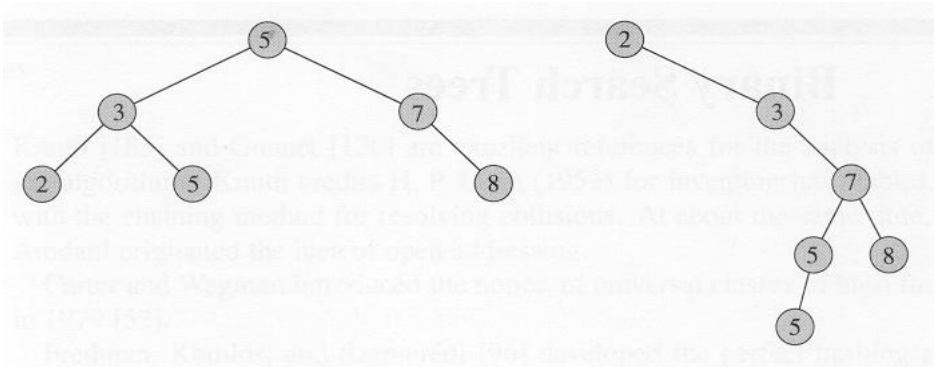
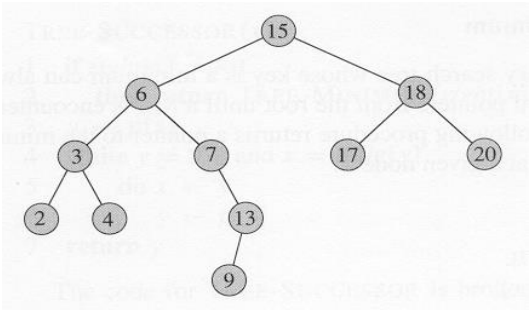
Binary Search Trees

정의

1) Binary tree
2) Binary-search-tree property
   ① if y is a node in the left subtree of x, then key[y] ≤ key[x]
   ② if y is a node in the right subtree of x, then key[x] ≤ key[y]





- 탐색 편리
  - 최소값: 맨 왼쪽 노드
  - 최대값: 맨 오른쪽 노드

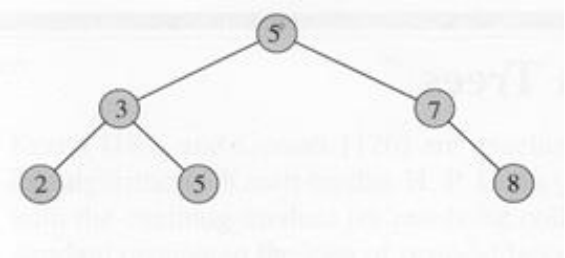
(Q) key = {1, 4, 5, 10, 16, 17, 21}, height = 2 인 B.S.T 를 그려라

Data Member

- T : **linked-list** (Left-child, right-child representation)
  root[T]: root node 의 index

- Node = key field + pointer fields (left, right, parent)

(ex)



Operations

INORDER-TREE-WALK(x)

TREE-SEARCH(x,k)

TREE-MINIMUM(x),

TREE-MAXIMUM(x)

TREE-SUCCESSOR(x)

TREE-PREDECESSOR(x)

TREE-INSERT(T,z)

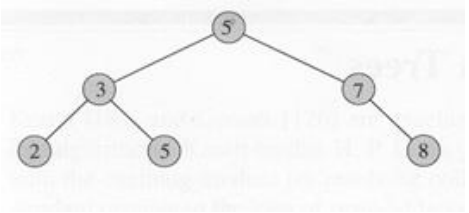
TREE-DELETE(T,z)

Operations

- **INORDER-TREE-WALK(x)**
  - binary search tree (root node 의 index = x) 의 모든 노드를 방문하여, 그 값을 print
  - θ (n), n: node 수

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2     then INORDER-TREE-WALK(left[x])
3          print key[x]
4          INORDER-TREE-WALK(right[x])
```

(ex)



(cf) PREORDER-TREE-WALK(x)
POSTORDER-TREE-WALK(x)

Operations

- **TREE-SEARCH(x, k)**
  - key 값 = k 인 노드의 index 를 찾아서 return
  - x: root node 의 index
  - **O(h)**, h: height of tree
  - 알고리즘
    - If k < key[x], go to left-child
    - If k > key[x], go to right-child

```
TREE-SEARCH(x, k)
1  if x = NIL or k = key[x]
2     then return x
3  if k < key[x]
4     then return TREE-SEARCH(left[x], k)
5     else  return TREE-SEARCH(right[x], k)
```

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ key[x]
2        do if k < key[x]
3              then x ← left[x]
4              else  x ← right[x]
5  return x
```

Operations

(Q) k = 363, TREE-SEARCH(x, k) 수행 시 잘못된 sequence 는 ?

(a) 2, 252, 401, 398, 330, 344, 397, 363

(b) 924, 220, 911, 244, 898, 258, 362, 363

(c) 925, 202, 911, 240, 912, 245, 363

Operations

- **TREE-MINIMUM(x)**
  - 최소값을 갖는 노드를 찾아, 그 index 를 return
  - x : root node 의 index
  - O(h)
  - 알고리즘 : 가장 좌측의 노드 탐색

```
TREE-MINIMUM(x)
1  while left[x] ≠ NIL
2        do x ← left[x]
3  return x
```

Operations

- **TREE-MAXIMUM(x)**
  - 최대값을 갖는 노드를 찾아, 그 index 를 return
  - x : root node 의 index
  - O(h)
  - 알고리즘: 가장 우측의 노드 탐색

```
TREE-MAXIMUM(x)
1  while right[x] ≠ NIL
2        do x ← right[x]
3  return x
```

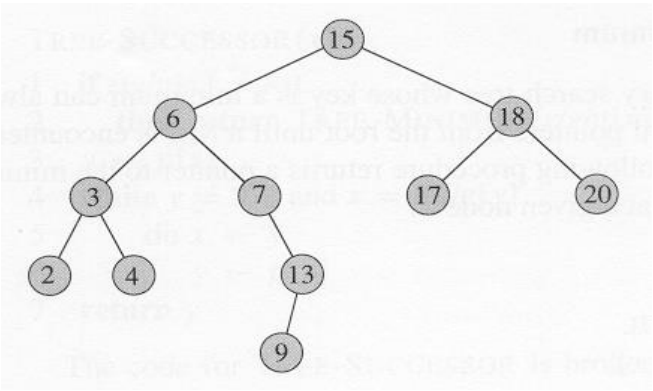
Operations

- **TREE-SUCCESSOR(x)**
  - 노드 x 의 key 값 보다 다음으로 값이 큰 노드를 찾아, 그 index 를 return
  - smallest key greater than key(x)
  - O(h)
  - 알고리즘

    (case 1) right[x] != NIL → TREE-MINIMUM(right[x])

    (case 2) right[x] == NIL → 자기 또는 자기 조상을 left-child 로 하는 노드

```
TREE-SUCCESSOR(x)
1  if right[x] ≠ NIL
2     then return TREE-MINIMUM(right[x])
3  y ← p[x]
4  while y ≠ NIL and x = right[y]
5        do x ← y
6           y ← p[y]
7  return y
```



Operations

(Q) Two-children node 의 successor 는 left child 가 없다.

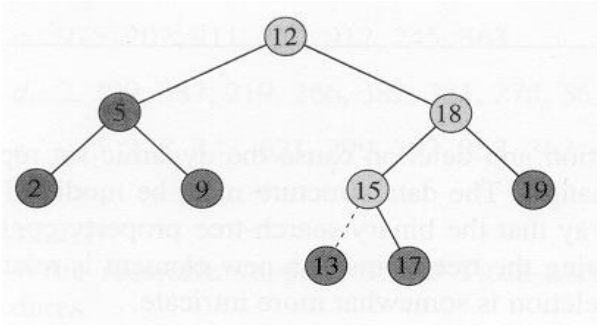
(Q) TREE-PREDECESSOR(x)

- largest key less than key(x)

Operations

- **TREE-INSERT(T,z)**
  - Tree T 에 노드 z 삽입 (key[z] = v, p[z]= NIL, left[z]=NIL, right[z]=NIL)
    단, binary-search-tree 의 property 유지 되도록
  - 알고리즘
    - search 가 종료된 위치에 노드 삽입
  - O(h)

```
TREE-INSERT(T, z)
 1  y ← NIL
 2  x ← root[T]
 3  while x ≠ NIL
 4        do y ← x
 5           if key[z] < key[x]
 6              then x ← left[x]
 7              else x ← right[x]
 8  p[z] ← y
 9  if y = NIL
10     then root[T] ← z                ▷ Tree T was empty
11     else if key[z] < key[y]
12              then left[y] ← z
13              else right[y] ← z
```



Operations

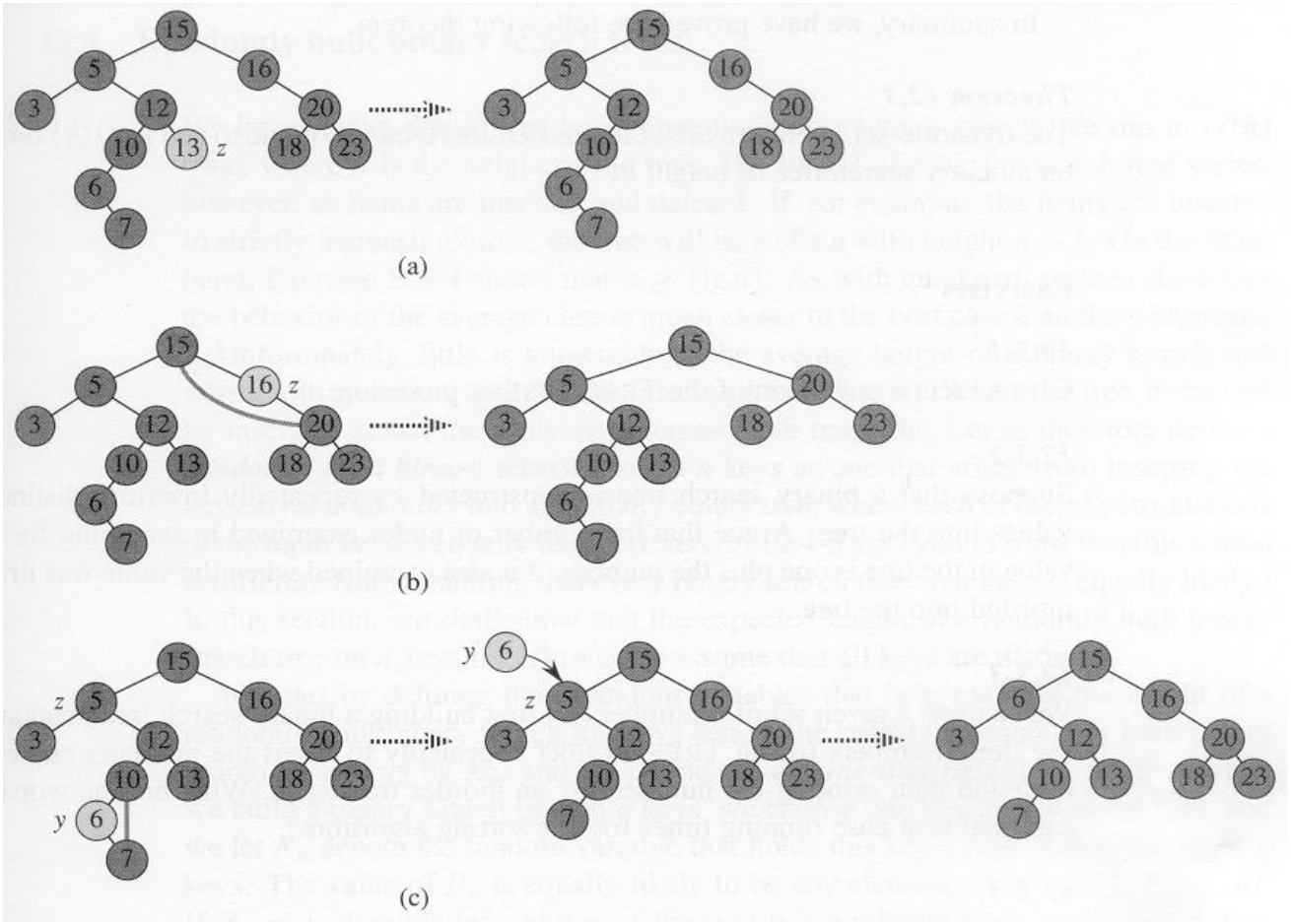
- **TREE-DELETE(T,z)**
  - Tree T 에서 노드 z 삭제
    단, binary-search-tree 의 property 유지 되도록
  - 알고리즘

    (case 1) no children case
    - z 노드 제거
      : parent node (p[z]) 의 left or right 를 z → NIL 로 대체

    (case 2) one child case
    - parent node 와 child node 를 연결
      : p[z] 의 left or right 에 z 의 left or right 대입
      left[z] or right[z] 의 parent 에 p[z] 대입

    (case 3) two children case
    - z 의 successor 노드 y (left child 없음) 에 대하여 (case 2) 실행
      : z 노드의 key 및 satellite 값을 y node 의 값들로 대체
  - O(h)

Operations


15
5
16
3
12
20
10
13
z
18
23
6
7
(a)
15
5
16
z
3
12
20
10
13
18
23
6
7
(b)
15
5
20
3
12
18
23
10
13
6
7
(c)
y
6
z

Operations

```
TREE-DELETE(T, z)
 1  if left[z] = NIL or right[z] = NIL
 2     then y ← z
 3     else y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ NIL
 5     then x ← left[y]
 6     else x ← right[y]
 7  if x ≠ NIL
 8     then p[x] ← p[y]
 9  if p[y] = NIL
10     then root[T] ← x
11     else if y = left[p[y]]
12             then left[p[y]] ← x
13             else right[p[y]] ← x
14  if y ≠ z
15     then key[z] ← key[y]
16          copy y's satellite data into z
17  return y
```