<HW1 과제 보고서>

1. 과제 목표

: 스택과 큐 클래스 구현

2. 과제 진행 과정(stack)

2 - 1. 스택 필요 개념 정리

스택이란

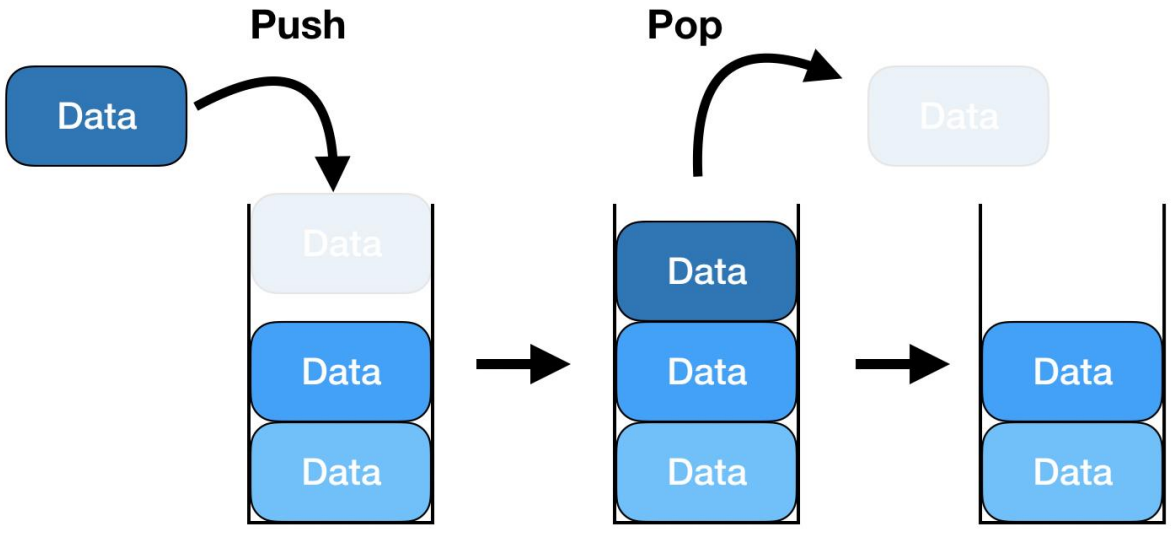
: 스택(Stack)은 컴퓨터 과학에서 매우 중요한 자료구조 중 하나다. 스택은 항목들이 쌓여 있는 구조를 뜻하며, LIFO(Last-In, First-Out) 원칙을 따른다. 이 말은, 가장 마지막에 스택에 추가된 항목이 가장 먼저 제거된다는 뜻이다.

스택의 장점

1. 간결성: 스택은 간단한 데이터 구조로서, 구현이 쉽고 이해하기도 쉽다. 이로 인해 코드를 간결하게 유지할 수 있다.
2. 순서 역술: 스택은 Last-In, First-Out (LIFO) 원칙을 따르기 때문에, 가장 마지막에 들어온 요소가 가장 먼저 나가는 방식을 자연스럽게 처리할 수 있다. 이는 특정한 종류의 문제, 예를 들어 실행 취소(undo) 기능, 괄호의 짝 맞추기 등에 유용하다.
3. 함수 호출 관리: 컴퓨터의 메모리 관리에서, 스택은 함수 호출과 관련된 정보(리턴 주소, 지역 변수 등)를 저장하는 데 사용된다. 이는 프로그램의 실행 흐름을 유지하고 관리하는 데 중요한 역할을 한다.
4. 깊이 우선 탐색(DFS): 스택은 트리나 그래프 데이터 구조의 깊이 우선 탐색에 이용된다. 스택을 이용하면 노드를 방문하는 순서를 쉽게 관리할 수 있다.
5. 표현식 평가 및 변환: 수학적 표현식의 평가나 변환에 있어서 스택은 중요한 역할을 한다. 중위, 전위, 후위 표현식의 변환 및 평가를 스택을 이용해 진행한다.
6. 메모리 효율성: 스택은 메모리를 효율적으로 사용합니다. 스택에서 데이터를 추가하거나 제거하는 연산은 일정한 시간(O(1))을 필요로 하며, 스택에 저장된 데이터는 연속적인 메모리 주소에 위치하므로 공간 효율성이 높다.

스택의 단점

1. 크기 제한: 일반적으로 스택의 크기는 고정되어 있습니다. 이는 스택이 사용하는 메모리를 미리 할당해야 하기 때문입니다. 스택이 꽉 차면, 더 이상의 요소를 추가할 수 없으며, 이를 스택 오버플로우라고 부릅니다. 이는 프로그램에서 심각한 오류를 발생시킬 수 있습니다.
2. 데이터 접근 및 검색: 스택은 LIFO(Last In, First Out) 원칙에 따라 동작하기 때문에, 가장 최근에 추가된 요소만이 직접 접근할 수 있습니다. 스택 내부의 중간 위치에 있는 요소에 접근하려면, 그 앞에 있는 요소들을 모두 제거해야 합니다. 이는 시간이 많이 소요될 수 있습니다. 또한, 특정 요소가 스택 내에 존재하는지 검색하는 것도 비효율적입니다.
3. 복잡한 사용: 스택이 재귀 함수나 반복문 등에 종종 사용되기 때문에, 이를 적절히 사용하기 위해선 재귀적 사고나 프로그램의 흐름을 잘 이해해야 합니다. 스택을 잘못 사용하면 프로그램의 로직을 이해하기 어렵게 만들 수 있습니다.



▲스택 자료구조 (출처 : https://velog.io/@hyhy9501/3-1.-%EC%8A%A4%ED%83%9DStack)

2 -2 예제 코드 분석(이전 c 언어 교육때 나왔던 스택 구조 참고)

연결리스트 형식의 구성

Node 구조체의 변수에는 data, next 가 존재하고, 각각은 노드가 저장하는 데이터 값과 다음 노드를 가리키는 포인터 값을 의미함.

생성자가 실행 될 때 data, next 라는 멤버 변수를 초기화함
소멸자는 메모리의 누수를 방지하기 위해 스택이 비어질 때까지 while 문을 통해 항목을 제거함.

push 함수 : 새로운 데이터를 스택의 최상단에 추가
pop : 최상단 노드를 제거하고 해당 데이터 반환
top:최상단 노드의 데이터 반환
isEmpty : 스택이 비어있는지 확인
getSize : 스택의 크기 반환

2 – 3 top, isEmpty, getSize 함수 제작

```
bool Stack::top() // 최상 노드 반환
{
    if (isEmpty())
    {
    // 스택이 비어있을 때 예외 처리
        throw std::out_of_range("Stack is empty, top is not exist.");
    }

    return topNode->data;
}


bool Stack::isEmpty() // stack 비어있는지 여부를 확인하는 함수
{
    // 스택이 비워져 있으면 true 반환 아닐경우 false 반환
    if (size == 0)
    {
        return true;
    }
    else
    {
        return false;
    }

}


int Stack::getSize() // 사이즈를 반환하는 함수
{
    return size;
}
```

▲top, isEmpty, getSize 함수 구현

2 - 3. printNodes 함수 제작

: 디버깅 과정중 코드의 구현이 제대로 이루어졌는지 확인하는데 어려움을 느끼고, 함수가 제대로 작동되고 있는지 확인하기 위해 제작하였다. printNodes 함수의 알고리즘은 빈 노드에 topNode 값을 저장하고 while 문을 통해 다음데이터를 순차적으로 가리키며 출력하게 된다.

```
void Stack::printNodes() // 현재 stack에 있는 값을 전체 출력하는 함수
{
    Node* tempNode = topNode; // 빈 노드에 topNode 값 저장
    while(tempNode != nullptr) // stack 끝까지 반복
    {
        std::cout << tempNode->data << " "; // 앞서 지정한 노드(tmepNode)가 가리키는 데이터 출력
        tempNode = tempNode->next; // 다음으로 넘어가기
    }

    std::cout << std::endl; // 줄바꿈
}
```

▲printfNodes 함수 구현

2. 과제 진행 과정 (queue)

2 – 1 queue 필요 개념 정리

: 스택과 마찬가지로 자료의 삽입과 삭제에 대한 규칙이 있는 자료구조 중 하나 가장 먼저 자료구조에 삽입된 데이터가 제일 처음에 삭제되고,이를 선입선출 (First In First Out)이라고 함.

큐의 장점

1. 구현이 간단하고, 메모리를 효율적으로 사용할 수 있다.

2. 접근 시간이 매우 빠르며, O(1)의 시간 복잡도를 갖는다.

3. 재귀 알고리즘 등에 활용할 수 있다.

4. 함수의 호출 스택, 웹 브라우저 방문 기록 등 다양한 분야에서 활용된다.

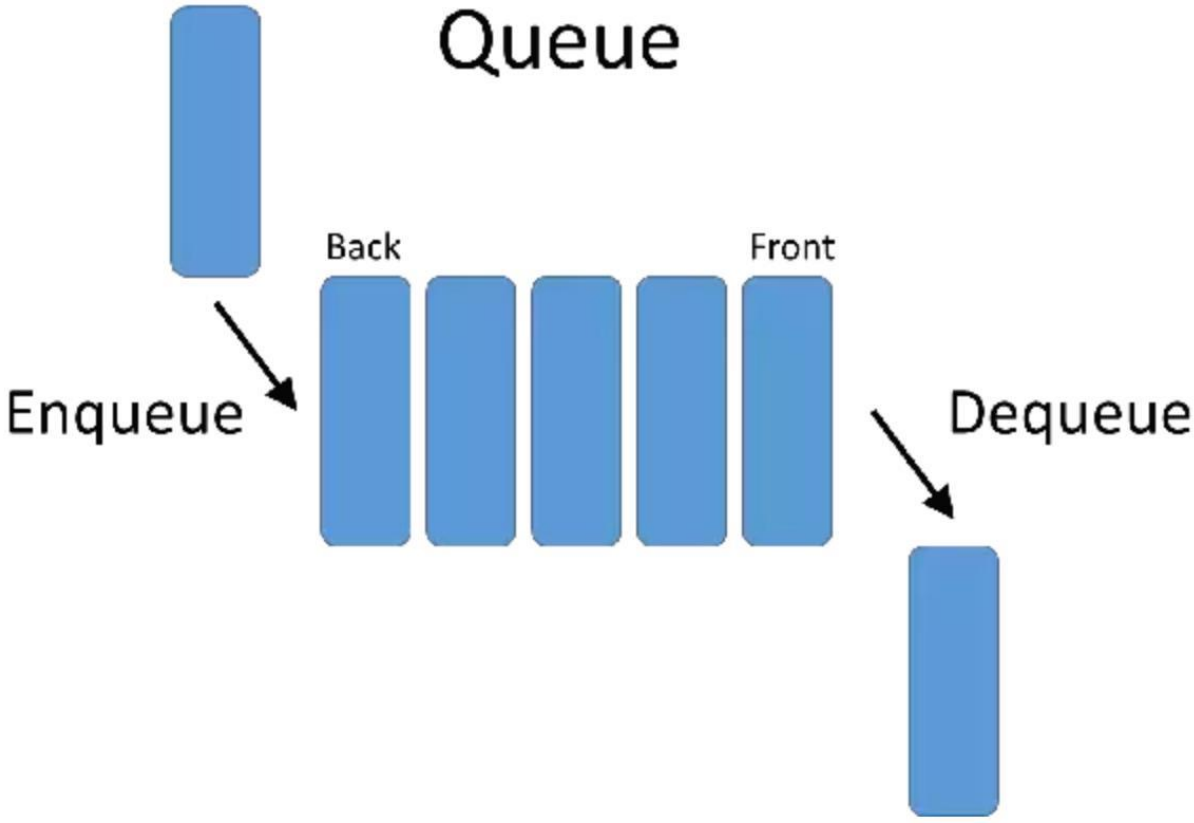
큐의 단점

1. 크기가 고정되어 있지 않아서 동적으로 메모리를 할당하는 경우에는 메모리를 할당해제하는 과정이 빈번해질 수 있어 비효율적이다.

2. 배열로 구현할 경우, 삽입/삭제 연산이 맨 앞에 위치할 때, 모든 원소의 위치를 옮겨줘야 하는 단점이 있다.

3. 이 경우, O(n)의 시간 복잡도를 가지게 되는데, 이를 해결하기 위해 링크드 리스트로 구현할 수 있다.

4. 후입선출(LIFO)의 특성 때문에, 선입선출(FIFO)이나 우선순위 큐와 같은 다른 자료구조와는 다르게, 특정 문제를 해결하기 어려울 수 있다.



▲큐 자료구조 (출처 : https://dev-lagom.tistory.com/28)

2 -2 예제 코드 분석(앞선 스택 구조 참고)

앞선 stack 과제의 구조체 변수, 연결 리스트 구조 등이 유사했음

enqueue : 큐의 뒤쪽 노드에 새로운 데이터 추가
dequeue : 가장 앞쪽 노드를 제거하고 해당 데이터 반환
front : 가장 앞쪽 노드 반환
isEmpty : 큐가 비어있는지 여부 판단
getSize : 큐의 크기를 반환

구현에 있어 stack 과제와의 가장 차이점은 topNode 만 다루면 되었던 stack 과제와 달리 queue 과제에서는 frontNode 와 rearNode 를 다루어야 했다는 점과 stack 과 queue 의 데이터 삭제 방향이 다르다는 점이었다

2 – 3 enqueue, dequeue, front, isEmpty, getsize, printfNodes 함수 구현

```
void Queue::enqueue(bool item) // 새로운 노드르 추가하는 함수
{
    Node* newNode = new Node(item);   // 새로운 노드 생성
    newNode->next = nullptr;

    if(isEmpty()) // 노드가 아무것도 없는 경우
    {
        rearNode = newNode;                // rearNode를 새 노드로 갱신
        frontNode = newNode;               // frontNode를 새 노드로 갱신
        size++;                            // 큐의 크기 증가
    }
    else
    {
        rearNode->next = newNode;          // 새 노드의 next를 기존 rearNode로 설정
        rearNode = newNode;                // rearNode를 새 노드로 갱신
        size++;                            // 큐의 크기 증가
    }
}
```

▲eneque 함수 구현

enequeue 함수를 만들 때에는 기존 노드에 노드가 아무것도 없는 경우와 이미 노드가 존재할 경우를 나누어 제작했다. 큐 구조상 rearNode 와 frontNode 를 둘다 다루어야 하기 때문에, 기존 노드에 아무것도 없는 경우 새로운 노드를 rearNode 와 frontNdoe 로 갱신해주어야 노드가 생성된 이후에는 rearNode 에만 새로운 노드를 갱신하는 식으로 알고리즘을 짤수 있기 때문이다.

```
bool Queue::dequeue()   // 노드 삭제하는 함수
{
    if (isEmpty())
    {
    // 큐가 비어있을 때 예외 처리
        throw std::out_of_range("Queue is empty, cannot dequeue.");
    }

    Node* tempNode = frontNode;        // 현재 frontNode 값 tempNode에 저장
    bool removedData = frontNode->data; // 현재 frontNode가 가리키는 데이터값 따로 저장
    frontNode = frontNode->next;     // forontNode를 새 노드로 갱신
    delete tempNode;                 // tempNode 메모리 할당 해제
    size--;                          //  크기 감소
    return removedData;               // 꺼낸 데이터 반환
}
```

▲dequeue 함수 구현

dequeue 함수의 구현 알고리즘은 frontNode 의 주소를 tempNode 라는 포인터에 저장하고, 현재 frontNode 가 가리키는 값을 removedData 에 따로 저장한다. (여기서 removedData 가 bool 형인 이유는 노드의 데이터가 bool 형이기 때문) 그 후에 큐의 다음 노드를 가리키도록 갱신한다. 여기까지 왔으면 frontNode 를 한칸 앞으로 이동시킨 것이다. 그 후에 frontNode 의 주소를 가리키는 tempNode 의 메모리할당을 해제해 메모리 누수를 방지한다.

```
bool Queue::front() // front 노드가 가리키는 데이터값을 반환 하는 함수
{
    if (isEmpty())
    {
    // 스택이 비어있을 때 예외 처리
        throw std::out_of_range("Queue is empty, top is not exist.");
    }

    return frontNode->data;

}

bool Queue::isEmpty() // stack 비어있는지 여부를 확인하는 함수
{
    // 스택이 비워져 있으면 true 반환 아닐경우 false 반환
    if (size == 0)
    {
        return true;
    }
    else
    {
        return false;
    }
}

int Queue::getSize()  // 사이즈를 반환하는 함수
{
    return size;
}
```

▲front, isEmpty, getSize 함수 구현

```
void Queue::printNodes() // 현재 stack에 있는 값을 전체 출력하는 함수
{
    Node* tempNode = frontNode; // 빈 노드에 topNode 값 저장
    while(tempNode != nullptr) // stack 끝까지 반복
    {
        std::cout << tempNode->data << " "; // 앞서 지정한 노드(tmepNode)가 가리키는 데이터 출력
        tempNode = tempNode->next; // 다음으로 넘어가기
    }

    std::cout << std::endl; // 줄바꿈
}
```

▲ printfNodes 함수 구현

배운점 및 고찰

: 우선 예시 사진에 이미 어느정도 구현이 되어있기도 하고, c 언어 교육때 한번 해봤던 과제여서 과제 자체가 어려운 난이도는 아니었지만 c++이라는 언어가 처음이기에 c++의 함수나 클래스 구조, 헤더파일을 다루는 방법이 미숙해 그런 부분에서 오히려 더 많은 어려움을 겪었던 것 같다. 또한 디버깅 과정에서 내가 짠 코드가 제대로 작동하는지 알지 못해 어려움을 겪었고, 이 부분은 스택과 큐에 있는 노드들을 출력하는 함수를 직접 짜는 방식으로 해결할 수 있었다. 과제를 끝내고 나서는 c++의 여러 함수들에 대해서 조금은 익숙해 진 것 같은 느낌을 받았고, 다음에는 시각화로 스택구조를 만들어 보고 싶다는 생각을 하게 되었다.