4주차 결과보고서

전공: 신문방송학과 학년: 3학년 학번: 20191150 이름: 전현길

1. 실험 시간에 작성한 프로그램의 알고리즘과 자료구조를 요약해 기술하시오.

4주차 실습에서 작성한 프로그램의 기반이 된 자료 구조는 연결 리스트(linked list)와 스택(stack)이었다. 먼저 연결 리스트는 배열에 비해 삽입, 삭제, 검색의 worst-case 시간 복잡도가 O(n)이 걸린다는 단점이 있으나, 메모리를 동적으로 할당할 수 있다는 장점을 갖는다. 다음으로 스택은 가장 마지막에 입력된 데이터부터 역순으로 출력하는 후입선출 구조의 자료구조이다. 이번 프로그램에서는 스택 자료 구조를 연결 리스트를 통해 구현하는 프로그램을 작성했다.

4주차 실습에선 상향식 프로그래밍을 수행했다. 먼저 int 자료형의 데이터를 연결 리스트 형태로 삽입, 삭제, 출력하는 LInkedList 클래스를 만들고, 이후에 LinkedList에 파라미터적 다형성을 적용하여 int 자료형뿐만 아니라 다양한 자료형으로 접근할 수 있도록 했다. 마지막으로 LinkedList를 상속받는 파생 클래스인 Stack 클래스를 구현한 뒤 Delete() 함수를 재정의하여 Stack 클래스를 구현했다.

최종적으로 [그림 2] 코드를 기반으로 파라미터적 다형성이 구현되었는지 검토했고, [그림 4] 코드를 기반으로 스택, 연결 리스트가 각각 잘 작동하는지 확인했다.

아래에서 설명될 클래스는 모두 템플릿 클래스이며, typename은 임의로 T로 작성했다. 먼저 LinkedList에 저장될 **Node 클래스**는 2개의 멤버 변수(T data, Node *link)를 가지며, 생성자 Node(T element)를 갖는다. T data 변수에 노드의 데이터가 저장되며, link 변수에는 다음 노드의 주소 포인터가 저장된다. 생성자는 원소를 입력받고 해당 원소의 값으로 element 멤버 변수를 초기화한 뒤, link 포인터 값을 NULL로 초기화한다.

다음으로 **LinkedList 클래스**는 2개의 멤버 변수와 5개의 멤버 함수를 갖는다. 멤버 변수 Node<T> *first, int current_size는 각각 연결 리스트의 첫 번째 노드를 가리키는 포인터, 연결 리스트의 길이를 저장한다. 멤버 함수는 각각 다음과 같다.

1. 생성자 LinkedList() : first를 널 값으로 초기화, current_size 0으로 초기화
2. int GetSize() : 배열의 길이를 반환
3. void Insert(T element) : 연결 리스트의 맨 앞에 원소를 삽입
4. virtual bool Delete(T &element) : 연결 리스트 맨 뒤의 원소를 삭제
5. void Print() : 연결 리스트에 저장된 데이터를 출력 형식에 맞춰 끝까지 출력

실제 멤버 함수를 구현하는 구현부에는 LinkedList<T> 형태로 typename을 꼭 표

시해 주어야 한다. Node 클래스 역시 Node<T> 형태로 선언해 주어야 요구되는 자료형을 지원하는 Node 클래스가 생성될 수 있다. 따라서, 노드를 생성하는 코드 역시 다음과 같이 <T> 형태로 자료형을 표기해 준다.

```
Node<T> *newnode = new Node<T>(element);
```

구체적인 동작의 구현을 설명하면 다음과 같다. 단, 생성자와 GetSize(), Print() 함수는 동작이 매우 단순하기 때문에 설명을 생략한다.

Insert() 함수: newnode를 생성한 뒤, newnode가 first 포인터를 가리키게 한 후 first 포인터의 주소를 newnode로 갱신하고 current_size를 1 더하여 요구되는 기능을 수행한다.

Delete() 함수: 배열이 비어 있을 때 삭제가 이뤄질 경우 false를 반환하고 종료한다. 배열이 비어 있지 않다면 마지막 노드를 찾기 위해 두 Node<T> 포인터 current, previous를 선언한다.

current, previous 포인터를 첫 번째 노드(first)에서부터 갱신하되 current->link가 널 포인터를 만난다면 반복문을 종료한다. 종료 시 previous 포인터가 널 포인터였다면(노드가 1개밖에 없었다면) first 포인터를 널로 초기화하고, 널 포인터가 아니었다면(노드가 2개 이상 있었다면) 이전 노드의 link 변수를 널로 초기화한다.

이후 current의 데이터를 int형 참조자 변수 element에 복사한 뒤, current를 삭제한 후 배열 크기를 1 뺀다. 마지막으로 true를 반환하고 종료한다.

이 때 Delete() 함수를 virtual 함수로 선언하는 이유는, 만약 파생 클래스인 Stack 클래스의 변수를 LinkedList 포인터로 호출하게 될 때 부모 클래스 멤버 함수를 부르지 않도록 하기 위해서이다. 만약 해당 함수를 virtual 함수로 선언하지 않았을 경우 Stack 클래스의 변수를 LinkedList 포인터로 호출한 뒤 Delete() 멤버 함수를 부르면, 부모 클래스의 Delete() 멤버 함수가 호출되어 끝에 있는 노드가 삭제된다.

마지막으로 **Stack 클래스**는 LinkedList 클래스의 대부분의 멤버 변수 및 함수를 상속받되, Delete() 함수만 재정의하게 된다. 스택은 후입선출 자료구조이므로 가장 앞에 있는 변수를 삭제해야 한다. 따라서, 이외의 동작은 모두 같되(first가 널 포인터일 경우 false 반환, 정상적으로 삭제가 이뤄질 시 true 반환) 노드의 끝까지 이동하는 동작이 사라진다. 덕분에 Stack 클래스의 Delete() 함수는 LinkedList 클래스에서 구현한 동작에 비해 훨씬 간단하다.

먼저 Node<T> *current 포인터를 선언한 뒤 first 포인터를 가리키도록 하고, first 포인터가 first->link를 가리키도록 한다. 이후 current가 가리키는 노드를 삭제한 뒤 current_size의 크기를 1 뺀다.

이외엔 부모 클래스의 멤버 변수를 불러오는 것이기 때문에 this 포인터를 사용해야 한다는 점, 부모 클래스의 멤버 변수를 protected로 선언했는지 확인해야 한다는 점만 주의하면 된다.

```
#ifndef __LINKEDLIST__
#define __LINKEDLIST__

#include <iostream>
using namespace std;

// LinkedList Node
template <typename T>
class Node {
 public:
  T data;       // 데이터
  Node *link;  // 다음 노드의 주소 포인터
  Node(T element) : data(element), link(NULL) {}
};

// LinkedList Class
template <typename T>
class LinkedList {
 protected:
  Node<T> *first;     // 첫 번째 노드의 주소를 저장할 포인터
  int current_size;  // 배열 크기

 public:
  LinkedList() : first(NULL), current_size(0) {}  // 생성자
  int GetSize() { return current_size; };          // 노드 개수 리턴
  void Insert(T element);                           // 맨 앞에 원소를 삽입
  virtual bool Delete(T &element);                  // 맨 뒤의 원소 삭제
  void Print();                                     // 리스트 출력
};
```

```
// 새 노드를 앞에 삽입
template <typename T>
void LinkedList<T>::Insert(T element) {
  Node<T> *newnode = new Node<T>(element);  // 노드 메모리 할당
  newnode->link = first;                    // 첫 번째 노드에 삽입
  first = newnode;                          // 첫 번째 포인터 초기화
  current_size++;                           // 배열 크기 갱신
}

// 마지막 노드의 값을 리턴하면서 메모리에서 할당 해제
template <typename T>
bool LinkedList<T>::Delete(T &element) {
  if (first == 0) return false;  // 배열 비어 있을 경우 0 반환

  Node<T> *current = first;  // 첫 번째 노드를 불러옴
  Node<T> *previous = NULL;

  // 마지막 노드까지 탐색
  while (1) {
    if (current->link == NULL) {
      if (previous)  // 이전 노드가 존재하면 (노드가 1개가 아니면)
        previous->link = current->link;  // 이전 노드의 링크 NULL로
      else
        first = first->link;  // 존재하지 않을 경우 first를 NULL로
      break;
    }
    previous = current;
    current = current->link;
  }
  element = current->data;  // element 값 삭제된 노드의 데이터로 갱신
```

```
  delete current;              // 마지막 포인터 메모리 반환(=삭제)
  current_size--;              // 배열 크기 갱신

  return true;
}

// 리스트를 출력하는 Print 함수
template <typename T>
void LinkedList<T>::Print() {
  Node<T> *i;
  int index = 1;

  if (current_size != 0) {
    for (i = first; i != NULL; i = i->link) {
      if (index == current_size) {
        cout << "[" << index << "|" << i->data << "]->";
      } else {
        cout << "[" << index << "|" << i->data << "]->";
        index++;
      }
    }
    cout << endl;
  }
}

#endif
```

```
#include "LinkedList.h"
```

```
// 1. 템플릿 클래스로 확장해야함
// 2. Stack형식으로 Delete 함수 재정의해야함
// 주의: first, current_size는 class의 멤버 변수이기 때문에
// this 포인터를 사용하여 가져와야함

// LinkedList class를 상속받음
template <typename T>
class Stack : public LinkedList<T> {
 public:
  bool Delete(T &element) {
    // first가 0이면 false반환
    if (this->first == NULL) return false;

    element = this->first->data;      // 원소 값 저장
    Node<T> *current = this->first;  // current 포인터로 first 포인터 대체
    this->first = this->first->link;  // first 포인터 다음 링크로 이동

    delete current;          // current 포인터가 가리키는 노드 삭제
    this->current_size--;  // 크기 갱신
    // LinkedList와 달리 Stack은 current가 가리키는 곳을 삭제
    return true;
  }
};
```