10장. 트리

❑ 트리

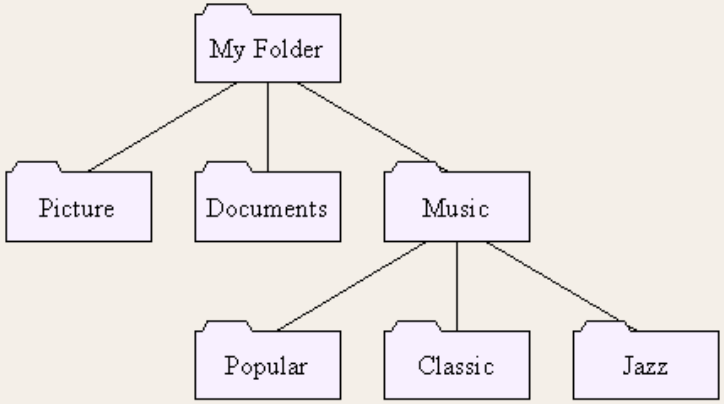
- 리스트나 스택, 큐가 데이터 집합을 한 줄로 늘어세운 선형 자료구조라면 트리는 두 갈래로 나누어 세운 비 선형 구조이다.비선형 구조를 고안한 동기는 바로 이 구조가 지닌 효율성 때문이다. 즉, 삽입, 삭제, 검색이라는 주 작업에 대해서 선형 구조보다 나은 시간적 효율을 보일 수 있다.

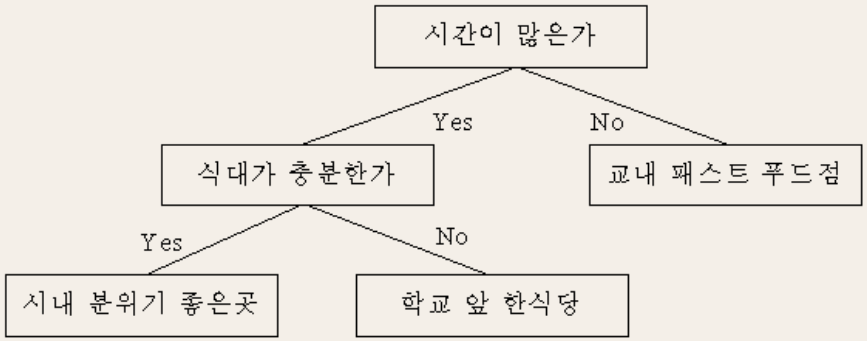
❑ 학습목표

- 트리와 관련된 용어를 정확히 이해한다.
- 이진 트리의 세 가지 순회방법의 차이점을 이해한다.
- 포인터로 구현한 이진 탐색트리의 탐색, 삽입, 삭제 코드를 이해한다.
- 이진 탐색트리의 균형에 따른 효율의 차이를 이해한다.

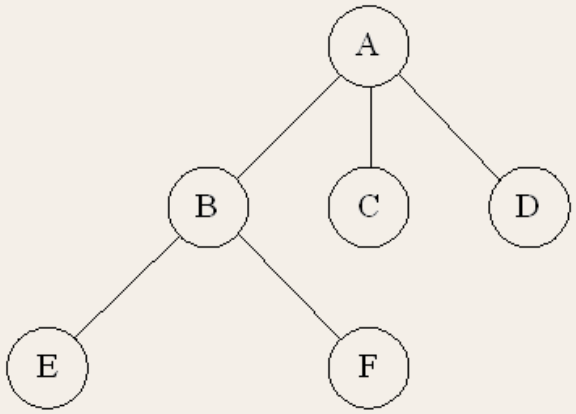
트리

- 트리
  - 계층구조를 표현: 디렉터리 구조, 기업 구조도, 족보, 결정트리



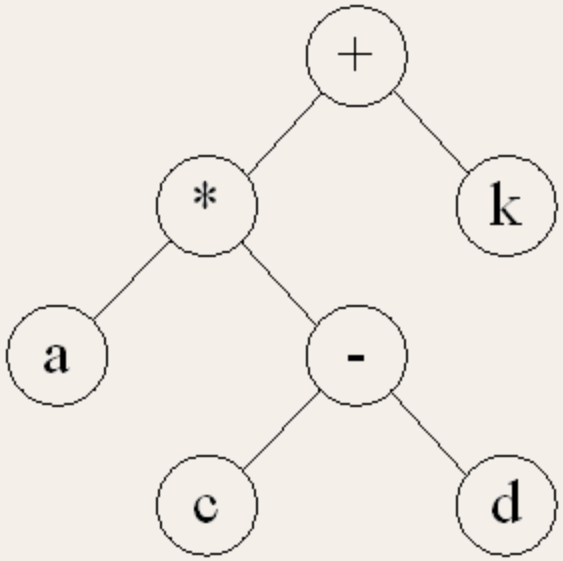


트리 관련용어



- 트리의 구성 요소에 해당하는 A, B, C, ..., F를 노드라 함. B의 바로 아래 있는 E, F를 B의 자식노드라 함. B는 E, F의 부모노드이고, A는 B, C, D의 부모노드임. 같은 부모 아래 자식 사이에 서로를 자매노드라 함. B, C, D는 서로 자매노드이며, E, F도 서로 자매노드임.주어진 노드의 상위에 있는 노드들을 조상노드라 함. B, A는 F의 조상노드임. 어떤 노드의 하위에 있는 노드를 후손노드라 함. B, E, F, C, D는 A의 후손노드임. 부모가 없는 노드 즉, 원조격인 노드를 루트 노드라 함. C, D, E, F처럼 자식이 없는 노드를 리프노드라 함. 리프노드를 터미널 노드라 함. 리프노드를 제외한 모든 노드를 내부노드라 함. 주어진 트리의 부분집합을 이루는 트리를 서브트리라 함. 이는 임의의 노드와 그 노드에 달린 모든 후손노드를 합한 것임.주어진 트리에는 여러 개의 서브트리가 존재할 수 있음. 그림의 B, E, F가 하나의 서브트리이며, 또 C 자체로서 하나의 서브트리임. 둘 이상의 자식노드를 가질 수 있는 트리를 일반트리라 함. 최대 두 개까지의 자식노드를 가질 수 있는 트리를 이진트리라 함.

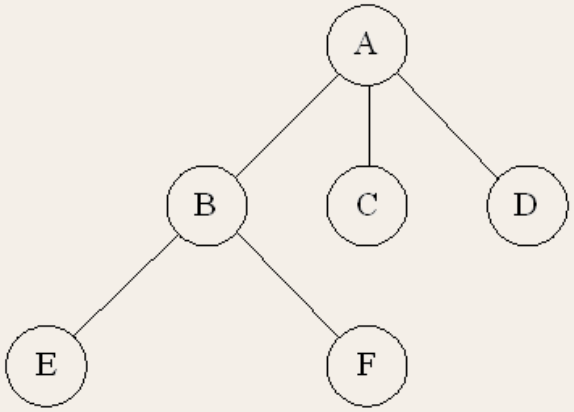
계산식 트리

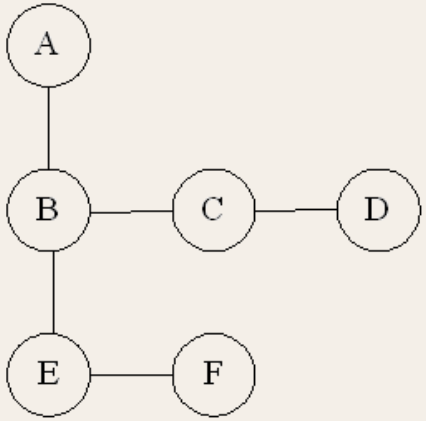


❑ 계산식 트리

- 컴파일 작업에 이용, 식 (a * (c - d)) + k를 평가하기 위한 것
- 연산자는 내부노드에, 피연산자는 리프노드에 위치
- 연산의 우선순위가 트리 모습 자체에 내장됨.
- (c - d)가 먼저 계산되어야 (a * (c - d))가 계산된다.

일반트리의 이진트리 변환





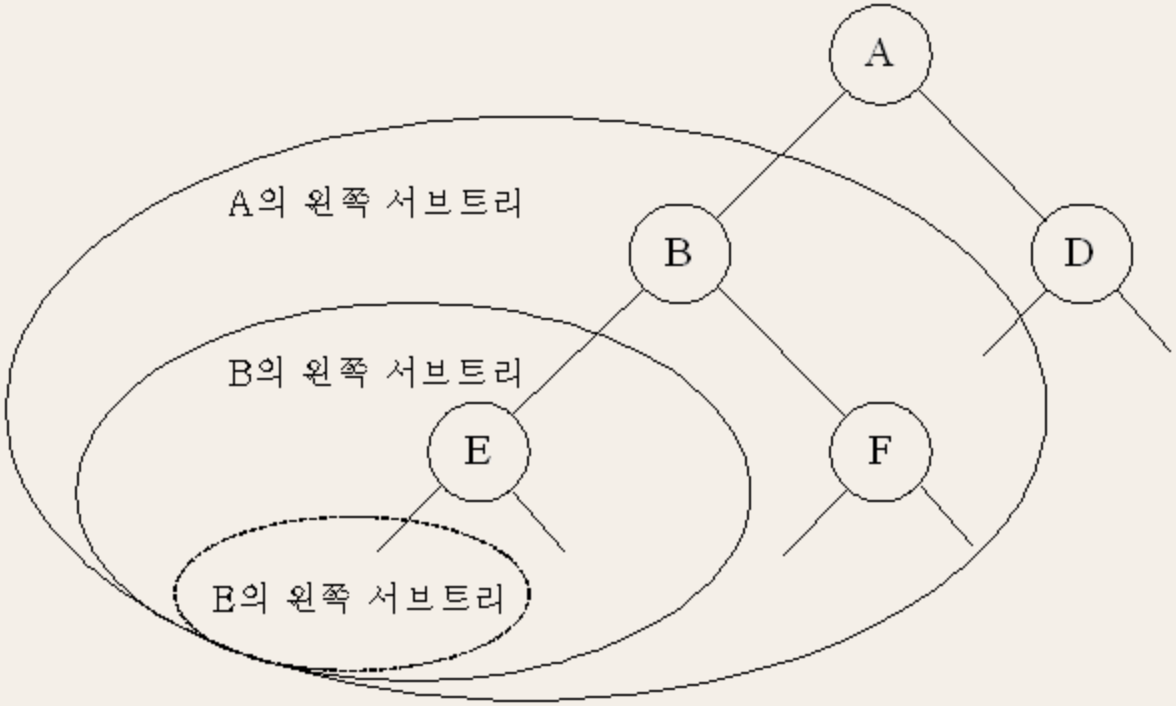
❑ 일반 트리

- 이진 트리로 바꾸어 표현 가능
- 일반트리의 자식노드 수는 가변. 따라서 확정적인 자료구조로 선언 불가
- 부모노드가 무조건 첫 자식노드를 가리키게
- 자식노드로부터 일렬로 자매 노드들을 이으면 이진 트리

이진트리 정의

재귀적 정의

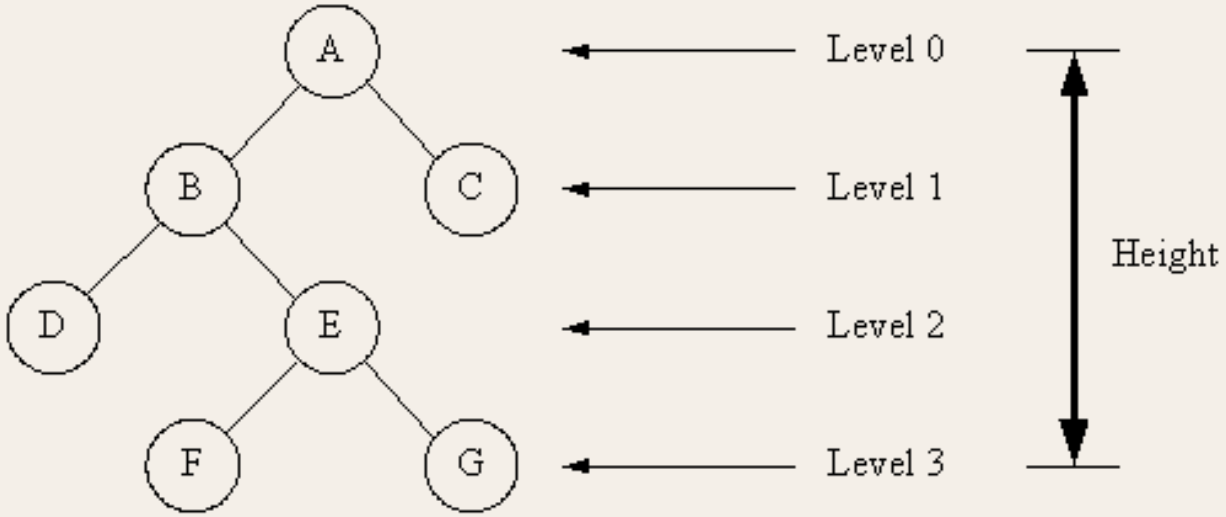
- 첫째, 아무런 노드가 없거나,
- 둘째, 가운데 노드를 중심으로 좌우로 나뉘되, 좌우 서브트리 모두 이진트리다.



트리의 레벨, 높이

❑ 트리의 레벨

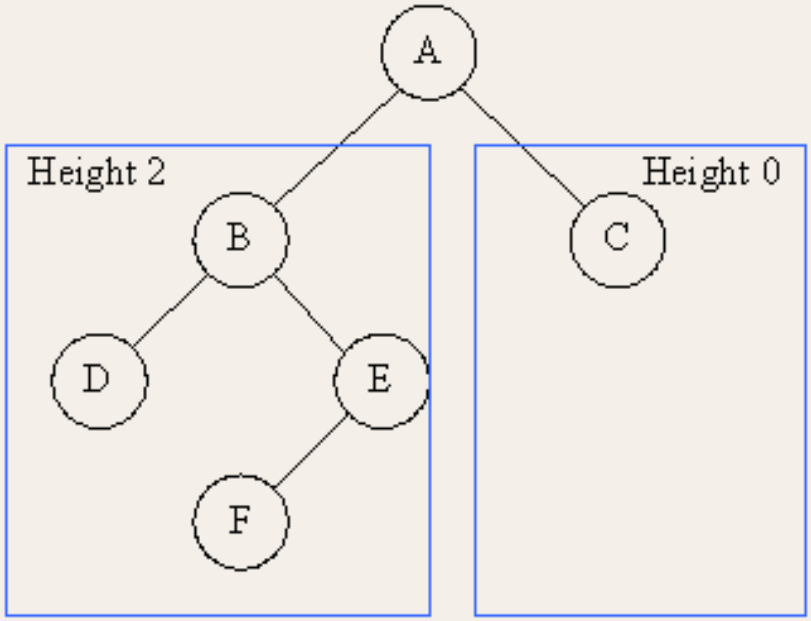
- 루트노드를 레벨 0으로 하고 아래로 내려오면서 증가
- 트리의 높이는 최대 레벨과 일치
- 루트만 있는 트리의 높이는 0
- 비어있는 트리의 높이는 -1



트리의 높이

재귀적 정의

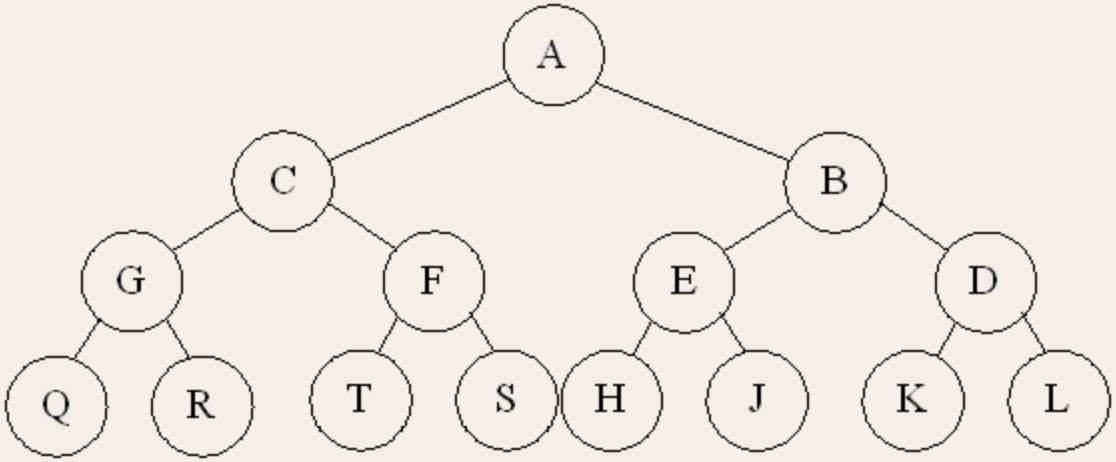
- 트리의 높이 = 1 + Max(왼쪽 서브트리의 높이, 오른쪽 서브트리의 높이)



포화 이진트리

❑ 포화 이진트리(飽和, Full Binary Tree)

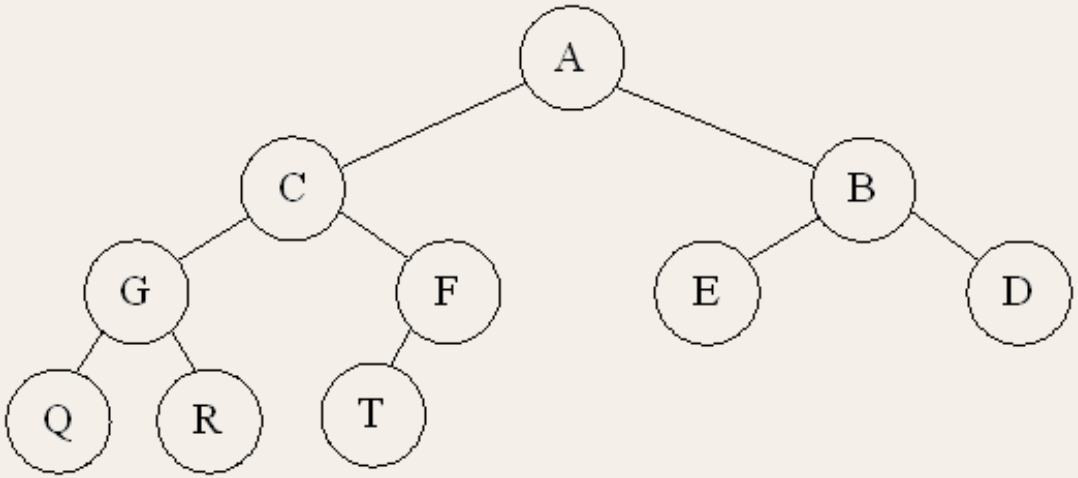
- 높이 h인 이진트리에서 모든 리프노드가 레벨 h에 있는 트리
- 리프노드 위쪽의 모든 노드는 반드시 두개의 자식노드를 거느려야 함.
- 시각적으로 볼 때 포화될 정도로 다 차 들어간 모습.



완전 이진트리

❑ 완전 이진트리(完全, Complete Binary Tree)

- 레벨 (h-1)까지는 포화 이진트리
- 마지막 레벨 h에서 왼쪽에서 오른쪽으로 가면서 리프노드가 채워짐.
- 하나라도 건너뛰고 채워지면 안됨.
- 포화 이진트리이면 완전 이진트리. 그러나 역은 성립 안 됨.



트리의 균형

- 균형(Balance)
  - 루트노드를 기준으로 왼쪽 서브트리와 오른쪽 서브트리 간의 높이 차이
- 균형트리(Height Balanced Tree)
  - 왼쪽, 오른쪽 서브트리 높이가 차이가 1 이하
- 완전 균형트리(Completely Height-Balanced Tree)
  - 왼쪽, 오른쪽 서브트리 높이가 완전히 일치하는 트리를 말한다.

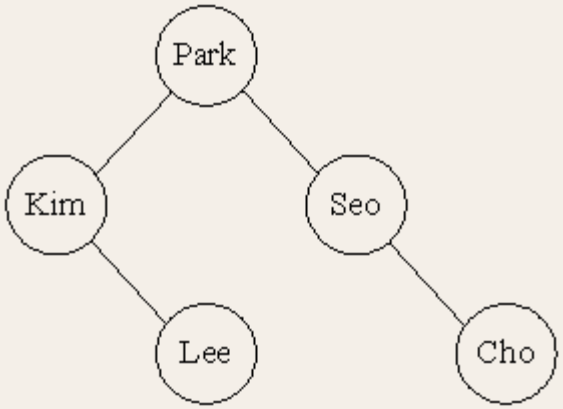
추상자료형 트리

작업

- Create: 새로운 트리를 만들기
- Destroy: 사용되던 트리를 파기하기
- Insert: 트리에 새로운 노드를 삽입하기
- Delete: 트리의 특정 노드를 삭제하기
- Search: 주어진 키를 가진 노드 레코드를 검색하기
- IsEmpty: 빈 트리인지를 확인하기
- CopyTree: 주어진 트리를 복사하기
- Traverse: 트리 내부 노드를 하나씩 순회하기

배열에 의한 이진트리

```
#define MAXNODE 100          최대 노드 수
typedef struct                    배열 요소는 구조체
{ char Name[ ];                   성명 필드
  int LChild;                     왼쪽 자식
  int RChild;                     오른쪽 자식
} node;
typedef node treeType[MAXNODE]; treeType은 구조체 배열 타입
```



Index	Structure		
	Name	LChild	RChild
0	Park	1	2
1	Kim	-1	3
2	Seo	-1	4
3	Lee	-1	-1
4	Cho	-1	-1
5	UnUsed		
6	UnUsed		
7	UnUsed		

배열에 의한 이진트리

- 인덱스
  - Park의 Lchild = 1이라는 것은 LChild인 Kim이 인덱스 1번에 있음을 의미
  - 인덱스 -1은 빈 트리(Empty Tree)를 의미

- Lee 삭제
  - 데이터 필드에서 Lee를 찾아서 해당 엔트리를 Unused로 표시
  - 나중에 그 공간을 재사용할 수 있도록 함.
  - 빈 공간을 관리하기 위해서는 삭제된 첫 노드를 가리키는 인덱스를 저장하고, 첫 노드의 RChild 필드가 두 번째 삭제된 노드를 가리키도록 체인을 형성

- Lee 삭제
  - 부모 노드인 Kim의 Rchild도 -1로
  - 부모노드를 찾기 위해서는 Lee의 인덱스 3을 가지고 배열 전체를 뒤져야 함.
  - 시간면에서 너무 비효율적임.
  - 배열에 의한 이진트리는 거의 사용 안 함 (예외: 완전 이진트리)

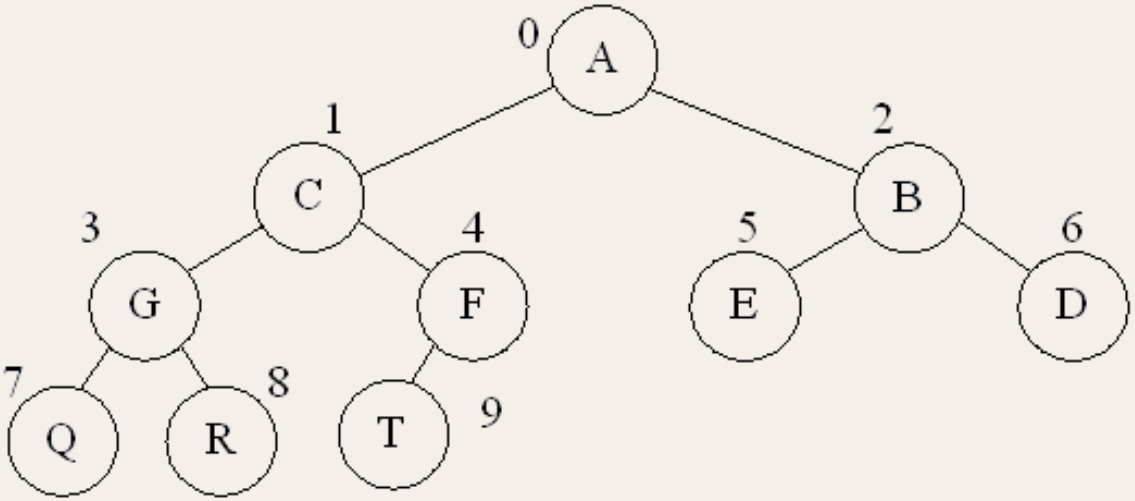
완전 이진트리와 배열

완전 이진트리와 배열

- 노트필기 순으로 배열에 저장
- 인덱스 K에 있는 노드의 왼쪽 자식은 (2K + 1)에 오른쪽 자식은 (2K + 2)에
- 인덱스 K에 있는 노드의 부모 노드는 (K - 1) / 2에 있다.

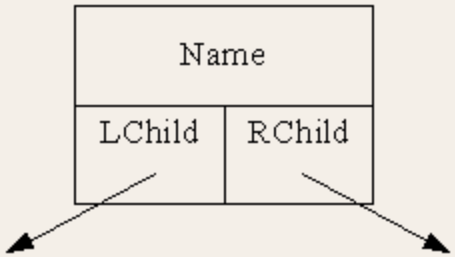
규칙 존재 이유

- 완전 이진트리의 정의



포인터에 의한 이진트리 구현

```
typedef struct
{ char Name[ ];      성명 필드
  node* LChild;      왼쪽 자식을 가리키는 포인터
  node* RChild;      오른쪽 자식을 가리키는 포인터
} node;              노드는 구조체 타입
typedef node* Nptr; 노드를 가리키는 포인터를 Nptr 타입으로 명명
```



이진트리 순회의 기본

- 트리 구성이유
  - 삽입, 삭제, 검색
  - 이를 위해서는 순회(Traversal)가 필요.
  - 트리 정의가 재귀적. 따라서 순회도 재귀적.

코드 10-1: 트리 순회

```
Traverse(T)
 { if (T is Empty)
  {
  }
  else
   {  Traverse(Left Subtree of T);
     Traverse(Right Subtree of T);
  }
 }
```

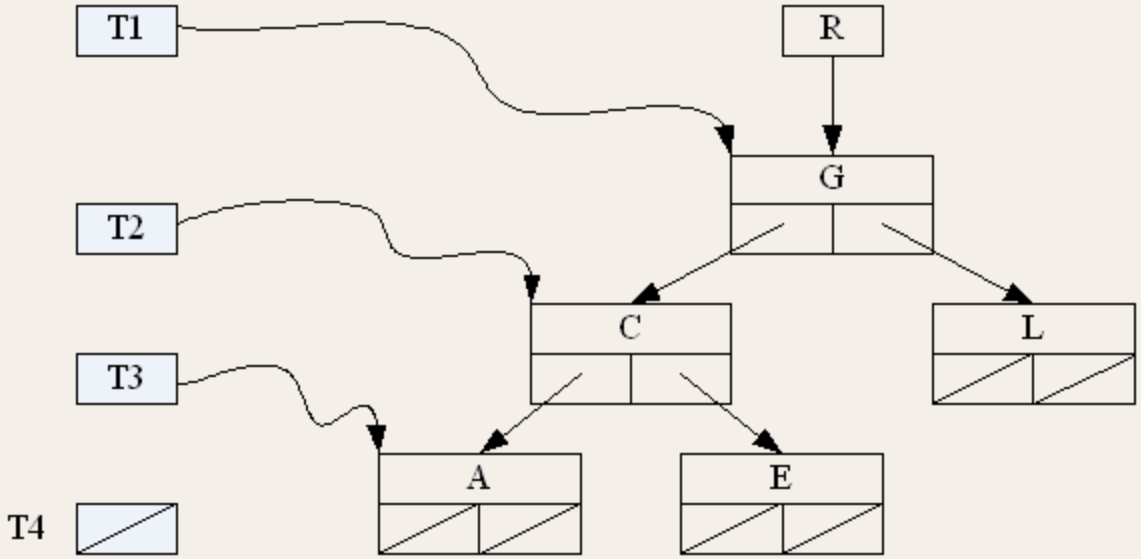
  - 왼쪽 서브트리와 오른쪽 서브트리로 나누어서 재귀호출

- 베이스 케이스
  - 처음부터 빈 트리를 의미하는 널.
  - 재귀호출에 의해 서브트리를 따라 내려가 리프노드의 왼쪽 오른쪽 서브트리인 LChild와 RChild 포인터 값이 널.

전위순회

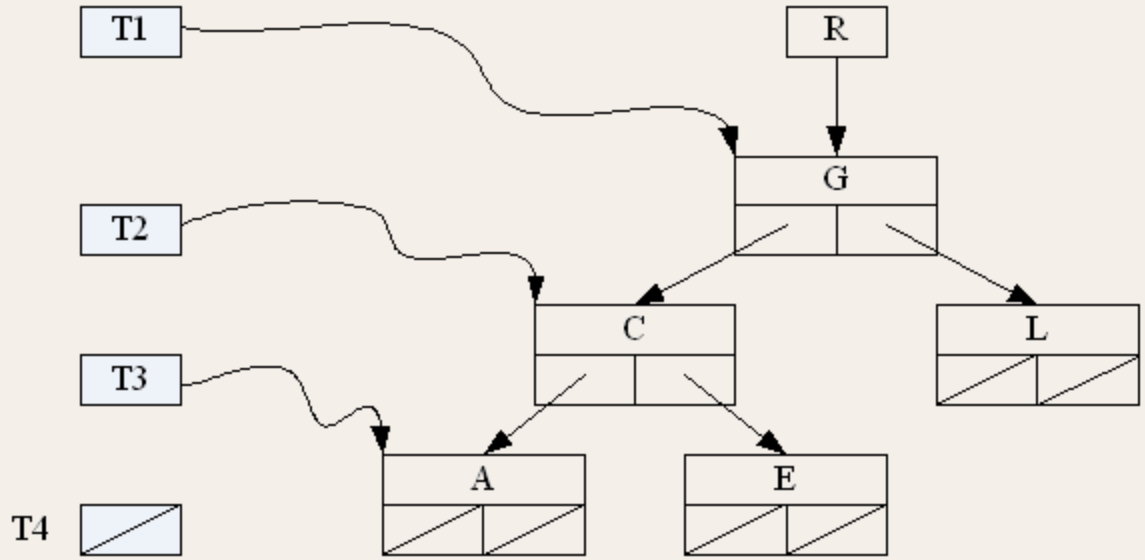
❑ 코드 10-2: 전위순회

```
void PreOrder(Nptr T)
{   if (T != NULL)
    {  Visit(T->Name);          전위, 아래 재귀호출보다 앞 위치
       PreOrder(T->LChild);     왼쪽 재귀호출
       PreOrder(T->RChild);     오른쪽 재귀호출
    }
}
```



전위순회

- 방문 순서: G, C, A, E, L
  - 호출함수가 루트노드 G를 가리키는 포인터 R을 넘김.
  - R이 복사되어 T = T1으로 들어감.
  - 데이터 G를 일단 찍고, G의 LChild를 넘기며 재귀호출
  - 이번에는 G의 LChild가 T = T2로 들어감.
  - 값호출에 의해 포인터 복사본이 넘겨짐.
  - 이 경우에는 읽기만 하는 작업이므로 사실상 참조호출이 필요 없음.



중위순회, 후위순회

❑ 방문의 위치

- 방문(데이터 처리) 명령의 위치에 따라서 중위 순회(中位, In-order Traversal), 후위 순회(後位, Post-order Traversal)로 구분.
- 전위, 중위, 후위라는 용어는 재귀호출의 상대적인 위치
- 전위, 중위, 후위 모두 각 노드를 단 한번씩만 방문. 따라서 효율은 O(N)

코드 10-3: 중위 순회

```
void InOrder(Nptr T)
{ if (T != NULL)
  {  InOrder(T->LChild);
     Visit(T->Name);
     InOrder(T->RChild);
  }
}
```

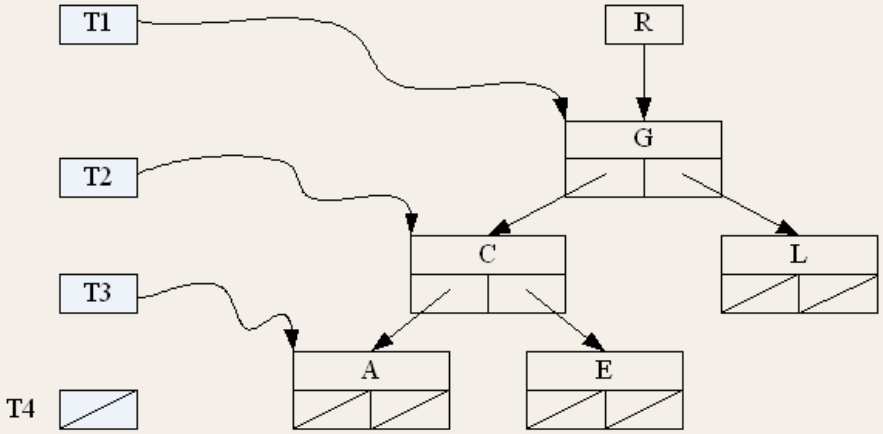
코드 10-4: 후위 순회

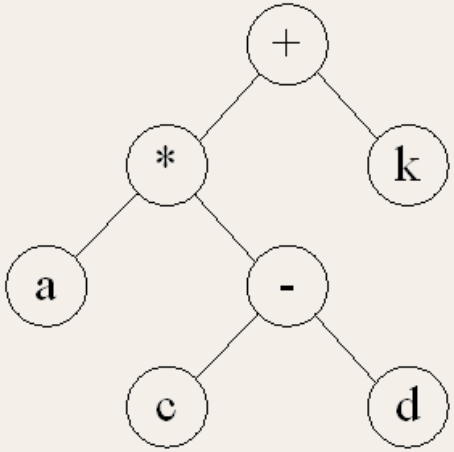
```
void PostOrder(Nptr T)
{ if (T != NULL)
  {  PostOrder(T->LChild);
     PostOrder(T->RChild);
     Visit(T->Name);
  }
}
```

전위, 중위, 후위표기

- 트리
  - 중위 순회 결과: A, C, E, G, L
  - 후위 순회 결과: A, E, C, L, G

- 계산식 트리
  - 전위표기: 전위 순회하면 + * a - c d k
  - 중위 표기: 중위 순회하면 a * c - d + k 프로그래머가 사용
  - 후위 표기: 후위 순회하면 a c d - * k + 컴파일러 번역 결과





레벨 순회

- 코드 10-5: 레벨 순회

```
void LevelOrder (Nptr T)
{ Q.Create( );                    새로운 큐를 만들고
  Q.Add(T);                       트리의 루트 포인터를 큐에 삽입
  while (! Q.IsEmpty( ))           큐가 비지 않을 때까지
  {   Nptr Current = Q.GetFront( ); 프런트 포인터를 꺼내
     Visit(Current->Name);         가리키는 노드를 방문하고
     if (Current->LChild != NULL)  왼쪽 자식이 있으면
       Q.Add(Current->LChild);     포인터를 큐에 삽입
     if (Current->RChild != NULL)  오른쪽 자식도 있으면
       Q.Add(Current->RChild);     포인터를 큐에 삽입
  }
}
```

스택에 의한 전위순회

코드 10-6: 스택에 의한 전위순회

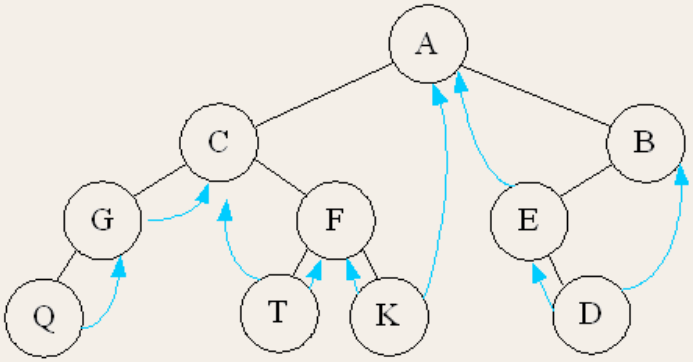
```
IterativePreorder (Nptr p)        루트를 가리키는 포인터 p
{  S.create( );                   새로운 스택을 만들기
   if (p != NULL)                 빈 트리가 아니면
   {  S.Push(p);                  루트를 스택에 푸쉬
      while (! S.IsEmpty( ))      스택이 비지 않을 때까지
      {  p = S.Pop( );            스택 탑을 꺼내서
         Visit(p);                일단 방문하고
         if (p->RChild != NULL)   오른쪽 자식이 있으면
            S.Push(p->RChild);    스택에 넣고
         if (p->LChild != NULL)   왼쪽 자식도 있으면
            S.Push(p->LChild);    스택에 넣는다.
      }
   }
}
```

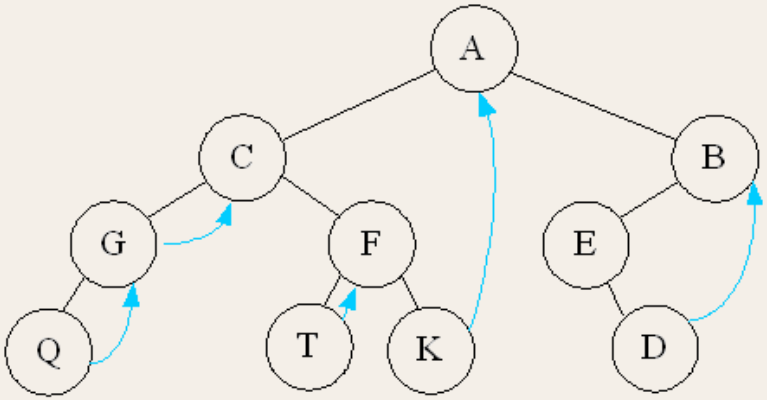
스택사용

- 시스템 스택이건 사용자 스택이건 스택 공간과 함께, 푸쉬 팝을 위한 실행시간이 요구됨.
- 스택의 사용을 완전히 배제함으로써 효율성을 추구할 수는 없는가.

스레드 이진트리

- 중위순회의 예
  - 노드마다 추가로 중위순회 선행자, 중위순회 후속자 포인터 추가
  - 이러한 포인터를 스레드(실, Thread)라 함.
  - 실제로는 중위순회 후속자만 있으면 되고, 또 Rchild가 널일 경우만 중위순회 후속자가 필요. 따라서 Rchild 필드를 스레드로 대용.
  - 재귀호출이나 사용자 스택 없이 중위순회
  - 스택 팝에 의해 이전 노드로 되돌아가는 대신 이전 포인터를 트리에 내장
  - 스레드 트리는 전위순회나 후위순회에도 이용





스레드 이진트리

- 코드 10-7: 스레드에 의한 중위순회

```
ThreadInorder (Nptr p)
{ if (p != NULL)
  {   while (p->LChild != NULL)
        p = p->LChild;              트리의 가장 왼쪽 자식으로
      while (p != NULL)             더 이상 순회할 곳이 없을 때까지
      {   Visit(p);                 일단 방문하고
        Prev = p;                   현재 노드를 Prev에 저장
        p = p->RChild;              오른쪽 자식노드 또는 중위후속자로 이동
        if (p != NULL && Prev->IsThread = = FALSE) 오른쪽 자식노드라면
          while (p->LChild != NULL)      다시 가장 왼쪽 자식으로
            p = p->LChild;
      }
  }
}
```

- 중위순회에서 가장 먼저 방문해야 할 것은 가장 왼쪽 자식
- 두 번째 while 루프 p는 현재 Q를 가리키고 있으므로 p = p->RChild에 의해서 중위 후속자인 G를 가리킴. 그림의 경우에는 Q의 RChild가 스레드로 사용. 따라서 중위 후속자인 G를 향해 순회
- 만약에 이것이 스레드가 아니라면 현재 p가 가리키는 것이 G가 아니라 Q의 오른쪽 자식노드임을 의미. 이 자식노드가 널이 아니라면 다시 그 자식노드로부터 가장 왼쪽 자식으로 가서 거기서부터 방문을 다시 시작

이진트리의 복사

- 코드 10-8: 이진트리의 복사

```
Nptr CopyTree(Nptr T)  T는 원본의 루트 포인터
{ if (T = = NULL)                원본 포인터가 가리키는 노드가 더 이상
  없다면
    return NULL;                 복사할 노드 포인터도 널로 세팅
 else                      원본 포인터에 매달린 노드가 있다면
  {   Nptr T1 = (node *)malloc(sizeof(node));    새로운 노드를 만들고
     T1->Name = T->Name;                    데이터를 복사한다.
      T1->LChild = CopyTree(T->LChild); 서브트리의 루트 포인터를 할당
     T1->RChild = CopyTree(T->RChild);  서브트리의 루트 포인터를 할당
      return T1;                         서브트리의 루트 포인터를 리턴
   }
 }
```

- 전위순회를 이용

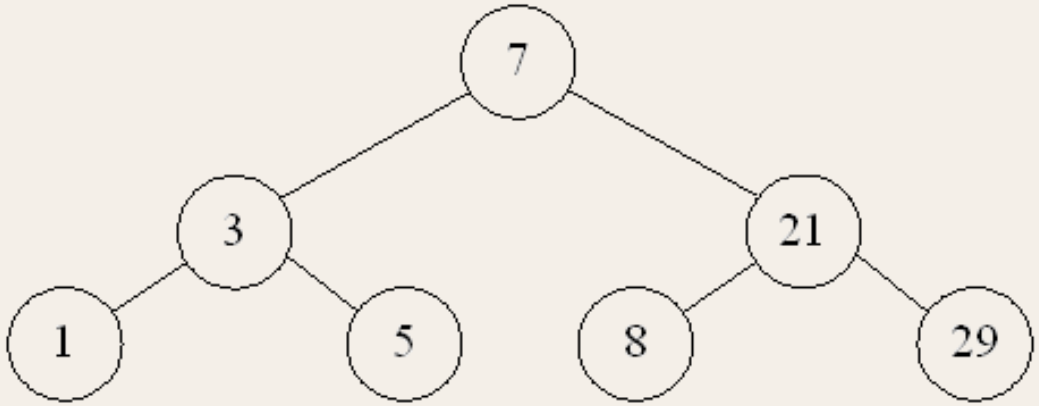
이진 탐색트리

```
typedef struct
{ int Key;            학번
  char Name[12];      성명
  char Address[200]   주소
} dataType;


typedef struct
{ dataType Data;      데이터는 하나의 구조체
  node* LChild;       왼쪽 자식을 가리키는 포인터
  node* RChild;       오른쪽 자식을 가리키는 포인터
} node;               구조체 타입 노드
```

이진 탐색트리

- 탐색에 유리. 노드 위치는 키 값을 기준으로 결정
- 모든 노두 N에 대하여,
  - N의 왼쪽 서브트리에 있는 노드는 모두 N보다 작고, N의 오른쪽 서브트리에 있는 노드는 모두 N보다 크다
  - N의 왼쪽 서브트리와 N의 오른쪽 서브트리도 이진 탐색 트리다.
- "약하게 정렬되어 있다"
  - 루트를 중심으로 왼쪽 서브트리는 작은 것, 오른쪽 서브트리는 큰 것
  - 쾌속정렬의 파티션과 유사. 루트 노드는 일종의 피벗에 해당
  - 피벗(Pivot)을 중심으로 작은 것은 모두 왼쪽, 큰 것은 모두 오른쪽
  - 서브트리만 놓고 볼 때도 다시 파티션 된 상태이다



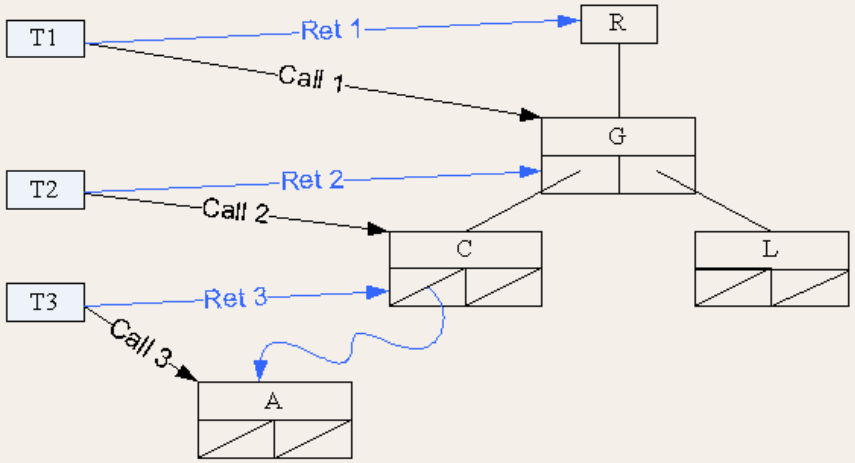
이진 탐색트리의 탐색

- 코드 10-9: 이진 탐색트리의 탐색

```
Nptr Search(Nptr T, int Key)
 { if (T = = NULL)
      printf("No Such Node");          오류 처리 후 빠져나옴
   else if (T->Data.Key = = Key)
      return T;                        찾아낸 노드 포인터를 리턴
   else if (T->Data.Key > Key)
      return Search(T->LChild, Key);Call by Value로 전달
   else
      return Search(T->RChild, Key); Call by Value로 전달
   }
```

이진 탐색트리의 삽입

```
Nptr Insert(Nptr T, int Key)
{ if (T = = NULL)                          리프노드의 Child 포인터
 {   T = (node*)malloc(sizeof(node));      삽입할 새 노드를 만들기
   T->Data.Key = Key;                      필요시 나머지 데이터 필드 복사
   T->LChild = NULL; T->RChild = NULL; 리프노드이므로 자식노드를 널로
 }
 else if (T->Data.Key > Key)
     T->LChild = Insert(T->LChild, Key); 왼쪽 서브트리로 재귀호출
 else
    T->RChild = Insert(T->RChild, Key);오른쪽 서브트리로 재귀호출
 return T;                                 현재 서브트리의 루트 포인터
 }
```

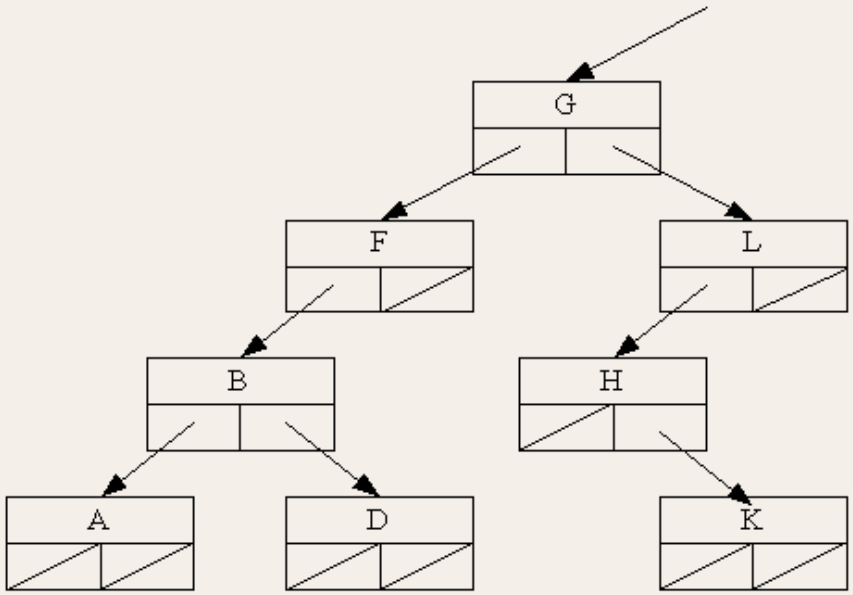


이진 탐색트리의 삽입

❑ 참조호출 효과

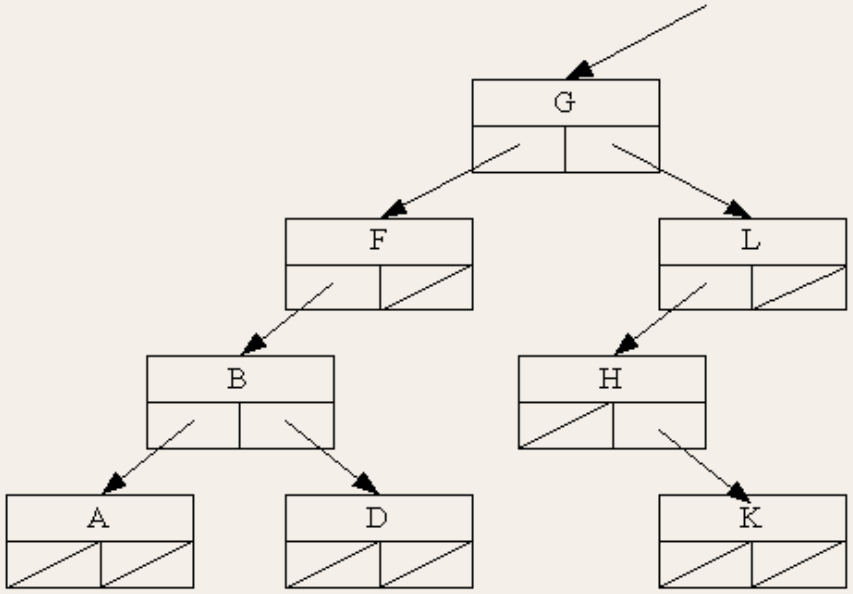
- 재귀호출에서 되돌아 오면서 트리 자체의 자식노드 포인터 값이 바뀜
- 베이스 케이스에서 T3가 호출함수에 리턴
- 호출함수에서는 그 값을 C의 LChild에 할당
- 같은 방식으로 T2가 G의 LChild로 할당된다.
- 같은 방식으로 T1이 R에 되돌려지면, 결국 R은 G를 가리킴
- 리프노드 C를 제외한 나머지 포인터 값은 원래의 값 그대로를 재할당.

이진 탐색트리의 삭제



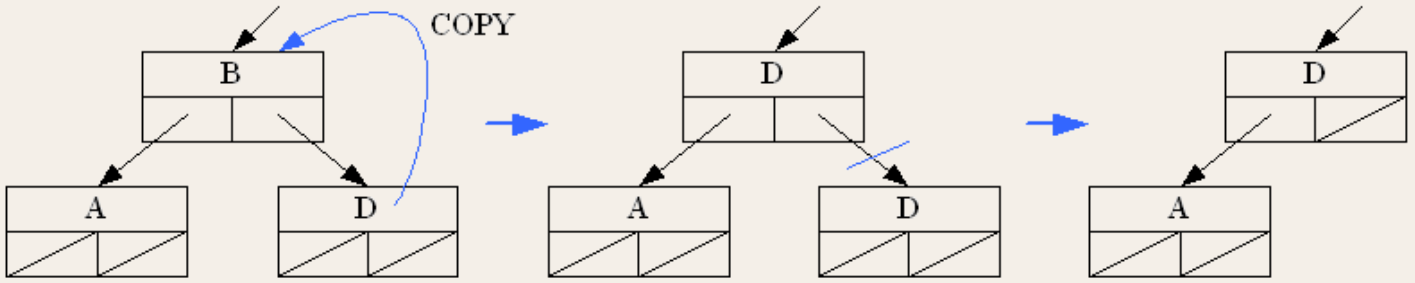
- 첫째, 삭제할 노드가 리프노드인 경우
  - 자식이 없어서 간단함.
  - 예를 들어 키 A인 노드를 삭제하려면 부모노드인 B의 LChild를 널로
  - 마찬가지로 노드 K를 삭제하려면 노드 H의 RChild를 널로 놓으면 된다.

이진 탐색트리의 삭제



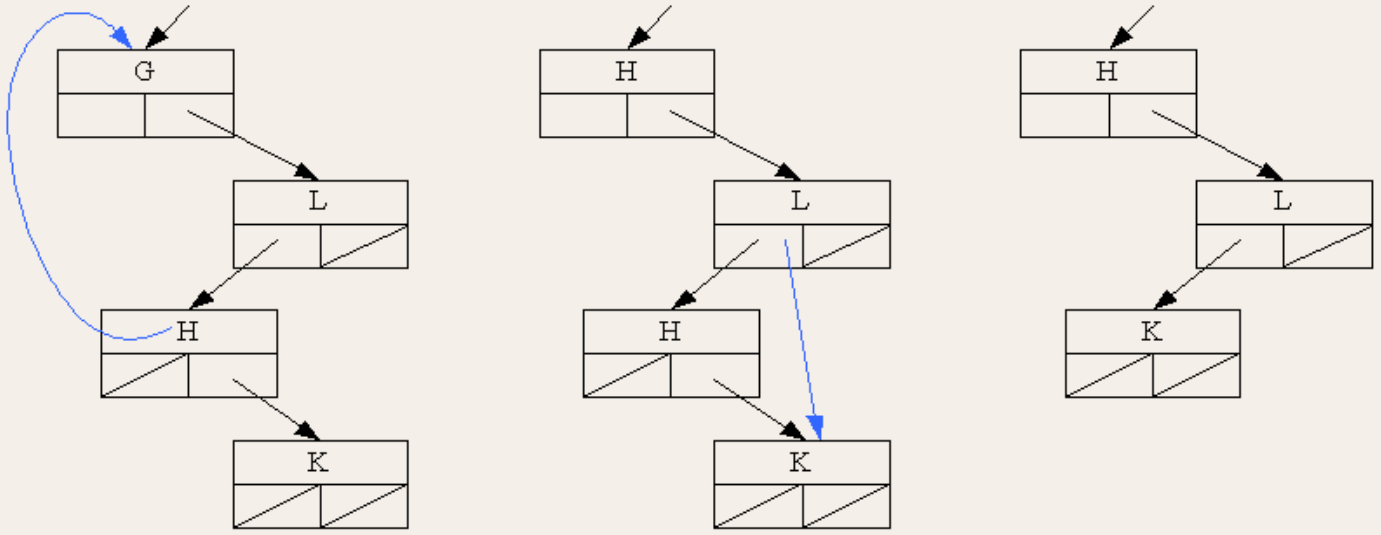
- **둘째, 삭제할 노드가 하나의 자식노드를 거느린 경우**
  - 왼쪽 자식 또는 오른쪽 자식. 역시 간단함. 노드 F를 삭제하려면 F의 부모노드가 F의 자식노드를 가리키면 됨. 마찬가지로 노드 L을 삭제하려면 G의 RChild가 H를 가리키면됨.
  - 이렇게 하더라도 이진 탐색트리의 특성은 유지
    - L, H, K 모두 G 보다 크기 때문에 G의 오른쪽 서브트리에 존재
    - L이 삭제되고 H가 G의 오른쪽 서브트리에 붙어도 이진 탐색트리의 크기 관계는 유지됨.

이진 탐색트리의 삭제



- 셋째, 삭제할 노드가 두개의 자식노드를 거느린 경우
  - 가장 복잡한 경우
  - 노드 B를 삭제. B의 부모인 F의 LChild가 동시에 A와 D를 가리키게?
  - D가 B로 복사. 원래의 D를 삭제.
  - 이진 탐색트리의 키 크기 관계는 유지
  - B의 자리를 B보다 큰 D가 차지하더라도, B보다 작았던 A가 불만할 수 없음.
  - A < B < D 관계이므로 B가 없어지면 당연히 A < D이고 따라서 A는 그대로 D의 왼쪽에 있어야 한다.

이진 탐색트리의 삭제



- G를 삭제하면 G의 자리로 복사되어 가야 할 것은?
  - 이진 탐색트리를 중위순회(In-order Traversal)하면 정렬된 결과
  - A, B, D, F, G, H, K, L. 크기 면에서 A < B < D < F < G < H < K < L
  - G가 삭제된 후데오 정렬된 순서를 그대로 유지하려면 G의 자리에는 G의 바로 오른쪽 H나, 아니면 G의 바로 왼쪽 F가 들어가야 함.
  - G 바로 다음에 나오는 H를 G의 중위 후속자(In-order Successor, 後續者)라 하고, G 바로 직전에 나오는 F를 G의 중위 선행자(In-order Predecessor, 先行者)라고 함.

이진 탐색트리의 삭제

```
void BSTclass::Delete(Nptr& T, int Key)
{ if (T = = NULL)              삭제할 노드 못 찾고 리프노드의 널까지 오면
    printf("No Record with Such Key"); 오류처리
  else if (T->Data.Key > Key)      삭제 키가 현재 노드의 키보다 작으면
    Delete(T->LChild, Key);        왼쪽 서브트리로 가서 삭제
  else if (T->Data.Key < Key)      삭제 키가 현태 노드의 키보다 크면
    Delete(T->RChild, Key);        오른쪽 서브트리로 가서 삭제
  else if (T->Data.Key = = Key)    현재 T가 가리키는 노드가 삭제할 노드
  {  if ((T->LChild = = NULL) && (T->RChild = = NULL)) 자식이  없는 경우
    {   Nptr Temp = T; T = NULL; Delete Temp;          T에 널 값을 할당
    }
    else if (T->LChild = = NULL)   오른쪽 자식만 있는 경우
    {   Nptr Temp = T; T = T->RChild; Delete Temp;  부모를 오른쪽 자식에
      }
    else if (T->RChild = = NULL)   왼쪽 자식만 있는 경우
    {   Nptr Temp = T; T = T->LChild; Delete Temp;  부모를 왼쪽 자식에 연결
    }
    else                                          자식이 둘인 경우
      SuccessorCopy(T->RChild, T->Data);     오른쪽 자식노드 포인터와
  }                                          현재 노드의 레코드를 넘김
}
```

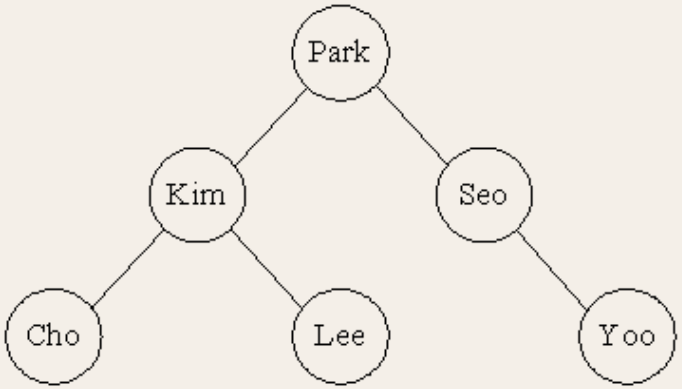
이진 탐색트리의 삭제

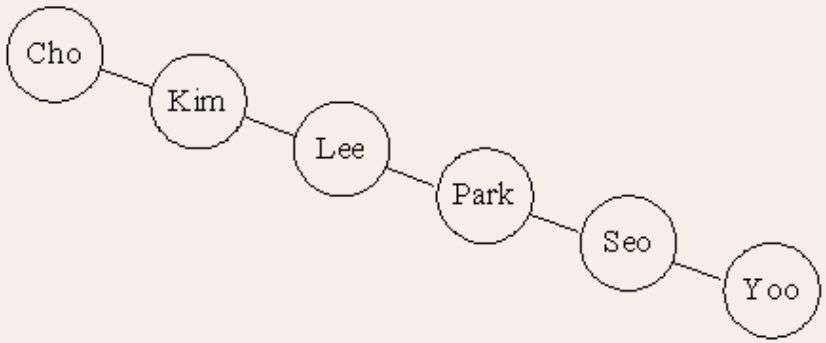
```
void BSTclass::SuccessorCopy(Nptr& T, dataType& DelNodeData)
{  if (T->LChild = = NULL)      중위 후속자이라면
  {  DelNodeData.Key = T->Key; 후속자 레코드를 넘어온 레코드
 에 복사
    Nptr Temp = T;
    T = T->RChild;                  후속자의 부모와 자식을 연결
    delete Temp;
  }
   else
   SuccessorCopy(T->LChild, DelNodeData); 후속자 찾아서 왼
 쪽으로 이동
}
```

- **후속자는 자식이 없거나 자식이 하나임**
  - 후속자의 Lchild는 항상 널이기 때문
  - 만약 널이 아니면 후속자가 아님.

이진 탐색트리의 효율

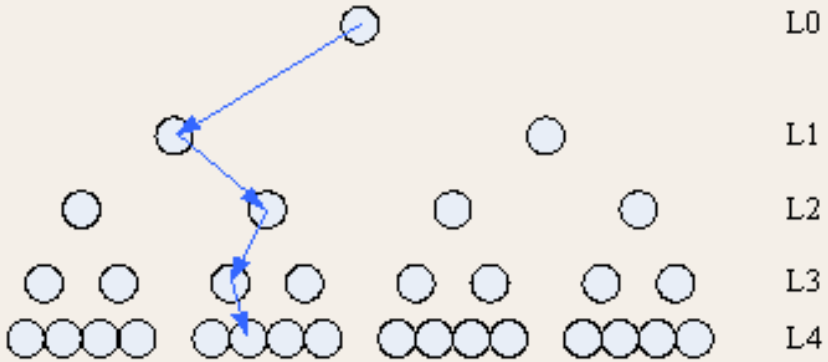
- 키 Lee인 노드를 찾기
  - Park, Kim, Lee(왼쪽 트리) 대 Cho, Kim, Lee(오른쪽 트리)
- 키 Yoo인 노드 찾기
  - Park, Seo, Yoo: (왼쪽 트리) 대 Cho, Kim, Lee, Park, Seo, Yoo (오른쪽 트리)
- 균형
  - 왼쪽 트리는 완전한 균형. 오른쪽 트리는 왼쪽 서브트리의 높이가 -1(빈 트리), 오른쪽 서브트리의 높이가 4로서 균형이 무너짐.
- 최악의 경우 탐색은 리프노드까지
  - 왼쪽 트리는 최악의 경우에도 높이가 2. 오른쪽 트리는 최악의 경우 높이 5를 모두 타고 내려와야 함.
  - 오른쪽 트리는 연결 리스트(Linked List)에 가까움. 연결이 한쪽으로 일직선으로(Linear Structure) 진행하는 트리= 편향 이진트리





이진 탐색트리의 효율

- 균형이 잘 잡혀있는 경우
  - 높이는 lgN에 가까움. 높이만큼 키 비교를 요함.
  - 효율 O(lgN)



- 균형이 무너진 경우
  - 최악의 경우 연결 리스트에 가까움
  - 효율 O(N)

자료구조별 탐색효율 비교

❑ 자료구조별 탐색효율 비교

	최악의 경우			평균적 경우		
	탐색	삽입	삭제	탐색	삽입	삭제
정렬된 배열	lgN	N	N	lgN	N/2	N/2
정렬 안 된 연결 리스트	N	1	1	N/2	1	1
이진 탐색트리	N	N	N	lgN	lgN	$N^{1/2}$