5.1 Introduction

5.2 Binary Trees

5.3 Binary Trees Traversals

5.4 Additional Binary Tree Operations

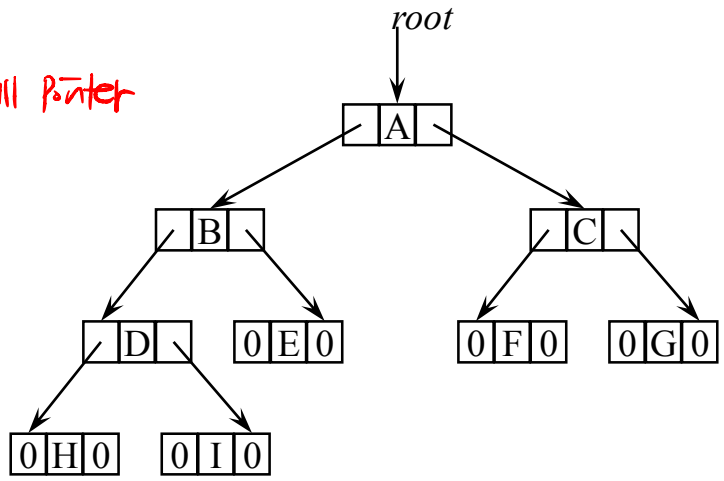
5.5 Threaded Binary Trees

5.6 Heaps

5.7 Binary Search Trees

5.5.1 Threads

- Drawback of the BT:
  - Too many null pointers;
  - Null links: $2n-(n-1) = n+1$



- Replace these null pointers with some useful "threads" to other nodes in the tree
  - To make inorder traversal faster and do it without stack and without recursion

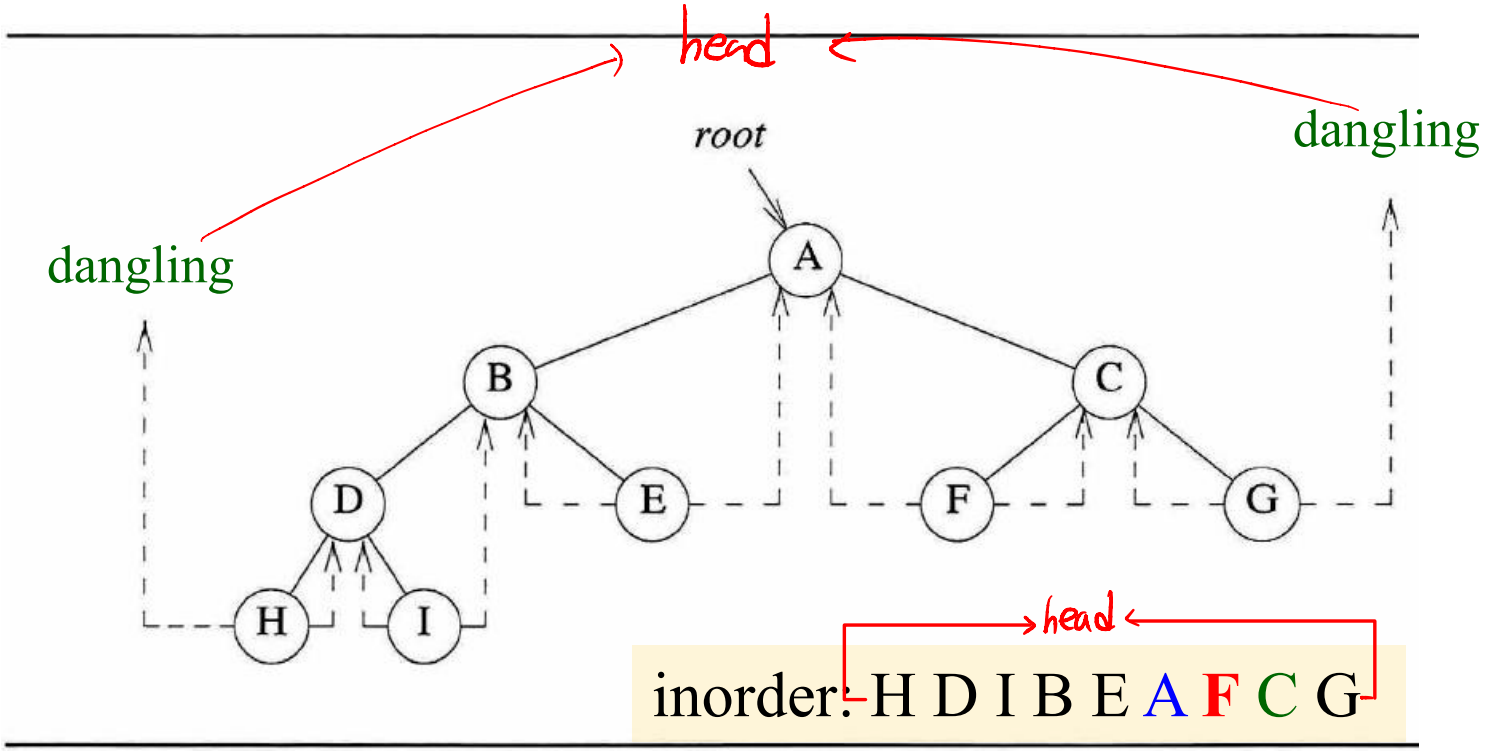


Figure 5.21: Threaded tree corresponding to Figure 5.10(b)

단말 노드에 연결됨

Assume that ***ptr*** represents a node :

1) If **ptr -> leftChild** is null, replace the null link with a pointer to the inorder predecessor of ptr

inorder 방식의 이전 노드

2) If **ptr -> rightChild** is null, replace the null link with a pointer to the inorder successor of ptr

inorder 방식의 다음 노드

- Assume a **header** node for all threaded binary trees:
  - If we don't want the left pointer of H and the right pointer of G to be dangling pointers

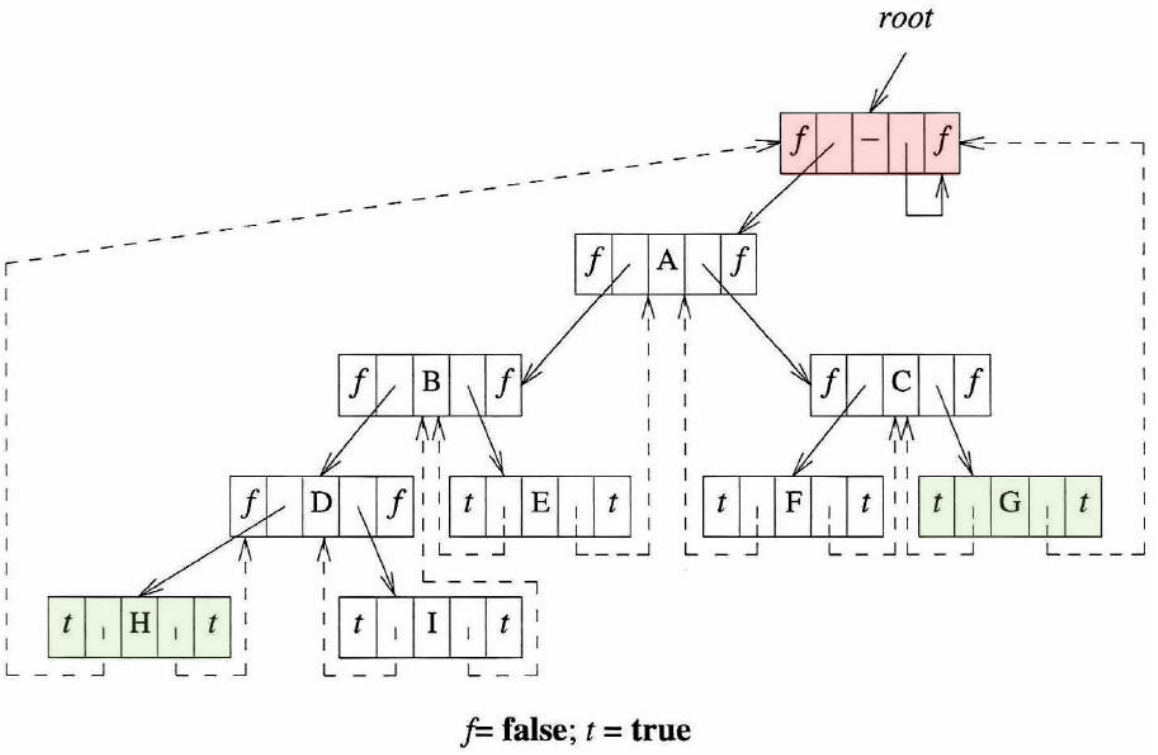


Figure 5.23: Memory representation of threaded tree

- Node structure:
  - Two additional fields : leftThread and rightThread
  - leftThread, rightThread : TRUE / FALSE

기존노드에 두 노드를 추가

```
typedef struct threadedTree *threadedPointer;
typedef struct threadedTree {
        short int leftThread;
        threadedPointer leftChild;
        char data;
        threadedPointer rightChild;
        short int rightThread;
        };
```

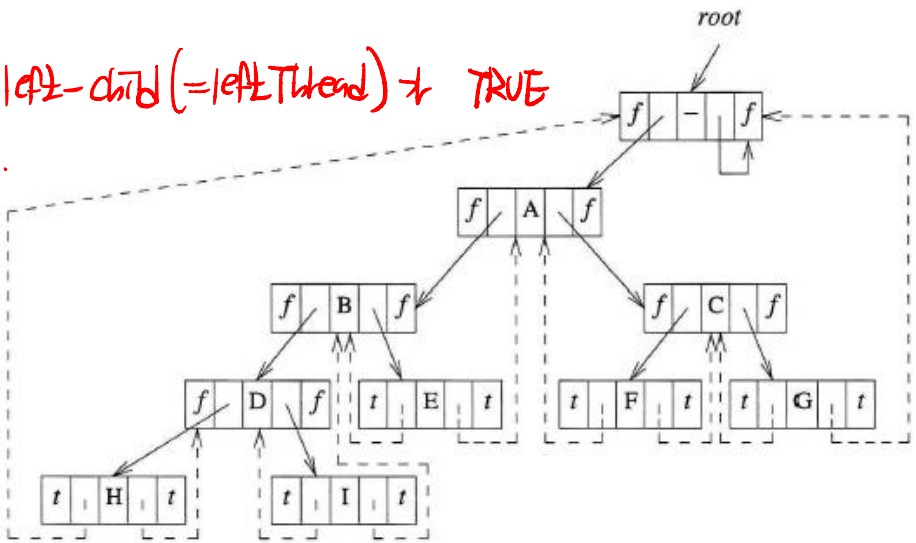
leftChild가 thread 경우면 true 경우 아니면 false

(right)

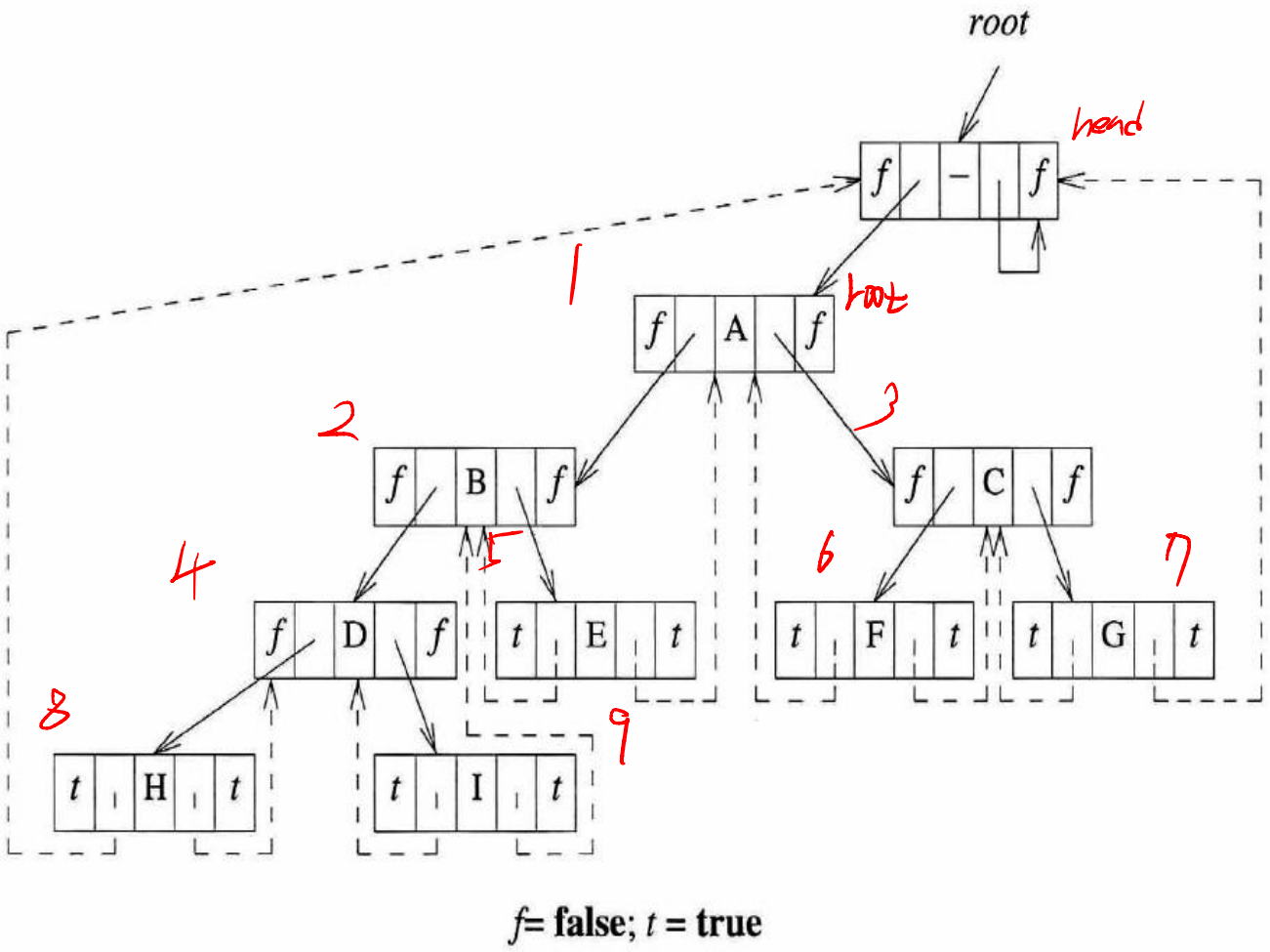
Figure 5.22: An empty threaded binary tree

5.5.2 Inorder Traversal of a Threaded BT

- We can perform an inorder traversal without making use of a stack
- The "**next**" node of inorder traversal for any node, ptr :
  - If ptr->rightThread == TRUE → ptr->rightChild
  - If ptr->rightThread == FALSE → Follow a path of left-child links from the right-child of *ptr* until reaching a node with leftThread = TRUE



f= false; *t* = true



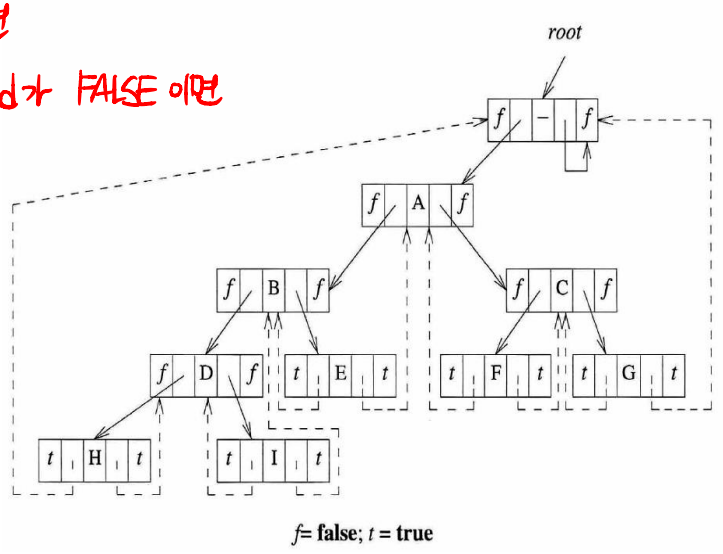
inorder: H D I B E A F C G

- The "next" node of inorder traversal

```
threadedPointer insucc(threadedPointer tree)
{
    threadedPointer temp;
    temp = tree->rightChild;
    if (!tree->rightThread)
        while (!temp->leftThread)
            temp = temp->leftChild;
    return temp;
}
```

Program 5.10:
Finding the inorder successor of a node

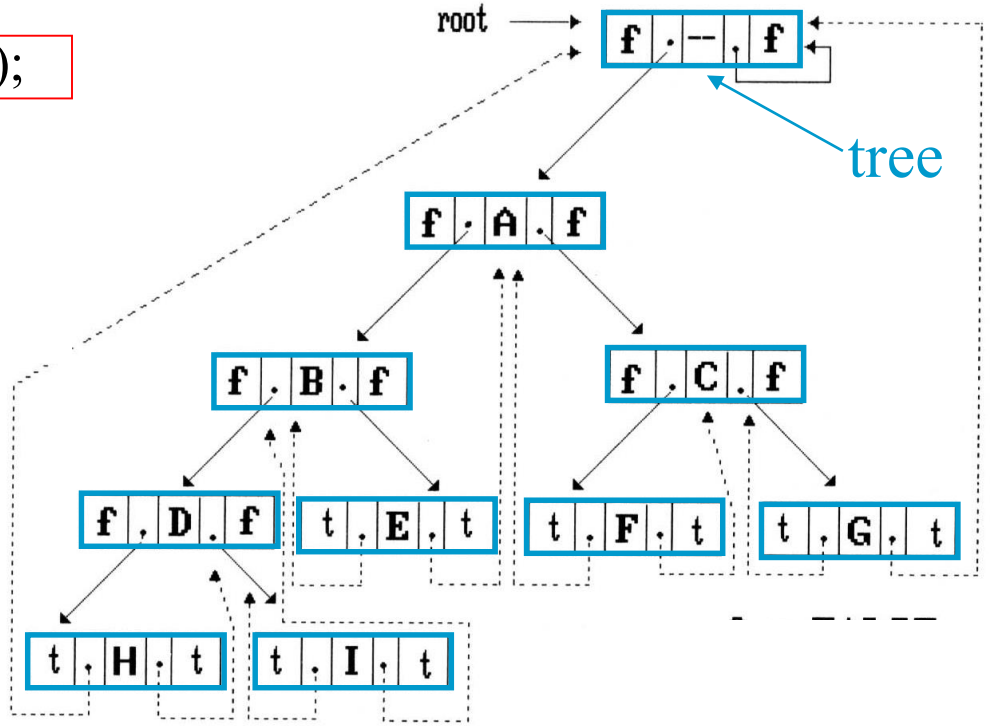
inorder: H D I B E A F C G



Program 5.11:
Inorder traversal of a threaded binary tree

```
void tinorder(threadedPointer tree) {
/* traverse the threaded binary tree inorder */
    threadedPointer temp = tree;
    for (;;) {
      temp = insucc(temp);
      if (temp==tree)
              break;
      printf("%3c",temp->data);
    }
}
```

output: H D I B E A F C G



5.1 Introduction

5.2 Binary Trees

5.3 Binary Trees Traversals

5.4 Additional Binary Tree Operations

5.5 Threaded Binary Trees

5.6 Heaps

5.7 Binary Search Trees

5.6.1 Priority Queues

- Priority queues
  - Collection of elements
  - Each element has a priority or key
  - The element to be deleted is the one with highest (or lowest) priority
  - At any time, an element with arbitrary priority can be inserted into the queue
- **Heaps**
  - A tree with some special properties
  - Frequently used to implement *priority queues*

ADT *MaxPriorityQueue* is

objects: a collection of $n > 0$ elements, each element has a key 키의 값

functions:

for all $q \in$ *MaxPriorityQueue*, *item* $\in$ *Element*, $n \in$ integer

MaxPriorityQueue create(*max_size*)	::=	create an empty priority queue.
Boolean isEmpty(q, n)	::=	**if** ($n > 0$) **return** *TRUE* **else return** *FALSE*
Element top(q, n) → return만 해줌	::=	**if** (!isEmpty(q, n)) **return** an instance of the largest element in q **else return** error.
Element pop(q, n)	::=	**if** (!isEmpty(q, n)) **return** an instance of the largest element in q and remove it from the heap **else return** error.
MaxPriorityQueue push(q, *item*, n)	::=	insert *item* into *pq* and return the resulting priority queue.

ADT 5.2: Abstract data type *MaxPriorityQueue*

- Representation of a priority queue
  - Unordered linear list
    - isEmpty() : O(1)
    - push() : O(1)
    - top() : $\Theta(n)$
    - pop() : $\Theta(n)$
  - Max heap
    - isEmpty() : O(1)
    - top() : O(1)
    - push() : O(log n)
    - pop(): O(log n)

5.6.2 Definition of a Max Heap

- Max heap
  - **Complete binary tree** that is also a **max tree**
- Max tree
  - Tree in which the key value in each node is **no smaller** than the key values in its children (if any)

parent's key ≥ children's keys

root:
the largest key

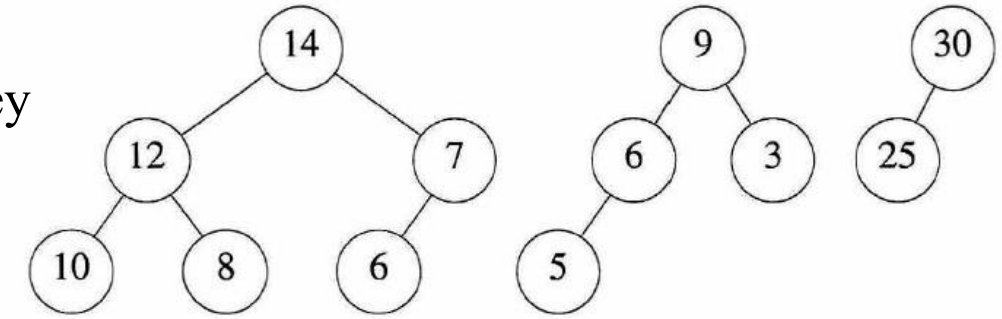


Figure 5.25: Max heaps

- Min heap
  - **Complete binary tree** that is also a **min tree**
- Min tree
  - Tree in which the key value in each node is no larger than the key values in its children (if any)

parent's key $\leq$ children's keys

root:
the **smallest** key

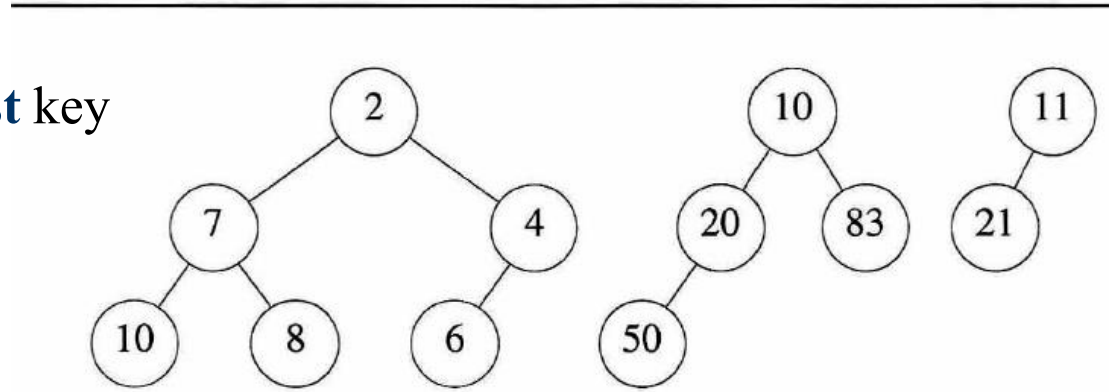


Figure 5.26: Min heaps

5.6.3 Insertion into a Max Heap

Insert: 1
Insert: 5
Insert: 21

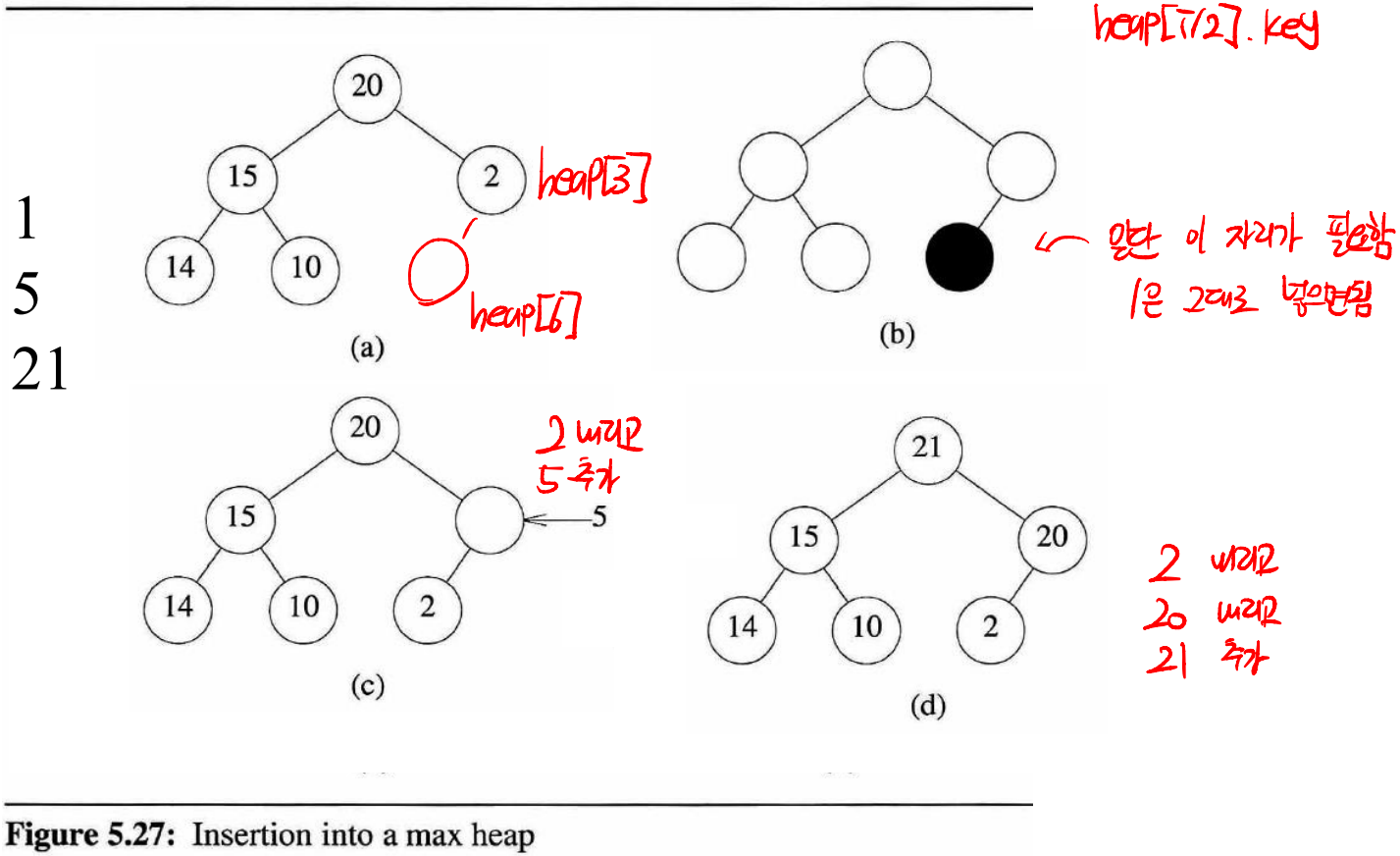


Figure 5.27: Insertion into a max heap

After adding an element, the resulting MUST be max heap:
→ Use a bubbling up process that begins at the new node and moves toward the root.

- Implmementing the insertion into a max heap

```
#define MAX_ELEMENTS 200  /* maximum heap size+1 */
#define HEAP_FULL(n)  (n == MAX_ELEMENTS-1)
#define HEAP_EMPTY(n) (!n)
typedef struct {
        int key;
        /* other fields */
} element;
element heap[MAX_ELEMENTS];
int n = 0;
```

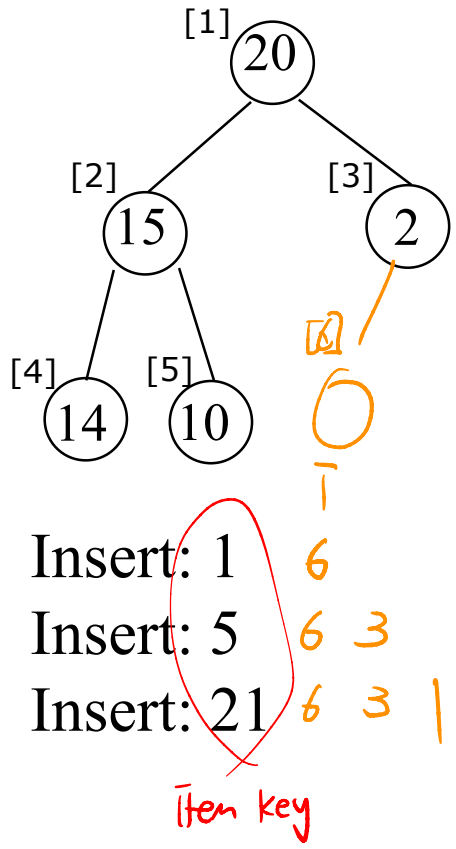
배열 is good
why? complete binary tree 이기 때문

```
void push(element item, int *n)
{/* insert item into a max heap of current size *n */
        int i;
        if ( HEAP_FULL(*n) ){
                fprintf(stderr, "The heap is full. \n");
                exit(EXIT_FAILURE);
        }
        i = ++(*n);
        while ((i != 1) && (item.key > heap[i/2].key)){
                heap[i] = heap[i/2];
                i /= 2;
        }
        heap[i] = item;
}
```

Program 5.13: Insertion into a max heap

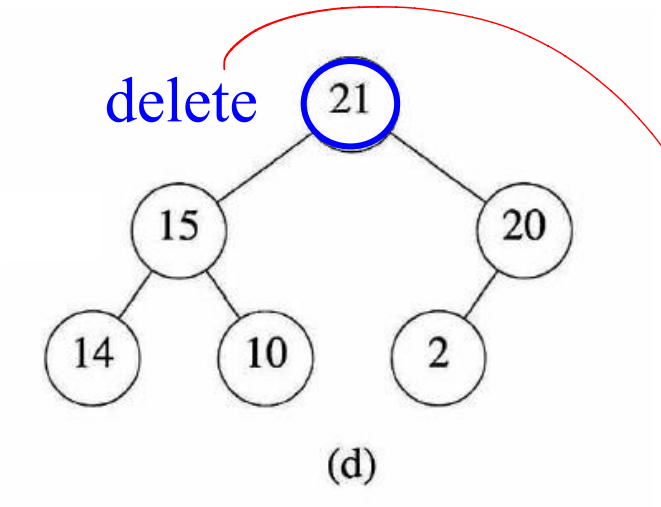
Time complexity:

…



5.6.4 Deletion from a Max Heap

⇒ root 노드의 삭제



마지막 level의 가장 오른쪽 노드

마지막 노드를 root에 옮겨 조정

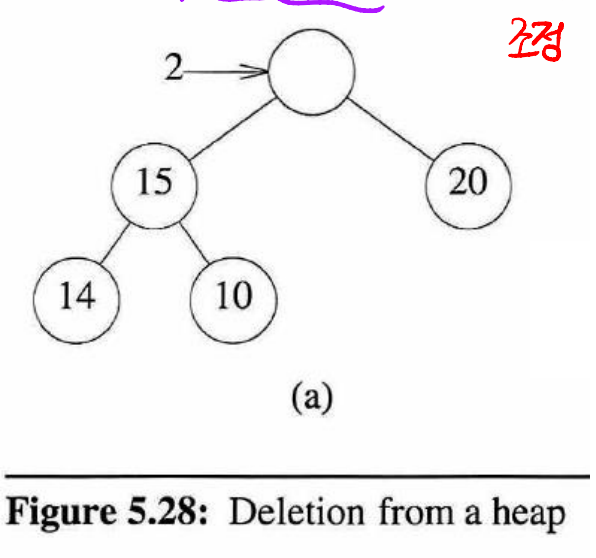
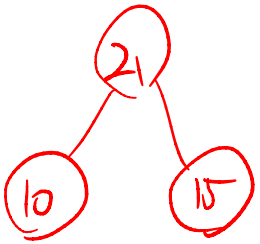
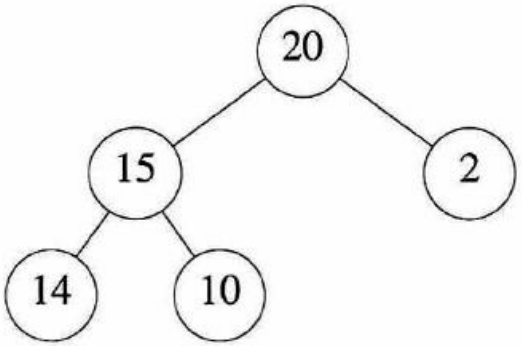


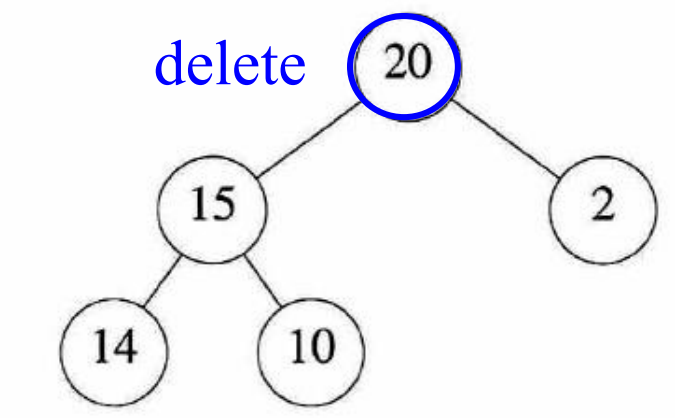
Figure 5.28: Deletion from a heap

완전이진트리 유지하기 위해 삭제

Restructure the max heap with 5 elements : …

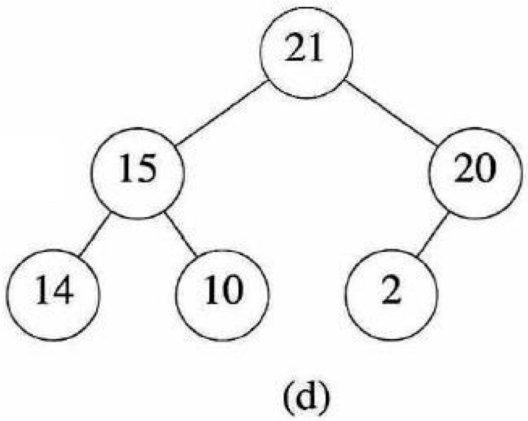






*n = 11
child = 3
Parent = 3
child = 6
Parent = 6
child = 12

```
element pop(int *n)
{
        int parent, child;
        element item, temp;
        if(HEAP_EMPTY(*n)) {
                fprintf(stderr, "The heap is empty\n");
                exit(EXIT_FAILURE);
        }
        item = heap[1];
        temp = heap[(*n)--];
        parent = 1;
        child = 2;
        while(child<= *n){
                if (child< *n) && (heap[child].key < heap[child+1].key)
                        child++;
                if(temp.key >= heap[child].key) break;
                heap[parent] = heap[child];
                parent = child;
                child *= 2;
        }
        heap[parent] = temp;
        return item;
}
```



(d)

Time complexity:

…

Program 5.14: Deletion from a max heap

5.1 Introduction

5.2 Binary Trees

5.3 Binary Trees Traversals

5.4 Additional Binary Tree Operations

5.5 Threaded Binary Trees

5.6 Heaps

5.7 Binary Search Trees

- Dictionary
  - A collection of pairs <key, item>
- Binary Search Tree
  - Easily extended to accommodate dictionaries

ADT *Dictionary* is
objects: a collection of $n > 0$ pairs, each pair has a key and an associated item
functions:
for all $d \in$ *Dictionary*, *item* $\in$ *Item*, $k \in$ *Key*, $n \in$ integer

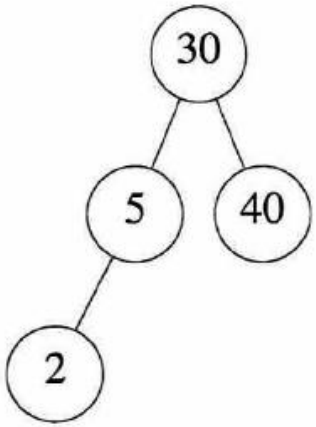
Dictionary Create(*max_size*)	::=	create an empty dictionary.
Boolean IsEmpty(d, n)	::=	**if** ($n > 0$) **return** ~~*TRUE*~~
		else return ~~*FALSE*~~
Element Search(d, k)	::=	**return** item with key k,
		return NULL if no such element.
Element Delete(d, k)	::=	delete and return item (if any) with key k;
void Insert(d, *item*, k)	::=	insert *item* with key k into d.

ADT 5.3: Abstract data type *dictionary*

5.7.1 Definition

- Binary Search Tree
  - A binary tree (may be empty)
  - If not empty :

1) Each node has exactly one key and the keys are distinct
2) Keys in the **left subtree** are **smaller** than the key in the root
3) Keys in the **right subtree** are **larger** than the key in the root
4) Left and right subtrees are also BST



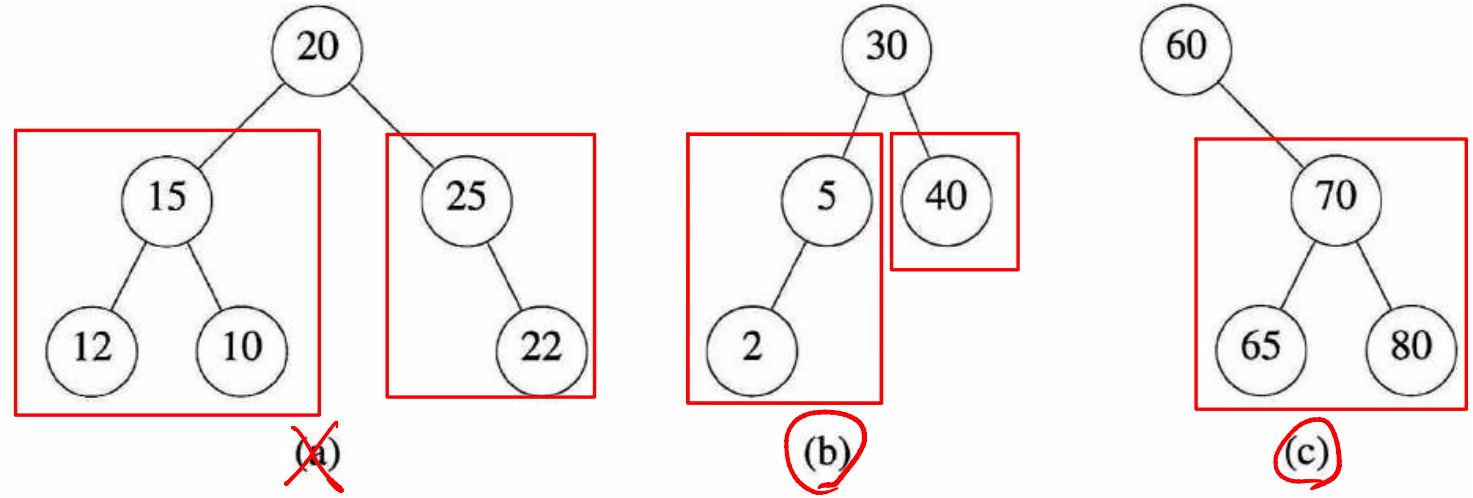


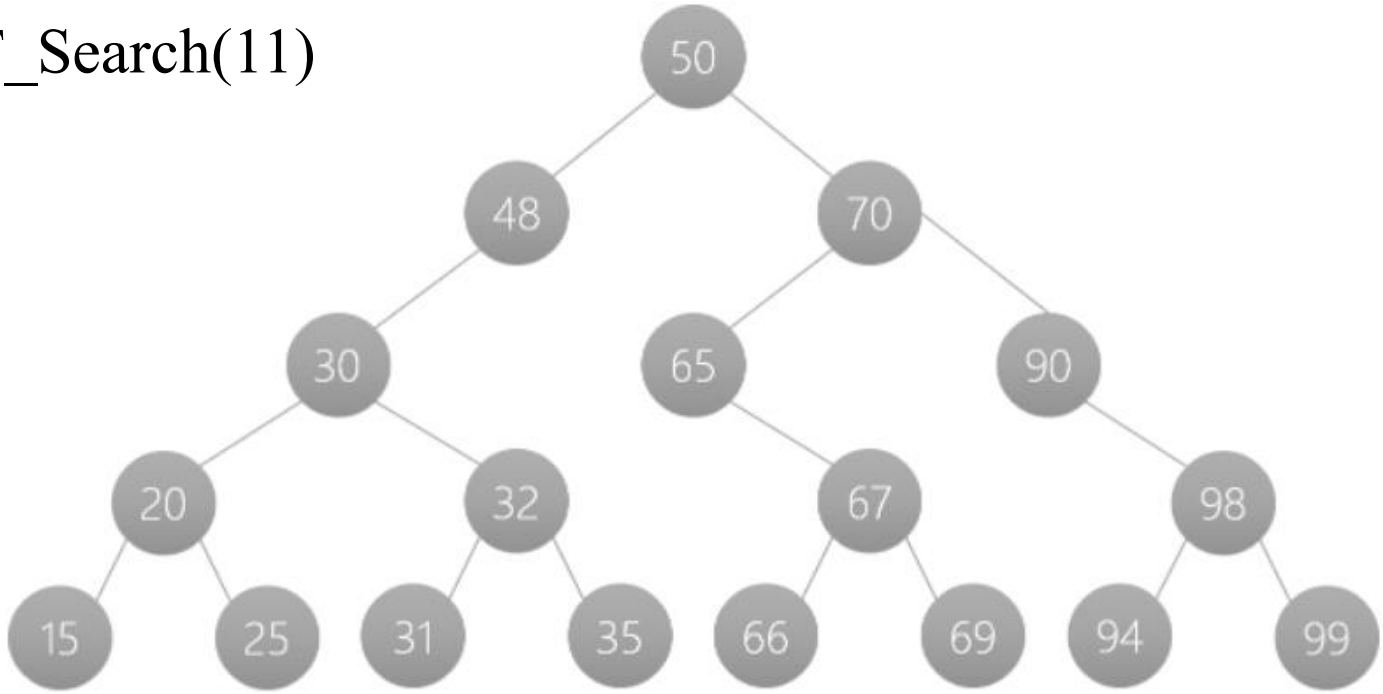
Figure 5.29: Binary trees

BST: ...

BST_Search(25)

BST_Search(67)

BST_Search(11)

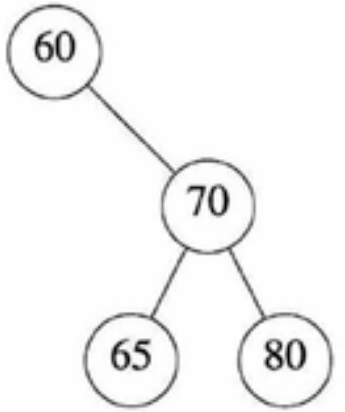


5.7.2 Searching a Binary Search Tree

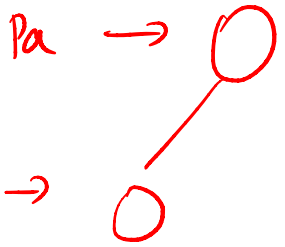
```
element* search(treePointer root, int k)
{/* return a pointer to the element whose key is k, if
                                    there is no such element, return NULL. */
      if (!root) return NULL;
      if (k == root->data.key) return &(root→data);
      if ( k < root->data.key )
            return search(root->leftChild, k);
      return search(root->rightChild, k);
}
```

Program 5.15: Recursive search of a binary search tree

1) k==root: Find
2) k < root: Search the left subtree
3) k > root: Search the right subtree



```
element* iterSearch(treePointer tree, int k)
{/* return a pointer to the element whose key is k, if there is no such element,
                                                    return NULL. */
    while (tree) {
        if (k == tree->data.key) return &(tree->data);
        if (k < tree->data.key)
                tree = tree->left_child;
        else
                tree = tree->right_child;
    }
    return NULL;
}
```



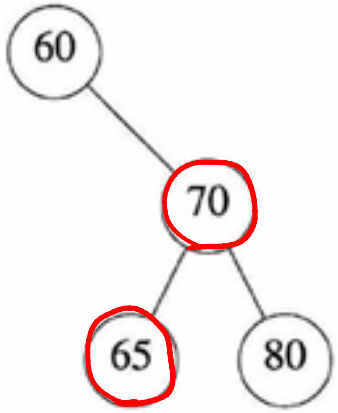
Program 5.16: Iterative search of a binary search tree

Time Complexity (search or iterSearch):

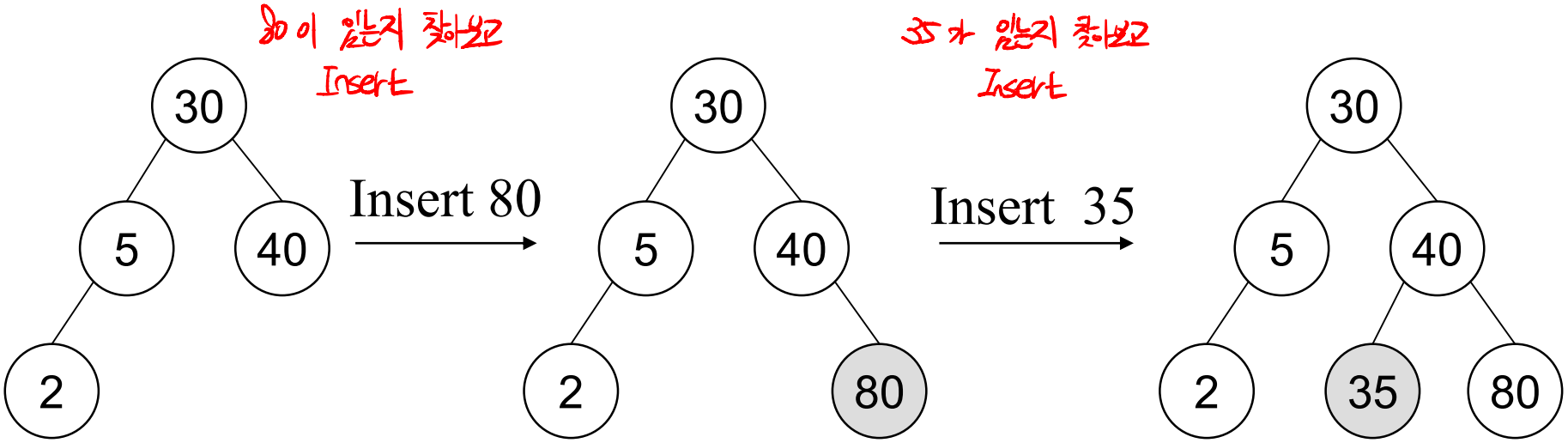
$O(h)$ ⇒ $O(n)$

↳ height

(↳ 완전 이진 트리가 아니기 때문에 skewed 된 경우 $O(n)$ 이됨 (최악의 경우))

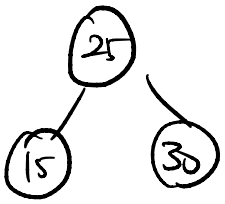


5.7.3 Inserting into a Binary Search Tree



root 노드의 pointer

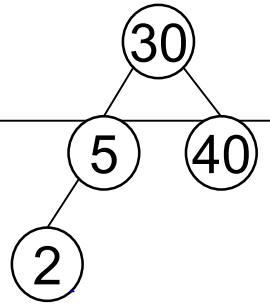
```
void insert (treePointer *node, int k, iType theItem)
{/* if k is in the tree pointed at by node do nothing; otherwise add a new node
                                                    with data = (k, theitem) */
    treePointer  ptr, temp = modifiedSearch(*node, k);
    if( temp || !(*node) ) {
        /* k is not in the tree */
        MALLOC(ptr, sizeof(*ptr));
        ptr->data.key=k;
        ptr->data.item = theItem;
        ptr->leftChild = ptr->rightChild = NULL;
        if(*node) /* insert as child of temp */
            if(k < temp->data.key) temp->leftChild = ptr;
            else temp->rightChild = ptr;
        else *node = ptr;
    }
}
```



if (tree is empty or k is present)
  return **NULL**
else return a **pointer**(to the last node)

Program 5.17: Inserting a dictionary pair into a binary search tree

Time complexity : O(h)



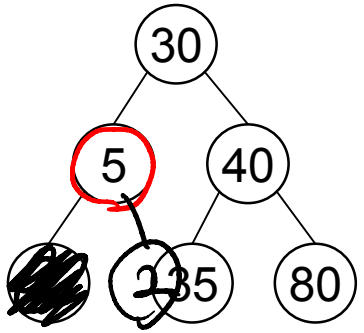
Insert 80

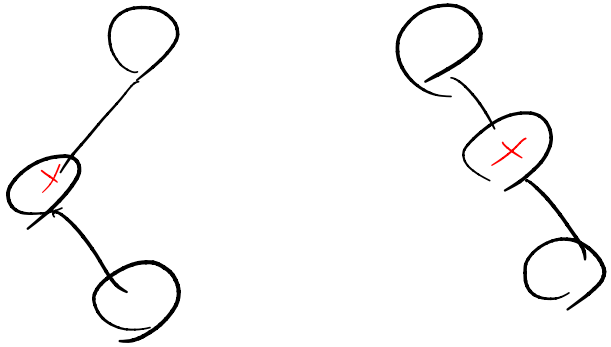
이 페이지 코드 구현해보기

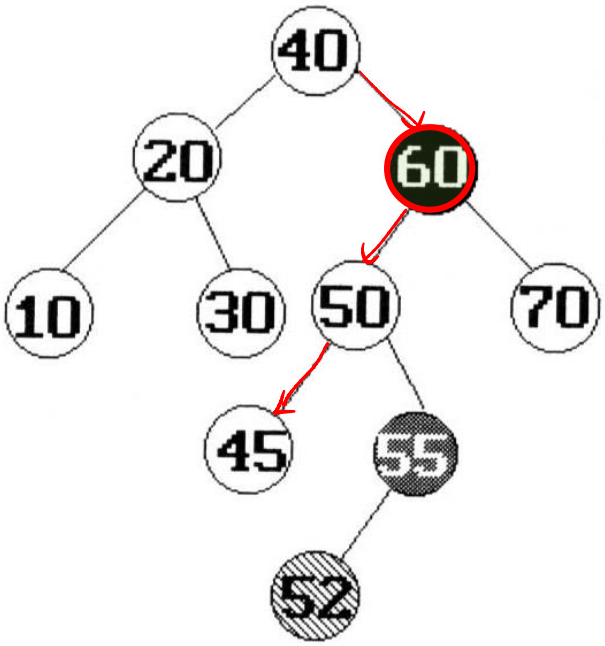
5.7.4 Deletion from a Binary Search Tree

다른 case는 없음... Binary Search tree의 예외

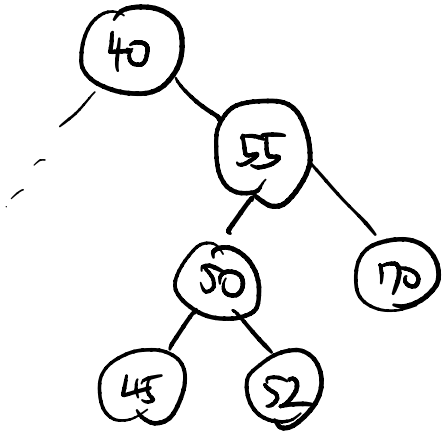
- Deletion
  - case 1. leaf node → 바로 삭제
    - ... 두 child가 모두 NULL이면..
  - case 2. nonleaf node with one child
    - ... child가 하나 있으면 → 삭제하고 자식을 올림
  - case 3. nonleaf node with two child
    - **Replaced by** either the **largest** pair in its left subtree **or** the **smallest** one in its right subtree ex) 30을 삭제하면 5를 넣거나, 40을 넣음
    - Then, proceed to **delete** this replacing pair from the subtree







(a) tree before deletion of 60



(b) tree after deletion of 60

5.7.6 Height of a Binary Search Tree

- Height of a BST with n elements
  - $O(n)$ on the worst case:
    - insert the keys $[1,2, 3, \ldots, n]$, in this order
  - $O(\log_2 n)$ on average
    - insertions and deletions are made at random

- Balanced search trees
  - Worst case height : $O(\log_2 n)$
  - Searching, insertion, deletion is bounded by $O(h)$
  - e.g., AVL tree, 2-3 tree, red-black tree

TREES

5.1 Introduction

5.2 Binary Trees

5.3 Binary Trees Traversals

5.4 Additional Binary Tree Operations

5.5 Threaded Binary Trees

5.6 Heaps

5.7 Binary Search Trees