song** class
- The playlist management class **Playlist**

In this assignment, the linked list is used as the primary list data structure due to its dynamic memory allocation, non-contiguous storage requirement, and efficiency in insertion and deletion operations during playlist management.

A linked list consists of nodes, where each node stores data and a pointer to the next node in the list. Thanks to this linking mechanism, the list can dynamically grow or shrink during program execution.

3.1 List-Based Data Structures

3.1.1 Linked List – BotkifyLinkedList

The linked list is the fundamental data structure used throughout this major assignment.

Class Node

Attributes:

- **T data:** The data stored in the node.
- **Node* next:** Pointer to the next node in the list.
- **Node* extra:** Reserved pointer for additional purposes (if needed).

Attributes of class BotkifyLinkedList

- **Node* head:** Pointer to the first node in the list.
- **Node* tail:** Pointer to the last node in the list.
- **int count:** The number of elements currently stored in the list.

Public methods of class BotkifyLinkedList

- **BotkifyLinkedList()**
 - **Function:** Initializes an empty linked list.
 - **Exception:** None.
 - **Time Complexity:** $O(1)$.
- **~BotkifyLinkedList()**
 - **Function:** Deallocates all dynamically allocated nodes to prevent memory leaks.
 - **Exception:** None.
 - **Time Complexity:** $O(n)$.

- `void add(T e)`
 - **Function:** Adds element `e` to the end of the list.
 - **Exception:** None.
 - **Time Complexity:** $O(n)$.
- `void add(int index, T e)`
 - **Function:** Inserts element `e` at position `index`.
 - **Exception:** Throws `out_of_range("Index is invalid!")` if `index` is invalid.
 - **Time Complexity:** $O(n)$.
- `T removeAt(int index)`
 - **Function:** Removes and returns the element at position `index`.
 - **Exception:** Throws `out_of_range("Index is invalid!")` if `index` is invalid.
 - **Time Complexity:** $O(n)$.
- `bool removeItem(T item)`
 - **Function:** Removes the first node whose value is equal to `item`. Returns `true` if the removal is successful, and `false` if the item is not found.
 - **Exception:** None.
 - **Time Complexity:** $O(n)$.
- `bool empty()`
 - **Function:** Checks whether the list is empty.
 - **Exception:** None.
 - **Time Complexity:** $O(1)$.
- `int size()`
 - **Function:** Returns the current number of elements in the list.
 - **Exception:** None.
 - **Time Complexity:** $O(1)$.
- `void clear()`
 - **Function:** Removes all nodes from the list.
 - **Exception:** None.
 - **Time Complexity:** $O(n)$.
- `T& get(int index)`
 - **Function:** Returns a reference to the data at position `index` in the list.
 - **Exception:** Throws `out_of_range("Index is invalid!")` if `index` is invalid.

- **Time Complexity:** $O(n)$.
- `string toString()`
 - **Function:** Returns a string representation of the entire list. Elements are printed from head to tail, separated by commas ‘,’ with no spaces between them.
 - **Exception:** None.
 - **Time Complexity:** $O(n)$.
 - **Output format:** <element 1>,<element 2>,<element 3>....

Example 3.1

If the list contains: 1, 2, 3, 4, 5

Returned value of `toString()`: 1,2,3,4,5

3.2 Class Song

The **Song** class is an object used to store and manage information of a song in the *BotKify* application. Each **Song** object represents a specific song and contains essential attributes that support searching, playback, and song data management within the system.

Attributes

- **int id:** Unique identifier of the song.
- **string title:** Title of the song.
- **string artist:** Name of the singer or band performing the song.
- **string album:** Name of the album containing the song.
- **int duration:** Duration of the song (in seconds).
- **int score:** Score of the song (in the range [0–1000]).
- **string url:** URL to the song’s audio source.

Functions

The **Song** class provides basic methods to:

- **int getID()**
 - **Function:** Returns the **id** of the song.
 - **Exception:** No exception is thrown.

- **Time Complexity:** $O(1)$.
- **int getDuration()**
 - **Function:** Returns the **duration** of the song.
 - **Exception:** No exception is thrown.
 - **Time Complexity:** $O(1)$.
- **string toString()**
 - **Function:** Converts the song into a string representation. The fields are concatenated using a hyphen character.
 - **Format:** <title>-<artist>

Example 3.2

If the song has (title, artist) = ("BabyShark", "Pinkfong")

Returned value of toString: BabyShark-Pinkfong

3.3 Class Playlist

The **Playlist** class is used to manage a list of songs in the BotKify application. The song list is implemented using the *BotkifyLinkedList* data structure, in which each node stores a **Song** object.

Attributes of class Playlist

- **string name:** Name of the playlist.
- **BotkifyLinkedList<Song *> lstSong:** A dynamic linked list containing all songs in the playlist.
- **int size:** Current number of songs in the playlist.

Methods of class Playlist

- **Playlist(string name)**
 - **Function:** Initializes a playlist with the given name.
 - **Exception:** None.
 - **Time Complexity:** $O(1)$.
- **int size()**

- **Function:** Returns the number of songs currently in the playlist.
- **Exception:** None.
- **Time Complexity:** $O(1)$.
- `bool empty()`
 - **Function:** Checks whether the playlist is empty.
 - **Exception:** None.
 - **Time Complexity:** $O(1)$.
- `void clear()`
 - **Function:** Removes all songs from the playlist.
 - **Exception:** None.
 - **Time Complexity:** $O(n)$.
- `void addSong(Song s)`
 - **Function:** Adds a new song to the end of the playlist's linked list.
 - **Exception:** None.
 - **Time Complexity:** $O(n)$.
- `void removeSong(int index)`
 - **Function:** Removes the song at position `index` from the linked list.
 - **Exception:** Throws `out_of_range("Index is invalid!")` if the index is invalid.
 - **Time Complexity:** $O(n)$.
- `Song* getSong(int index)`
 - **Function:** Retrieves the song at position `index` in the playlist.
 - **Exception:** Throws `out_of_range("Index is invalid!")` if the index is invalid.
 - **Time Complexity:** $O(n)$.
- `Song* playNext()`
 - **Function:** Starting from the current song, plays the next song (returns the next song). If the current song is the last one, playback wraps around to the first song. If the playlist contains only one song, that song is played.
 - **Exception:** Throws `out_of_range("Index is invalid!")` if the playlist is empty.
 - **Time Complexity:** $O(1)$.
- `Song* playPrevious()`
 - **Function:** Starting from the current song, plays the previous adjacent song. If the current song is the first one, playback wraps around to the last song. If the playlist contains only one song, that song is played.

- **Exception:** Throws `out_of_range("Index is invalid!")` if the playlist is empty.
- **Time Complexity:** $O(1)$.
- `int getTotalScore()`
 - **Function:** Returns the total score of the playlist. The playlist score is defined as the sum of scores of all consecutive song groups. The score of a consecutive group is computed as the product of:
 - * The minimum score within that group.
 - * The sum of scores of all songs in the group.

Example 3.3

For example, a playlist has song scores: $\{4,1,3,5\}$

Score calculation:

- Groups of length 1: $4 \times 4 + 1 \times 1 + 3 \times 3 + 5 \times 5 = 51$
- Groups of length 2: $1 \times (1 + 4) + 1 \times (1 + 3) + 3 \times (3 + 5) = 33$
- Groups of length 3: $1 \times (4 + 1 + 3) + 1 \times (1 + 3 + 5) = 17$
- Groups of length 4: $1 \times (4 + 1 + 3 + 5) = 13$

Therefore, the total playlist score is: $51 + 33 + 17 + 13 = 114$

- **Exception:** None.
- **Time Complexity:** $O(N)$.
- `bool compareTo(Playlist p, int numSong)`
 - **Function:** In addition to computing the total playlist score, BotKify allows users to compare playlists based on score definitions and a specified number of consecutive songs (`numSong`). To support this feature, the **compareTo** method performs the following tasks:
 - * For each playlist, consider all groups of `numSong` consecutive songs and determine the maximum score in each group.
 - * Compute the average score of all such groups for each playlist and compare them. Return *true* if the current playlist has an average score greater than or equal to playlist **p**; otherwise, return *false*.

Example 3.4

Playlist A has song scores {4,1,3,5}, Playlist B has song scores {1,2,3}. Compare playlists with `numSong = 2`.

Score of Playlist A:

$$(\max\{4, 1\} + \max\{1, 3\} + \max\{3, 5\})/3 = 4$$

Score of Playlist B:

$$(\max\{1, 2\} + \max\{2, 3\})/2 = 2.5$$

The call **A.compareTo(B, 2)** returns *true*.

- **Exception:** None.
- **Time Complexity:** $O(N)$.
- `void playRandom(int index)`
 - **Function:** Performs random-like playback starting from the song at position **index**. Each subsequent song selected is the closest song whose **duration** is greater than the duration of the previously played song. Playback continues until no further song can be selected.
 - **Exception:** None.
 - **Time Complexity:** $O(N)$.
 - **Output format:** Songs are printed in playback order as:
`<title1>-<artist1>,<title2>-<artist2>,...`
- `int playApproximate(int step)`
 - **Function:** This feature optimizes listening time by selecting songs with approximately similar durations, avoiding large duration differences between consecutive songs. Due to network constraints, up to **step** songs may be skipped. The playlist must start from the first song and end at the last song. The objective is to minimize the total duration difference between consecutive played songs. The method returns the minimum total duration difference.

Example 3.5

For example, playlist A has song durations: {50, 60, 30, 90, 100}. The number of skippable songs is **step = 1**.

Selected songs: song 0, song 1, song 3, song 4

Returned total difference: $10 + 30 + 10 = 50$

- **Exception:** None.

- **Time Complexity:** $O(N \times \text{step})$.

4 Requirements and Grading

4.1 Requirements

To complete this assignment, students are required to:

1. Read the entire description file.
2. Download the file **initial.zip** and extract it. After extraction, students will obtain the following files: `utils.h`, `main.cpp`, `main.h`, `Playlist.h`, `Playlist.cpp`, and `BotkifyLinkedList.h`. Students are only required to submit 3 files: `BotkifyLinkedList.h`, `Playlist.h`, and `Playlist.cpp`. Therefore, modifying the file `main.h` is not allowed during program evaluation.
3. Students use the following command to compile the program: `g++ -o main main.cpp Playlist.cpp -I . -std=c++17` The above command is used in the Command Prompt/Terminal to compile the program. If students use an IDE to run the program, they should note the following: add all files to the IDE's project/workspace; adjust the compilation command in the IDE accordingly. IDEs usually provide Build (compile) and Run buttons. When clicking Build, the IDE will execute the corresponding compilation command, which typically compiles only the file `main.cpp`. Students need to find a way to configure the compilation command, specifically: add the file `Playlist.cpp`, add the option `-std=c++17`, and `-I .`
4. The program will be graded on a Unix platform. The student's environment and compiler may differ from the actual grading environment. The submission area on the LMS is configured similarly to the grading environment. Students are required to test their program on the submission page and fix all errors that arise on the LMS system to ensure accurate results during the final grading.
5. Modify the files `BotkifyLinkedList.h`, `Playlist.h`, and `Playlist.cpp` to complete the assignment, while ensuring the following two requirements:
 - All methods described in the instruction file must be implemented so that the program can compile successfully. If a student has not yet implemented a particular method, an empty implementation must be provided for that method. Each testcase will call certain methods to check the returned results.
 - In the file `Playlist.h`, only `include "main.h"` and `#include "BotkifyLinkedList.h"`

are allowed, and in the file `Playlist.cpp`, only one line `#include "Playlist.h"` is allowed. Apart from these lines, no other `#include` directives are permitted in these files.

6. Students are encouraged to write additional supporting classes, methods, and attributes within the required classes. However, students must ensure that these classes and methods do not change the requirements of the methods described in the assignment.
7. Students must design and use the data structures that have been learned.
8. Students are required to free all dynamically allocated memory when the program terminates.

4.2 Submission Deadline

The deadline is **as announced on LMS**. Students must submit their assignments to the system before the specified deadline. Students are fully responsible for any issues arising from submissions made too close to the deadline.

4.3 Grading

All student source code will be evaluated using hidden test cases. The score is calculated separately for each requirement, as follows:

- Implementation of the linked list for BotKify: **4 points**.
- Implementation of the Playlist: **6 points**.

5 Harmony Questions

The final exam for the course will include several “Harmony” questions related to the content of the Assignment.

Students must complete the Assignment by their own ability. If a student cheats in the Assignment, they will not be able to answer the Harmony questions and will receive a score of 0 for the Assignment.

Students **must** pay attention to completing the Harmony questions in the final exam. Failing to do so will result in a score of 0 for the Assignment, and the student will fail the course. **No explanations and no exceptions.**

6 Regulations and Handling of Cheating

The Assignment must be done by the student THEMSELVES. A student will be considered cheating if:

- There is an unusual similarity between the source code of submitted projects. In this case, ALL submissions will be considered as cheating. Therefore, students must protect their project source code.
- The student does not understand the source code they have written, except for the parts of code provided in the initialization program. Students can refer to any source of material, but they must ensure they understand the meaning of every line of code they write. If they do not understand the source code from where they referred, the student will be specifically warned NOT to use this code; instead, they should use what has been taught to write the program.
- Submitting someone else's work under their own account.
- Students use AI tools during the Assignment process, resulting in identical source code.

If the student is concluded to be cheating, they will receive a score of 0 for the entire course (not just the assignment).

NO EXPLANATIONS WILL BE ACCEPTED AND THERE WILL BE NO EXCEPTIONS!

After the final submission, some students will be randomly selected for an interview to prove that the submitted project was done by them.

Other regulations:

- All decisions made by the lecturer in charge of the assignment are final decisions.
- Students are not provided with test cases after the grading of their project.
- The content of the Assignment will be harmonized with questions in the exam that has similar content.

—————**THE END**—————