Chap 5. Trees

Contents

A ***tree*** is a data structure made up of ***nodes*** or ***vertices*** and ***edges*** without having any ***cycle***. The tree with no nodes is called the **null** or **empty** tree. A tree that is not empty consists of a ***root node*** and potentially many ***levels*** of additional nodes that form a ***hierarchy***.

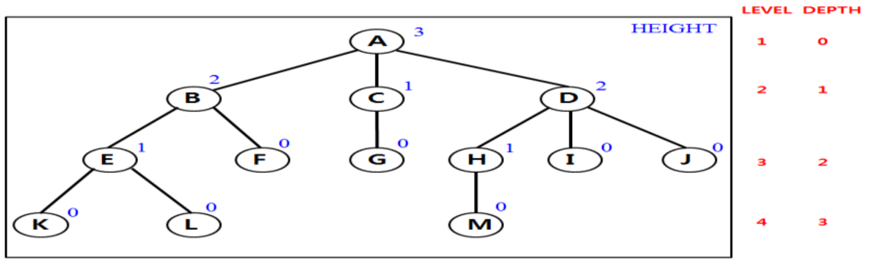


Figure 5.2: A sample tree

- **Root** – The top node in a tree.
- **Child** – A node directly connected to another node when moving away from the root.
- **Parent** – The converse notion of a child. The children of D are H, I, and J; the parent of D is A.
- **Grandparent** – The grandparent of M, which is D.
- **Siblings** – Nodes with the same parent. H, I, and J are siblings.
- **Descendant** – A node reachable by repeated proceeding from parent to child.
- **Ancestor** – A node reachable by repeated proceeding from child to parent. All the nodes along the path from the root to that node. The ancestors of M are A, D, and H.
- **Degree of a node** – Number of subtrees of a node. The degree of A is 3, of C is 1, and of F is zero.
- **Degree of a tree** – The maximum of the degree of the nodes in the tree. The tree of Figure 5.2 has degree 3.
- **Leaf** (or **terminal node**) – A node with no children. I.e., nodes that have degree zero. {K, L, F, G, M, I, J} is the set of leaf nodes.
- **Internal node** (or **nonterminal node**) – A node with at least one child.
- **External node** – A node with no children.
- **Edge** – Connection between one node to another.
- **Path** – A sequence of nodes and edges connecting a node with a descendant.
- **Level** – The level of a node is defined by 1 + (the number of connections between the node and the root).
- **Height of node** – The height of a node is **the number of edges** on the longest path between that node and a leaf.
- **Height of tree** – The height of a tree is **the height of its root node**.
- **Depth** – The depth of a node is the number of edges from the node to the tree's root node.
- **Forest** – A forest is a set of n ≥ 0 disjoint trees.

5.1.2 Representation of Trees

There are several ways to draw a tree besides the one presented in Figure 5.2. One useful way is as a list. The tree of Figure 5.2 could be written as the list

(A (B (E (K, L), F), C (G), D(H (M), I, J)))

The information in the root node comes first, followed by a list of the subtrees of that node. Figure 5.3 shows the resulting memory representation for the tree of Figure 5.2. If we use this representation, we can make use of many of the general functions that we originally wrote for handling lists.

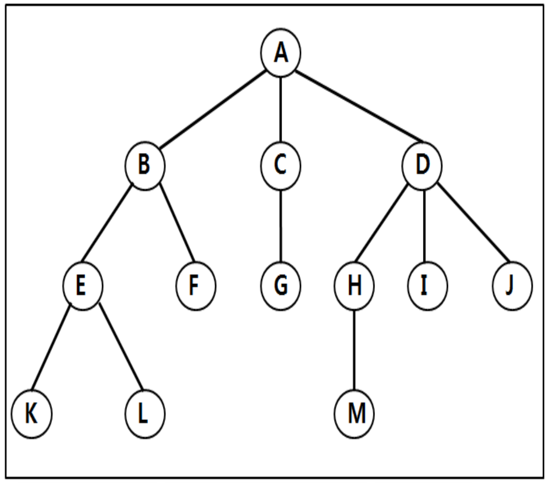


Figure 5.2: A sample tree

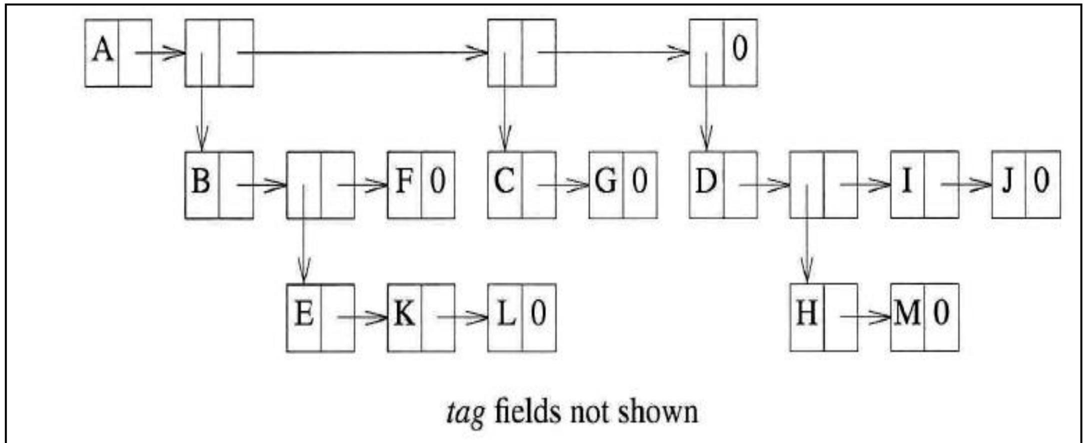


Figure 5.3: List representation of the tree of Figure 5.2

For a tree of degree k, we could use the node structure of Figure 5.4. Each child field is used to point to a subtree. Lemma 5.1 shows that using this node structure is very wasteful of space.

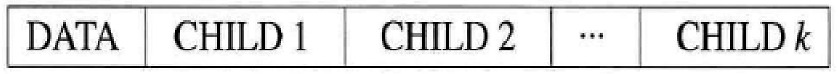


Figure 5.4: Possible node structure for a tree of degree k

Lemma 5.1: If T is **a *k*-ary tree** (i.e., **a tree of degree *k***) with n nodes, each having a fixed size as in Figure 5.4, then $n(k - 1) + 1$ of the nk child fields are 0, $n \geq 1$.

Proof: Since each non-zero child field points to a node and there is exactly one pointer to each node other than the root, the number of non-zero child fields in an n-node tree is exactly $n - 1$. The total number of child fields in a k-ary tree with n nodes is nk. Hence, the number of zero fields is $nk - (n- 1) = n(k - 1) + 1$. □

We shall develop two specialized **fixed-node-size representations** for trees. Both of these require exactly two link, or pointer, fields per node.

5.1.2.2 Left Child-Right Sibling Representation

Data	
Left Child	Right Sibling

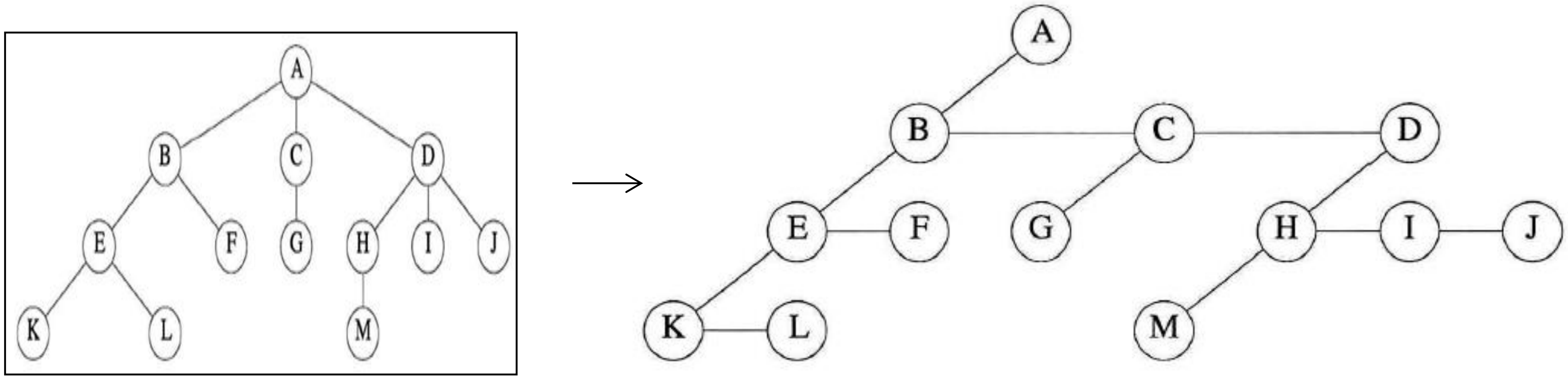
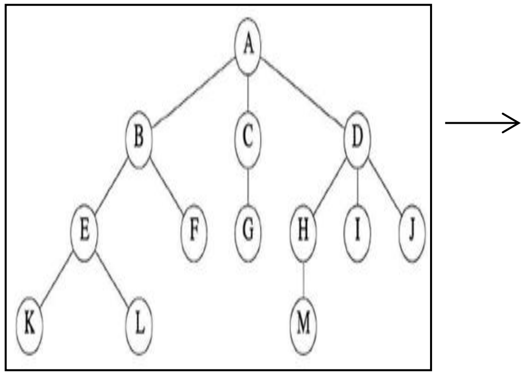


Figure 5.6: Left child-right sibling representation of tree of Figure 5.2

5.1.2.3 Representation as a Degree Two Trees

To obtain the degree-two tree representation of a tree, we simply rotate the right-sibling pointers in **a left child-right sibling tree** clockwise by 45 degrees. This gives us the degree-two tree displayed in Figure 5.7. In the degree-two representation, we refer to the two children of a node as the left and right children. Notice that the right child of the root node of the tree is empty. This is always the case since the root of the tree we are transforming can never have a sibling. Figure 5.8 shows two additional examples of trees represented as left child-right sibling trees and as left child-right child (or degree two) trees. **Left child-right child trees** are also known as ***binary trees***.

Data	
Left Child	Right Child



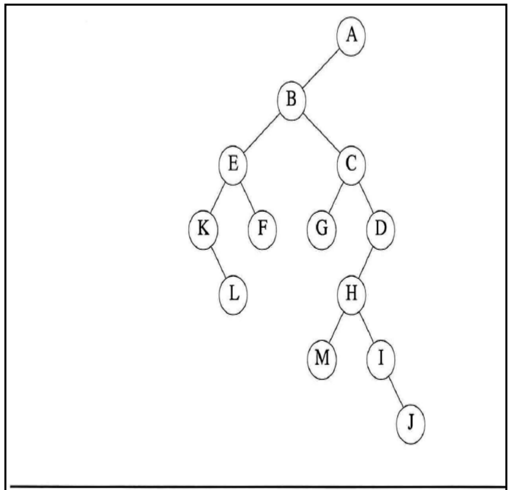


Figure 5.7: Left child-right child tree representation of tree of Figure 5.2

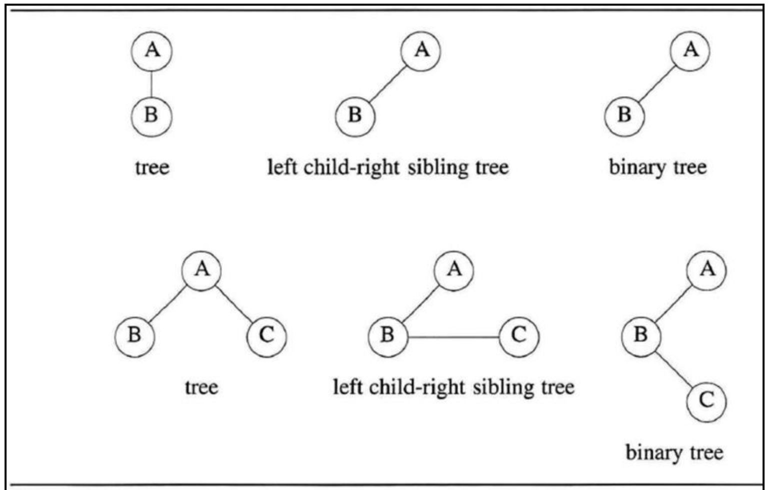


Figure 5.8: Tree representations

5.2 Binary Trees

Definition : A ***binary tree*** is a finite set of nodes that is either ***empty*** or consists of a ***root*** and two disjoint binary trees called the ***left subtree*** and the ***right subtree***. □

The terms that we introduced for trees such as degree, level, height, leaf, parent, and child all apply to binary trees in the natural way.

In a binary tree we distinguish between **the order of the children** while in a tree we do not.

The two binary trees of Figure 5.9 are different.

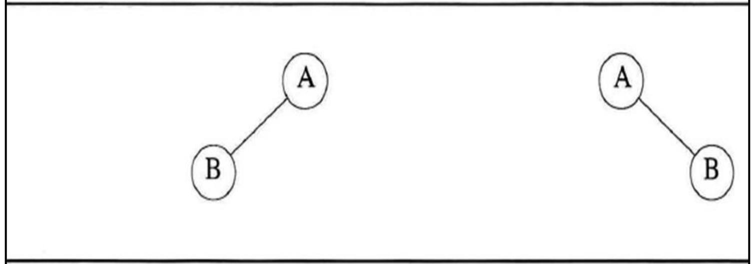
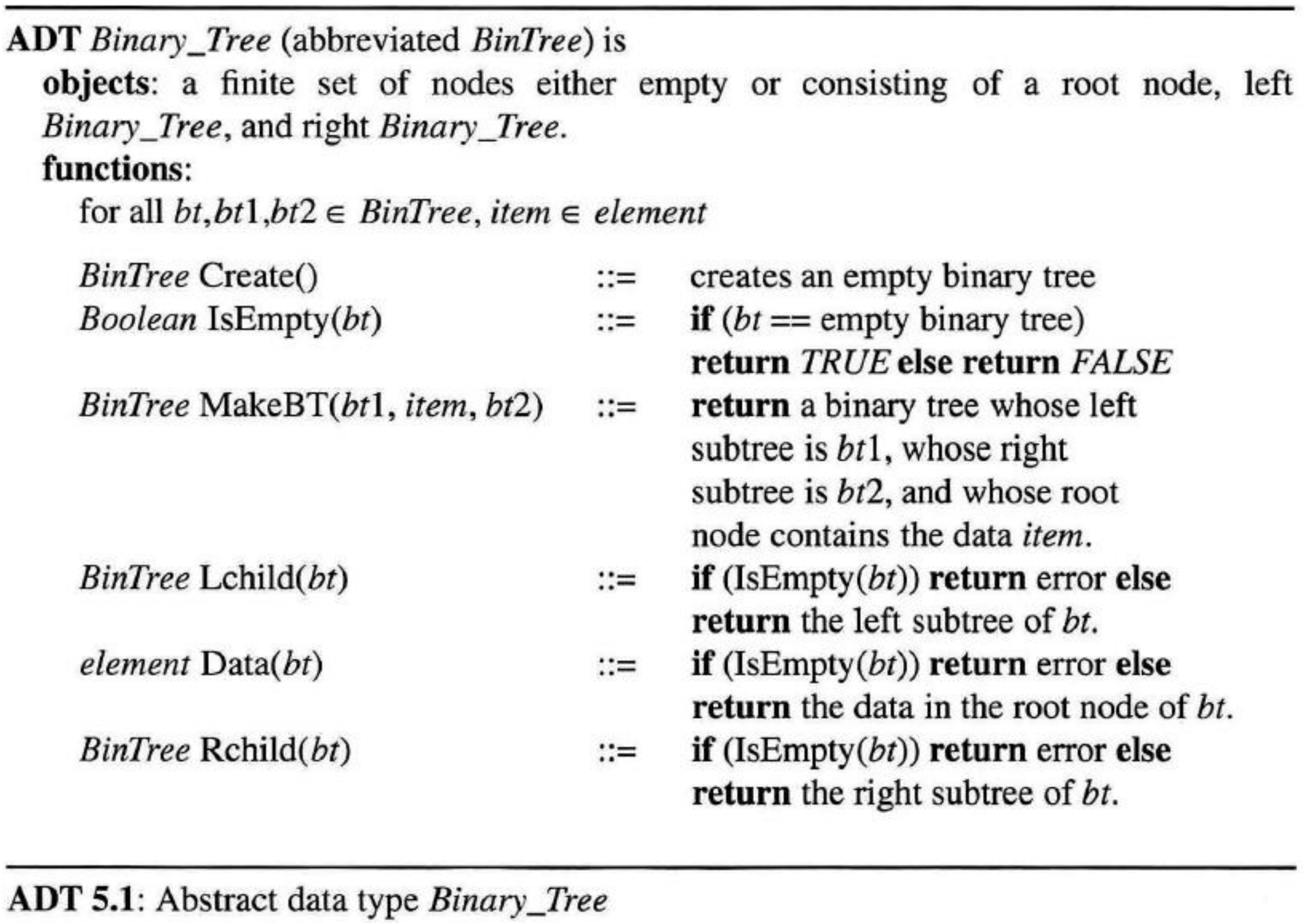


Figure 5.9: Two different binary trees

5.2.1 The Abstract Data Type



ADT *Binary_Tree* (abbreviated *BinTree*) is

objects: a finite set of nodes either empty or consisting of a root node, left *Binary_Tree*, and right *Binary_Tree*.

functions:

for all $bt, bt1, bt2 \in BinTree$, $item \in element$

BinTree Create()	::=	creates an empty binary tree
Boolean IsEmpty(*bt*)	::=	**if** (*bt* == empty binary tree) **return** *TRUE* **else return** *FALSE*
BinTree MakeBT(*bt*1, *item*, *bt*2)	::=	**return** a binary tree whose left subtree is *bt*1, whose right subtree is *bt*2, and whose root node contains the data *item*.
BinTree Lchild(*bt*)	::=	**if** (IsEmpty(*bt*)) **return** error **else return** the left subtree of *bt*.
element Data(*bt*)	::=	**if** (IsEmpty(*bt*)) **return** error **else return** the data in the root node of *bt*.
BinTree Rchild(*bt*)	::=	**if** (IsEmpty(*bt*)) **return** error **else return** the right subtree of *bt*.

ADT 5.1: Abstract data type *Binary_Tree*

Definition : A ***full binary tree*** of depth k is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 0$. □

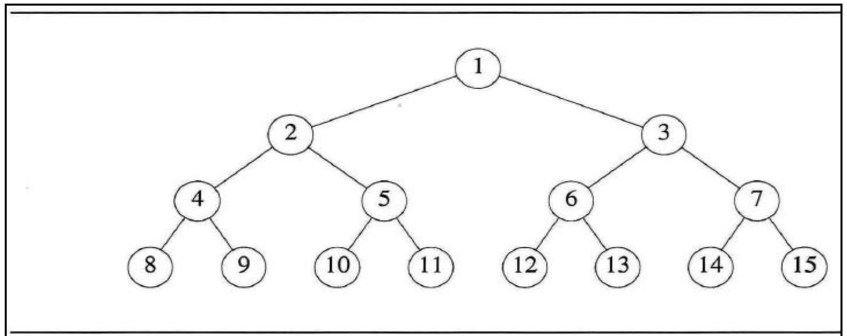


Figure 5.11: Full binary tree of depth 4 with sequential node numbers

Definition [*Complete Binary Tree*]: A binary tree with n nodes and depth k is ***complete*** iff its nodes correspond to the nodes numbered from 1 to n in the full binary tree of depth k. □

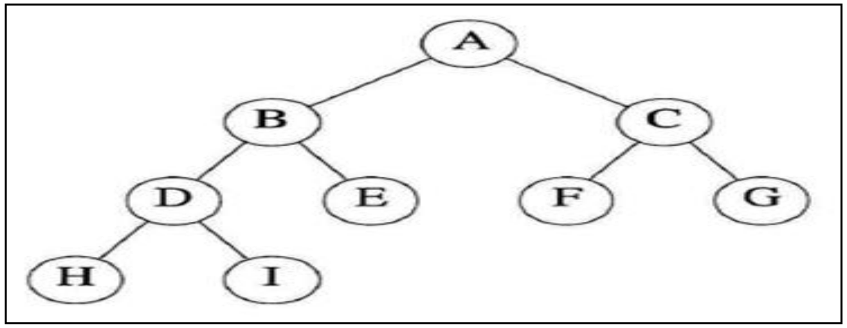
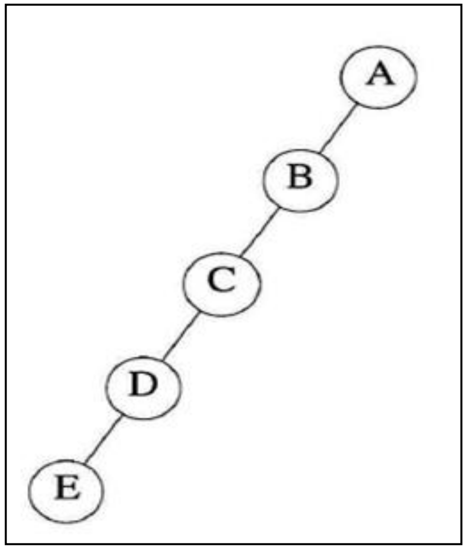


Figure . A complete binary tree



A left *skewed* tree.

A right *skewed* tree.

5.2.2 Properties of Binary Trees

Lemma 5.2 [*Maximum number of nodes*]:

(1) The maximum number of nodes on **level** i of a binary tree is 2^{i-1}, $i \geq 1$.

(2) The maximum number of nodes in a binary tree of **depth** k is $2^k - 1$, $k \geq 1$

Proof:

(1) The proof is by induction on i.

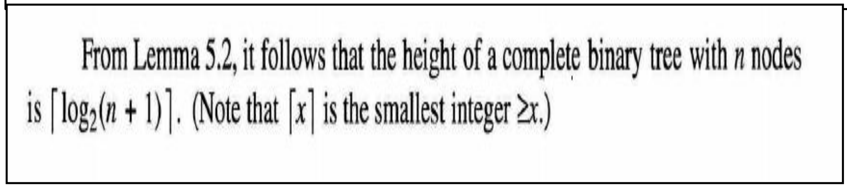
Induction Base: The root is the only node on level i = 1. Hence, the maximum number of nodes on level $i = 1$ is $2^{i-1} = 2^0 = 1$.

Induction Hypothesis: Let i be an arbitrary positive integer greater than 1. Assume that the maximum number of nodes on level $i - 1$ is 2^{i-2} .

Induction Step: The maximum number of nodes on level $i - 1$ is 2^{i-2} by the induction hypothesis. Since each node in a binary tree has a maximum degree of 2, the maximum number of nodes on level i is two times the maximum number of nodes on level i-1, or 2^{i-1}.

(2) The maximum number of nodes in a binary tree of depth k is

$$\sum_{i=1}^{k} (\text{maximum number of nodes on level } i) = \sum_{i=1}^{k} 2^{i-1} = 2^k - 1 \qquad \square$$



From Lemma 5.2, it follows that the height of a complete binary tree with n nodes is $\lceil \log_2(n+1) \rceil$. (Note that $\lceil x \rceil$ is the smallest integer $\geq x$.)

Lemma 5.3 [***Relation between number of leaf nodes and degree-2 nodes***]: For any nonempty binary tree *T*, if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0 = n_2+1$.

Proof: Let n_1 be the number of nodes of degree one and *n* the total number of nodes. Since all nodes in *T* are at most of degree two, we have

$$n = n_0 + n_1 + n_2 \qquad (5.1)$$

If we count the number of branches in a binary tree, we see that every node except the root has a branch leading into it. If *B* is the number of branches, then $n = B + 1$. All branches stem from a node of degree one or two. Thus, $B = n_1 + 2n_2$. Hence, we obtain

$$n = B + 1 = n_1 + 2n_2 + 1 \qquad (5.2)$$

Subtracting Eq. (5 .2) from Eq. (5.1) and rearranging terms, we get

$$n_0 = n_2 + 1 \qquad \square$$

In Figure 5.10(a), $n_0 = 1$ and $n_2 = 0$; in Figure 5.10(b), $n_0 = 5$ and $n_2 = 4$.

5.2.3 Binary Tree Representation

5.2.3.1 Array Representation

Lemma 5.4 : If a **complete binary tree** with n nodes is represented **sequentially**, then *for any node with index i*, $1 \le i \le n$, we have:

(1) *parent(i)* is at $\lfloor i/2 \rfloor$ if $i \neq 1$. If $i = 1$, i is at the root and has no parent.

(2) *leftChild(i)* is at $2i$ if $2i \le n$. If $2i > n$, then i has no left child.

(3) *rightChild(i)* is at $2i + 1$ if $2i + 1 \le n$. If $2i + 1 > n$, then i has no right child.

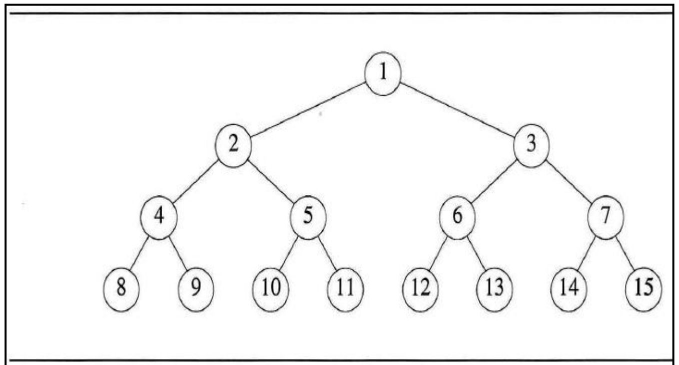
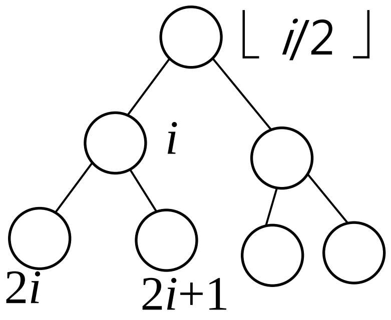
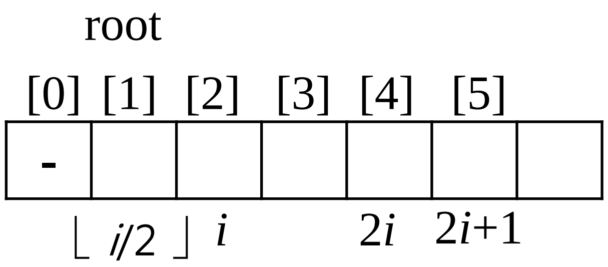


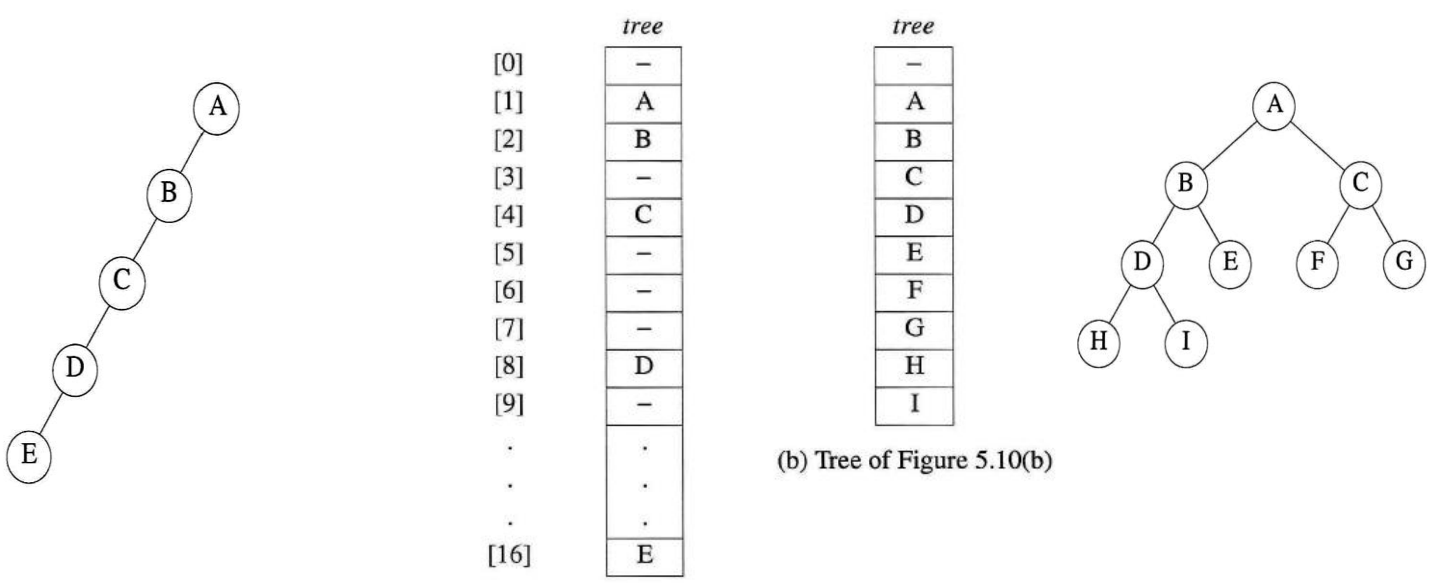
Figure 5.11: Full binary tree of depth 4 with sequential node numbers





Proof: We prove (2). (3) is an immediate consequence of (2) and the numbering of nodes on the same level from left to right. (1) follows from (2) and (3). We prove (2) by induction on i. For i = 1, clearly the left child is at 2 unless $2 > n$, in which case i has no left child. Now assume that for all j, $1 \le j \le i$, *leftChild* (j) is at $2j$. Then the two nodes immediately preceding *leftChild* ($i + 1$) are the right and left children of i. The left child is at $2i$. Hence, the left child of $i + 1$ is at $2i + 2 = 2(i + 1)$ unless $2(i + 1) > n$, in which case $i + 1$ has no left child. □

This representation can clearly be used for all binary trees, though in most cases there will be a lot of unutilized space. Figure 5.12 shows the array representation for both trees of Figure 5.10. For **complete binary trees** such as the one in Figure 5.10(b), the representation is ideal, as no space is wasted. For **the skewed tree** of Figure 5.10(a), however, less than half the array is utilized. In the worst case a skewed tree of depth *k* will require 2^k - l spaces. Of these, only *k* will be used.



(a) Tree of Figure 5.10(a)

(b) Tree of Figure 5.10(b)

Figure 5.12: Array representation of the binary trees of Figure 5.10

배열을 이용한 binary tree

(1) 구조체 정의

```
typedef struct {
            int id;
            char name[20];
            char address[100];
} element;
```

(2) Creation of a tree space

```
element btree[MAX_SIZE];
```

(3) Creation of a new empty list

```
int first = -1;
```

(4) Test for an empty list

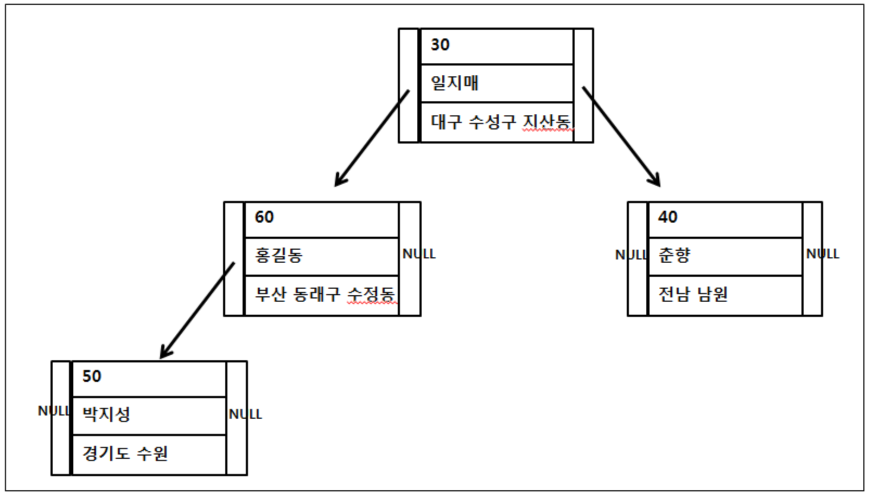
```
#define IS_EMPTY(first)    (first == -1)
```

(5) List space의 초기화

(6) Node의 할당과 반환

(7) Node의 연결

0	30	일지매	대구 수성구 지산동
1	60	홍길동	부산 동래구 수정동
2	40	춘향	전남 남원
3	50	박지성	경기도 수원
4			
5			
6			
7			

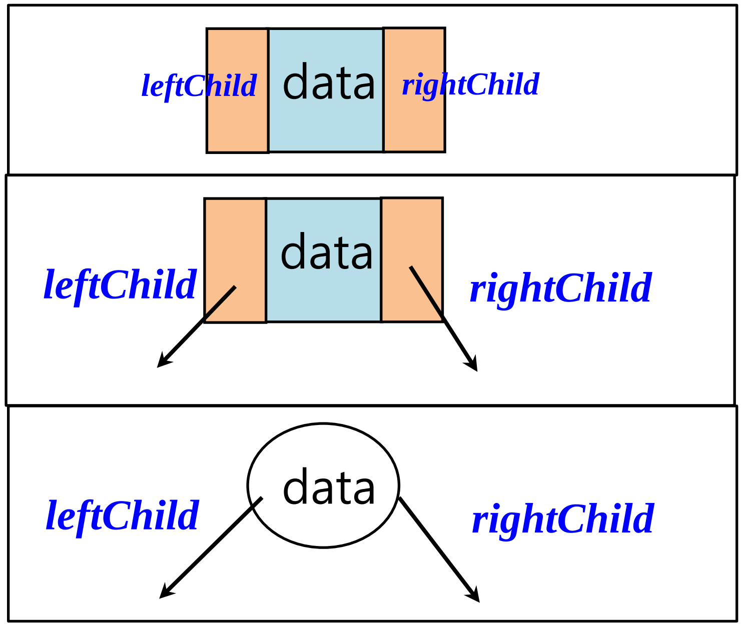


5.2.3.2 Linked Representation

```
typedef struct node *nodePointer;
typedef struct node {
    int   data;
    nodePointer  leftChild, rightChild;
} ;
nodePointer root;
```

```
typedef struct {
                int key;
                char name[20];
                char address[100];
} element;
typedef struct node *nodePointer;
typedef struct node {
                nodePointer leftChild;
                element data;
                nodePointer rightChild;

} ;
nodePointer root;
```



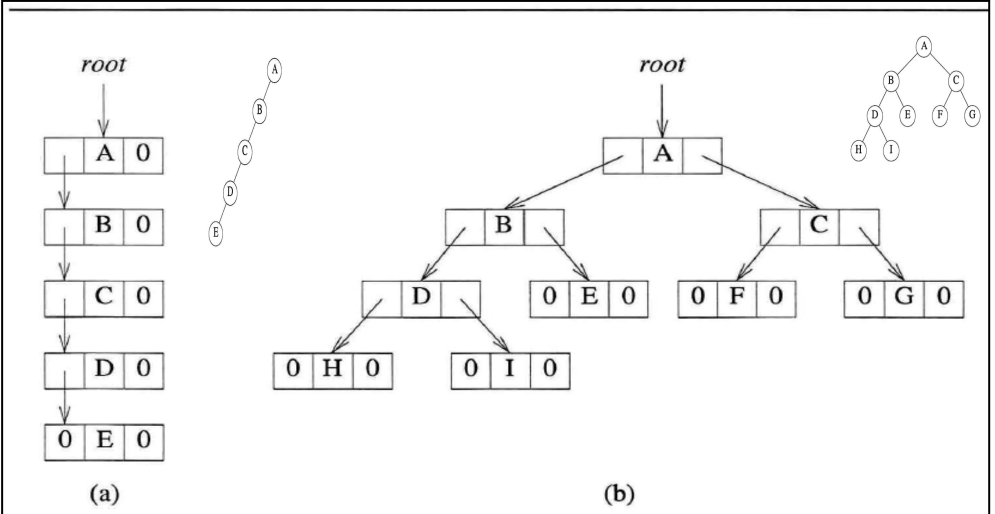


Figure 5.14: Linked representation for the binary trees of Figure 5.10

5.3 Binary Tree Traversal

Traverse a tree == **visit each node in the tree exactly once**.

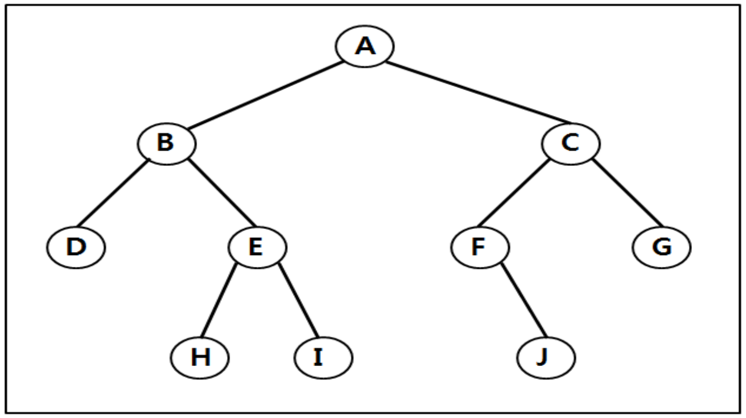
- **A full traversal ==** produce **a linear order for the nodes in a tree**.
- **linear order** == the order in which the nodes are visited.
  - If we let *L*, *V*, and *R* stand for moving left, visiting the node, and moving right at a node, then there are six possible combinations of traversal: *LVR*, *LRV*, *VLR*, *VRL*, *RVL*, and *RLV*.

    - *LVR* : ***inorder*** ← visit a node after traversing its left subtree but before the traversal of its right subtree.
    - *LRV* : ***postorder*** ← visit a node after traversing its left and right subtrees.
    - *VLR* : ***preorder*** ← visit a node before the traversal of its subtrees.

5.3.1 Inorder Traversal

```
void inorder(nodePointer ptr)
{
    /* inorder tree traversal */
    if (ptr) {
        inorder(ptr->leftChild);
        printf("%d",ptr->data);
        inorder(ptr->rightChild);
    }
}
```

Program 5.1: Inorder traversal of a binary tree



output: D B H E I A F J C G

5.3.4 Iterative Inorder Traversal

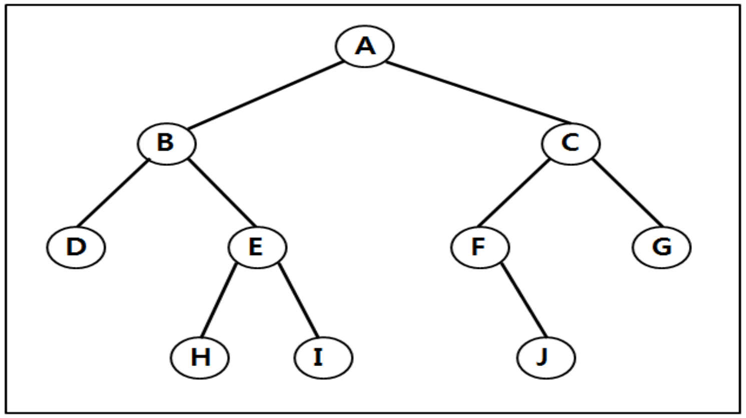
```
void iterinorder(nodePointer node)
{
    int top = -1;                /* initialize stack */
    nodePointer stack[MAX_STACK_SIZE];
    while (1) {
        for(; node; node= node->leftChild)
            push(node);
        node = pop();
        if (!node) break;        /* empty stack */
        printf( "%d", node->data);
        node = node->rightChild;
    }
}
```

Program 5.4: Iterative inorder traversal

5.3.2 Preorder Traversal

```
void preorder(nodePointer ptr)
{
    /* preorder tree traversal * /
    if (ptr) {
        printf("%d",ptr->data);
        preorder(ptr->leftChild);
        preorder(ptr->rightChild);
    }
}
```

Program 5.2: Preorder traversal of a binary tree

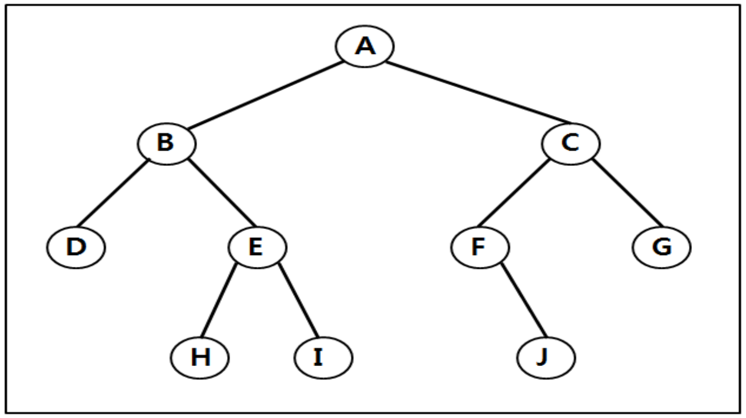


output: A B D E H I C F J G

5.3.3 Postorder Traversal

```
void postorder(nodePointer ptr)
{
    /* postorder tree traversal */
    if (ptr) {
        postorder(ptr->leftChild);
        postorder(ptr->rightChild);
        printf("%d",ptr->data);
    }
}
```

Program 5.3: Postorder traversal of a binary tree

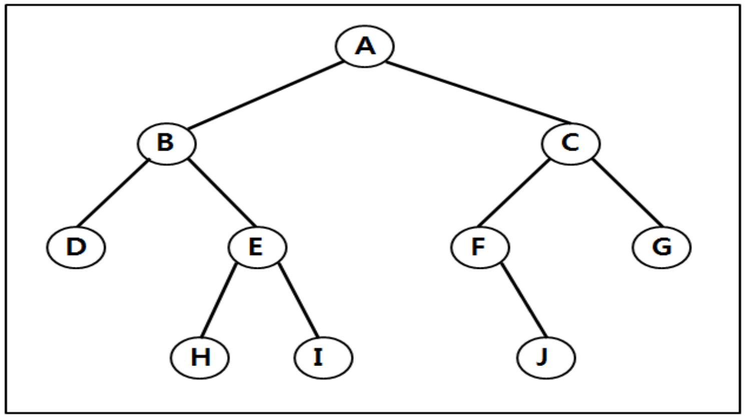


output: D H I E B J F G C A

5.3.5 Level-Order Traversal

```
void levelOrder(nodePointer ptr)
{
    / * level order tree traversal */
    int front = rear = 0;
    nodePointer queue[MAX_QUEUE_SIZE];
    if (!ptr) return; / *empty tree* /
    addq(ptr);
    while (1) {
        ptr = deleteq();
        if (ptr) {
                printf("%d", ptr->data);
                if(ptr->leftChild) addq(ptr->leftChild);
                if (ptr->rightChild) addq(ptr->rightChild);
        } else
                break;
    }
}
```

Program 5.5: Level-order traversal of a binary tree



output: A B C D E F G H I J

Whether written iteratively or recursively, the inorder, preorder, and postorder traversals all require a **stack**.

We now turn to a traversal that requires a **queue**. This traversal, called ***level-order traversal***, visits the nodes using the ordering suggested by the node numbering scheme of Figure 5.11. Thus, we visit the root first, then the root's left child, followed by the root's right child. We continue in this manner, visiting the nodes at each new level from the leftmost node to the rightmost node.

5.3.6 Traversal without a Stack

Is binary tree traversal possible without the use of extra space for a stack? (Note that a recursive tree traversal algorithm also implicitly uses a stack.)

- One simple solution is to add **a parent field** to each node. Then we can trace our way back up to any root and down again.
- Another solution, which requires two bits per node, represents binary trees as **threaded binary trees**. We study this in Section 5.5. If the allocation of this extra space is too costly, then we can use the leftChild and rightChild fields to maintain the paths back to the root. The stack of addresses is stored in the leaf nodes.

5.4 Additional Binary Tree Operations

5.4.1 Copying Binary trees

A slightly modified version of *postorder* to copy a binary tree.

```
nodePointer copy(nodePointer original)
{
        / * this function returns a nodePointer to an exact copy of the original tree */
        nodePointer temp;
        if (original) {
                MALLOC(temp, sizeof(*temp));
                temp->leftChild = copy(original->leftChild);
                temp->rightChild = copy(original->rightChild);
                temp->data = original->data;
                return temp;
        }
        return NULL;
}
```

Program 5.6: Copying a binary tree

5.4.2 Testing Equality

- Determining the equivalence of two binary trees.
- Equivalent binary trees have the same structure and the same information in the corresponding nodes. By the same structure we mean that every branch in one tree corresponds to a branch in the second tree, that is, the branching of the two trees is identical.
- A modification of *preorder* traversal to test for equality. This function returns TRUE if the two trees are equivalent and FALSE if they are not.

```
int equal(nodePointer first, nodePointer second)
{
    /* function returns FALSE if the binary trees first and second are not equal,
       Otherwise it returns TRUE */
        return ((!first && !second) || (first && second &&
                (first->data == second->data) &&
                equal(first->leftChild, second->leftChild) &&
                equal(first->rightChild, second->rightChild))
}
```

Program 5.7: Testing for equality of binary trees

5.4.3 The Satisfiability Problem

Consider the set of formulas that we can construct by taking variables $x_1, x_2, ..., x_n$ and operators $\wedge$ (*and*), $\vee$ (*or*), $\neg$(*not*). The variables can hold only one of two possible values, *true* or *false*. The set of expressions that we can form using these variables and operators is defined by the following rules:

(1) A variable is an expression.

(2) If x and y are expressions, then $\neg x$, $x \wedge y$, $x \vee y$ are expressions.

(3) Parentheses can be used to alter the normal order of evaluation, which is $\neg$ before $\wedge$ before $\vee$.

These rules comprise the formulas in the **propositional calculus** since other operations, such as **implication**, can be expressed using $\neg$, $\vee$, and $\wedge$.

The expression:

$$x_1 \vee (x_2 \wedge \neg x_3)$$

is a formula (read as " x_1 *or* x_2 *and not* x_3"). If x_1 and x_3 are false and x_2 is true, then the value of the expression is:

$$\begin{aligned} &\text{false} \vee (\text{true} \wedge \neg\text{false}) \\ &= \text{false} \vee \text{true} \\ &= \text{true} \end{aligned}$$

The satisfiability problem for formulas of the propositional calculus asks if there is an assignment of values to the variables that causes the value of the expression to be true. This problem was originally used by Newell, Shaw, and Simon in the late 1950s to show the viability of heuristic programming (The Logic Theorist) and is still of keen interest to computer scientists.

$$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$$

The most obvious algorithm to determine satisfiability is to let (x_1, x_2, x_3) take on all possible combinations of true and false values and to check the formula for each combination. For *n* variables there are 2^n possible combinations of *true*= t and *false* = f. For example, for *n* = 3, the eight combinations are: (t,t,t), (t,t,f), (t,f,t), (t,f,f), (f,t,t), (f,t,f), (f,f,t), (f,f,f). The algorithm will take $O(g2^n)$, or exponential time, where *g* is the time to substitute values for x_1, x_2, …, x_n and evaluate the expression.

Again, let us assume that our formula is already in a binary tree, say

$$(x_1 \land \neg x_2) \lor (\neg x_1 \land x_3) \lor \neg x_3$$

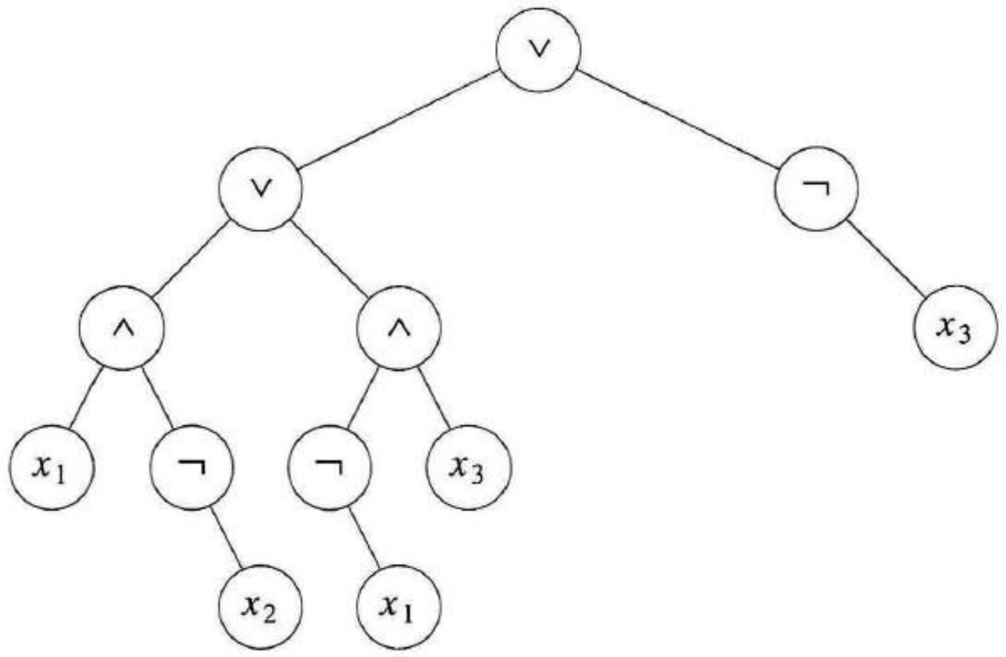


Figure 5.18: Propositional formula in a binary tree

To evaluate an expression, we traverse its tree in postorder.

Notice that a node containing $\neg$ has only a right branch, since, is **a unary operator**.

```
typedef enum {not, and, or, true, false} logical;
typedef struct node *nodePointer;
typedef struct node {
        nodePointer leftChild;
        logical data;
        short int value;
        nodePointer rightChild;
};
```

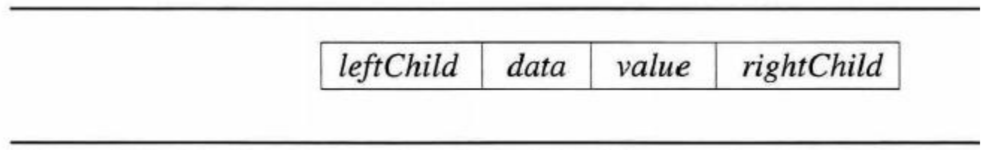


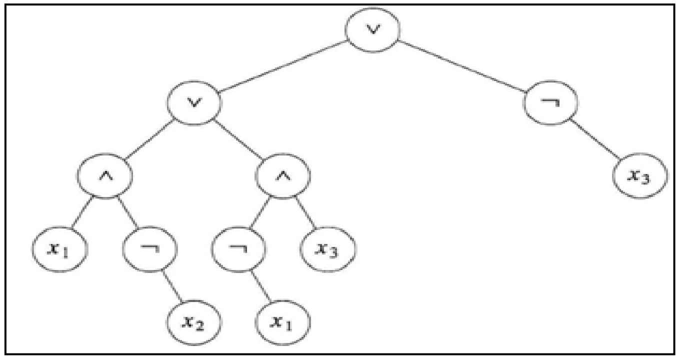
Figure 5.19: Node structure for the satisfiability problem

We assume that for leaf nodes, *node->data* contains the current value of the variable represented at this node. For example, we assume that the tree of Figure 5.18 contains either TRUE or FALSE in the data field of x_1, x_2, and x_3. We also assume that an expression tree with *n* variables is pointed at by *root*. With these assumptions we can write our first version of a satisfiability algorithm (Program 5.8).

```
for (all $2^n$ possible combinations) {
        generate the next combination;
        replace the variables by their values;
        evaluate root by traversing it in postorder;
        if (root->value) {
                printf(<combination>);
                return;
        }
}
printf("No satisfiable combination\n");
```

Program 5.8: First version of satisfiability algorithm

The C function that evaluates the tree is easily obtained by modifying the original, recursive postorder traversal. The function postOrderEval (Program 5.9) shows the C code that implements this portion of the satisfiability algorithm.



```
void postOrderEval(nodePointer node)
{/* modified post order traversal to evaluate a propositional calculus tree */
   if (node) {
            postOrderEval(node->leftChild);
            postOrderEval(node->rightChild);
            switch (node->data) {
               case not: node->value =!node->rightChild->value;
                        break;
               case and: node->value = node->leftChild->value && node->rightChild->value;
                        break;
               case or: node->value = node->leftChild->value || node->rightChild->value;
                        break;
               case true: node->value = TRUE;
                        break;
               case false: node->value = FALSE;
            }
   }
}
```

Program 5.9: Postorder evaluation function

과제 1 (수식을 읽어 이진 트리를 형성하고 그를 평가) [10점]

$(x1 \wedge \neg x2) \vee (\neg x1 \wedge x3) \vee \neg x3$

위 수식을 읽어 그림 5.18의 이진트리를 형성하고 그를 evaluate하고자 한다. 아래 절차에 따라 그 작업을 수행하시오. 단, 프로그램의 편의를 위하여 x1, x2, x3를 대신하여 x, y, z를 사용하며, ¬ 대신 !를 사용하고, ∧ 대신 &를 사용하고, ∨ 대신 |를 사용한다. 즉, 그 입력은 다음과 같다.

(x&!y)|(!x&z)|!z

(1) 위 입력을 키보드로부터 받아들이고 그를 출력하시오. [1점]

(2) 입력한 infix expression을 postfix expression으로 바꾸어 postfix_expr이라는 배열에 저장하고 그를 출력하시오. [3점]
이 작업은 우리가 2장에서 수행한 방법을 수정하여 사용하면 된다.
참고로, 위 입력에 대한 postfix expression은 다음과 같다.
xy!&x!z&|z!|

(3) 구한 postfix_expr을 바탕으로 그를 그림 5.18의 binary tree로 나타낸 후, 그 tree를 preorder traversal로 출력하시오. [4점]

(4) x, y, z에 1과 0을 대입하여 각 값의 조합에 대한 위 binary tree를 Program 5.9의 방법으로 평가한 결과를 출력하시오. 변수가 3개이므로 모든 조합은 7가지가 나온다. [2점]

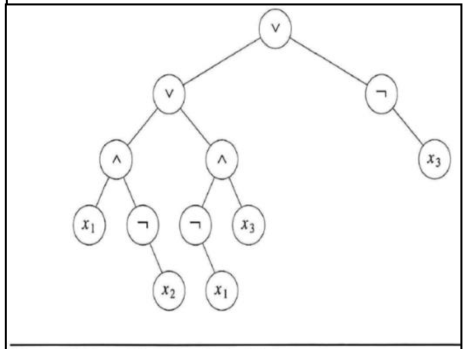
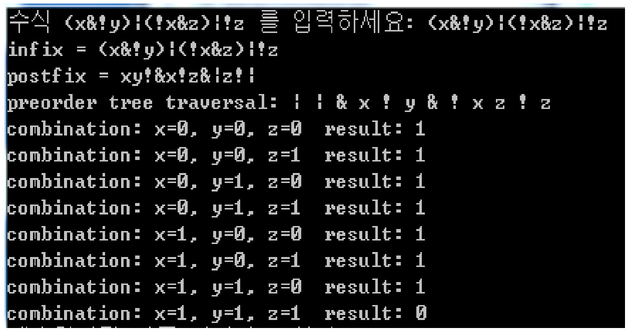
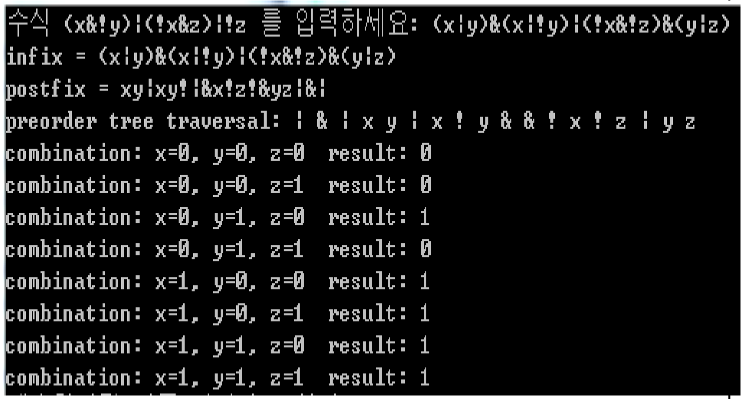


Figure 5.18: Propositional formula in a binary tree


```
수식 (x&!y)|(!x&z)|!z 를 입력하세요: (x&!y)|(!x&z)|!z
infix = (x&!y)|(!x&z)|!z
postfix = xy!&x!z&|z!|
preorder tree traversal: | | & x ! y & ! x z ! z
combination: x=0, y=0, z=0  result: 1
combination: x=0, y=0, z=1  result: 1
combination: x=0, y=1, z=0  result: 1
combination: x=0, y=1, z=1  result: 1
combination: x=1, y=0, z=0  result: 1
combination: x=1, y=0, z=1  result: 1
combination: x=1, y=1, z=0  result: 1
combination: x=1, y=1, z=1  result: 0
```


```
수식 (x&!y)|(!x&z)|!z 를 입력하세요: (x|y)&(x|!y)|(!x&!z)&(y|z)
infix = (x|y)&(x|!y)|(!x&!z)&(y|z)
postfix = xy|xy!|&x!z!&yz|&|
preorder tree traversal: | & | x y | x ! y & & ! x ! z | y z
combination: x=0, y=0, z=0  result: 0
combination: x=0, y=0, z=1  result: 0
combination: x=0, y=1, z=0  result: 1
combination: x=0, y=1, z=1  result: 0
combination: x=1, y=0, z=0  result: 1
combination: x=1, y=0, z=1  result: 1
combination: x=1, y=1, z=0  result: 1
combination: x=1, y=1, z=1  result: 1
```

이 과제의 경우, 수업시간에 완료하기에는 무리가 있을 수 있다. 따라서, 화요일 자정까지 완료하여 ABEEK에 올리면 감점없이 인정한다. 참고로, 나는 했으며 그 결과는 위와 같다. 왼 쪽의 예는 우리 교재의 예를 보인 것이고, 오른 쪽의 예는 다음을 입력한 것이다. **(x|y)&(x|!y)|(!x&!z)&(y|z)**

5.5 THREADED BINARY TREES

5.5.1 Threads

If we look carefully at the linked representation of any binary tree, we notice that there are more null links than actual pointers. Specifically, there are $n + 1$ null links out of $2n$ total links. A. J. Perlis and C. Thornton have devised a clever way to make use of these null links. They replace the null links by pointers, called ***threads***, to other nodes in the tree. To construct the threads we use the following rules (assume that *ptr* represents a node):

(1) If ***ptr->leftChild*** is null, replace *ptr->leftChild* with a pointer to the node that would be visited before *ptr* in an inorder traversal. That is we replace the null link with a pointer to the ***inorder predecessor*** of *ptr*.

(2) If ***ptr->rightChild*** is null, replace *ptr->rightChild* with a pointer to the node that would be visited after *ptr* in an inorder traversal. That is we replace the null link with a pointer to the ***inorder successor*** of *ptr*.

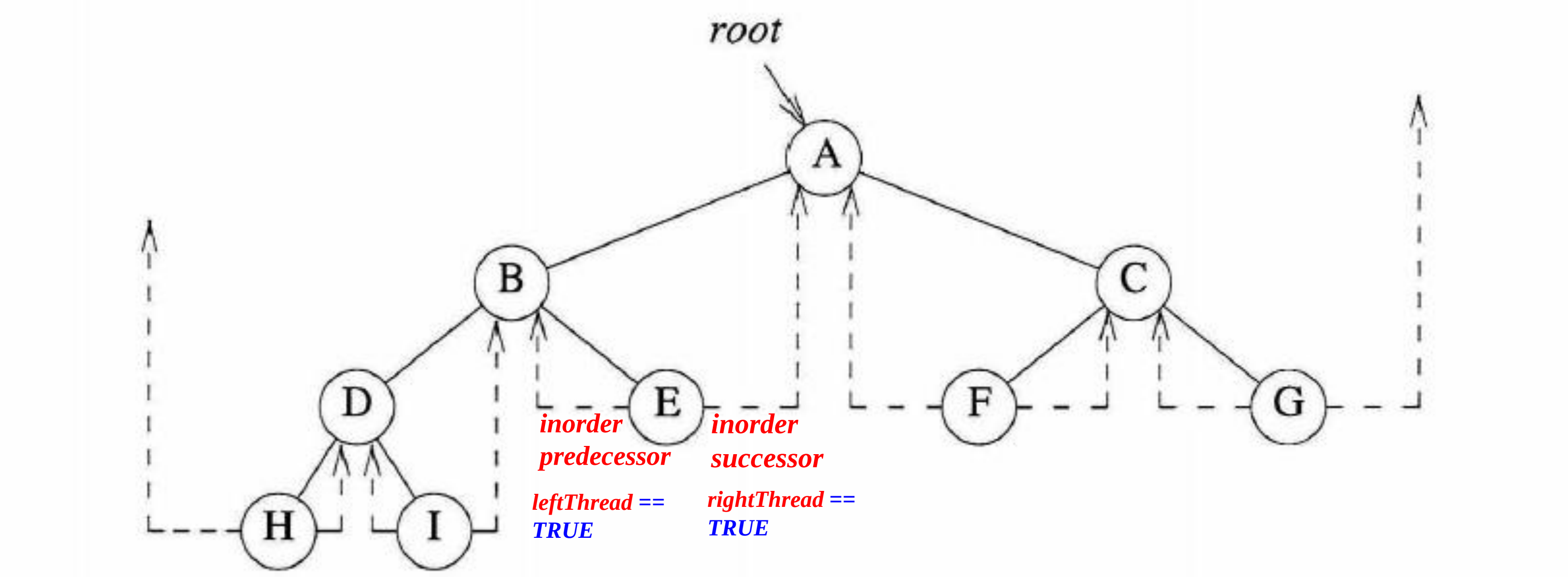


Figure 5.21: Threaded tree corresponding to Figure 5.10(b)

When we represent the tree in memory, we must be able to distinguish between threads and normal pointers. This is done by adding two additional fields to the node structure, *leftThread* and *rightThread*. Assume that *ptr* is an arbitrary node in a threaded tree. If *ptr->leftThread* = *TRUE*, then *ptr->leftChild* contains a thread; otherwise it contains a pointer to the left child. Similarly, if *ptr->rightThread* = *TRUE*, then *ptr->rightChild* contains a thread; otherwise it contains a pointer to the right child.
This node structure is given by the following C declarations:

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

In Figure 5.21 two threads have been left dangling: one in the left child of *H*, the other in the right child of *G*. In order that we leave no loose threads, we will assume **a header node** for all threaded binary trees. The original tree is the left subtree of the header node. An empty binary tree is represented by its header node as in Figure 5.22. The complete memory representation for the tree of Figure 5.21 is shown in Figure 5.23.

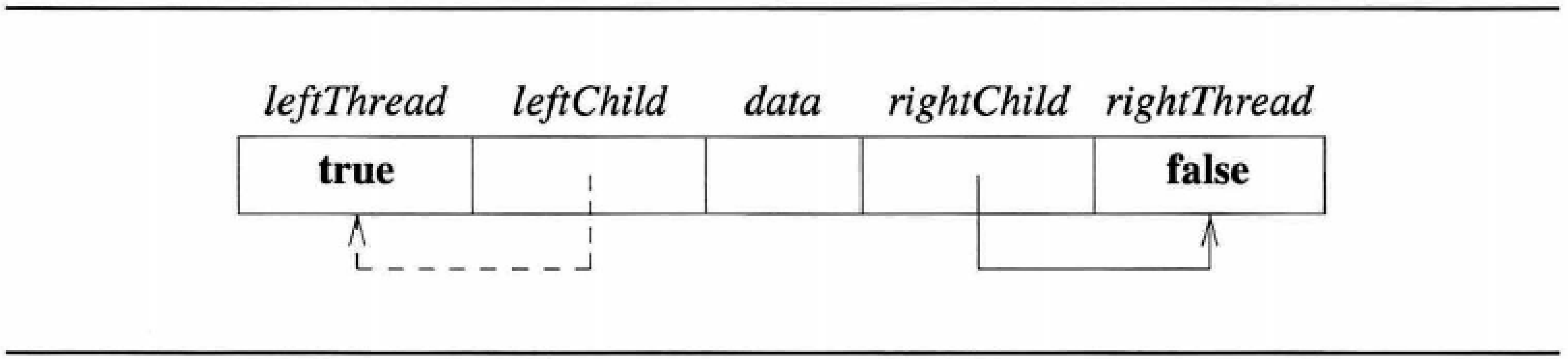


Figure 5.22: An empty threaded binary tree

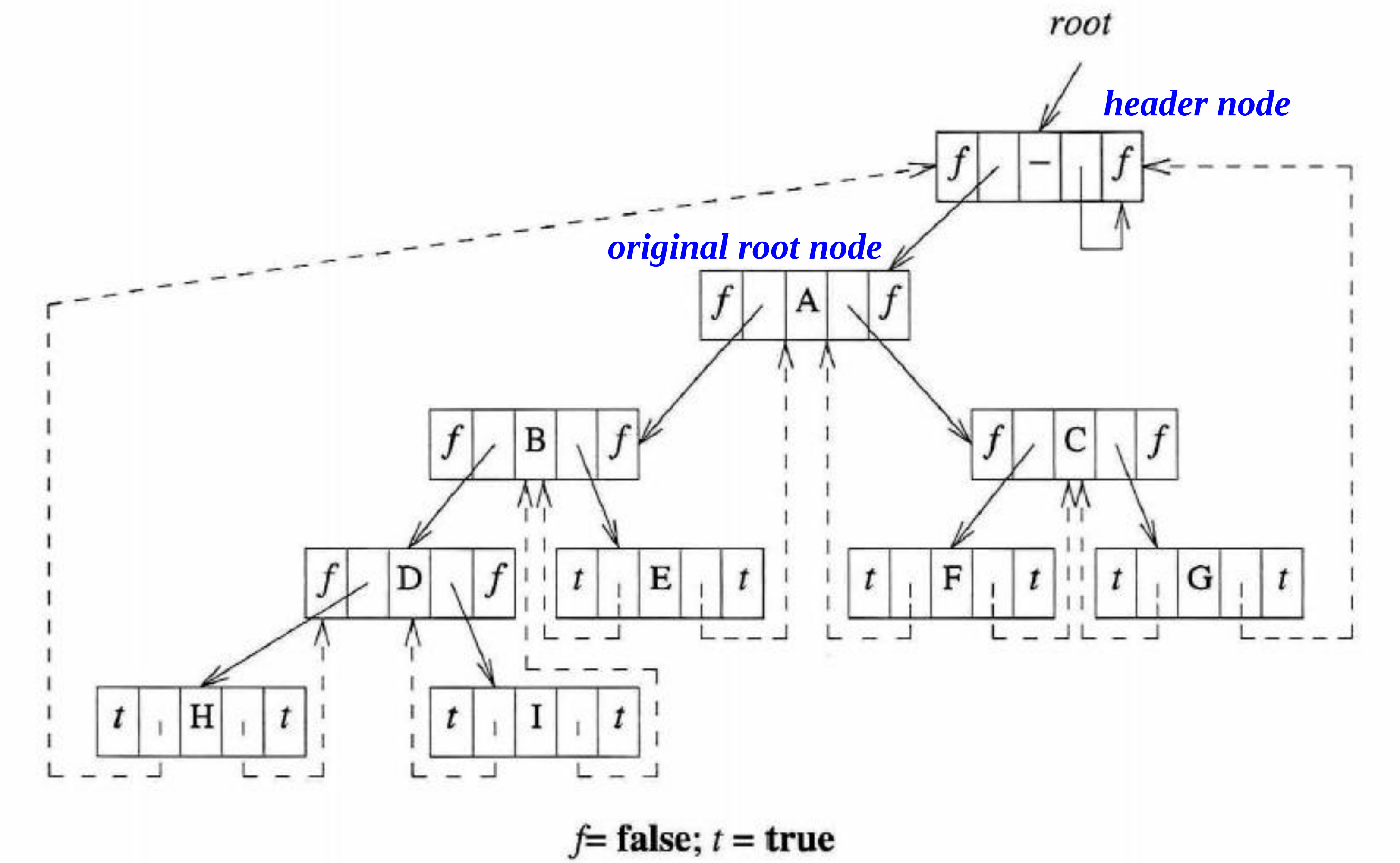


Figure 5.23: Memory representation of threaded tree

5.5.2 Inorder Traversal of a Threaded Binary Tree

By using the threads, we can perform an inorder traversal without making use of a stack. Observe that for any node, *ptr*, in a threaded binary tree, if *ptr->rightThread = TRUE*, the inorder successor of *ptr* is *ptr->rightChild* by definition of the threads. Otherwise we obtain the inorder successor of *ptr* by following a path of left-child links from the right-child of *ptr* until we reach a node with *leftThread = TRUE*. The function *insucc* (Program 5.10) finds the inorder successor of any node in a threaded tree without using a stack.

```
threadedPointer insucc(threadedPointer tree)
{
    /* find the inorder sucessor of tree in a threaded binary tree */
    threadedPointer temp;
    temp = tree->rightChild;
    if (!tree->rightThread) // internal node
        while (!temp->leftThread)
            temp = temp->leftChild;
    return temp;
}
```

Program 5.10: Finding the inorder successor of a node

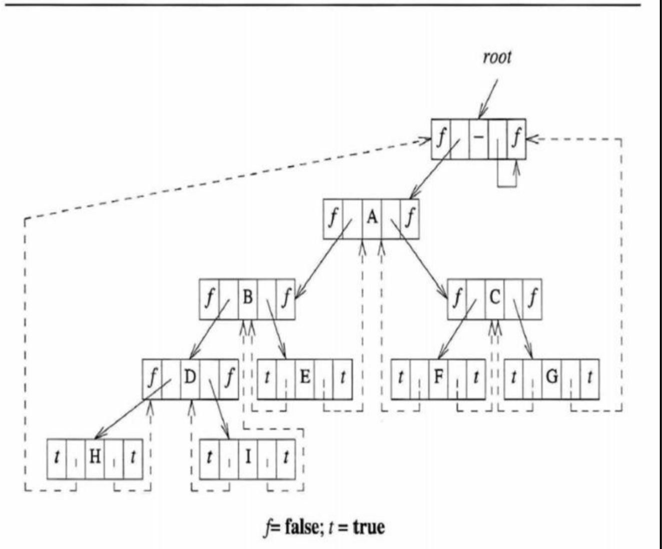


Figure 5.23: Memory representation of threaded tree

To perform **an inorder traversal** we make repeated calls to *insucc*. The operation is implemented in *tinorder* (Program 5.11). This function assumes that the tree is pointed to by the header node's left child and that the header node's right thread is *FALSE*. The computing time for *tinorder* is still O(*n*) for a threaded binary tree with *n* nodes, although the constant factor is smaller than that of *iterInorder*.

```
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);
        }
}
```

```
tinorder(root);
```

Program 5.11: Inorder traversal of a threaded binary tree

5.5.3 Inserting a Node into a Threaded Binary Tree

We now examine how to make insertions into a threaded tree. This will give us a function for growing threaded trees. We shall study only the case of inserting *r* as the right child of a node *s*. The case of insertion of a left child is given as an exercise. The cases for insertion are

(1) If *s* has an empty right subtree, then the insertion is simple and diagrammed in Figure 5.24(a).

(2) If the right subtree of *s* is not empty, then this right subtree is made the right subtree of *r* after insertion. When this is done, *r* becomes the inorder predecessor of a node that has a *leftThread* == *true* field, and consequently there is a thread which has to be updated to point to *r*. The node containing this thread was previously the inorder successor of *s*. Figure 5.24(b) illustrates the insertion for this case. The function *insertRight* (Program 5.12) contains the C code which handles both cases.

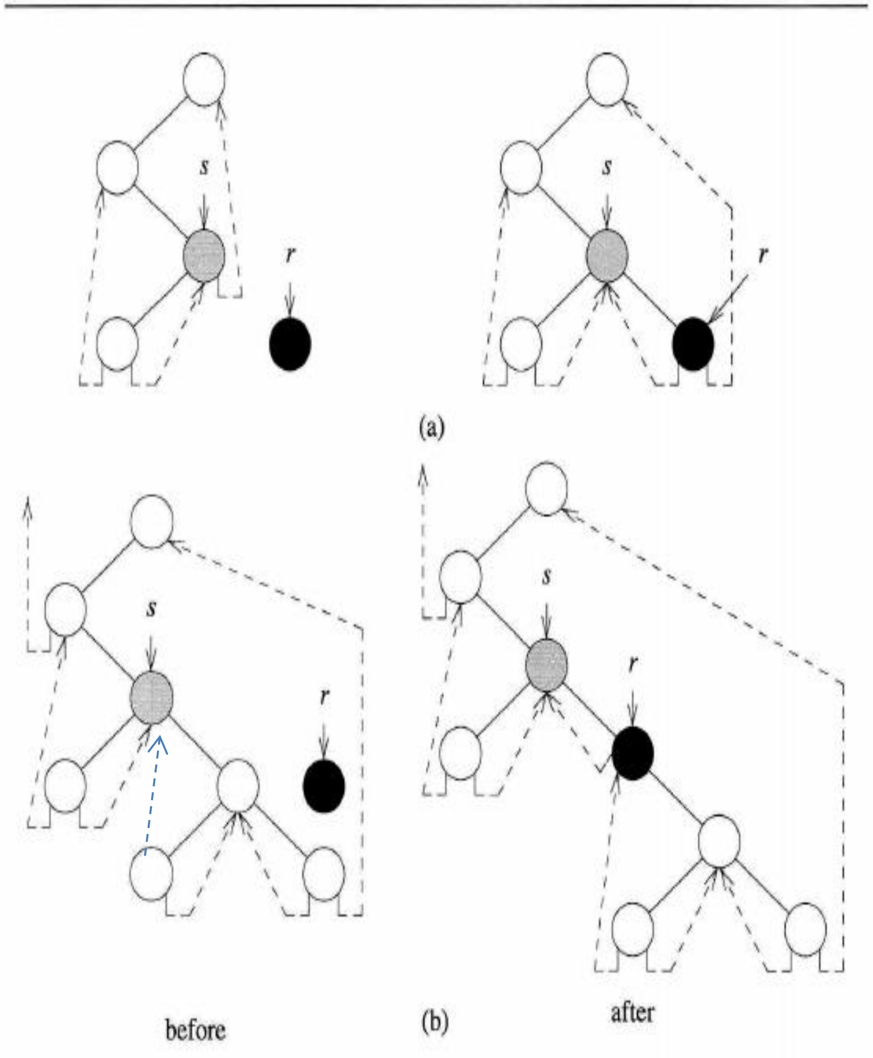


Figure 5.24: Insertion of *r* as a right child of *s* in a threaded binary tree

```
void insertRight(threadedPointer s, threadedPointer r)
{/* insert r as the right child of s */
        threadedPointer temp;
        r->rightChild = s->rightChild;
        r->rightThread = s->rightThread;
        r->leftChild = s;
        r->leftThread = TRUE;
        s->rightChild = r;
        s->rightThread = FALSE;
        if (! r->rightThread) {
                temp= insucc(r);
                temp->leftChild = r;
        }
}
```

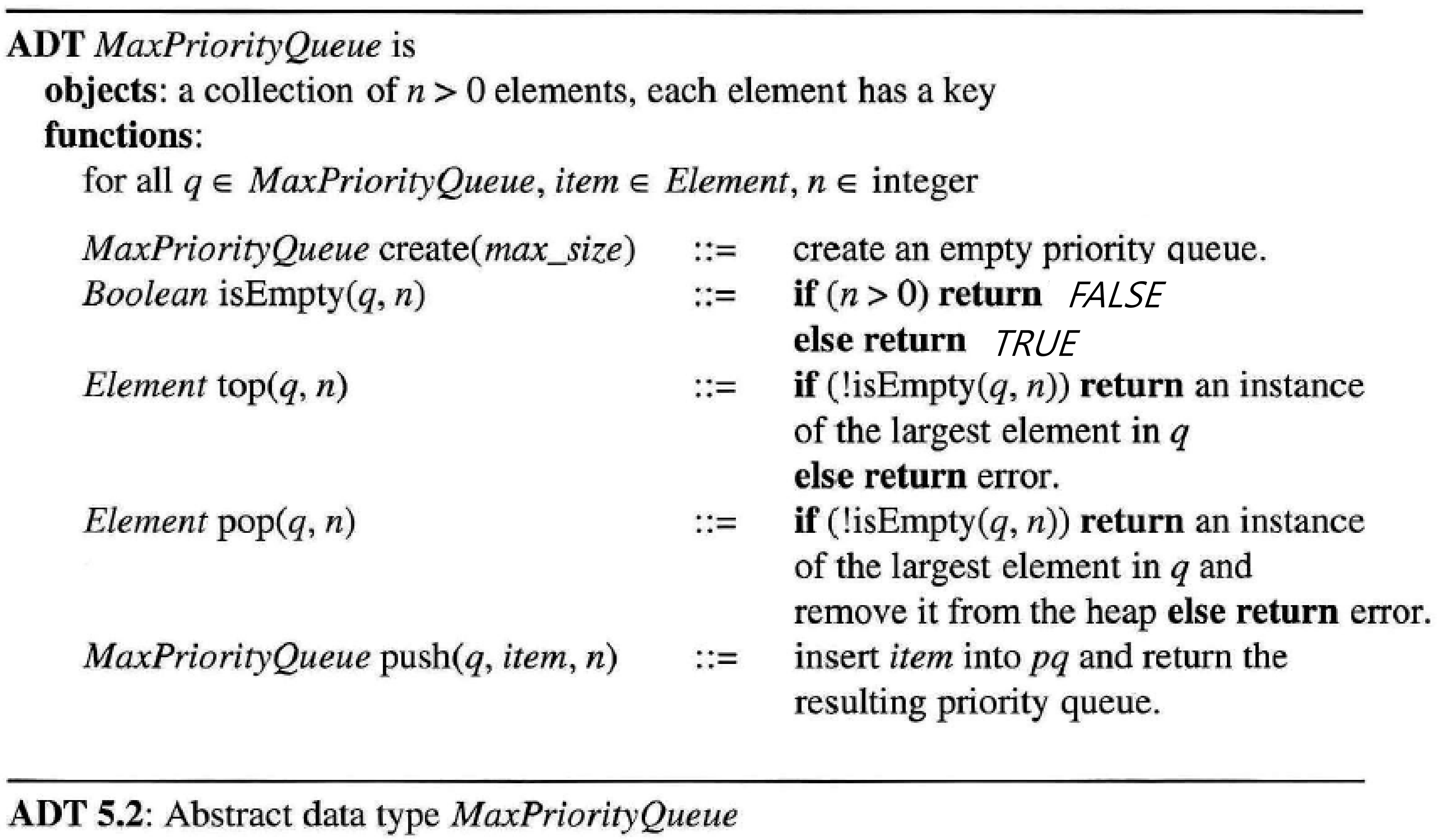
Program 5.12: Right insertion in a threaded binary tree

5.6 Heaps

5.6.1 Priority Queues

Heaps are frequently used to implement ***priority queues***. In this kind of queue, 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. ADT 5.2 specifies **a max priority queue**.

The simplest way to represent a **priority queue** is as an **unordered linear list**. Regardless of whether this list is represented **sequentially** or **as a chain**, the *isEmpty* function takes O(1) time; the *top*() function takes Θ(*n*) time, where *n* is the number of elements in the priority queue; a push can be done in O(1) time as it doesn't matter where in the list the new element is inserted; and a *pop* takes Θ(*n*) time as we must first find the element with max priority and then delete it. As we shall see shortly, when a ***max heap*** is used, the complexity of *isEmpty* and *top* is O(1) and that of *push* and *pop* is O(log *n*).



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** *FALSE* **else return** *TRUE*
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*

5.6.2 Definition of a Max Heap

Definition: A ***max (min) tree*** is a tree in which the key value in each node is no smaller (larger) than the key values in its children (if any). A ***max heap*** is **a complete binary tree** that is also **a max tree**. A ***min heap*** is a complete binary tree that is also **a min tree**. □

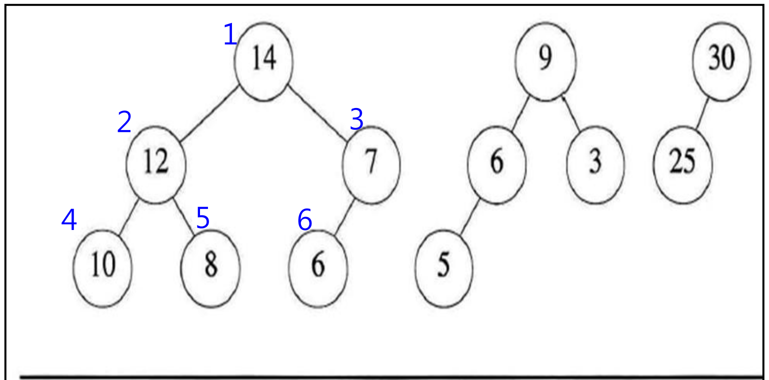


Figure 5.25: Max heaps

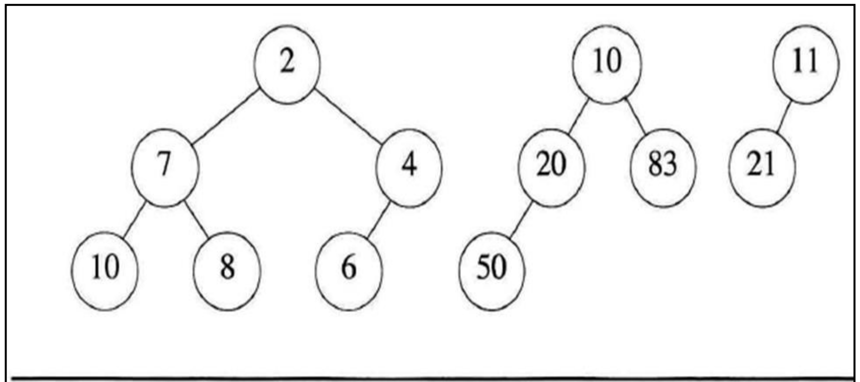
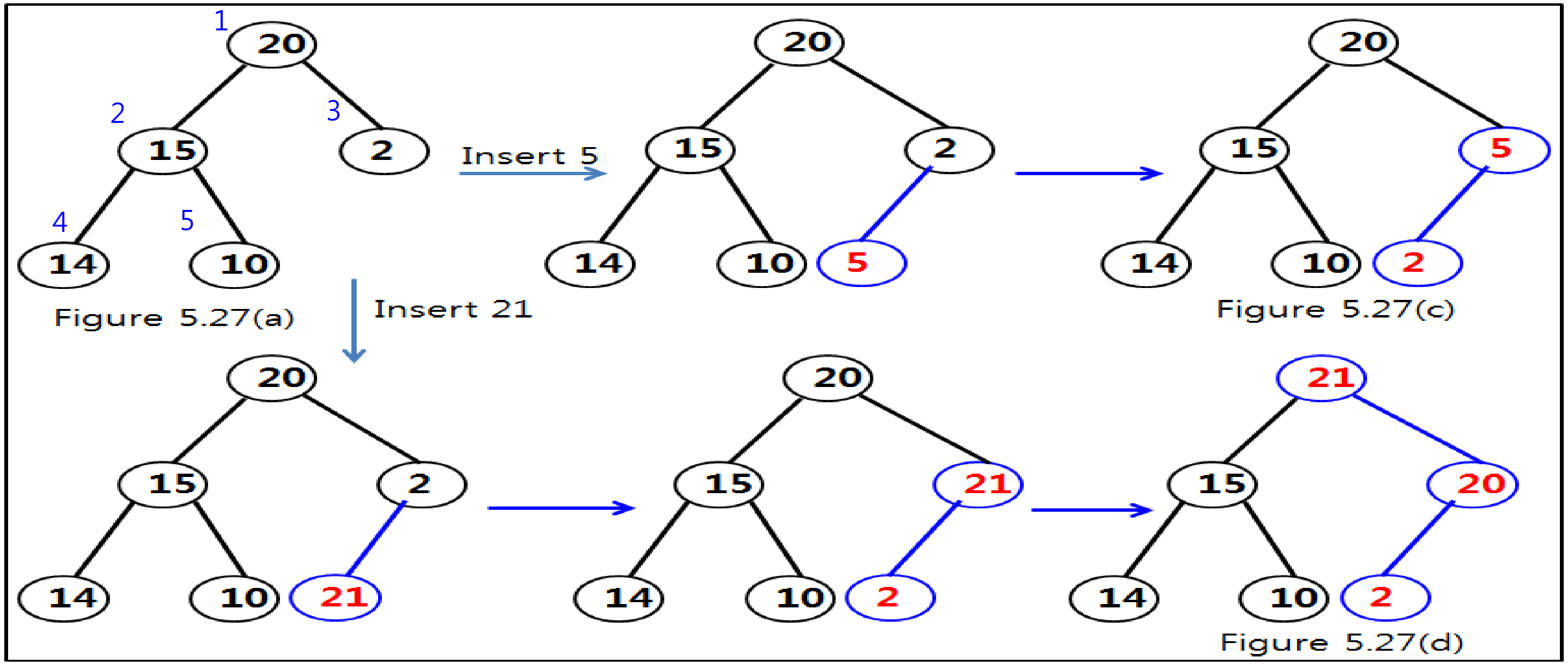


Figure 5.26: Min heaps

5.6.3 Insertion into a Max Heap

A max heap with five elements is shown in Figure 5.27(a). When an element is added to this heap, the resulting six-element heap must have the structure shown in Figure 5.27(b), because a heap is a complete binary tree. To determine the correct place for the element that is being inserted, we use a ***bubbling up*** process that begins at the new node of the tree and moves toward the root. The element to be inserted bubbles up as far as is necessary to ensure a max heap following the insertion.



To implement the insertion strategy just described, we need to go from an element to its parent. Lemma 5.4 enables us to locate the parent of any element easily. Program 5.13 performs an insertion into a max heap. We assume that the heap is created using the following C declarations:

```
#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 current_size = 0;

/* insert item into a max heap of current size *current_size_ptr */
void push(element item, int *current_size_ptr)
{
            int i;
            if (HEAP_FULL(*current_size_ptr)) {
                        fprintf(stderr, "The heap is full. \n");
                        exit(EXIT_FAILURE);
            }
            i = ++(*current_size_ptr);
            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

삽입 x

(1) X를 Complete binary tree의 끝에 append;

(2) X를 부모 node와 비교하면서 제자리 찾는다.

Since a heap is a complete binary tree with n elements, it has a height of $\lceil \log_2(n+1) \rceil$. This means that the while loop is iterated $O(\log_2 n)$ times. Hence, the complexity of the insertion function is $O(\log_2 n)$.

5.6.4 Deletion from a Max heap

When an element is to be deleted from a max heap, it is taken from the root of the heap. For instance, a deletion from the heap of Figure 5.27(d) results in the removal of the element 21. Since the resulting heap has only five elements in it, the binary tree of Figure 5.27(d) needs to be restructured to correspond to **a complete binary tree** with five elements. To do this, we remove the element in position 6 (i.e., the element 2). Now we have the right structure (Figure 5.28(a)), but the root is vacant and the element 2 is not in the heap. If the 2 is inserted into the root, the resulting binary tree is not a max heap. The element at the root should be the largest from among the 2 and the elements in the left and right children of the root. This element is 20. It is moved into the root, thereby creating a vacancy in position 3. Since this position has no children, the 2 may be inserted here. The resulting heap is shown in Figure 5.27(a).

Now, suppose we wish to perform another deletion. The 20 is to be deleted. Following the deletion, the heap has the binary tree structure shown in Figure 5.28(b). To get this structure, the 10 is removed from position 5. It cannot be inserted into the root, as it is not large enough. The 15 moves to the root, and we attempt to insert the 10 into position 2. This is, however, smaller than the 14 below it. So, the 14 is moved up and the 10 inserted into position 4. The resulting heap is shown in Figure 5.28(c).
Program 5.14 implements this ***trickle down*** strategy to delete from a heap.

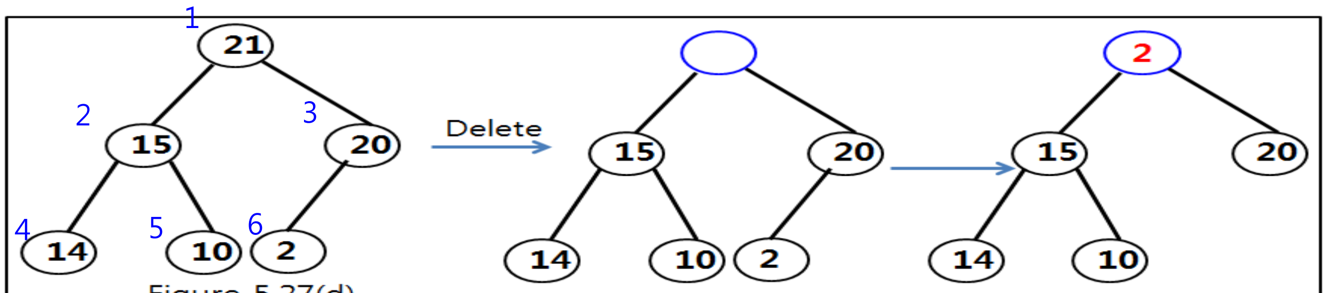


Figure 5.27(d)

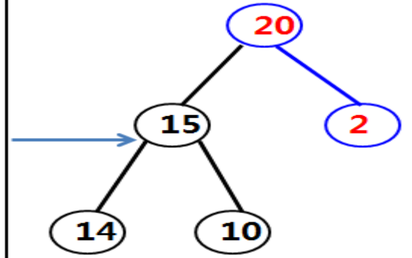


Figure 5.27(a)

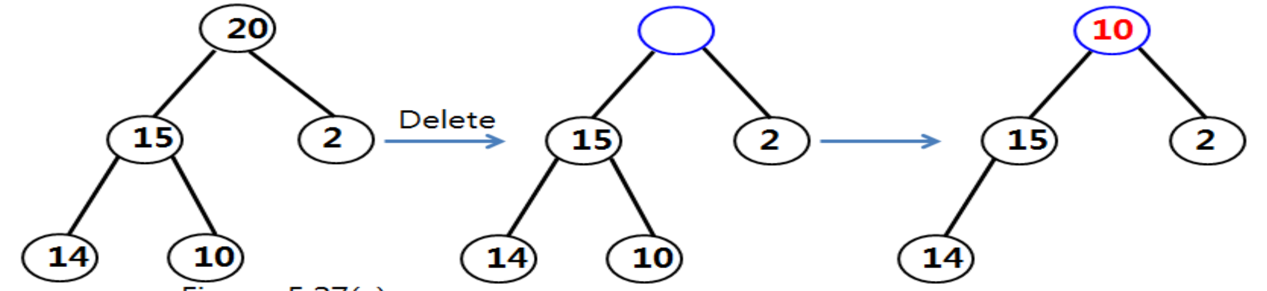


Figure 5.27(a)

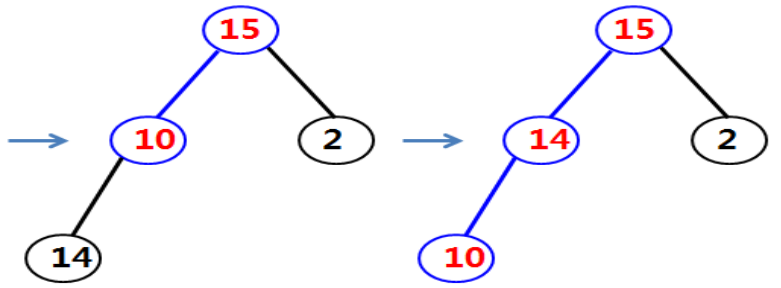


Figure 5.28(c)

```
/* delete element with the highest key from the heap */
element pop(int *current_size_ptr)
{
        int parent, child;
        element item, temp;
        if (HEAP_EMPTY(*current_size_ptr)) {
                fprintf(stderr, "The heap is empty\n");
                exit(EXIT_FAILURE);
        }
        /* save value of the element with the highest key */
        item = heap[1];         // heap[0]이 아니고 heap[1]이 root node이다.
        /* use last element in heap to adjust heap */
        temp = heap[(*current_size_ptr)--] ;
        parent = 1;
        child = 2;
        while (child <= *current_size_ptr) {
                /* find the larger child of the current parent */
                if (child < *current_size_ptr) && (heap[child].key < heap[child+l].key) child++;
                if (temp.key >= heap[child].key) break;
                /* move to the next lower level */
                heap[parent] = heap[child];
                parent = child;
                child *= 2;
        }
        heap[parent] = temp;
        return item;
}
```

Program 5.14: Deletion from a max heap

삭제

(1) Root node의 값을 보관한다.

(2) X = Complete binary tree의 끝을 delete;

(3) 빈 root node를 시작으로 X의 자리를 찾아가기.

Since the height of a heap with n elements is $\lceil \log_2(n+1) \rceil$, the while loop of *pop* is iterated $O(\log_2 n)$ times. Hence, the complexity of a deletion is $O(\log_2 n)$.

과제 2 (Max Heap을 배열로 형성) [4점]

아래 작업을 수행하는 C 프로그램을 작성하시오.

Max heap을 array를 이용한 sequential representation으로 표현하고자 한다. 아래 사항들을 수행하시오. [4점]

(1) 각 레코드는 <학번, 이름>으로 구성된다. 다음 9 개의 레코드들을 그 Max heap에 차례로 insert 하시오. 단, Max Heap은 각 레코드의 학번을 기준으로 형성한다.

<7, '일지매'>, <16, '홍길동'>, <49, '춘향'>, <82, '월매'>, <5, '이순신'>,
<31, '홍명보'>, <6, '차두리'>, <2, '박지성'>, <10, '메시'>.

(1) Root 노드의 키를 삭제하면서 그 레코드를 출력하시오. The resulting heap must satisfy the max heap definition.

a. Root 노드의 레코드를 삭제할까요 ? : y

<82, '월매'>

b. Root 노드의 레코드를 삭제할까요 ? : y

<49, '춘향'>

c. Root 노드의 레코드를 삭제할까요 ? : y

<31, '홍명보'>

a. Root 노드의 레코드를 삭제할까요 ? : n

5.7 Binary Search Trees

5.7.1 Definition

Definition: A **binary search tree** is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:

(1) Each node has exactly one key and the keys in the tree are distinct.

(2) The keys (if any) in the left subtree are smaller than the key in the root.

(3) The keys (if any) in the right subtree are larger than the key in the root.

(4) The left and right subtrees are also binary search trees. □

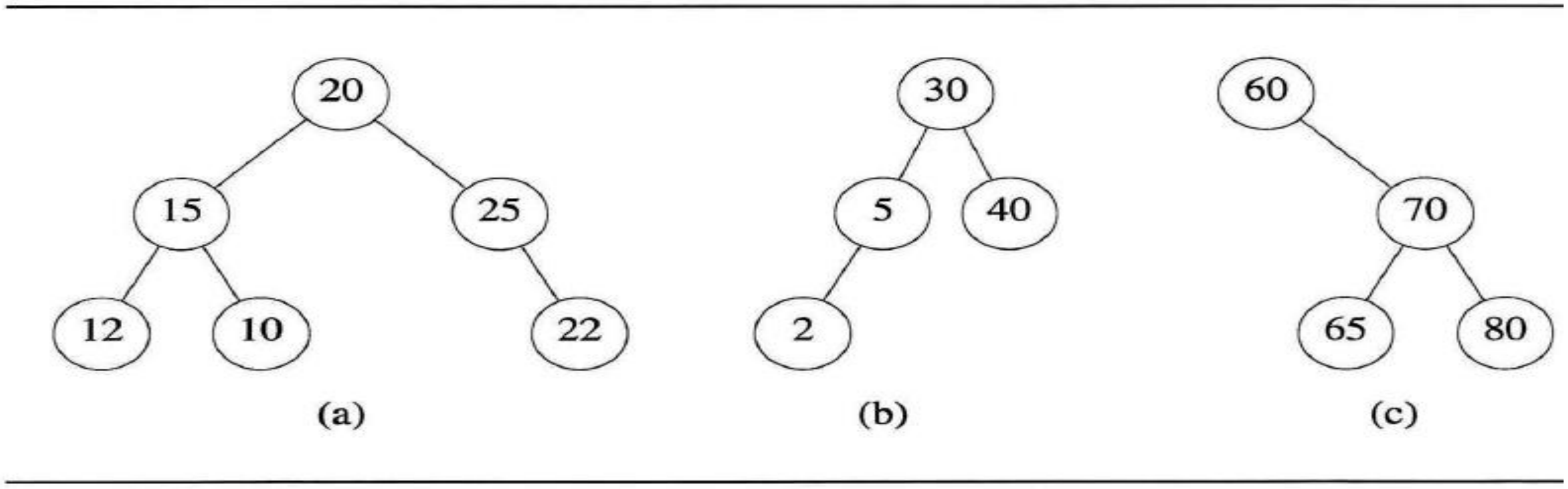


Figure 5.29: Binary trees

Not a BST BST BST

```
void insertBST(node **rootPtr, element x)
{
    node *ptr, *parent;
    int found = 0;
    found = searchParentToInsert(*rootPtr, x.key, &parent);
    if (found == 1) {
        printf("The key already exists in the tree!\n");
    } else { /* x.key is not in the tree */
        ptr = (node *)malloc(sizeof(node));
        ptr->data = x;
        ptr->leftChild = ptr->rightChild = NULL;
        if (*rootPtr == NULL) {
            *rootPtr = ptr;
        } else { /* insert ptr as a child of the parent */
            if (x.key < parent->data.key) parent->leftChild = ptr;
           else parent->rightChild = ptr;
        }
    }
}
```

```
/* tree에 k가 있으면 1을 return.
   그렇지 않으면, k를 삽입할 위치의 parent node를 *parentPtr에 두고 0을 return.
 */
int searchParentToInsert(node *root, int key, node **parentPtr)
{
    int found = 0;
    node *parent = NULL, *search = root;
    while (search != NULL) {
        if (key == search->data.key) {
            found = 1;
            break;
        } else {
            parent = search;
            if (key < search->data.key)
                search = search->leftChild;
            else
                search = search->rightChild;
        }
    }
    *parentPtr = parent;
    return found;
}
```

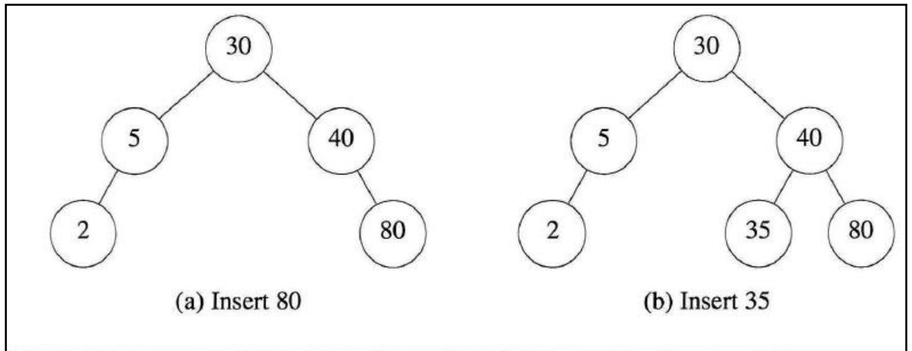


Figure 5.30: Inserting into a binary search tree

- Time complexity: O(h)

5.7.2 Searching a Binary Search Tree

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

Program 5.15: Recursive search of a binary search tree

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

Program 5.16: Iterative search of a binary search tree

Time complexity of *search* and *iterSearch*:

- Average case : O(h)
- Worst case : O(n) for skewed binary trees.

```
node* searchBST(node *root, int key)
{
    node *search;
    search = root;
    while (search != NULL) {
        if (key == search->data.key) break;
        if (key < search->data.key) search = search->leftChild;
        else search = search->rightChild;
    }
    return search;
}
```

5.7.4 Deletion from a Binary Search Tree

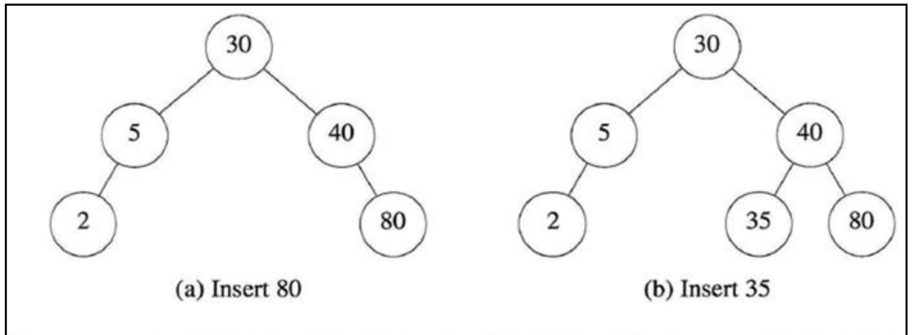


Figure 5.30: Inserting into a binary search tree

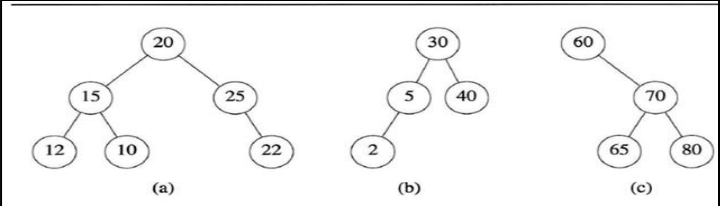


Figure 5.29: Binary trees

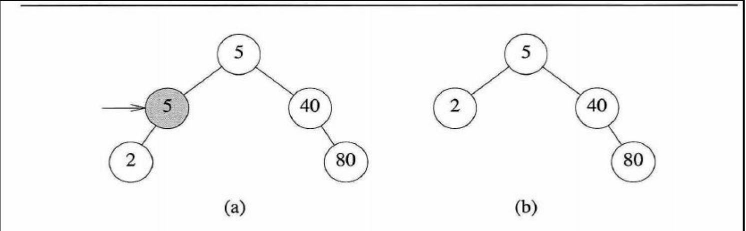


Figure 5.31: Deletion from a binary search tree

삭제할 node를 search, 그 노드의 부모 노드를 parent라 하자.

1. Deletion of a leaf node

(1) parent의 search에 대한 pointer를 NULL로 설정.

(2) free(search).

예) Delete 80 from Figure 5.30(a). ➔ Figure 5.29(b)

예) Delete 35 from Figure 5.30(b). ➔ Figure 5.30(a)

2. Deletion of a nonleaf node that has only one child

(1) parent의 search에 대한 pointer를 search의 single-child로 설정.

(2) free(search).

예) Delete 5 from Figure 5.30(a).➔ Make left child of node 30 to node 2

3. Deletion of a nonleaf that has two children

삭제할 노드 search의 the smallest one in its right subtree를 small이라 하고 그 노드의 부모 노드를 parentSmall이라 하자.

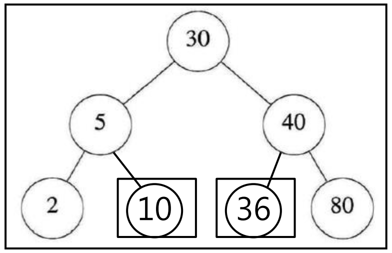
(1) search의 data를 small의 data로 대체.

(2) If (search == parentSmall) parentSmall->rightChild = small->rightChild;
else parentSmall->leftChild = small->rightChild;

(3) free(small);

예) Delete 30 from Figure 5.30(a) ➔ Figure 5.31(a) ➔ Figure 5.31(b)

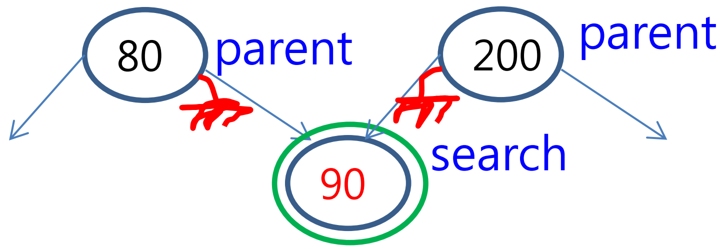
Note that
search == root 인 경우도 반드시 고려하여야 한다.



```
// 예: delete 30 from the tree
node *root;
…
if (deleteBST(&root, 30) == 0) {
    printf("The key does not exist in the tree!\n");
} else { // 1
    // yes! Deleted!
}
```

int deleteBST(node **rootPtr, int key);
이 함수를 우리는 개발하여야 한다.
나는 개발했음.

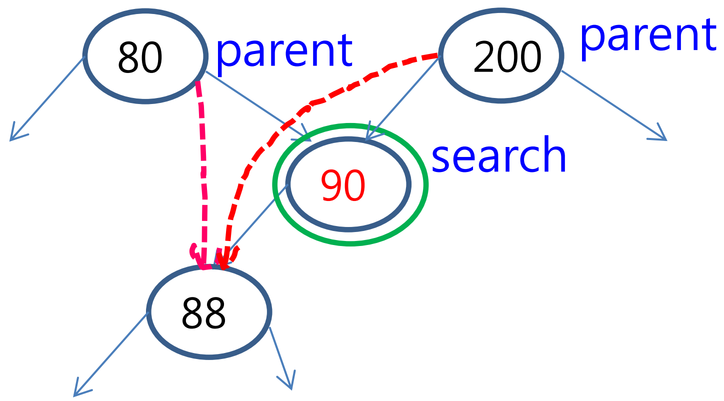
Delete 90

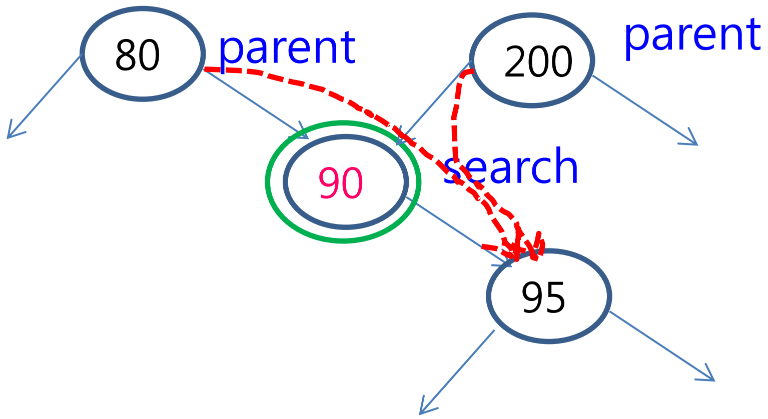


1. Deletion of a leaf node

(1) parent의 search에 대한 pointer를 NULL로 설정.

(2) free(search).





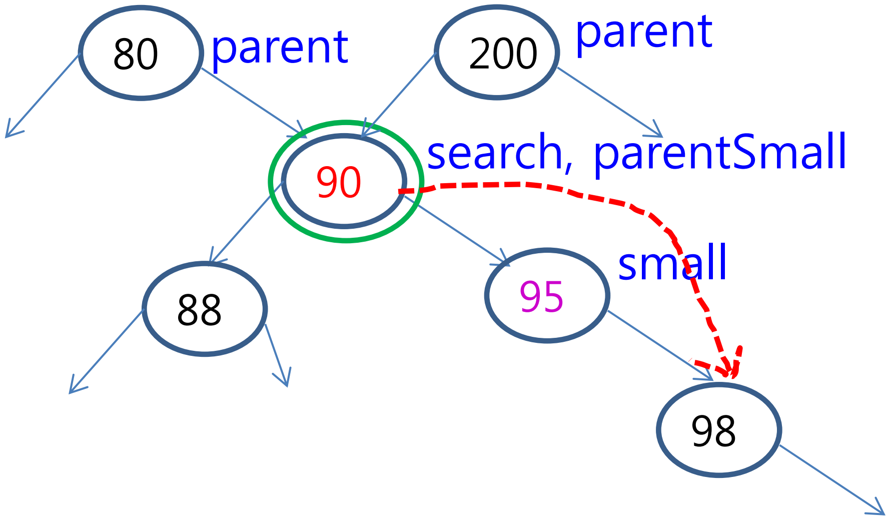
2. Deletion of a nonleaf node that has only one child

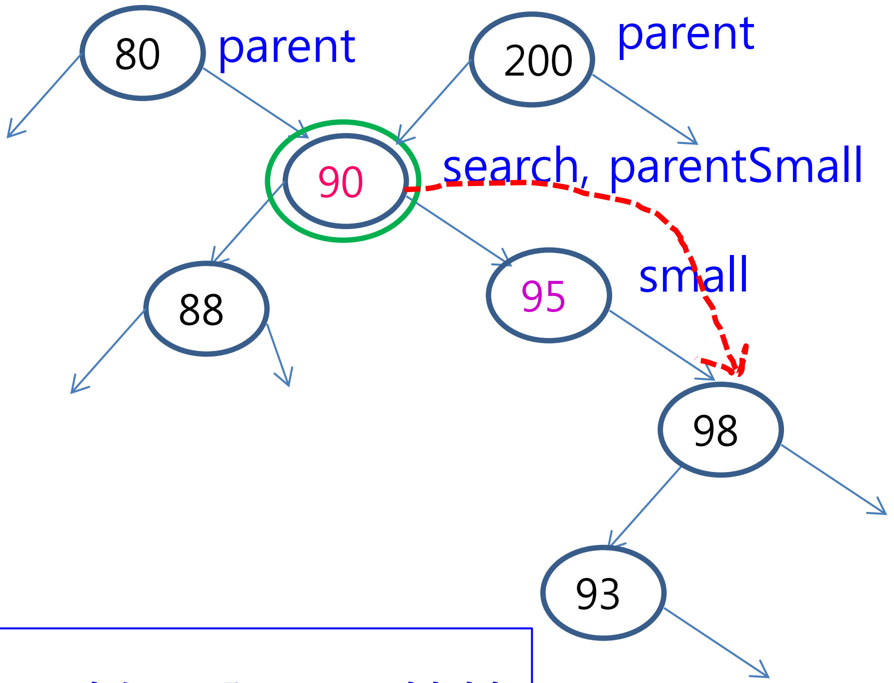
(1) parent의 search에 대한 pointer를 search의 single-child로 설정.

(2) free(search).

Delete 90

Case_1: search == parentSmall





3. Deletion of a nonleaf that has two children

삭제할 노드 search의 the smallest one in its right subtree를 small이라 하고 그 노드의 부모 노드를 parentSmall이라 하자.

(1) search의 data를 small의 data로 대체.

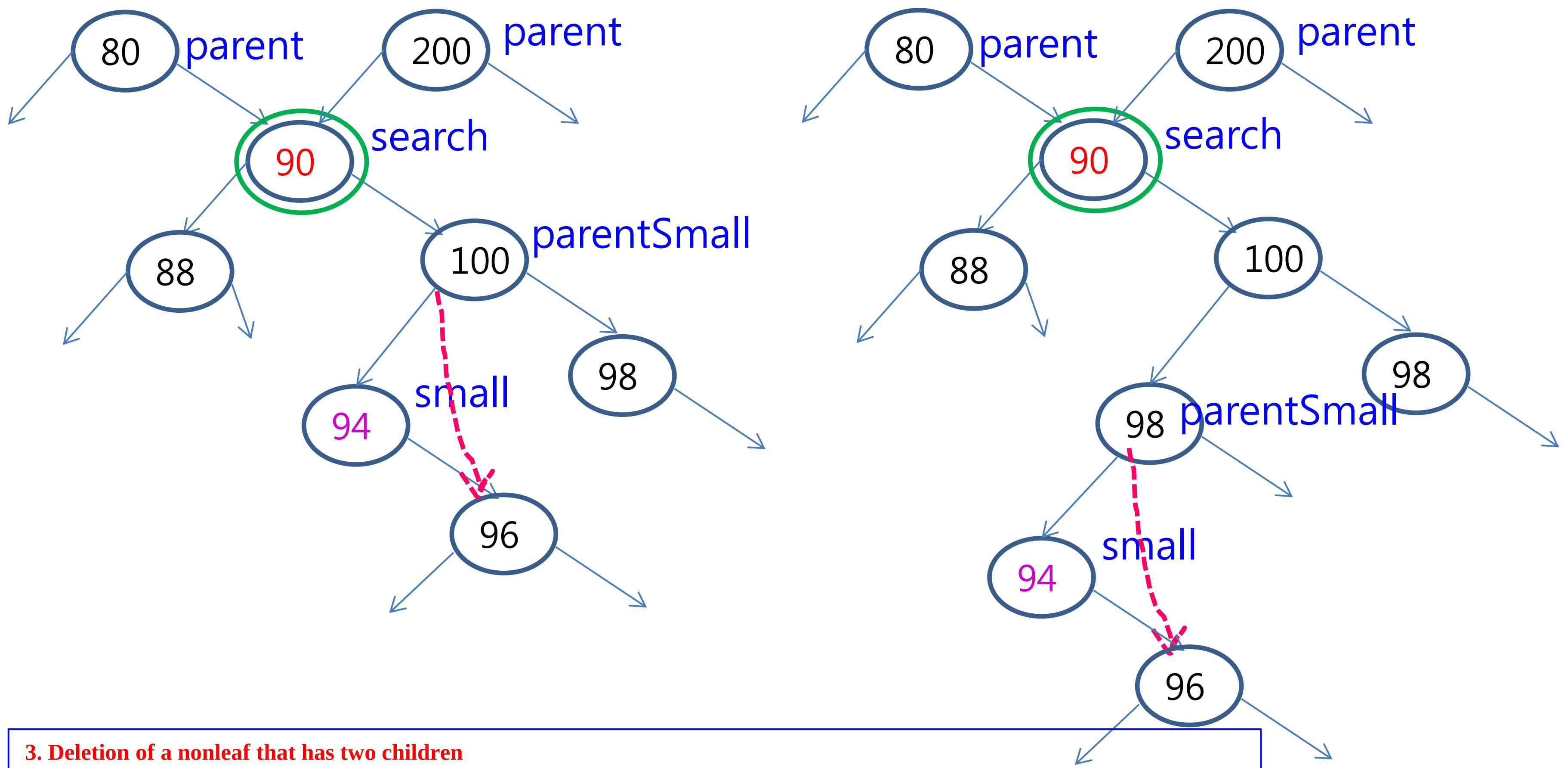
(2) If (search == parentSmall) parentSmall->rightChild = small->rightChild;

else parentSmall->leftChild = small->rightChild;

(3) free(small);

Delete 90

Case_2: search != parentSmall



3. Deletion of a nonleaf that has two children

삭제할 노드 search의 the smallest one in its right subtree를 small이라 하고 그 노드의 부모 노드를 parentSmall이라 하자.

(1) search의 data를 small의 data로 대체.

(2) If (search == parentSmall) parentSmall->rightChild = small->rightChild;

else parentSmall->leftChild = small->rightChild;

(3) free(small);

```
int deleteBST(node **rootPtr, int key)
{
    node *root, *parent, *search, *parentSmall, *searchSmall;
    root = *rootPtr;
    if (searchParentTargetToDelete(root, key, &parent, &search) == 0) {
        return(0);     // NOT FOUND
    }
    /* case 1. Deletion of a leaf node. */
    if ((search->leftChild == NULL) && (search->rightChild == NULL)) {



    }
    /* Case 2. Deletion of a nonleaf node that has only one child. */
    if ((search->leftChild == NULL) || (search->rightChild == NULL)) {



    }
    /* Case 3. Deletion of a nonleaf node that has two children. */
    if ((search->leftChild != NULL) && (search->rightChild != NULL)) {
        /* find the smallest among the larger than the key */
        searchSmallestAmongLargerNodes(search, key, &parentSmall, &searchSmall);



    }
}
```

```
/* tree에 key k가 없으면 0을 return.
   그렇지 않으면, k를 가진 노드의 주소를 *searchPtr,
   그 노드의 부모 노드의 주소를 *parentPtr에 보관한 후 1을 return한다.
 */
int searchParentTargetToDelete(node *root, int k, node **parentPtr, node **searchPtr)
{



}
```

```
/*
삭제할 노드 search의 the smallest one in its right subtree를 small이라 하고 그 노드의 부
모 노드를 parentSmall이라 하자.
 */
void searchSmallestAmongLargerNodes(node *target, int k, node **parentSmallPtr, nod
e **smallPtr)
{



}
```

5.7.5 Joining and Splitting Binary Trees

Although search, insert, and delete are the operations most frequently performed on a binary search tree, the following additional operations are useful in certain applications:

(a) *threeWayJoin* (*small*,*mid*,*big*): This creates a binary search tree consisting of the pairs initially in the binary search trees *small* and *big*, as well as the pair *mid*. It is assumed that each key in *small* is smaller than *mid*.*key* and that each key in *big* is greater than *mid*.*key*. Following the join, both *small* and *big* are empty.

(b) *twoWayJoin* (*small*,*big*): This joins the two binary search trees *small* and *big* to obtain a single binary search tree that contains all the pairs originally in *small* and *big*. It is assumed that all keys of *small* are smaller than all keys of *big* and that following the join both *small* and *big* are empty.

(c) *split* (*theTree*,*k*,*small*,*mid*,*big*): The binary search tree *theTree* is split into three parts: *small* is a binary search tree that contains all pairs of *theTree* that have key less than *k*; *mid* is the pair (if any) in *theTree* whose key is *k*; and *big* is a binary search tree that contains all pairs of *theTree* that have key larger than *k*. Following the split operation *theTree* is empty. When *theTree* has no pair whose key is *k*, *mid.key* is set to -1 (this assumes that -1 is not a valid key for a dictionary pair).

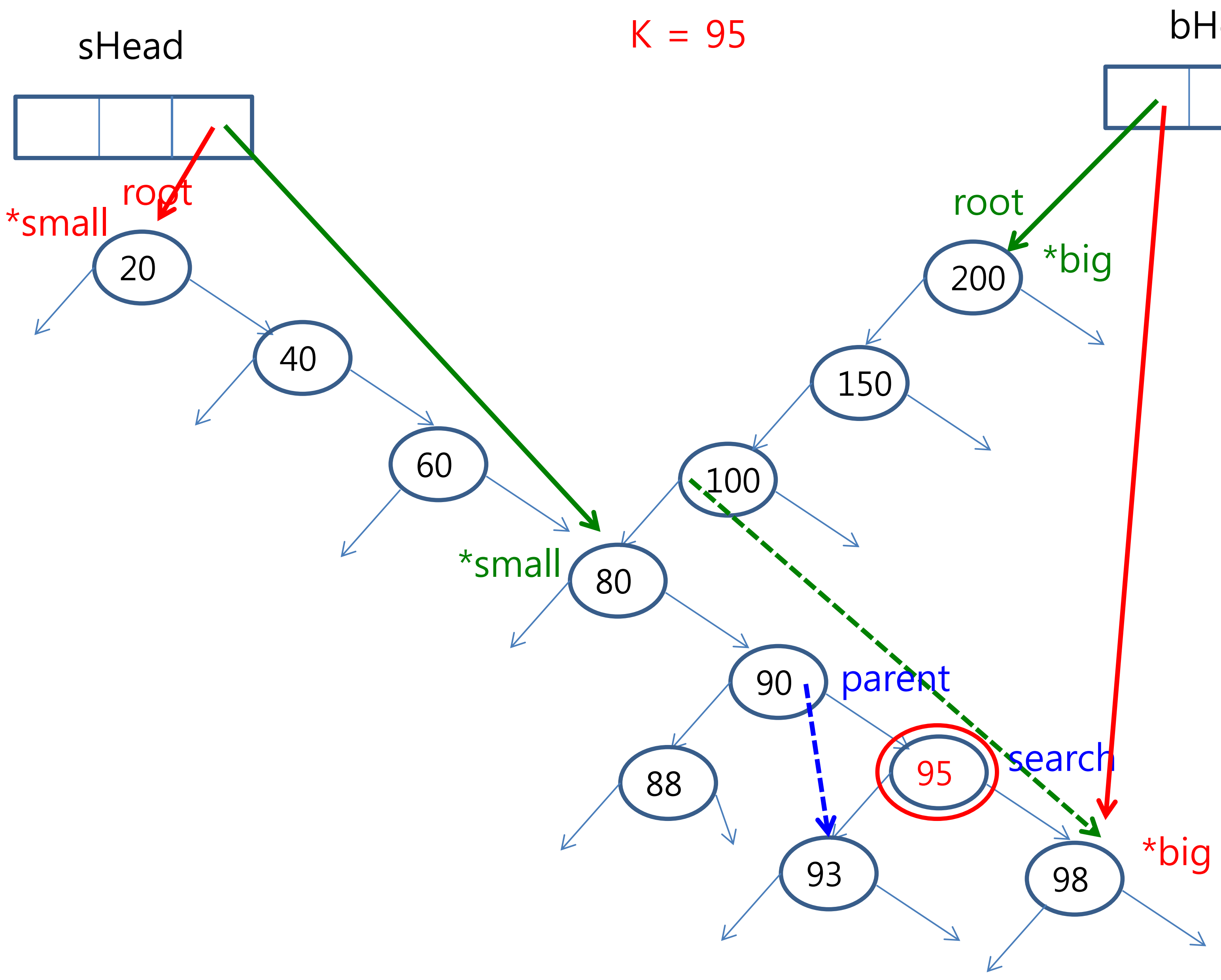
A **three-way join operation** is particularly easy to perform. We simply obtain a new node and set its data field to mid, its left-child pointer to small, and its right-child pointer to big. This new node is the root of the binary search tree that was to be created. Finally, small and big are set to NULL. The time taken for this operation is 0(1), and the height of the new tree is max {height (small), height (big)} + 1.

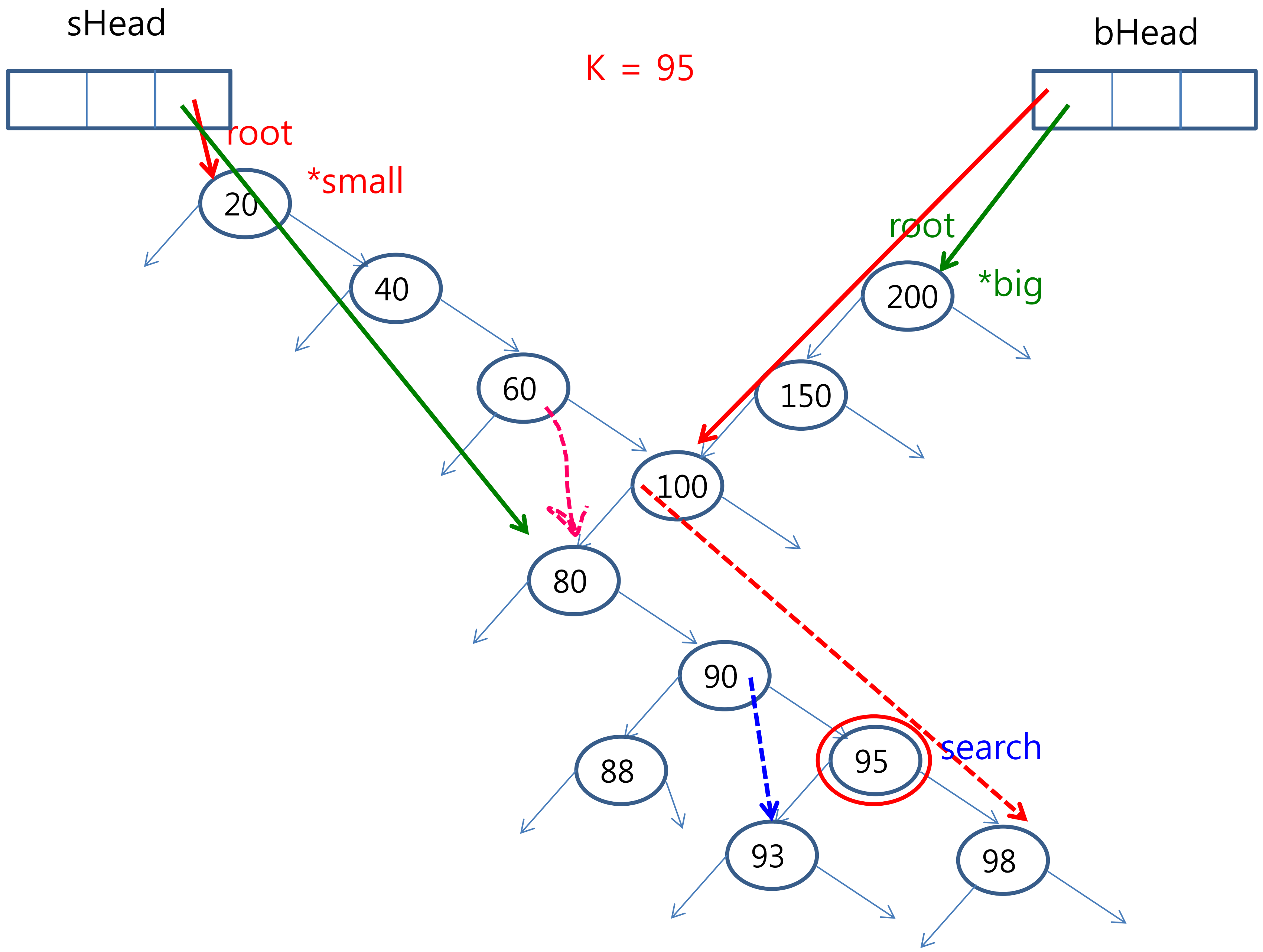
Consider **the two-way join operation**. If either small or big is empty, the result is the other tree. When neither is empty, we may first delete from small the pair mid with the largest key. Let the resulting binary search tree be small'. To complete the operation, we perform the three-way join operation threeWayJoin (small',mid,big). The overall time required to perform the two-way join operation is O(height (small)), and the height of the resulting tree is max {height (small'), height (big)} + 1. The run time can be made O(min {height (small), height (big)}) if we retain with each tree its height. Then we delete the pair with the largest key from small if the height of small is no more than that of big; otherwise, we delete from big the pair with the smallest key. This is followed by a three-way join operation.

To perform **a split**, we first make the following observation about splitting at the root (i.e., when k = theTree->data.key). In this case, small is the left subtree of theTree, mid is the pair in the root, and big is the right subtree of theTree. If k is smaller than the key at the root, then the root together with its right subtree is to be in big. When k is larger than the key at the root, the root together with its left subtree is to be in small. Using these observations, we can perform a split by moving down the search tree theTree searching for a pair with key k. As we move down, we construct the two search trees small and big. The function to split theTree is given in Program 5.18. To simplify the code, we begin with two header nodes sHead and bHead for small and big, respectively. small is grown as the right subtree of sHead; big is grown as the left subtree of bHead. S(b) points to the node of sHead (bHead) at which further subtrees of theTree that are to be part of small (big) may be attached. Attaching a subtree to small (big) is done as the right (left) child of s(b).

```
void split(nodePointer *theTree, int k, nodePointer *small, element *mid, nodePointer *big)
{
        /* split the binary search tree with respect to key k */
        if (! theTree) {*small = *big = 0; (*mid).key= -1; return; } /*empty tree*/
        nodePointer sHead, bHead, s, b, search;
        /* create header nodes for small and big */
        MALLOC(sHead, sizeof(*sHead));
        MALLOC(bHead, sizeof(*bHead));
        s = sHead; b = bHead;
        /* do the split */
        search = *theTree;
        while (search) {
                if (k < search->data.key) {/* add to big */
                        b->leftChild = search; b = search; search = search->leftChild;
                } else if (k > search->data.key) {/* add to small */
                        s->rightChild = search; s = search; search = search->rightChild;
                } else { // k == search->data.key
                        /* split at search */
                        s->rightChild = search->leftChild;
                        b->leftChild = search->rightChild;
                        *small= sHead->rightChild; free(sHead);
                        *big= bHead->leftChild; free(bHead);
                        (*mid).item = search->data.item;
                        (*mid).key = search->data.key;
                        free(search);
                        return;
                }
        }
        /* no pair with key k */
        s->rightChild = b->leftChild = 0;
        *small= sHead->rightChild; free(sHead);
        *big= bHead->leftChild; free(bHead);
        (*mid).key = -1;
        return;
}
```

Program 5.18 Splitting a binary search tree


sHead
K = 95
bHead
root
*small
root
*big
20
40
60
*small
80
200
150
100
90
parent
88
95
search
93
98
*big


sHead
bHead
K = 95
root
*small
root
*big
20
40
60
80
90
88
93
95
98
100
150
200
search

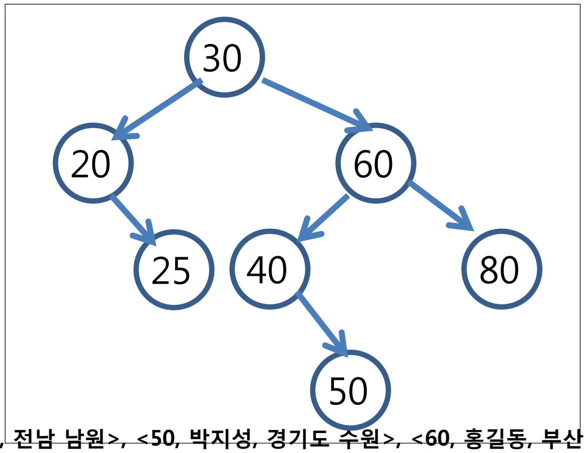
5.7.6 Height of a Binary Search Tree

Unless care is taken, the height of a binary search tree with *n* elements can become as large as *n*. This is the case, for instance, when Program 5.17 is used to insert the keys [1, 2, 3, . . . , *n*], in this order, into an initially empty binary search tree. It can, however, be shown that when insertions and deletions are made at random using the functions given here, the height of the binary search tree is $O(\log_2 n)$ on the average.

Search trees with a worst-case height of $O(\log n)$ are called ***balanced search trees***. Balanced search trees that permit searches, inserts, and deletes to be performed in $O(h)$ time exist. Most notable among these are AVL, red/black, 2-3, 2-3-4, B, and B^+ trees. These are discussed in Chapters 10 and 11.

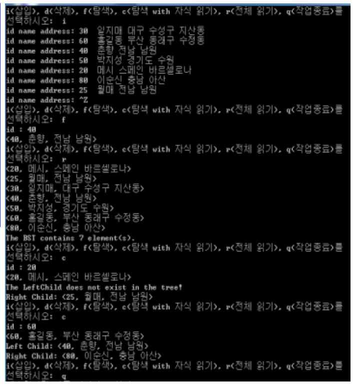
과제 1 (Binary Search Tree에서 삽입, 탐색)

동적 할당으로 생성되는 각 node는 필요한 모든 필드들을 다 가지는 structured data type을 가지도록 하여 binary search tree를 형성하도록 함으로써, 아래 기능을 수행하는 C 프로그램을 작성하시오. 단, 각 node는 삽입시에 동적으로 할당되어야 하며 <학번, 이름, 주소>를 가지며 학번으로 binary search tree의 특성을 유지한다.

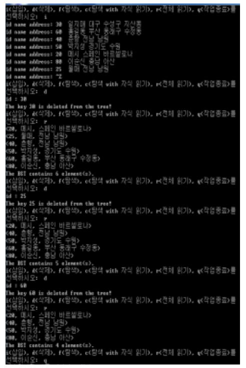


i(삽입), d(삭제), f(탐색), r(전체 읽기), c(자식들 읽기), q(작업종료)를 선택하시오: i
삽입할 자료(학번, 이름, 주소)를 입력하시오: 30 일지매 대구 수성구 지산동
삽입할 자료(학번, 이름, 주소)를 입력하시오: 60 홍길동 부산 동래구 수정동
삽입할 자료(학번, 이름, 주소)를 입력하시오: 40 춘향 전남 남원
삽입할 자료(학번, 이름, 주소)를 입력하시오: 50 박지성 경기도 수원
삽입할 자료(학번, 이름, 주소)를 입력하시오: 20 메시 스페인 바르셀로나
삽입할 자료(학번, 이름, 주소)를 입력하시오: 80 이순신 충남 아산
삽입할 자료(학번, 이름, 주소)를 입력하시오: 25 월매 전남 남원
삽입할 자료(학번, 이름, 주소)를 입력하시오: ^z
i(삽입), d(삭제), f(탐색), r(전체 읽기), c(자식들 읽기), q(작업종료)를 선택하시오: f
탐색할 자료의 학번을 입력하시오: 40
<학번, 이름, 주소> = <40, 춘향, 전남 남원>
i(삽입), d(삭제), f(탐색), r(전체 읽기), c(자식들 읽기), q(작업종료)를 선택하시오: r
<학번, 이름, 주소> : <20, 메시, 스페인 바르셀로나>, <25, 월매, 전남 남원>, <30, 일지매, 대구 수성구 지산동>, <40, 춘향, 전남 남원>, <50, 박지성, 경기도 수원>, <60, 홍길동, 부산 동래구 수정동>, <80, 이순신, 충남 아산>
i(삽입), d(삭제), f(탐색), r(전체 읽기), c(자식들 읽기), q(작업종료)를 선택하시오: c
탐색할 자료의 학번을 입력하시오: 20
<학번, 이름, 주소> = < 20, 메시, 스페인 바르셀로나>
왼쪽 자식: 없음
오른쪽 자식: <25, 월매, 전남 남원>
i(삽입), d(삭제), f(탐색), r(전체 읽기), c(자식들 읽기), q(작업종료)를 선택하시오: c
탐색할 자료의 학번을 입력하시오: 60
<학번, 이름, 주소> = <60, 홍길동, 부산 동래구 수정동>
왼쪽 자식: <40, 춘향, 전남 남원>
오른쪽 자식: <80, 이순신, 충남 아산>
i(삽입), d(삭제), f(탐색), r(전체 읽기), c(자식들 읽기), q(작업종료)를 선택하시오: d
삭제할 자료의 학번을 입력하시오: 30
i(삽입), d(삭제), f(탐색), r(전체 읽기), c(자식들 읽기), q(작업종료)를 선택하시오: r
<학번, 이름, 주소> : <20, 메시, 스페인 바르셀로나>, <25, 월매, 전남 남원>, <40, 춘향, 전남 남원>, <50, 박지성, 경기도 수원>, <60, 홍길동, 부산 동래구 수정동>, <80, 이순신, 충남 아산>
i(삽입), d(삭제), f(탐색), r(전체 읽기), c(자식들 읽기), q(작업종료)를 선택하시오: d
삭제할 자료의 학번을 입력하시오: 25
i(삽입), d(삭제), f(탐색), r(전체 읽기), c(자식들 읽기), q(작업종료)를 선택하시오: r
<학번, 이름, 주소> : <20, 메시, 스페인 바르셀로나>, <40, 춘향, 전남 남원>, <50, 박지성, 경기도 수원>, <60, 홍길동, 부산 동래구 수정동>, <80, 이순신, 충남 아산>
i(삽입), d(삭제), f(탐색), r(전체 읽기), c(자식들 읽기), q(작업종료)를 선택하시오: d
삭제할 자료의 학번을 입력하시오: 60
i(삽입), d(삭제), f(탐색), r(전체 읽기), c(자식들 읽기), q(작업종료)를 선택하시오: r
<학번, 이름, 주소> : <20, 메시, 스페인 바르셀로나>, <40, 춘향, 전남 남원>, <50, 박지성, 경기도 수원>, <80, 이순신, 충남 아산>
i(삽입), d(삭제), f(탐색), r(전체 읽기), c(자식들 읽기), q(작업종료)를 선택하시오: d
삭제할 자료의 학번을 입력하시오: 20
i(삽입), d(삭제), f(탐색), r(전체 읽기), c(자식들 읽기), q(작업종료)를 선택하시오: r
<학번, 이름, 주소> : <40, 춘향, 전남 남원>, <50, 박지성, 경기도 수원>, <80, 이순신, 충남 아산>
i(삽입), d(삭제), f(탐색), r(전체 읽기), f(작업종료)를 선택하시오: q

```
30  일지매 대구 수성구 지산동
60  홍길동 부산 동래구 수정동
40  춘향 전남 남원
50  박지성 경기도 수원
20  메시 스페인 바르셀로나
80  이순신 충남 아산
25  월매 전남 남원
```



```
i(삽입), d(삭제), f(탐색), c(탐색 with 자식 읽기), r(전체 읽기), q(작업종료)를
선택하시오:  i
id name address: 30  일지매 대구 수성구 지산동
id name address: 60  홍길동 부산 동래구 수정동
id name address: 40  춘향 전남 남원
id name address: 50  박지성 경기도 수원
id name address: 20  메시 스페인 바르셀로나
id name address: 80  이순신 충남 아산
id name address: 25  월매 전남 남원
id name address: ^Z
i(삽입), d(삭제), f(탐색), c(탐색 with 자식 읽기), r(전체 읽기), q(작업종료)를
선택하시오:  f
id : 40
<40, 춘향, 전남 남원>
i(삽입), d(삭제), f(탐색), c(탐색 with 자식 읽기), r(전체 읽기), q(작업종료)를
선택하시오:  r
<20, 메시, 스페인 바르셀로나>
<25, 월매, 전남 남원>
<30, 일지매, 대구 수성구 지산동>
<40, 춘향, 전남 남원>
<50, 박지성, 경기도 수원>
<60, 홍길동, 부산 동래구 수정동>
<80, 이순신, 충남 아산>
The BST contains 7 element(s).
i(삽입), d(삭제), f(탐색), c(탐색 with 자식 읽기), r(전체 읽기), q(작업종료)를
선택하시오:  c
id : 20
<20, 메시, 스페인 바르셀로나>
The LeftChild does not exist in the tree!
Right Child: <25, 월매, 전남 남원>
i(삽입), d(삭제), f(탐색), c(탐색 with 자식 읽기), r(전체 읽기), q(작업종료)를
선택하시오:  c
id : 60
<60, 홍길동, 부산 동래구 수정동>
Left Child: <40, 춘향, 전남 남원>
Right Child: <80, 이순신, 충남 아산>
i(삽입), d(삭제), f(탐색), c(탐색 with 자식 읽기), r(전체 읽기), q(작업종료)를
선택하시오:  q
```



```
i(삽입), d(삭제), f(탐색), c(탐색 with 자식 읽기), r(전체 읽기), q(작업종료)를
선택하시오:  i
id name address: 30  일지매 대구 수성구 지산동
id name address: 60  홍길동 부산 동래구 수정동
id name address: 40  춘향 전남 남원
id name address: 50  박지성 경기도 수원
id name address: 20  메시 스페인 바르셀로나
id name address: 80  이순신 충남 아산
id name address: 25  월매 전남 남원
id name address: ^Z
i(삽입), d(삭제), f(탐색), c(탐색 with 자식 읽기), r(전체 읽기), q(작업종료)를
선택하시오:  d
id : 30
The key 30 is deleted from the tree!
i(삽입), d(삭제), f(탐색), c(탐색 with 자식 읽기), r(전체 읽기), q(작업종료)를
선택하시오:  r
<20, 메시, 스페인 바르셀로나>
<25, 월매, 전남 남원>
<40, 춘향, 전남 남원>
<50, 박지성, 경기도 수원>
<60, 홍길동, 부산 동래구 수정동>
<80, 이순신, 충남 아산>
The BST contains 6 element(s).
i(삽입), d(삭제), f(탐색), c(탐색 with 자식 읽기), r(전체 읽기), q(작업종료)를
선택하시오:  d
id : 25
The key 25 is deleted from the tree!
i(삽입), d(삭제), f(탐색), c(탐색 with 자식 읽기), r(전체 읽기), q(작업종료)를
선택하시오:  r
<20, 메시, 스페인 바르셀로나>
<40, 춘향, 전남 남원>
<50, 박지성, 경기도 수원>
<60, 홍길동, 부산 동래구 수정동>
<80, 이순신, 충남 아산>
The BST contains 5 element(s).
i(삽입), d(삭제), f(탐색), c(탐색 with 자식 읽기), r(전체 읽기), q(작업종료)를
선택하시오:  d
id : 60
The key 60 is deleted from the tree!
i(삽입), d(삭제), f(탐색), c(탐색 with 자식 읽기), r(전체 읽기), q(작업종료)를
선택하시오:  r
<20, 메시, 스페인 바르셀로나>
<40, 춘향, 전남 남원>
<50, 박지성, 경기도 수원>
<80, 이순신, 충남 아산>
The BST contains 4 element(s).
i(삽입), d(삭제), f(탐색), c(탐색 with 자식 읽기), r(전체 읽기), q(작업종료)를
선택하시오:  q
```

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX_ARRAY_SIZE 100

char data[MAX_ARRAY_SIZE];
typedef struct {
    int key;
    char name[20];
    char address[100];
} element;
typedef struct node *nodePointer;
typedef struct node {
    element           data;
    nodePointer     leftChild;
    nodePointer     rightChild;
} node;
nodePointer   root = NULL;

…

int main()
{
    char selection;

    while (1) {
        printf("i(삽입), d(삭제), f(탐색), c(탐색 with 자식 읽기), r(전체 읽기), q(작업
종료)를 선택하시오: ");
        fscanf(stdin, "%c", &selection);
        fflush(stdin);
        switch (selection) {
            case 'i' : insertion(); break;
            case 'd' : deletion(); break;
            case 'f' : find(); break;
            case 'c' : findChildren(); break;
            case 'r' : printList(); break;
            case 'q' : return(0);
        }
    }
}
```

5.8 SELECTION TREES

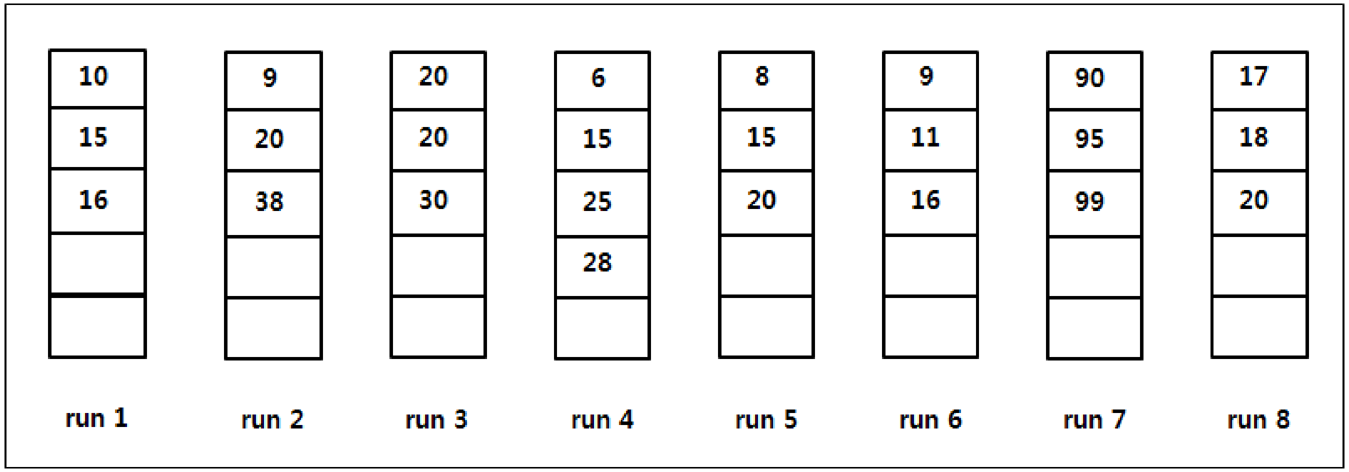
5.8.1 Introduction

아래 그림에 *k* (8) 개의 run (ordered sequence)들이 있다.

그들을 merge하여 하나의 run으로 만들고자 한다. 어떻게 하면 되는가?

단, 각 run은 *key*의 nondecreasing order로 정렬되어 있으며 전체 *k* 개의 run들에 있는 레코드들의 수는 *n* 이다.

The most direct way to merge *k* runs is to make k – 1 comparisons to determine the next record to output. For k > 2, we can achieve a reduction in the number of comparisons needed to **find the next smallest element** by using the ***selection tree*** data structure. There are two kinds of selection trees: ***winner trees*** and ***loser trees***.



5.8.2 Winner Trees

A ***winner tree*** is a complete binary tree in which each node represents the smaller of its two children. Thus, the root node represents the smallest node in the tree. Figure 5.32 illustrates a winner tree for the case k = 8.

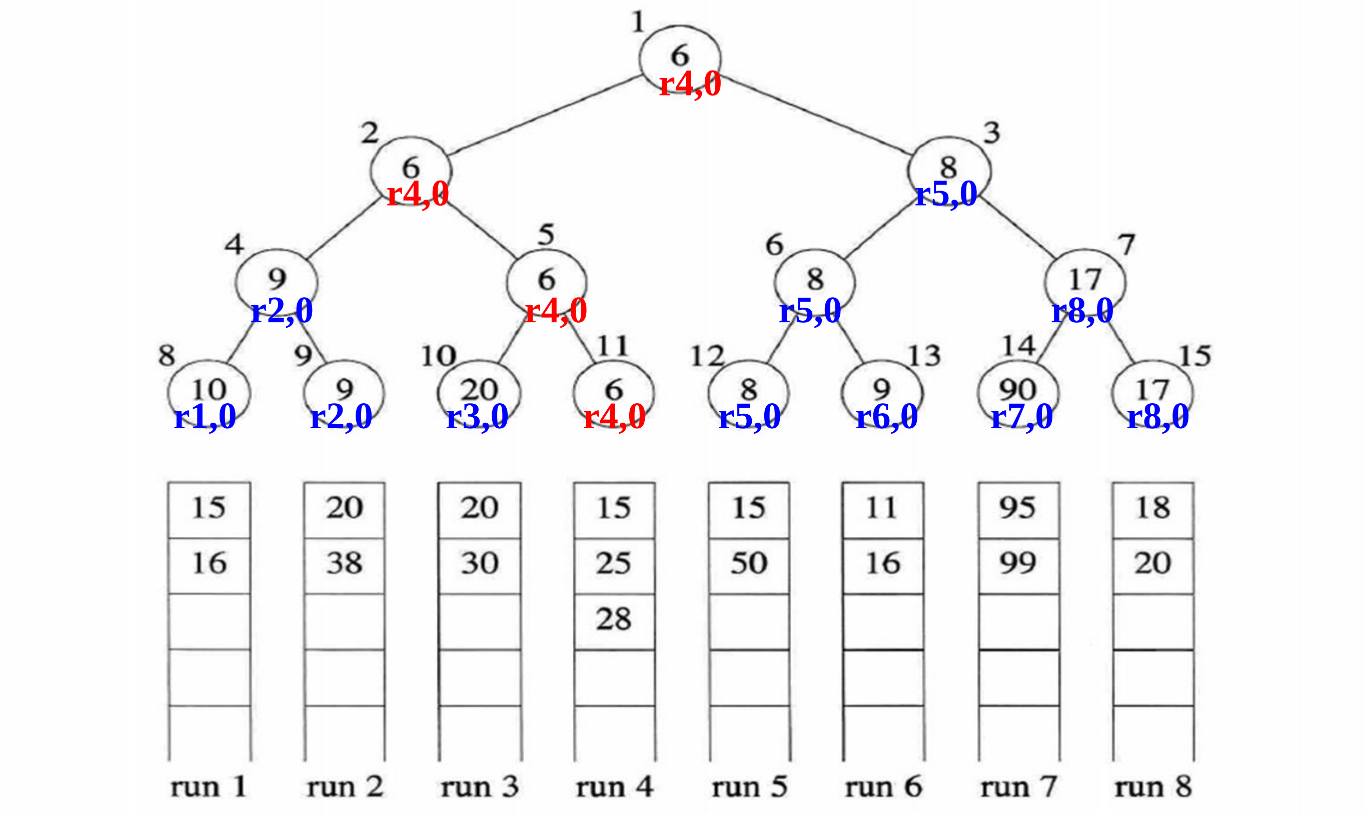
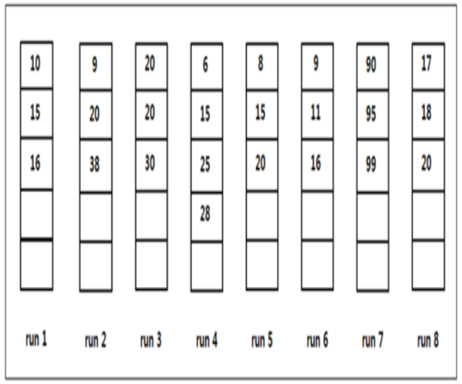


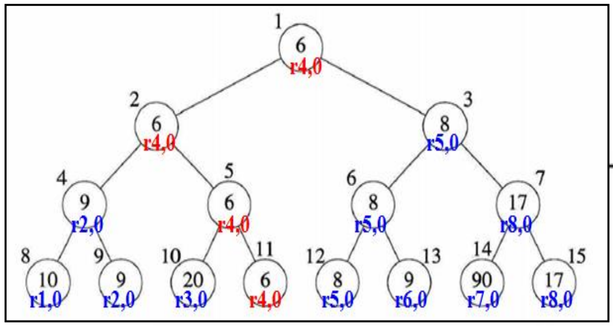
Figure 5.32: Winner tree for $k = 8$, showing the first three keys in each of the eight runs



The construction of this winner tree may be compared to the playing of a **tournament** in which the winner is the record with the smaller key. Then, each nonleaf node in the tree represents the winner of a tournament, and the root node represents the overall winner, or the smallest key. Each leaf node represents the first record in the corresponding run. Since the records being merged are generally large, each node will contain only a pointer to the record it represents. Thus, the root node contains a pointer to the first record in run 4.

A winner tree may be represented using the sequential allocation scheme for binary trees that results from Lemma 5.4. The number above each node in Figure 5.32 is the address of the node in this **sequential representation**. The record pointed to by the root has the smallest key and so may be output. Now, the next record from run 4 enters the winner tree. It has a key value of 15. To restructure the tree, the tournament has to be replayed only along the path from node 11 to the root. Thus, the winner from nodes 10 and 11 is again node 11 ($15 < 20$). The winner from nodes 4 and 5 is node 4 ($9 < 15$). The winner from 2 and 3 is node 3 ($8 < 9$). The new tree is shown in Figure 5.33. The tournament is played between sibling nodes and the result is put in the parent node. Lemma 5.4 may be used to compute the address of sibling and parent nodes efficiently. Each new comparison takes place at the next higher level in the tree.

Analysis of merging runs using winner trees: The number of levels in the tree is $\lceil \log_2(k+1) \rceil$. So, the time to restructure the tree is $O(\log_2 k)$. The tree has to be restructured each time a record is merged into the output file. Hence, the time required to merge all n records is $O(n \log_2 k)$. The time required to set up the selection tree the first time is $O(k)$. Thus, the total time needed to merge the k runs is $\mathbf{O(n \log_2 k)}$. □



the winner tree of Figure 5.32

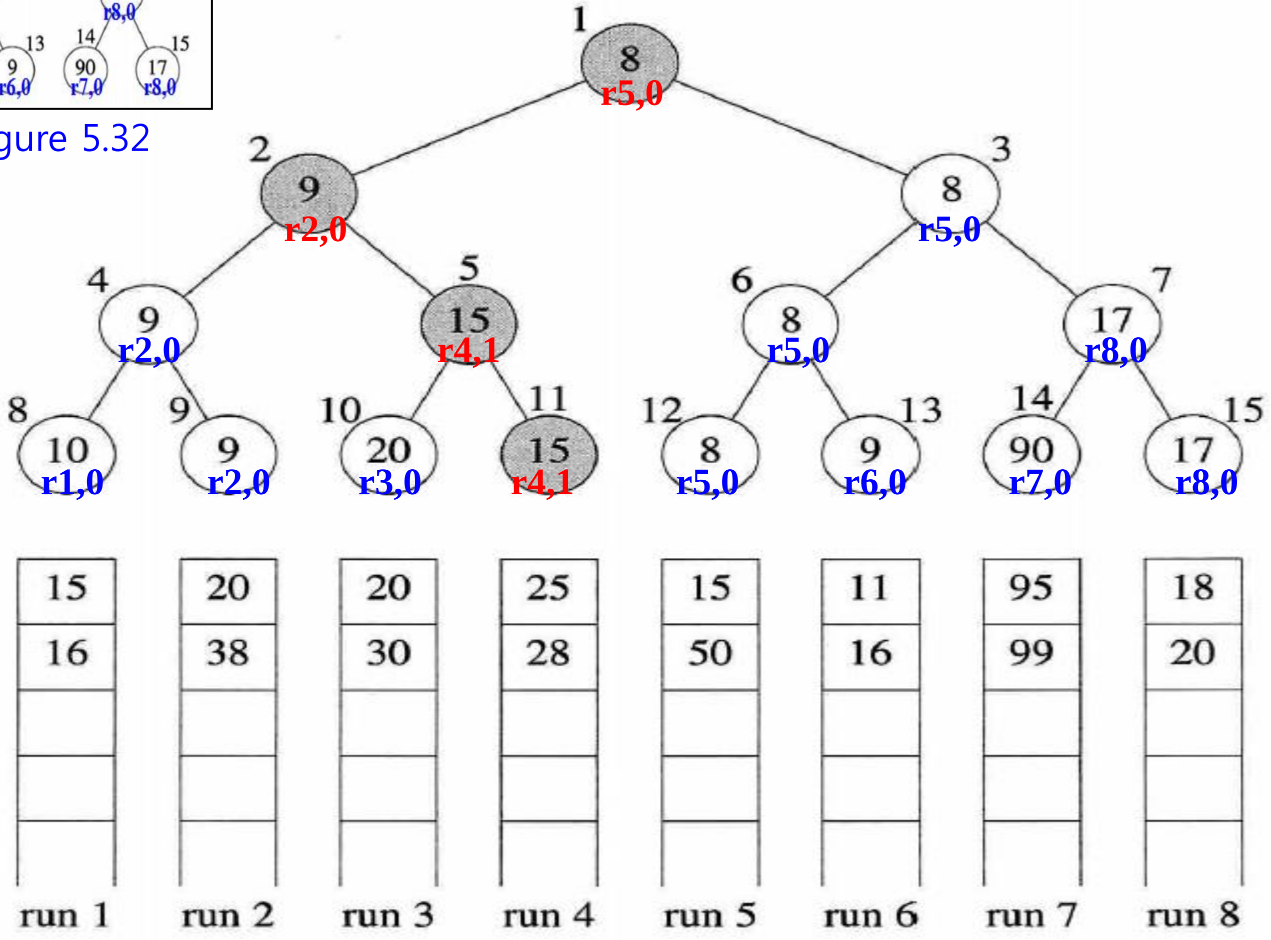
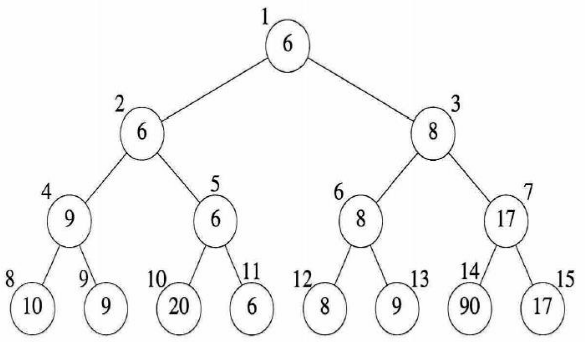


Figure 5.33: Winner tree of Figure 5.32 after one record has been output and the tree restructured (nodes that were changed are shaded)

5.8.3 Loser Trees

After the record with the smallest key value is output, the winner tree of Figure 5.32 is to be restructured. Since the record with the smallest key value is in run 4, this restructuring involves inserting the next record from this run into the tree. The next record has key value 15. Tournaments are played between sibling nodes along the path from node 11 to the root. Since these sibling nodes represent the losers of tournaments played earlier, we can simplify the restructuring process by placing in each nonleaf node a pointer to the record that loses the tournament rather than to the winner of the tournament. A selection tree in which each nonleaf node retains a pointer to the loser is called ***a loser tree***. Figure 5.34 shows the loser tree that corresponds to the winner tree of Figure 5.32. For convenience, each node contains the key value of a record rather than a pointer to the record represented. The leaf nodes represent the first record in each run. An additional node, node 0, has been added to represent the overall winner of the tournament. Following the output of the overall winner, the tree is restructured by playing tournaments along the path from node 11 to node 1. The records with which these tournaments are to be played are readily available from the parent nodes. As a result, sibling nodes along the path from 11 to 1 are not accessed.



the winner tree of Figure 5.32

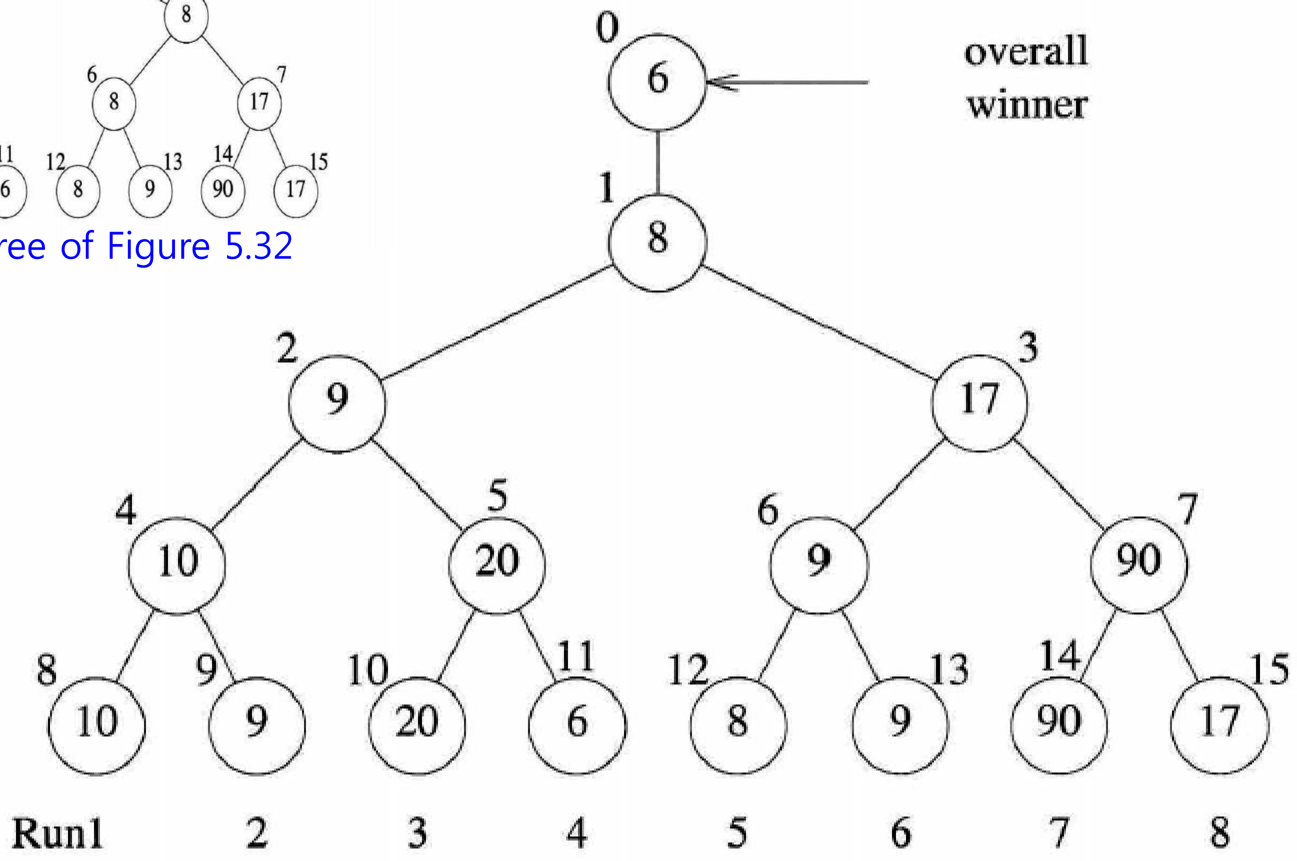


Figure 5.34: Loser tree corresponding to winner tree of Figure 5.32

과제 4 (Winner Tree)

아래 작업을 수행하는 C 프로그램을 작성하시오.

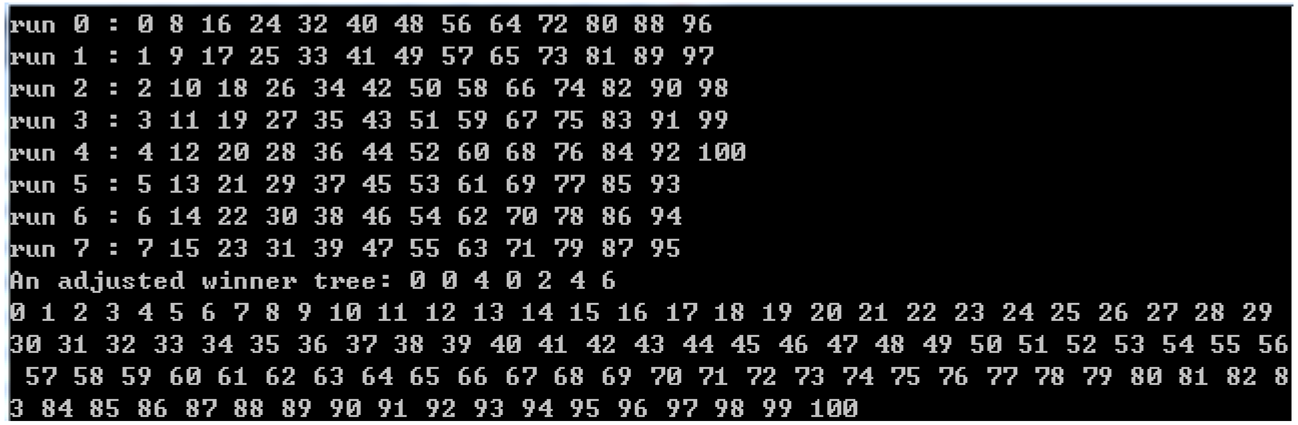
Figure 5.32의 Winner Tree를 array를 이용하여 8-way merge sort를 수행하고자 한다. 아래 사항들을 수행하시오.

(1) 8개의 run을 다음과 같이 작성하시오. 각 run을 하나의 배열로 수행한다. 이들 배열은 서로 다른 배열명을 가져야 한다. Run1 부터 run 8 까지 차례로 다음과 같이 레코드들을 각 run에 차례로 insert 한 후 각 run에 보관된 값들을 출력하시오. [1점]

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**
16	**17**	**18**	**19**	**20**	**21**	**22**	**23**
31	**30**	**29**	**28**	**27**	**26**	**25**	**24**
32	**33**						

의 방법으로 100 까지 각 run에 배치하시오.

(2) 그 run들을 대상으로 Figure 5.32의 Winner Tree를 array를 이용하여 8-way merge sort를 수행하고자 한다. 그 결과를 배열 result에 넣으시오. 정렬이 모두 끝난 후 배열 result의 내용을 출력하시오. [3점]



```
run 0 : 0 8 16 24 32 40 48 56 64 72 80 88 96
run 1 : 1 9 17 25 33 41 49 57 65 73 81 89 97
run 2 : 2 10 18 26 34 42 50 58 66 74 82 90 98
run 3 : 3 11 19 27 35 43 51 59 67 75 83 91 99
run 4 : 4 12 20 28 36 44 52 60 68 76 84 92 100
run 5 : 5 13 21 29 37 45 53 61 69 77 85 93
run 6 : 6 14 22 30 38 46 54 62 70 78 86 94
run 7 : 7 15 23 31 39 47 55 63 71 79 87 95
An adjusted winner tree: 0 0 4 0 2 4 6
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 8
3 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
```

5.9 FORESTS

Definition: A ***forest*** is a set of n ≥ 0 disjoint trees. □

A three-tree forest is shown in Figure 5.35. The concept of a forest is very close to that of a tree because if we remove the root of a tree, we obtain a forest. For example, removing the root of any binary tree produces a forest of two trees. In this section, we briefly consider several forest operations, including transforming a forest into a binary tree and forest traversals. In the next section, we use forests to represent disjoint sets.

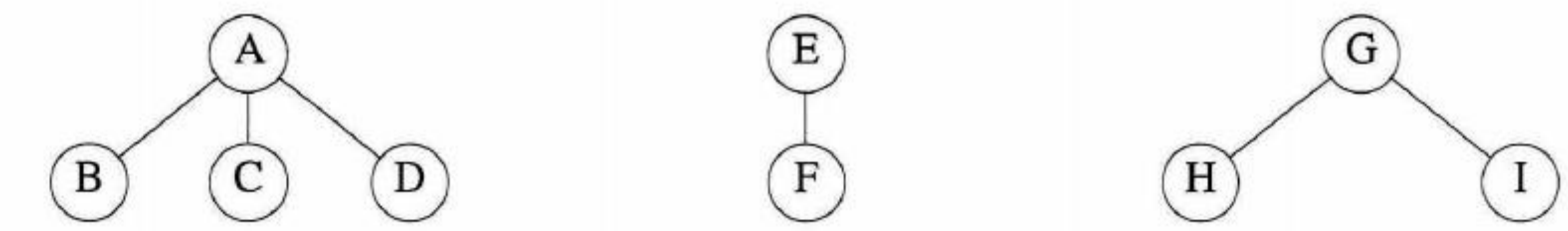


Figure 5.35: Three-tree forest

5.9.1 Transforming a Forest into a Binary Tree

To transform a forest into a single binary tree, we first obtain the binary tree representation of each of the trees in the forest and then link these binary trees together through the *rightChild* field of the root nodes. Using this transformation, the forest of Figure 5.35 becomes the binary tree of Figure 5.36.

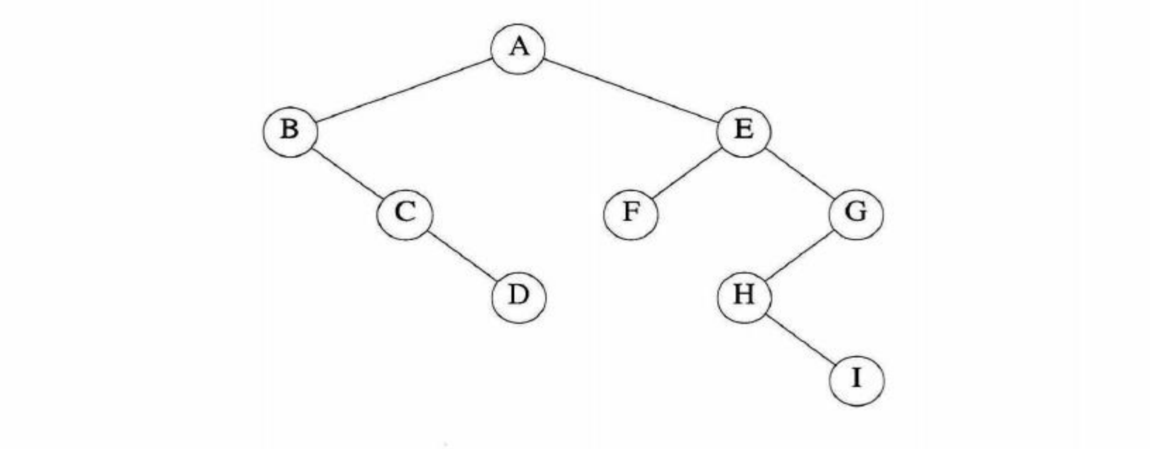


Figure 5.36: Binary tree representation of forest of Figure 5.35

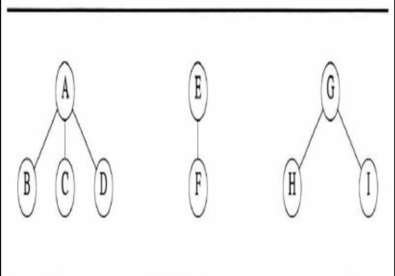


Figure 5.35: Three-tree forest

We can define this transformation in a formal way as follows:

Definition: If $T_1, \cdots, T_n$ is a forest of trees, then the binary tree corresponding to this forest, denoted by $B(T_1, \ldots, T_n)$,

(1) is empty if $n = 0$

(2) has root equal to root (T_1); has left subtree equal to $B(T_{11}, T_{12}, \cdots, T_{1m})$, where $T_{11}, T_{12}, \cdots, T_{1m}$ are the subtrees of root(T_1); and has right subtree $B(T_2, \ldots, T_n)$. □

5.9.2 Forest Traversals

Preorder and inorder traversals of the corresponding binary tree T of a forest F have a natural correspondence to traversals on F. Preorder traversal of T is equivalent to visiting the nodes of F in ***forest preorder***, which is defined as follows:

(1) If F is empty then return.

(2) Visit the root of the first tree of F.

(3) Traverse the subtrees of the first tree in forest preorder.

(4) Traverse the remaining trees of F in forest preorder.

Inorder traversal of T is equivalent to visiting the nodes of F in ***forest inorder***, which is defined as follows:

(1) If F is empty then return.

(2) Traverse the subtrees of the first tree in forest inorder.

(3) Visit the root of the first tree.

(4) Traverse the remaining trees in forest inorder.

The proofs that preorder and inorder traversals on the corresponding binary tree are the same as preorder and inorder traversals on the forest are left as exercises. There is no natural analog for postorder traversal of the corresponding binary tree of a forest. Nevertheless, we can define the ***postorder traversal of a forest*** as follows:

(1) If F is empty then return.

(2) Traverse the subtrees of the first tree of F in forest postorder.

(3) Traverse the remaining trees of F in forest postorder.

(4) Visit the root of the first tree of F.

In a ***level-order traversal of a forest***, nodes are visited by level, beginning with the roots of each tree in the forest. Within each level, nodes are visited from left to right. One may verify that the level-order traversal of a forest and that of its associated binary tree do not necessarily yield the same result.