Lee Zong Yang 100073484

Table of Contents

Task 1

Description

Implement an event loop. Define a class of player's character and instantiate an object to represent the player during the event loop. The class should have stub methods that can check whether a skill is existed, add and remove skill. Testing should be done too.

Concept

I choose while loop instead of GOTO because it is a better programming practice. Error handling of input had been implemented too. At this step, only stub methods are required so I just write some dummy code inside the stub methods.

Implementation

```
#pragma once
#include <iostream>
#include <string>

using namespace std;

class PlayerCharacter
{
public:
    bool skillCheck(string skillName, int level) {
        return true;
    }

    void addNewSkill(string skillName, int level) {
        cout << "Added new skill" << endl;
    }

    void removeSkill(string skillName, int level) {
        cout << "Removed skill" << endl;
    }
};
```

```
#include <iostream>
#include "PlayerCharacter.h"

int main()
{
    PlayerCharacter pC;
    pC.addNewSkill("Intimidation Skill", 3);
    pC.addNewSkill("Persuasion Skill", 5);
    pC.removeSkill("Useless Skill", 1);
    while (true) {
        cout << endl << "Hand over your money" << endl;
        cout << endl << "(1) Here's my money." << endl;
        cout << "(2) You can try, when you get older." << endl;
        cout << "(3) Money? Life? I'm an Engineer." << endl;
        cout << "(4) *Fight*" << endl;

        int selection;
        cin >> selection;

        if (selection == 1) {
            exit(0);
        }
        else if (selection == 2) {
            pC.skillCheck("Intimidation Skill", 3);
            cout << endl << "Ok..." << endl;
        }
        else if (selection == 3) {
            pC.skillCheck("Persuasion Skill", 5);
            cout << endl << "Acceptable..." << endl;
        }
        else if (selection == 4) {
            cout << "Combat !!!" << endl;
        }
    }
}
```

Output

Figure: console output

Troubleshooting

Like the task I attempted before so no problem.

Task 2

Description

The following tasks do require a singly linked-list or doubly linked-list, so a class to represent a node in list need to complete in this task. The node should fulfil the requirement of holding the data and pointer to next node.

Concept

The reason I choose Singly Linked-List over Doubly Linked-List is zero need to traverse the list in two direction. With a doubly linked-list, it would take more space for each node as an extra pointer to previous node is required. The main operation is to find whether a skill exist or not, so it is O(n) time complexity for both singly and doubly.

Implementation

```cpp
#pragma once
#include <stdio.h>
#include <iostream>

using namespace std;

template <class DataType>
class LinkedListNode
{
public:
    DataType val;
    LinkedListNode<DataType>* next;
    LinkedListNode(DataType val) {
        this->val = val;
        next = NULL;
    }

    LinkedListNode() {
        next = NULL;
    }
};
```

Figure: Linked list node (LinkedListNode.h)

```cpp
#include <iostream>
#include "PlayerCharacter.h"
#include "LinkedListNode.h"
#include "Skill.h"

int main()
{
    Skill skill("Intimidation Skill", 3);
    LinkedListNode<Skill>* skillNode = new LinkedListNode<Skill>(skill);
    cout << skillNode->val << endl;

    Skill skill2("Persuasion Skill", 5);
    LinkedListNode<Skill>* skillNode2 = new LinkedListNode<Skill>(skill2);
    cout << skillNode2->val << endl;
}
```

Figure: main function (main.cpp)

Output

```
Intimidation Skill: 3
Persuasion Skill: 5

D:\Swinburne Units\Data Structure\Problem Set 3\Task 2\Debug\Task 2.exe (process 22844) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Figure: console output

Troubleshooting

To make the node as generic as possible, I found out template class is the way to go.

Task 3

Description

Implement a linked list, either directly into the PlayerCharacter class or an iterator to manage the linked list. Adequate interface and functionalities should be implemented for the stub functions of PlayerCharacter, namely Skill Check, Skill Add, and Skill Remove. Testing should be done.

Concept

Singly linked list is used to solve this problem. In my implementation, there is head and tail pointer, thus the time complexity to prepend and append is both O(1). The linked list consists solely of data and pointer to the next node. Therefore, only added skill will increase the storage cost of linked list and potential unused space problem in array will not happen. To prepend and append in array, the time complexity is O(n) as it is needed to move all elements in the array to arrange for the position. There is a need to search for certain skill, and the time complexity for the operation is O(n) for linked list, but it is O(n) for array too, as the position of the item is not known and traverse through the whole array is needed in the worst-case scenario. Frequent delete at certain index operation is carried out when skill is removed, and linked-list triumph over array with O(1) time complexity.

I did not choose to implement an iterator to manage the linked list is due to the idiom “Pre-optimization is the root of devil”. Iterator could enhance the open closed principle, but there is no need at this moment as the purpose of this program is clear, to finish problem set 3.

Implementation

```cpp
#pragma once
#include <iostream>
#include <string>

using namespace std;

class Skill
{
public:
    string name;
    int level;

    Skill(string Name, int Level) : name(Name), level(Level) {

    }

    Skill() {

    }

    friend ostream& operator<<(ostream& aOstream, Skill skill) {
        aOstream << skill.name << ": " << skill.level;
        return aOstream;
    }

    friend bool operator== (Skill s1, Skill s2) {
        return (s1.name == s2.name);
    }
};
```

Figure: Skill class (Skill.h)

```cpp
#pragma once
#include "LinkedListNode.h"
#include "windows.h"

template <class DataType>
class LinkedList
{
public:
    /** Initialize your data structure here. */
    int size = 0;
    LinkedListNode<DataType>* head = new LinkedListNode<DataType>();
    LinkedListNode<DataType>* tail = new LinkedListNode<DataType>();

    LinkedList() {}

    // copy constructor
    LinkedList(const LinkedList& src) {
        LinkedListNode<DataType>* node = src.head;
        while (node->next != NULL)
        {
            node = node->next;
            append(node->val);
        }
    }

    // assignment operator
    LinkedList& operator=(LinkedList src)
    {
        swap(head, src.head);
        size = src.size;
        return *this;
    }

    //destructor
    ~LinkedList()
    {
        LinkedListNode<DataType>* curr = head;
        while (curr != NULL) {
            LinkedListNode<DataType>* next = curr->next;
            delete curr;
            curr = next;
        }
        head = NULL;
    }

    /** Get the value of the index-th node in the linked list. If the index is invalid, throw error. */
    DataType& getValueByIndex(int index) {
        if (index >= size) throw "Index Invalid";
        LinkedListNode<DataType>* temp = head;
        for (int i = 0; i <= index; i++) temp = temp->next;
        return temp->val;
    }

    DataType& getValue(DataType val) {
        LinkedListNode<DataType>* temp = head;
        for (int i = 0; i < size; i++) {
            temp = temp->next;
            if (temp->val == val) {
                return temp->val;
            }
        }
        throw "Not Found";
    }

    /** Get the value of the index-th node in the linked list. If the index is invalid, throw error. */
    DataType get(int index) {
        if (index >= size) throw "Index Invalid";
        LinkedListNode<DataType>* temp = head;
        for (int i = 0; i <= index; i++) temp = temp->next;
        return temp->val;
    }

    bool exist(DataType val) {
        LinkedListNode<DataType>* temp = head;
        for (int i = 0; i < size; i++) {
            temp = temp->next;
            if (temp->val == val) {
                return true;
            }
        }
        return false;
    }
```

```
    bool remove(DataType val) {
        LinkedListNode<DataType>* temp = head;
        LinkedListNode<DataType>* deleteNode;
        for (int i = 0; i < size; i++) {
            if (temp->next->val == val) {
                size--;
                deleteNode = temp->next;
                if (tail->next == deleteNode) {
                    delete deleteNode;
                    tail->next = temp;
                    temp->next = NULL;
                    return true;
                }
                else {
                    temp->next = deleteNode->next;
                    delete deleteNode;
                    return true;
                }
            }
            temp = temp->next;
        }
        return false;
    }

    //correct
    /** Add a node of value val before the first element of the linked list. After the insertion, the new node will be the first node of the linked list. */
    void prepend(DataType val) {
        LinkedListNode<DataType>* node = new LinkedListNode<DataType>(val);
        if (head->next == NULL && tail->next == NULL) {
            head->next = node;
            tail->next = node;
        }
        else {
            node->next = head->next;
            head->next = node;
        }
        size++;
    }

    //correct
    /** Append a node of value val to the last element of the linked list. */
    void append(DataType val) {
        LinkedListNode<DataType>* node = new LinkedListNode<DataType>(val);
        if (head->next == NULL && tail->next == NULL) {
            head->next = node;
            tail->next = node;
        }
        else {
            tail->next->next = node;
            tail->next = node;
        }
        size++;
    }

    void display() {
        LinkedListNode<DataType>* temp = head;
        int counter = 0;
        while (temp->next != NULL) {
            Sleep(100);
            counter++;
            temp = temp->next;
            //will call the overloaded operator of the data type.
            cout << "(" << counter << ")" << temp->val << endl;
        }
    }

    //should be correct
    /** Add a node of value val before the index-th node in the linked list. If index equals to the length of linked list, the node will be appended to the end of linked list. If index is greater than
      the length, the node will not be inserted. */
    void addAtIndex(int index, DataType val) {
        if (index > size) return;
        LinkedListNode<DataType>* temp = head;
        for (int i = 0; i < index; i++) {
            temp = temp->next;
        }
        LinkedListNode<DataType>* node = new LinkedListNode<DataType>(val);
        if (head->next == NULL && tail->next == NULL) {
            head->next = node;
            tail->next = node;
        }
        else {
            node->next = temp->next;
            temp->next = node;
        }
        size++;
    }
};
```

Figure: Linked list (LinkedList.h)

```cpp
// Task 1.cpp : This file contains the 'main' function. Program execution begins and ends there.
//

#include <iostream>
#include "PlayerCharacter.h"

int main()
{
    Skill skill("Intimidation Skill", 3);
    Skill skill2("Persuasion Skill", 5);
    LinkedList<Skill>* skills = new LinkedList<Skill>();

    if (!skills->exist(skill)) {
        skills->append(skill);
    }
    else {
        skills->getValue(skill).level += skill.level;
    }

    if (!skills->exist(skill2)) {
        skills->append(skill2);
    }
    else {
        skills->getValue(skill2).level += skill2.level;
    }

    skills->display();

    if (!skills->exist(skill)) {
        skills->append(skill);
    }
    else {
        skills->getValue(skill).level += skill.level;
    }

    skills->remove(skill2);

    skills->display();

}
```

Figure: main (main.cpp)

```
(1)Intimidation Skill: 3
(2)Persuasion Skill: 5
(1)Intimidation Skill: 6

D:\Swinburne Units\Data Structure\Problem Set 3\Task 3\Debug\Task 3.exe (process 27820) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Figure: console output

Troubleshooting

There are certain functions of the linked list to search for an item. This could be done easily is the item is primitive data, but not composite data. For composite data, we need comparison operator overloading. Luckily, the compiler reports the problem clearly and I managed to find this solution on StackOverFlow.

Task 4

Description

Add a set of skills into the player character object at the start of the game. Implement the game loop to mimic a conversation. The response options should be available and will perform the necessary skill check. Then, correct feedback needs to be done based on the skill check.

Concept

This is to test the correctness of the linked list implementation and the control statement in the game loop. A check on the programmer's cautiousness. The understanding of linked list function and manipulation of it in the functions of the player character class.

Code

```cpp
#include <iostream>
#include "PlayerCharacter.h"

int main()
{
    PlayerCharacter pC;
    pC.addNewSkill("Intimidation Skill", 3);
    pC.addNewSkill("Persuasion Skill", 4);

    while (true) {
        cout << endl << "Hand over your money" << endl;
        cout << endl << "(1) Here's my money." << endl;
        cout << "(2) You can try, when you get older." << endl;
        cout << "(3) Money? Life? I'm an Engineer." << endl;
        cout << "(4) *Fight*" << endl;

        int selection;
        cin >> selection;

        if (selection == 1) {
            exit(0);
        }
        else if (selection == 2) {
            if (pC.skillCheck("Intimidation Skill", 3)) {
                cout << endl << "Ok..." << endl;
            }
            else {
                cout << endl << "#%*@^$" << endl;
            }
        }
        else if (selection == 3) {
            if (pC.skillCheck("Persuasion Skill", 5)) {
                cout << endl << "Acceptable..." << endl;
            }
            else {
                cout << endl << "???" << endl;
                pC.addNewSkill("Persuasion Skill", 1);
            }
        }
        else if (selection == 4) {
            cout << "Combat !!!" << endl;
            pC.removeSkill("Persuasion Skill", 5);
        }
    }
}
```

Figure: main function (main.cpp)

```cpp
class PlayerCharacter
{
public:
    LinkedList<Skill> skills;

    bool skillCheck(string skillName, int level) {
        Skill skill(skillName, level);
        if (skills.exist(skill)) {
            if (skills.getValue(skill).level >= level) {
                return true;
            }
            else {
                return false;
            }
        }
        else {
            return false;
        }
    }

    void addNewSkill(string skillName, int level) {
        Skill skill(skillName, level);
        if (skills.exist(skill)) {
            skills.getValue(skill).level += skill.level;
            cout << "Upgrade skill level" << endl;
        }
        else {
            skills.append(skill);
            cout << "Added new skill: " << skill.name << endl;
        }
    }

    void removeSkill(string skillName, int level) {
        Skill skill(skillName, level);
        if (skills.exist(skill)) {
            skills.remove(skill);
            cout << "Removed skill" << endl;
        }
    }
};
```

Figure: PlayerCharacter class (PlayerCharacter.h)

Output

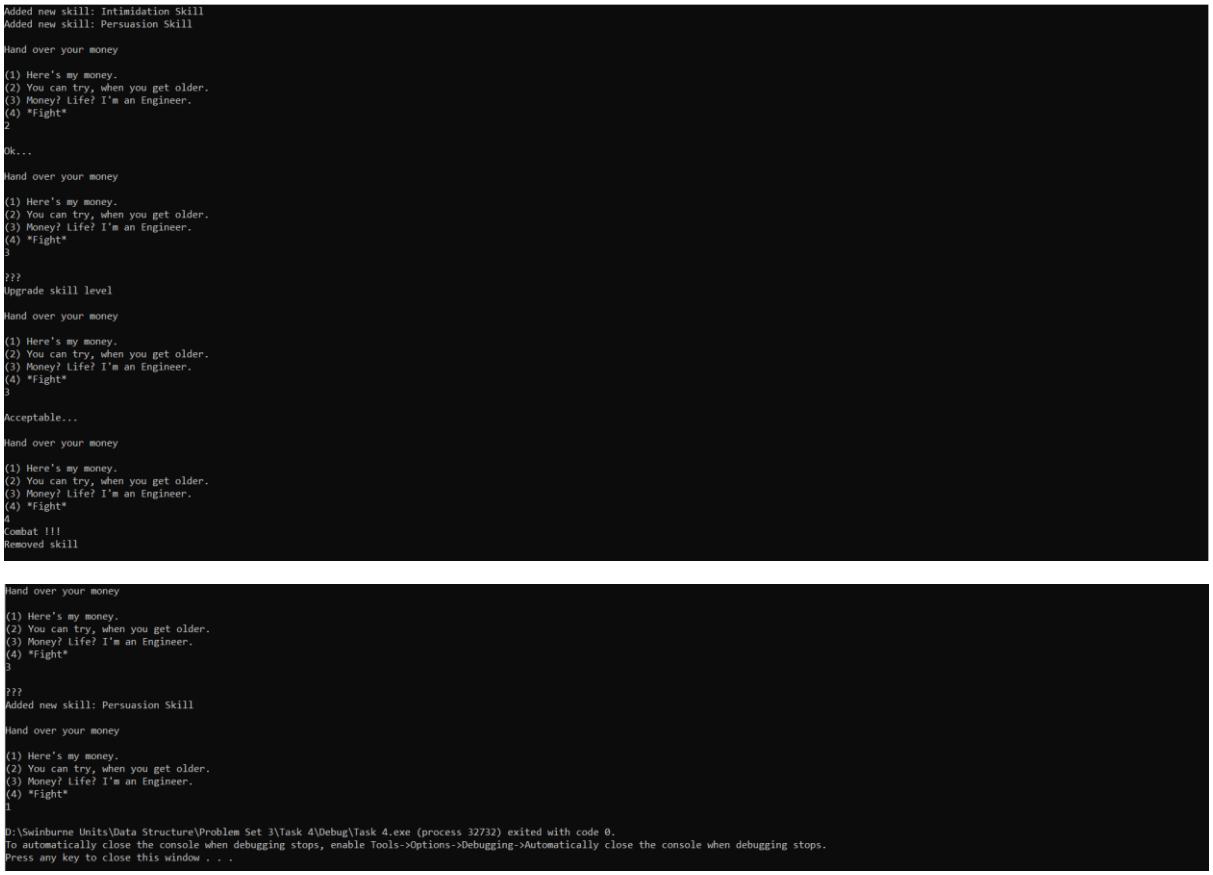

Figure: Console output

Troubleshooting

I must implement custom copy constructor to manage the dynamically allocated resource. Thanks to https://stackoverflow.com/questions/3279543/what-is-the-copy-and-swap-idiom , I managed to do it easily. The assignment operator could be implemented easily if the custom copy constructor had been implemented.

Task 5

Description

Review the ADT called “List” and its standard characteristics. Then, build the ADT and replace the current implementation in PlayerCharacter class with it.

Concept

The main motive is to accomplish the abstraction and encapsulation principles. Prior to the refactoring, the management code of linked list is hardcoded inside the character class. If we could move all these code into a separate class, then the code of character class will be cleaner and easier to modify. The character object only manipulates the linked list through the ADT functions.

Implementation

```cpp
#pragma once
#include "SkillList.h"

class PlayerCharacter
{
public:
    SkillList skills;

    bool skillCheck(string skillName, int level) {
        return skills.skillCheck(skillName, level);
    }

    void addNewSkill(string skillName, int level) {
        skills.addNewSkill(skillName, level);
    }

    void removeSkill(string skillName, int level) {
        skills.removeSkill(skillName, level);
    }
};
```

Figure: PlayerCharacter class (PlayerCharacter.h)

```cpp
class SkillList
{
public:
    LinkedList<Skill> skills;

    bool skillCheck(string skillName, int level) {
        Skill skill(skillName, level);
        if (skills.exist(skill)) {
            if (skills.getValue(skill).level >= level) {
                return true;
            }
            else {
                return false;
            }
        }
        else {
            return false;
        }
    }

    void addNewSkill(string skillName, int level) {
        Skill skill(skillName, level);
        if (skills.exist(skill)) {
            skills.getValue(skill).level += skill.level;
            cout << "Upgrade skill level" << endl;
        }
        else {
            skills.append(skill);
            cout << "Added new skill: " << skill.name << endl;
        }
    }

    void removeSkill(string skillName, int level) {
        Skill skill(skillName, level);
        if (skills.exist(skill)) {
            skills.remove(skill);
            cout << "Removed skill" << endl;
        }
    }
};
```

Figure: ADT (SkillList.h)

Output

```
Added new skill: Intimidation Skill
Added new skill: Persuasion Skill

Hand over your money

(1) Here's my money.
(2) You can try, when you get older.
(3) Money? Life? I'm an Engineer.
(4) *Fight*
2

Ok...

Hand over your money

(1) Here's my money.
(2) You can try, when you get older.
(3) Money? Life? I'm an Engineer.
(4) *Fight*
3

???
Upgrade skill level

Hand over your money

(1) Here's my money.
(2) You can try, when you get older.
(3) Money? Life? I'm an Engineer.
(4) *Fight*
3

Acceptable...

Hand over your money

(1) Here's my money.
(2) You can try, when you get older.
(3) Money? Life? I'm an Engineer.
(4) *Fight*
4
Combat !!!
Removed skill
```

Figure: Console output

Troubleshooting

Fairly easy task. No problem.

Appendix

```cpp
int main()
{
    PlayerCharacter pC;
    pC.addNewSkill("Intimidation Skill", 3);
    pC.addNewSkill("Persuasion Skill", 4);

    while (true) {
        cout << endl << "Hand over your money" << endl;
        cout << endl << "(1) Here's my money." << endl;
        cout << "(2) You can try, when you get older." << endl;
        cout << "(3) Money? Life? I'm an Engineer." << endl;
        cout << "(4) *Fight*" << endl;

        int selection;
        cin >> selection;

        if (selection == 1) {
            exit(0);
        }
        else if (selection == 2) {
            if (pC.skillCheck("Intimidation Skill", 3)) {
                cout << endl << "Ok..." << endl;
            }
            else {
                cout << endl << "#%*@^$" << endl;
            }
        }
        else if (selection == 3) {
            if (pC.skillCheck("Persuasion Skill", 5)) {
                cout << endl << "Acceptable..." << endl;
            }
            else {
                cout << endl << "???" << endl;
                pC.addNewSkill("Persuasion Skill", 1);
            }
        }
        else if (selection == 4) {
            cout << "Combat !!!" << endl;
            pC.removeSkill("Persuasion Skill", 5);
        }
    }
}
```

Figure: main.cpp

```cpp
class PlayerCharacter
{
public:
    SkillList skills;

    bool skillCheck(string skillName, int level) {
        return skills.skillCheck(skillName, level);
    }

    void addNewSkill(string skillName, int level) {
        skills.addNewSkill(skillName, level);
    }

    void removeSkill(string skillName, int level) {
        skills.removeSkill(skillName, level);
    }
};
```

Figure: PlayerCharacter.h

```
class SkillList
{
public:
    LinkedList<Skill> skills;

    bool skillCheck(string skillName, int level) {
        Skill skill(skillName, level);
        if (skills.exist(skill)) {
            if (skills.getValue(skill).level >= level) {
                return true;
            }
            else {
                return false;
            }
        }
        else {
            return false;
        }
    }

    void addNewSkill(string skillName, int level) {
        Skill skill(skillName, level);
        if (skills.exist(skill)) {
            skills.getValue(skill).level += skill.level;
            cout << "Upgrade skill level" << endl;
        }
        else {
            skills.append(skill);
            cout << "Added new skill: " << skill.name << endl;
        }
    }

    void removeSkill(string skillName, int level) {
        Skill skill(skillName, level);
        if (skills.exist(skill)) {
            skills.remove(skill);
            cout << "Removed skill" << endl;
        }
    }
};
```

Figure: SkillList.h

```
template <class DataType>
class LinkedList
{
public:
    /** Initialize your data structure here. */
    int size = 0;
    LinkedListNode<DataType>* head = new LinkedListNode<DataType>();
    LinkedListNode<DataType>* tail = new LinkedListNode<DataType>();

    LinkedList() {}

    // copy constructor
    LinkedList(const LinkedList& src) {
        LinkedListNode<DataType>* node = src.head;
        while (node->next != NULL)
        {
            node = node->next;
            append(node->val);
        }
    }

    // assignment operator
    LinkedList& operator=(LinkedList src)
    {
        swap(head, src.head);
        size = src.size;
        return *this;
    }

    //destructor
    ~LinkedList()
    {
        LinkedListNode<DataType>* curr = head;
        while (curr != NULL) {
            LinkedListNode<DataType>* next = curr->next;
            delete curr;
            curr = next;
        }
        head = NULL;
    }
```

```
    /** Get the value of the index-th node in the linked list. If the index is invalid, throw error. */
    DataType& getValueByIndex(int index) {
        if (index >= size) throw "Index Invalid";
        LinkedListNode<DataType>* temp = head;
        for (int i = 0; i <= index; i++) temp = temp->next;
        return temp->val;
    }

    DataType& getValue(DataType val) {
        LinkedListNode<DataType>* temp = head;
        for (int i = 0; i < size; i++) {
            temp = temp->next;
            if (temp->val == val) {
                return temp->val;
            }
        }
        throw "Not Found";
    }

    /** Get the value of the index-th node in the linked list. If the index is invalid, throw error. */
    DataType get(int index) {
        if (index >= size) throw "Index Invalid";
        LinkedListNode<DataType>* temp = head;
        for (int i = 0; i <= index; i++) temp = temp->next;
        return temp->val;
    }

    bool exist(DataType val) {
        LinkedListNode<DataType>* temp = head;
        for (int i = 0; i < size; i++) {
            temp = temp->next;
            if (temp->val == val) {
                return true;
            }
        }
        return false;
    }
```

```
bool remove(DataType val) {
    LinkedListNode<DataType>* temp = head;
    LinkedListNode<DataType>* deleteNode;
    for (int i = 0; i < size; i++) {
        if (temp->next->val == val) {
            size--;
            deleteNode = temp->next;
            if (tail->next == deleteNode) {
                delete deleteNode;
                tail->next = temp;
                temp->next = NULL;
                return true;
            }
            else {
                temp->next = deleteNode->next;
                delete deleteNode;
                return true;
            }
        }
        temp = temp->next;
    }
    return false;
}

//correct
/** Add a node of value val before the first element of the linked list. After the insertion, the new node will be the first node of the linked list. */
void prepend(DataType val) {
    LinkedListNode<DataType>* node = new LinkedListNode<DataType>(val);
    if (head->next == NULL && tail->next == NULL) {
        head->next = node;
        tail->next = node;
    }
    else {
        node->next = head->next;
        head->next = node;
    }
    size++;
}
```

```
//correct
/** Append a node of value val to the last element of the linked list. */
void append(DataType val) {
    LinkedListNode<DataType>* node = new LinkedListNode<DataType>(val);
    if (head->next == NULL && tail->next == NULL) {
        head->next = node;
        tail->next = node;
    }
    else {
        tail->next->next = node;
        tail->next = node;
    }
    size++;
}

void display() {
    LinkedListNode<DataType>* temp = head;
    int counter = 0;
    while (temp->next != NULL) {
        Sleep(100);
        counter++;
        temp = temp->next;
        //will call the overloaded operator of the data type.
        cout << "(" << counter << ")" << temp->val << endl;
    }
}
```

```
    //should be correct
    /** Add a node of value val before the index-th node in the linked list. If index equals to the length of linked list, the node will be appended to the end of linked list. If index is greater than
       the length, the node will not be inserted. */
    void addAtIndex(int index, DataType val) {
        if (index > size) return;
        LinkedListNode<DataType>* temp = head;
        for (int i = 0; i < index; i++) {
            temp = temp->next;
        }
        LinkedListNode<DataType>* node = new LinkedListNode<DataType>(val);
        if (head->next == NULL && tail->next == NULL) {
            head->next = node;
            tail->next = node;
        }
        else {
            node->next = temp->next;
            temp->next = node;
        }
        size++;
    }
};
```

Figure: LinkedList.h

```
template <class DataType>
class LinkedListNode
{
public:
    DataType val;
    LinkedListNode<DataType>* next;
    LinkedListNode(DataType val) {
        this->val = val;
        next = NULL;
    }

    LinkedListNode() {
        next = NULL;
    }
};
```

Figure: LinkedListNode.h