stack

Data Structures and Algorithms

목차

- 스택의 이해와 ADT 정의
- 스택의 배열 기반 구현
- 계산기 프로그램 구현

스택의 이해와 ADT 정의

스택(Stack)의 이해

- 스택은 '먼저 들어간 것이 나중에 나오는 자료구조'
  - 'LIFO(Last-in, First-out) 구조'의 자료구조

스택의 기본 연산
- Push
- Pop

옷을 장에 넣는다 (push)
옷을 서랍에서 꺼낸다(pop)

스택의 ADT 정의

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

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

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

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

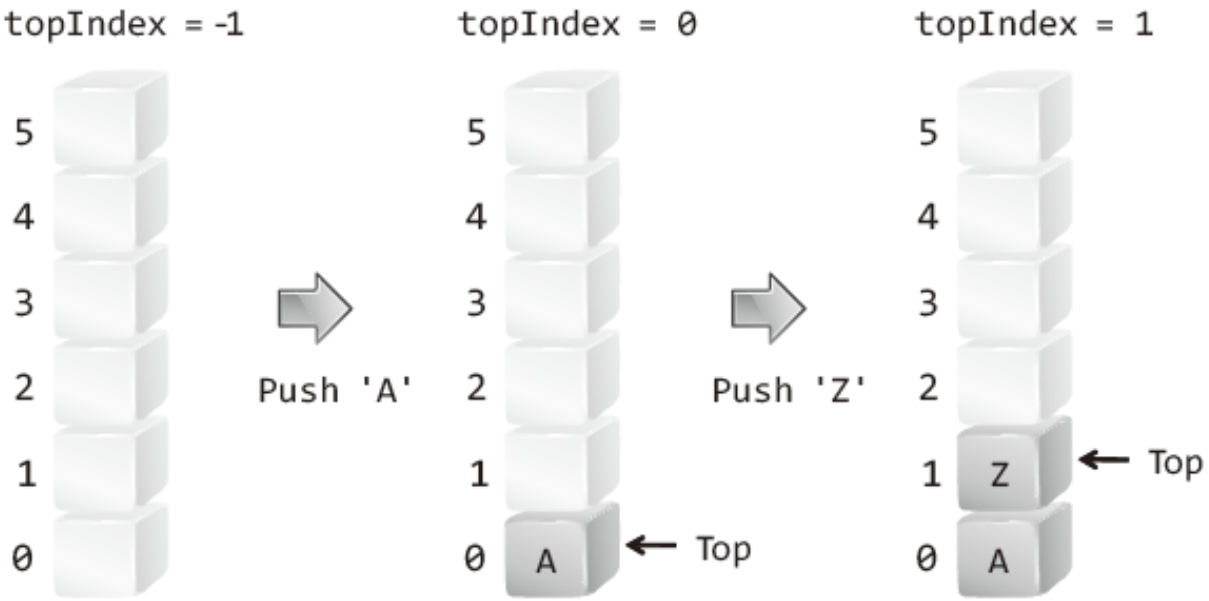
- `Data SPeek(Stack * pstack);`
  - 마지막에 저장된 요소를 반환하되 삭제하지 않는다.
  - 본 함수의 호출을 위해서는 데이터가 하나 이상 존재함이 보장되어야 한다.

구현 방법
1. 배열
2. 연결리스트

스택의 배열 기반 구현

구현의 논리

- 배열의 0번 인덱스가 스택의 바닥으로 정의
  - 마지막 저장된 데이터의 인덱스만 기억하면 됨
- Push: Top을 하나 증가, Top이 가리키는 위치에 데이터 저장
- Pop: Top이 가리키는 데이터를 반환하고, Top을 하나 감소



▶ [그림 06-1: 배열 기반 스택의 push 연산]

스택의 헤더파일

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

typedef int Data;

typedef struct _arrayStack
{   // 배열을 기준으로 정의된 스택 구조체
    Data stackArr[STACK_LEN];
    int topIndex;
} ArrayStack;

typedef ArrayStack Stack;

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

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

배열 기반 스택의 구현: 초기화 및 기타 함수

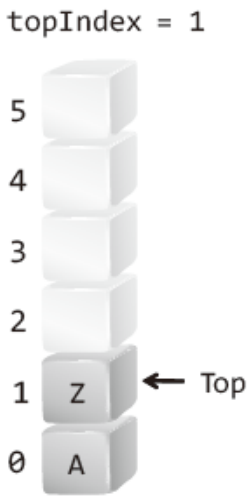
```
void StackInit(Stack * pstack)
{   // -1 은 비어 있음을 뜻함
    pstack->topIndex = -1;
}


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

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

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];
}
```



스택의 연결 리스트 기반 구현

연결 리스트 기반 스택의 논리와 헤더파일의 정의 ListBaseStack.h

- 메모리 구조만으로는 스택과 연결리스트 구분 불가
  - 핵심: 저장된 순서의 역순으로 데이터(노드)를 참조(삭제)

▶ [그림 06-2: 스택의 구현에 활용할 리스트 모델]

```
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

- 새 노드를 머리에 추가
- 삭제 시 머리부터 삭제

```
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;
}
```

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;
}
```

스택 자체의 구현 보다
스택을 활용하는 것이 더 중요

계산기 프로그램 구현

구현할 계산기 프로그래금의 요구명세

- 괄호와 사칙연산이 함께 있는 식을 계산
  - 괄호 내의 식을 먼저 연산
  - 연산자의 우선순위에 따라 연산

(3 + 4) * (5 / 2) + (7 + (9 – 5))

수식의 세 가지 표기법

- 중위 표기법(infix notation) 예) 5 + 2 / 7
- 수식 내에 연산 순서 정보가 없음
- 괄호와 연산자의 우선순위를 정의하여 순서를 따라야 함

- 전위 표기법(prefix notation) 예) + 5 / 2 7
- 수식 내에 연산의 순서가 있음
- 괄호와 연산 우선순위 결정 필요 없음

- 후위 표기법(postfix notation) 예) 5 2 7 / +
- 수식 내에 연산의 순서가 있음
- 괄호와 연산 우선순위 결정 필요 없음

중위 → 후위 : 소괄호 고려하지 않고 1

- 수식의 왼쪽부터 하나씩 처리

▶ [그림 06-3: 수식 변환의 과정 1/7]

- 피 연산자는 무조건 변환된 수식이 위치할 자리로 이동

▶ [그림 06-4: 수식 변환의 과정 2/7]

중위 → 후위 : 소괄호 고려하지 않고 2

- 연산자는 무조건 쟁반으로 push

▶ [그림 06-5: 수식 변환의 과정 3/7]

- 숫자는 변환된 수식이 위치할 자리로 이동

▶ [그림 06-6: 수식 변환의 과정 4/7]

중위 → 후위 : 소괄호 고려하지 않고 3

- / 연산자의 우선순위가 높으므로 + 연산자 위로 push

▶ [그림 06-7: 수식 변환의 과정 5/7]

- 쟁반(스택)에 기존 연산자가 있는 경우의 동작
  - 쟁반의 연산자의 우선순위가 높다면
    - 쟁반의 연산자를 변환된 수식이 위치할 자리로 이동 (pop)
    - 새 연산자는 쟁반으로 push
  - 쟁반에 위치한 연산자의 우선순위가 낮다면
    - 쟁반의 연산자의 위에 새 연산자를 push

중위 → 후위 : 소괄호 고려하지 않고 4

- 피 연산자는 무조건 변환된 수식의 자리로 이동!

▶ [그림 06-8: 수식 변환의 과정 6/7]

- 나머지 연산자들은 쟁반(스택)에서 차례로 이동 (pop)

▶ [그림 06-9: 수식 변환의 과정 7/7]

중위 → 후위 : 정리

- 피 연산자는 이동
- 연산자는 쟁반(스택)으로 이동(push)
- 쟁반에 있는 연산자의 우선순위에 따라 처리방법 결정
- 마지막까지 쟁반에 남은 연산자들은 하나씩 pop

중위 → 후위 : 고민 될 수 있는 상황

- 같은 우선순위의 연산자가 들어 온 경우
  - 먼저 들어온 연산자를 pop
- 케이스 1

▶ [그림 06-10: 우선순위가 같은 경우]

밀어내기

- 케이스 2

▶ [그림 06-12: 둘 이상의 연산자가 쌓여 있는 경우]

둘다 밀어내기

중위 → 후위 : 괄호 고려 1

▶ [그림 06-14: 소괄호가 포함된 수식의 변환 1/6]

▶ [그림 06-15: 소괄호가 포함된 수식의 변환 2/6]

중위 → 후위 : 소괄호 고려 2

▶ [그림 06-16: 소괄호가 포함된 수식의 변환 3/6]

▶ [그림 06-17: 소괄호가 포함된 수식의 변환 4/6]

중위 → 후위 : 소괄호 고려 3

-) 연산자를 만나면 (연산자까지 연산자 pop

▶ [그림 06-18: 소괄호가 포함된 수식의 변환 5/6]

▶ [그림 06-19: 소괄호가 포함된 수식의 변환 6/6]

중위 → 후위 : 프로그램 구현 1

- ConvToRPNExp(exp)의 RPN은 후위 표기법 Reverse Polish Notation의 약자
  - 중위 표기법 수식을 배열에 담아 함수의 인자로 전달
  - 계산 결과가 exp에 저장

```
void ConvToRPNExp(char exp[]);

int main(void)
{
    char exp1[] = "1+2*3";

    ConvToRPNExp(exp1);

    printf("%s \n", exp1);
    return 0;
}
```

중위 → 후위 : 프로그램 구현 2

-) 연산자는 괄호의 끝이므로 쟁반(스택)에 push 할 필요 없음
-) 연산자에 대한 반환 값 정의 불필요

```
// 함수 ConvToRPNExp의 첫 번째 helper function
int GetOpPrec(char op)
{   // 연산자의 연산 우선순위 정보를 반환
    switch(op)
    {
        case '*' :
        case  '/' :    // 값이 클수록 우선순위가 높음
            return 5;  // 가장 높은 연산의 우선순위
        case '+' :
        case '-' :
            return 3;  // 5보다 작고 1보다 높은 연산의 우선순위
        case '( ' :    // )연산자 전까지 스택에 있어야 함
            return 1;  // 가장 낮은 연산 우선순위
    }

    return -1;   // 등록되지 않은 연산자
}
```

중위 → 후위 : 프로그램 구현 3

- ConvToRPNExp 함수의 실질적인 Helper Function

```
// 함수 ConvToRPNExp의 두 번째 helper function
// 두 연산자의 우선순위 비교 결과를 반환
int WhoPrecOp(char op1, char op2)
{
    int op1Prec = GetOpPrec(op1);
    int op2Prec = GetOpPrec(op2);

    if(op1Prec > op2Prec)       // op1의 우선순위가 더 높다면
        return 1;
    else if(op1Prec < op2Prec)  // op2의 우선순위가 더 높다면
        return -1;
    else
        return 0;               // op1과 op2의 우선순위가 같다면
}
```

중위 → 후위 : 프로그램 구현 4

```
void ConvToRPNExp(char exp[])
{
     Stack stack;
     int expLen = strlen(exp);
     char * convExp = (char*)malloc(expLen+1); // 변환된 수식을 담을 공간 마련

     int i, idx=0;
     char tok, popOp;

     memset(convExp, 0, sizeof(char)*expLen+1); // 마련한 공간 0으로 초기화
     StackInit(&stack);

     for(i=0; i<expLen; i++)  {
          . . . . // 일련의 변환 과정을 이 반복문 안에서 수행
     }

     while(!SIsEmpty(&stack))
          convExp[idx++] = SPop(&stack); // 스택의 모든 연산자 pop용 반복문

     strcpy(exp, convExp); // 변환된 수식을 반환!
     free(convExp);
}
```

중위 → 후위 : 프로그램 구현 5

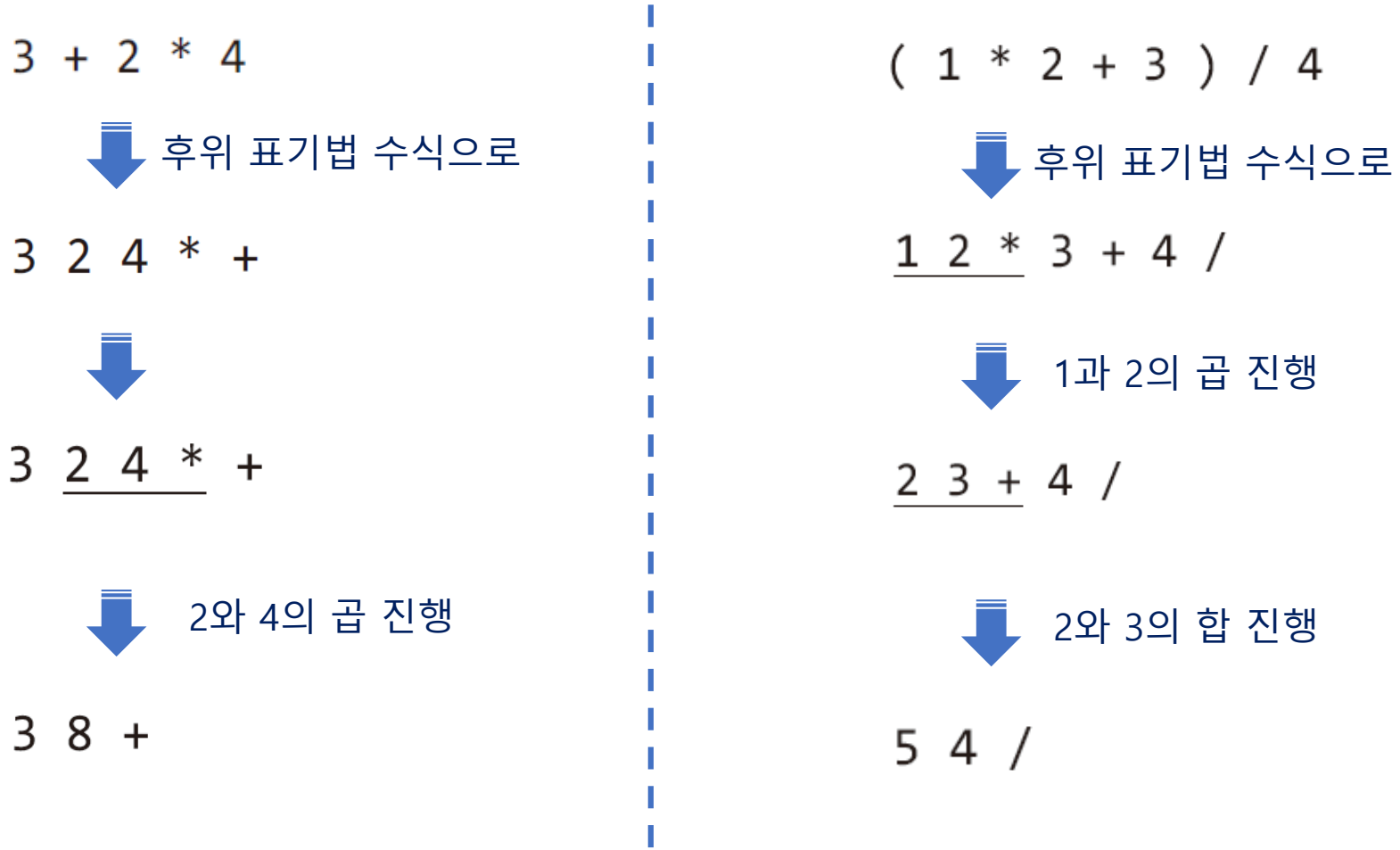
```
void ConvToRPNExp(char exp[])
{
      . . . .

      for(i=0; i<expLen; i++)
      {
            tok = exp[i];

            if(isdigit(tok)) // tok에 저장된 문자가 피연산자라면

            {
                  convExp[idx++] = tok;
            }
            else                    // tok에 저장된 문자가 연산자라면
            {
                  switch(tok)
                  {
                        . . . . // 연산자일 때의 처리 루틴을 switch문에 담는다!
                  }
            }
      }
}
```

중위 → 후위 : 프로그램 구현 6

```
switch(tok) // 함수 ConvToRPNExp의 일부인 switch문
{
    case '(':                  // 여는 괄호라면
        SPush(&stack, tok); // 스택에 push
        break;
    case ')':                  // 닫는 괄호라면
       while(1) // 반복
       {
          popOp = SPop(&stack); // 스택에서 연산자를 꺼내어
          if(popOp == '( ' )     // 연산자 (을 만날 때까지
              break;
          convExp[idx++] = popOp; // 배열 convExp에 저장
       }
       break;
       case '+': case '-': case '*': case '/ ' :
       // tok에 저장된 연산자를 스택에 저장하기 위한 과정
       while(!SIsEmpty(&stack) && WhoPrecOp(SPeek(&stack), tok) >= 0)
           convExp[idx++] = SPop(&stack);
       SPush(&stack, tok);
       break;
}
```

후기 표기법 수식의 계산

- 피연산자 두 개가 연산자 앞에 항상 위치하는 구조



후기 표기법 수식 계산 프로그램의 구현

- 계산의 규칙
  - 피연산자는 무조건 스택에 push
  - 연산자를 만나면 스택의 두 개의 피연산자를 pop하여 계산
  - 계산결과는 다시 스택에 넣음

▶ [그림 06-20: 후위 표기법의 수식 계산 1]

▶ [그림 06-21 : 후위 표기법의 수식 계산 2]

후기 표기법 수식 계산 프로그램의 구현

```
int EvalRPNExp(char exp[])
{
    Stack stack;
    int expLen = strlen(exp);
    int i;
    char tok, op1, op2;

    StackInit(&stack);

    for(i=0; i<expLen; i++)
    {
        tok = exp[i];

        if(isdigit(tok))
        {   // 숫자로 변환하여 PUSH!
            SPush(&stack, tok - '0');
        else
        {   // 먼저 꺼낸 값이 두 번째 피연산자!
            op2 = SPop(&stack);
            op1 = SPop(&stack);
            switch(tok)
            {
                case '+' :
                    SPush(&stack, op1+op2);
                    break;
                case '-' :
                    SPush(&stack, op1-op2);
                    break;
                case '*' :
                    SPush(&stack, op1*op2);
                    break;
                case '/' :
                    SPush(&stack, op1/op2);
                    break;
            }
        }
    }
    return SPop(&stack);
}
```

계산기 프로그램의 완성

- 계산 과정
  - 중위 표기법 수식 → ConvToRPNExp → EvalRPNExp → 연산결과
- 계산기 프로그램의 파일 구성
  - 스택
  - 중위 표기 수식을 후위 표기법으로 변환
  - 후위 표기법 수식 계산