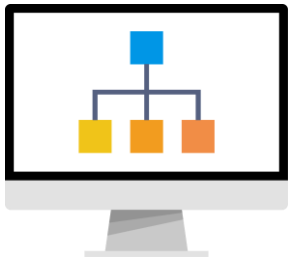


Data Structure & Algorithm
자료구조 및 알고리즘

14. 트리 (Tree, Part 2)

12강 보고서 돌아보기

- 큐의 enqueue, dequeue, peek, isEmpty 연산의 시간 복잡도는 얼마일까?

- 큐 두 개로 스택을 만들 수 있을까? 가능하다면 어떻게 하면 될까?

12강 보고서 돌아보기

- 큐의 enqueue, dequeue, peek, isEmpty 연산의 시간 복잡도는 얼마일까?

- 큐 두 개로 스택을 만들 수 있을까? 가능하다면 어떻게 하면 될까?

재귀 호출 및 이분탐색 연습 - 1

4개의 알파벳 abcd 중 서로 다른 알파벳 2개를 순서 없이 고르는 가지수는 (a, b), (a, c), (a, d), (b, c), (b, d), (c, d)의 6가지이다.

이처럼 n개의 물건 중에서 r개의 물건을 고르는 방법의 가지수를 조합 수라고 부르며 ${}_nC_r$로 정의한다.

임의의 n에 대해 ${}_nC_0 = {}_nC_n = 1$ 이며 아래 등식이 성립한다:

$${}_nC_r = {}_{n-1}C_{r-1} + {}_{n-1}C_r$$

두 양수 n과 r이 주어질 때 ${}_nC_r$을 재귀 호출을 통해 구하여 출력하시오.

```c
#include <stdio.h>

int ncr(int n, int r) {
    if(r == 0) return 1;
    if(n == r) return 1;

    return ncr(n-1, r-1) + ncr(n-1, r);
}

int main(){
    int n, r;
    scanf("%d %d", &n, &r);
    printf("%d\n", ncr(n, r));
    return 0;
}
```

재귀 호출 및 이분탐색 연습 - 2

2개의 알파벳 a와 b를 사용해서 길이가 3인 문자열을 만들어보면 아래의 8개이다.

aaa
aab
aba
abb
baa
bab
bba
bbb

16 이하의 문자열의 길이 n이 주어질 때 a와 b를 이용하여 길이 n의 문자열을 만드는 가지수를 사전순으로 (위 예제 처럼) 한 줄에 하나씩 출력하시오.

재귀 호출 및 이분탐색 연습 - 2

```c
#include <stdio.h>

int n;
char str[20];

void fill(int i) {
    if(i == n) {
        str[i] = '\0';
        printf("%s\n", str);
        return;
    }
    str[i] = 'a';
    fill(i + 1);

    str[i] = 'b';
    fill(i + 1);
}

int main() {
    scanf("%d", &n);
    fill(0);
    return 0;
}
```

재귀 호출 및 이분탐색 연습 - 3

이분 탐색을 구현하시오.

입력의 첫 줄에는 배열의 길이 n이 주어진다. n은 10만 이하의 정수이다.
입력의 두 번째 줄에는 배열에 있는 n개의 정수가 띄어쓰기로 구분되어 주어진다. 배열은 오름차순으로 정렬되어 있다.
입력의 세 번째 줄에는 찾고자 하는 정수 m이 주어진다.

출력의 첫째 줄에는 m이 배열에 있다면 m의 인덱스를(0부터 시작) 없다면 -1을 출력하라.
출력의 둘째 줄에는 위의 답을 찾기 위해 mid=(first + last) / 2 코드가 몇 번 실행됐는지를 출력하라. m을 발견하지 못하더라도 코드의 실행 횟수를 출력하여야 한다. 특히, 이분탐색의 실패조건을 확인할 때에는 mid를 다시 계산할 필요가 없는데, 불필요하게 mid를 계산하지 않도록 주의하라.

For example:

Input	Result
12 1 3 5 7 8 9 10 11 12 14 16 17 5	2 2

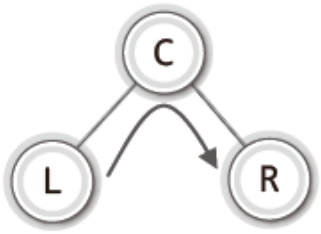
재귀 호출 및 이분탐색 연습 - 3

```c
#include <stdio.h>
#include <stdlib.h>

int arr[100001];
int count = 0;

int BSearchRecur(int ar[], int first, int last, int m) {
    if (first > last) return -1;

    int mid = (first + last) / 2;
    count++;
    if (m == ar[mid])
        return mid;

    if (m < ar[mid])
        return BSearchRecur(ar, first, mid - 1, m);

    return BSearchRecur(ar, mid + 1, last, m);
}

int main() {
    int n, m, i;
    scanf("%d", &n);
    for (i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    scanf("%d", &m);

    printf("%d\n", BSearchRecur(arr, 0, n-1, m));
    printf("%d\n", count);
    return 0;
}
```

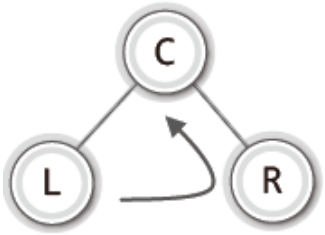
순회의 세 가지 방법

기준은 루트 노드를 언제 방문하느냐에 있다!

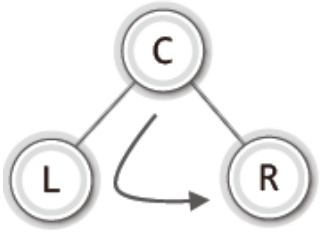
즉 루트 노드를 방문하는 시점에 따라서 중위, 후위, 전위 순회로 나뉘어 진다.



▶ [그림 08-21: 중위 순회]



▶ [그림 08-22: 후위 순회]

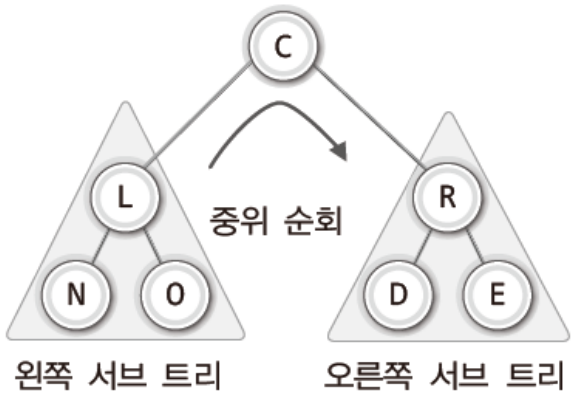


높이가 2 이상인 트리의 순회도 이와 다르지 않다.

재귀적인 형태로 순회의 과정을 구성하면 높이에 상관없이 순회가 가능하다!

▶ [그림 08-23: 전위 순회]

순회의 재귀적 표현



▶ [그림 08-24: 이진 트리의 중위 순회]

- 1단계 왼쪽 서브 트리의 순회
- 2단계 루트 노드의 방문
- 3단계 오른쪽 서브 트리의 순회

재귀적 표현

```
void InorderTraverse(BTreeNode * bt)
{
    InorderTraverse(bt->left);
    printf("%d \n", bt->data);
    InorderTraverse(bt->right);
}
```

탈출 조건이 명시되지 않은 불완전한 구현

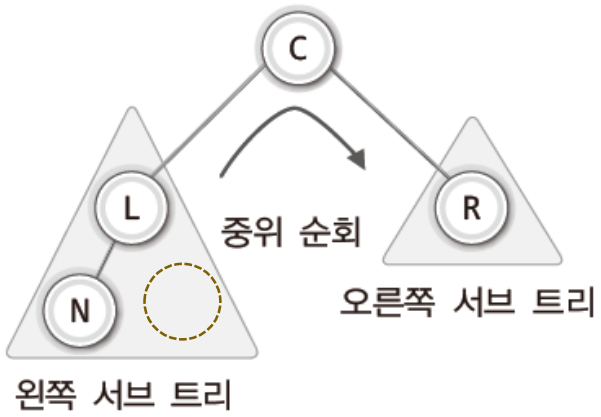
순회의 재귀적 표현 완성!

```
void InorderTraverse(BTreeNode * bt)
{
    if(bt == NULL)     // bt가 NULL이면 재귀 탈출!
        return;

    InorderTraverse(bt->left);
    printf("%d \n", bt->data);
    InorderTraverse(bt->right);
}
```

적용 가능

노드가 단말 노드인 경우,

단말 노드의 자식 노드는 NULL이다!



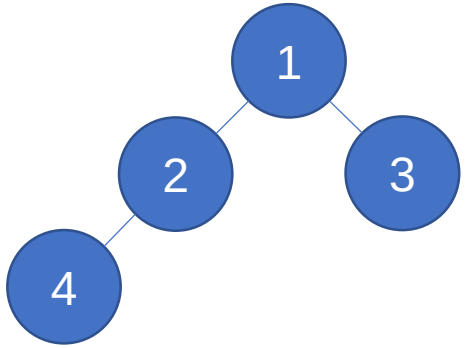
지금까지의 결과물 실행

```
int main(void)
{
    BTreeNode * bt1 = MakeBTreeNode();
    BTreeNode * bt2 = MakeBTreeNode();
    BTreeNode * bt3 = MakeBTreeNode();
    BTreeNode * bt4 = MakeBTreeNode();

    SetData(bt1, 1);
    SetData(bt2, 2);
    SetData(bt3, 3);
    SetData(bt4, 4);

    MakeLeftSubTree(bt1, bt2);
    MakeRightSubTree(bt1, bt3);
    MakeLeftSubTree(bt2, bt4);

    InorderTraverse(bt1);
    return 0;
}
```

BinaryTree.h
BinaryTree.c
BinaryTreeTraverseMain.c

실행결과

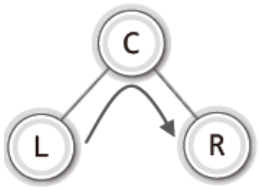
```
4
2
1
3
```



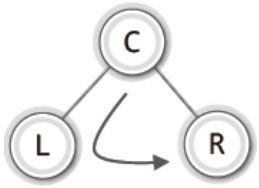
전위 순회와 후위 순회

```
void InorderTraverse(BTreeNode * bt)
{
     if(bt == NULL)     // bt가 NULL이면 재귀 탈출!
         return;

     InorderTraverse(bt->left);
     printf("%d \n", bt->data);
     InorderTraverse(bt->right);
}
```

중위 순회

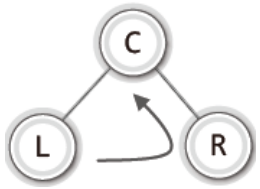



```
void PreorderTraverse(BTreeNode * bt)
{
     if(bt == NULL)
         return;

     printf("%d \n", bt->data);
     PreorderTraverse(bt->left);
     PreorderTraverse (bt->right);
}
```



전위 순회

```
void PostorderTraverse(BTreeNode * bt)
{
     if(bt == NULL)
         return;

     PostorderTraverse(bt->left);
     PostorderTraverse(bt->right);
     printf("%d \n", bt->data);
}
```



후위 순회

노드의 방문 이유! 자유롭게 구성하기!

(*VisitFuncPtr) 로 대신해도 됩니다.

```
typedef void VisitFuncPtr(BTData data);
```

함수 포인터 형 VisitFuncPtr의 정의

```
void InorderTraverse(BTreeNode * bt, VisitFuncPtr action)
{
    if(bt == NULL)
        return;

    InorderTraverse(bt->left, action);
    action(bt->data);       // 노드의 방문
    InorderTraverse(bt->right, action);
}
```

action이 가리키는 함수를 통해서 방문을 진행~

```
void ShowIntData(int data)
{
    printf("%d ", data);
}
```

VisitFuncPtr형을 기준으로 정의된 함수

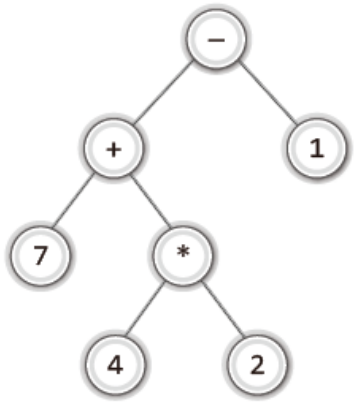
수식 트리(Expression Tree)의 구현

수식 트리의 이해

중위 표기법의 수식을 수식 트리로 변환하는 프로그램의 작성이 목적!

```
int main(void)
{
    int result = 0;
    result = 7 + 4 * 2 – 1;
    . . . .
}
```

수식 트리

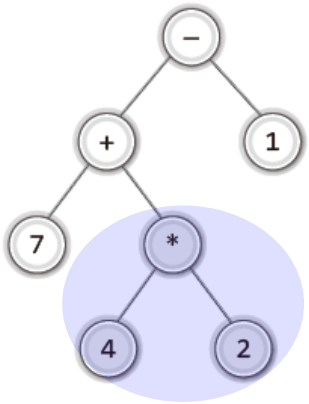


두 개의 자식 노드에는 피연산자를!

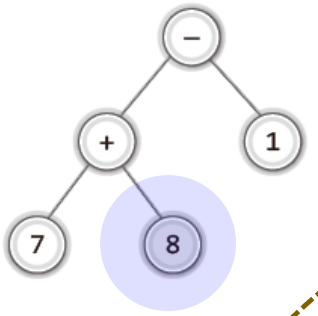
- 중위 표기법의 수식은 사람이 인식하기 좋은 수식이다. 컴퓨터의 인식에는 어려움이 있다.
- 그래서 컴파일러는 중위 표기법의 수식을 '수식 트리'로 재구성한다.
- 수식 트리는 해석이 쉽다. 연산의 과정에서 우선순위를 고려하지 않아도 된다!

수식 트리의 계산과정

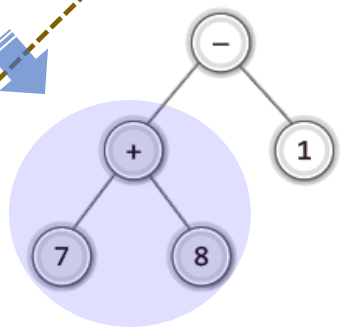
두 개의 자식 노드기 피연산자라는
단순하지만 전부인 하나의 특성을 근거로
연산이 매우 쉽게 진행된다!



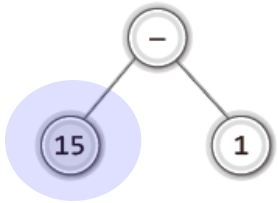
곱셈 연산 후



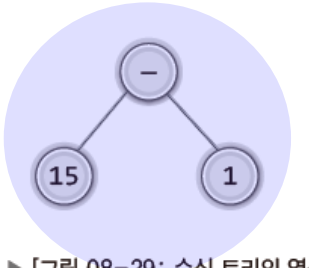
▶ [그림 08-27: 수식 트리의 연산과정 1/3]



덧셈 연산 후



▶ [그림 08-28: 수식 트리의 연산과정 2/3]



뺄셈 연산 후

▶ [그림 08-29: 수식 트리의 연산과정 3/3]

수식 트리를 만드는 절차!

Ch06의 ConvToRPNExp 함수에서 구현

중위 표기법의 수식 → 후위 표기법의 수식 → 수식 트리

그래서 후위 표기법의 수식을 수식 트리로 구성하는 방법만 고민!

중위 표기법의 수식을 바로 수식 트리로 표현하는 것은 쉽지 않다.
하지만 일단 후위 표기법의 수식으로 변경한 다음에 수식 트리로 표현하는 것은 어렵지 않다!

앞서 구현한 필요한 도구들!

- 수식 트리 구현에 필요한 이진 트리 BinaryTree2.h, BinaryTree2.c
- 수식 트리 구현에 필요한 스택 ListBaseStack.h, ListBaseStack.c

수식 트리의 구현과 관련된 헤더파일

```
#include "BinaryTree2.h"
```

트리 만드는 도구를 기반으로 함수를 정의한다!

```
BTreeNode * MakeExpTree(char exp[]);          // 수식 트리 구성
```

후위 표기법의 수식을 인자로 받아서 수식 트리를 구성하고
루트 노드의 주소 값을 반환한다!

```
int EvaluateExpTree(BTreeNode * bt);      // 수식 트리 계산
```

MakeExpTree가 구성한 수식 트리의 수식을 계산하여 그 결과를 반환한다!

```
void ShowPrefixTypeExp(BTreeNode * bt);       // 전위 표기법 기반 출력
void ShowInfixTypeExp(BTreeNode * bt);        // 중위 표기법 기반 출력
void ShowPostfixTypeExp(BTreeNode * bt);      // 후위 표기법 기반 출력
```

전위, 중위, 후위 순회하여 출력 시
각각 전위, 중위, 후위 표기법의 수식이 출력된다.

수식 트리의 구성 방법: 그림상 이해하기

후위 표기법의 수식에서 먼저 등장하는 피연산자와 연산자를 이용해서 트리의 하단부터 구성해 나가고 이어서 점진적으로 윗부분을 구성해 나간다.

수식 트리의 구성 방법: 코드로 옮기기1

▶ [그림 08-31 : 수식 트리의 구성 1/7]

피연산자는 무조건 스택으로!

▶ [그림 08-32 : 수식 트리의 구성 2/7]

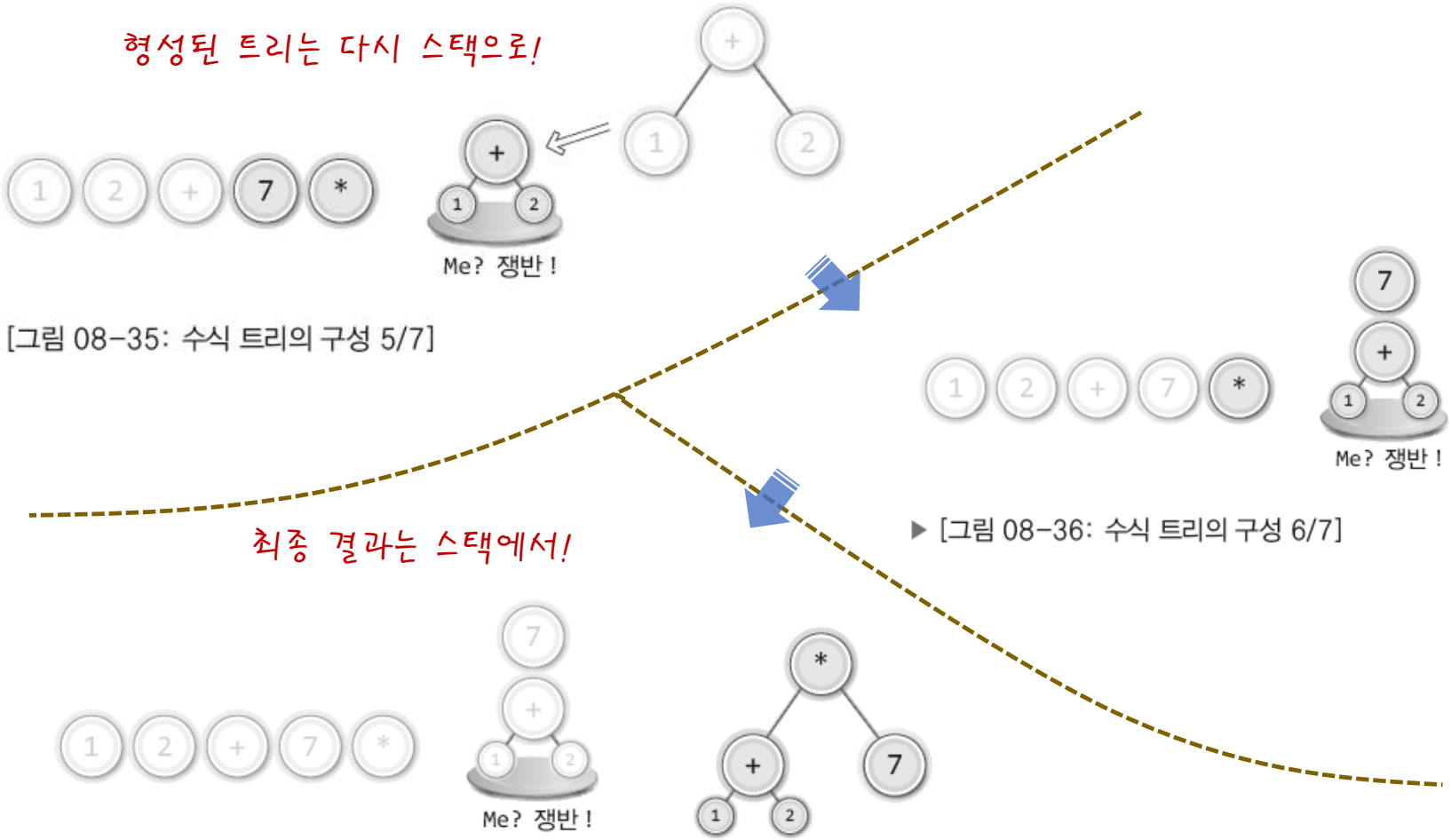
▶ [그림 08-33 : 수식 트리의 구성 3/7]

연산자 만나면 스택에서

피연산자 두 개 꺼내어 트리 구성!

▶ [그림 08-34 : 수식 트리의 구성 4/7]

수식 트리의 구성 방법: 코드로 옮기기2



▶ [그림 08-35: 수식 트리의 구성 5/7]

▶ [그림 08-36: 수식 트리의 구성 6/7]

▶ [그림 08-37: 수식 트리의 구성 7/7]

수식 트리의 구성 방법: 코드로 옮기기3

```
BTreeNode * MakeExpTree(char exp[])
{
    Stack stack;
    BTreeNode * pnode;
    int expLen = strlen(exp);
    int i;
    for(i=0; i<expLen; i++)
    {
        pnode = MakeBTreeNode();
        if(isdigit(exp[i])) {                    // 피연산자라면...
            SetData(pnode, exp[i]-'0');          // 문자를 정수로 바꿔서 저장
        }
        else {                                   // 연산자라면...
            MakeRightSubTree(pnode, SPop(&stack));
            MakeLeftSubTree(pnode, SPop(&stack));
            SetData(pnode, exp[i]);
        }
        SPush(&stack, pnode);
    }
    return SPop(&stack);
}
```

- 피연산자는 스택으로 옮긴다.
- 연산자를 만나면 스택에서 두 개의 피연산자 꺼내어 자식 노드로 연결!
- 자식 노드를 연결해서 만들어진 트리는 다시 스택으로 옮긴다.

수식 트리의 순회: 그 결과

- 전위 순회하여 데이터를 출력한 결과 → 전위 표기법의 수식
- 중위 순회하여 데이터를 출력한 결과 → 중위 표기법의 수식
- 후위 순회하여 데이터를 출력한 결과 → 후위 표기법의 수식

수식 트리를 구성하면, 전위, 중위, 후위 표기법으로의 수식 표현이 쉬워진다.

그리고 전회, 중위, 후위 순회하면서 출력되는 결과물을 통해서 MakeExpTree 함수를 검증할 수 있다.

수식 트리의 순회: 방법

```
void ShowPrefixTypeExp(BTreeNode * bt)
{
    PreorderTraverse(bt, ShowNodeData);
}
```
전위 표기법 수식 출력

```
void ShowInfixTypeExp(BTreeNode * bt)
{
    InorderTraverse(bt, ShowNodeData);
}
```
중위 표기법 수식 출력

```
void ShowPostfixTypeExp(BTreeNode * bt)
{
    PostorderTraverse(bt, ShowNodeData);
}
```
후위 표기법 수식 출력

VisitFuncPtr형 함수

```
void ShowNodeData(int data)
{
    if(0<=data && data<=9)
        printf("%d ", data);      // 피연산자 출력
    else
        printf("%c ", data);      // 연산자 출력
}
```

수식 트리 관련 예제의 실행

ListBaseStack.h의 type 선언 변경 필요하다!

typedef BTreeNode * BTData;

```
int main(void)
{
    char exp[] = "12+7*";
    BTreeNode * eTree = MakeExpTree(exp);

    printf("전위 표기법의 수식: ");
    ShowPrefixTypeExp(eTree); printf("\n");

    printf("중위 표기법의 수식: ");
    ShowInfixTypeExp(eTree); printf("\n");

    printf("후위 표기법의 수식: ");
    ShowPostfixTypeExp(eTree); printf("\n");

    printf("연산의 결과: %d \n", EvaluateExpTree(eTree));

    return 0;
}
```

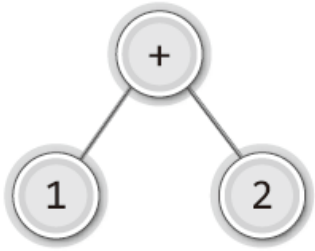
- 이진 트리 관련
  BinaryTree2.h, BinaryTree2.c
- 스택 관련
  ListBaseStack.h, ListBaseStack.c
- 수식 트리 관련
  ExpressionTree.h, ExpressionTree.c
- main 함수 관련
  ExpressionMain.c

실행결과

```
전위 표기법의 수식: * + 1 2 7
중위 표기법의 수식: 1 + 2 * 7
후위 표기법의 수식: 1 2 + 7 *
연산의 결과: 21
```

수식 트리의 계산: 기본 구성

```
int EvaluateExpTree(BTreeNode * bt)
{
    int op1, op2;
    op1 = GetData(GetLeftSubTree(bt));      // 첫 번째 피연산자
    op2 = GetData(GetRightSubTree(bt));     // 두 번째 피연산자
    switch(GetData(bt))             // 연산자를 확인하여 연산을 진행
    {
    case '+':
        return op1+op2;
    case '-':
        return op1-op2;
    case '*':
        return op1*op2;
    case '/':
        return op1/op2;
    }
    return 0;
}
```



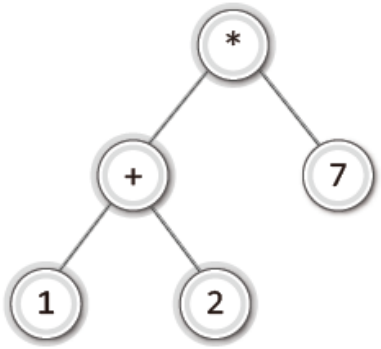
이 모델을 대상으로 한 구현

수식 트리의 계산: 재귀적 구성

```
int EvaluateExpTree(BTreeNode * bt)
{
    int op1, op2;
    op1 = GetData(GetLeftSubTree(bt));      // 첫 번째 피연산자
    op2 = GetData(GetRightSubTree(bt));     // 두 번째 피연산자
    switch(GetData(bt))                     // 연산자를 확인하여 연산을 진행
    {
    case '+':
        return op1+op2;
    case '-':
        return op1-op2;
    case '*':
        return op1*op2;
    case '/':
        return op1/op2;
    }
    return 0;
}
```

재귀적 구성

```
op1 = EvaluateExpTree(GetLeftSubTree(bt));
op2 = EvaluateExpTree(GetRightSubTree(bt));
```



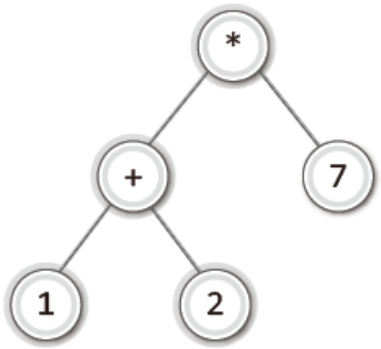
이 모델을 대상으로 한 구현

단! 단말노드에 대해서는 고려되지 않았다!

수식 트리의 계산

```
int EvaluateExpTree(BTreeNode * bt)
{
    int op1, op2;

    if(GetLeftSubTree(bt)==NULL && GetRightSubTree(bt)==NULL) // 단말 노드라면
        return GetData(bt);

    op1 = EvaluateExpTree(GetLeftSubTree(bt));
    op2 = EvaluateExpTree(GetRightSubTree(bt));
    switch(GetData(bt))
    {
        . . . . 이전과 동일 . . .
    }
    return 0;
}
```

탈출조건!



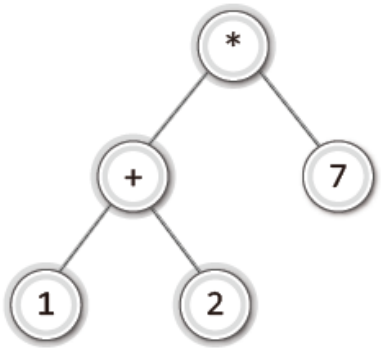
이 모델을 대상으로 한 구현

단말노드는 탈출의 조건이다!

수식 트리의 계산

```
int EvaluateExpTree(BTreeNode * bt)
{
    int op1, op2;

    if(GetLeftSubTree(bt)==NULL && GetRightSubTree(bt)==NULL) // 단말 노드라면
        return GetData(bt);

    op1 = EvaluateExpTree(GetLeftSubTree(bt));
    op2 = EvaluateExpTree(GetRightSubTree(bt));
    switch(GetData(bt))
    {
        . . . . 이전과 동일 . . .
    }
    return 0;
}
```

탈출조건!



이 모델을 대상으로 한 구현

단말노드는 탈출의 조건이다!

요약

- 트리의 세 가지 순회 방법
  - 전위 순회(preorder traversal): **나**, 왼쪽, 오른쪽
    - = 깊이 우선 순회
  - 중위 순회(inorder traversal): 왼쪽, **나**, 오른쪽
  - 후위 순회(postorder traversal): 왼쪽, 오른쪽, **나**

- 수식 트리는 수식을 트리 형태로 표현한 것
  - 연산자를 내부 노드에, 피연산자를 단말 노드에
  - 높은 레벨 노드부터(아래부터) 계산

출석 인정을 위한 보고서 작성

1) 어떤 트리의 각 노드를 색으로 칠하고 싶다. 단, 간선으로 연결된 두 노드는 다른 색으로 칠해야 한다. 몇 가지 색이 필요할까?

2) 어떤 트리의 전위 순회 결과가 “1 2 3 4 5 6”이고 중위 순회 결과가 “2 1 4 3 6 5“라고 한다. 이 트리는 어떻게 생겼을까?

3) 어떤 트리에 대해 함수 *f*는 노드의 번호 *i*를 인자로 받아 여기서 가장 멀리 떨어진 노드 번호 *j*를 돌려준다고 하자. *f*를 최소 몇 번 호출해야 트리에서 가장 멀리 떨어진 두 노드를 찾을 수 있을까?