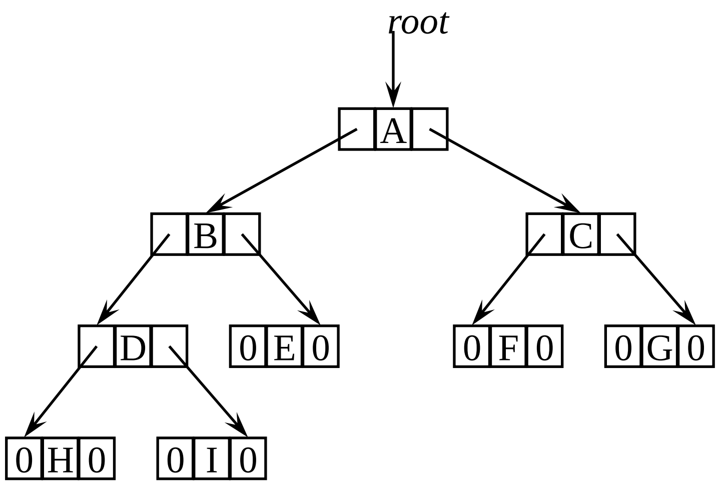
TREES

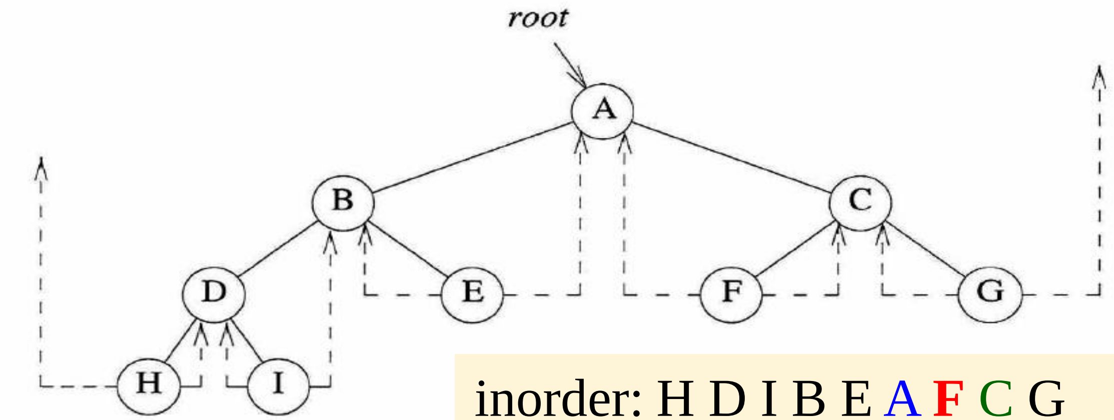
5.5.1 Threads

- Drawback of the BT
  - Too many null pointers;
  - # of Null links: …



- Replace these null pointers with "threads"
  - To make inorder traversal faster and do it without stack and without recursion

- Assume that ***ptr*** represents a node :
  - If ptr -> leftChild == NULL,
    ptr -> leftChild = pointer to the inorder predecessor of ptr
  - If ptr -> rightChild == NULL,
    ptr -> rightChild = pointer to the inorder successor of ptr



inorder: H D I B E A **F** C G

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

5.5.1 Threads

- Node structure:
  - Two additional fields : leftThread and rightThread
    - leftThread, rightThread : true / false
  - ptr→ rightThread = true: …
  - ptr→ rightThread = false: …

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

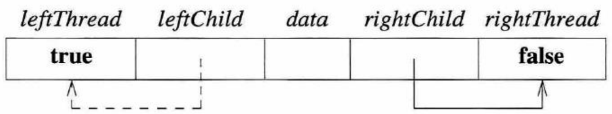


Figure 5.22: An empty threaded binary tree

5.5.1 Threads

- A **header node** for all threaded binary trees:
  - No dangling threads

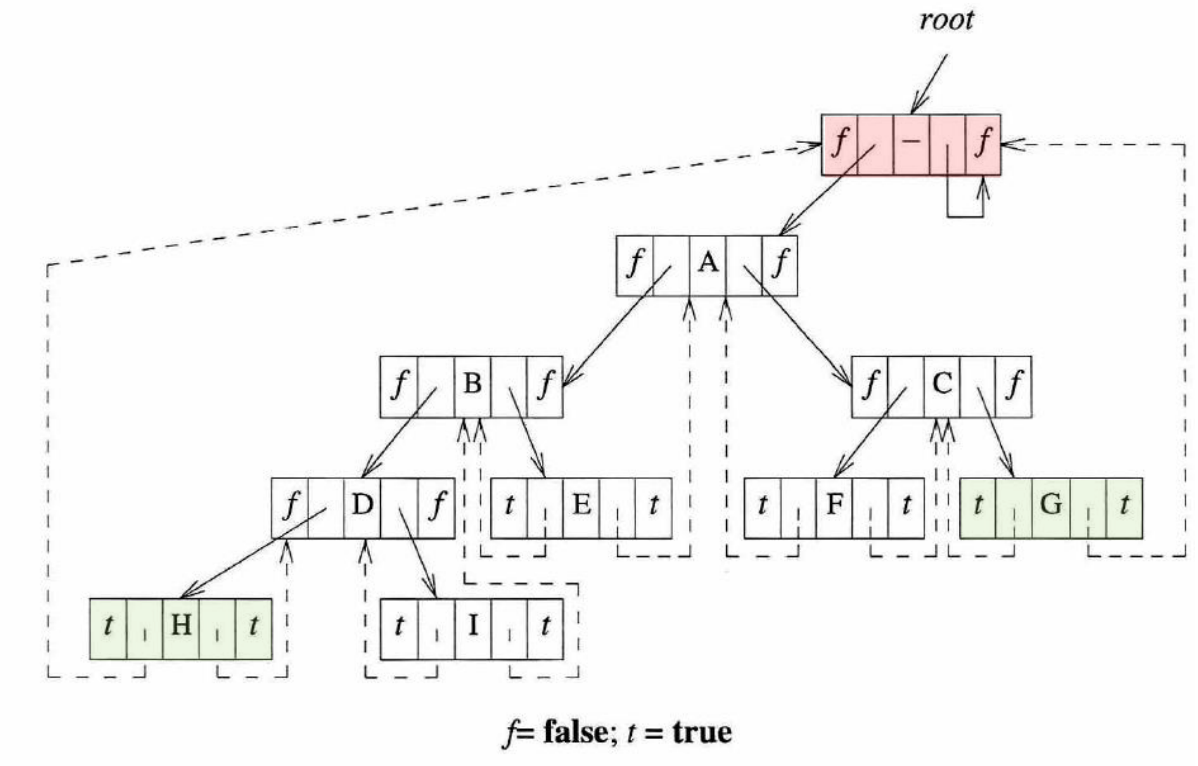


Figure 5.23: Memory representation of threaded tree

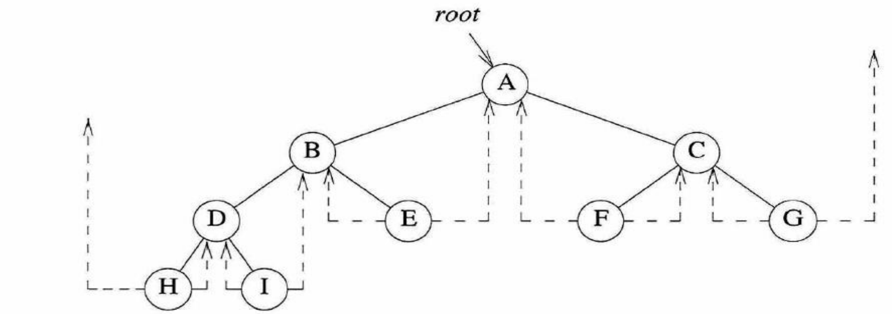
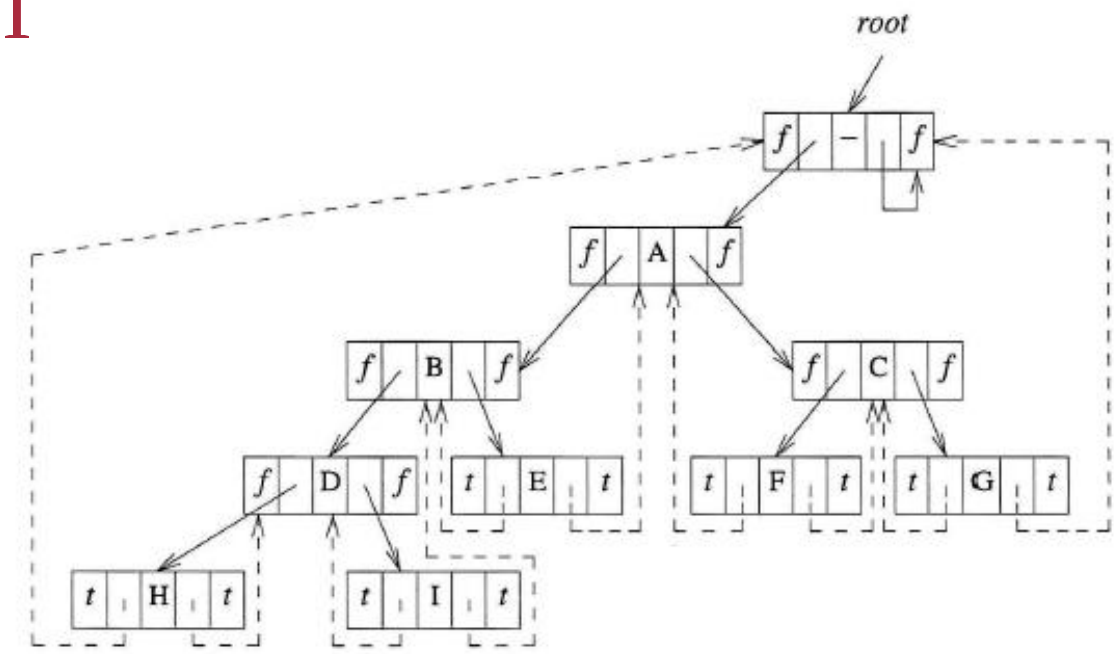


Figure 5.21: Threaded tree corresponding to Figure 5.10(b)

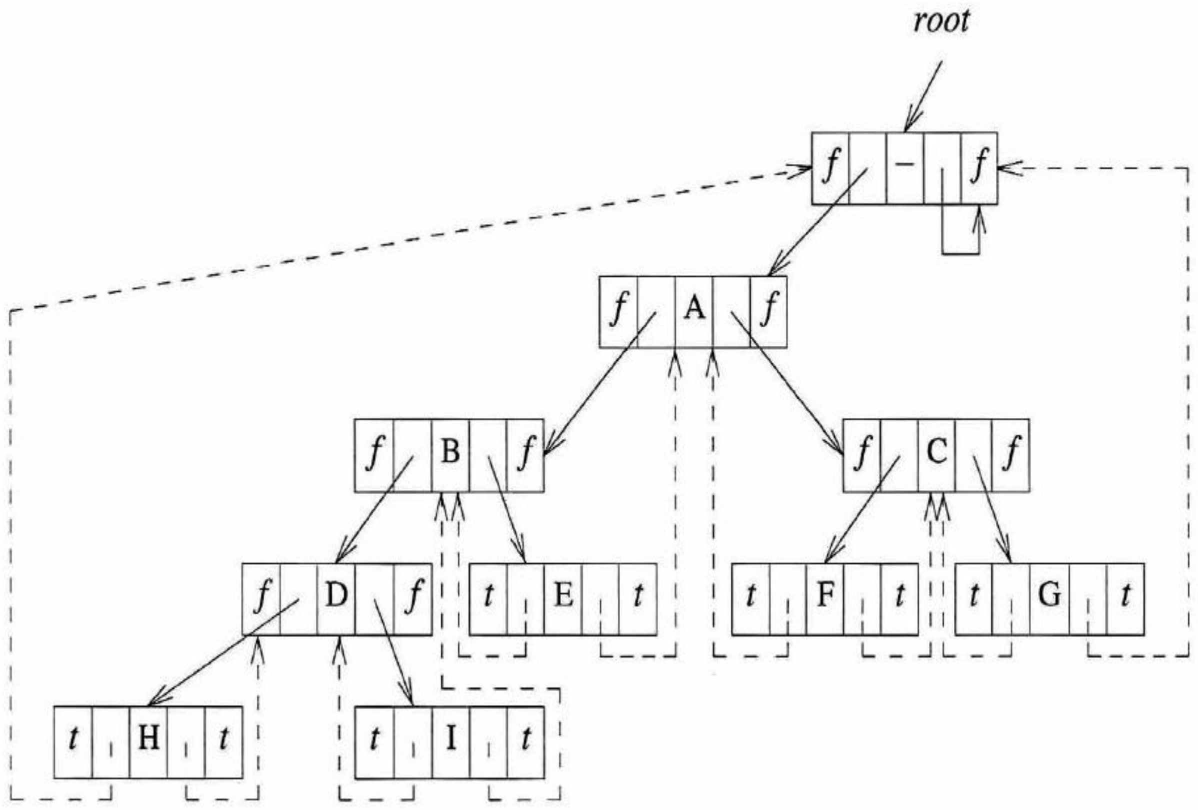
5.5.2 Inorder Traversal of a Threaded BT

- Inorder traversal
  - without making use of a stack



f = false; *t* = true

- The "**next**" node of ptr
  - If ptr->rightThread == true
    → ptr->rightChild
  - If ptr->rightThread == false
    → Follow <u>a path of left-child links from the right-child of *ptr*</u>
    until reaching a node with leftThread = TRUE



f= **false**; *t* = **true**

inorder: H D I B E A F C G

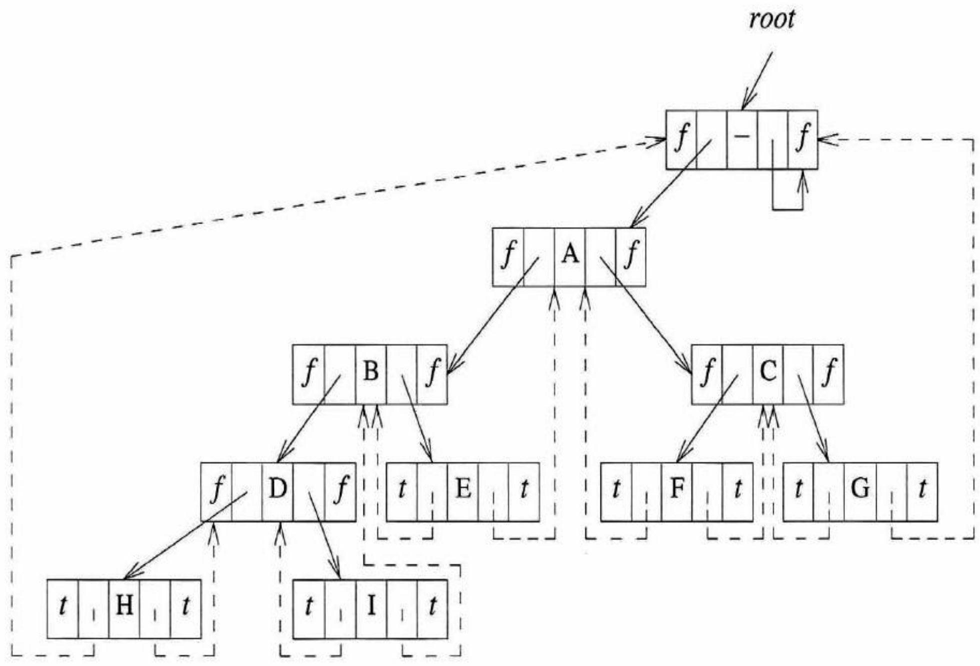
5.5.2 Inorder Traversal of a Threaded BT

- 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

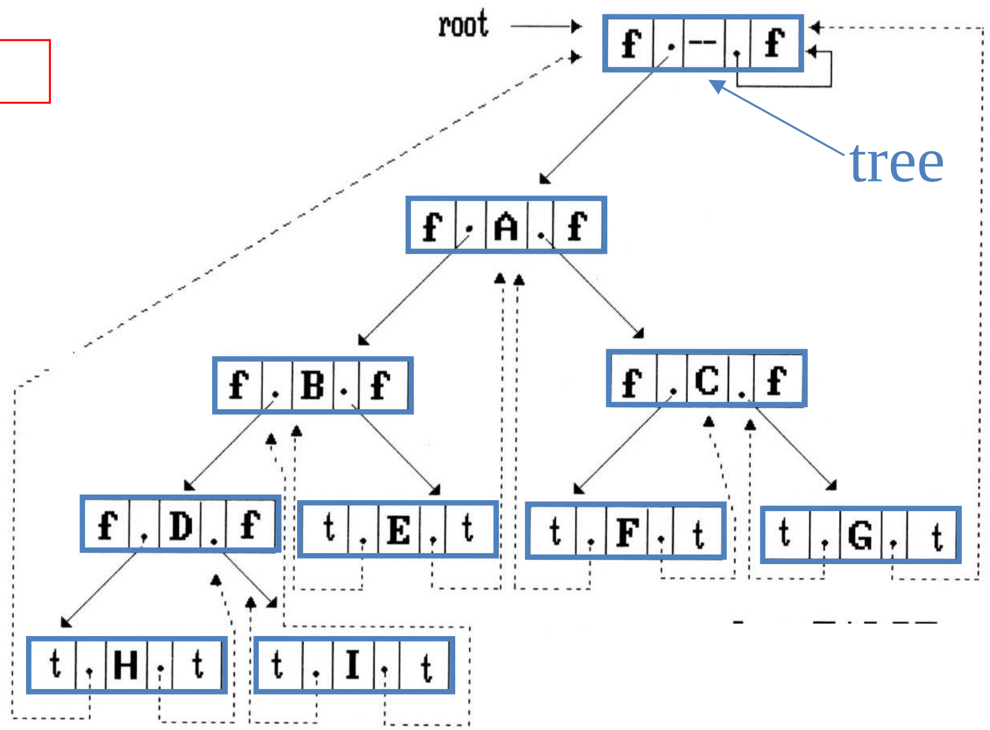


f = false; *t* = true

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



5.6.1 Priority Queues

- Priority queues
  - Collection of elements
  - Each element has a **priority** (or key)
- Two kinds of priority queues
  - Min priority queue
  - Max priority queue
- Heaps
  - A tree with some special properties
  - Frequently used to implement *priority queues*

우선순위 기반 처리

우선순위에 따라 삽입, 삭제 연산의 시간 복잡도가 효율적

힙(Heap) 자료구조를 이용해 구현

활용—작업 스케줄링, 네트워크 패킷 처리, 최대/최소값 찾기 등

ADT *MaxPriorityQueue* is

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

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)	::=	**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*

5.6.1 Priority Queues

- 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 <u>key value in each node</u> is **no smaller** than the key values in its children (if any)

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

root:
the largest key

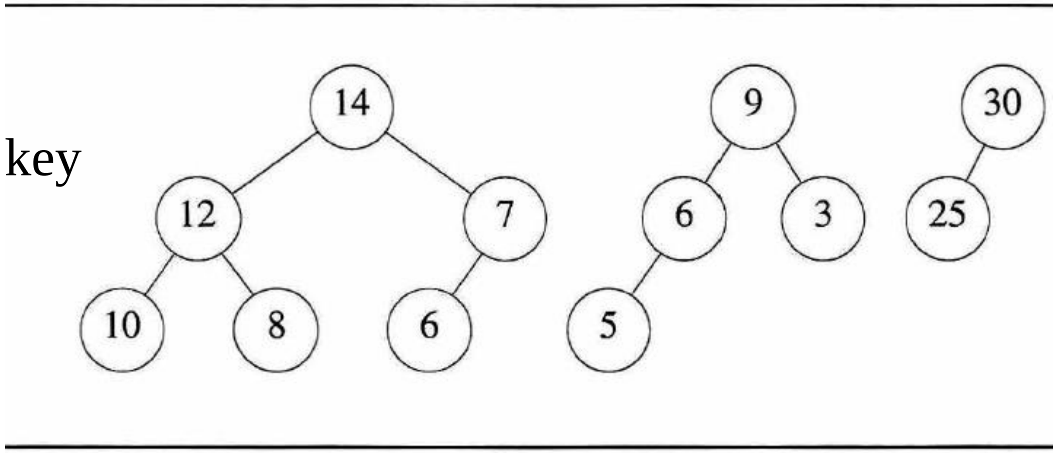


Figure 5.25: Max heaps

- 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

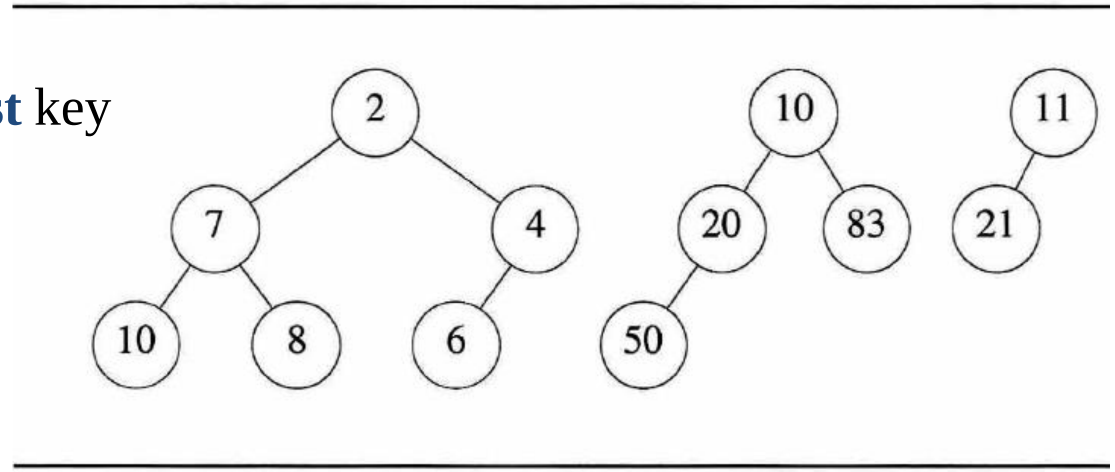
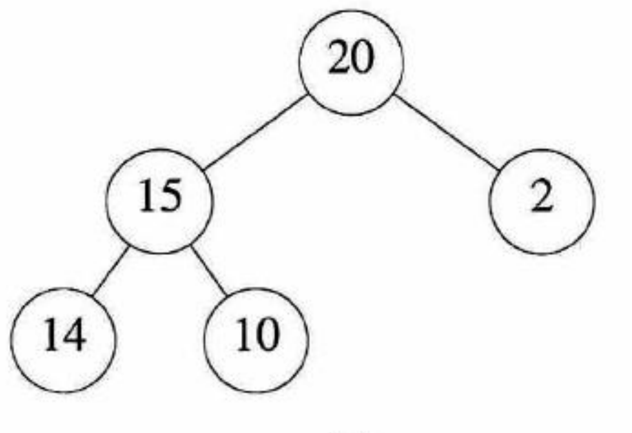


Figure 5.26: Min heaps

5.6.3 Insertion into a Max Heap

- After adding an element, the resulting MUST be a complete BT with max heap
  - Use a bubbling up process;
  - It begins at the new node and moves toward the root

Insert: 1
Insert: 5
Insert: 21



(a)

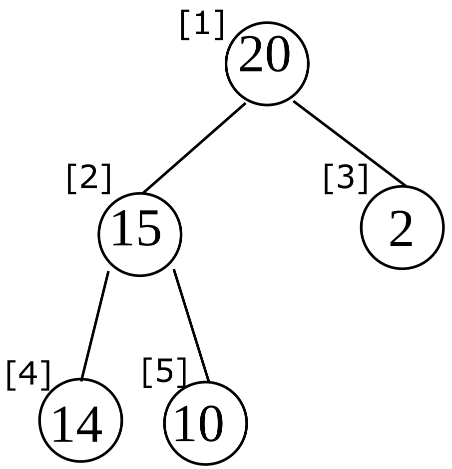
5.6.3 Insertion into a Max Heap

- Implmementing using an array *heap*

```
#define MAX_ELEMENTS 200  /* maximum heap size+l */
#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;
```

```
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;
}
```

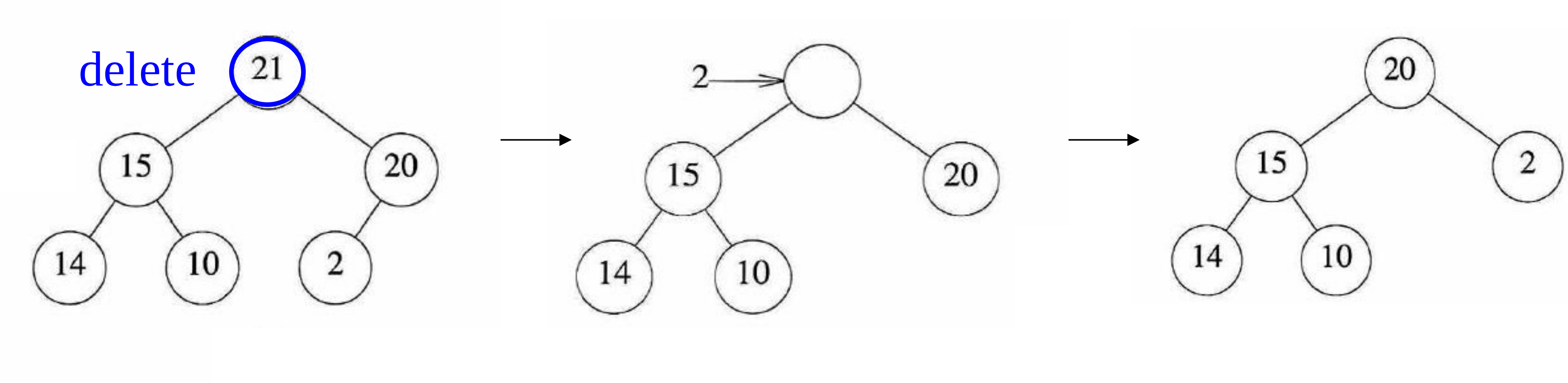
Program 5.13: Insertion into a max heap

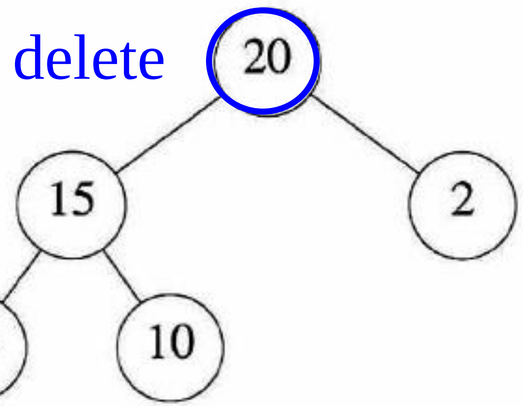


Insert: 1
Insert: 5
Insert: 21

5.6.4 Deletion from a Max Heap

- Root deletion :
  - 1) Replace the root node with the last element
  - 2) Bubbling down
    - Restore the heap order property by repeatedly

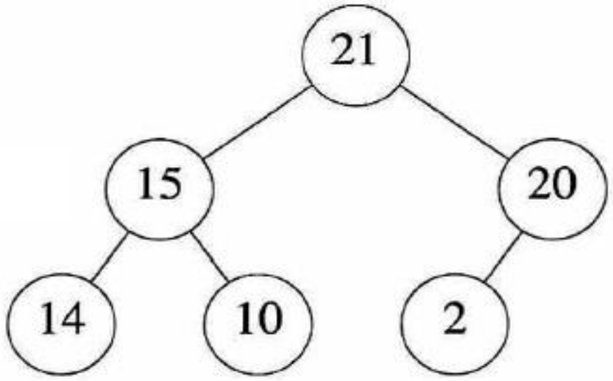





```
          100
        /     \
      90       85
     /  \     /  \
   80    75  70   65
  / \   / \  / \  / \
 60  55 50 45 40 35 30 25
/ \ / \ / \
20 15 10 5 12 8
```

```
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;
}
```

Program 5.14: Deletion from a max heap



(d)

Analysis of push/pop

- The complexity : .

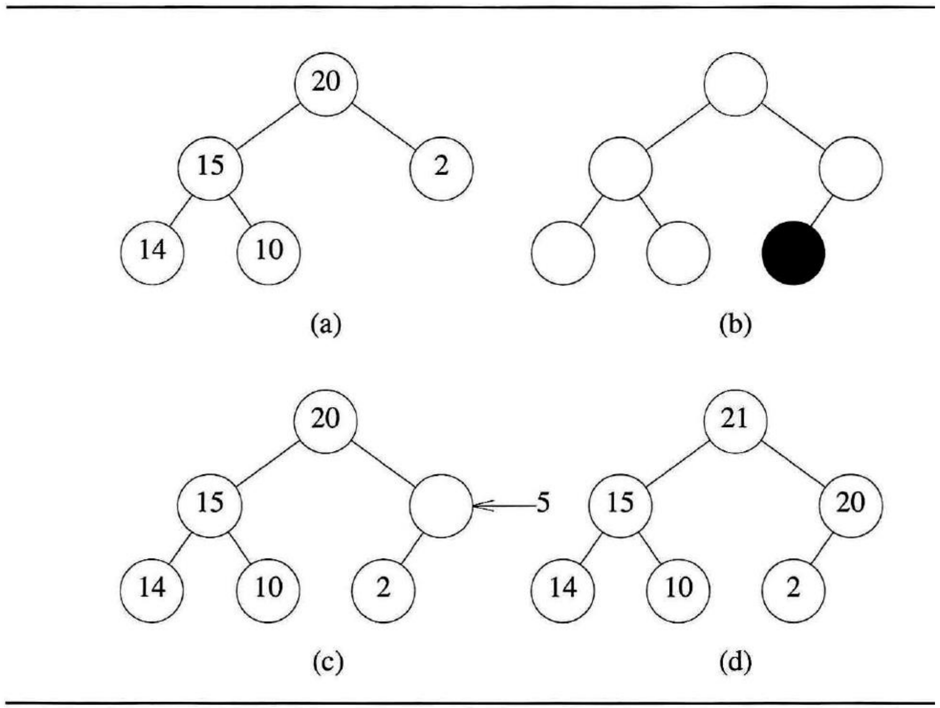


Figure 5.27: Insertion into a max heap

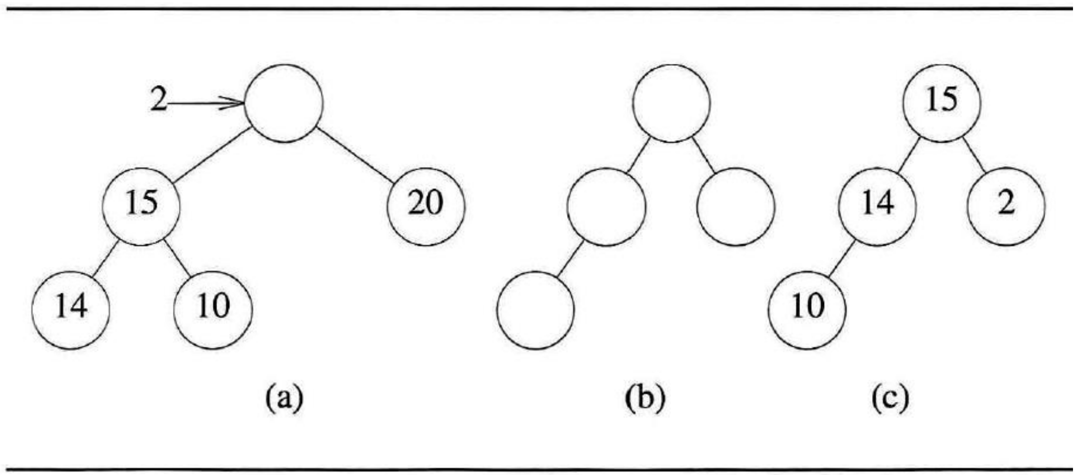


Figure 5.28: Deletion from a heap

TREES

5.7.1 Definition

- A dictionary is a general-purpose data structure for storing a group of objects
  - Dictionary can be implemented using BST

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

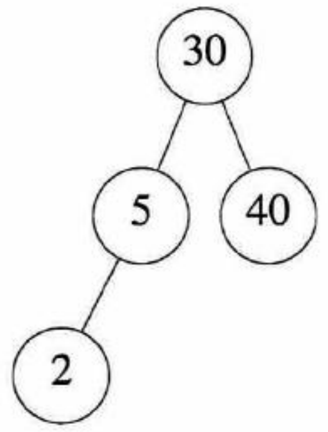
- Binary Search Tree
  - A binary tree (may be empty)
  - If not empty :

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



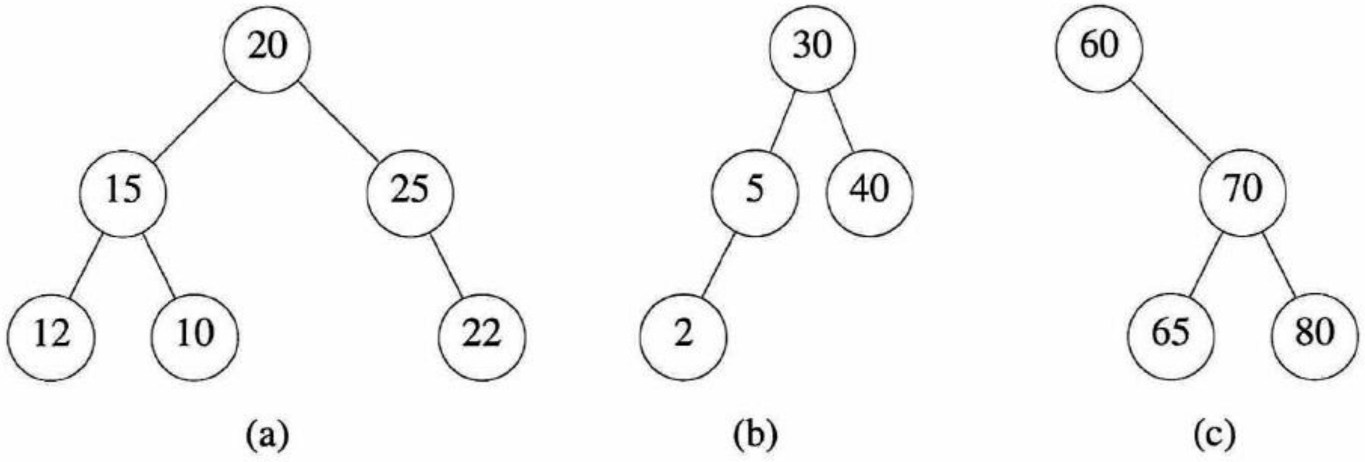


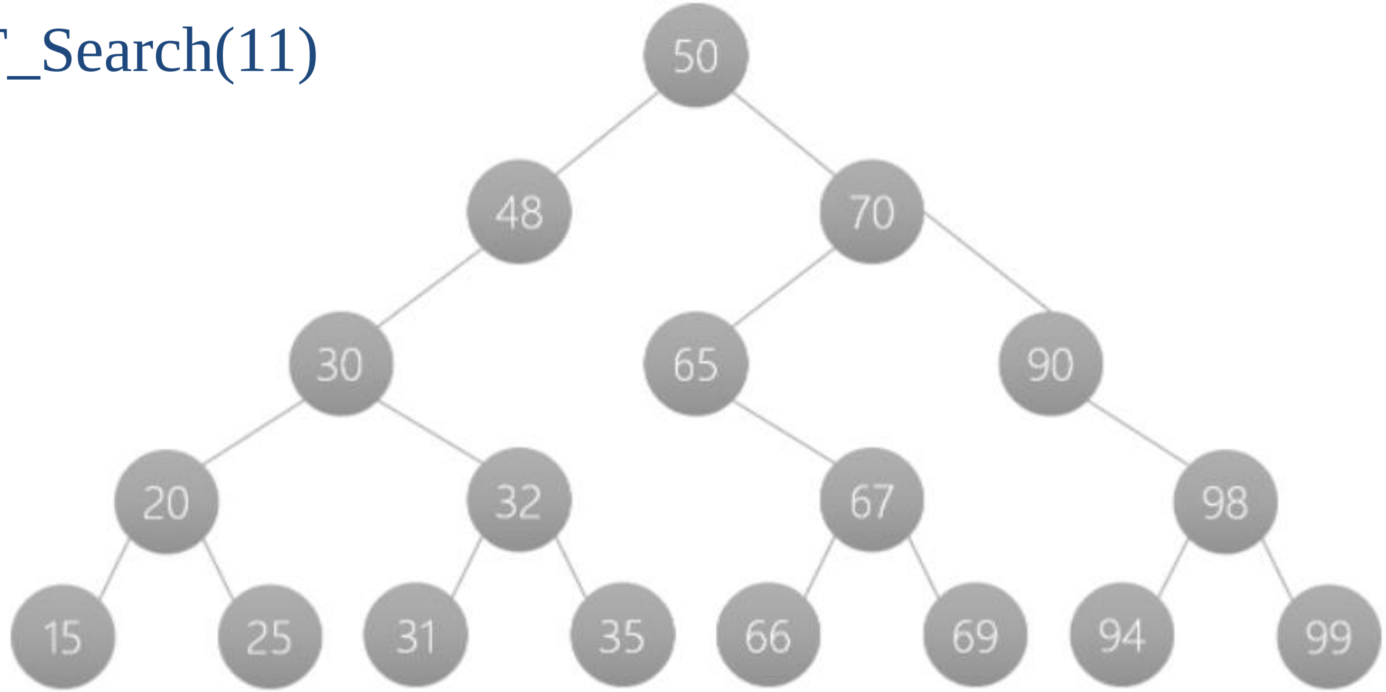
Figure 5.29: Binary trees

BST: ...

BST_Search(25)

BST_Search(67)

BST_Search(11)

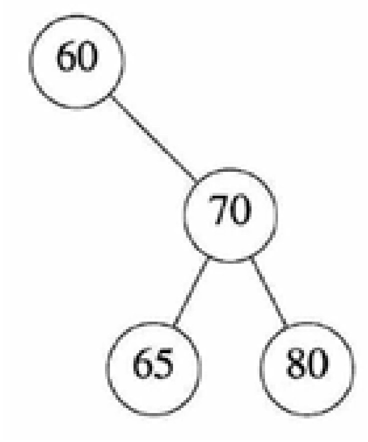


5.7.2 Searching a Binary Search Tree

- Recursive Search
  - k==root: Find
  - k < root: Search the left subtree
  - k > root: Search the right subtree

```
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);
}
```

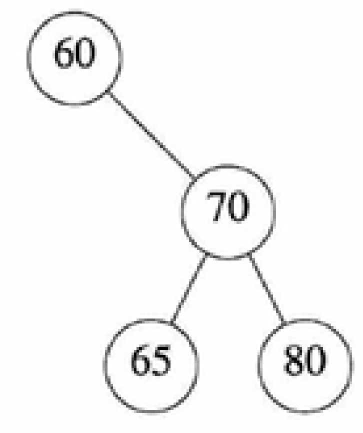
Program 5.15: Recursive search of a binary search tree



5.7.2 Searching a Binary Search Tree

- Iterative search

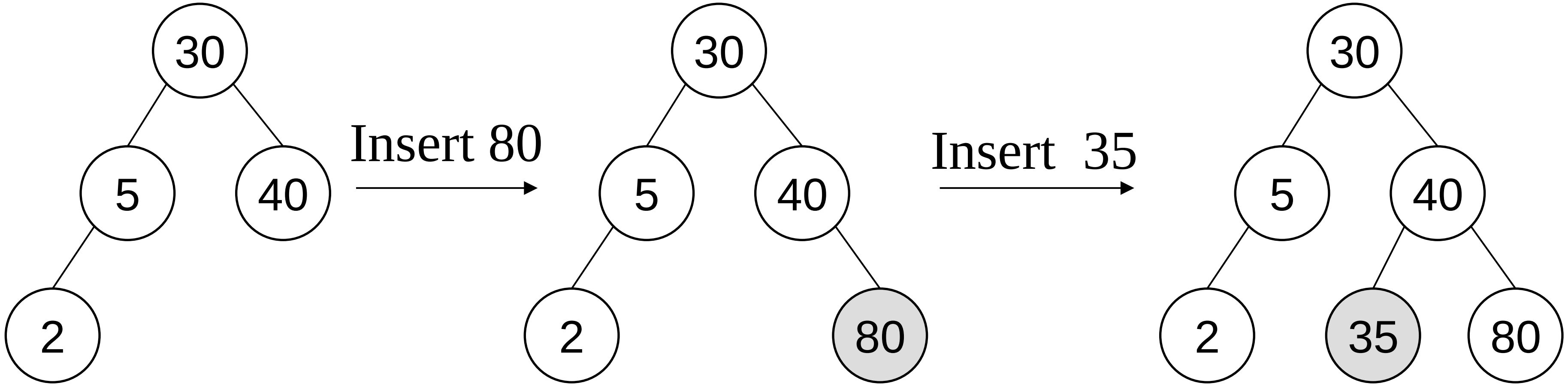
```
element* iterSearch(treePointer tree, int k)
{/* return a pointer to the element whose key is k, if there is no such element,
                                                    return NULL. */
    while (tree) {
        if (k == tree->data.key) return &(tree->data);
        if (k < tree->data.key)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



Program 5.16: Iterative search of a binary search tree

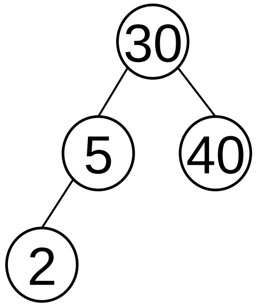
5.7.3 Inserting into a Binary Search Tree

- To insert a dictionary pair (key: k)
  - First verify that the key is different from those of existing pairs
  - To do this, we search the tree
  - If the search is unsuccessful, insert the pair at the point the search terminated



```
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)**



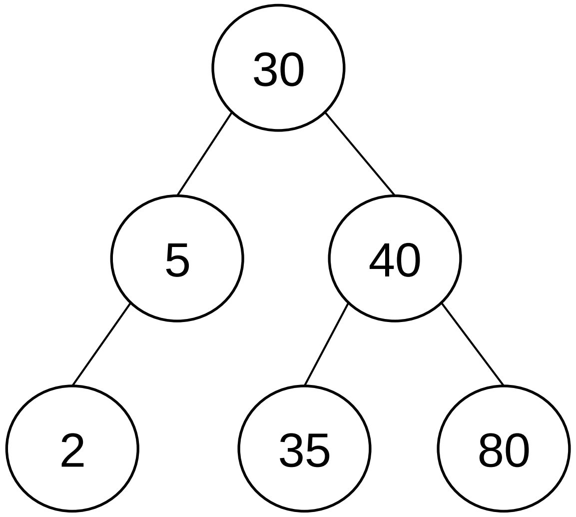
Insert 80

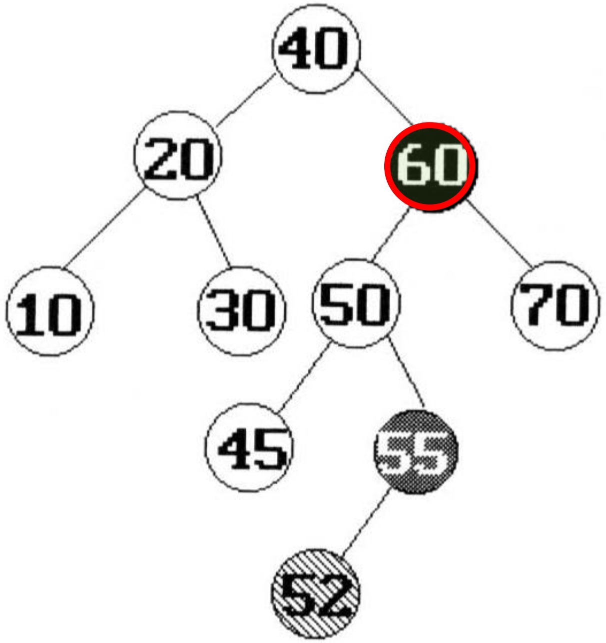
Program 5.17: Inserting a dictionary pair into a binary search tree

Time Complexity : ….

5.7.4 Deletion from a Binary Search Tree

- Deletion
  - case 1. leaf node
    - (1)(2)
  - case 2. nonleaf node with one child
    - (1)(2)
  - 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
    - 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, . . . , 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