윤성우의 열혈 자료구조

: C언어를 이용한 자료구조 학습서

Chapter 08. 트리(Tree)

Introduction To Data Structures Using C

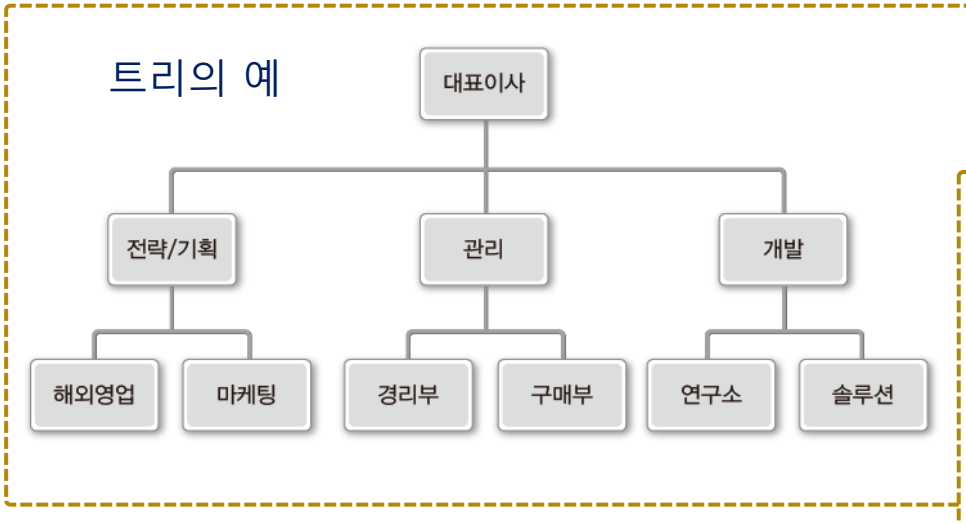
Chapter 08. 트리(Tree)

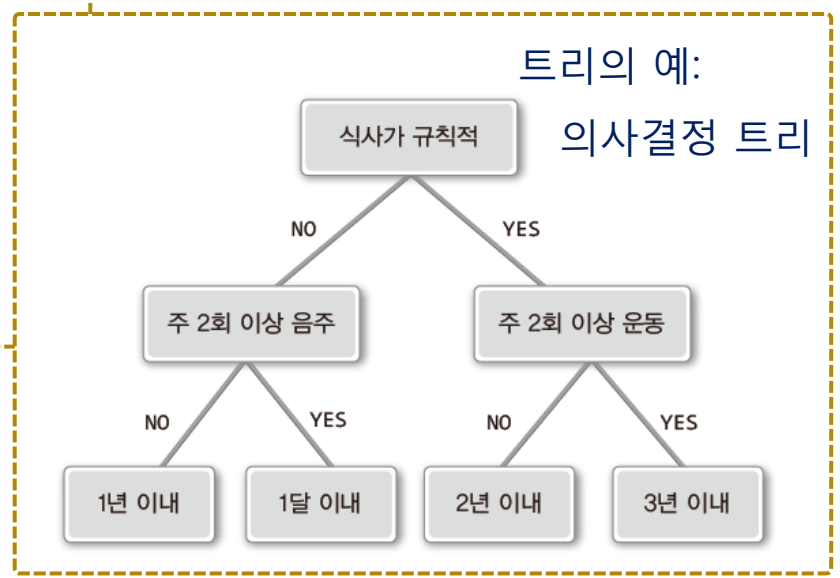
Chapter 08-1:

트리의 개요

트리의 접근과 이해

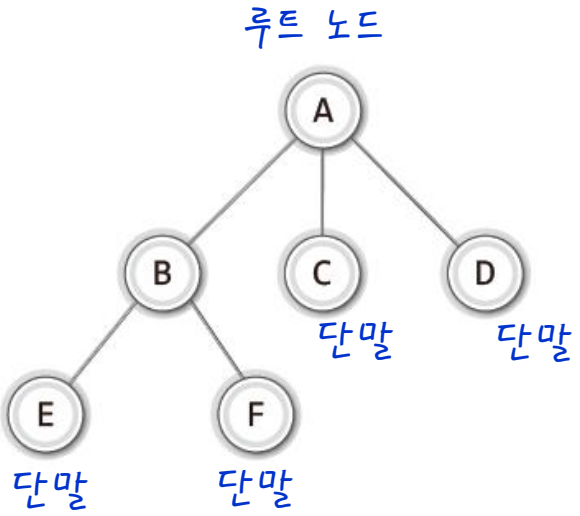
"트리는 계층적 관계(Hierarchical Relationship)를 표현하는 자료구조이다."





트리는 단순한 데이터의 저장을 넘어서
데이터의 표현을 위한 도구이다!

트리 관련 용어의 소개



- 노드: node
  트리의 구성요소에 해당하는 A, B, C, D, E, F와 같은 요소

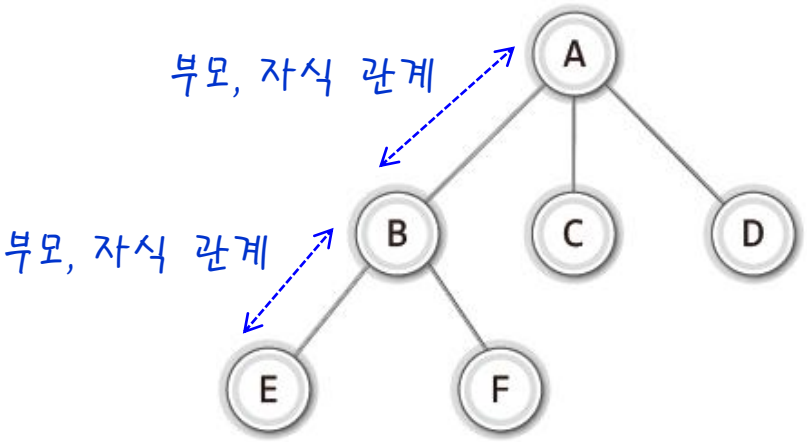
- 간선: edge
  노드와 노드를 연결하는 연결선

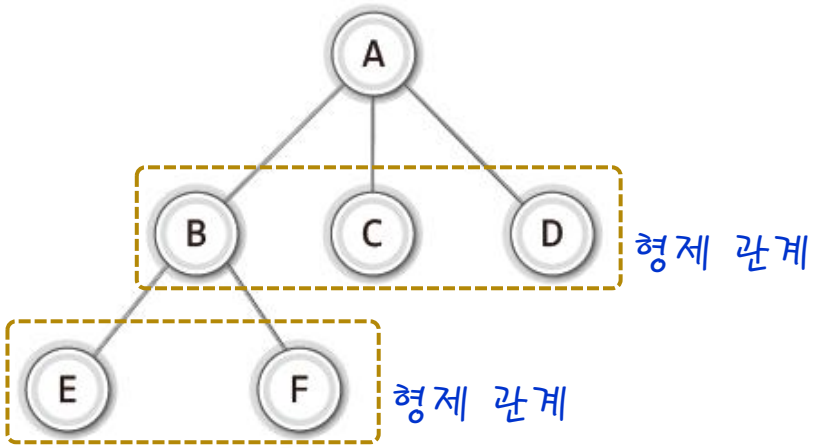
- 루트 노드: root node
  트리 구조에서 최상위에 존재하는 A와 같은 노드

- 단말 노드: terminal node
  아래로 또 다른 노드가 연결되어 있지 않은 E, F, C, D와 같은 노드

- 내부 노드: internal node
  단말 노드를 제외한 모든 노드로 A, B와 같은 노드

트리의 노드간 관계

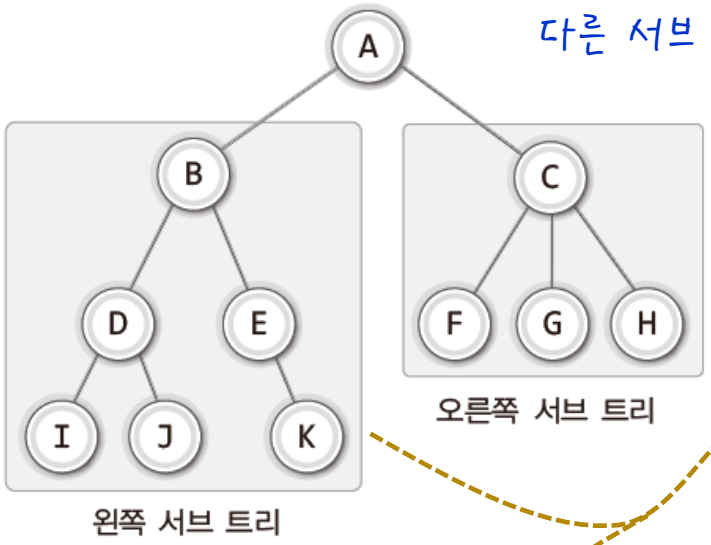




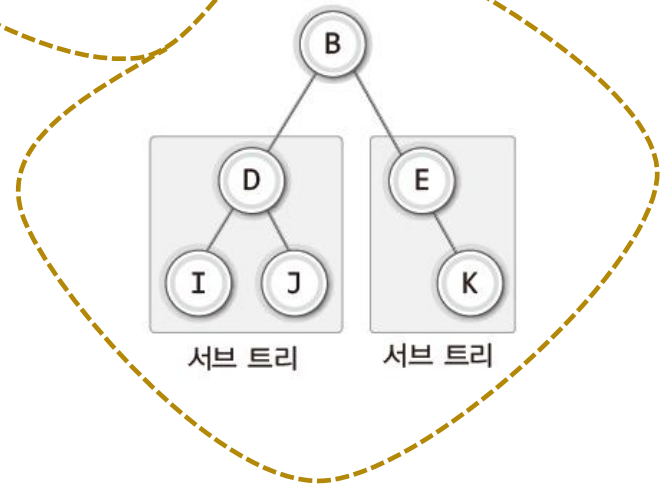
- 노드 A는 노드 B, C, D의 부모 노드(parent node)이다.
- 노드 B, C, D는 노드 A의 자식 노드(child node)이다.
- 노드 B, C, D는 부모 노드가 같으므로, 서로가 서로에게 형제 노드(sibling node)이다.

서브 트리의 이해

서브 트리 역시 서브 트리로 이뤄져 있으며, 그 서브 트리 역시 또 다른 서브 트리로 이뤄져 있다. 이렇듯 트리는 그 구조가 재귀적이다!



하나의 트리를 구성하는 왼쪽과 오른쪽의 작은 트리를 가리켜 서브 트리라 한다.

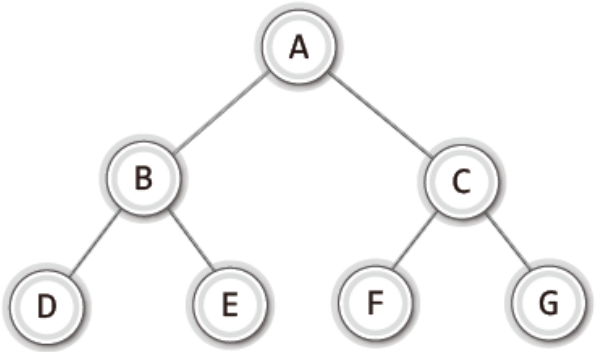


서브 트리 역시 또 다른 서브 트리를 갖는다!

이진 트리의 이해

이진 트리의 조건

- 루트 노드를 중심으로 두 개의 서브 트리로 나뉘어진다.
- 나뉘어진 두 서브 트리도 모두 이진 트리이어야 한다.



이진 트리의 예

"이진 트리가 되려면, 루트 노드를 중심으로 둘로 나뉘는 두 개의 서브 트리도 이진 트리이어야 하고, 그 서브 트리의 모든 서브 트리도 이진 트리이어야 한다."

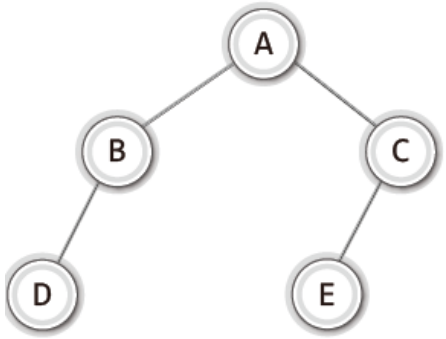
재귀적인 성향을 담아내지 못한 완전하지 않은 표현

트리 그리고 이진 트리는 그 구조가 재귀적이다!

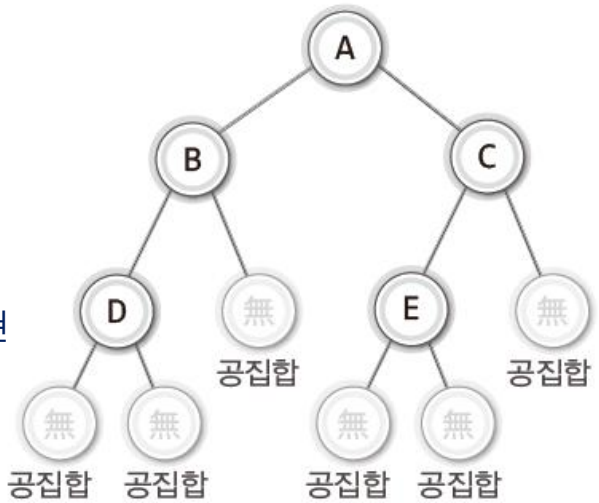
따라서 트리와 관련된 연산은 재귀적으로 사고하고 재귀적으로 구현할 필요가 있다!

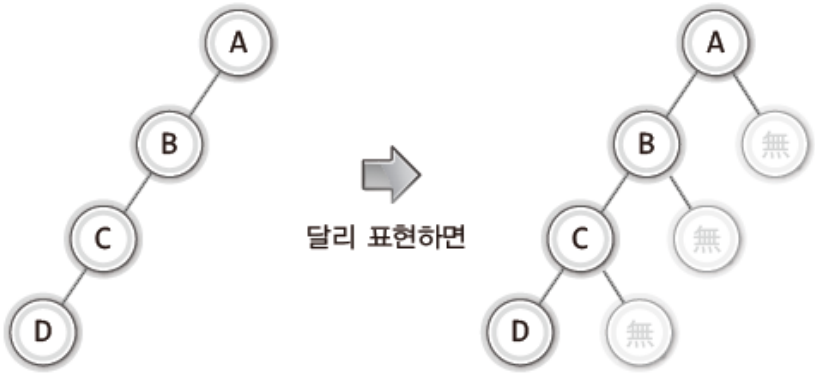
이진 트리와 공집합 노드

공집합(empty set)도 이진 트리에서는 노드로 간주한다!



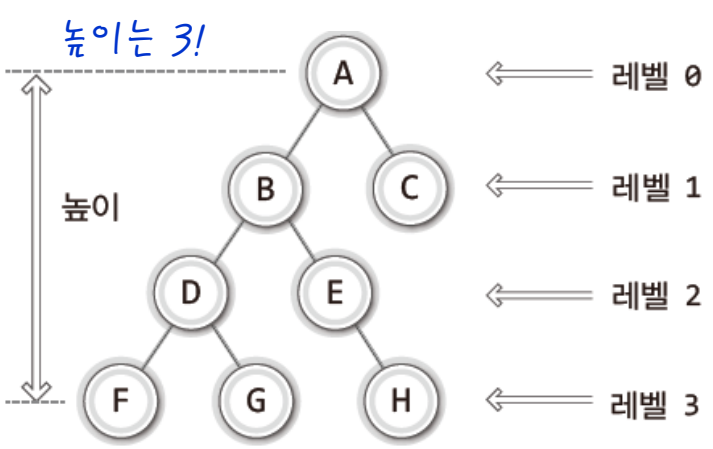
다른 표현



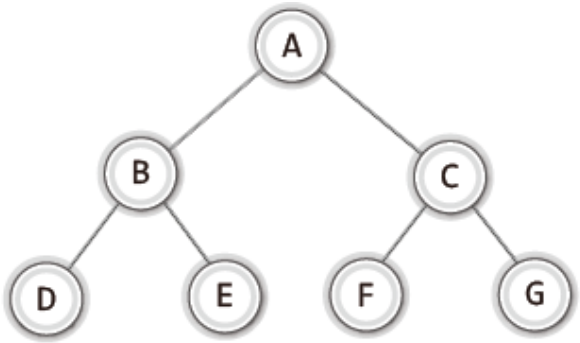


하나의 노드에 두 개의 노드가 달리는 형태의 트리는 모두 이진 트리이다.

레벨과 높이, 그리고 포화, 완전 이진 트리



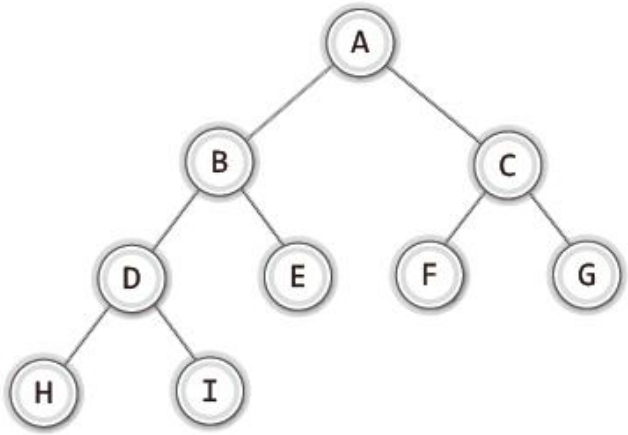
트리의 높이와 레벨의 최대 값은 같다!



모든 레벨에 노드가 꽉 찬! 포화 이진 트리

완전 이진 트리는 위에서 아래로 왼쪽에서 오른쪽으로 채워진 트리를 의미한다.

따라서 포화 이진 트리는 동시에 완전 이진 트리이지만 그 역은 성립하지 않는다.

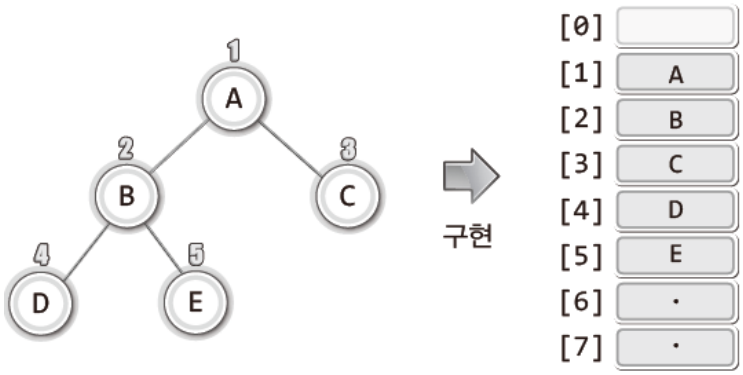


빈 틈 없이 차곡차곡 채워진! 완전 이진 트리

Chapter 08-2:

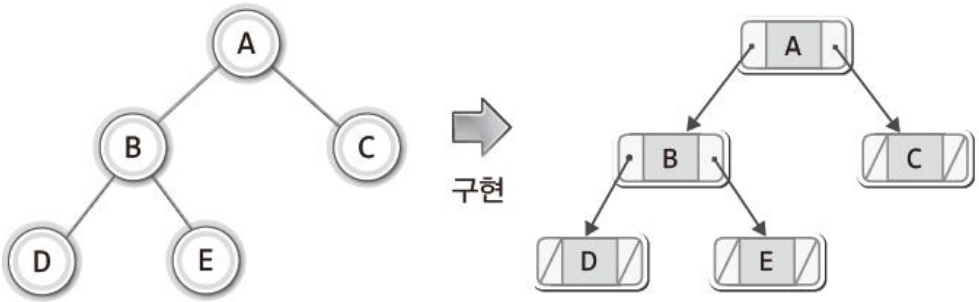
이진 트리의 구현

이진 트리 구현의 두 가지 방법



- 노드에 번호를 부여하고 그 번호에 해당하는 값을 배열의 인덱스 값으로 활용한다.
- 편의상 배열의 첫 번째 요소는 사용하지 않는다.

배열의 기본적인 장점! 접근이 용이하다라는 특성이 트리에서도 그대로 반영이 된다! 뿐만 아니라 배열을 기반으로 했을 때 완성하기 용이한 트리 관련 연산도 존재한다.



연결 리스트 기반에서는 트리의 구조와 리스트의 연결 구조가 일치한다.
따라서 구현과 관련된 직관적인 이해가 더 좋은 편이다.

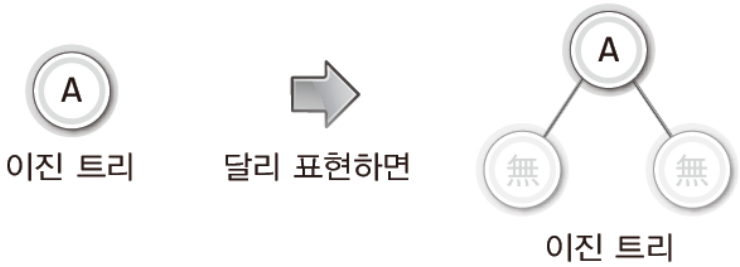
헤더파일에 정의된 구조체의 이해

```
// 이진 트리의 노드를 표현한 구조체
typedef struct _bTreeNode
{
    BTData data;
    struct _bTreeNode * left;
    struct _bTreeNode * right;
} BTreeNode;
```

이것이 노드이자 이진 트리를 표현한 구조체의 정의이다!

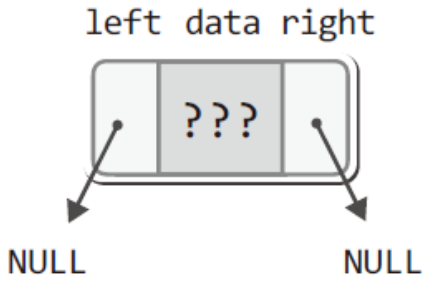
이진 트리의 모든 노드는 직/간접적으로 연결되어 있다. 따라서 루트 노드의 주소 값만 기억하면, 이진 트리 전체를 가리키는 것과 다름이 없다.

논리적으로도 하나의 노드는 그 자체로 이진 트리이다. 따라서 노드를 표현한 구조체는 실제로 이진 트리를 표현한 구조체가 된다.



헤더파일에 선언된 함수들1

- BTreeNode * MakeBTreeNode(void); // 노드의 생성



이러한 형태의 노드를 동적으로 할당하여 생성한다!
유효한 데이터는 SetData 함수를 통해서 채우되 포인터 변수 left와 right는 NULL로 자동 초기화 된다.

- BTData GetData(BTreeNode * bt); // 노드에 저장된 데이터를 반환

- void SetData(BTreeNode * bt, BTData data); // 노드에 데이터를 저장

노드에 직접 접근하는 것보다. 함수를 통한 접근이 보다 칭찬받을 수 있는 구조!

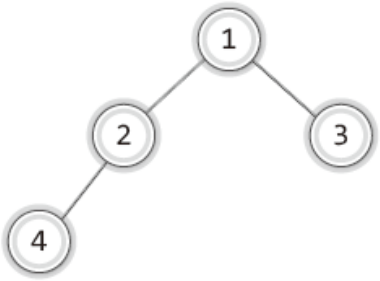
헤더파일에 선언된 함수들2

- BTreeNode * GetLeftSubTree(BTreeNode * bt);

  왼쪽 서브 트리의 주소 값 반환!

- BTreeNode * GetRightSubTree(BTreeNode * bt);

  오른쪽 서브 트리의 주소 값 반환!



√ 루트 노드를 포함하여 어떠한 노드의 주소 값도 인자로 전달될 수 있다.

√ 전달된 노드의 왼쪽, 오른쪽 '서브 트리의 루트 노드 주소 값' 또는 그냥 '노드의 주소 값'이 반환된다.

- 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!

하나의 노드도 일종의 이진 트리이다! 따라서 위와 같이 함수를 이름을 짓는 것이 타당하다!

위의 함수들은 단순히 노드가 아니라 트리를 대상으로도 그 결과를 보인다는 사실을 기억하자!

정의된 함수들의 이해를 돕는 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...

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

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

    . . . .
}
```

앞서 정의한 이진 트리 관련 함수들은 이진 트리를 만드는 도구이다!

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;
}
```

구현 자체는 크게 어려움이 없다!

```
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;
}
```

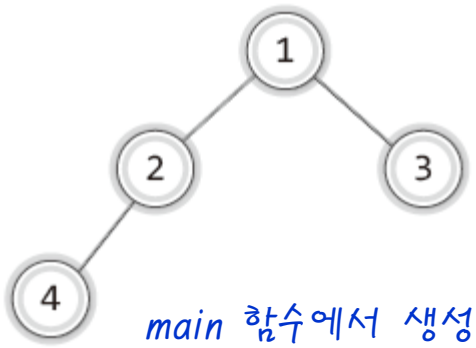
기존에 연결된 노드는 삭제되게 구현!

기존에 연결된 노드는 삭제되게 구현!

이진 트리 관련 main 함수

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

    SetData(bt1, 1);    // bt1에 1 저장
    SetData(bt2, 2);    // bt2에 2 저장
    SetData(bt3, 3);    // bt3에 3 저장
    SetData(bt4, 4);    // bt4에 4 저장

    MakeLeftSubTree(bt1, bt2);          // bt2를 bt1의 왼쪽 자식 노드로
    MakeRightSubTree(bt1, bt3);         // bt3를 bt1의 오른쪽 자식 노드로
    MakeLeftSubTree(bt2, bt4);          // bt4를 bt2의 왼쪽 자식 노드로

    // bt1의 왼쪽 자식 노드의 데이터 출력
    printf("%d \n", GetData(GetLeftSubTree(bt1)));

    // bt1의 왼쪽 자식 노드의 왼쪽 자식 노드의 데이터 출력
    printf("%d \n", GetData(GetLeftSubTree(GetLeftSubTree(bt1))));

    return 0;
}
```

트리를 완전히 소멸시키는 방법은? ☞ 순회!



main 함수에서 생성하는 트리

BinaryTree.h
BinaryTree.c
BinaryTreeMain.c

실행결과

```
2
4
```

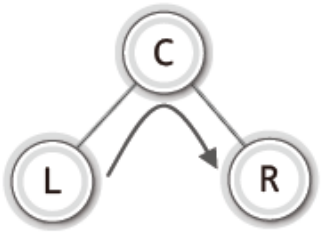
Chapter 08-3:

이진 트리의 순회(Traversal)

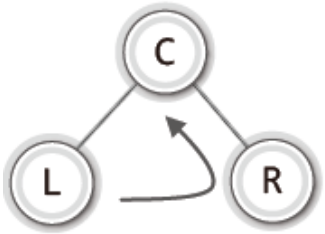
순회의 세 가지 방법

기준은 루트 노드를 언제 방문하느냐에 있다!

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



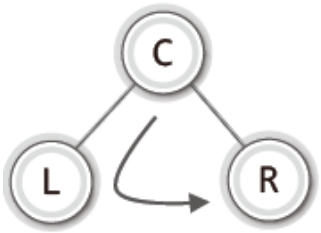
▶ [그림 08-21: 중위 순회]



▶ [그림 08-22: 후위 순회]

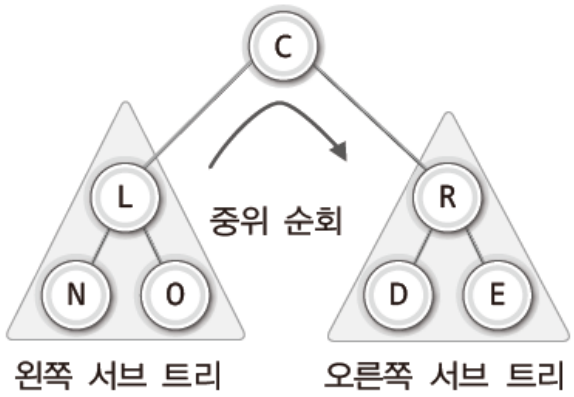
높이가 2 이상인 트리의 순회도 이와 다르지 않다.

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

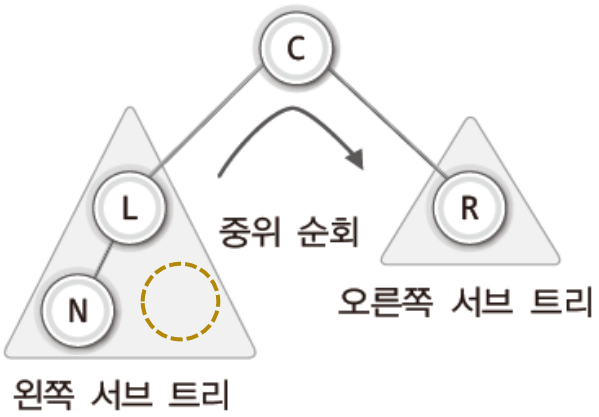
순회의 재귀적 표현 완성!

```
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;
}
```

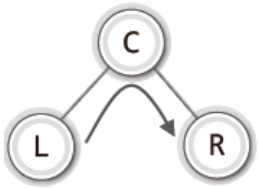
BinaryTree.h
BinaryTree.c
BinaryTreeTraverseMain.c

실행결과

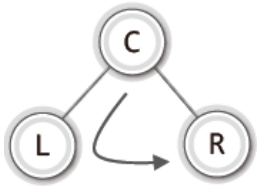
```
4
2
1
3
```

전위 순회와 후위 순회

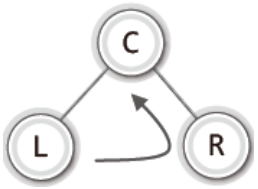
```
void InorderTraverse(BTreeNode * bt)
{
     if(bt == NULL)     // bt가 NULL이면 재귀 탈출!
         return;

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




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

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




```
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

Chapter 08-4:

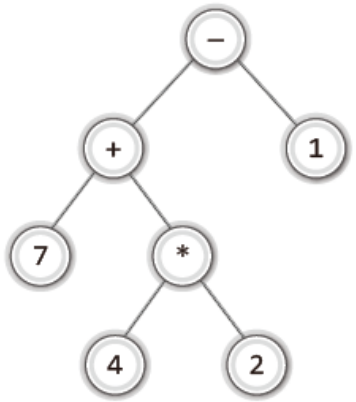
수식 트리(Expression Tree)의 구현

수식 트리의 이해

중위 표기법의 수식을 수식 트리로 변환하는 프로그램의 작성이 목적!

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

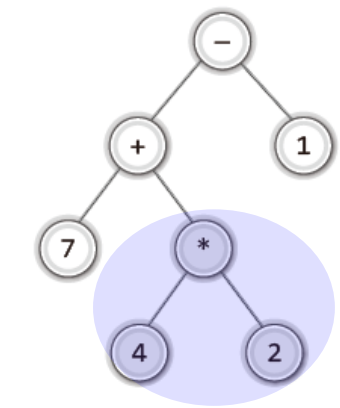
수식 트리



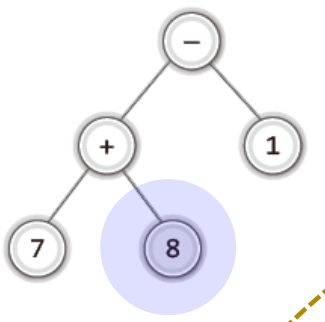
두 개의 자식 노드에는 피연산자를!

- 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

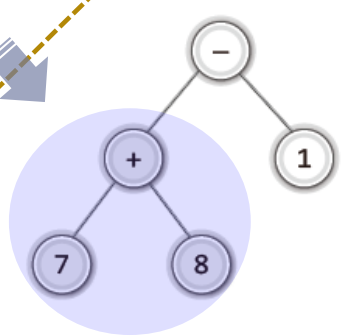


곱셈 연산 후

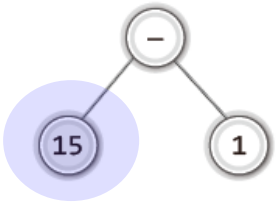


▶ [그림 08-27: 수식 트리의 연산과정 1/3]

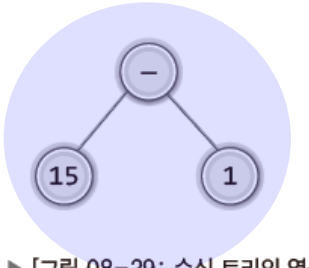
두 개의 자식 노드기 피연산자라는
단순하지만 전부인 하나의 특성을 근거로
연산이 매우 쉽게 진행된다!



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

트리 만드는 도구를 기반으로 함수를 정의한다!

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

```
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
```

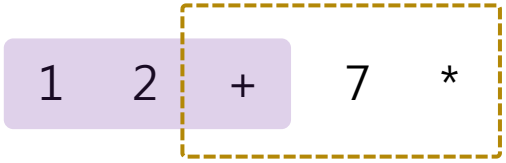
MakeExpTree가 구성한 수식 트리의 수식을 계산하여 그 결과를 반환한다!

```
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

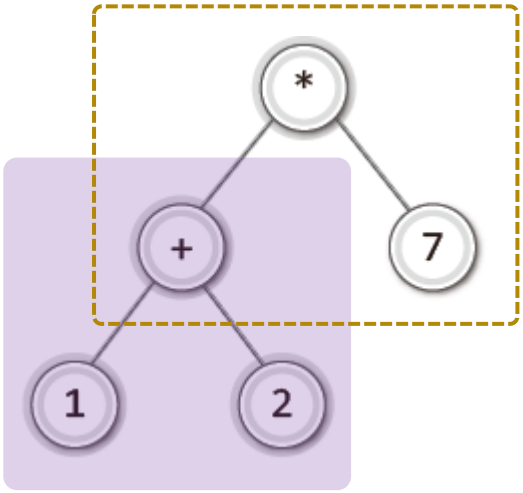
각각 전위, 중위, 후위 표기법의 수식이 출력된다.

수식 트리의 구성 방법: 그림상 이해하기



후위 표기법의 수식

수식 트리



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

이 사실을 이해하는 것과 이를 코드로 옮기는 것은 다소 다른 문제이다!

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]

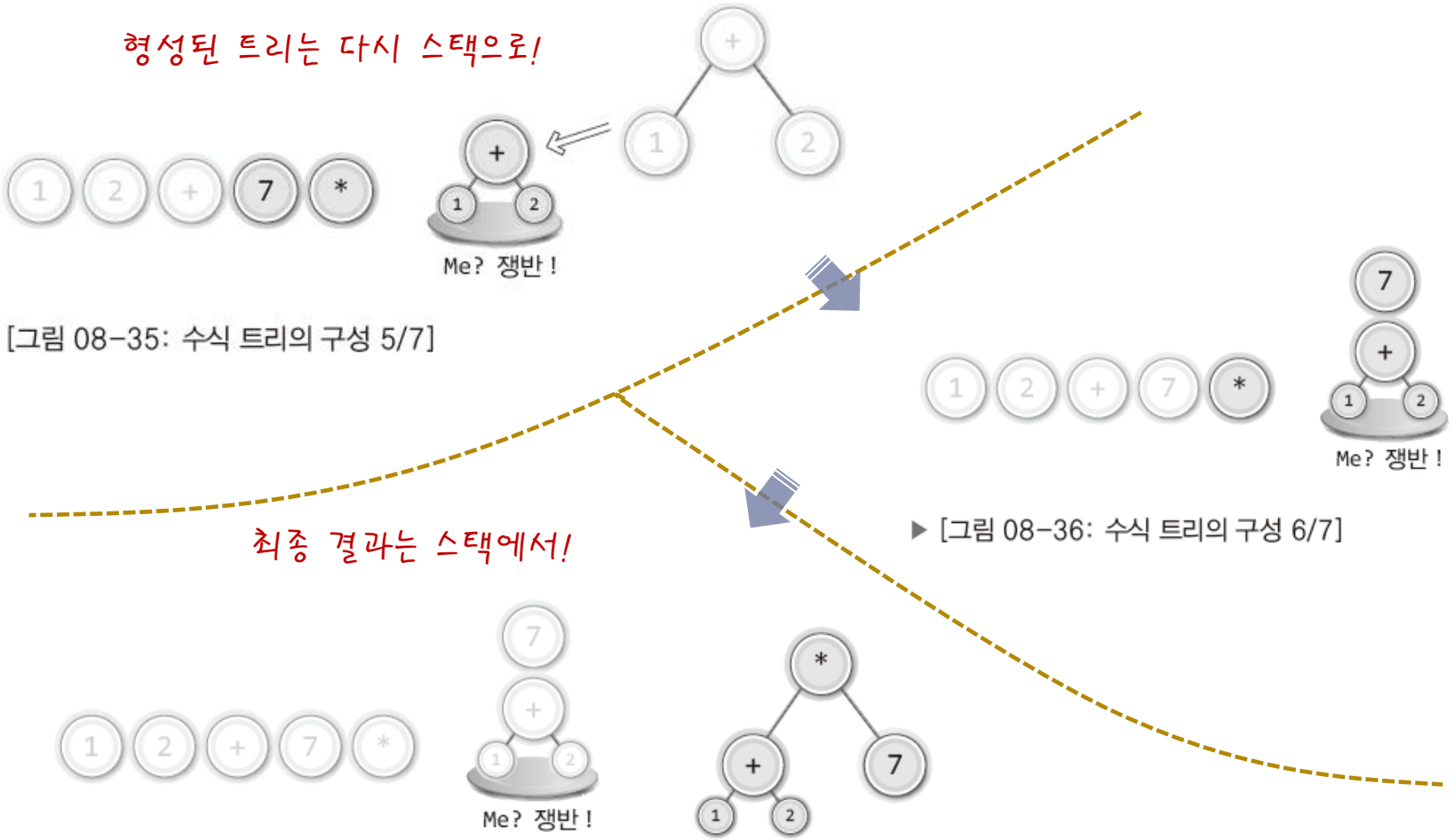
▶ [그림 08-33 : 수식 트리의 구성 3/7]

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;
}
```

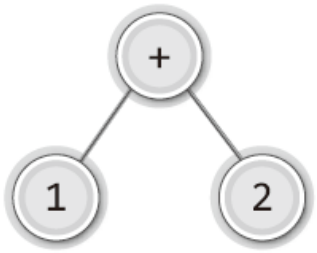
- 이진 트리 관련
  BinaryTree2.h, BinaryTree2.c
- 스택 관련
  ListBaseStack.h, ListBaseStack.c
- 수식 트리 관련
  ExpressionTree.h, ExpressionTree.c
- main 함수 관련
  ExpressionMain.c

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;
}
```



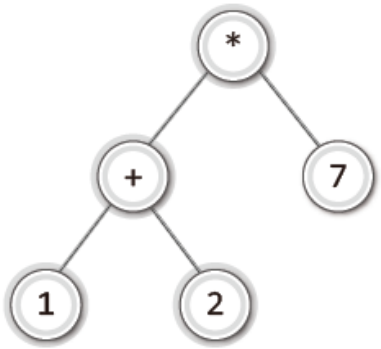
이 모델을 대상으로 한 구현

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));
```



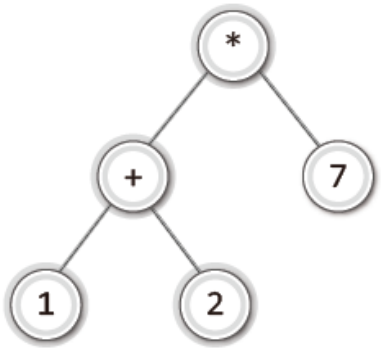
이 모델을 대상으로 한 구현

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!

수고하셨습니다~

Chapter 08에 대한 강의를 마칩니다!