1. 리스트

1. 배열을 이용한 리스트

구현체

```c
#define MAX_LIST_SIZE 100 // 리스트의 최대크기

typedef int element; // 항목의 정의

typedef struct {
    element array[MAX_LIST_SIZE]; // 배열 정의
    int size; // 현재 리스트에 저장된 항목들의 개수
} ArrayListType;
```

기능부

```c
// 리스트 초기화 함수
void init(ArrayListType *L)
{
    L->size = 0;
}
```

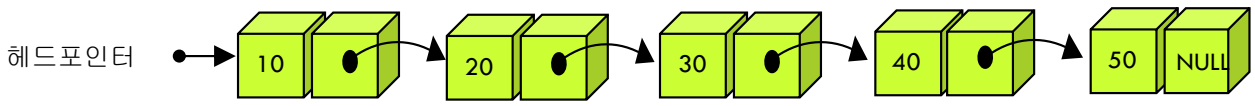
```c
// 리스트가 비어 있으면 1을 반환
// 그렇지 않으면 0을 반환
int is_empty(ArrayListType *L)
{
    return L->size == 0;
}
```

size에 현재 배열 사이즈를 담고,
배열을 기준으로 MAX_LIST_SIZE와 비교 후
맨 마지막 arr에 계속 추가.,..

2. 연결 리스트

1. 단순 연결 리스트

- 하나의 링크 필드를 이용하여 연결
- 마지막 노드의 링크 값은 NULL



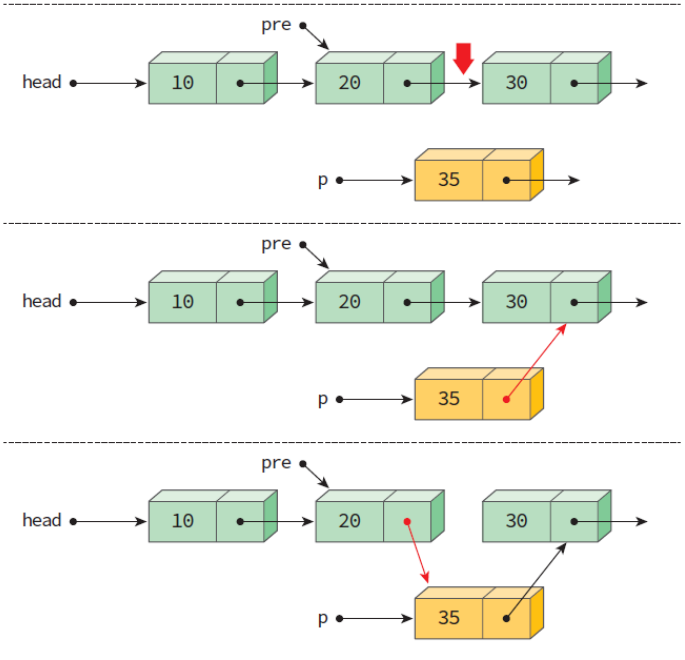
구현체

```
typedef int element;

typedef struct ListNode { // 노드 타입을 구조체로 정의한다.
    element data;
    struct ListNode *link;
} ListNode;
```

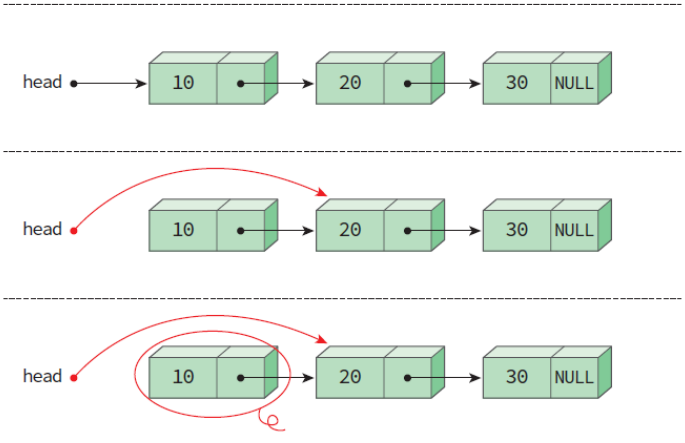
리스트의 생성

```
ListNode *head = NULL;

head = (ListNode *)malloc(sizeof(ListNode));

head->data = 10;
head->link = NULL;
```

노드 삽입



노드 삭제



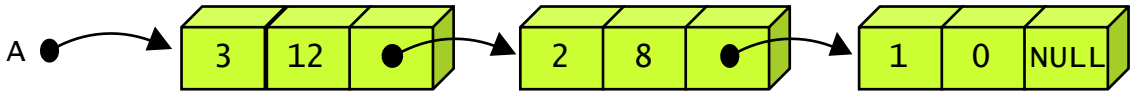
1. 응용

1. 다항식

구현체

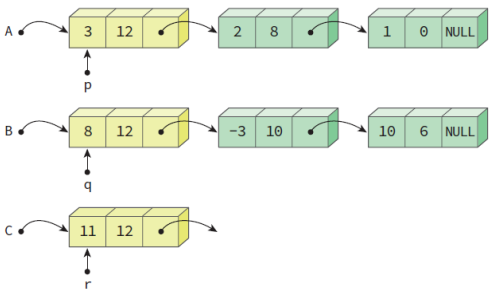
```
typedef struct ListNode {     // 노드 타입
    int coef;
    int expon;
    struct ListNode *link;
} ListNode;
```

$A = 3x^{12} + 2x^8 + 1$은 다음과 같이 나타낸다

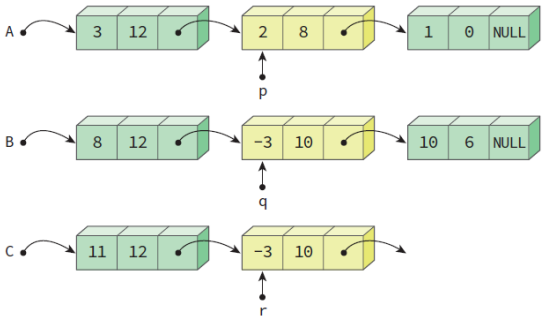


다항식의 덧셈

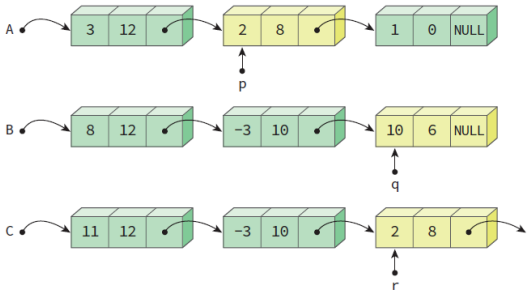
$A = 3x^{12} + 2x^8 + 1,\ B = 8x^{12} - 3x^{10} + 10x^6$ 일때,



(a) p와 q가 가리키는 항들의 지수가 같으면 계수를 더한다.



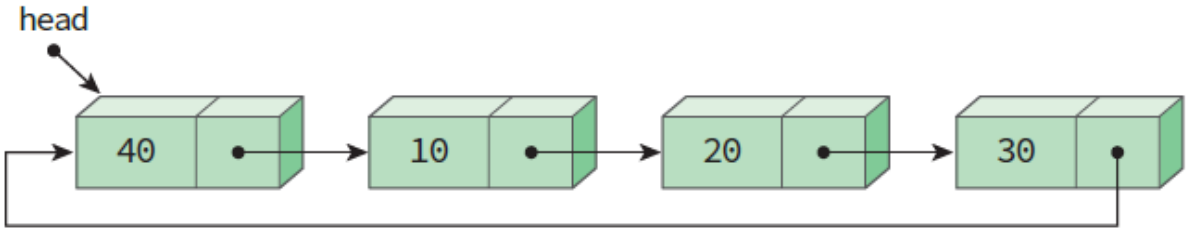
(b) q가 가리키는 항의 지수가 높으면 그대로 C로 옮긴다.



(c) p가 가리키는 항의 지수가 높으면 그대로 C로 옮긴다.

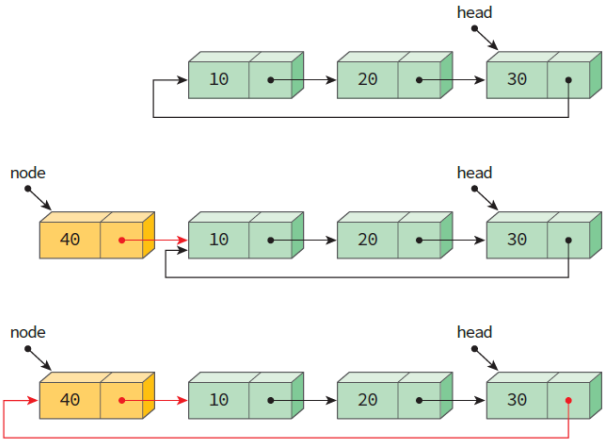
2. 원형 연결 리스트

- 마지막 노드의 링크가 첫 번째 노드를 가르키는 리스트
- 한 노드에서 다른 모든 노드로의 접근이 가능 (원형이라 계속 이어짐)



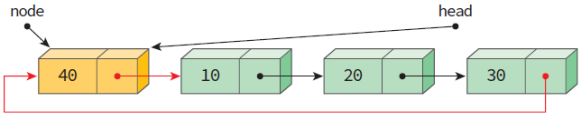
- Head를 마지막 노드로 두면, First_Insert나 Last_Insert에 용이
  (마지막에서 한칸 더 가면 첫번째 노드니깐)
  - 이 경우에서 First_Insert 사진

```
ListNode* insert_first(
       ListNode* head,element data){

       ListNode *node = (ListNode
*)malloc(sizeof(ListNode));
       node->data = data;
       if (head == NULL) {
              head = node;
              node->link = head;
       }
       else {
              node->link = head->link;   // (1)
              head->link = node;         // (2)
       }
       return head;  // 변경된 헤드 포인터를 반환한다.
}
```



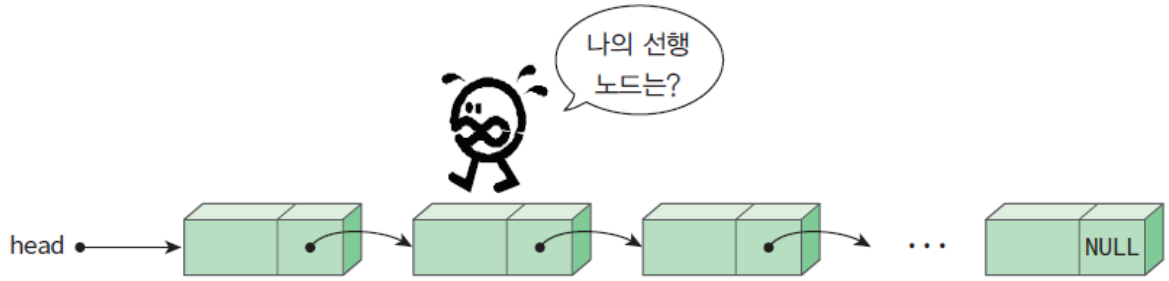
- 리스트 끝에 삽입은 새로 만들고 Head를 바꾸면 된다.

```
ListNode* insert_last
     (ListNode* head, element data){

       ListNode *node =
          (ListNode *)malloc(sizeof(ListNode));
       node->data = data;
       if (head == NULL) {
              head = node;
              node->link = head;
       }
       else {
              node->link = head->link;// (1)
              head->link = node;     // (2)
              head = node;           // (3)
       }
       return head;    // 변경된 헤드 포인터를 반환한다.
}
```



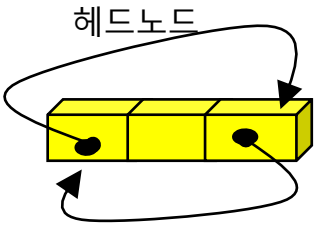
3. 이중 연결 리스트

- 단순 연결리스트의 문제점 (선행 노드를 찾기 힘들다)를 해소한 리스트



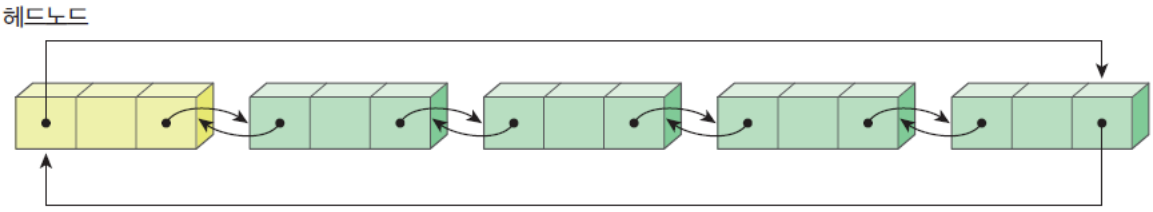
[사진] 단순 연결리스트의 문제점 (하나의 노드가 앞 노드의 정보만 가지고 있음)

- 이중 연결 리스트 : 하나의 노드가 앞/뒤 노드의 정보를 가지고 있는 연결 리스트
  단점 : 공간을 많이 차지하고 코드가 복잡



- 이중 연결리스트에서 사용 (단순에서는 헤더에 값 담김. 사용 X)
- 데이터를 가지지 않는다. (NULL)
- 단지 삽입 삭제코드를 간단하게 만들기 위해 존재 한다.
- 공백 일때는 헤드 노드만 존재(left, right = 본인 스스로를 가리킴)

헤드 노드에 데이터를 담게 되면, 해당 헤드노드를 삭제할때 불편함.
(헤드노드를 변경하는 것이기 때문에)

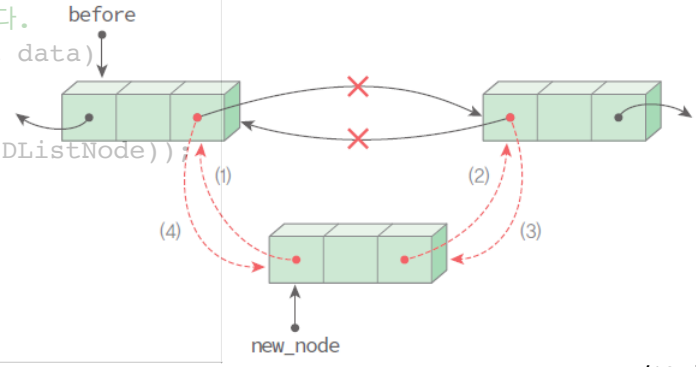


구현체

```
typedef int element;
typedef struct DlistNode {
    element data;
    struct DlistNode *llink; //왼쪽노드
    struct DlistNode *rlink; //오른쪽노드
} DlistNode;
```

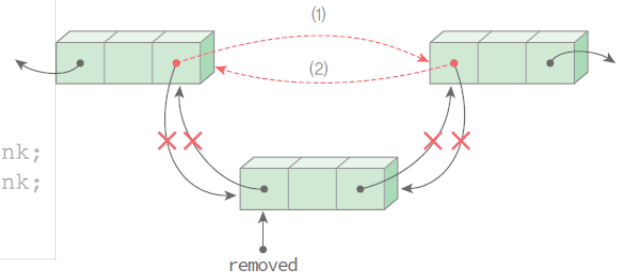
삽입연산

```
// 새로운 데이터를 노드 before의 오른쪽에 삽입한다.
void dinsert(DListNode *before, element data)
{
    DListNode *newnode =
        (DListNode *)malloc(sizeof(DListNode));
    strcpy(newnode->data, data);
    newnode->llink = before;
    newnode->rlink = before->rlink;
    before->rlink->llink = newnode;
    before->rlink = newnode;
}
```



삭제 연산

```
1 // 노드 removed를 삭제한다.
2 void ddelete(DListNode* head,
3       DListNode* removed){
4
5        if (removed == head) return;
6        removed->llink->rlink = removed->rlink;
7        removed->rlink->llink = removed->llink;
8        free(removed);
9 }
```

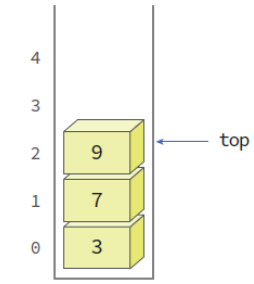


이중 연결 리스트 테스트 코드 및 결과

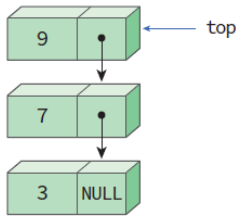
```
// 이중 연결 리스트 테스트 프로그램
int main(void)
{
     DListNode* head = (DListNode *)malloc(sizeof(DListNode));
     init(head);
     printf("추가 단계\n");
     for (int i = 0; i < 5; i++) {
          // 헤드 노드의 오른쪽에 삽입
          dinsert(head, i);
          print_dlist(head);
     }
     printf("\n삭제 단계\n");
     for (int i = 0; i < 5; i++) {
          print_dlist(head);
          ddelete(head, head->rlink);
     }
     free(head);
     return 0;
}
```

```
추가 단계
<-| |0| |->
<-| |1| |-> <-| |0| |->
<-| |2| |-> <-| |1| |-> <-| |0| |->
<-| |3| |-> <-| |2| |-> <-| |1| |-> <-| |0| |->
<-| |4| |-> <-| |3| |-> <-| |2| |-> <-| |1| |-> <-| |0| |->

삭제 단계
<-| |4| |-> <-| |3| |-> <-| |2| |-> <-| |1| |-> <-| |0| |->
<-| |3| |-> <-| |2| |-> <-| |1| |-> <-| |0| |->
<-| |2| |-> <-| |1| |-> <-| |0| |->
<-| |1| |-> <-| |0| |->
<-| |0| |->
```

연결 리스트를 이용한 스택



(a) 배열을 이용한 스택



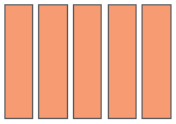
(b) 연결 리스트를 이용한 스택

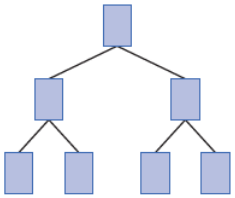
연결 리스트를 이용해서 스택도 구현할 수 있다.

C언어 상에서
배열을 이용한 스택은 MAX_SIZE가 정해져 있지만
연결 리스트를 이용한 스택은 무한정이다.

그치만 연결 리스트를 이용한 스택은
하나의 index에 데이터 + link pointer 까지 있어서
메모리가 좀 더 마니 쓰긴 함.

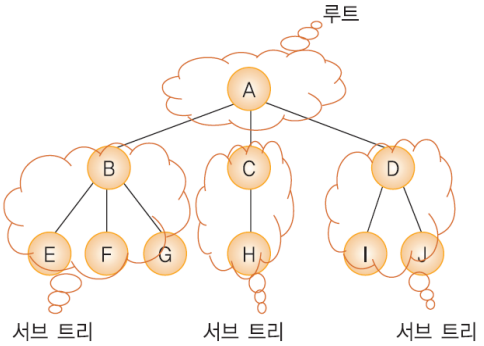
2. 트리


선형 자료구조

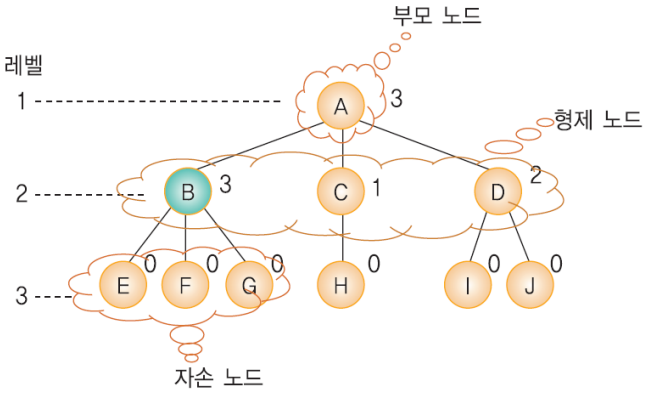

비선형 자료구조

리스트, 스택 큐 등은 선형구조

트리는 계층 적인 구조를
나타내는 **비선형 자료구조**

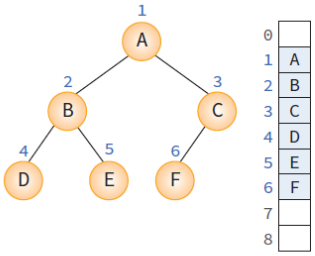


■노드(node)
트리의 구성요소
■루트(root)
부모가 없는 노드(A)
■서브트리(subtree)
하나의 노드와 그 노드들의 자손들로 이루어진 트리

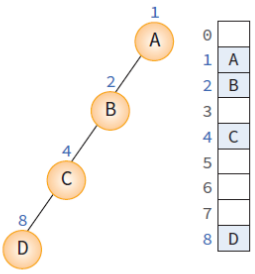


n 자식, 부모, 형제, 조상, 자손 노드
인간과 동일
n레벨(level)
트리의 각층의 번호 (**부모노드 1부터 시작**)
n높이(height)
트리의 최대 레벨(3)
n차수(degree)
노드가 가지고 있는 자식 노드의 개수

1. 이진트리의 표현법
   1. 배열을 이용한 방법



(a) 완전 이진트리

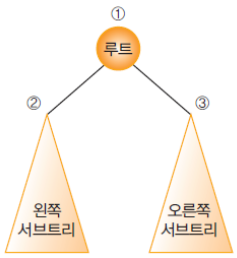


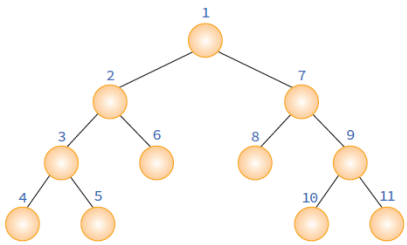
(b) 경사 이진트리

모든 이진 트리를 포화 이진 트리라고 가정하고 각 노드에 번호를 붙여서 배열에 저장함.

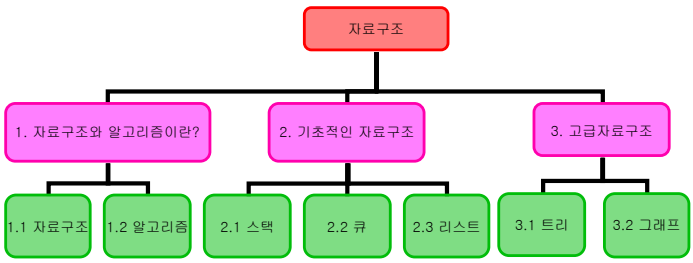
   2. 포인터를 이용한 방법

2. 이진 트리의 순회 방법
   1. 전위 순회 : 루트노드 -> 자손노드

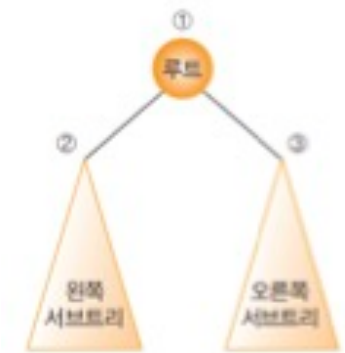


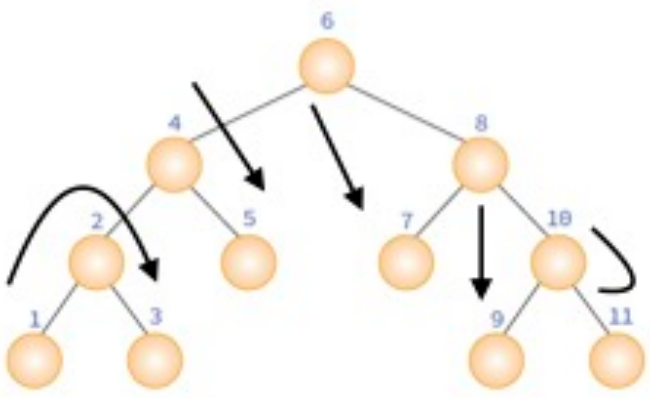


전위 순회의 응용 예 : 구조화된 문서 출력

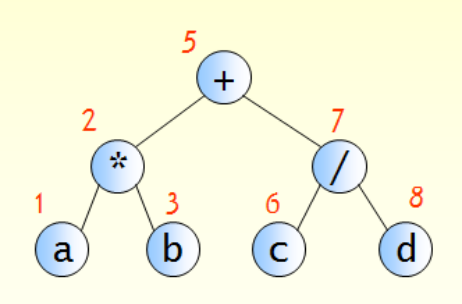


   2. 중위 순회 : 왼쪽 자손 -> 루트 -> 오른쪽 자손

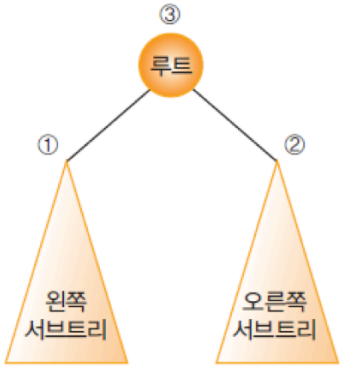


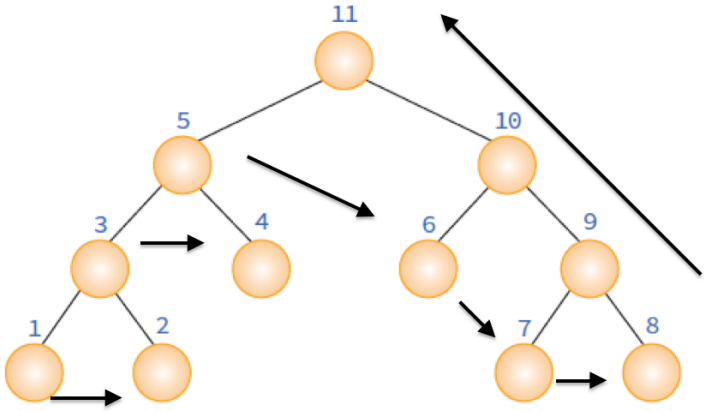


중위 순회 응용 예 : 수식 트리

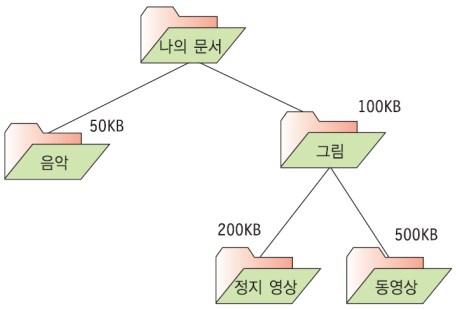


3. 후위 순회 : 자손 -> 루트





후위 순회 응용 예 : 디렉토리 용량 계산



구현체

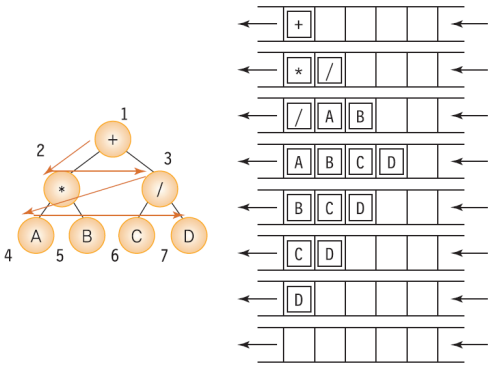
```
typedef struct TreeNode {
      int data;
      struct TreeNode *left, *right;
} TreeNode;
//           15
//       4        20
//    1         16 25
  TreeNode n1={1, NULL, NULL};
  TreeNode n2={4, &n1, NULL};
  TreeNode n3={16, NULL, NULL};
  TreeNode n4={25, NULL, NULL};
  TreeNode n5={20, &n3, &n4};
  TreeNode n6={15, &n2, &n5};
  TreeNode *root= &n6;
}
```

순회 기능 구현

```
// 중위 순회
inorder( TreeNode *root ){
  if ( root ){
    inorder( root->left );     // 왼쪽서브트리 순회
    printf("%d", root->data );       // 노드 방문
    inorder( root->right );    // 오른쪽서브트리 순회
  }
}

// 전위 순회
preorder( TreeNode *root ){
  if ( root ){
    printf("%d", root->data );       // 노드 방문
    preorder( root->left );    // 왼쪽서브트리 순회
    preorder( root->right );   // 오른쪽서브트리 순회
  }
}

// 후위 순회
postorder( TreeNode *root ){
  if ( root ){
    postorder( root->left );   // 왼쪽 서브 트리 순회
    postorder( root->right ); // 오른쪽 서브 트리 순회
    printf("%d", root->data );       // 노드 방문
  }
}
```

```
중위 순회=[1] [4] [15] [16] [20] [25]
전위 순회=[15] [4] [1] [20] [16] [25]
후위 순회=[1] [4] [16] [25] [20] [15]
```

4. 레벨 순회

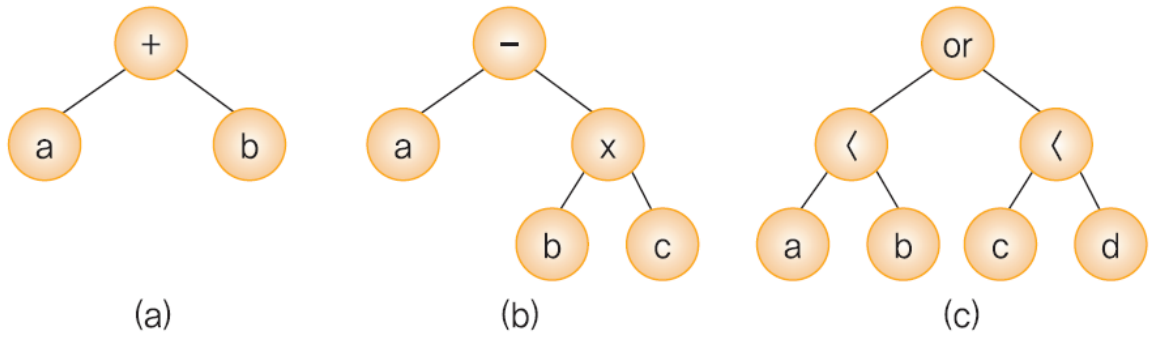


레벨 순회는 각 노드를 레벨 순으로 검사함.

기존 순회법은 스택을 사용 했지만,
레벨 순회는 큐를 사용하는 방법이다.

3. 수식트리

- 후위 순회를 사용



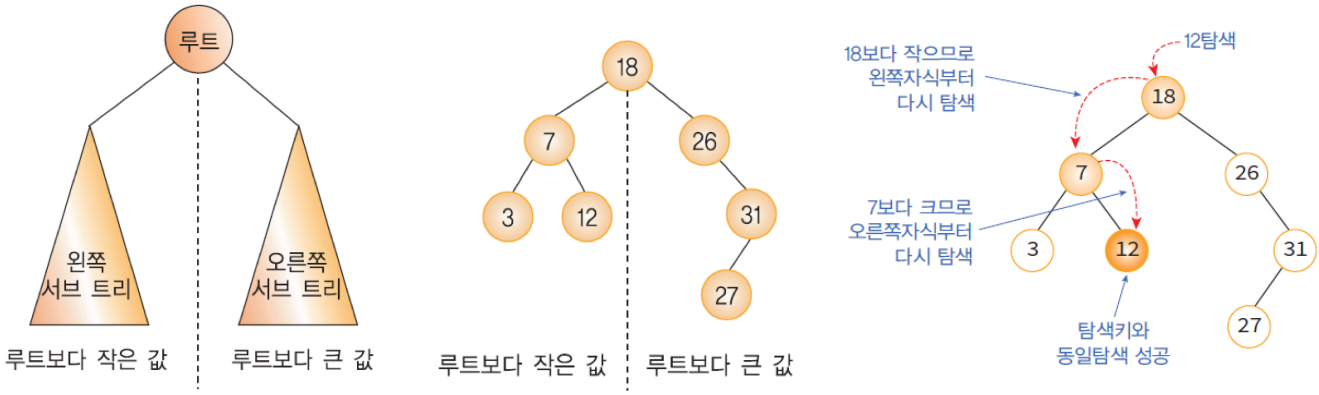
(a) (b) (c)

수식	a + b	a − (b × c)	(a ⟨ b) or (c ⟨ d)
전위순회	+ a b	− a × b c	or < a b < c d
중위순회	a + b	a − (b × c)	(a < b) or (c < d)
후위순회	a b +	a b c × −	a b < c d < or

4. 이진 탐색 트리

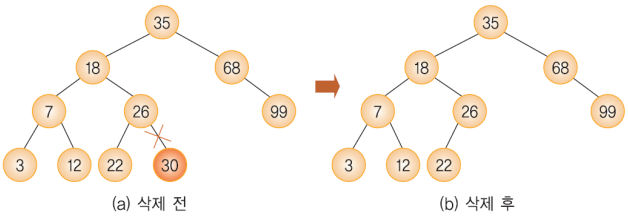
1. 탐색 연산

- 탐색을 효율 적으로 하기 위한 자료구조
- **key(왼쪽서브트리) ≤ key(루트노드) ≤ key(오른쪽서브트리)**
- 이진 탐색을 중위순회 하면, 오름차순으로 정렬된 값을 얻을 수 있음.



1. 삭제연산

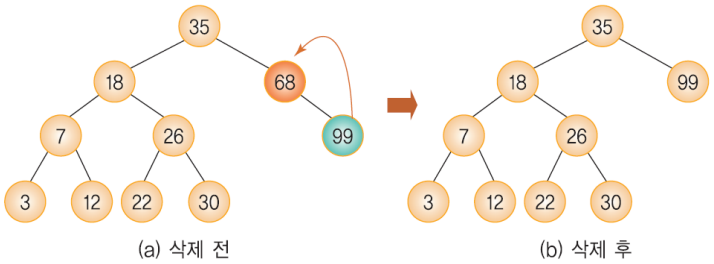
경우 1 : 삭제하려는 노드가 단말 노드일 경우



(a) 삭제 전 (b) 삭제 후

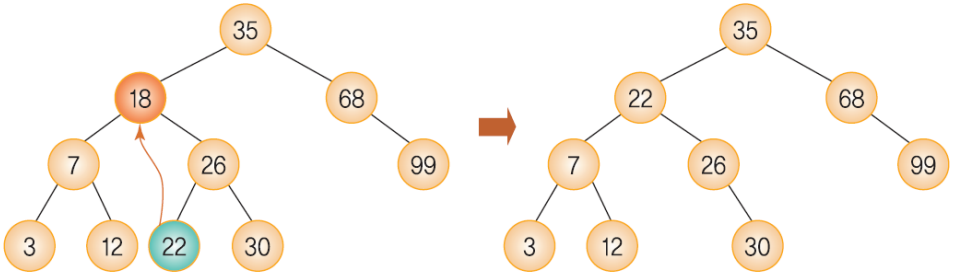
-> 단말노드의 부모노드를 찾아서 연결을 끊으면 된다.

경우 2 : 삭제하려는 노드가 하나의 서브 트리만 갖고 있는 경우



-> 그 노드는 삭제하고 서브 트리는 부모 노드에 붙여준다.

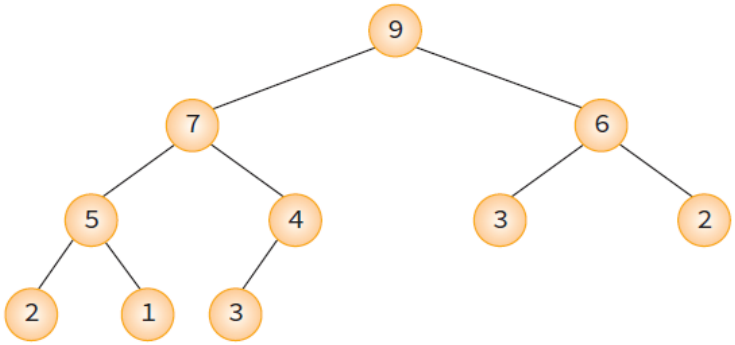
경우 3 : 삭제하려는 노드가 두개의 서브 트리 모두 가지고 있는 경우



-> 삭제노드와 가장 비슷한 값을 가진 노드를
삭제노드 위치로 가져온다. (자식 중 비슷한 값 끌어옴)
-> 가장 비슷한 값은 왼쪽 서브에서 큰 값 or 오른쪽 서브에서 작은 값

3. 우선순위 큐
우선순위를 가진 항목들을 저장하는 큐

1. 배열을 이용한 우선순위 큐
2. 연결 리스트를 이용한 우선순위 큐
3. 히프(heap)를 이용한 우선순위 큐
: key(부모노드) ≥ key(자식노드)을 만족하는 완전 이진 트리

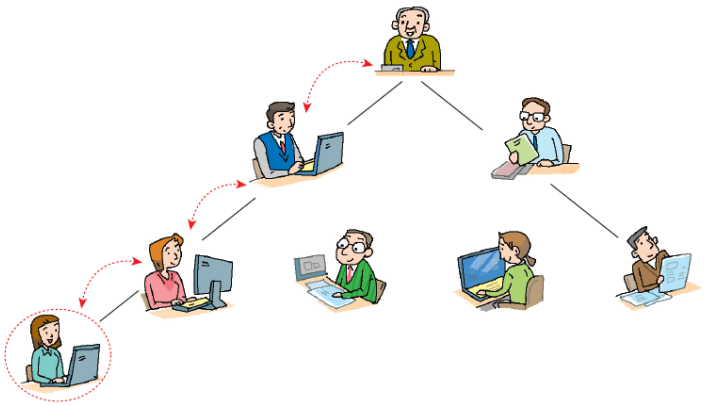


히프에서의 삽입

- 히프에 있어서 삽입 연산은 회사에서 신입 사원이 들어오면 일단 말단 위치에 앉힌 다음에, 신입 사원의 능력을 봐서 위로 승진시키는 것과 비슷

(1) 히프에 새로운 요소가 들어 오면, 일단 새로운 노드를 히프의 마지막 노드에 이어서 삽입

(2) 삽입 후에 새로운 노드를 부모 노드들과 교환해서 히프의 성질을 만족



히프에서의 삭제

- 최대 히프에서의 삭제는 가장 큰 키값을 가진 노드를 삭제하는 것을 의미

-> 따라서 루트 노드가 삭제된다.

- 삭제 연산은 회사에서 사장의 자리가 비게 되면 먼저 제일 말단 사원을 사장 자리로 올린 다음에, 능력에 따라 강등시키는 것과 비슷하다.

(1) 루트 노드를 삭제한다
(2) 마지막 노드를 루트 노드로 이동한다.
(1) 루트에서부터 단말 노드까지의 경로에 있는 노드들을 교환하여 히프 성질을 만족시킨다.