제 13 강 . 검색(Search)

학습 목차

1. 선형검색
2. 이진검색
3. 해시탐색
4. 이진탐색트리(BST)
5. AVL tree
6. B-tree

학습 가이드

정렬과 함께 문자 자료처리의 중요한 부분인 탐색(검색) 알고리즘에 대하여 살펴본다.

탐색은 주어진 데이터에서 키 값에 해당하는 자료를 찾는 것을 말한다. 데이터의 개수가 작을 때는 수행시간이 문제가 되지 않지만 데이터 개수가 많을 때는 효율적인 탐색 프로그램이 필요하다.

기초적인 탐색 알고리즘인 선형탐색과 이진탐색은 선형 데이터를 비교를 통하여 데이터를 찾는 알고리즘이다. 이진탐색은 선형탐색보다 효율적이지만 데이터가 정렬된 리스트에 대하여 사용할 수 있다.

비교를 통하지 않고 데이터를 찾는 방법으로는 해싱이 있다 해싱은 해시함수를 이용하여 탐색을 하며 비교를 통한 탐색보다 훨씬 효율적이다.

트리를 이용한 탐색 방법으로는 이진 탐색 트리, AVL 트리, B-트리 등이 있다. 트리 탐색은 데이터를 찾는 시간이 빠를 뿐 아니라 데이터를 삽입, 삭제하는 시간도 효율적이다. 또 B-트리 같은 경우는 외부기억장치에 저장된 대용량의 데이터에 대하여 사용할 수 있는 알고리즘이다.

1. 선형검색(linear search)

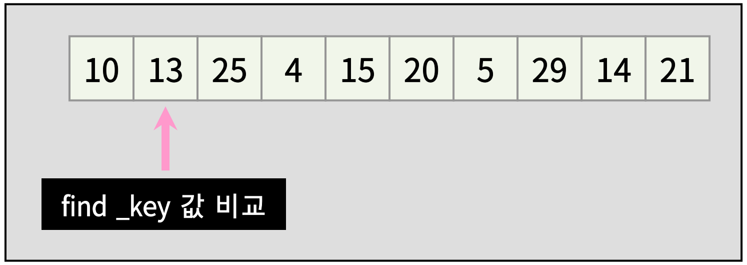
(선형검색이란?)

선형검색(혹은 선형탐색, 순차탐색)은 주어진 데이터에서 키 값에 의하여 데이터를 찾는 과정이다. 검색 방법 중 가장 간단하다.

(선형검색 알고리즘)

```
/* 선형검색 알고리즘 – 배열 keys[]에서 find_key와 일치하는 것을
찾는다. */
int sequential_search(int keys[], int find_key, int n)
{
            int i =0;
            while(i <= n)
            {    i++
                 if(keys[i] == find_key return(i);
            }
            return(0);
}
```

(선형검색 알고리즘의 진행)



10	13	25	4	15	20	5	29	14	21

find _key 값 비교

(선형검색 알고리즘의 분석)

프로그램은 key 값을 데이터와 비교를 한다. 운이 좋으면 1번에 찾지만, 최악의 경우는 맨 마지막에 데이터가 있거나 찾는 데이터가 없는 경우이며 n개의 데이터를 모두 비교해야 한다. 이것을 평균을 내면 n / 2 번 비교를 하게 된다. 알고리즘의 효율도로 표시하면 O(n)이 된다.

대개의 경우 데이터가 1000-10000개 일 경우 비교적 시간이 작게 걸리지만 1백만개 혹은 1억 개의 데이터가 있다고 가정하면 선형 검색은 사용하기 어려운 방법이 된다.

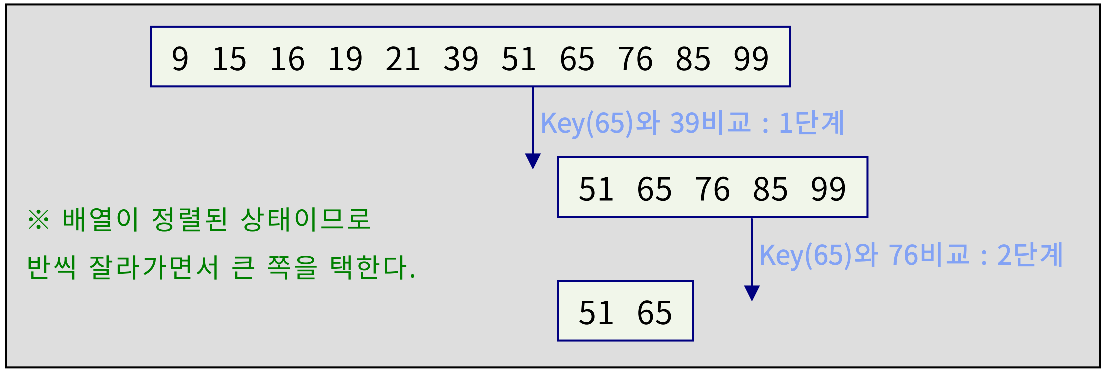
다음 절부터 나오는 검색 알고리즘들은 모두 O(n)보다 더 효율적인 방법으로 O(logn)정도의 시간이 걸린다.

2. 이진검색(Binary Search)

(이진검색이란?)

이진검색은 정렬된 데이터에서 사용하는 방법이다. 1000개의 숫자 자료를 가진 리스트가 있다고 할 때(오름차순으로, 1부터 10000까지), 여러분이 만약 “5001”을 찾으려고 한다면 대략 리스트의 가운데 데이터를 집어서 비교하는 것이 첫 데이터부터 하나씩 비교하는 것 보다 더 빠르다. 이진 검색은 이렇게 정렬된 데이터에서 찾으려는 값과 리스트의 가운데와 비교하여 찾는 방법이다. 한번 비교해서 찾는 데이터가 아니더라도 정렬되어 있기 때문에 찾는 값이 가운데 값보다 크면 오른쪽 반, 작으면 왼쪽 반을 비교해 나가면 된다. 즉 이런 방법으로 매번 비교할 때 마다 비교 대상 리스트를 ½로 줄여 나갈 수 있다.

- 검색의 예 - 65 찾기



9 15 16 19 21 39 51 65 76 85 99

Key(65)와 39비교 : 1단계

51 65 76 85 99

Key(65)와 76비교 : 2단계

51 65

※ 배열이 정렬된 상태이므로 반씩 잘라가면서 큰 쪽을 택한다.

이진 탐색 알고리즘

```
/* 이진 탐색 알고리즘 – list[]에서 searchnum을 찾는다 */
/*searchnum 에 대해 list [0]<=list[1]<= ...<=list[n-1]을 탐색.
찾으면 그 위치를 반환하고 못 찾으면 –1을 반환한다.*/

int binsearch(int list[],int searchnum, int left,int right)
{
        int middle;
        while(left <= right) {
              middle = (left + right) / 2; /* middle 값 계산 */
              switch(COMPARE(list[middle],searchnum)) {
                   case -1: left = middle + 1;  break;
                case  0: return middle;
                case  1: right = middle - 1; break;
              }
         }
         return -1;
}
```

2. 이진검색(Binary Search)

(이진검색 알고리즘의 분석)

While 문을 반복하는 횟수가 이진탐색의 시간이 된다. While 문 안에서 프로그램은 key 값을 리스트의 데이터와 비교를 한다. 찾으면 1번에 끝나지만, 못 찾는 경우 리스트의 반을 대상으로 다시 비교를 시작한다. 마지막까지 진행하여 데이터가 1개인 리스트까지 비교를 한다고 하면 비교 대상 데이터 개수는 다음과 같이 변한다.

n -> n/2 -> n/4 -> n/8 -> ...

$n = 2^k$ 라고 가정하면 데이터의 개수는

2^k -> 2^{k-1} -> 2^{k-2} -> 2^{k-3} ..., 2^0 으로
매 반복마다 비교 대상 데이터 수 ½로 변한다.

데이터 개수가 1개 될 때까지 반복을 계속하게 되므로, k번 반복을 하게 된다.

여기서 k를 계산하면

$n = 2^k$ 에서 양변에 베이스가 2인 log를 씌우면
log n = k 이므로 log n번 반복하게 된다.

그러므로 프로그램의 복잡도는 O(log n) 이 된다.

3. 해시탐색(Hash Search)

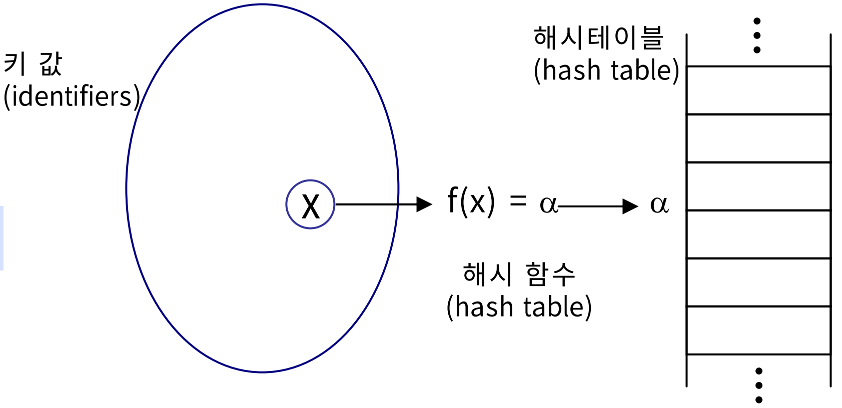
(해싱이란?)

해싱은 찾으려는 키 값를 조작하여 데이터를 바로 찾는 것을 말한다. 쉬운 예를 들면 찾으려는 키 값을 제곱하여 끝 2자리의 위치에 데이터가 있다고 하면 데이터가 123이라고 할 때 123*123=15129 이며 데이터는 끝 두자리 값인 29에 저장되어 있다는 것을 알 수 있다. 즉 키 값이 123이면 데이터가 X[29]에 저장되어 있다는 것을 알 수 있다.

앞의 선형검색이 평균 n/2번 비교하고 이진검색이 log n 비교하는 것과 비교하면 비교도 없이 바로 찾을 수 있어서 검색 중에서는 가장 빠르다고 할 수 있다.

데이터가 충돌 될 수도 있다. 또 방법을 잘못 택하면 충돌이 심해진다는 것을 알 수 있다. 키 값이 23, 123, 223, 323, 423, 523 이면 모두 같은 위치에 저장된다는 것을 알 수 있다.

- 해싱과 해시 함수



3. 해시탐색(Hash Search)

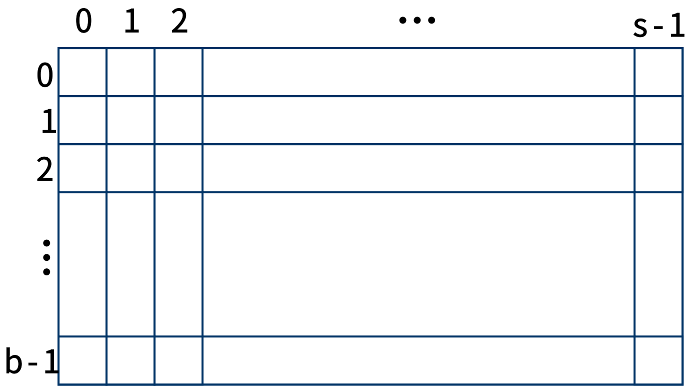
(정의)

- 해시 테이블(hash tables) - 키 값을 저장하는 테이블

b : 버킷(bucket) : 테이블의 크기, 해시될 키 값의 범위

s : 슬롯(slot : 한 개의 버킷에 저장될 키 값의 개수

전체 저장할 수 있는 데이터의 개수는 s*b개이다.



- 해시테이블의 식별자 밀도(identifier density) : n/T

n: 테이블에 식별자 수

T: 전체 가능한 식별자 수

예) 테이블에 저장된 식별자 수를 50개라 하고 전체 가능한 식별자 수를 100개라고 하면 식별자 밀도는 50/100 = 0.5이다.

- 적재 밀도 : a = n/(s·b)

s: 슬롯(slot, socket)의 수

b: 버킷(bucket)의 수

예) 저장될 데이터가 n=90개이고 해시테이블의 버킷이 100개, 슬롯이 2개라면 적재밀도는 90/(100*2)= 0.45이다. 즉 데이터가 다 저장되면 기억장소 활용률은 0.45가 된다.

- 동의어(synonym) : 두 식별자 i_1 와 i_2 가 해시 함수 f에 대하여 계산을 했을 때 같은 값을 갖는 경우 동의어라 한다.

$$f(i_1) = f(i_2)$$

- 오버플로우(overflow)

full인 상태의 bucket에 식별자 i가 해시 되어 들어오는 경우 더 이상 데이터를 저장할 수 없는 상태를 말한다.

- 충돌(collision) - 서로 다른 식별자가 같은 bucket에 해시 되는 경우

* 충돌이 일어나면 한 버킷에 데이터를 저장할 슬롯이 충분하면 저장이 가능하고 충분치 않으면 오버플로우가 발생한다. 오버플로우가 일어나면 데이터를 다른 곳에 저장을 해야 한다. 충돌과 오버플로우는 버킷 크기가 1일 경우 항상 같이 일어난다.

3. 해시탐색(Hash Search)

(해싱의 예)

변수 이름을 해시테이블에 저장하는 예,
해시 테이블 이름 ht, 버킷의 수 b = 26, 슬롯 s = 2,
해시함수 f = (식별자의 영문 첫 글자)
식별자 = { acos, atan, char, ceil, exp, float, define, floor, … }

	slot()	slot 1
0	acos	atan
1		
2	char	ceil
3	define	
4	exp	
5	float	floor
6		
•••		
25		

(해시 함수)

1. 해시 함수는 식별자 x를 변환하여 해시테이블의 버킷 주소로 바꾼다.
2. 좋은 해시 함수의 조건
   - 계산이 편해야 한다.
   - 충돌을 최소화하여야 한다.
3. uniform hash function : 충돌을 최소화하는 함수의 성질
   확률 함수 f(x)=i: 1/b 를 갖는 함수이다.
   즉 random 값 x가 버킷의 어느 위치에 들어갈 확률이 같은 경우이다.

3. 해시탐색(Hash Search)

(해싱 함수의 기법)

(1) mid-square

식별자의 제곱 값의 가운데 값을 취한다.

1) 식별자의 제곱을 계산한다.

2) 계산된 값의 가운데 숫자(비트)를 이용하여 버킷을 계산한다.

가운데 값을 이용하는 것이 uniform 분포를 가질 가능성이 많다.

(예를 들면 3으로 끝나는 수의 곱은 9가 되기 때문에 동의어가 될 가능 성이 높다. 따라서 제곱 값의 뒷부분을 이용하면 좋지 않은 방법이다.)

(2) division

나머지 연산자(modulus, %)를 이용한다.

$f_D(x)$ = x % M, M : 테이블 사이즈

- bucket 주소의 범위 : 0 ~ M-1
- M값의 선택이 중요하다. M을 소수(prime number)로 정한다.

(3) digit analysis

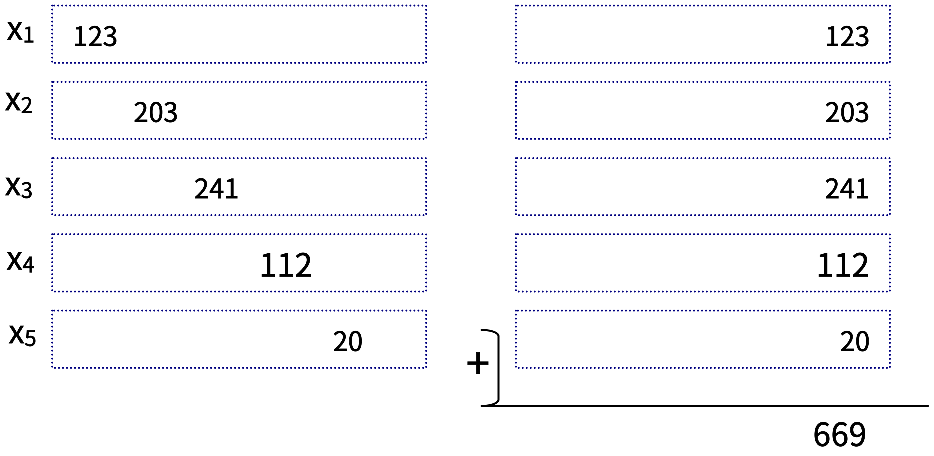
- 키 값의 자리 수를 분석하여 분포가 고른 자리 수를 해시 값으로 사용한다.

(4) folding - 접지법이라고 하며 키 값을 접어서 조작을 하는 방법이다. 접지하는 방법이 여러 가지 이다.

예) 식별자 x = 12320324111220

(4) folding

Case 1) 이동 접지법(shift folding)



x_1 123

x_2 203

x_3 241

x_4 112

x_5 20

123

203

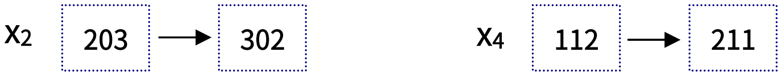
241

112

+ 20

669

Case 2) 경계 접지법



x_2 203 → 302

x_4 112 → 211

123 + 302 + 241 + 211 + 20 = 897

3. 해시탐색(Hash Search)

(오버플로우 문제해결)

해싱에서는 오버플로우는 반드시 일어난다. 오버플로우를 해결하는 방법은 다양하고 성능에 영향을 미친다.

1) 개방 주소법(open addressing) - 해시 영역 내에서 빈 공간을 찾아서 해결하는 방법이다.

선형조사법과 2차조사법이 있다.

① 선형조사법(linear probing)

- 해시테이블을 1차원 배열로 보고 충돌이 일어나면 다음 위치에 저장하는 방법이다.

```
/* 1차원 배열로 선언된 Hash Table */
#define MAX_CHAR 10
/* max number of characters in an identifier*/
#define TABLE_SIZE 13
/* max table size = prime number*/
typedef struct {
        char key[MAX_CHAR];
        /* other filed */
        } element;
element hash_table[TABLE_SIZE];
```

(해시함수 계산 프로그램 예)

```
/* 해시 함수 문자 변수 포인터인 key 값을
증가시키면서 number 변수에 문자 값을 계속
더한다. 문자 열의 끝에서 덧셈을 종료 한다.*/
int transform(char *key) {
        int number = 0;
        while (*key)
                number = number + *key++;
        return number;
}
/* 버킷 주소 생성 함수 */
int hash(char *key) {
        return(transform(key) % TABLE_SIZE);
}
```

(해싱의 예)

앞의 해시함수를 이용하여 아래의 식별자를 해싱하여 해시 값을 구한다음 해시테이블에 저장한다.

- 식별자 : “for, do, while, if, else, function”
- 해시테이블 : b = 13, s = 1

식별자	계산 과정	x	hash
for	102+111+114	327	2
do	110+111	211	3
while	119+104+105+108+101	537	4
if	105+102	207	12
else	101+108+115+101	425	9
function	102+117+110+99+116+105+111+110	870	12

[접지와 나눗셈을 이용한 해싱]

[0]	function
[1]	
[2]	for
[3]	do
[4]	while
[5]	
[6]	
[7]	
[8]	
[9]	else
[10]	
[11]	
[12]	if

해시테이블에 저장된 결과(오버플로우는 다음 슬롯에 저장)

(13 buckets, 1 slot/bucket)

3. 해시탐색(Hash Search)

```
/* 선형조사법을 이용한 해시 키의 삽입 */
void linear_insert(element item, element ht[])
{
        int i, hash_value;
        hash_value = hash(item.key);
        i = hash_value;
        while(strlen(ht[i].key)) {
                if(!strcmp(ht[i].key, item.key)) {
                        fprintf(stderr,”duplicate entry₩n”);
                        exit(1);
                }
                i = (i + 1) % TABLE_SIZE;
                if(i == hash_value) {
                        fprintf(stderr,”the table is full₩n”);
                        exit(1);
                }
        }
        ht[i] = item;
}
```

(선형조사법의 단점)

- 충돌이 한 버킷에 집중하는 경향이 있다.
- 검색시간을 증가시킨다.

예) C 언어의 내장함수(built-in function) 이름을 26 버킷과 1개의 슬롯을 가진 해시 테이블에 다음과 같이 삽입한다.

“acos, atoi, char, define, exp, ceil, cos, float, atol, floor, ctime”

➔ 단점 : atol 같은 키워드는 원래 저장될 위치보다 훨씬 아래쪽에 위치한다.

선형조사법에 의한 오버플로우 조작
(26 버킷, 1 슬롯/버킷)

“acos, atoi, char, define, exp, ceil, cos, float, atol, floor, ctime”

bucket	x	bucket searched
0	acos	1
1	atoi	2
2	char	1
3	define	1
4	exp	1
5	ceil	4
6	cos	5
7	float	3
8	atol	9
9	floor	5
10	ctime	9
…		
25		

② 2차조사법(quadratic probing)

해시테이블의 버킷을 조사하여 1의 제곱, 2의 제곱, 3의 제곱 순으로 새로운 위치를 찾는다. 평균 조사 시간을 줄일 수 있다.

- $ht[(f(x) + i^2) \% b]$ and $ht[(f(x) - i^2) \% b]$, where $0 \le i \le (b-1)/2$

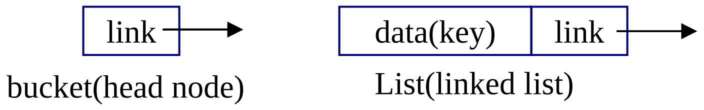
b: 테이블의 버킷의 수

3. 해시탐색(Hash Search)

2) 재해싱(rehashing)

오바플로우 상태에서 다른 해시 함수를 사용하는 것을 말한다.

- 새로운 해시 함수 f_1, f_2, … , f_b 를 계속해서 적용한다.
- 버킷 f_i (x) i = 1, 2, … , b 를 조사한다.



link

bucket(head node)

data(key) link

List(linked list)

3) chaning

- 선형조사법의 단점을 극복한다.
- 식별자 저장 버킷의 기억장소 공간을 늘려서 해결한다.
- 버킷에 연결리스트를 만든다. 연결리스트는 무한정 증가시킬 수 있기 때문에 오버플로우를 쉽게 해결할 수 있다.
- 각 연결리스트는 동의어를 저장한다.

해시테이블(chaining) 선언

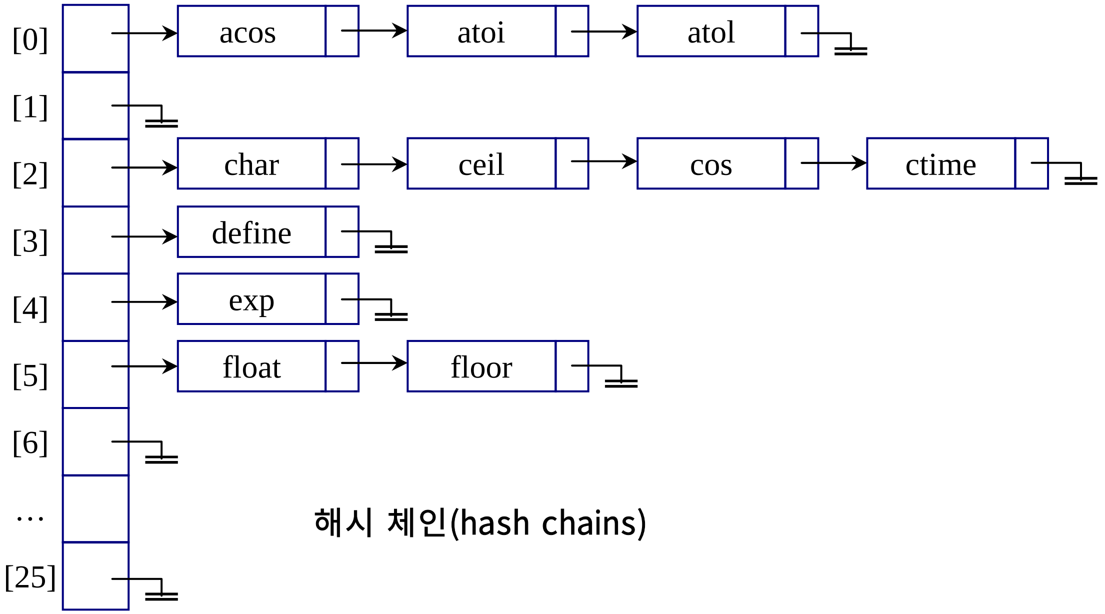
```
/* 해시테이블(chaining)을 위한 연결리스트의
선언 */
#define MAX_CHAR 10
#define TABLE_SIZE 13
#define IS_FULL(ptr) (!(ptr))
typedef struct {
        char key[MAX_CHAR];
        /* other fields */
} element;

typedef struct list *list_ptr;
typedef struct list {
        element item;
        list_ptr link;
}
list_ptr hash_table[TABLE_SIZE];
```

3. 해시탐색(Hash Search)

chaining 에서의 삽입 프로그램

```
/* chaining 에서의 삽입 프로그램 */
void chain_insert(element item,list_ptr ht[])
{
        int hash_value = hash(item.key);
        list_ptr ptr, trail = NULL;
        list_ptr lead = ht[hash_value];
        for(; lead; trail=lead, lead = lead->link)
                if(!strcmp(lead->item.key, item.key))
                {       fprintf(stderr,”the key is in the table₩n”);
                        exit(1);
                }
        }
        ptr = (list_ptr)malloc(sizeof(list));
        if(IS_FULL(ptr)) {
              fprintf(stderr,“the memory is full₩n”);
              exit(1);
        }
        ptr->item = item;
        ptr->link = NULL;
        if(trail) trail->link = ptr;
        else ht[hash_value] = ptr;
  }
```



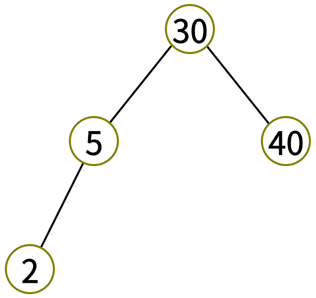
해시 체인(hash chains)

4. 이진 탐색 트리(BST, Binary Search Tree)

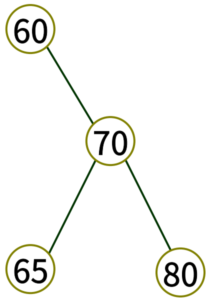
(이진 탐색 트리의 정의)

정의 : 이진 탐색 트리는 이진 트리이며 empty이거나 다음을 만족하는 트리이다.

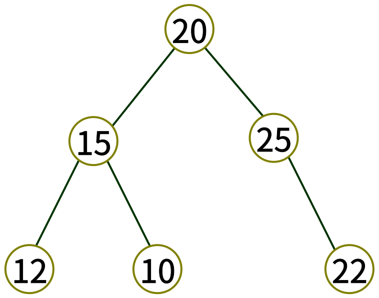
1) 모든 노드는 키 값을 갖고 있으며, 같은 키 값을 갖는 경우는 없다.
2) 왼쪽 부속 트리의 모든 노드의 키 값은 루트의 키 값보다 작다.
3) 오른쪽 부속 트리의 모든 노드의 키 값은 루트의 키 값보다 크다.
4) 왼쪽과 오른쪽 부속 트리도 마찬가지로 BST이다.



(a) BST



(b) BST



(c) BST가 아닌 경우

(이진탐색트리의 성질)

- 탐색(searching), 삽입(insertion), 삭제(deletion) 시간은 트리의 높이 만큼 시간이 걸린다.
O(h), h : BST의 깊이(height)

- 키 값에 의한 탐색은 루트 노드로부터 시작을 한다. 비교하여 키 값이 더 크면 오른쪽 트리로 이동하고 작으면 왼쪽 트리로 이동한다. 이 같은 방법으로 원하는 데이터를 찾아 내려간다.

- 트리를 중위 탐색(inorder traversal)하면 정렬된 리스트가 출력된다.

4. 이진 탐색 트리(BST, Binary Search Tree)

```
/* 이진탐색트리에서의 키값 찾는 알고리즘 – 반복 알고리즘 */
tree_ptr iter_search(tree_ptr tree, int key)
{
        while(tree) {
                if(key == tree->data) return tree;          /*찾음*/
                if(key < tree->data)
                        tree = tree->left_child;            /*왼쪽 트리에서 검색 */
                else
                        tree = tree->right_child;           /* 오른쪽 트리에서 검색 */
        }
        return NULL;
}
```

```
/* 이진 탐색 트리에서의 키 값 찾는 알고리즘 –순환 알고리즘 */
tree_ptr search(tree_ptr root, int key)
{
    if(!root) return NULL;                              /*비어 있는트리*/
        if(key == root->data) return root;              /*찾음*/
        if(key < root->data)
                return search(root->left_child, key);   /*왼쪽 트리*/
        return search(root->right_child, key);          /*오른쪽 트리*/
}
```

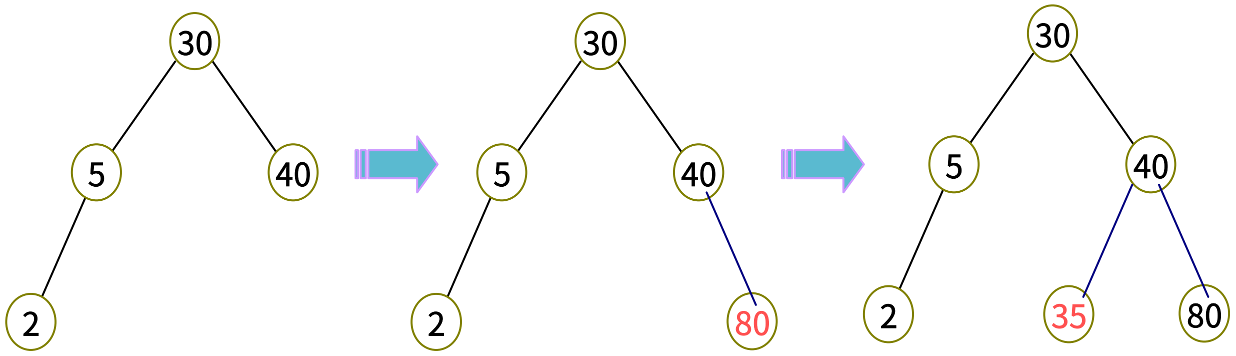
이진탐색트리에서의
키값 찾는 알고리즘

삽입 알고리즘

```
/* 삽입 알고리즘 */
void insert_node(tree_ptr *node, int num) {
    tree_ptr ptr, temp = modified_search(*node, num);
    /* modified search() 는 삽입될 노드의 부모를 찾아 temp에 저장 */
    if(temp || !(*node)) {
        ptr = (tree_ptr)malloc(sizeof(node));
        if(IS_FULL(ptr)) {
            fprintf(stderr,"The momory is full₩n");
            exit(1);
      }
      ptr->data = num;
      ptr->left_child = ptr->right_child = NULL;
      if(*node)
          if(num < temp->data) temp->left_child = ptr;
            else temp->right_child = ptr;
      else *node=ptr;
    }
}
```

4. 이진 탐색 트리(BST, Binary Search Tree)

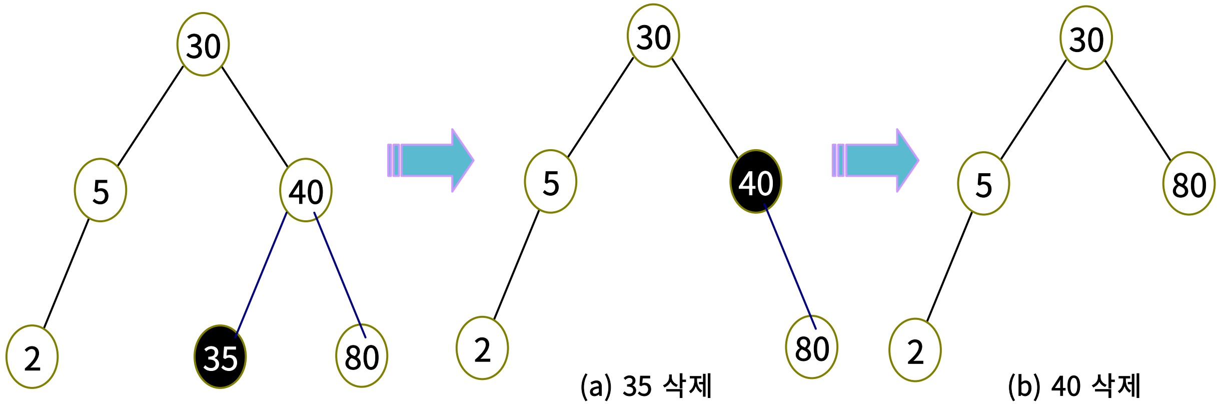
[BST 에서 삽입 예]



(a) 80 삽입

(b) 35 삽입

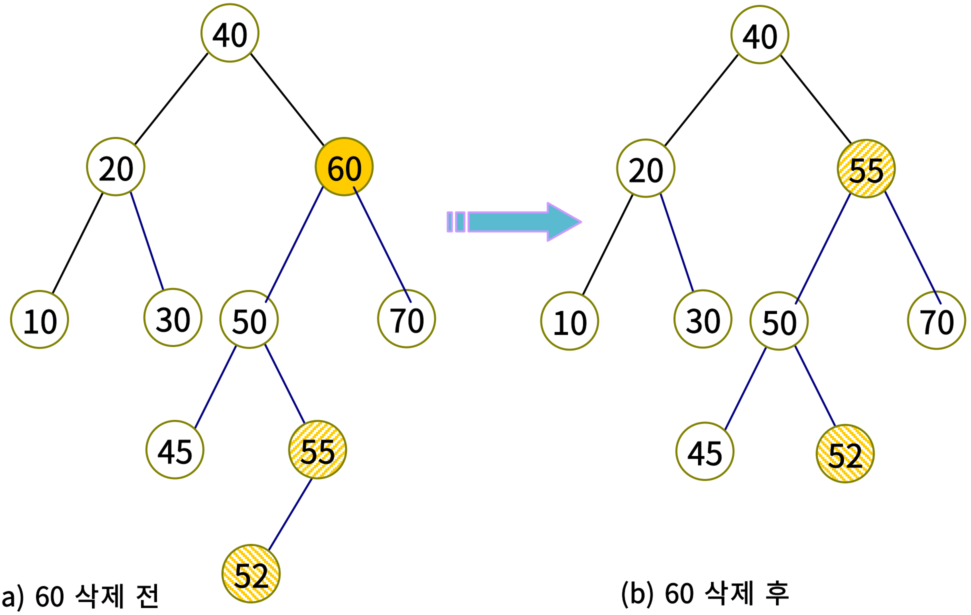
[BST 에서 삭제]



(a) 35 삭제

(b) 40 삭제

[자식노드가 2개인 노드의 삭제 예]



(a) 60 삭제 전

(b) 60 삭제 후

4. 이진 탐색 트리(BST, Binary Search Tree)

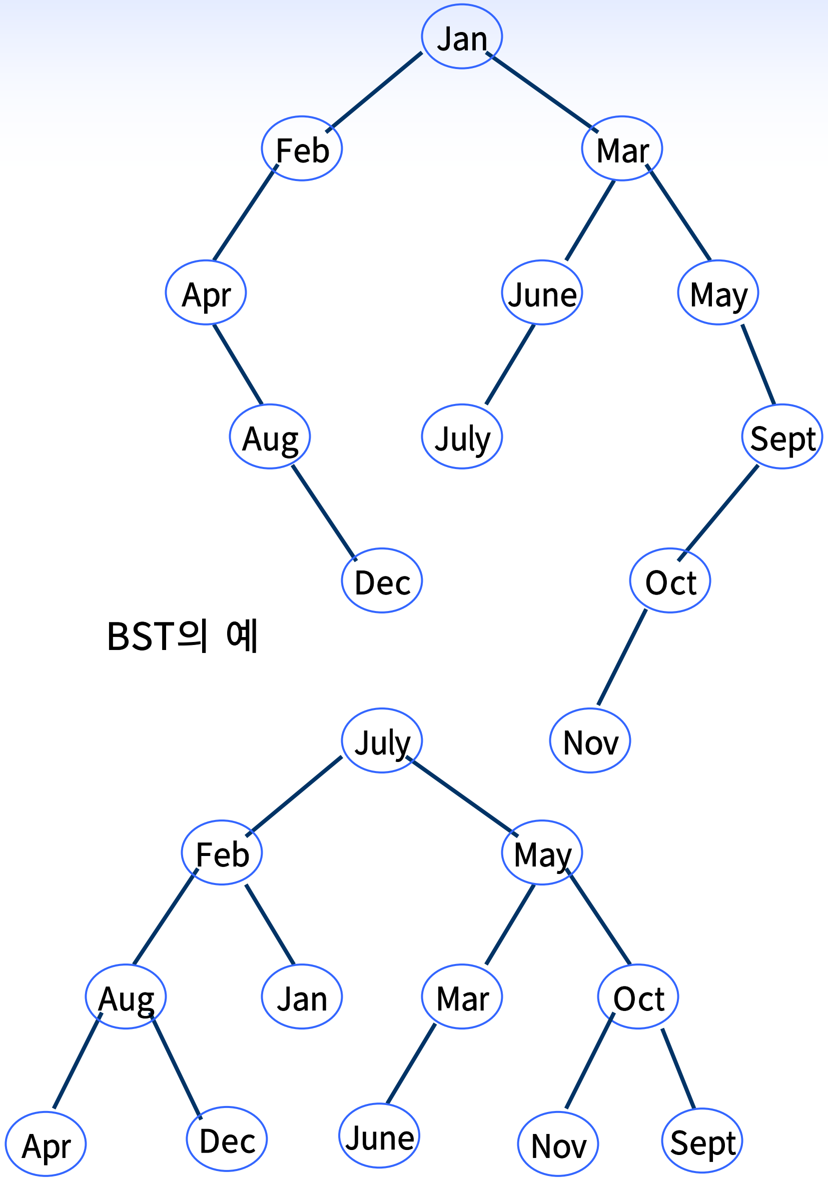
(BST의 단점)

시간복잡도가 다음과 같다. 최악의 경우가 문제가 된다. BST 에서 삽입 순서에 때라 최악의 경우 skewed 트리가 된다.

- 평균(average case) : $O(\log_2 n)$
- 최악(worst case) : $O(n)$

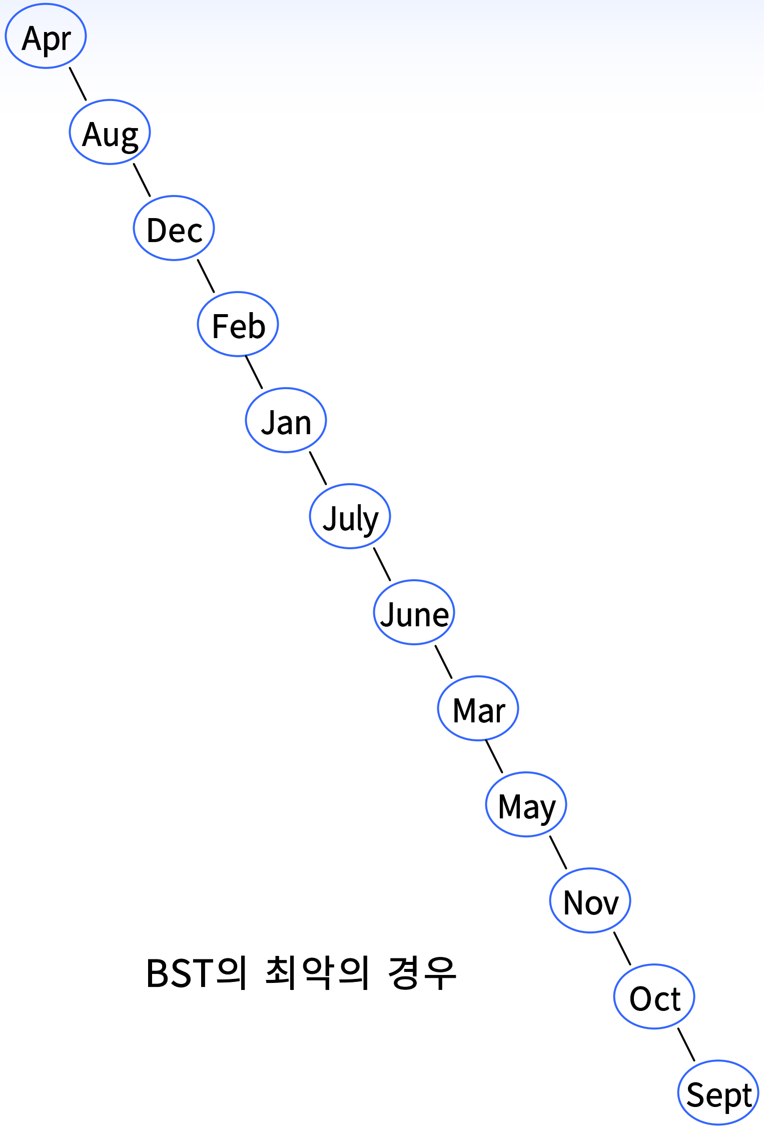
이진 탐색 트리를 완전 이진 트리로 바꿀 수 있다면 최악의 경우를 없앨 수 있다.

- 새로운 원소를 삽입할 때 삽입 시간을 줄인다.
- 평균과 최악의 시간이 같다. $O(\log_2 n)$



BST의 예

균형을 잡은 트리



BST의 최악의 경우

5. AVL 트리

(AVL 트리)

- 균형 이진 트리(balanced binary trees)이다. 이진 트리에서 모든 노드의 왼쪽과 오른쪽 트리의 높이차를 1이하로 만든 트리이다.

-평균과 최악의 경우(average and worst case) : $O(\log_2 n)$

(정의) 높이균형 이진 트리(height balanced binary tree)

- 빈 트리이거나, 아닐 경우

- T가 이진 트리이고 T_L 과 T_R 을 왼쪽과 오른쪽 부속 트리라고 하면, T를 높이균형(height balanced) 트리라고 하며 다음을 만족한다.

1) T_L 과 T_R 이 높이균형(height balanced) 트리이고,

2) $|_{h}L - h_R| \leq 1$, h_L 과 h_R 이 T_L 과 T_R의 높이

(정의) 균형인자(balance factor), BF(T)

- $h_L - h_R$, h_L 와 h_R는 트리의 왼쪽과 오른쪽 트리의 높이(height)

- AVL 트리의 모든 노드에 대하여 BF(T) = -1, 0, 혹은 1이다.

(높이균형 이진트리 만들기)

비어있는 트리로 부터 혹은 이미 구성된 AVL 트리로 부터 노드를 삽입하거나 삭제할 경우 트리의 높이 균형이 깨지면서 BF 값이 +2 혹은 ?2가 될 수 있다. 균형이 깨진 트리의 모양에 따라 다음과 같이 구분하여 균형을 다시 맞춘다. 균형을 맞추려면 다음과 같이 회전을 하여야 한다.

4가지 회전 - BF 값이 +2, -2가 되는 노드에 대하여 다음과 같이 회전한다.

- LL, LR, RR, RL
- LL 과 RR 은 대칭(symmetric)
- LR 과 RL 은 대칭(symmetric)

- Y : 새로 삽입된 노드
- A : Y의 조상으로 균형인자가 ±2 되는 노드

LL : Y 가 A의 왼쪽의 왼쪽 부속 트리에 삽입됐을 때

LR : Y 가 A의 왼쪽의 오른쪽 부속 트리에 삽입됐을 때

RR : Y 가 A의 오른쪽의 오른쪽 부속 트리에 삽입됐을 때

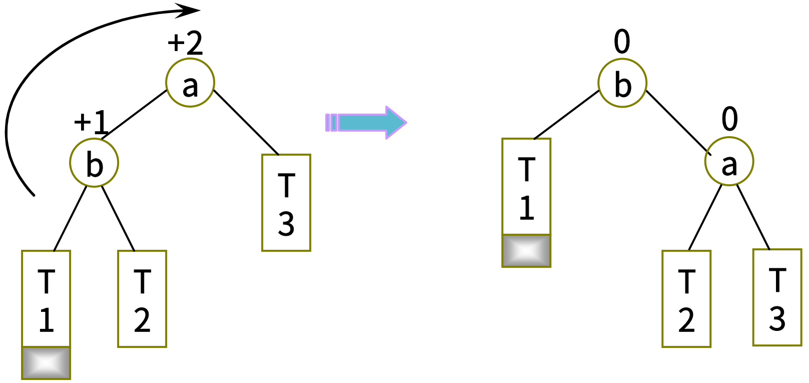
RL : Y 가 A의 오른쪽의 왼쪽 부속 트리에 삽입됐을 때

5. AVL 트리

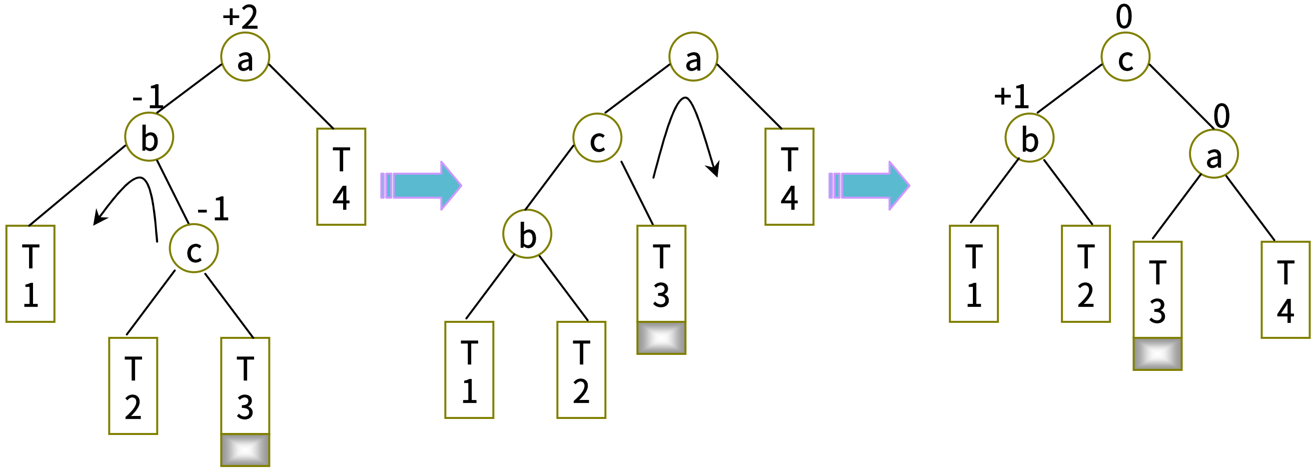
1) LL(Left high, go Left) rotation

- 왼쪽자식을 루트로 만든다.

```
algorithm_LL
        temp <- left(pivot)
        left(pivot) <- right(temp)
        right(temp) <- pivot
        pivot <- temp
```



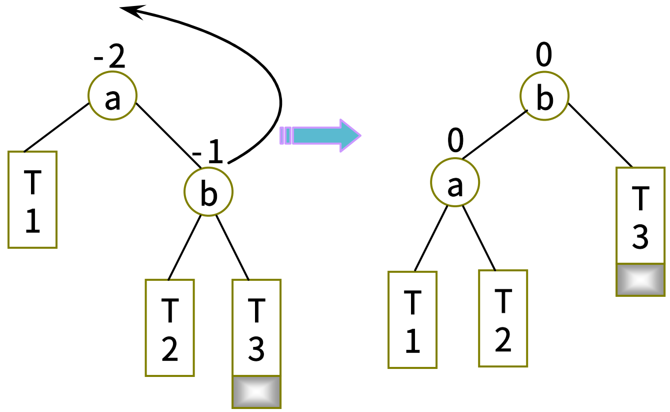
2) LR rotation



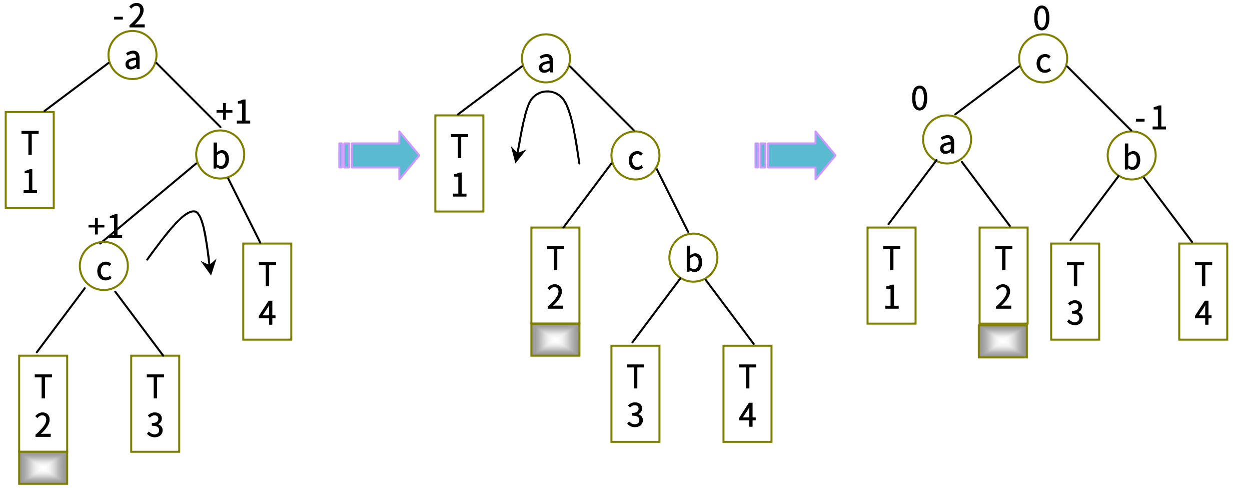
3) RR(Right high, go right) rotation

- 오른쪽 자식을 루트로 만든다.

```
algorithm_RR
    temp <- right(pivot)
    right(pivot) <- left(temp)
    left(temp) <- pivot
    pivot <- temp
```

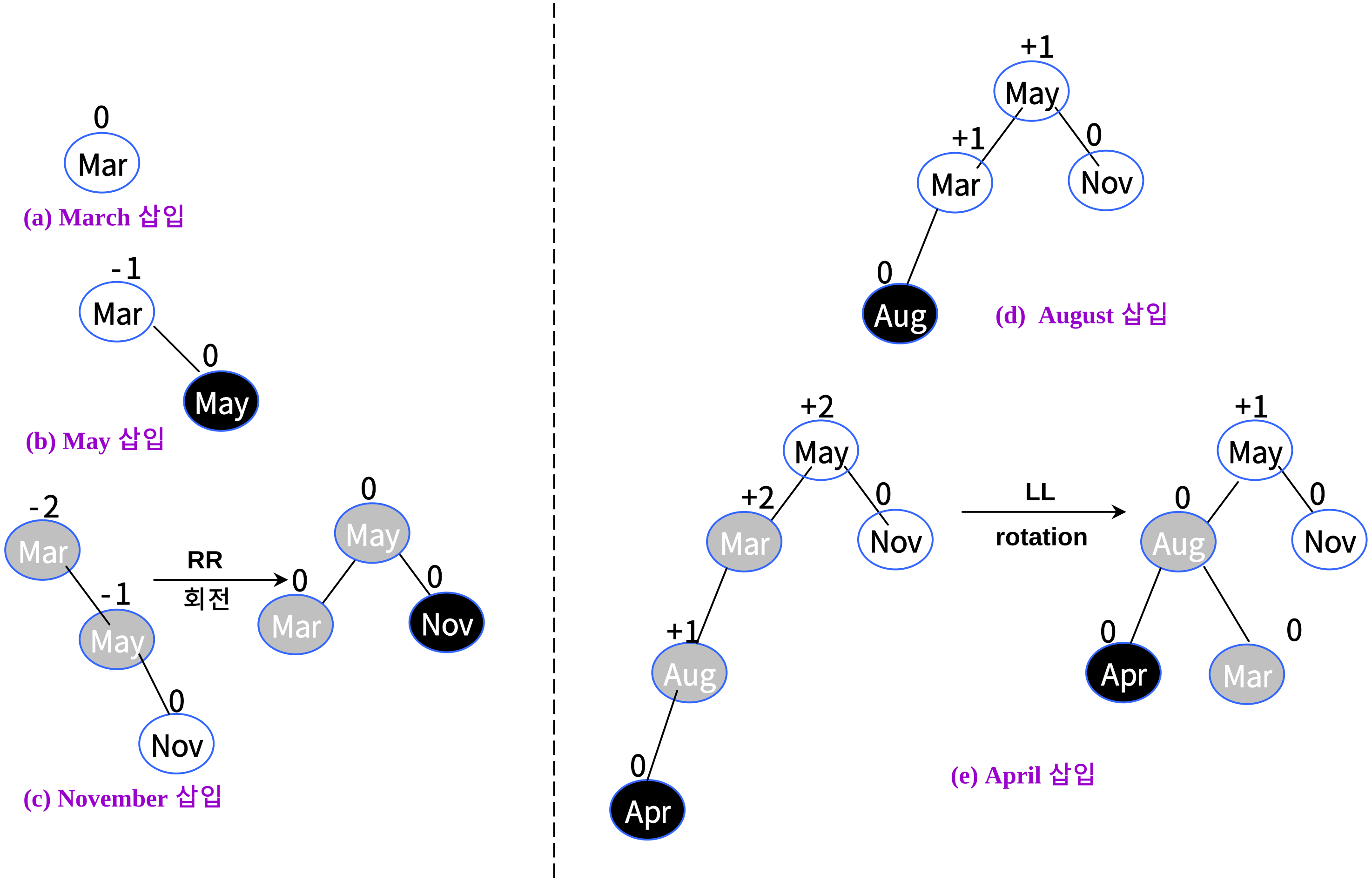


4) RL rotation



5. AVL 트리

예) AVL 트리에 데이터를 삽입하고 회전하는 예



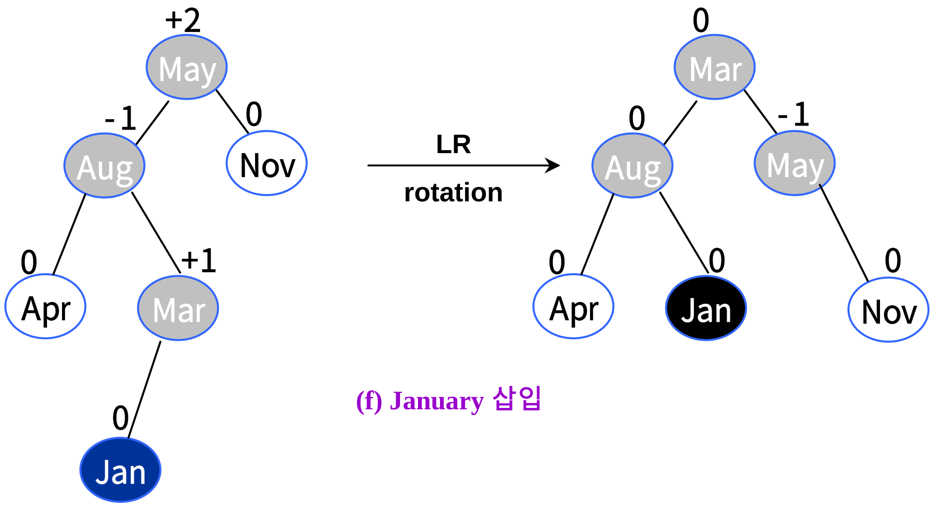
(a) March 삽입

(b) May 삽입

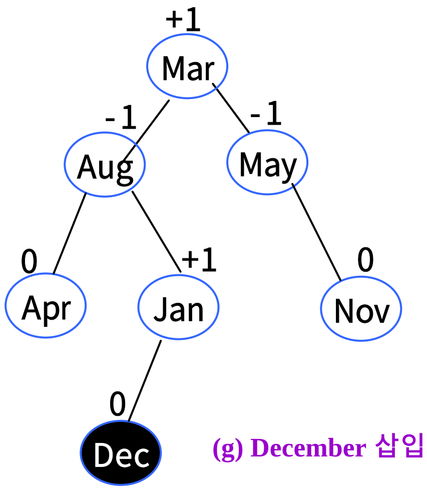
(c) November 삽입

(d) August 삽입

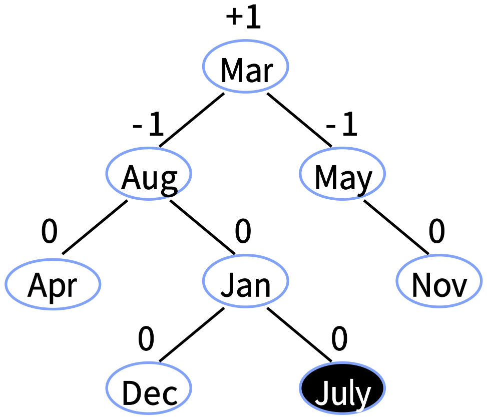
(e) April 삽입



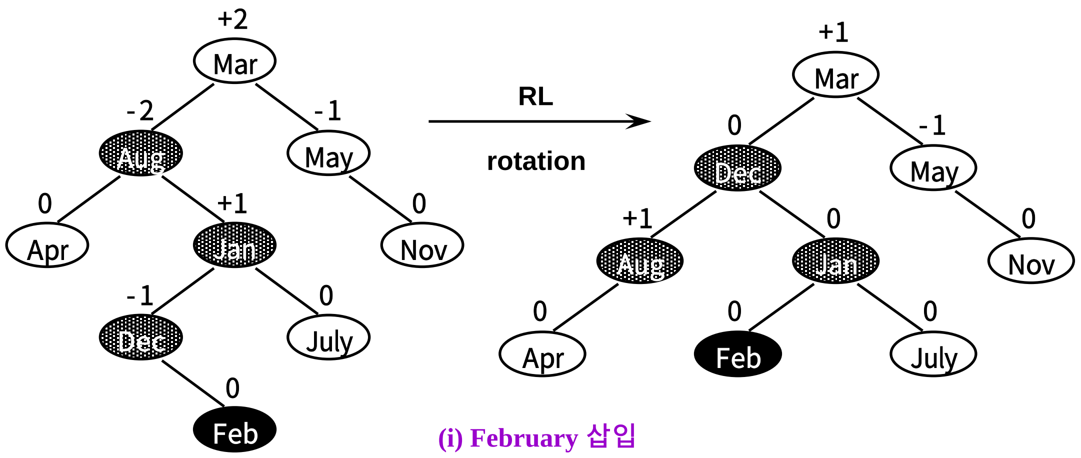
(f) January 삽입



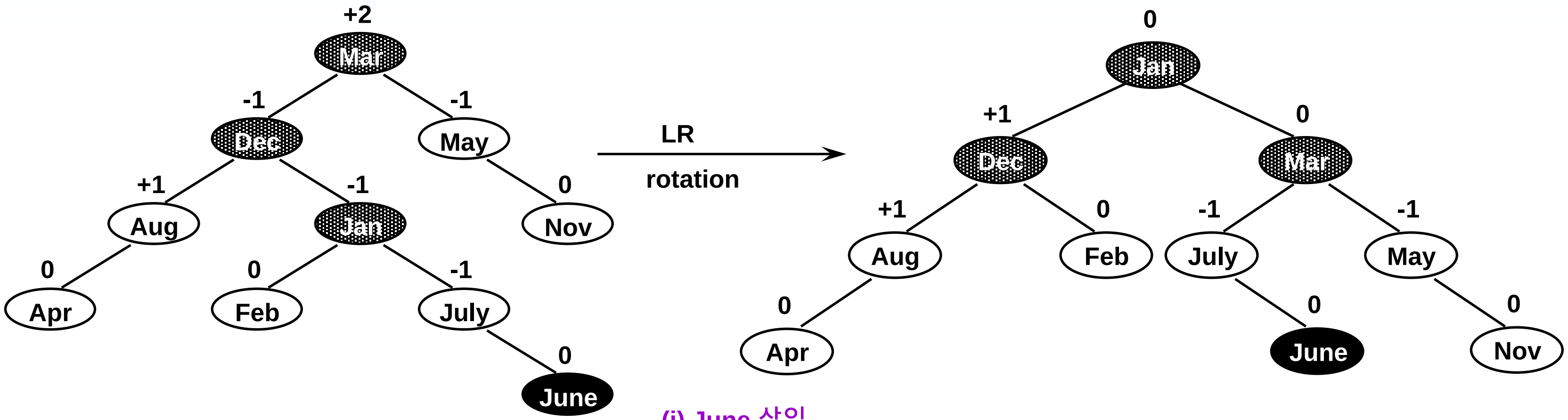
(g) December 삽입



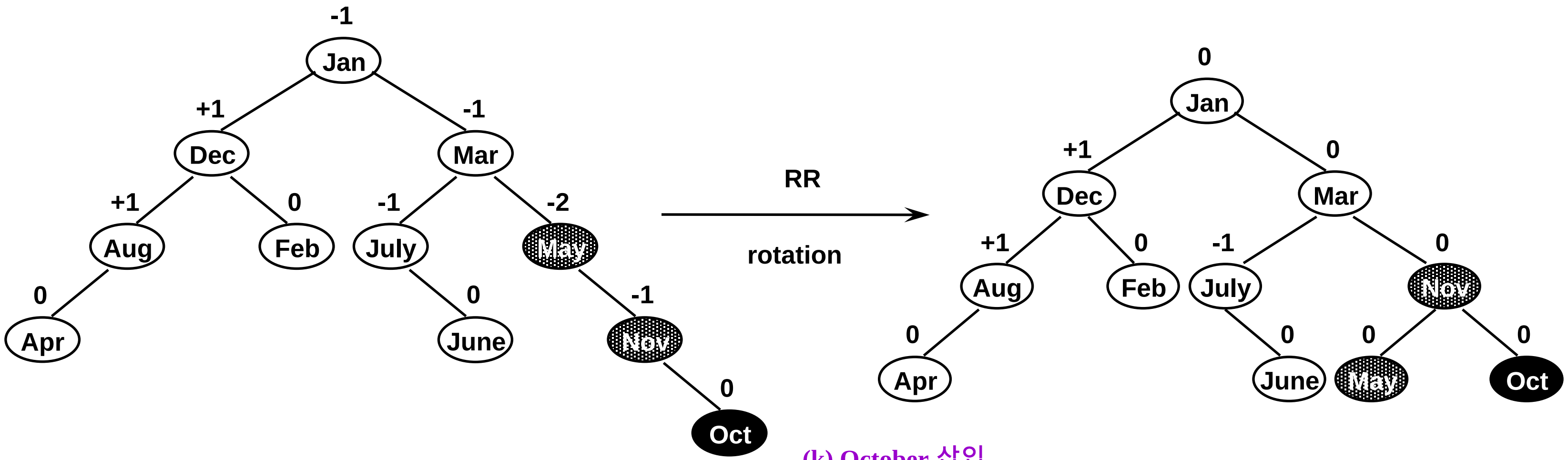
(h) July 삽입



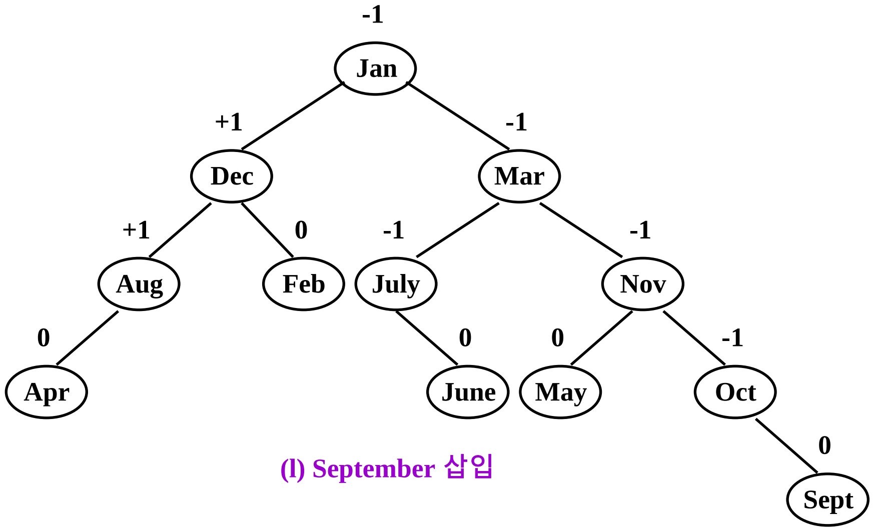
(i) February 삽입



(j) June 삽입



(k) October 삽입



(l) September 삽입

5. AVL 트리

AVL 트리의 자료구조 선언

```
/* AVL 트리의 자료구조 선언 */
#define IS_FULL(ptr) (!(ptr))
#define FALSE = 0
#define TRUE = 1
typedef struct {
        int key;
} element;
typedef struct tree_node *tree_ptr;
struct tree_node {
        tree_ptr left_child;
        element data;
        short int bf;
        tree_ptr right_child;
};
int unbalanced = FALSE;
tree_ptr root = NULL;
```

(AVL 트리의 분석)

AVL 트리는 이진 탐색 트리의 단점을 극복하여 트리의 높이를 균형을 맞춘 것이다.

트리의 높이가 log n이기 때문에 검색시간의 복잡도는 O(log n)이다.

데이터를 삽입하고 삭제할 때 트리의 균형인자(balance factor)가 깨지는 경우가 있기 때문에 균형인자가 +2 혹은 ?2되는 노드를 대상으로 트리를 회전하여 균형을 다시 맞추어야 한다.

검색,삽입,삭제의 평균과 최악의 경우(average and worst case) : $O(log_2n)$

6. B-tree

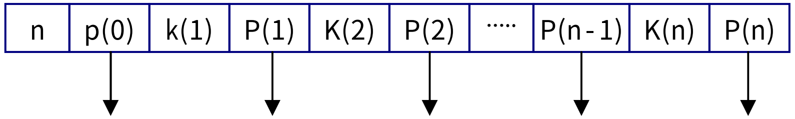
(m-way 탐색트리)

m-way 탐색 트리는 이진 트리와 달리 각 노드에서 나가는 자식노드가 최대 m인 탐색 트리를 말한다.

m-way 탐색 트리의 성질

- 트리에 있는 각 노드는 아래와 같은 구조를 갖는다.

  n은 노드에 있는 키 값 개수, P(i)는 자식 노드 링크, K(i)는 키 값
- 한 노드 안에 키 값은 오름차순으로 정렬
- P(i)가 가리키는 자식 노드 트리에 있는 노드의 모든 키 값은 K(i) 값보다 크고 K(i+1)보다 작다
- P(i)가 가리키는 자식 노드도 역시 m-way 탐색 트리이다.

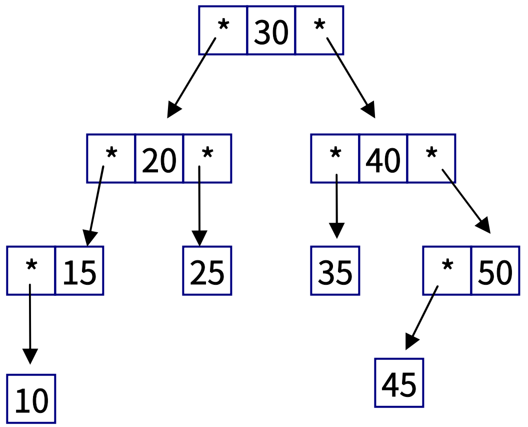


n	p(0)	k(1)	P(1)	K(2)	P(2)	·····	P(n-1)	K(n)	P(n)

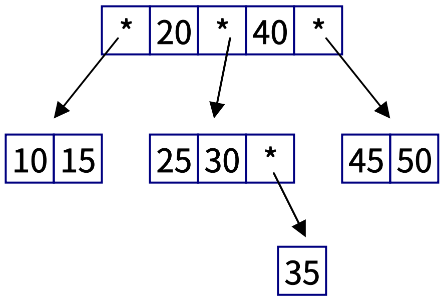
[노드의 구조]

예) 2진 탐색 트리와 3-way 탐색 트리의 비교

(데이터 : 30, 20, 40, 15, 25, 10, 35, 50, 45)



이진탐색트리



3-way 탐색트리

- 3-way 트리는 2진 트리에 비해 높이가 낮으며 폭이 넓다.
- 2진 탐색트리에서의 탐색과정은 한 단계 진행할 때에 노드 개수가 반으로 줄어 들지만 3-원트리에서는 약 1/3로 줄어든다.

6. B-tree

(차수 m의 B-트리)

m-원 탐색 트리의 성능향상을 위해서는 m을 크게 하고 트리가 균형이 되도록 해야 한다. m-원 탐색 트리에 트리의 높이에 제한을 두어 탐색을 효율적으로 진행되도록 한 것이 B-트리이다.

(정의) B-트리는 m-way 탐색 트리로 다음을 만족한다.

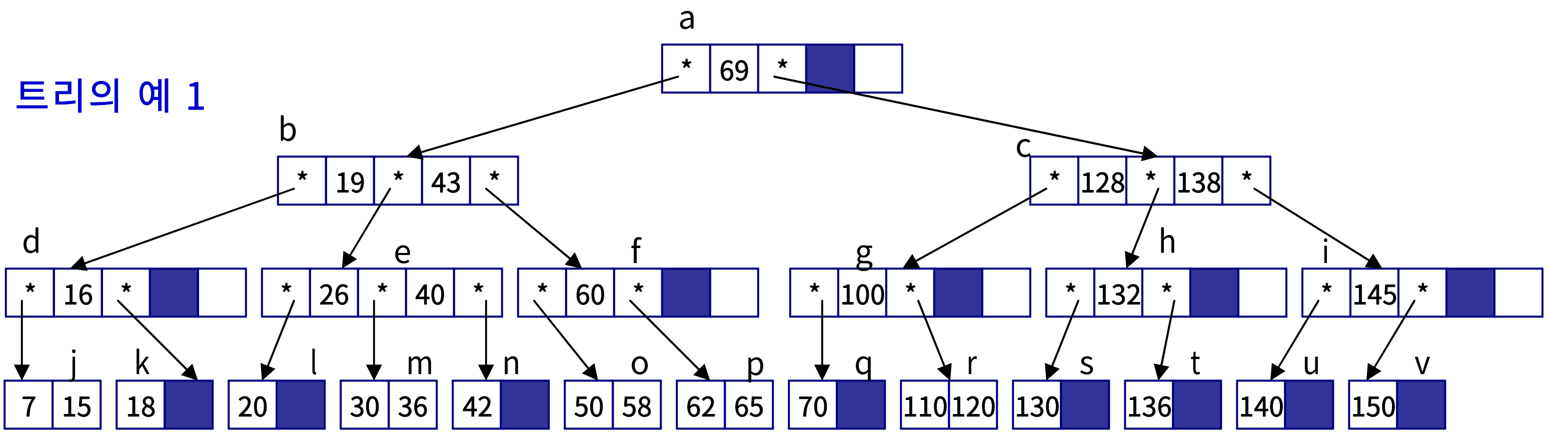
1) 트리에 있는 각 노드는 최대 m개, 최소 m/2개의 종속 트리를 갖는다.

2) 뿌리노드는 최소한 두개의 종속 트리를 갖는다.

3) 모든 잎 노드는 같은 서열에 있어야 한다.

4) 노드에 있는 키 값 개수는 종속 트리의 개수보다 하나 적으며 최소 (m/2)-1개, 최대 m-1개이다.

B 트리의 예 1



6. B-tree

(B 트리에서 키 값의 삽입)

① B-트리에서 연산을 수행한 후에도 B-트리의 성질을 보존해야 함.

② B -트리는 거의 균형된 트리이므로, AVL트리에서의 경우와 같이 연산이 끝난 후에 트리가 균형되어 있어야 한다.

③ 새로운 키값은 항상 잎에 삽입된다.

④ 해당 노드가 가득 차 있지 않은 경우는 키 값을 오름차순으로 채운다.

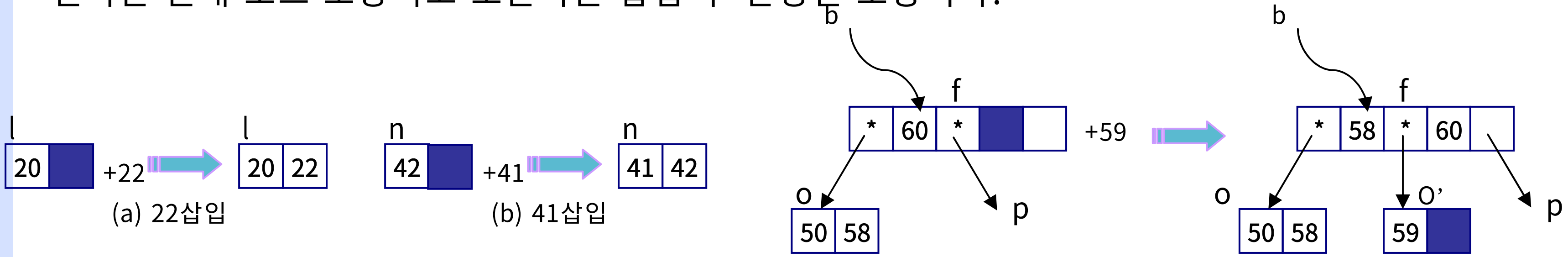
⑤ 해당 노드가 가득 찬 경우 해당노드를 두개의 노드로 분열한다.

※ 노드의 분열 방법

- 해당 노드의 키 값들과 새로운 키 값 중에서 중간 키 값을 부모 노드로 올려 보내고, 나머지 키 값들을 전반으로 나누어 각 절반씩 분열된 두개의 노드에 옮기며, 이 두 노드들을 부모 노드로 올라간 키 값의 왼쪽과 오른쪽 종속 트리가 되도록 한다.

(B 트리에서 삽입의 예)

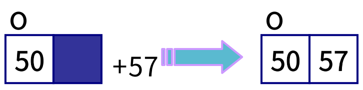
그림 1의 B-트리에 다음의 키 값을 차례로 추가 삽입한다. (22, 41, 59, 57, 54)
왼쪽은 원래 노드 모양이고 오른쪽은 삽입 후 변경된 모양이다.



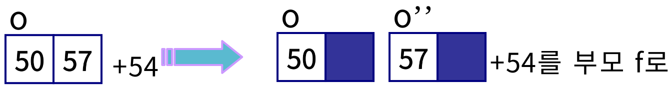
(a) 22삽입

(b) 41삽입

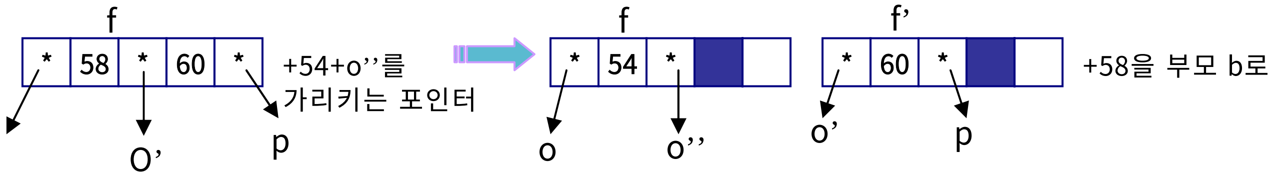
(c) 59 삽입 – 노드의 분열



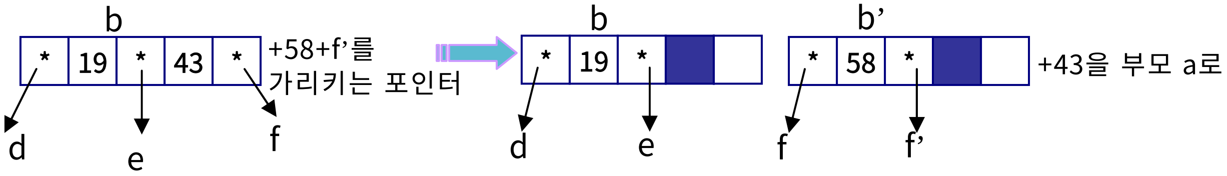
(d) 57삽입



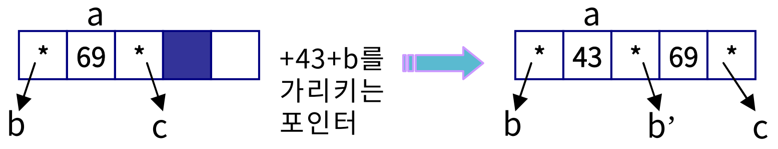
(e) 54삽입 – 노드 분열,54를 부모로 보냄



(f)



(g)



(h)

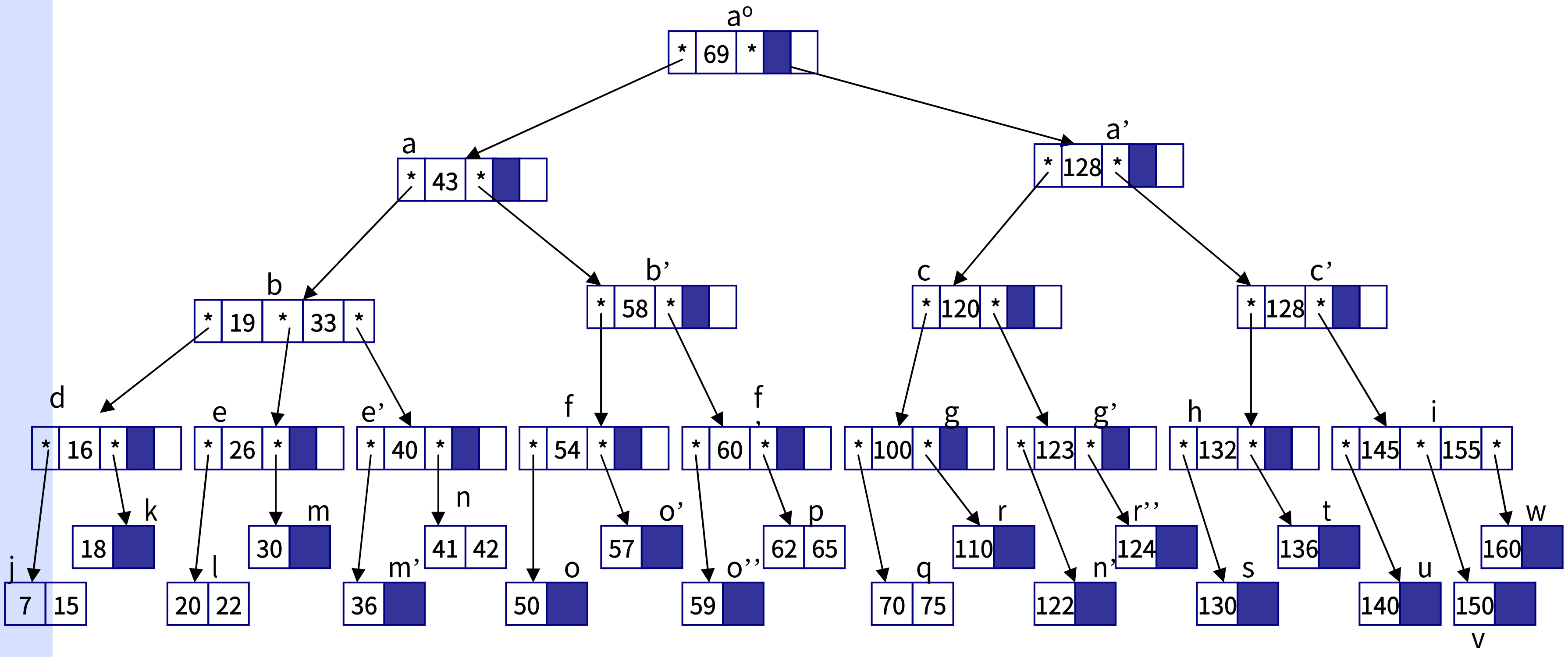
이후에 33,75, 124, 122, 160, 155의 삽입은 쉽게 수행된다.

6. B-tree

(차수 m의 B-트리의 데이터 삽입)

그림 1에 앞에서 설명한 과정을 거쳐서 데이터를 삽입한 후의 결과는 다음과 같다.

삽입된 데이터 : (22, 41, 59, 57, 54, 33, 75, 124, 122, 160, 155, 123)



B 트리의 예 2 – 예 1 에서 데이터의 삽입

6. B-tree

(B 트리에서의 삭제)

① 삭제 연산도 삽입연산에서와 같이 잎 노드에서 시작한다.

② 삭제하려는 키 값이 잎이 아닌 노드에 있다면 그 키 값의 후행 키 값과 일단 자리를 바꾸어 잎 노드로 옮긴 후에 삭제한다.

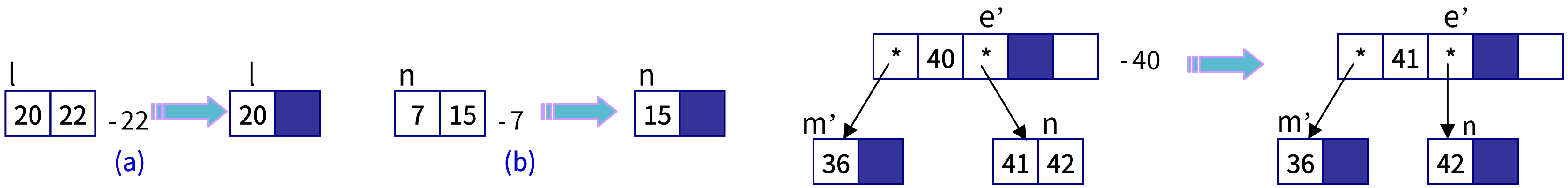
③ 재배치 방법은 해당 노드의 오른쪽 또는 왼쪽 형제노드에 최소 키 값 개수보다 많은 키 값이 있을 경우 사용된다.

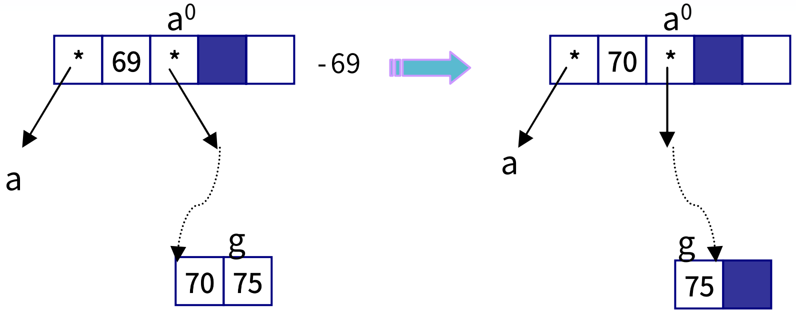
- 선택된 형제노드로부터 한 개의 키 값을 해당 노드로 이동
- 결국 부모노드에 있던 키 값이 해당 노드로 이동되고 그 자리에 형제 노드로부터 키 값이 이동된다.

④ 재배치 방법이 불가능하면 삽입연산에서 사용한 분열방법의 반대 과정인 합병방법을 사용한다.

- 해당노드와 오른쪽쪽 또는 왼쪽 형제노드에 있는 키 값들과, 이 두 노드의 부모노드에 있는 해당 키 값을 하나의 노드로 합병한다.
- 합병당한 두 노드중의 하나를 새로 만들어진 노드로 대치하고 나머지 한 노드는 제거한다.

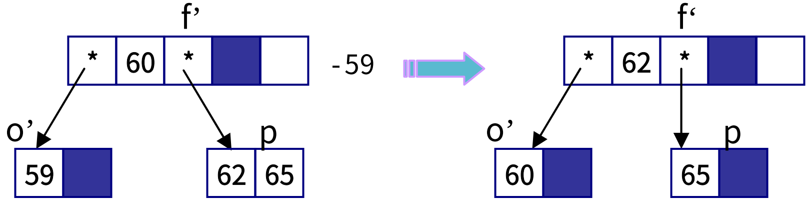
예) B-트리에서 데이터 삭제 예 (22, 7, 40, 69, 59, 150, 16, 128)





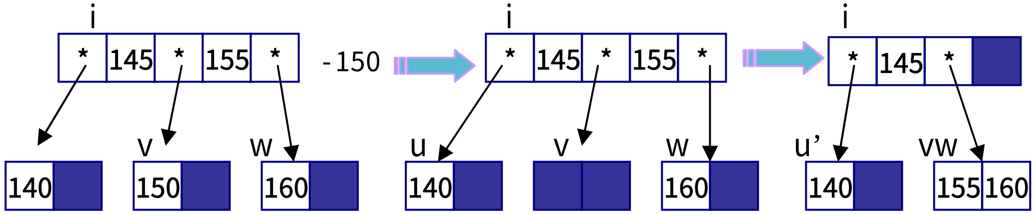
a0
*
69
*
- 69
a
g
70
75
a0
*
70
*
a
g
75

(d)



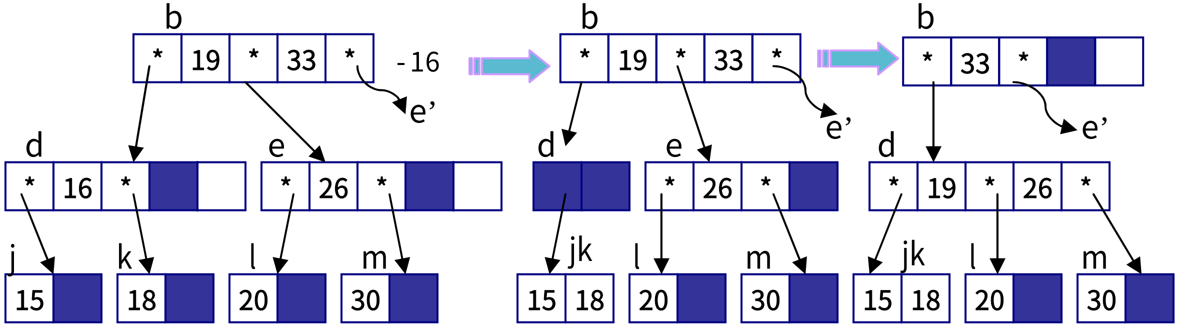
f'
*
60
*
- 59
o'
p
59
62
65
f'
*
62
*
o'
p
60
65

(e)



i
*
145
*
155
*
- 150
v
w
140
150
160
i
*
145
*
155
*
u
v
w
140
160
i
*
145
*
u'
vw
140
155
160

(f)



b
*
19
*
33
*
- 16
e'
d
*
16
*
e
*
26
*
j
k
l
m
15
18
20
30
b
*
19
*
33
*
e'
d
e
*
26
*
jk
l
m
15
18
20
30
b
*
33
*
e'
d
*
19
*
26
*
jk
l
m
15
18
20
30

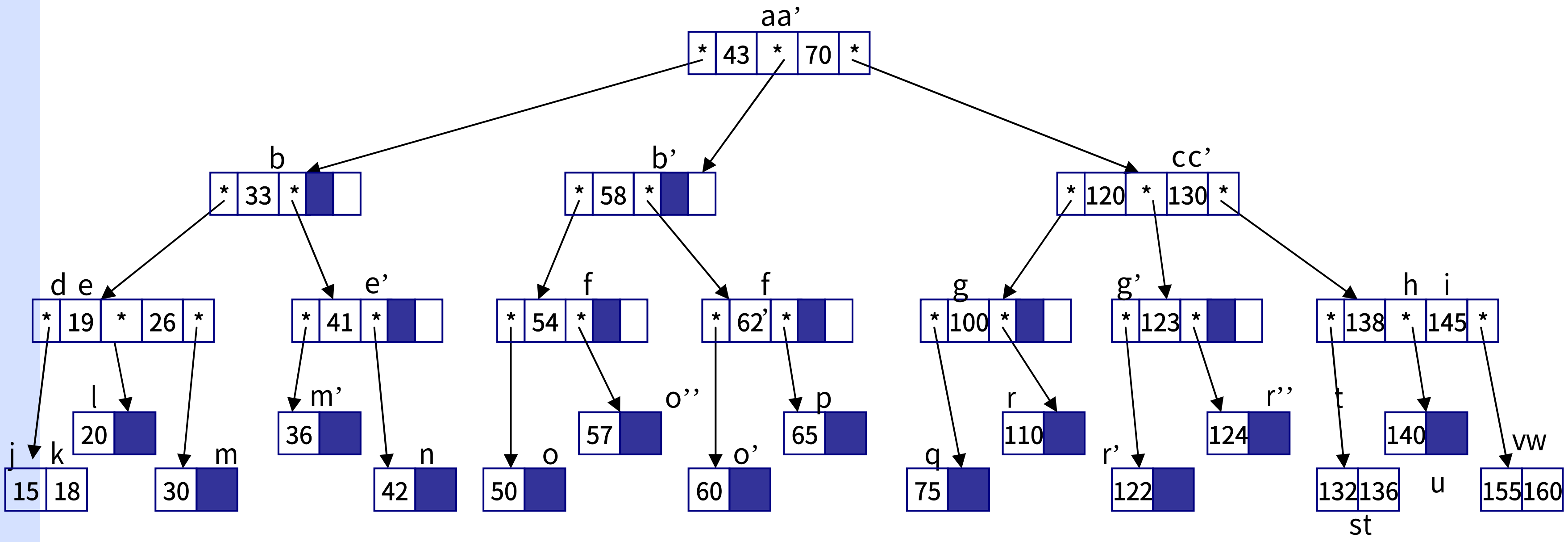
(g)

6. B-tree

(차수 m의 B-트리의 데이터 삭제)

그림 2에 앞에서 설명한 과정을 거쳐서 데이터를 삭제한 후의 결과는 다음과 같다.

삭제된 데이터 : (22, 7, 59, 40, 69, 16, 128)



B 트리의 예 3 – 예 2 에서 데이터의 삭제

6. B-tree

(B 트리의 분석)

- B-트리는 주로 보조기억장치에 저장된 데이터의 탐색에 이용된다. 보조기억장치의 데이터 탐색은 키값의 비교회수보다 디스크 블록을 읽어오는 횟수가 성능에 더 많은 영향을 끼친다. B-트리의 노드는 디스크의 블록(block) 크기 정도에 저장되면 탐색 시 이 블록을 읽어오게 된다.
- 데이터 검색 시 디스크 접근 횟수는 $O(\log_m n)$에 비례한다.
- 일반적으로 m의 값은 32 ~ 256 정도이다.
- 가득 찬 노드가 두 개의 노드로 분열되어도 두 노드는 최소한의 키 값 개수와 포인터 개수를 갖도록 보장된다.
- 또한 두 노드가 합병될 때에도 합병된 노드는 최대 허용 키 값 개수 이상을 갖지 않도록 보장된다.
- 키 값을 순차적으로 배열한 것은 트리의 어느 키 값으로부터도 그 값보다 작거나 또는 큰 키 값을 쉽게 찾도록 하기 위해서 이다.

학습내용정리

선형탐색은 기본적인 탐색으로 데이터의 정렬여부에 관계없이 사용할 수 있다. 탐색시간은 O(n)이다.

이진탐색은 정렬된 데이터에 대하여 탐색할 수 있는 방법이다. 탐색시간은 O(log n)이다.

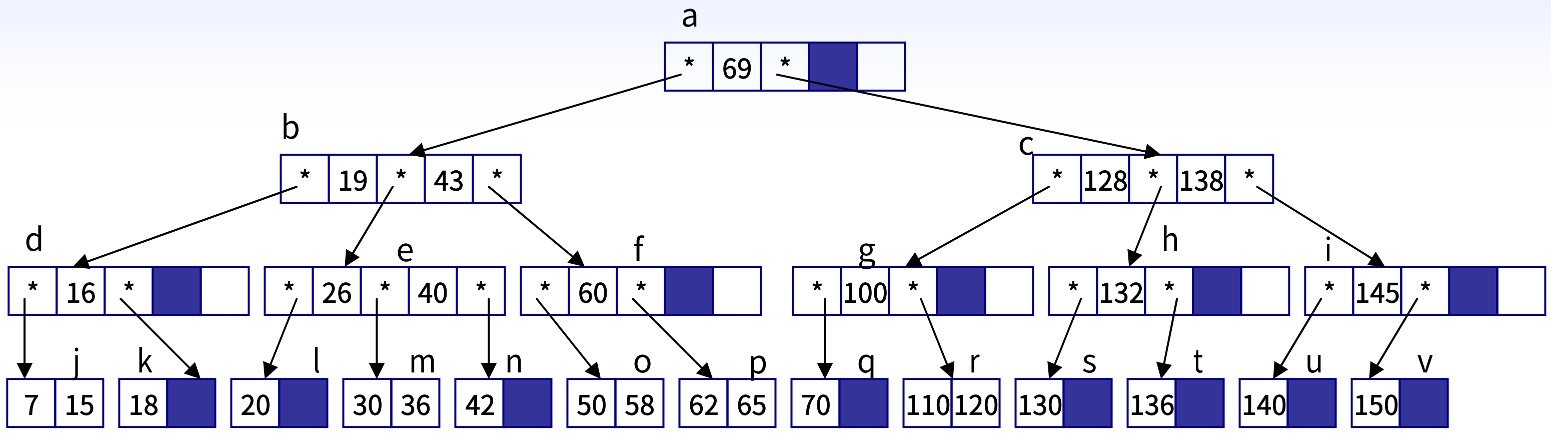
해싱은 비교를 하지 않고 데이터를 찾는 방법이다. 해시 함수를 이용하여 데이터 삽입, 삭제, 검색을 한다. 검색 중에서는 가장 빠른 방법이다. 그러나 좋은 해시함수와 충분한 기억공간이 필요하다.

이진 탐색 트리는 트리 구조를 이용한 탐색 방법의 시작이다. 탐색시간은 O(log n)이다. 그러나 트리가 균형을 이루지 못하면 탐색시간이 오래 걸린다.

AVL 트리는 이진 탐색 트리를 보완한 방법으로 탐색시간이 항상 O(log n)이 걸린다.

m-way 탐색 트리는 이진탐색의 단점을 보완하여 m개의 키 값을 노드에 저장하여 한번 노드 탐색으로 대상 데이터를 1/m으로 줄이는 방법이다.

B-tree는 m-way 트리의 단점을 보완하여 트리의 깊이를 균형을 이루도록 만든 트리 구조이다. 주로 보조기억장치의 데이터를 탐색할 때 많이 사용한다.



(22, 41, 59, 57, 54, 33, 75, 124, 122, 160, 155, 123)