Tree

Data Structures and Algorithms

목차

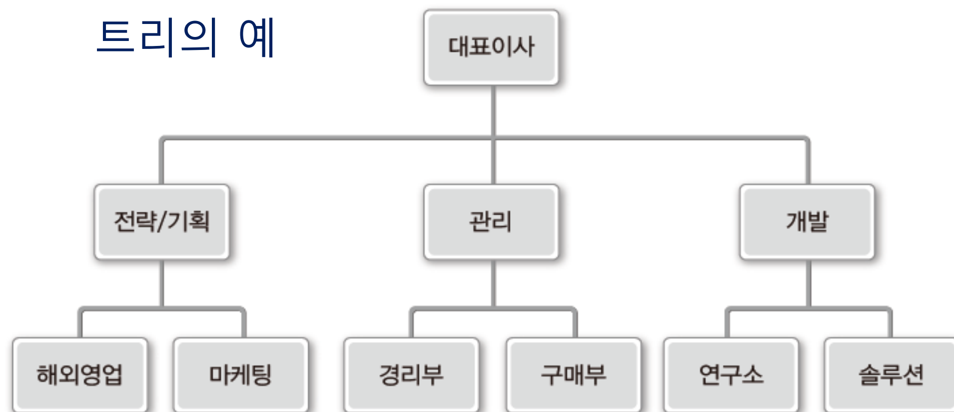
- 트리의 개요
- 이진 트리의 구현
- 이진 트리의 순회(Traversal)
- 수식 트리(Expression Tree)의 구현

트리의 개요

트리의 접근과 이해

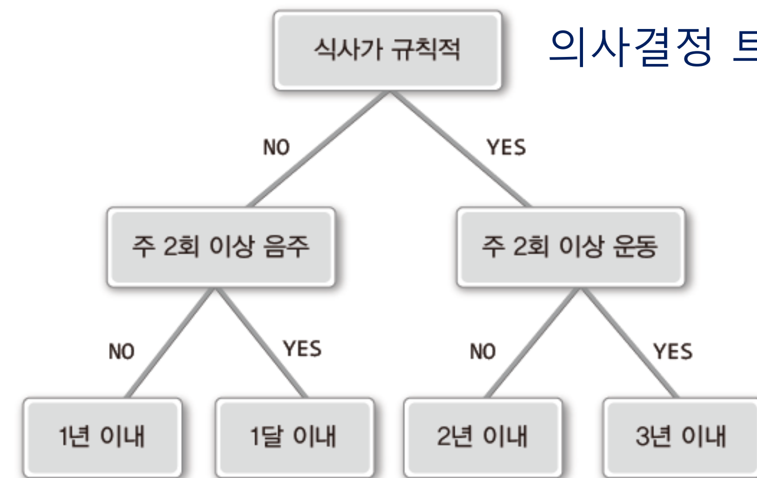
- 트리는 계층적 관계(Hierarchical Relationship)를 표현하는 자료구조
    - 데이터 저장과 데이터의 표현

트리의 예



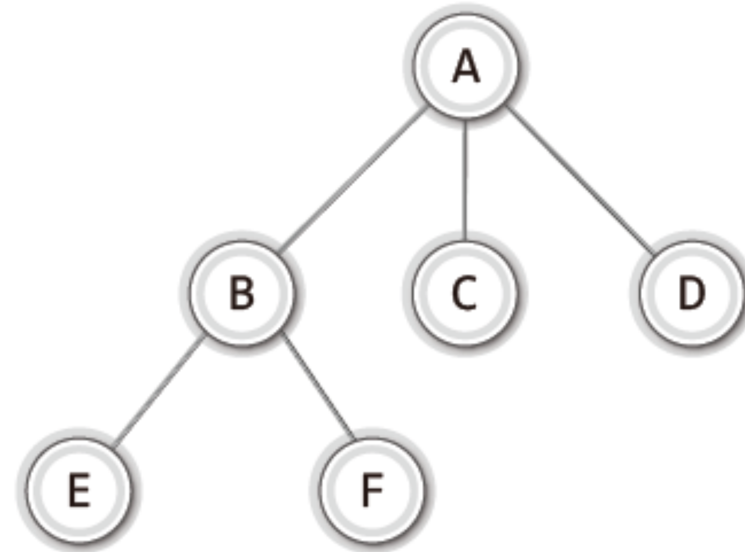
트리의 예:

의사결정 트리



트리 관련 용어의 소개

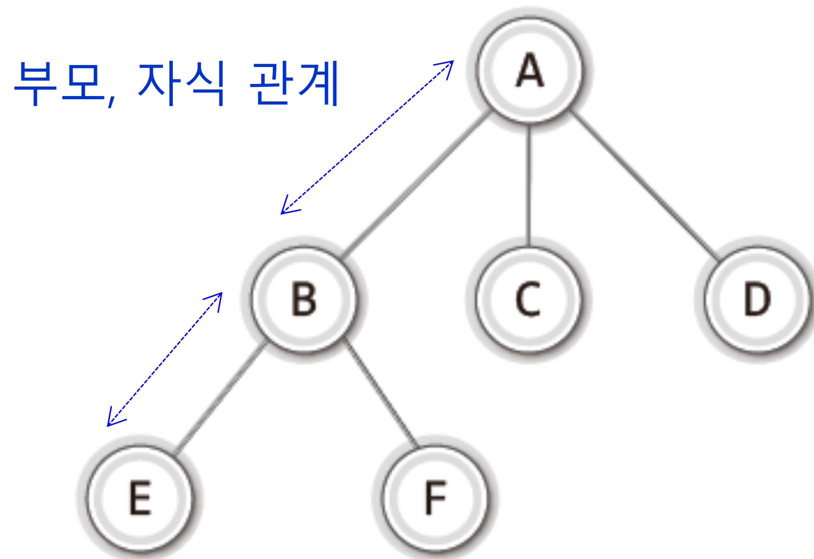
- 노드: node
  - 트리의 구성요소 (A, B, C, D, E, F)

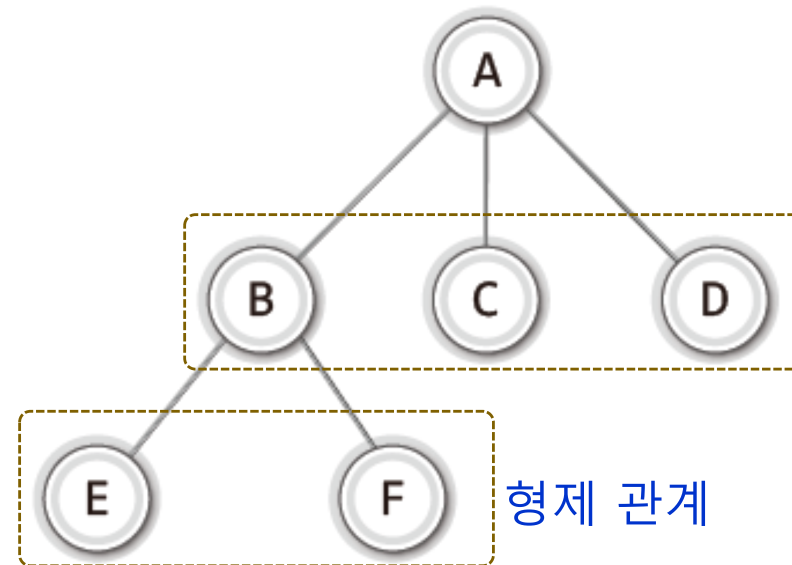


- 간선: edge
  - 노드와 노드를 연결하는 선
- 루트 노드: root node
  - 트리 구조에서 최상위의 노드 (A)
- 단말 노드: terminal node
  - 아래로 또 다른 노드가 연결되어 있지 않은 노드 (E, F, C, D)
- 내부 노드: internal node
  - 단말 노드를 제외한 모든 노드 (A, B)

트리의 노드간 관계

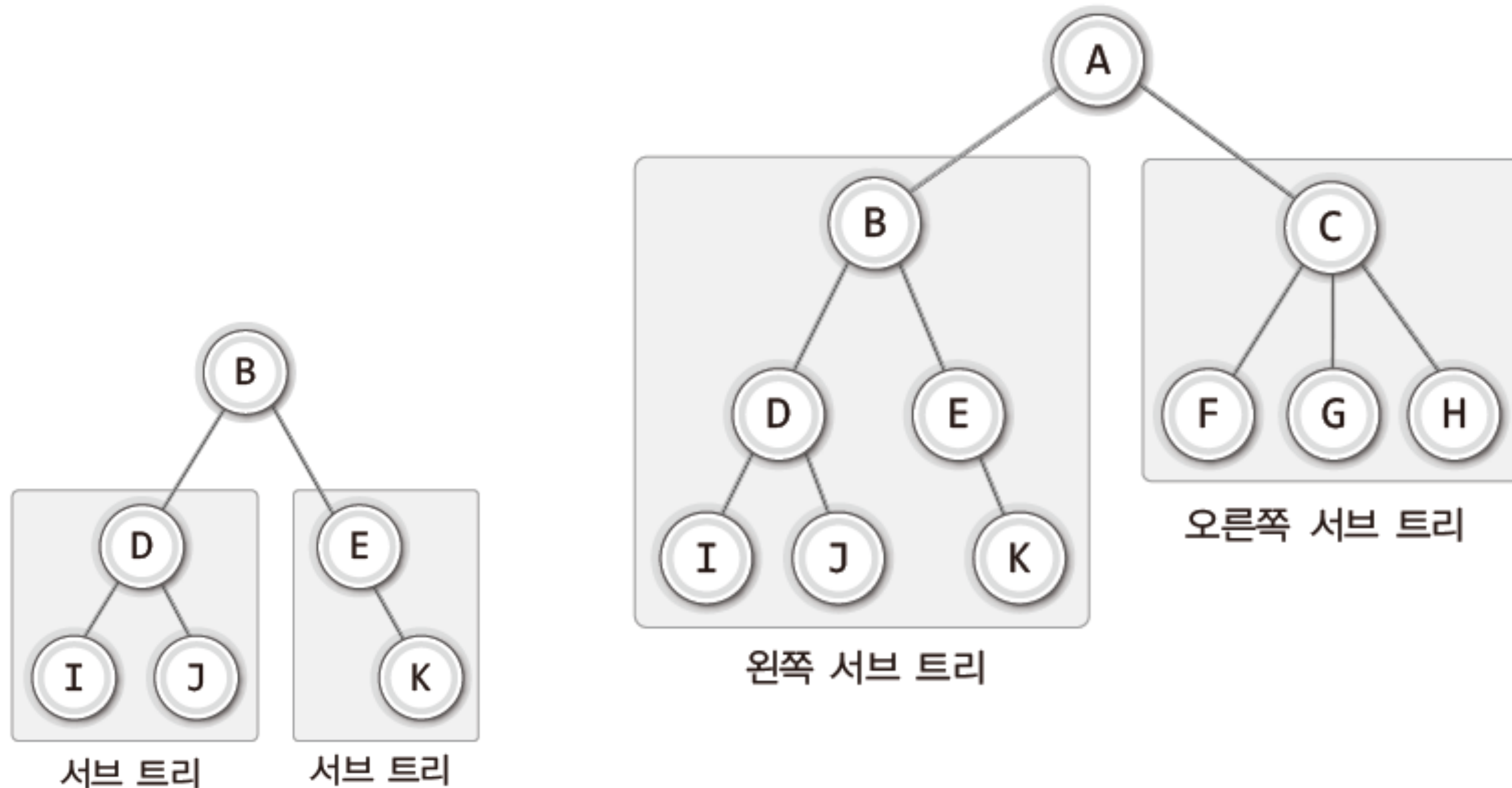
- 노드 A는 노드 B, C, D의 부모 노드(parent node)
- 노드 B, C, D는 노드 A의 자식 노드(child node)
- 부모 노드가 같은 노드 B, C, D는 형제 노드(sibling node)





서브 트리

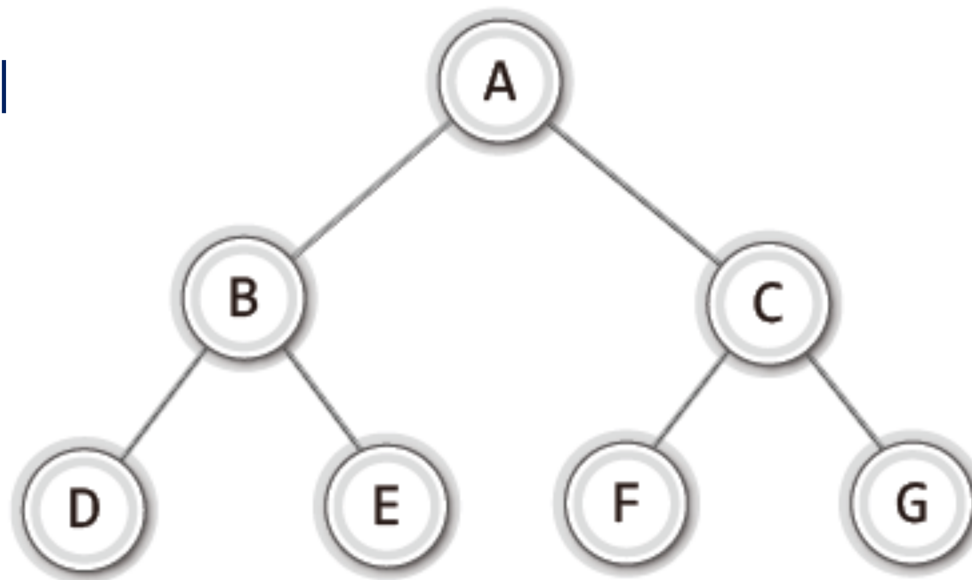
- 하나의 트리를 구성하는 왼쪽과 오른쪽의 작은 트리
  - 서브 트리 역시 서브 트리로 이뤄져 있음
  - 서브 트리는 재귀적



이진 트리

- 조건
  - 루트 노드를 중심으로 두 개의 서브 트리로 나뉨
  - 나뉜 두 서브 트리도 모두 이진 트리이어야 함

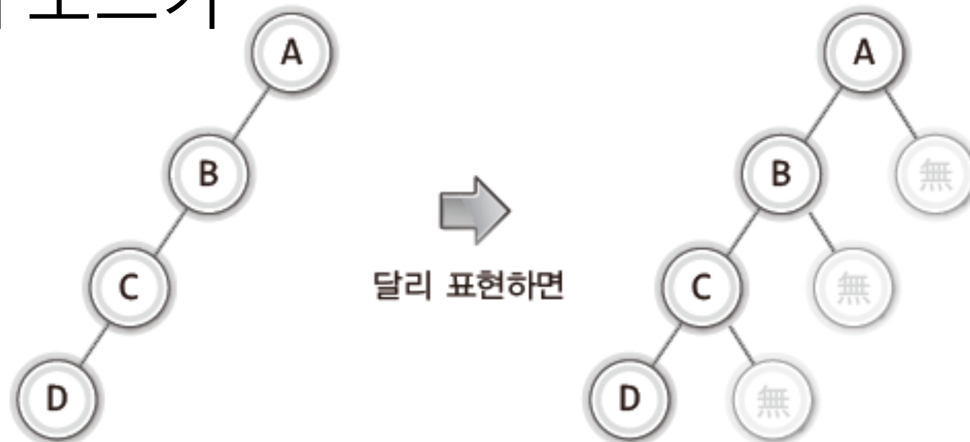
이진 트리의 예



이진 트리와 공집합 노드

- 공집합(empty set)도 이진 트리에서는 노드

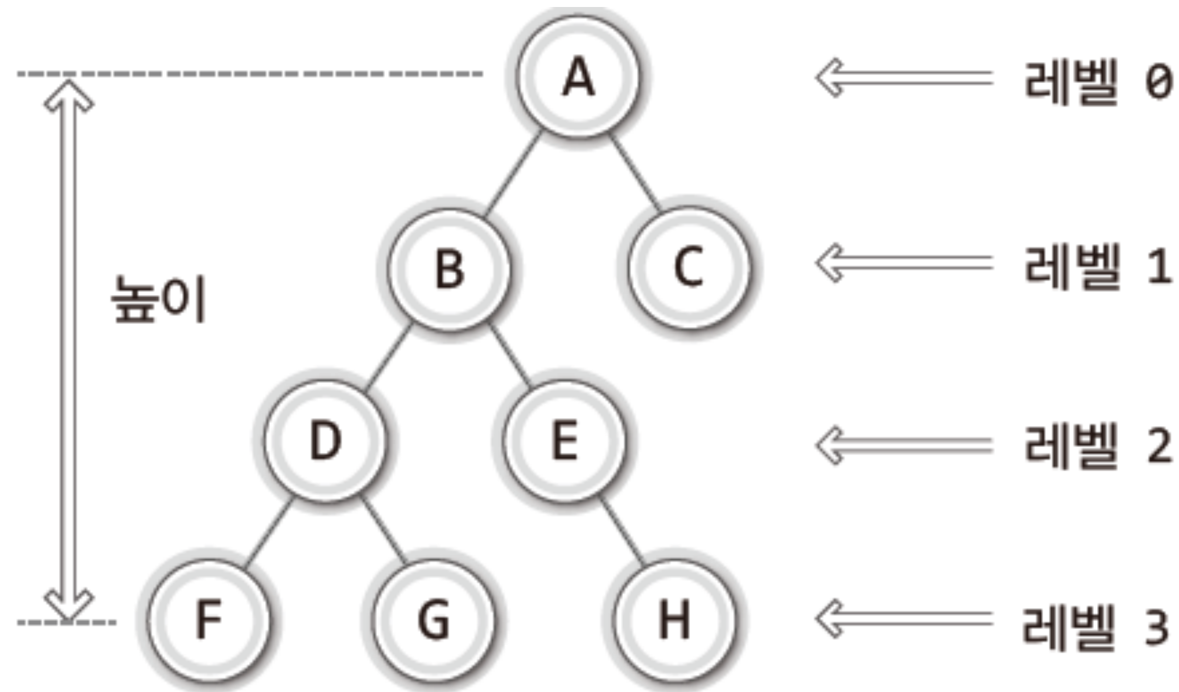
- 하나의 노드에 두 개의 노드가 달리는 형태의 트리는 모두 이진 트리



레벨과 높이

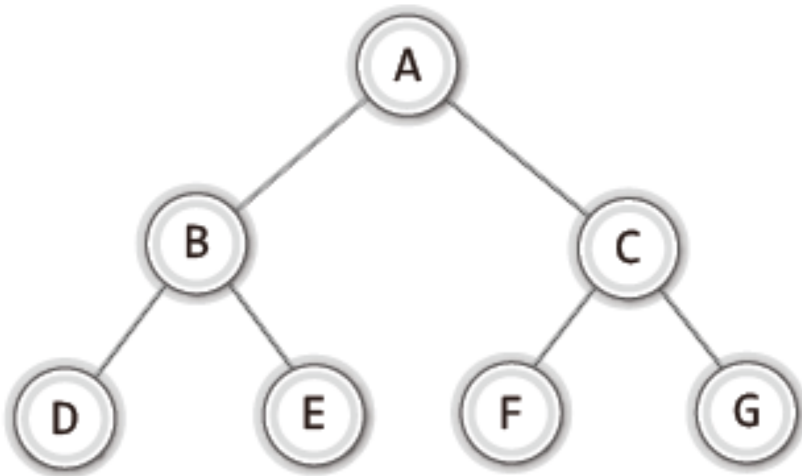
- 트리의 높이 = 레벨의 최대 값

높이 = 3



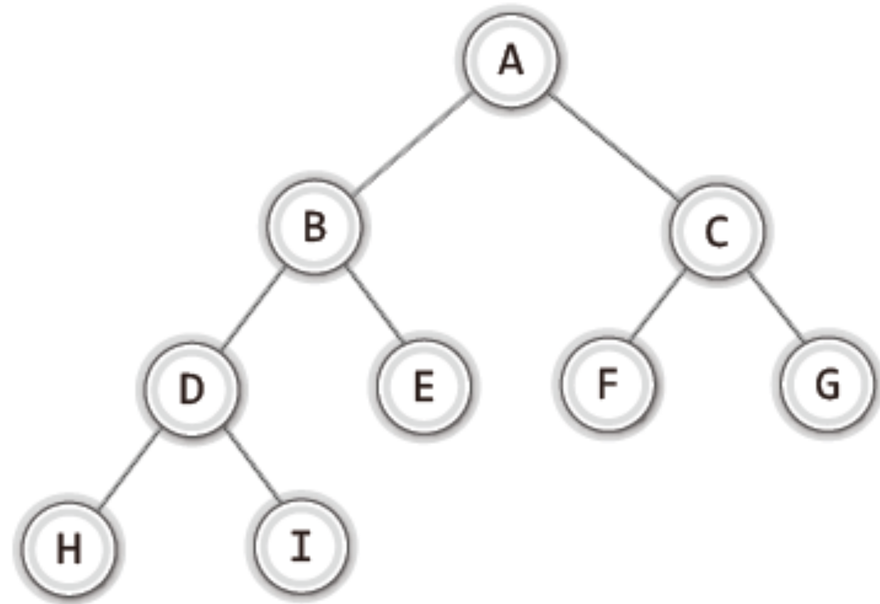
포화, 완전 이진 트리

- 완전 이진 트리는 위에서 아래로 왼쪽에서 오른쪽으로 채워진 트리
- 포화 이진 트리는 완전 이진 트리이지만 그 역은 성립하지 않음



모든 레벨에 노드가 꽉 찬

포화 이진 트리



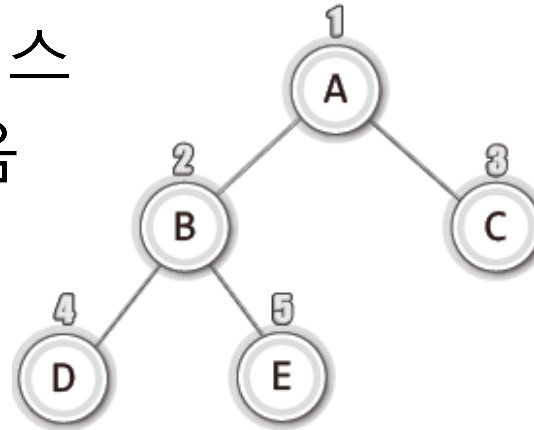
빈 틈 없이 차곡차곡 채워짐

완전 이진 트리

이진 트리의 구현

이진 트리 구현의 두 가지 방법

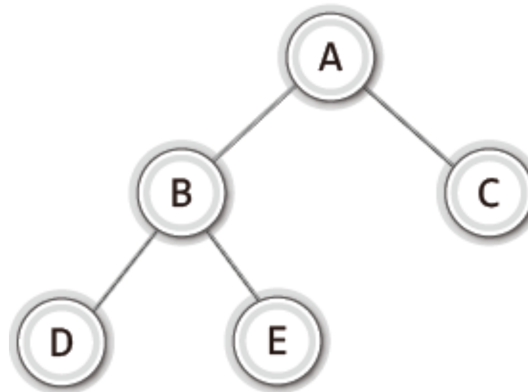
- 배열: 인덱스를 활용하는 쉬운 접근
  - 노드 번호가 배열의 인덱스
  - 편의상 첫 요소 사용 않음



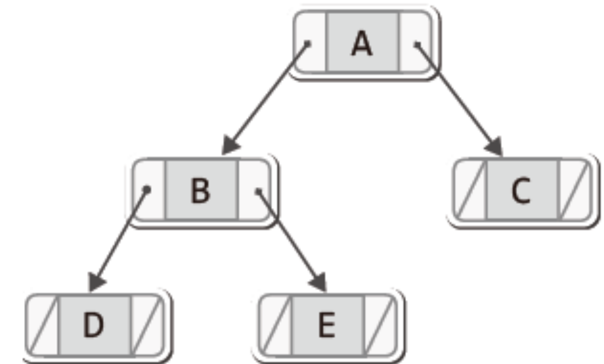
구현

[0]	
[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	·
[7]	·

- 연결리스트: 직관적 구조
  - 트리의 구조와 리스트의 연결 구조가 일치



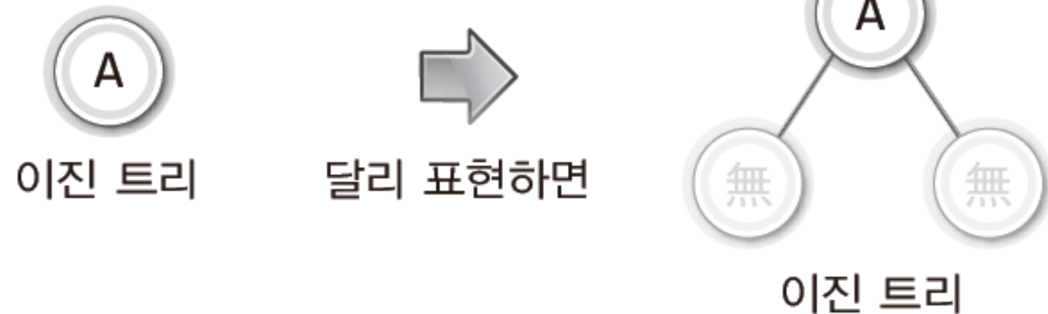
구현



헤더파일에 정의된 구조체

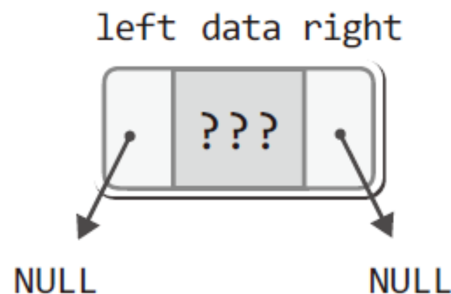
- 이진 트리의 모든 노드는 직/간접적으로 연결됨
- 루트 노드의 주소만 알면 이진 트리 전체 탐색 가능

```
typedef struct _bTreeNode
{
    BTData data;
    struct _bTreeNode * left;
    struct _bTreeNode * right;
} BTreeNode;
```



헤더파일에 선언된 함수들1

```
BTreeNode * MakeBTreeNode(void);  // 노드의 생성
BTData GetData(BTreeNode * bt);   // 노드에 저장된 데이터를 반환
void SetData(BTreeNode * bt, BTData data);
                                  // 노드에 데이터를 저장
```



동적으로 노드를 생성
포인터 변수 left와 right 는 NULL로 초기화

헤더파일에 선언된 함수들2

- 인자로 트리의 어떤 노드도 전달 가능

```
BTreeNode * GetLeftSubTree(BTreeNode * bt);
                            // 좌측 서브 트리의 주소 값 반환!
BTreeNode * GetRightSubTree(BTreeNode * bt);
                            // 우측 서브 트리의 주소 값 반환!

void MakeLeftSubTree(BTreeNode * main, BTreeNode * sub);
                            // main의 서브 왼쪽 서브 트리로 sub를 연결!
void MakeRightSubTree(BTreeNode * main, BTreeNode * sub);
                            // main의 오른쪽 서브 트리로 sub를 연결!
```

main 함수

```
int main(void)
{
    BTreeNode * ndA = MakeBTreeNode();        // 노드 A 생성

    BTreeNode * ndB = MakeBTreeNode();        // 노드 B 생성

    BTreeNode * ndC = MakeBTreeNode();        // 노드 C 생성

    ndB, ndB, ndC를 이용한 SetData 함수 호출을 통해 유용한 데이터 채운 후…

    MakeLeftSubTree(ndA, ndB); // 노드 A의 왼쪽 자식 노드로 노드 B 연결

    MakeRightSubTree(ndA, ndC); // 노드 A의 오른쪽 자식 노드로 노드 C 연결

    . . . .
}
```

이진 트리의 구현

```
BTreeNode * MakeBTreeNode(void)
{
    BTreeNode * nd = (BTreeNode*)malloc(sizeof(BTreeNode));

    nd->left = NULL;
    nd->right = NULL;
    return nd;
}

BTData GetData(BTreeNode * bt)
{
    return bt->data;
}

void SetData(BTreeNode * bt, BTData data)
{
    bt->data = data;
}

BTreeNode * GetLeftSubTree(BTreeNode * bt)
{
    return bt->left;
}
```

이진 트리의 구현

```
BTreeNode * GetRightSubTree(BTreeNode * bt)
{
    return bt->right;
}

void MakeLeftSubTree(BTreeNode * main, BTreeNode * sub)
{
    if(main->left != NULL)
        free(main->left);

    main->left = sub;
}

void MakeRightSubTree(BTreeNode * main, BTreeNode * sub)
{
    if(main->right != NULL)
        free(main->right);

    main->right = sub;
}
```

이진 트리 활용 main 함수

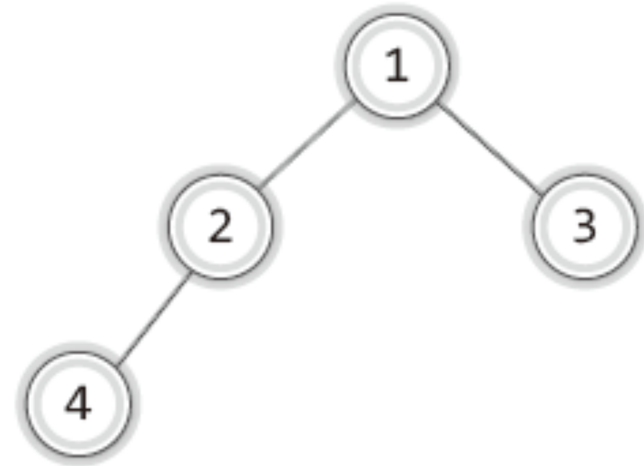
```
int main(void)
{
    BTreeNode * bt1 = MakeBTreeNode();
    BTreeNode * bt2 = MakeBTreeNode();
    BTreeNode * bt3 = MakeBTreeNode();
    BTreeNode * bt4 = MakeBTreeNode();

    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;
}
```

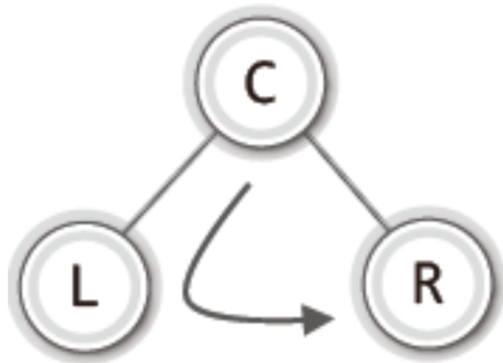
main 함수에서 생성하는 트리



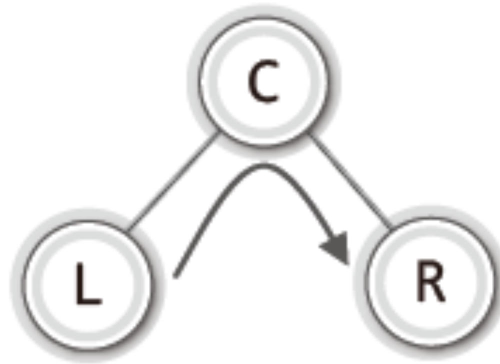
이진 트리의 순회(Traversal)

순회의 세 가지 방법

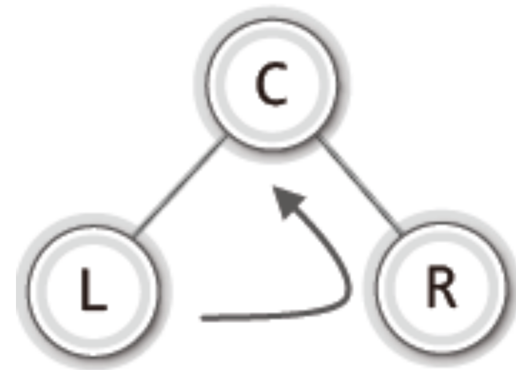
- 루트 방문 시점에 따라 전위, 중위 후위 순회
  - 높이 2 이상도 재귀적으로 순회 가능



전위



중위



후위

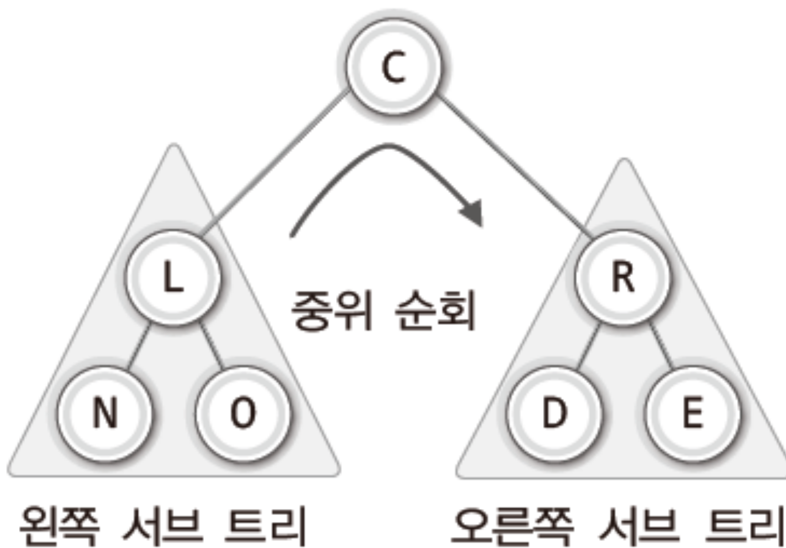
순회의 재귀적 표현

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

재귀적 표현

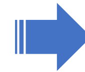
```
void InorderTraverse(BTreeNode * bt)
{
    InorderTraverse(bt->left);
    printf("%d \n", bt->data);
    InorderTraverse(bt->right);
}
```

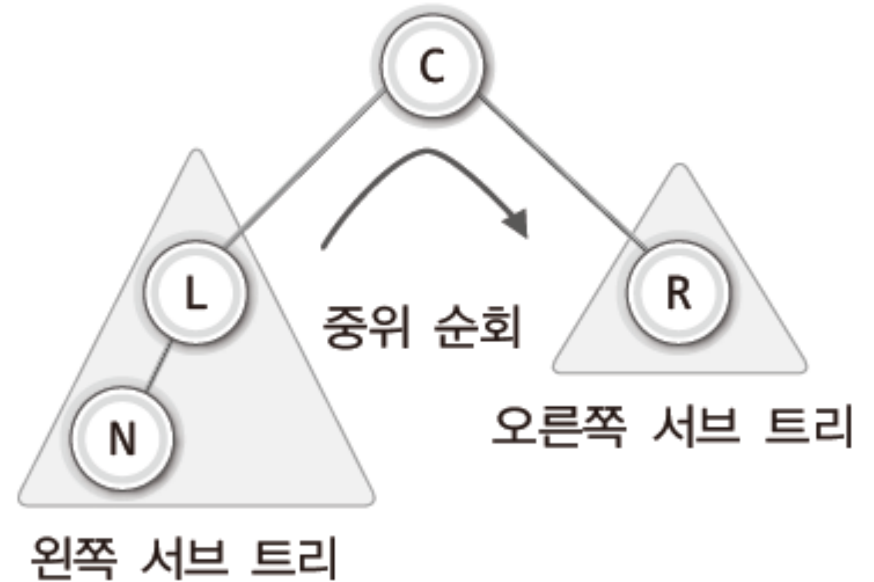
탈출 조건이 있어야 함



순회의 재귀적 표현 완성!

```
void InorderTraverse(BTreeNode * bt)
{
    if(!bt)    // bt가 NULL이면 재귀 탈출!
    {
        InorderTraverse(bt->left);
        printf("%d \n", bt->data);
        InorderTraverse(bt->right);
    }
}
```





단말 노드의 자식 노드는 NULL

전위 순회와 후위 순회

```
void PreorderTraverse(BTreeNode * bt)
{
    if(!bt)
    {

            ?

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

```
void PostorderTraverse(BTreeNode * bt)
{
    if(!bt)
    {

            ?

    }
}
```

Level Order traversing?

Option: 노드의 방문

- 함수 포인터 형 VisitFuncPtr의 정의

```
typedef void VisitFuncPtr(BTData data);

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

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

```
int main()
{
    ...
    InorderTraverse(bt1, ShowIntData);
    ...
}
```

```
// VisitFuncPtr형을 기준으로 정의된 함수
void ShowIntData(int data)
{
    printf("%d ", data);
}
```

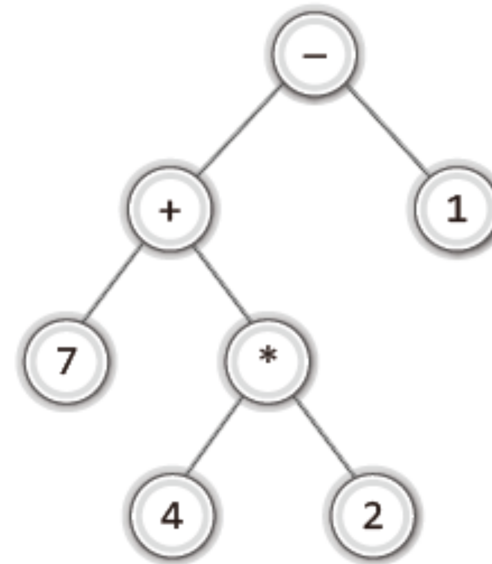
수식 트리
(Expression Tree)의 구현

수식 트리의 이해

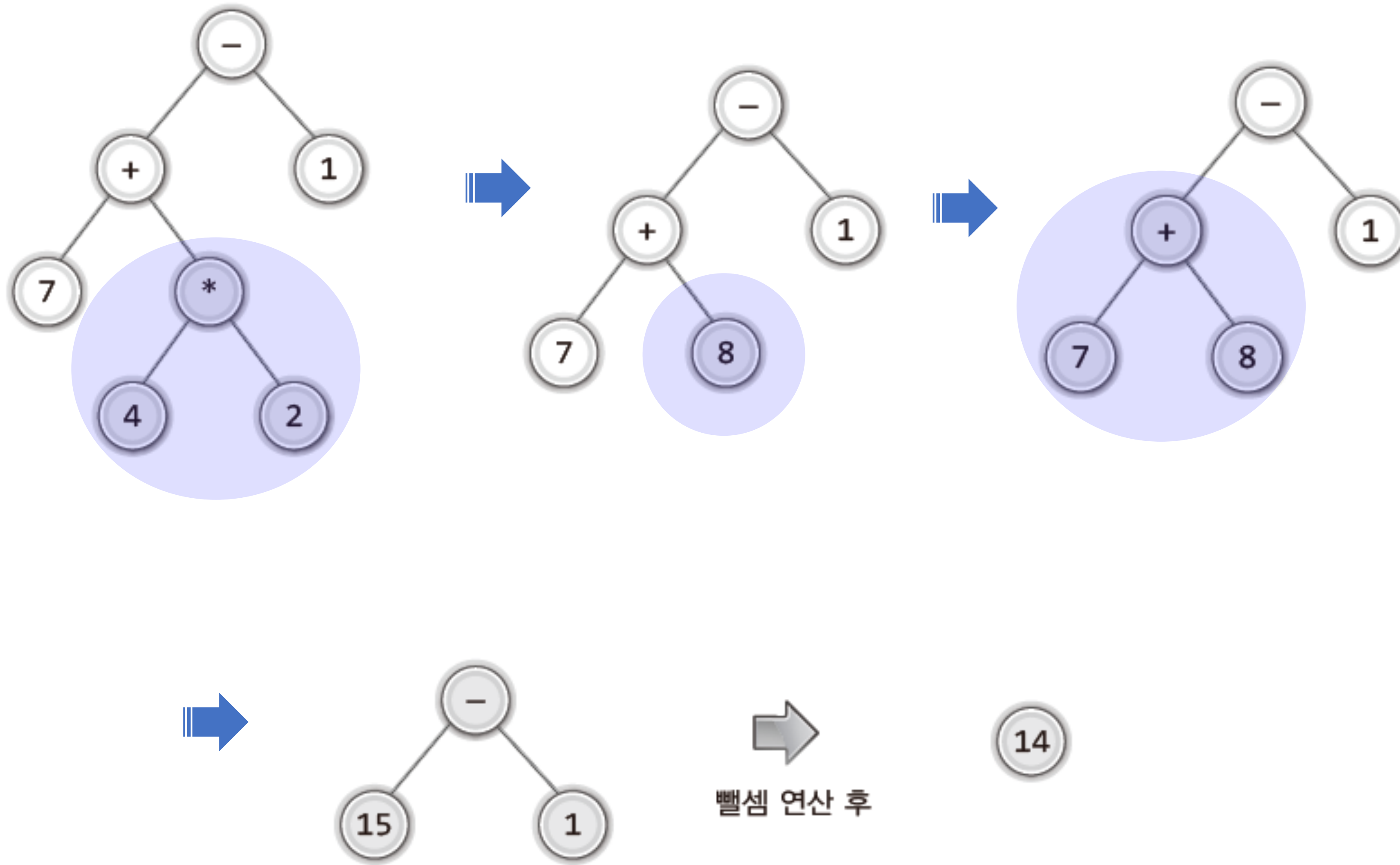
- 중위 표기법의 수식을 수식 트리로 변환하는 프로그램
  - 중위 표기법의 수식은 사람이 인식하기 좋은 수식
  - 컴퓨터의 인식에는 어려움이 있음
  - 컴파일러는 중위 표기법의 수식을 ‘수식 트리’로 재구성
- 수식 트리는 해석이 쉬움
  - 연산의 과정에서 우선순위를 고려하지 않아도 됨

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

수식 트리



수식 트리의 계산과정



▶ [그림 08-29: 수식 트리의 연산과정 3/3]

수식 트리를 만드는 절차!

중위 표기법의 수식 후위 표기법의 수식 수식 트리

- 중위 표기법에서 수식 트리로 변환은 어려움
  - 후위 표기법으로 변경 후 수식 트리로 변경

- 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

- 트리 만드는 도구를 기반으로 함수를 정의

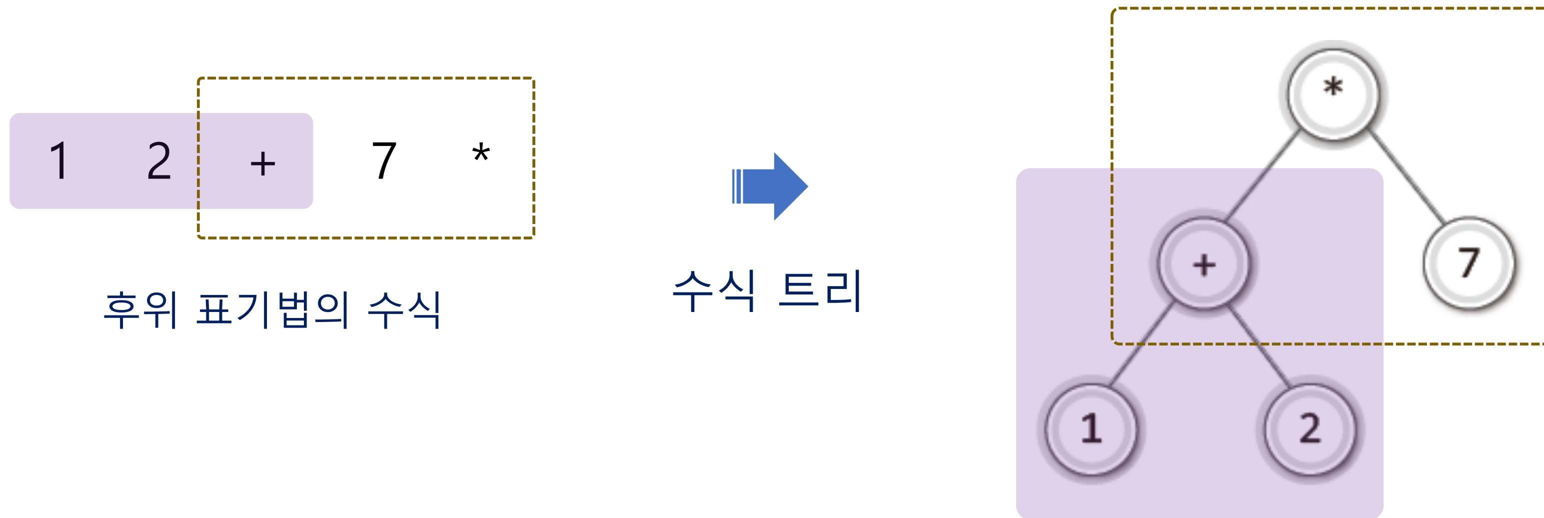
```
#include "BinaryTree2.h"

BTreeNode * MakeExpTree(char exp[]);              // 수식 트리 구성
int EvaluateExpTree(BTreeNode * bt);              // 수식 트리 계산

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

수식 트리의 구성 방법: 그림상 이해하기

- 먼저 등장하는 피연산자와 연산자부터 트리의 하단을 구성



수식 트리의 구성 방법: 코드로 옮기기1

피연산자는 무조건 스택으로!

연산자 만나면 스택에서

피연산자 두 개 꺼내어 트리 구성!

수식 트리의 구성 방법: 코드로 옮기기2

형성된 트리는 다시 스택으로!

수식 트리의 구성 방법: 코드로 옮기기3

최종 결과는 스택에서!

수식 트리의 구성 방법: 코드로 옮기기4

```
BTreeNode * MakeExpTree(char exp[])
{
    Stack stack;
    BTreeNode * pnode;

    int expLen = strlen(exp);
    int i;

    StackInit(&stack);
    for(i=0; i<expLen; i++)
    {
        pnode = MakeBTreeNode();

        if(isdigit(exp[i])) // 피연산자라면...
        {
            SetData(pnode, exp[i]-'0' );
        }
        else // 연산자라면...
        {
            MakeRightSubTree(pnode, SPop(&stack));
            MakeLeftSubTree(pnode, SPop(&stack));
            SetData(pnode, exp[i]);
        }
        SPush(&stack, pnode);
    }
    return SPop(&stack);
}
```

- 피연산자는 스택에 push
- 연산자를 만나면 스택에서 두 개의 피연산자 꺼내어 자식 노드로 연결!
- 자식 노드를 연결해서 만들어진 트리는 다시 스택에 push

수식 트리의 순회: 그 결과

- 수식 트리를 구성하면, 전위, 중위, 후위 표기법으로의 수식 표현이 쉬움
- 전회, 중위, 후위 순회하면서 출력되는 결과물을 통해서 MakeExpTree 함수를 검증 가능

- 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 트리의 순회: 방법

```c
void ShowPrefixTypeExp(BTreeNode * bt)
{
    PreorderTraverse(bt, ShowNodeData);
}

void ShowInfixTypeExp(BTreeNode * bt)
{
    InorderTraverse(bt, ShowNodeData);
}

void ShowPostfixTypeExp(BTreeNode * bt)
{
    PostorderTraverse(bt, ShowNodeData);
}
```

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

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

수식 트리 관련 예제의 실행

// ListBaseStack.h의 type 선언 변경 필요

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

수식 트리의 계산: 재귀적 구성

```
int EvaluateExpTree(BTreeNode * bt)
{
    int op1, op2;
    // 탈출 조건
    if(GetLeftSubTree(bt)==NULL &&  GetRightSubTree(bt)==NULL)
        return GetData(bt);
    // 재귀적 구성
    op1 = EvaluateExpTree(GetLeftSubTree(bt)); // 첫번째 피연자
    op2 = EvaluateExpTree(GetRightSubTree(bt)); // 두번째 피연자

    switch(GetData(bt))    // 연산자 확인
    {
        case '+':
            return op1+op2;
        case '-':
            return op1-op2;
        case '*':
            return op1*op2;
        case '/':
            return op1/op2;
    }

    return 0;
}
```