C++ Korea 자료구조 스터디

3주차 : 스택 / 큐

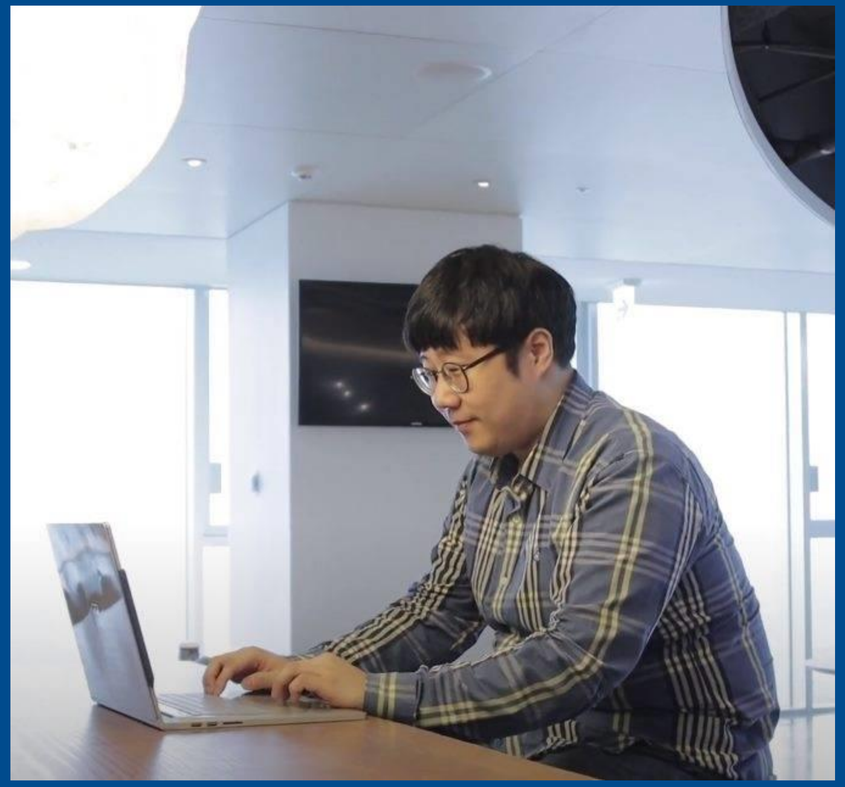
옥찬호

utilForever@gmail.com

강의자 소개

utilForever@gmail.com
utilForever

- 옥찬호 (Chris Ohk)
  - (현) Momenti Engine Engineer
  - (전) Nexon Korea Game Programmer
  - Microsoft Developer Technologies MVP
  - C++ Korea Founder & Administrator
  - Reinforcement Learning KR Administrator
  - IT 전문서 집필 및 번역 다수
    - 게임샐러드로 코드 한 줄 없이 게임 만들기 (2013)
    - 유니티 Shader와 Effect 제작 (2014)
    - 2D 게임 프로그래밍 (2014), 러스트 핵심 노트 (2017)
    - 모던 C++ 입문 (2017), C++ 최적화 (2019)



Part 1

스택 (Stack)

스택의 정의

- 스택(Stack)은 톱(Top)이라고 하는 곳을 통해
  모든 삽입(Push)과 삭제(Pop)가 일어나는 자료구조를 말한다.
- 스택을 $S = (a_0, \dots, a_{n-1})$이라고 정의할 때,
  a_0는 가장 아래쪽에 있는 원소, a_{n-1}는 가장 위쪽(Top)에 있는 원소다.
  - 만약 스택에 원소 A, B, C, D, E를 순서대로 삽입했다면, 제일 먼저 삭제되는 원소는 E
- 제일 마지막으로 삽입된 원소가 제일 먼저 삭제되기 때문에
  후입선출(LIFO : Last-In, First-Out) 자료구조라고도 한다.

스택 ADT


```
template <typename T>
class Stack
{ // A finite ordered list with zero or more elements
public:
    // Create an empty stack whose initial capacity is stackCapacity
    Stack(int stackCapacity = 10);

    // If number of elements in the stack is 0, return true else return false
    bool isEmpty() const;

    // Return top element of stack
    T& top() const;

    // Insert item into the top of the stack
    void push(const T& item);

    // Delete the top element of the stack
    void pop();
};
```

스택 ADT – 데이터 멤버 선언


```
private:
    // Array for stack elements
    T* stack;
    // Array position of top element
    int topPosition;
    // Capacity of stack array
    int capacity;
```

스택 ADT – 생성자


```
template <typename T>
Stack<T>::Stack(int stackCapacity) : capacity(stackCapacity)
{
    if (capacity < 1) throw std::runtime_error("Stack capacity must be > 0");
    stack = new T[capacity];
    topPosition = -1;
}
```

스택 ADT – push()


```
template <typename T>
void Stack<T>::push(const T& x)
{
    if (topPosition == capacity - 1)
    {
        ChangeSize1D(stack, capacity, 2 * capacity);
        capacity *= 2;
    }

    stack[++topPosition] = x;
}
```

스택 ADT – pop()


```
template <typename T>
void Stack<T>::pop()
{
    if (isEmpty()) throw std::runtime_error("Stack is empty. Cannot delete");
    stack[topPosition--].~T();
}
```

스택 ADT – isEmpty(), top()


```
template <typename T>
inline bool Stack<T>::isEmpty() const { return topPosition == -1; };

template <typename T>
inline T& Stack<T>::top() const
{
    if (isEmpty()) throw std::runtime_error("Stack is empty");
    return stack[topPosition];
}
```

Part 2

큐 (Queue)

큐의 정의

- 큐(Queue)는 한쪽 끝에서 삽입이 일어나고
  반대쪽 끝에서 삭제가 일어나는 자료구조를 말한다.
- 새로운 원소가 삽입되는 끝을 리어(Rear),
  원소가 삭제되는 끝을 프론트(Front)라고 한다.
  - 만약 큐에 원소 A, B, C, D, E를 순서대로 삽입했다면, 제일 먼저 삭제되는 원소는 A
- 제일 먼저 삽입된 원소가 제일 먼저 삭제되기 때문에
  선입선출(FIFO : First-In, First-Out) 자료구조라고도 한다.

큐 ADT


```
template <typename T>
class Queue
{ // A finite ordered list with zero or more elements
public:
    // Create an empty queue whose initial capacity is queueCapacity
    Queue(int queueCapacity = 10);

    // If number of elements in the queue is 0, return true else return false
    bool isEmpty() const;

    // Return the element at the front of the queue
    T& front() const;

    // Return the element at the rear of the queue
    T& rear() const;

    // Insert item at the rear of the queue
    void push(const T& item);

    // Delete the front element of the queue
    void pop();
};
```

큐 ADT – 데이터 멤버 선언


```
private:
    // Array for queue elements
    T* queue;
    // One counterclockwise from front
    int frontPos;
    // Array position of rear element
    int rearPos;
    // Capacity of queue array
    int capacity;
```

큐 ADT – 생성자


```
template <typename T>
Queue<T>::Queue(int queueCapacity) : capacity(queueCapacity)
{
    if (capacity < 1) throw std::runtime_error("Queue capacity must be > 0");
    queue = new T[capacity];
    frontPos = rearPos = 0;
}
```

큐 ADT – push()


```
template <typename T>
void Queue<T>::push(const T& x)
{
    // If Queue is full, double capacity
    if ((rearPos + 1) % capacity == frontPos)
    {
        // Allocate an array with twice the capacity
        T* newQueue = new T[2 * capacity];

        // Copy from queue to newQueue
        int start = (frontPos + 1) % capacity;
        if (start < 2)
        {
            // No warp around
            std::copy(queue + start, queue + start + capacity - 1,
                stdext::checked_array_iterator<T*>(newQueue, capacity - 1));
        }
```

큐 ADT – push()


```
        else
        {
            // Queue wraps around
            std::copy(queue + start, queue + capacity,
                stdext::checked_array_iterator<T*>(newQueue, capacity - start));
            std::copy(queue, queue + rearPos + 1, stdext::checked_array_iterator<T*>
                (newQueue + capacity - start, rearPos + 1));
        }
        // Switch to newQueue
        frontPos = 2 * capacity - 1;
        rearPos = capacity - 2;
        capacity *= 2;
        delete[] queue;
        queue = newQueue;
    }
    rearPos = (rearPos + 1) % capacity;
    queue[rearPos] = x;
}
```

큐 ADT – pop()


```
template <typename T>
void Queue<T>::pop()
{
    if (isEmpty()) throw std::runtime_error("Queue is empty. Cannot delete");
    frontPos = (frontPos + 1) % capacity;
    queue[frontPos].~T();
}
```

큐 ADT – isEmpty(), front(), rear()

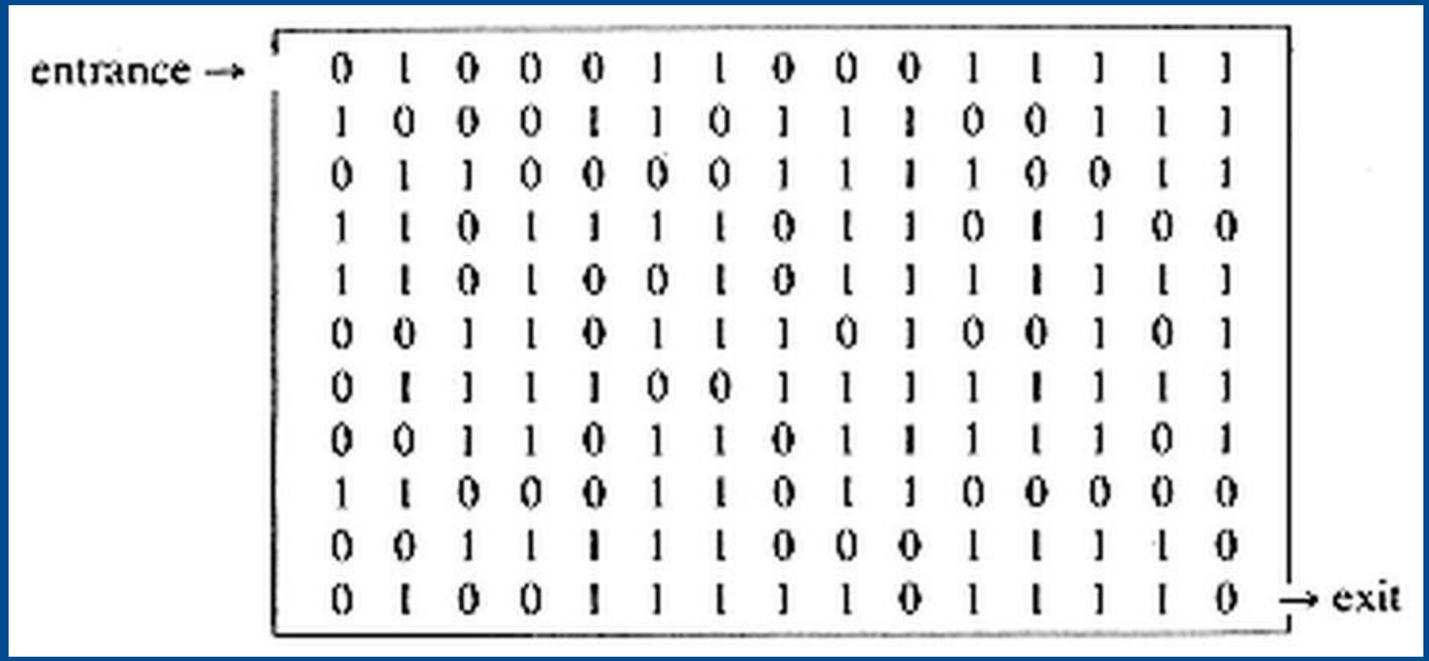

```
template <typename T>
inline bool Queue<T>::isEmpty() const { return frontPos == rearPos; }

template <typename T>
inline T& Queue<T>::front() const
{
    if (isEmpty()) throw std::runtime_error("Queue is empty. No front element");
    return queue[(frontPos + 1) % capacity];
}

template <typename T>
inline T& Queue<T>::rear() const
{
    if (isEmpty()) throw std::runtime_error("Queue is empty. No rear element");
    return queue[rearPos];
}
```

Part 3

미로 문제 (Maze)

미로의 정의

- $1 \leq i \leq m$이고 $1 \leq j \leq p$인 2차원 배열 $maze[i][j]$
  - 통로가 막혀서 통과할 수 없는 경우 1, 통과할 수 있는 경우 0
  - 입구는 $maze[1][1]$, 출구는 $maze[m][p]$라고 가정한다.



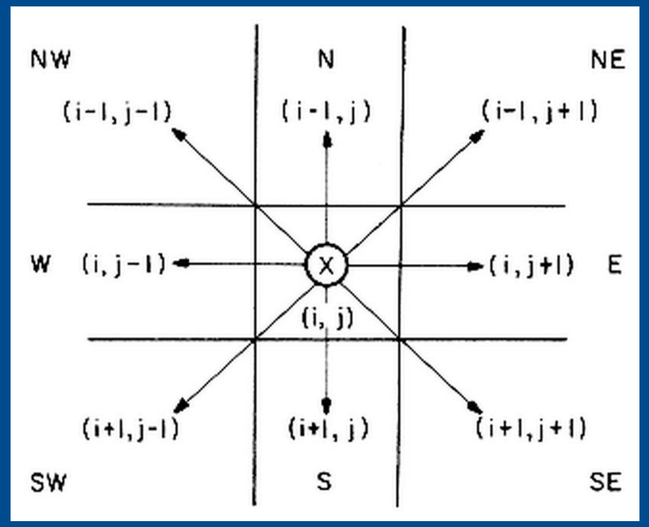
미로에서의 이동

- 총 8방향
  - 북(N), 북동(NE), 동(E), 남동(SE), 남(S), 남서(SW), 서(W), 북서(NW)
  - $maze[i][j]$가 $i = 1$ 또는 m, $j = 1$ 또는 p인 경계선에 있게 되면 3방향만 있을 수도 있다

    → 경계 조건을 매번 검사해야 하기 때문에 번거롭다.

    → 이를 피하고 싶다면, 미로 주위를 1로 둘러싸면 된다.

    → 배열을 $maze[m + 2][p + 2]$로 선언한다.



미로 – 데이터 타입


```
struct Offsets {
    Directions dx, dy;
};
using Offsets = struct Offsets;

enum class Directions { N, NE, E, SE, S, SW, W, NW };
Offsets move[8];

struct Items {
    int x, y
    Directions dir;
};
using Items = struct Items;
```

d	$move[d].dy$	$move[d].dx$
N	-1	0
NE	-1	1
E	0	1
SE	1	1
S	1	0
SW	1	-1
W	0	-1
NW	-1	-1

- 미로의 위치가 $[i][j]$일 때, 남서쪽 위치 $[g][h]$는?
  - $g = i + move[SW].dy$, $h = j + move[SW].dx$
  - 예를 들어, 위치 [3][4]에서 남서쪽 위치는 $[3 + 1 = 4][4 + (-1) = 3]$이다.

미로 – path()


```
void Maze::path()
{   // Print path if it is exist
    // math[0][i] = maze[m + 1][i] = maze[j][0] = maze[j][p + 1] = 1
    // 0 <= i <= p + 1, 0 <= j <= m + 1
    // Start at (1, 1)
    mark[1][1] = 1;
    Stack<Items> stack(width * height);
    Items temp(1, 1, Maze::Directions::E);
    // Set temp.x, temp.y, and temp.dir
    stack.push(temp);
```

미로 – path()


```
while (!stack.isEmpty())
{   // Stack is not empty
    temp = stack.top();
    stack.pop(); // Remove from stack
    int i = temp.x;
    int j = temp.y;
    Maze::Directions d = temp.dir;
    while (static_cast<int>(d) < 8) { // Move forward
        int g = i + move[static_cast<int>(d)].dy;
        int h = j + move[static_cast<int>(d)].dx;

        if ((g == width) && (h == height)) { // Arrive at exit
            // Print path
            std::cout << stack;
            std::cout << i << " " << j << std::endl;
            std::cout << width << " " << height << std::endl;
            return;
        }
```

미로 – path()


```
            if ((!maze[g][h]) && (!mark[g][h])) {
                mark[g][h] = 1;
                temp.x = i;
                temp.y = j;
                temp.dir = static_cast<Maze::Directions>(static_cast<int>(d) + 1);
                stack.push(temp); // Insert into stack
                i = g;
                j = h;
                d = Maze::Directions::N; // Move to (g, h)
            }
            else { // Try to next direction
                d = static_cast<Maze::Directions>(static_cast<int>(d) + 1);
            }
        }
    }

    std::cout << "No path in maze." << std::endl;
}
```

path 함수 분석

- 첫번째 while문의 반복 횟수는 미로의 크기에 따라 달라진다.
- 새로 방문하는 위치 $[i][j]$마다 표시를 해두기 때문에
  같은 경로를 두 번 선택하는 일은 없다.
- 표시된 각 위치에 대해 두번째 while문은 최대 8번 반복한다.
- 두번째 while문은 항상 상수 시간 $O(1)$ 동안 반복하며,
  $maze$에 있는 0의 개수를 z라 할 때, 최대 z개의 위치가 표시된다.
- z는 최대 mp이므로 시간 복잡도는 $O(mp)$

Part 4

수식의 계산 (Expression)

수식

- 연산의 수행 순서는 어떻게 파악해야 할까?

  → 각 연산자에 우선순위를 지정해야 한다.

우선순위	연산자
1	단항 -, !
2	*, /, %
3	+, -
4	<, <=, >=, >
5	==, !=
6	&&
7	||

중위 / 후위 표기법

- 다음과 같은 수식이 있다고 하자.

$$X = \Big(\big((A/B) - C\big) + (D * E)\Big) - (A * C)$$

- 중위 표기법 (Infix Notation) : 수식을 표현하는 일반적인 방법
  - $X = A/B - C + D * E - A * C$
- 후위 표기법 : 컴파일러가 수식을 읽고 해석할 때 사용하는 방법
  - $X = AB/C + DE * + AC * -$

중위 / 후위 표기법

- 컴파일러는 후위 표기법을 통해 어떻게 식을 계산할까?
  - 값을 하나 계산할 때마다 임시 장소 $T_i(i \geq 1)$에 저장해 놓는다고 가정하자.
  - 아래의 후위 표기식을 왼쪽에서부터 훑어 나가면 첫 연산자는 나눗셈이며
    이에 선행하는 두 피연산자는 A와 B다. 따라서 A/B의 결과가 T_1에 저장되고
    수식은 아래와 같이 변하게 된다. 이 과정을 계속 진행한 뒤, 전체 결과는 T_6에 저장된다.

연산	후위 표기식
$T_1 = A/B$	$T_1C - DE * + AC * -$
$T_2 = T_1 - C$	$T_2DE * + AC * -$
$T_3 = D * E$	$T_2T_3 + AC * -$
$T_4 = T_2 + T_3$	$T_4AC * -$
$T_5 = A * C$	$T_4T_5 -$
$T_6 = T_4 - T_5$	T_6

중위 표기 → 후위 표기 변환

- 중위 표기식을 후위 표기식으로 변환하는 알고리즘
  - (1) 식을 전부 괄호로 묶는다.
  - (2) 이항 연산자들을 모두 그들의 오른쪽 괄호로 이동시킨다.
  - (3) 괄호를 전부 삭제한다.

$$\Big(\big((A/B)\ -C\big)+(D*E)\Big)-(A*C)\Big)$$

$$AB/C-DE*+AC*-$$

수식의 계산 – postfix()


```
void Postfix()
{
    Stack<Token> stack;
    stack.push('#');

    for (Token x = nextToken(e); x != '#'; x = nextToken(e))
        if (x is an operand) std::cout << x;
        else if (x == ')')
        {
            for (; stack.top() != '('; stack.pop())
                std::cout << stack.top();
            stack.pop();
        }
```

수식의 계산 – postfix()


```
        else {
            for (; isp(stack.top()) <= icp(x); stack.pop())
                std::cout << stack.top();
            stack.push(x);
        }

    while (!stack.isEmpty()) {
        std::cout << stack.top();
        stack.pop();
    }
}
```

수식의 계산 – postfix()

- isp = in-stack priority

  icp = in-coming priority

우선순위	연산자	isp	icp
0	**(**	**8**	**0**
1	단항 -, !	1	1
2	*, /, %	2	2
3	+, -	3	3
4	<, <=, >=, >	4	4
5	==, !=	5	5
6	&&	6	6
7	||	7	7
8	**#**	**8**	**0**

postfix 함수 분석

- postfix 함수는 입력을 왼쪽에서 오른쪽으로 훑는다.
- 각 피연산자에 소요되는 시간은 $O(1)$이다.
- 각 연산자들은 스택에 한 번씩 삽입 및 삭제된다.
  따라서 각 연산자에 소요되는 시간도 $O(1)$이다.
- 그러므로 수식에서 토큰의 수를 n이라 할 때, 복잡도는 $\Theta(n)$이다.

감사합니다.

utilForever@gmail.com

https://github.com/utilForever

Facebook, Twitter: @utilForever