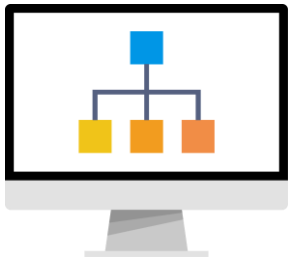


Data Structure & Algorithm
자료구조 및 알고리즘

10. 스택 (Stack, Chapter 6)

스택의 이해와 ADT 정의

스택(Stack)의 이해

- 스택은 '먼저 들어간 것이 나중에 나오는 자료구조'로써 초코볼이 담겨있는 통에 비유할 수 있다.
- 스택은 'LIFO(Last-in, First-out) 구조'의 자료구조이다.

- 초코볼 통에 초코볼을 넣는다. push
- 초코볼 통에서 초코볼을 꺼낸다. pop
- 이번에 꺼낼 초코볼의 색이 무엇인지 통 안을 들여다 본다. peek

} 스택의 기본 연산

일반적인 자료구조의 학습에서 스택의 이해와 구현은 어렵지 않다. 오히려 활용의 측면에서 생각할 것들이 많다!

큐(Queue)의 이해

큐는 'FIFO(First-in, First-out) 구조'의 자료구조이다.
때문에 먼저 들어간 것이 먼저 나오는, 일종의
줄서기에 비유할 수 있는 자료구조이다.

- 큐에 데이터를 넣는 연산 — enqueue
- 큐에서 데이터를 꺼내는 연산 — dequeue

} '큐'의 기본 연산

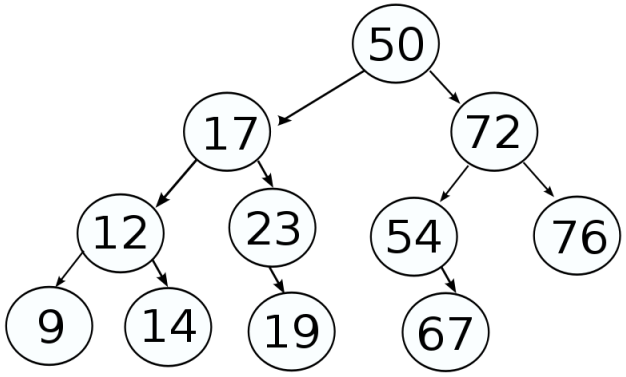
큐는 운영체제 관점에서 보면 프로세스나 쓰레드의 관리에 활용이 되는 자료구조이다. 이렇듯 운영체제의 구현에도 자료구조가 사용이 된다. 따라서 운영체제의 이해를 위해서는 자료구조에 대한 이해가 선행되어야 한다.

스택과 큐의 비교

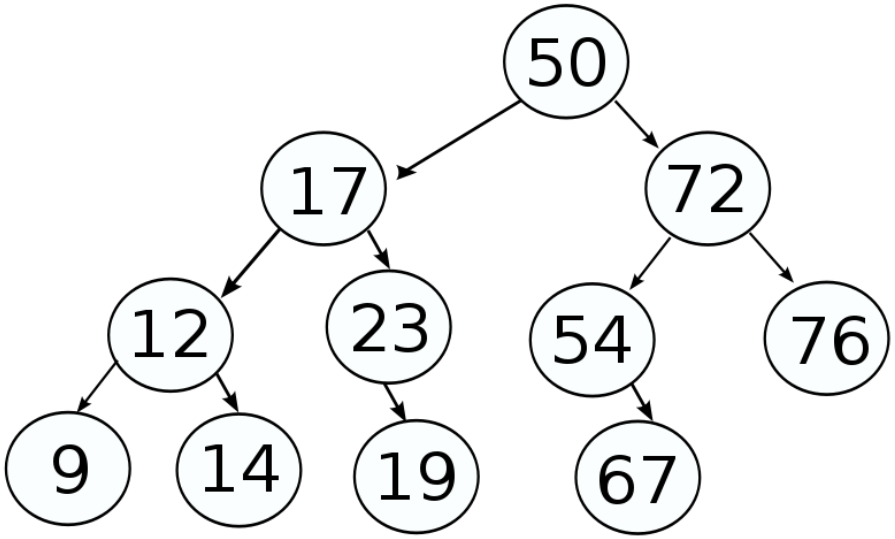
	스택 (Stack)	큐 (Queue)
비유	물건을 담는 통	줄 서기
구조	LIFO (Last In, First Out)	FIFO (First In, First Out)
삽입	push	enqueue
삭제	pop	dequeue

스택과 큐를 이용해서 트리 탐색하기

- 트리 탐색 알고리즘
  1. 자료구조에서 노드 v를 뺀다.
  2. v를 방문했다고 체크
  3. v의 자식 노드를 자료구조에 삽입
  4. 1로 돌아가 반복



스택과 큐를 이용해서 트리 탐색하기

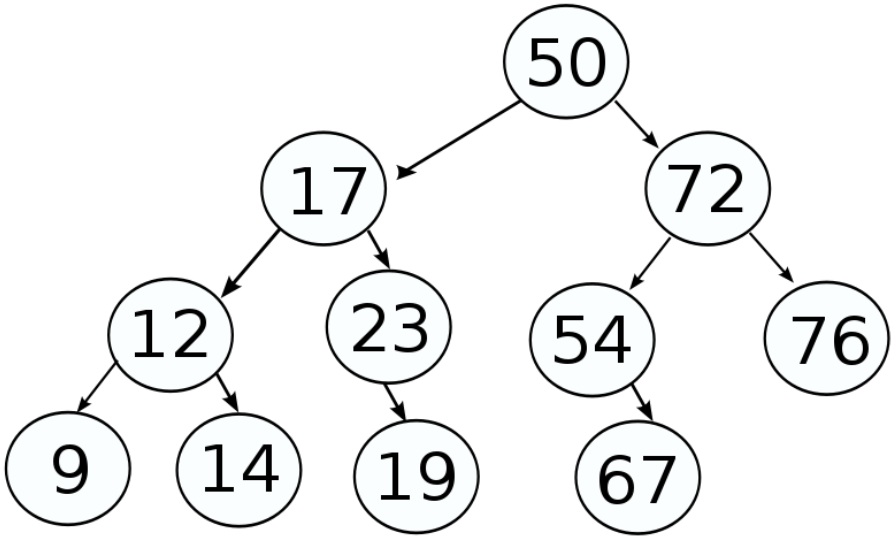


- 트리 탐색 알고리즘
  1. 자료구조에서 노드 v를 뺀다.
  2. v를 방문했다고 체크
  3. v의 자식 노드를 자료구조에 삽입
  4. 1로 돌아가 반복

스택 (3에서 자식을 역순으로 넣는다고 가정)
50, 17, 12, 9, 14, 23, 19, 72, 54, 67, 76

깊이 우선 탐색 = Depth First Search (DFS)

스택과 큐를 이용해서 트리 탐색하기



- 트리 탐색 알고리즘
  1. 자료구조에서 노드 v를 뺀다.
  2. v를 방문했다고 체크
  3. v의 자식 노드를 자료구조에 삽입
  4. 1로 돌아가 반복

큐
50, 17, 72, 12, 23, 54, 76, 9, 14, 19, 67

너비 우선 탐색 = Breadth First Search (BFS)

스택의 ADT 정의

- `void StackInit(Stack * pstack);`
  - 스택의 초기화를 진행한다.
  - 스택 생성 후 제일 먼저 호출되어야 하는 함수이다.

ADT를 대상으로 배열 기반의 스택
또는 연결 리스트 기반의 스택을 구현할 수 있다.

- `int SIsEmpty(Stack * pstack);`
  - 스택이 빈 경우 TRUE(1)을, 그렇지 않은 경우 FALSE(0)을 반환한다.

- `void SPush(Stack * pstack, Data data);` PUSH 연산
  - 스택에 데이터를 저장한다. 매개변수 data로 전달된 값을 저장한다.

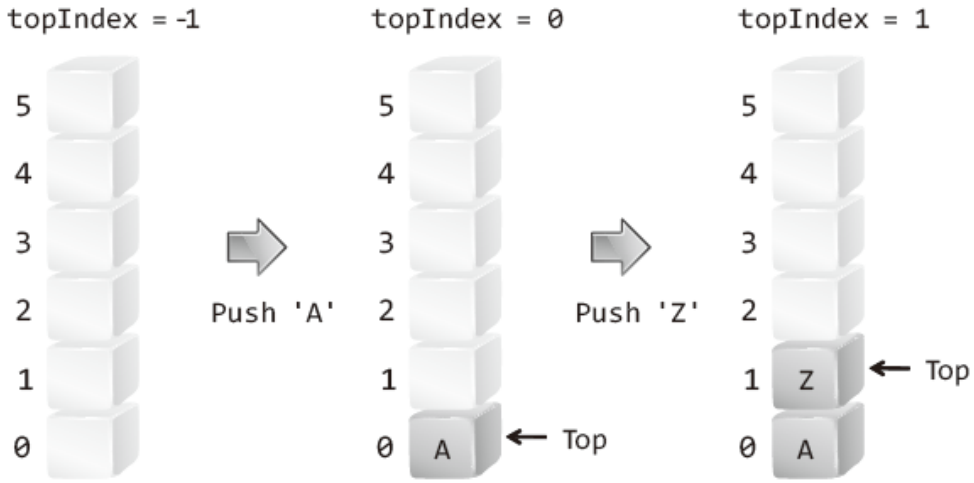
- `Data SPop(Stack * pstack);` POP 연산
  - 마지막에 저장된 요소를 삭제한다.
  - 삭제된 데이터는 반환이 된다.
  - 본 함수의 호출을 위해서는 데이터가 하나 이상 존재함이 보장되어야 한다.

- `Data SPeek(Stack * pstack);` PEEK 연산
  - 마지막에 저장된 요소를 반환하되 삭제하지 않는다.
  - 본 함수의 호출을 위해서는 데이터가 하나 이상 존재함이 보장되어야 한다.

스택의 배열 기반 구현

구현의 논리

두 번의 PUSH 연산



▶ [그림 06-1: 배열 기반 스택의 push 연산]

인덱스가 0인 위치를 스택의 바닥으로 정의해야 배열 길이에 상관없이 바닥의 인덱스 값이 동일해진다.

· 인덱스 0의 배열 요소가 '스택의 바닥(초코볼 통의 바닥)'으로 정의되었다.
· 마지막에 저장된 데이터의 위치를 기억해야 한다.

· push Top을 위로 한 칸 올리고, Top이 가리키는 위치에 데이터 저장
· pop Top이 가리키는 데이터를 반환하고, Top을 아래로 한 칸 내림

스택의 헤더파일

```
#define TRUE        1
#define FALSE       0
#define STACK_LEN   100

typedef int Data;

typedef struct _arrayStack
{
    Data stackArr[STACK_LEN];
    int topIndex;
} ArrayStack;

typedef ArrayStack Stack;

void StackInit(Stack * pstack);             // 스택의 초기화
int SIsEmpty(Stack * pstack);               // 스택이 비었는지 확인

void SPush(Stack * pstack, Data data);      // 스택의 push 연산
Data SPop(Stack * pstack);                  // 스택의 pop 연산
Data SPeek(Stack * pstack);                 // 스택의 peek 연산
```

배열 기반을 고려하여 정의된 스택의 구조체!

배열 기반 스택의 구현: 초기화 및 기타 함수

```
typedef struct _arrayStack
{
    Data stackArr[STACK_LEN];
    int topIndex;
} ArrayStack;
```

```
void StackInit(Stack * pstack)
{
    pstack->topIndex = -1;
}
```

-1은 스택이 비었음을 의미

```
int SIsEmpty(Stack * pstack)
{
    if(pstack->topIndex == -1)
        return TRUE;
    else
        return FALSE;
}
```

빈 경우 TRUE를 반환

```
int SIsEmpty(Stack * pstack)
{
    return pstack->topIndex == -1;
}
```

배열 기반 스택의 구현: PUSH, POP, PEEK

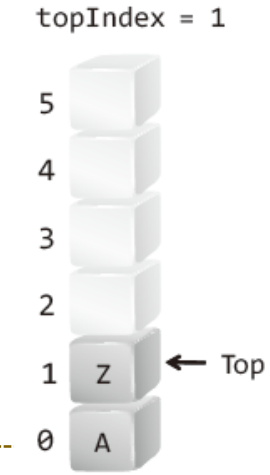
```
void SPush(Stack * pstack, Data data)
{
    pstack->topIndex += 1;
    pstack->stackArr[pstack->topIndex] = data;
}
```

```
Data SPop(Stack * pstack)
{
    int rIdx;

    if(SIsEmpty(pstack)) {
        printf("Stack Memory Error!");
        exit(-1);
    }

    rIdx = pstack->topIndex;
    pstack->topIndex -= 1;

    return pstack->stackArr[rIdx];
}
```




```
Data SPeek(Stack * pstack)
{
    if(SIsEmpty(pstack))
    {
        printf("Stack Memory Error!");
        exit(-1);
    }

    return pstack->stackArr[pstack->topIndex];
}
```

배열 기반 스택의 활용: main 함수

```
int main(void)
{
    // Stack의 생성 및 초기화 ///////
    Stack stack;
    StackInit(&stack);

    // 데이터 넣기 ///////
    SPush(&stack, 1); SPush(&stack, 2);
    SPush(&stack, 3); SPush(&stack, 4);
    SPush(&stack, 5);

    // 데이터 꺼내기 ///////
    while(!SIsEmpty(&stack))
        printf("%d ", SPop(&stack));

    return 0;
}
```

Arraybasestack.h
Arraybasestack.c
Arraybasestackmain.c

```
5 4 3 2 1
```

실행결과

스택의 연결 리스트 기반 구현

연결 리스트 기반 스택의 논리와 헤더파일의 정의

이렇듯 메모리 구조만 보아서는 스택임이 구분되지 않는다!

▶ [그림 06-2: 스택의 구현에 활용할 리스트 모델]

저장된 순서의 역순으로 데이터(노드)를

참조(삭제)하는 연결 리스트가 바로 연결 기반의

스택이다!

```
typedef int Data;

typedef struct _node
{
    Data data;
    struct _node * next;
} Node;

typedef struct _listStack
{
    Node * head;
} ListStack;
```

```
typedef ListStack Stack;

void StackInit(Stack * pstack);
int SIsEmpty(Stack * pstack);

void SPush(Stack * pstack, Data data);
Data SPop(Stack * pstack);
Data SPeek(Stack * pstack);
```

연결 리스트 기반 스택의 구현 1

```
void StackInit(Stack * pstack)
{
     pstack->head = NULL;
}
```

```
int SIsEmpty(Stack * pstack)
{
     if(pstack->head == NULL)
         return TRUE;
     else
         return FALSE;
}
```

```
Data SPop(Stack * pstack)
{
     Data rdata;
     Node * rnode;

     if(SIsEmpty(pstack)) {
         printf("Stack Memory Error!");
         exit(-1);
     }

     rdata = pstack->head->data;
     rnode = pstack->head;

     pstack->head = pstack->head->next;
     free(rnode);
     return rdata;
}
```

```
int SIsEmpty(Stack * pstack)
{
     return !pstack->head;
}
```

새 노드를 머리에 추가하고, 삭제 시 머리부터 삭제하는 단순 연결 리스트의 코드에 지나지 않는다.

연결 리스트 기반 스택의 구현 2

```
void SPush(Stack * pstack, Data data)
{
    Node * newNode = (Node*)malloc(sizeof(Node));

    newNode->data = data;
    newNode->next = pstack->head;

    pstack->head = newNode;
}
```

```
Data SPeek(Stack * pstack)
{
    if(SIsEmpty(pstack)) {
        printf("Stack Memory Error!");
        exit(-1);
    }

    return pstack->head->data;
}
```

연결 기반 스택의 활용: main 함수

```
int main(void)
{
    // Stack의 생성 및 초기화 ///////
    Stack stack;
    StackInit(&stack);

    // 데이터 넣기 ///////
    SPush(&stack, 1); SPush(&stack, 2);
    SPush(&stack, 3); SPush(&stack, 4);
    SPush(&stack, 5);

    // 데이터 꺼내기 ///////
    while(!SIsEmpty(&stack))
        printf("%d ", SPop(&stack));

    return 0;
}
```

배열 기반 리스트 관련 main 함수와 완전히 동일하게 정의된 main 함수!

ListBaseStack.h
ListBaseStack.c
ListBaseStackMain.c

```
5 4 3 2 1
```

실행결과

요약

- 스택(stack)은 나중에 들어간 데이터가 먼저 나오는 구조(LIFO)로 push, pop, peek 인터페이스를 가진다.
- 큐(queue)는 먼저 들어간 데이터가 먼저 나오는 구조(FIFO)로 enqueue, dequeue 인터페이스를 가진다.
- 배열 하나와 스택의 최 상단 인덱스를 저장하는 변수로 간단하게 스택을 구현할 수 있다 (실제로도 많이 쓰임).
- 스택은 단순 연결리스트에서 머리 노드서만 삽입/삭제가 발생하는 구조로 볼 수 있다.

출석 인정을 위한 보고서 작성

- A4 반 장 이상으로 아래 질문에 답한 후 포털에 있는 과제 제출 란에 PDF로 제출

- 스택의 push, pop, peek, isEmpty 연산의 시간 복잡도는 얼마일까?

- 스택 두 개로 큐를 만들 수 있을까? 가능하다면 어떻게 하면 될까?