2024 Spring OOP Assignment Report

과제 번호 : Assignment #6
학번 : 20230563
이름 : 김홍근
Povis ID : hongsimi7

명예서약 (Honor Code)

나는 이 프로그래밍 과제를 다른 사람의 부적절한 도움 없이 완수하였습니다.

I completed this programming task without the improper help of others.

1. 프로그램 개요

i) 문제1

- □ 본 프로그램은 Doubly Linked List를 template을 통해 구현하는 프로그램이다.
- □ Class template를 통해 여러 가지 타입을 정의하지 않고, 하나의 정의로 대부분의 타입을 선언할 수 있도록 한다.
- □ Doubly Linked List의 정의와 특징에 대해 알고, 이에 대한 구현을 실시한다.
- □ Iterator 및 Reverseiterator를 통해 더 유용한 함수 구현을 한다.
- □ Overloading을 통해, 다양한 연산자를 사용자 정의에 맞게 실행시키도록 한다.

ii) 문제2

- □ 본 프로그램은 문제1에서 제작한 Doubly Linked List를 통해 n단 논법을 적용하는 프로그램이다.
- □ 논리를 저장하고 연결하는 기능을 하며, 질문을 통해 논리를 출력할 수 있도록 한다.

2. 프로그램의 구조 및 알고리즘

i) 문제1

□ [Class MultiHeadList]

1. MultiHeadList 클래스는 다중 연결 리스트를 구현하기 위한 클래스이다. 각 Linked List의 Head는 벡터를 통해 headlist에 저장되는 구조이다. 따라서 여러 개의 독립적인 Linked List를 효율적으로 관리할 수 있으며, 다양한 기능을 적용할 수 있다. 또한,

Iterator class 및 ReverseIterator class가 정의되어 있다.

2. vector<Node<T>*> headList는 여러 개의 리스트 Head를 저장하는 벡터로 각 리스트의 Head를 접근하고 관리하는 데 사용된다.

3. int headSize()는 headList가 private이기 때문에 외부에서 접근하기 위해, 만들어진 메소드로 현재 리스트의 개수를 반환한다.

4. void push_back(const T& data, int headIdx = -1)는 매개변수로 data와 headIndex를 받으며, headIdx의 값은 디폴트로 -1로 지정되어 있다. 만약 headIdx가 음수거나, 범위에서 벗어났다면 새로운 head를 생성하여 저장한다. 그렇지 않다면, headIdx가 가리키는 리스트 끝에 새로운 노드를 생성하여 추가한다.

5. void push_front(const T& data, int headIdx = -1)는 매개변수로 data와 headIndex를 받으며, headIdx의 값은 디폴트로 -1로 지정되어 있다. 만약 headIdx가 음수거나, 범위에서 벗어났다면 새로운 head를 생성하여 저장한다. 그렇지 않다면, headIdx가 가리키는 리스트 맨 앞에 새로운 노드를 생성하여 추가한다.

6. void insert(Iterator pos, const T& data)는 매개변수로 data와 iterator pos를 받으며, pos가 지정한 위치 이전에 데이터를 삽입하는 메소드이다. 먼저 데이터를 담는 새로운 노드를 생성하고, prev와 next를 지정해준다. 그러고 pos 이전이 nullptr이 아닐 때, 삽입 노드 이전의 prev를 변경한다. Pos가 head라면, 몇 번째 head인지 찾은 다음, head를 변경한다. 그리고 pos의 prev를 변경하여 올바르게 삽입이 일어나도록 한다.

7. void pop_back(int headIdx)은 매개변수로 headIndex를 받고, 해당 인덱스의 리스트의 맨 뒤에서 데이터를 삭제하는 메소드이다. 삭제할 노드를 가리키는 리버스 iterator를 생성한 다음, pop대상이 head가 아닐 때와 head일 때를 나누어 정상적으로 작동하도록 제작했다.

8. void pop_front(int headIdx) 은 매개변수로 headIndex를 받고, 해당 인덱스의 리스트의 맨 앞에서 데이터를 삭제하는 메소드이다. 삭제할 노드를 가리키는 iterator를 생성한 다음, pop대상이 혼자 리스트에 있을 때와 아닐 때를 나누어 정상적으로 작동하도록 제작했다.

9. void merge(int frontHeadIdx, int backHeadIdx)은 매개변수로 앞에 올 headIndex와 뒤에 올 headIndex를 받고, 두 리스트를 병합하는 메소드이다. 앞에 올 리스트의 리버스 iterator과 뒤에 올 리스트의 iterator를 생성한다음, 알맞게 병합을 진행한다.

10. bool erase(const T& data, int targetHeadIdx)은 매개변수로 데이터와 삭제하고 싶은 데이터가 존재하는 리스트의 headIndex를 받고, 특정 값을 가진 노드를 삭제하는 메소드이다. 이 때, 리스트에서 주어진 값이 두 개 이상이라면, 가장 첫번째에 위치한 노

드를 삭제한다. 먼저 해당 리스트에 iterator를 생성하여, 데이터와 맞는 리스트 요소를 찾는다. 찾았다면, 그 노드의 이전 및 이후가 nullptr인지 확인한 다음, 아니라면 적절히 연결을 끊고 새로운 헤드를 만드는 작업 등을 실행한다. 만약 찾은 노드가 헤드라면, headList에서 삭제까지 진행을 한다.

11. bool erase(Iterator pos)는 iterator pos를 매개변수로 받아 직접 노드를 지정하여 삭제를 진행한다. 그 노드의 이전 및 이후가 nullptr인지 확인한 다음, 아니라면 적절히 연결을 끊고 새로운 헤드를 만드는 작업 등을 실행한다. 만약 해당 노드가 헤드라면, headList에서 삭제까지 진행을 한다.
12. Iterator begin(int headIdx)는 headIndex를 매개변수로 받고 해당 리스트의 head를 이터레이터 형태로 반환한다.
13. Iterator end()는 이터레이터의 nullptr 형태로 반환한다.
14. Iterator rbegin(int headIdx)는 headIndex를 매개변수로 받고 해당 리스트의 tail를 리버스 이터레이터 형태로 반환한다.
15. Iterator rend()는 리버스 이터레이터의 nullptr 형태로 반환한다.

□ [Class Iterator]

1. Iterator 클래스는 MutiHeadList 클래스 내에서 리스트를 순회하는 데 사용되는 이터레이터를 구현한 클래스이다. STL의 iterator와 유사한 기능을 한다.
2. Node<T>* curr는 현재 노드를 가리키는 포인터다.
3. Iterator(Node<T>* node)는 생성자로 curr 포인터를 주어진 node를 통해 초기화를 한다.
4. Iterator operator++()는 전위 증가 연산자로 이터레이터를 다음 노드로 이동시키고, 새로운 위치의 이터레이터를 반환한다.
5. Iterator operator++(int)는 후위 증가 연산자로 이터레이터를 다음 노드로 이동시키고, 이전 위치의 이터레이터를 반환한다.
6. Iterator operator--()는 전위 감소 연산자로 이터레이터를 이전 노드로 이동시키고, 새로운 위치의 이터레이터를 반환한다.
7. Iterator operator--(int)는 후위 감소 연산자로 이터레이터를 이전 노드로 이동시키고, 이전 위치의 이터레이터를 반환한다.
8. Iterator operator+(int n)는 n번째 다음 노드로 이동시키는 연산자이다. Operator++()를 이용하여 구현했다.

9. Iterator operator-(int n)는 n번째 이전 노드로 이동시키는 연산자이다. Operator—()를 이용하여 구현했다.

10. bool operator!=(const Iterator& other)는 두 이터레이터가 같은 노드를 가리키는지 비교하는 연산자로 bool 타입으로 반환을 한다.

11. bool operator==(const Iterator& other)는 두 이터레이터가 같은 노드를 가리키는지 비교하는 연산자로 bool 타입으로 반환을 한다.

12. T operator*()는 현재 노드의 데이터를 반환하는 연산자이다.

□ [Class ReverseIterator]

1. ReverseIterator 클래스는 MutiHeadList 클래스 내에서 리스트를 순회하는 데 사용되는 리버스 이터레이터를 구현한 클래스이다. STL의 ReverseIterator와 유사한 기능을 한다.

2. Node<T>* curr는 현재 노드를 가리키는 포인터다.

3. ReverseIterator(Node<T>* node)는 생성자로 curr 포인터를 주어진 node를 통해 초기화를 한다.

4. ReverseIterator operator++()는 전위 증가 연산자로 이터레이터를 이전 노드로 이동시키고, 새로운 위치의 이터레이터를 반환한다.

5. ReverseIterator operator++(int)는 후위 증가 연산자로 이터레이터를 이전 노드로 이동시키고, 이전 위치의 이터레이터를 반환한다.

6. ReverseIterator operator--()는 전위 감소 연산자로 이터레이터를 다음 노드로 이동시키고, 새로운 위치의 이터레이터를 반환한다.

7. ReverseIterator operator--(int)는 후위 감소 연산자로 이터레이터를 다음 노드로 이동시키고, 이전 위치의 이터레이터를 반환한다.

8. ReverseIterator operator+(int n)는 n번째 이전 노드로 이동시키는 연산자이다. Operator++()를 이용하여 구현했다.

9. ReverseIterator operator-(int n)는 n번째 다음 노드로 이동시키는 연산자이다. Operator—()를 이용하여 구현했다.

10. bool operator!=(const ReverseIterator & other)는 두 이터레이터가 같은 노드를 가리키는지 비교하는 연산자로 bool 타입으로 반환을 한다.

11. bool operator==(const ReverseIterator & other)는 두 이터레이터가 같은 노드를 가리키는지 비교하는 연산자로 bool 타입으로 반환을 한다.

12. T operator*()는 현재 노드의 데이터를 반환하는 연산자이다.

ii) 문제2

□ [class Syllogism]

1. Syllogism 클래스는 MultiHeadList 클래스를 이용하여 삼단 논법을 관리하고 처리하는 기능을 제공하는 클래스이다. 각 노드는 논리 쌍(pair)로 구성되고, 삼단 논법을 추가하고 질의하는 기능을 제공하는 클래스이다.

2. MultiHeadList<pair<string, string>> syl는 논리 쌍을 저장하는 다중 리스트로, 각 리스트는 하나의 노리 사슬로 구성되어, 삼단 논법의 전제와 결론을 저장하고 관리한다.

3. void put(const pair<string, string>& argument)은 매개변수로 새로운 논리 쌍을 받아 리스트에 추가하는 메소드이다. 먼저 기존 리스트의 앞이나 뒤에 논리 쌍을 삽입할 수 있는지 확인하고, 삽입이 가능할 때 리스트의 앞이나 뒤에 논리 쌍을 추가한다. 이후, 다른 리스트와 병합할 수 있는지 확인을 하고 할 수 있다면, 병합을 실시한다. 만약 삽입을 할 수 없을 때에는 새로운 리스트를 생성해 논리 쌍을 추가한다.

4. void qna(const string& q)는 매개변수로, 전제를 받고 이에 대한 결론을 출력하는 메소드이다. 이터레이터를 통해 주어진 전제가 있는 리스트를 찾고, 이에 리버스 이터레이터가 결론을 가리키게 하여 전제 및 결론을 출력하게 한다.

5. void print()는 현재 리스트에 저장된 모든 삼단 논법을 차례대로 출력하는 메소드이다.

3. 토론 및 개선

i) 문제1

□ Template를 통해 클래스를 정의하는 방법을 알 수 있었고, 이러한 방법을 사용한다면, 다양한 타입의 클래스를 반복하지 않고 하나만 작성하여 다룰 수 있다는 것을 알 수 있었다.

□ 연산자를 overloading하여 사용자 지정 타입에서 연산자에 알맞게 프로그램이 진행되도록 할 수 있다는 것을 알 수 있었다.

- Iterator의 종류에는 Iterator와 ReverseIterator가 있다는 것을 알 수 있었고, 필요에 따라 둘 중 하나를 사용하면 더 편리하다는 것을 알 수 있었다.
- 클래스 안에 클래스를 또 다시 정의할 수 있다는 것을 알 수 있었고, 접근 권한에 대해 더 자세히 알 수 있는 계기가 되었다.
- Doubly Linked List의 정의와 특징에 대해 알 수 있었고, 많은 포인터로 인해 조금 어려웠던 것 같다.
- 참조를 사용하지 않아, 데이터 누수가 발생한 것 같다. 참조에 대해 더 자세히 공부해볼 필요가 있다고 생각했다.

ii) 문제2

- 문제1에서 만들었던 MultiHeadList를 타입을 지정하여 직접 사용해보면서 확실히 template를 사용한 클래스가 유용하다는 것을 알 수 있었다.
- Put 메서드를 제작하면서 여러 가지 경우의 수를 나누고 줄이고 하다 보니 상당히 애를 먹었다. 지금 짠 코드보다 더 좋은 코드가 있을 거라 생각하여 더 많은 알고리즘에 대해 생각해보는 것도 실력 향상에 도움이 될 것이라 생각했다.

4. 참고 문헌

- 해당사항 없음.