Chapter 5 : Trees

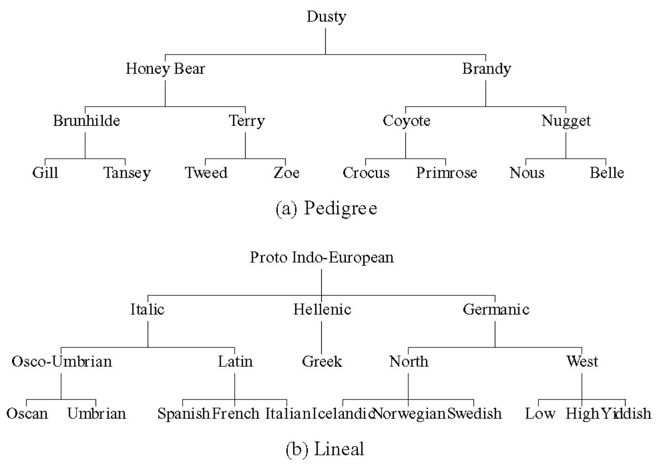
Data Structures Lecture Note

Prof. Sungwon Jung
Big Data Processing Laboratory
Dept. of Computer Science and Engineering
Sogang University

5.1 Introduction

5.1.1 Terminology

- The intuitive concept of a *tree* implies that we organize the data so that items of information are related by the branches.
- [Figure 5.1] Two types of genealogical charts



Tree

- A *tree* is a finite set of one or more nodes such that:

  (1) There is a specially designated node called the *root*.

  (2) The remaining nodes are partitioned into $n \geq 0$ disjoint sets T_1, T_2, … , T_n where each of these sets is a tree. We call T_1, T_2, … , T_n the subtrees of the root.

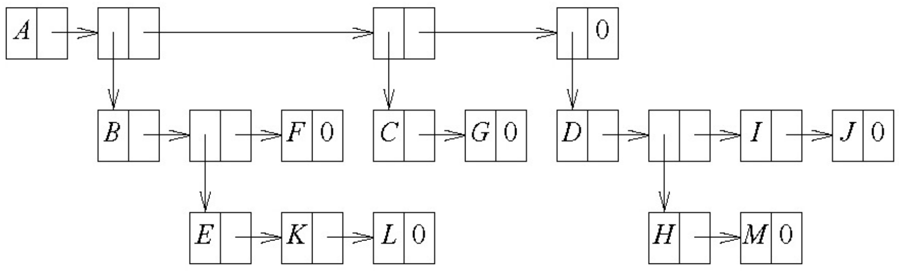
- A *node* stands for the item of information and the branches to other nodes.
- The *degree* of a node is the number of subtrees of the node.
- The *degree of a tree* is the maximum degree of the nodes in the tree.
- A node with degree zero is a *leaf* or *terminal* node.

Tree

- A node that has subtrees is the *parent* of the roots of the subtrees, and the roots of the subtrees are the *children* of the node.
- Children of the same parent are *siblings*.
- The *ancestors* of a node are all the nodes along the path from the root to the node.
- Conversely, the *descendants* of a node are all the nodes that are in its subtrees.
- The *level* of a node is defined by : Initially letting the root be at level one.
- For all subsequent nodes, the level is the level of the node's parent plus one.
- The *height* or *depth* of a tree is the maximum level of any node in the tree.

5.1.2 Representation Of Trees

- **List Representation**
- Representing a tree as a list in which each of the subtrees is also a list.
- For example, the tree of Figure 5.2 is written as : (A(B(E(K,L),F),C(G),D(H(M),I,J)))
- [Figure 5.2]



5.1.2 Representation Of Trees

- If we wish to use linked lists, then a node must have a varying number of fields depending on the number of branches.
- [Figure 5.3] a possible representation for trees

data	*link 1*	*link 2*	. . .	*link n*

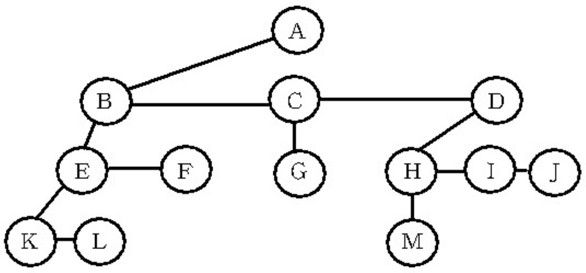
- It is often easier to work with nodes of a fixed size.

Left Child-Right Sibling Representation

- The representations we consider require exactly two link or pointer fields per node.
- Note that every node has only one leftmost child and one closest right sibling. (* Strictly speaking, the order of children in a tree is not important. *)

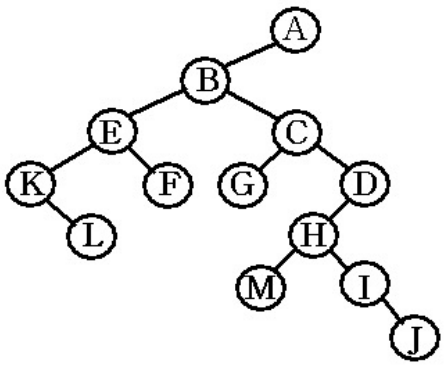
data	
left child	right sibling

[Figure 5.4] structure of node

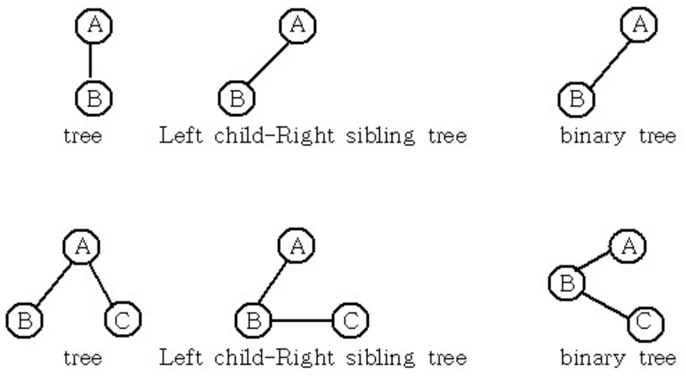


[Figure 5.5] tree

Representation As A Degree Two Tree



[Figure 5.6] binary tree



[Figure 5.7] representation of tree

5.2 Binary trees

- **5.2.1 The Abstract Data Type**
- The chief characteristic of a binary tree is the stipulation that the degree of any given node must not exceed two.
- For binary trees, we distinguish between the left subtree and the right subtree, while for trees the order of the subtrees is irrelevant.

- *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.

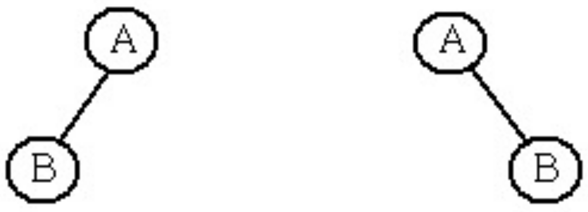
5.2 Binary trees

- [Distinction between a binary tree and a tree]

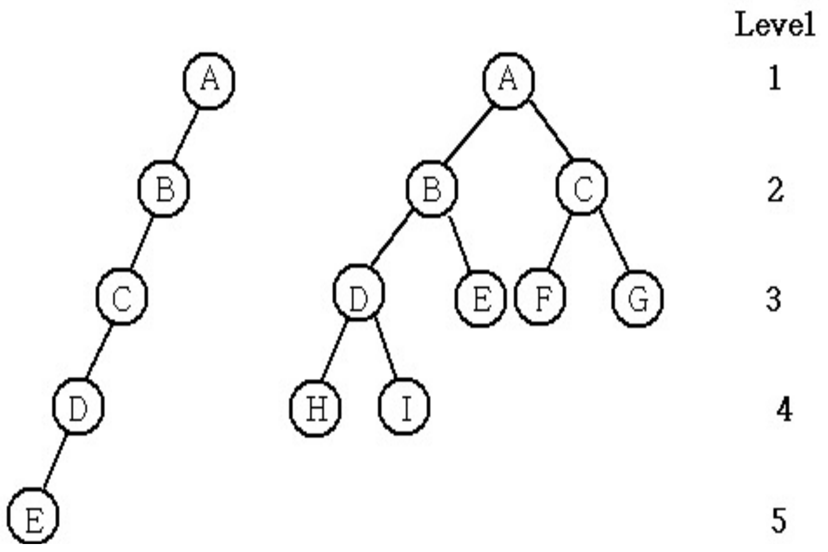
(1) There is an empty binary tree.

(2) In a binary tree, we distinguish between the order of the children while in a tree we do not.

- [Figure 5.8] Two different binary trees



Special types of binary trees



[Figure 5.9] (a) skewed tree (b) complete binary tree

Structure 5.1 : Abstract data type *Binary_Tree*.

- **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* ∈ *BinTree*, *item* ∈ *element*

BinTree Create() ::= creates an empty binary tree

Boolean IsEmpty(*bt*) ::= if (*bt* == empty binary tree) return *TRUE*

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*.

5.2.2 Properties Of Binary Trees

- Lemma 5.1 [*Maximum number of nodes*]:

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

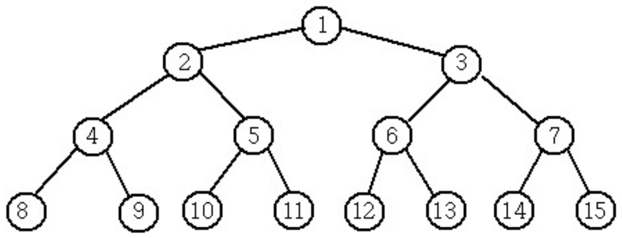
  (2) The maximum number of nodes in a binary tree of depth k is $2^{k}-1$, $k \geq 1$.

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

- We can number the nodes of a full binary tree, starting with the root on level 1, continuing with the nodes on level 2, and so on.
- Nodes on any level are numbered from left to right.

5.2.2 Properties Of Binary Trees

- [Figure 5.10] Full binary tree of depth 4 with sequential node numbers



- *Definition* : 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. □

5.2.3 Binary Tree Representations

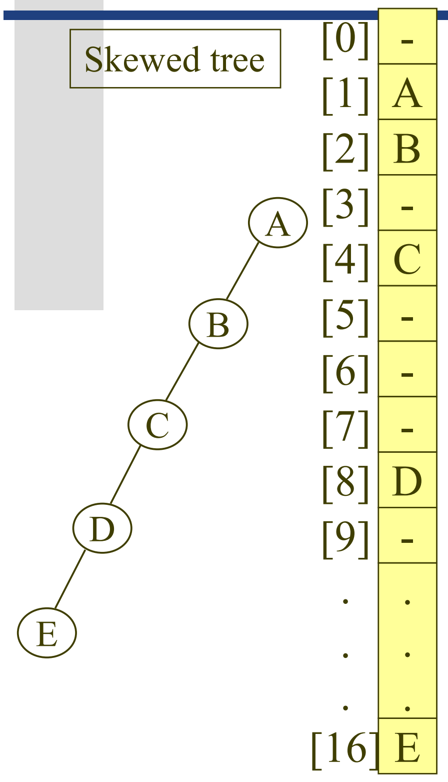
- **Array Representation**
- By using the numbering scheme shown in Figure 5.10, we can use a one-dimensional array to store the nodes in a binary tree. (We do not use the 0-th position of the array.)
- Lemma 5.3 :
  If a complete binary tree with n nodes (depth $= \lfloor \log_2 n + 1 \rfloor$) is represented sequentially, then for any node with index i, $1 \le i \le n$, we have :

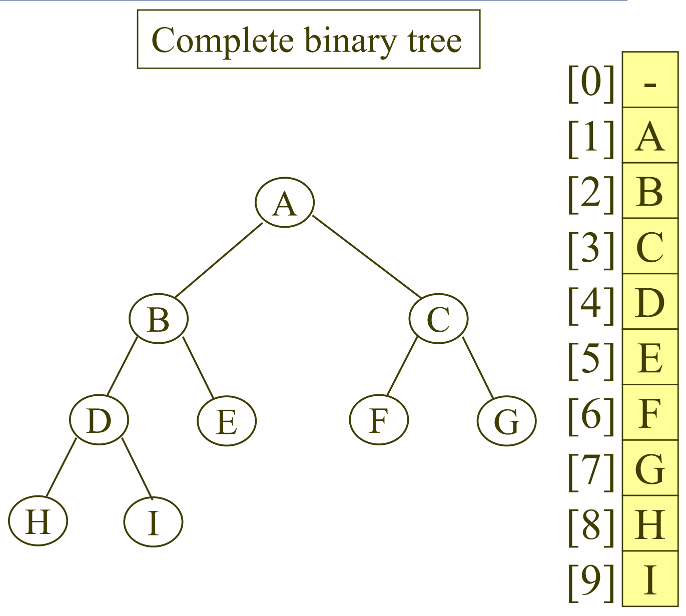
(1) *parent*(i) is at $\lfloor i/2 \rfloor$ if $i \ne 1$. If $i = 1$, i is at the root and has no parent.

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

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

5.2.3 Binary Tree Representations



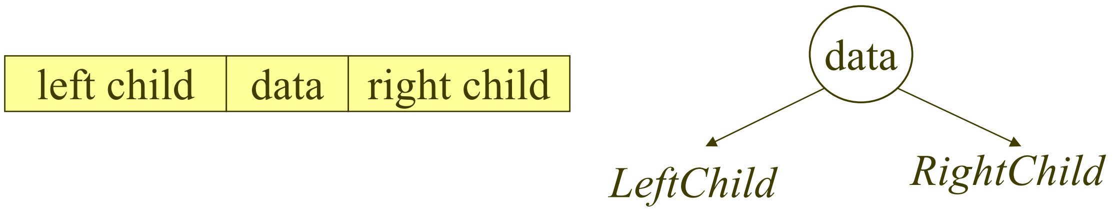


5.2.3 Binary Tree Representations

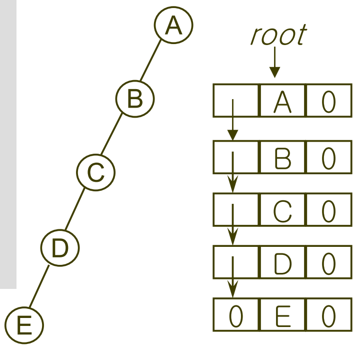
- **Linked Representation**

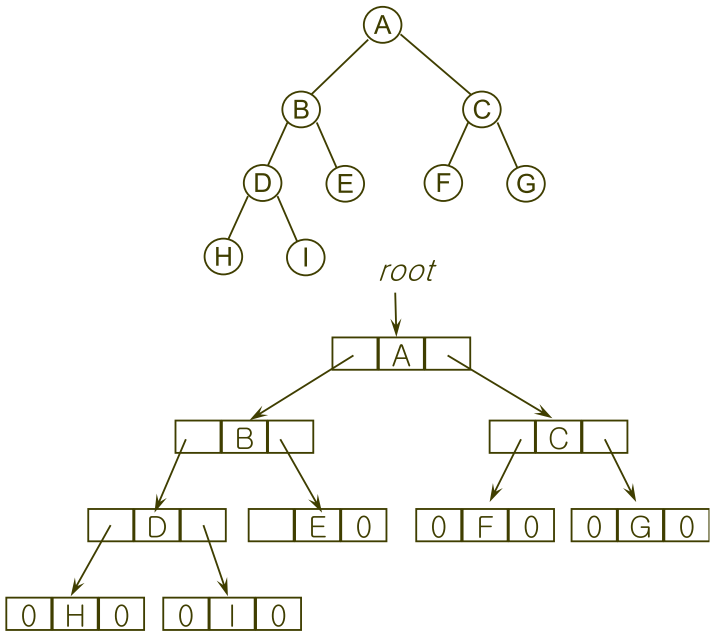
```
typedef struct node * treePointer;
typedef struct node{
          int data;
          treePointer leftChild, rightChild;
};
```

- [Figure 5.12] Node representation for binary trees.



5.2.3 Binary Tree Representations





[Figure 5.13] The linked representation for the trees in Figure 5.9.

5.3 BINARY TREE TRAVERSALS

- One of the operations that arises frequently is traversing a tree, that is, visiting each node in the tree exactly once.
- A full traversal produces a linear order for the information in a tree.
- When traversing a tree we want to treat each node and its subtrees in the same way.

- Let, for each node in a tree, L stands for moving left, V stands for visiting the node (e.g., printing out the data field), R stands for moving right.

5.3 BINARY TREE TRAVERSALS

- Six possible combinations of traversal :

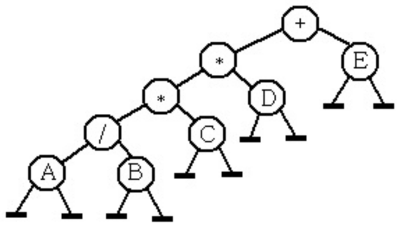
  LVR : *inorder traversal*

  LRV : *postorder traversal*

  VLR : *preorder traversal*

  VRL, RVL, RLV.

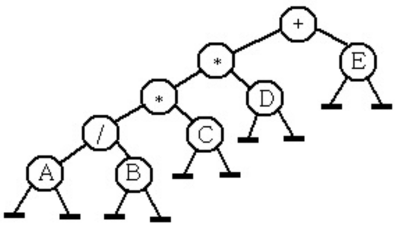
- There is a natural correspondence between these traversals and producing the infix, postfix, and prefix forms of an expression.
- [Figure 5.15] Binary tree with arithmetic expression



Inorder Traversal

- **LVR :** A/B*C*D+E

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



- The data fields of Figure 5.15 are output in the order : A / B * C * D + E

Inorder Traversal

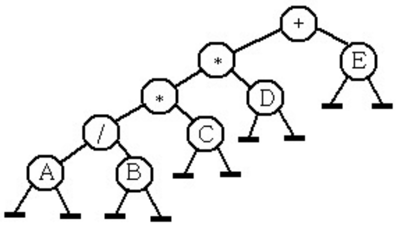
- [Figure 5.16]

Call of inorder	Value in root	Action	inorder	in root	Value Action
1	+		11	C	
2	*		12	NULL	
3	*		11	C	printf
4	/		13	NULL	
5	A		2	*	printf
6	NULL		14	D	
5	A	printf	15	NULL	
7	NULL		14	D	printf
4	/	printf	16	NULL	
8	B		1	+	printf
9	NULL		17	E	
8	B	printf	18	NULL	
10	NULL		17	E	printf
3	*	printf	19	NULL	

Preorder Traversal

- **VLR :** + * * / A B C D E

```
void preorder(treePointer ptr)
/* Preorder Traversal */
{
        if (ptr) {
                printf("%d", ptr->data);
                preorder(ptr->leftChild);
                preorder(ptr->rightChild);
        }
}
```

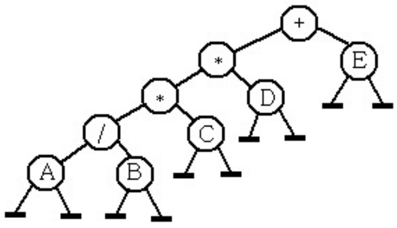


- The data fields of Figure 5.15 are output in the order : + * * / A B C D E

Postorder Traversal

- **LRV :** A B / C * D * E +

```
void postorder(treePointer ptr)
/* Postorder Traversal */
{
        if (ptr) {
                postorder(ptr->leftChild);
                postorder(ptr->rightChild);
                printf("%d", ptr->data);
        }
}
```



- The data fields of Figure 5.15 are output in the order : A B / C * D * E +

Iterative Inorder Traversal

- Figure 5.16 implicitly shows the stacking and unstacking of Program 5.1.

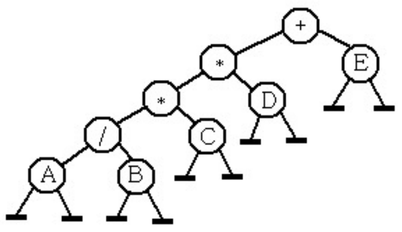
  - a node that has no action indicates that the node is added to the stack,

  - while a node that has a *printf* action indicates that the node is removed from the stack.

- Notice that :
  - the left nodes are stacked until a null node is reached,
  - the node is then removed from the stack, and
  - the node's right child is stacked.

Iterative Inorder Traversal

```
void iterInorder(treePointer node)
{
    int top = -1; /* init stack*/
    treePointer stack[MAX_STACK_SIZE];
    for (;;) {
            for (; node; node = node->leftChild)
                        add(&top, node);
            node = delete(&top);
            if (!node) break;
            printf("%d", node->data);
            node = node->rightChild;
    } }
```

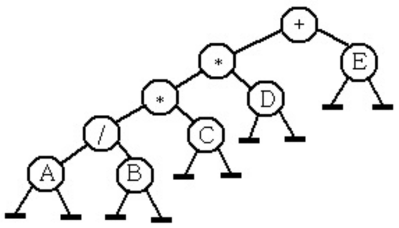


- **Analysis of *iter_inorder*** : Let n be the number of nodes in the tree. Note that every node of the tree is placed on and removed from the stack exactly once.
- The time complexity is $\Theta(n)$.
- The space complexity is equal to the depth of the tree which is $O(n)$.

Level order Traversal

- A traversal that requires a queue.
- *Level order traversal* visits the nodes using the ordering scheme suggested in Figure 5.11.

```
void levelOrder(treePointer ptr) {
        int front = rear = 0;
        treePointer queue[MAX_QUEUE_SIZE];
        if (!ptr) return;
        addq(front, &rear, ptr);
        for ( ; ; ) {
                ptr = deleteq(&front, rear);
                if (ptr) {
                        printf("%d", ptr->data);
                        if (ptr->leftChild)
                                addq(front, &rear, ptr->leftChild);
                        if (ptr->rightChild)
                                addq(front, &rear, ptr->rightChild);
                }
                else break;
        } }
```

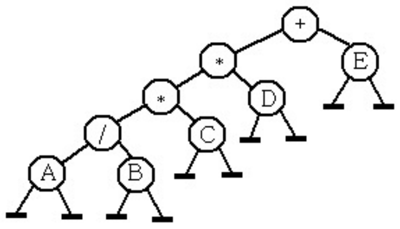


- The data fields of Figure 5.15 are output in the order : + * E * D / C A B

5.4 ADDITIONAL BINARY TREE OPERATIONS

- By using the definition of a binary tree and the recursive versions of inorder, preorder, and post order traversals, we can easily create C functions for other binary tree operations.
- **Copying Binary Trees**
- One practical operation is copying a binary tree. (Program 5.6)
- Note that this function is only a slightly modified version of *postorder* (Program 5.3)

Copying Binary Trees



- [Program 5.6] copying binary tree

```
treePointer copy(treePointer original) {
        treePointer temp;
        if (orginal) {
                temp = (treePointer) malloc(sizeof(node));
                if ( IS_FULL(temp)) {
                        fprintf(stderr, “The memory is full\n”);
                        exit(1);
                }
                temp->leftChild = copy(original->leftChild);
                temp->rightChild = copy(original->rightChild);
                temp->data = original->data;
                return temp;
        }
        return NULL;
}
```

Testing For Equality Of Binary Trees

- Equivalent trees have the same structure and the same information in the corresponding nodes.

```
int equal(treePointer first, treePointer second)
{
        return ((!first && !second) || (first && second &&
                (first->data == second->data) &&
                equal(first->leftChild, second->leftChild) &&
                equal(first->rightChild, second->rightChild));
}
```

The Satisfiability Problem

- Consider the formulas constructed by taking variables $x_1, x_2, \ldots, x_n$ and operators $\wedge$ (and), $\vee$ (or), and $\neg$ (not). The variables can hold only one of two possible values, *true* or *false*.

- The expressions are 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 $\neg$, $\wedge$, and $\vee$.

The Satisfiability Problem

- Consider an expression :

$$\mathbf{x_1 \vee (x_2 \wedge \neg x_3)}$$

- If x_1 and x_3 are *false* and x_2 is *true*, the value of this expression is *true*.

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

- The *satisfiability problem* for formulas of the propositional calculus asks if there is an assignment of values to the variables that cause the value of the expression to be true.

A first version of satisfiability algorithm

- [Program 5.8]

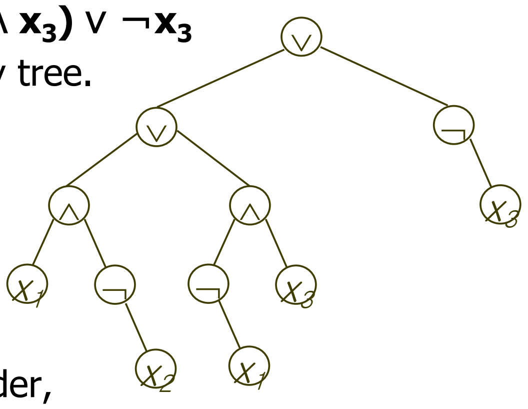
```
for (all $2^n$ possible combinations) {
    generate the next combination;
    replace the variables by their values;
    evaluate the expression;
    if (its value is true) {
        printf(<combination>);
        return;
    }
}
printf("No satisfiable combination₩n");
```

Evaluating an propositional formula

- Assume that our formula is already in a binary tree.
  For a formula :

$$(\mathbf{x_1} \wedge \neg \mathbf{x_2}) \vee (\neg \mathbf{x_1} \wedge \mathbf{x_3}) \vee \neg \mathbf{x_3}$$

- [Figure 5.18] Corresponding binary tree.



- To evaluate an expression
  we can traverse the tree in postorder,
  evaluating the subtrees until entire expression is reduced to a single value.
- This corresponds to the postfix evaluation of an arithmetic expression.

Evaluating an propositional formula

- [Figure 5.19] structure of node

leftChild	data	value	rightChild

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

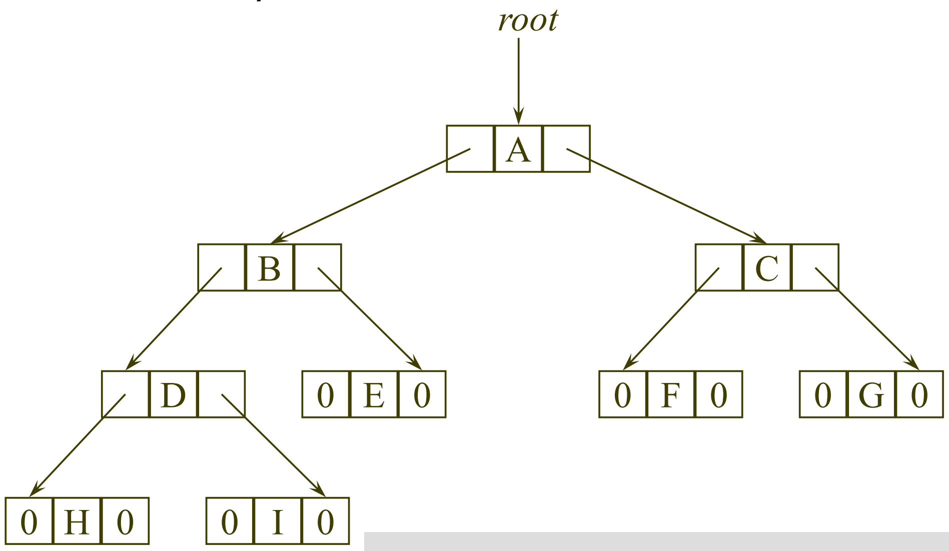
Evaluating an propositional formula

- [Program 5.9]

```
void postOrderEval(treePointer node){
            if (node) {
                        postOrderEval (node->leftChild);
                        postOrderEval (node->rightChild);
                        switch(node->data) {
                                    case not:
                                                node->value = !node->rightChild->value; break;
                                    case and:
                                                node->value = node->rightChild->value &&
                                                                        node->leftChild->value; break;
                                    case or:
                                                node->value = node->rightChild->value ||
                                                                        node->leftChild->value; break;
                                    case true:
                                                node->value = TRUE; break;
                                    case false:
                                                node->value = FALSE;
                        }
            }
}
```

5.5 THREADED BINARY TREES

- A binary tree T with *n* nodes has 2*n* links and among them, (*n*+1) are NULL links. A.J. Perlis and C. Thornton have devised a clever way to make use of these null links.



5.5 THREADED BINARY TREES

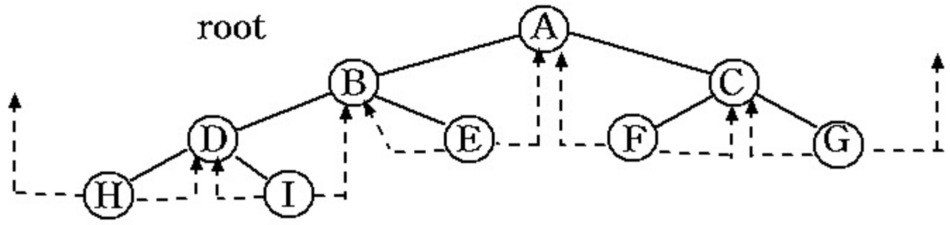
- They replace the null links by pointers, called *threads*, to other nodes in the tree by using the following rules (assume that *ptr* represents a node) :

(1) If *ptr->left_child* is null, replace *ptr->left_child* 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->right_child* is null, replace *ptr->right_child* 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*.

5.5 THREADED BINARY TREES

- [Figure 5.21] Threaded binary tree

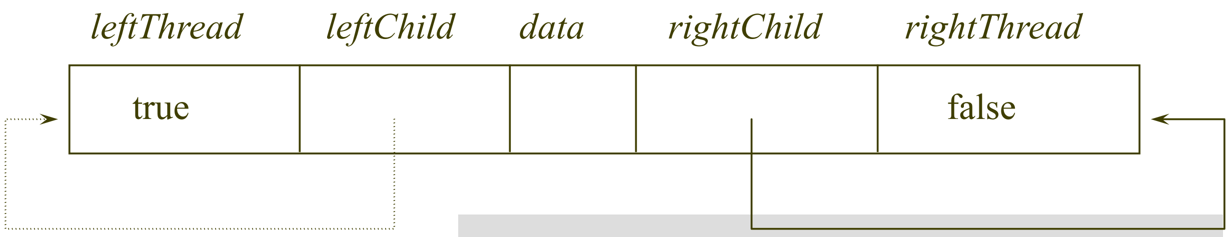


- When we represent the tree in memory, we must be able to distinguish between threads and normal pointers.
- This is done by adding two additional fields to the node structure, *left_thread* and *right_thread*.

5.5 THREADED BINARY TREES

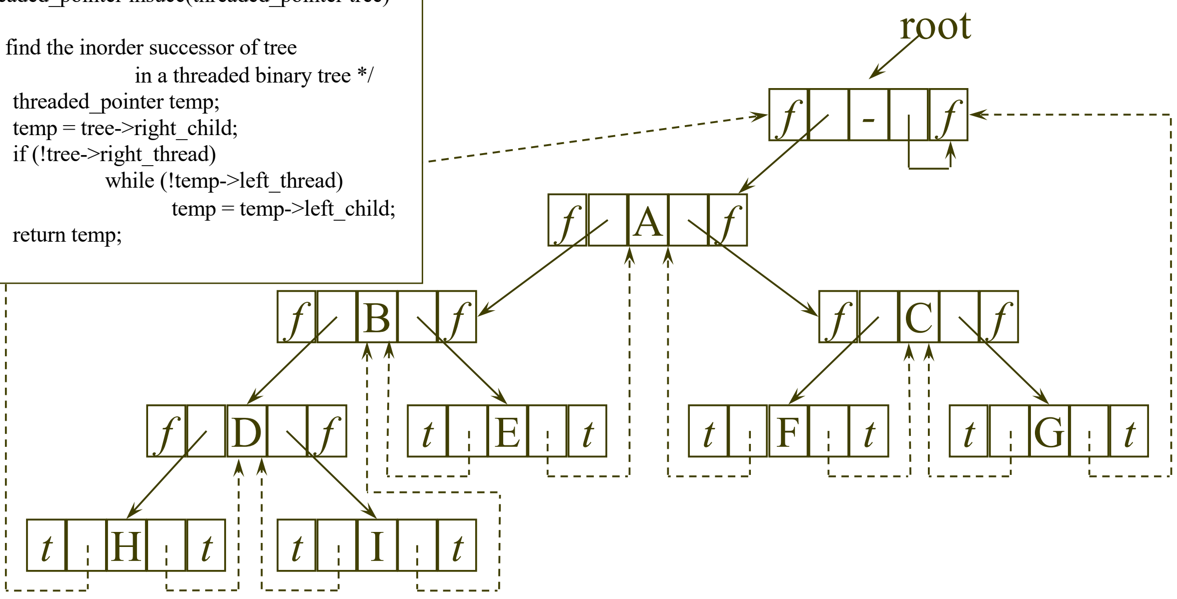
```
typedef struct threaded_tree *threaded_pointer;
typedef struct threaded_tree {
        short int left_thread;
        threaded_pointer left_child;
        char  data;
        threaded_pointer right_child;
         short int right_thread;
};
```

- We assume that all threaded binary trees have a head node.
- [Figure 5.21] An empty threaded tree



5.5 THREADED BINARY TREES

```
threaded_pointer insucc(threaded_pointer tree)
 {
 /* find the inorder successor of tree
                    in a threaded binary tree */
     threaded_pointer temp;
     temp = tree->right_child;
     if (!tree->right_thread)
                 while (!temp->left_thread)
                         temp = temp->left_child;
     return temp;
  }
```



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

Inorder Traversal of a Threaded Binary Tree

- Determining the inorder successor of a node.
- [Program 5.10] Finding the inorder successor of a node.

```
threaded_pointer insucc(threaded_pointer tree)
 {
 /* find the inorder successor of tree
                    in a threaded binary tree */
     threaded_pointer temp;
     temp = tree->right_child;
     if (!tree->right_thread)
           while (!temp->left_thread)
           temp = temp->left_child;
     return temp;
  }
```

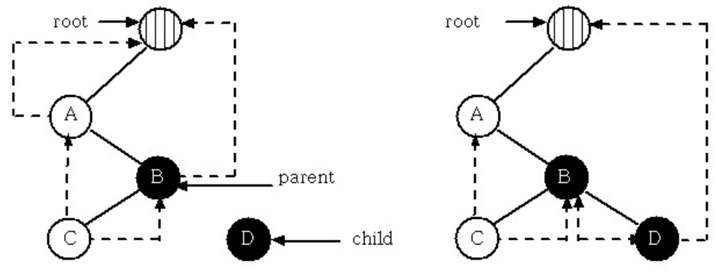
Inorder Traversal of a Threaded Binary Tree

- To perform an inorder traversal we make repeated calls to *insucc*.
- [Program 5.11] Inorder traversal of a threaded binary tree.

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

Inserting A Node Into A Threaded Binary Tree

- Assume that we have a node, *parent*, that has an empty right subtree. We wish to insert *child* as the right child of parent.
- [Figure 5.24] (a)



(a)

- To do this we must

(1) change *parent-> right_thread* to FALSE

(2) set *child->left_thread* and *child->right_thread* to TRUE

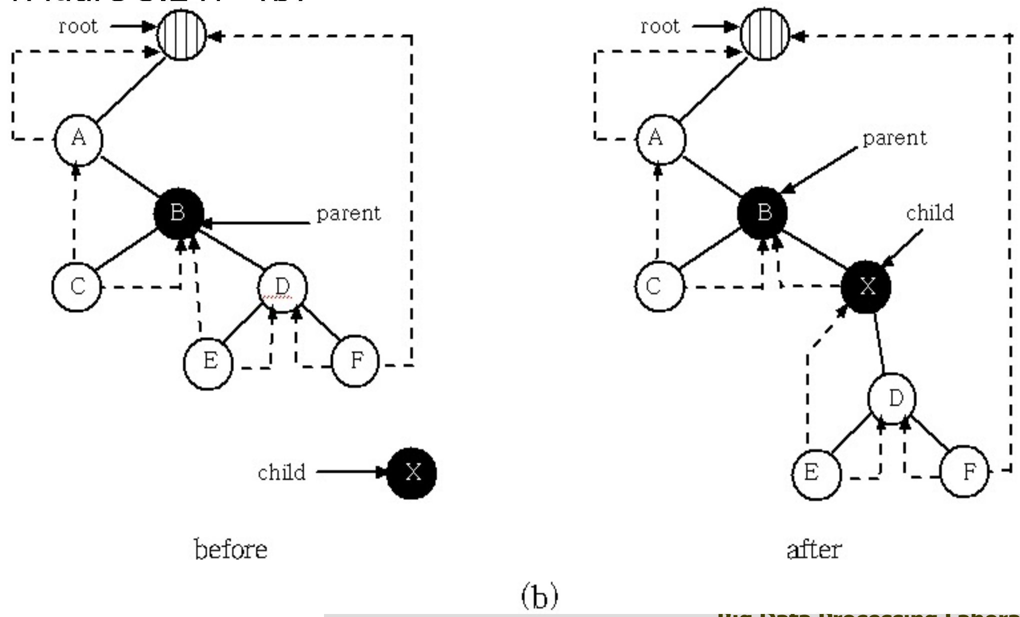
(3) set *child->left_child* to point to *parent*

(4) set *child->right_child* to *parent->right_child*

(5) change *parent->right_child* to point to *child*

Inserting A Node Into A Threaded Binary Tree

- For the case that *parent* has a nonempty right subtree,
- [Figure 5.24] (b)



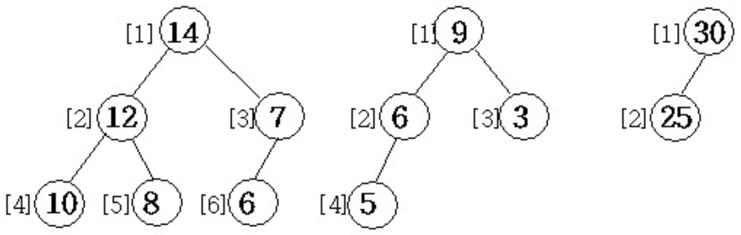
Inserting A Node Into A Threaded Binary Tree

- [Program 5.12] : Right insertion in a threaded binary tree

```
void insert_right(threaded_pointer parent,
                              threaded_pointer child)   {
 /* insert child as the right child of parent
                    in a threaded binary tree */
      threaded_pointer temp;
      child->right_child = parent->right_child;
      child->right_thread = parent->right_thread;
      child->left_child = parent;
      child->left_thread = TRUE;
      parent->right_child = child;
      parent->right_thread = FALSE;
      if (!child->right_thread) {
             temp = insucc(child);
             temp->left_child = child;
      }
 }
```

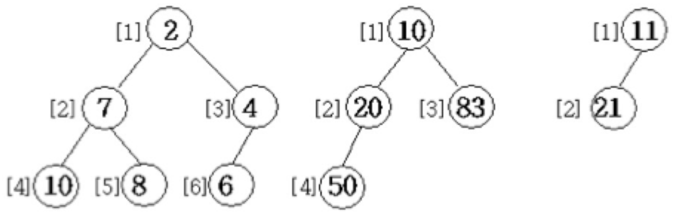
5.6 HEAPS

- **5.6.1 The Heap Abstract Data Type**
- Definition : A *max tree* is a tree in which the key value in each node is no smaller than the key values in its children (if any). A *max heap* is a complete binary tree that is also a max tree.
- Definition : A *min tree* is a tree in which the key value in each node is no larger than the key values in its children (if any). A *min heap* is a complete binary tree that is also a min tree.
- [Figure 5.25] sample max heaps



5.6 HEAPS

- [Figure 5.26] sample min heaps



- Notice that we represent a heap as an array, although we do not use position 0.
- From the heap definitions it follows that
  - the root of a min tree contains the smallest key in the tree.
  - the root of a max tree contains the largest key in the tree.
- Basic operations on a max heap :
  (1) Creation of an empty heap
  (2) Insertion of a new element into the heap
  (3) Deletion of the largest element from the heap

5.6 HEAPS

- [Structure 5.2] : Abstract data type *MaxHeap*.

ADT *MaxHeap* is

object: a complete binary tree of n>=0 elements organized so that the value in each node is at least as large as those in its children

functions: for all *heap* *MaxHeap*, *item* *Element, n, max_size* integer

MaxHeap **Create**(*max_size*) ::= create an empty heap that can hold a maximum of *max_size* elements.

Boolean **HeapFull**(*heap, n*) ::= if (*n == max_size*) return *TRUE* else return *FALSE*

MaxHeap **Insert**(*heap, item, n*) ::= if (!HeapFull(*heap, n*)) insert an item into heap and return the resulting heap else return error.

Boolean **HeapEmpty**(*heap, n*) ::= if (*n*<=0) return *TRUE* else return *FALSE*

MaxHeap **Delete**(*heap, n*) ::= if (!HeapEmpty(*heap, n*)) return one of the largest element in the heap and remove it from the heap else return error.

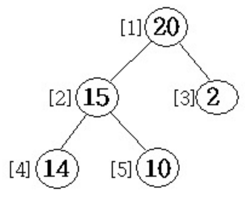
5.6.2 Priority Queues

- Heaps are frequently used to implement *priority queues*.
- Unlike the queues, FIFO lists, a priority queue deletes the element with the highest (or the lowest) priority.
- At any time an element with arbitrary priority can be inserted into a priority queue.
- Heaps are used as an efficient implementation of the priority queues. To examine some of the other representations see Figure 5.27.
- [Figure 5.27]

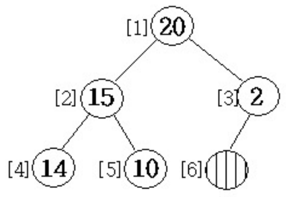
Representation	Insertion	Deletion
Unordered array	Θ(1)	Θ(*n*)
Unordered linked list	Θ(1)	Θ(*n*)
Sorted array	O(*n*)	Θ(1)
Sorted linked list	O(*n*)	Θ(1)
Max heap	O($\log_2 n$)	O($\log_2 n$)

5.6.3 Insertion Into A Max Heap

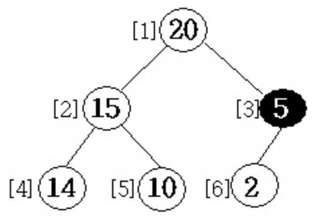
- To illustrate the insertion operation, See Figure 5.28
- [Figure 5.28]



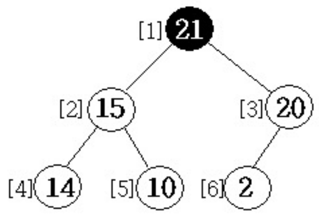
(a) before heap insertion



(b) initial location of new node



(c) insertion 5 into heap(a)



(d) insertion 21 into heap(a)

5.6.3 Insertion Into A Max Heap

- We use the array representation discussed in Section 5.2.3.
- C declaration:

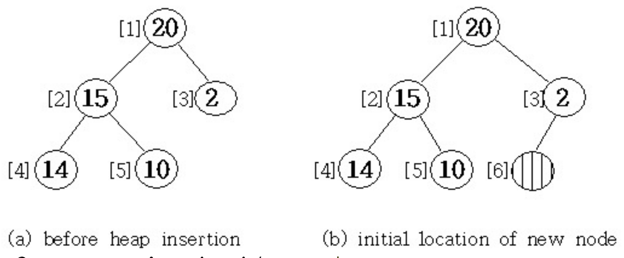
```
#define MAX_ELEMENTS 200   /*maximum heap size+1 */
#define HEAP_FULL(n) (n == MAX_ELEMENTS-1)
#define HEAP_EMPTY(n) (!n)
typedef struct {
            int key;
            /* other fields */
} element;
element heap[MAX_ELEMENTS];
int n = 0;
```

- We can insert a new element in a heap with *n* elements by following the steps below :
  (1) place the element in the new node (i.e., *n*+1 th position)
  (2) move along the path from the new node to the root,
  if the element of the current node is larger than the one of its parent then interchange them and repeat.

5.6.3 Insertion Into A

- [Program 5.13] Insertion into

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



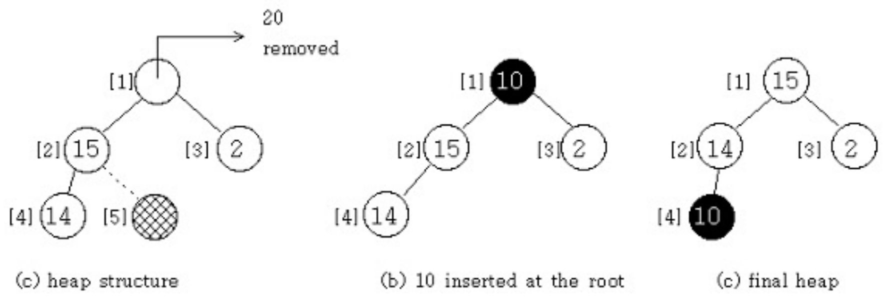
5.6.3 Insertion Into A Max Heap

- Analysis of *insert_max_heap* :
- The function first checks for a full heap.
- If not, set *i* to the size of the new heap (n+1).
- Then determines the correct position of *item* in the heap by using the *while* loop.
- This while loop is iterated O($\log_2$ n) times.

- Hence the time complexity is O($\log_2$ n).

5.6.4 Delete From A Max Heap

- When we delete an element from a max heap, we always take it from the root of the heap.
- If the heap had *n* elements, after deleting the element in the root, the heap must become a complete binary tree with one less nodes, i.e., (*n*-1) elements.
- We place the element in the node at position *n* in the root node and to establish the heap we move down the heap, comparing the parent node with its children and exchanging out-of-order elements until the heap is reestablished.
- [Figure 5.29] Deletion from a max heap



5.6.4 Delete From A Max Heap

- [Program 5.14] : Deletion from a max_heap

```
element delete_max_heap(int *n) {
/* delete element with the
highest key from the heap */
   int  parent, child;
   element  item, temp;
   if (HEAP_EMPTY(*n))  {
      fprintf(stderr,"The heap is empty");
      exit(1);
   }
/* save value of the element
             with the largest key */
   item = heap[1];
     /* use last element in
             heap to adjust heap  */
   temp = heap[(*n)--];
   parent = 1;         child = 2;
    while (child <= *n)  {
    /* find the larger
             child of the current parent  */
    if ((child < *n) &&
(heap[child].key< heap[child+1].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;
}
```

5.6.4 Delete From A Max Heap

- Analysis of *delete_max_heap* [Program 5.14]:
  - The function *delete_max_heap* operates by moving down the heap,
  - comparing and exchanging parent and child nodes until the heap definition is re-established.
  - Since the height of a heap with n elements is the while loop is iterated O($\log_2$ n) times.
- Hence the time complexity is O($\log_2$ n).

5.7 Binary search trees

- **5.7.1 Introduction**
- While a heap is well suited for applications that require priority queues, it is not well suited for applications in which we delete and search arbitrary elements.
- A *binary search tree* has a better performance than any of the data structures studied so far for operations, insertion, deletion, and searching of arbitrary element.
- In fact, with a binary search tree we can perform these operations by both key value (e.g., delete the element with key x) and by rank (e.g., delete the fifth smallest element).

5.7 Binary search trees

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

(1) Every element has a key, and no two elements have the same key, that is, the keys are unique.

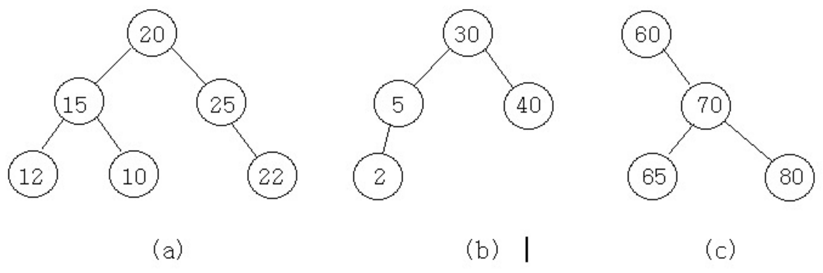
(2) The keys in a nonempty left subtree must be smaller than the keys in the root.

(3) The keys in a nonempty right subtree must be larger than the keys in the root.

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

5.7 Binary search trees

- [Figure 5.30] some sample binary trees



- If we traverse a binary search tree in inorder and print the data of the nodes in the order visited, what would be the order of data printed?

5.7.2 Searching A Binary Search Tree

- [Program 5.15] : Recursive search for a binary search tree

```
tree_pointer search(tree_pointer root, int key)
  {
  /* return a pointer to the node that contains key.
    If there is no such node, return NULL.   */
          if (!root)  return NULL;
          if (key == root->data)  return root;
          if (key < root->data)
                    return  search(root->left_child, key);
          return  search(root->right_child, key);
  }
```

5.7.2 Searching A Binary Search Tree

- [Program 5.16] Iterative search for a binary search tree

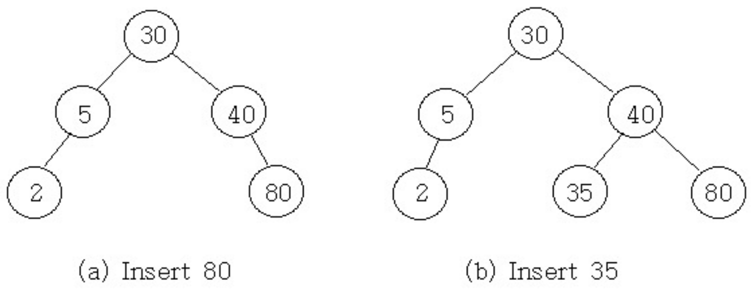
```
tree_pointer search2(tree_pointer tree, int key)
  {
  /* return a pointer to the node that contains key.
    If there is no such node, return NULL.   */
          while (tree)  {
            if (key == tree->data)  return tree;
            if (key < tree->data)
                    tree = tree->left_child;
            else
                    tree = tree->right_child;
         }
    return NULL;
  }
```

5.7.2 Searching A Binary Search Tree

- **Analysis of *search* and *search2*** :
  - If *h* is the height of the binary search tree, then the time complexity of both *search* and *search2* is O(*h*).
  - However, *search* has an additional stack space requirement which is O(*h*).
- Searching a binary tree is similar to the binary search of a sorted list.

5.7.3 Inserting Into A Binary Search Tree

- To insert a new element, *key* :
  - First, we verify that the key is different from those of existing elements by searching the tree.
  - If the search is unsuccessful, then we insert the element at the point the search terminated.
- [Figure 5.31] inserting into a binary search tree



5.7.3 Inserting Into A Binary Search Tree

```
void insert_node(tree_pointer *node, int num) {
   /* If num is in the tree pointed at by node do nothing;
      otherwise add a new node with data = num  */
           tree_pointer  ptr, temp = modified_search(*node, num);
           if (temp || !(*node))  {
              /* num is not in the tree  */
                ptr = (tree_pointer) malloc(sizeof(node));
                if (IS_FULL(ptr)) {
                     fprintf(stderr, "The memory is full");
                     exit(1);
                }
                ptr->data = num;
                ptr->left_child = ptr->right_child = NULL;
                if (*node)      /* insert as child of temp  */
                    if (num < temp->data)
                       temp->left_child = ptr;
                    else  temp->right_child = ptr;
                else  *node = ptr;
} }
```

[Program 5.17] :
Inserting an element
into a binary search tree

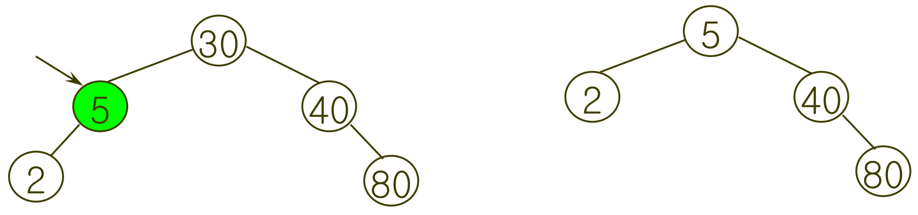
5.7.3 Inserting Into A Binary Search Tree

- function ***modified_search*** searches the binary search tree **node* for the key *num*.
- If the tree is empty or if num is presented, it returns NULL.
- Otherwise, it returns a pointer to the last node of the tree that was encountered during the search.

- **Analysis of *insert_node* :**
- ❑ Let h be the height of the binary search tree.
- ❑ Since the search requires $O(h)$ time and the remainder of the algorithm takes $\Theta(1)$ time.
- So overall time needed by *insert_node* is $O(h)$.

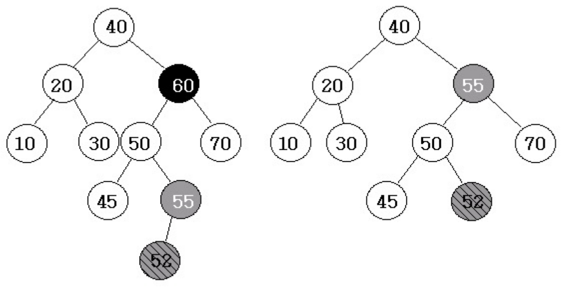
5.7.4 Deletion From A Binary Search Tree

- <u>Deletion of a leaf node</u> :
  - Set the corresponding child field of its parent to NULL and free the node.
- <u>Deletion of a nonleaf node with single child</u> :
  - Erase the node and then place the single child in the place of the erased node.
- [Figure 5.32] Deletion from a binary tree



5.7.4 Deletion From A Binary Search Tree

- **<u>Deletion of a nonleaf node with two children</u> :**
  - Replace the node with either the largest element in its left subtree or the smallest element in its right subtree. Then delete this replacing element from the subtree from which it was taken.
  - Note that the largest and smallest elements in a subtree are always in a node of degree zero or one.
- [Figure 5.33]



(a) tree before deletion of 60 (b) tree after deletion of 60

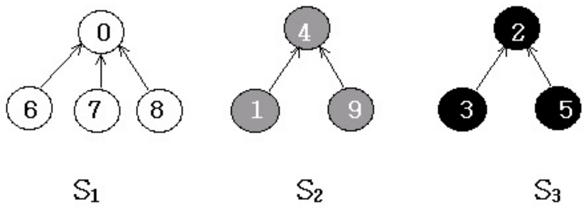
- It is easy to see that a deletion can be performed in O(h) time, where h is the height of the binary search tree.

5.7.5 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*.
- However, when insertion and deletions are made at random, the height of the binary search tree is $O(\log_2 n)$, on the average.
- Search trees with a worst case height of $O(\log_2 n)$ are called *balanced search trees*.
- Examples : AVL tree, 2-3 tree, and red-black tree, etc.

5.10 SET REPRESENTATION

- We study the use of trees in the representation of sets.
- For simplicity, we assume that the elements of the sets are the numbers 0, 1, 2, . . ., *n*-1.
- We also assume that the sets being represented are pairwise disjoint.
- [Figure 5.39] for a possible representation.



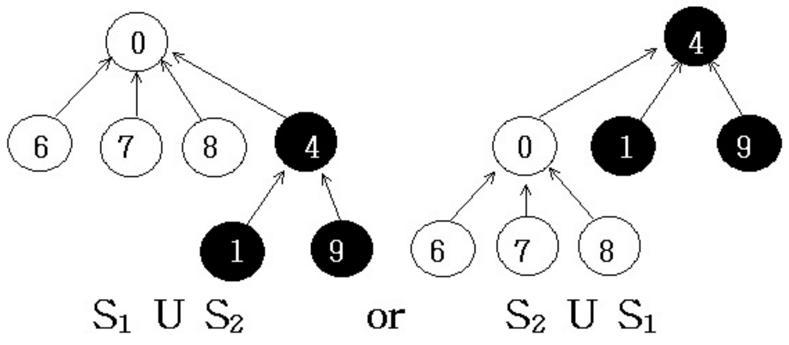
- Notice that for each set the nodes are linked from the children to the parent. The operations to perform on these sets are:

(1) *Disjoint set union*. If we wish to get the union of two disjoint sets Si and Sj, replace Si and Sj by SiUSj.

(2) *Find(i)*. Find the set containing the element, *i*.

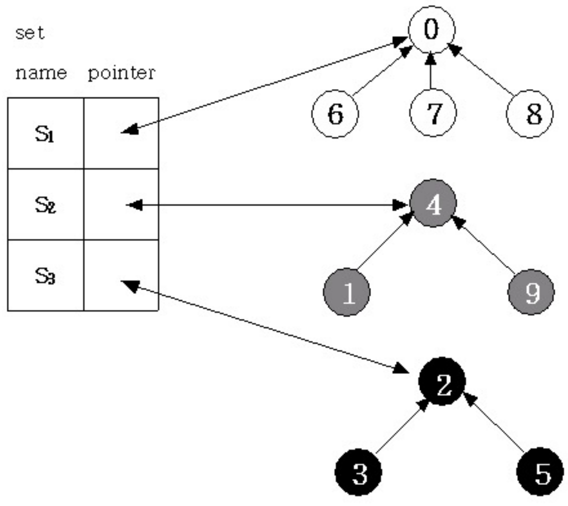
5.10.1 Union and Find Operations

- Suppose that we wish to obtain the union of S1 and S2.
- We simply make one of the trees a subtree of the other.
- S1∪S2 could have either of the representations of Figure 5.40
- [Figure 5.40] Possible representation of $S_1 \cup S_2$



5.10.1 Union and Find Operations

- To implement the set union operation, we simply set the parent field of one of the roots to the other root. Figure 5.41 shows how to name the sets.
- [Figure 5.41] Data representation of S_1,S_2 and S_3



5.10.1 Union and Find Operations

- To simplify the discussion of the union and find algorithms, we will ignore the set names and identify the sets by the roots of the trees representing them.
- Since the nodes are in the trees are numbered 0 through *n*-1, we can use the node's number as an index.

- [Figure 5.42] : Array representation of the trees in Figure 5.39.

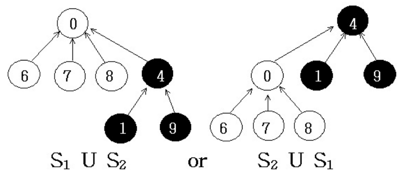
i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	-1	2	0	0	0	4

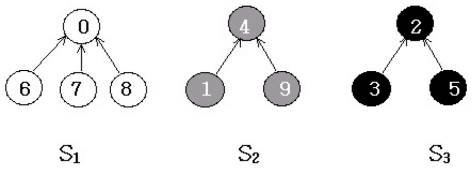
- Notice that root nodes have a parent of –1.

5.10.1 Union and Find Operations

- We can implement *find*(*i*) by simply following the indices starting at *i* and continuing until we reach a negative parent index.
- [Program 5.18] : Initial attempt at union-find functions.

```
int find(int i)
{
     for ( ; parent[i] >= 0 ; i = parent[i]) ;
     return  i;
}
void union1(int i, int j)
{
     parent[i] = j;
}
```





i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	-1	2	0	0	0	4

5.10.1 Union and Find Operations

- Since the time taken for a union is constant, all the *n*-1 unions can be processed in time O(*n*).
- For each *find*, if the element is at level *i*, then the time required to find its root is O(*i*).
- Hence the total time needed to process the *n*-1 finds is :

$$\sum_{i=2}^{n} i = O(n^2)$$

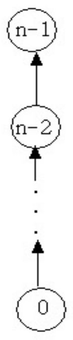
- By avoiding the creation of degenerate trees, we can attain far more efficient implementations of the union and find operations.

5.10.1 Union and Find Operations

- **Analysis of *union*1 and *find*1 :**
- Let us process the following sequence of union-find operations:

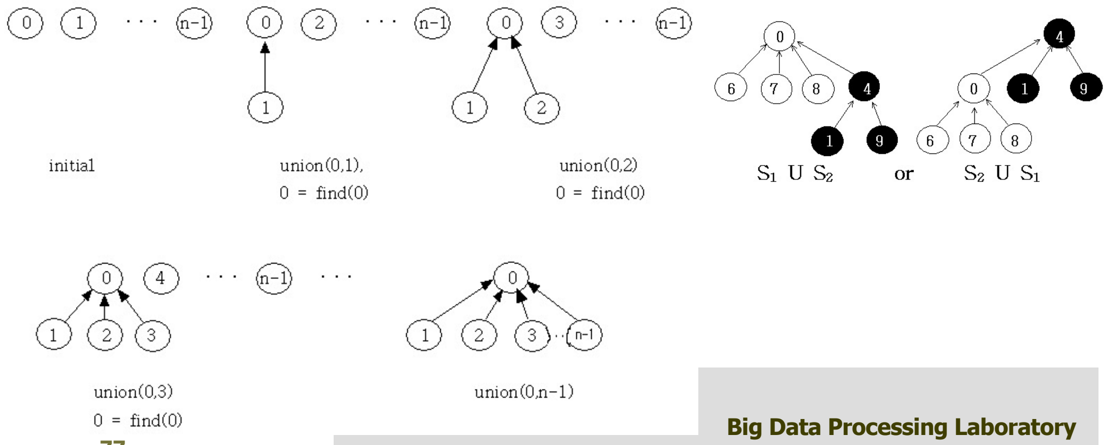
union(0, 1), *find*(0)
union(1, 2), *find*(0)
.
.
union(n-2, n-1), *find*(0)

- This sequence produces the degenerate tree of Figure 5.43.
- [Figure 5.43] Degenerate tree



5.10.1 Union and Find Operations

- Definition : *Weighting rule for union*(*i, j*). If the number of nodes in tree *i* is less than the number in tree *j* then make *j* the parent of *i*; otherwise make *i* the parent of *j*. □
- When we use this rule on the sequence of set unions described above, we obtain the trees of Figure 5.44.
- [Figure 5.44] Trees obtained using the weighting rule



5.10.1 Union and Find Operations

- To implement the weighting rule, we need to know how many nodes there are in every tree. That is, we need to maintain a count field in the root of every tree. We can maintain the count in the parent field of the roots as a negative number.
- [Program 5.19] : Union operation incorporating the weighting rule.

```
void union2(int i, int j) {
  /*  parent[i] = -count[i] and parent[j] = -count[j]  */
          int temp = parent[i] + parent[j];
          if (parent[i] > parent[j])   {
                   parent[i] = j;   /* make j the new root  */
                   parent[j] = temp;
          }
          else  {
                   parent[j] = i;   /* make i the new root  */
                   parent[i] = temp;
          }
 }
```

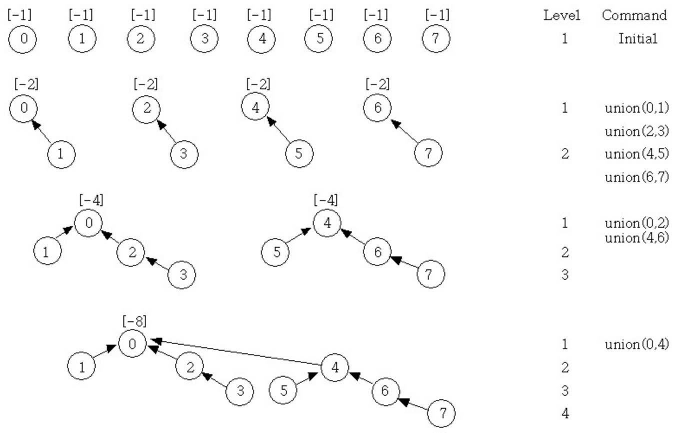
- Lemma 5.4 : Let T be a tree with *n* nodes created as a result of *union*2. Then the depth of T ≤ $\lfloor \log_2 n \rfloor + 1$

5.10.1 Union and Find Operations

- Example 5.1 : Consider the behavior of union2 on the following sequence of unions starting from the initial configuration :

  union(*0*, *1*) *union*(*2*, *3*) *union*(*4*, *5*) *union*(*6*, *7*)
  union(*0*, *2*) *union*(*4*, *6*) *union*(*0*, *4*)

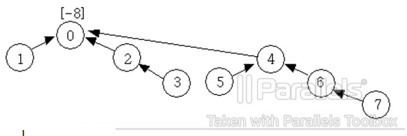
- Figure 5.45 shows the result.



Further Improvement

- Definition [Collapsing rule] : If *j* is a node on the path from *i* to its root then make *j* a child of the root. □
- [Program 5.20] : Find function incorporated with the collapsing rule.

```
int find2(int i)
  {
  /* find the root of the tree containing element i.  Use the
   collapsing rule to collapse all nodes from i to root  */
            int root, trail, lead;
            for (root = i; parent[root] >= 0; root = parent[root]) ;
            for (trail = i; trail != root; trail = lead)  {
                         lead = parent[trail];
                         parent[trail] = root;
            }
            return  root;
   }
```



Further Improvement

- Example 5.2 : Consider the tree created by *union*2 on the sequence of unions of Example 5.1. Now process the following 8 finds:

  find(7), *find*(7), . . ., *find*(7)

- The Worst Case Behavior of the union-find algorithms while processing a sequence of unions and finds.
- Let $\alpha(m,n)$ be a function defined as

  $$\alpha(m,n) = \min\{z \geq 1 \mid A(z, \lfloor m/n \rfloor) > \log_2 n\}, m \geq n \geq 1$$

  where A(p,q) is Ackermann's function which grows very rapidly.

  Thus, $\alpha(m,n)$ is a very slowly growing function.

- Definition of Ackermann's function :

  $A(1,j) = 2^j$ for $j \geq 1$
  $A(i,1) = A(i-1,2)$ for $i \geq 2$
  $A(i,j) = A(i-1, A(i,j-1))$ for $i, j \geq 2$

Further Improvement

- Ackermann's function is a very rapidly growing function. We may prove that:

  (1) $A(3,4) = 2^{2^{\cdot^{\cdot^{2}}}}$; 65,536 twos

  (2) A(p,q+1) > A(p,q)

  (3) A(p+1,q) ≥A(p,q)

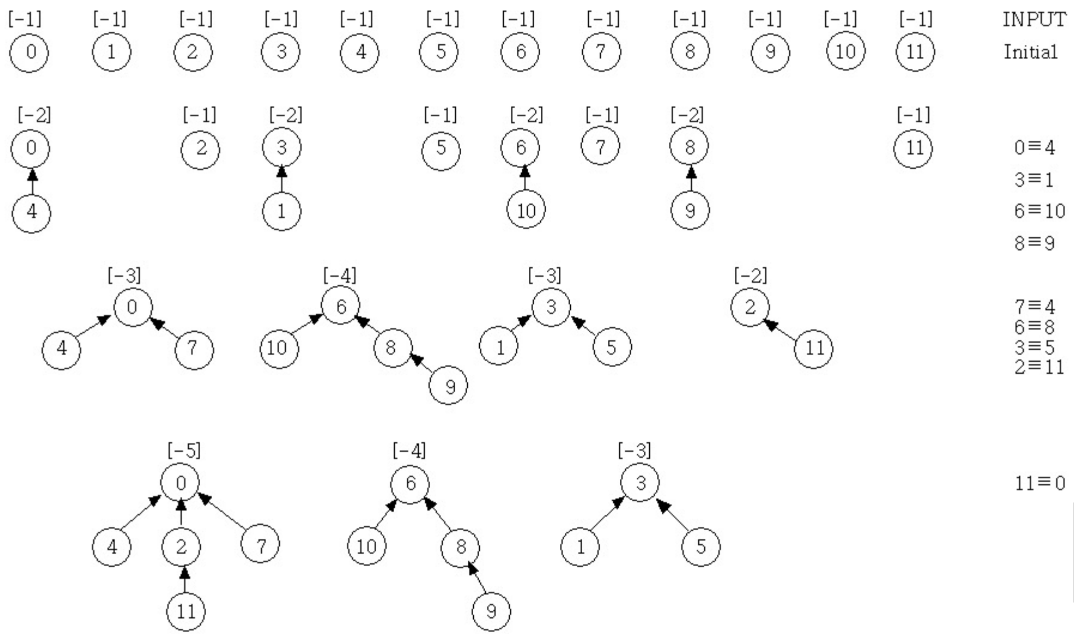
- Lemma 5.5 [Tarjan] : Let T(m,n) be the maximum time required to process an intermixed sequence of m≥n finds and n-1 unions. Then:

  $k_1 m\alpha(m,n) \leq T(m,n) \leq k_2 m\alpha(m,n)$

  for some positive constants k_1 and k_2. □

5.10.2 Equivalence Classes

- If we have n polygons and m≥n equivalence pairs, the total processing time is at most $O(n + m\alpha(2m, \min\{n-1, m\}))$.
- [Figure 5.46] Trees for equivalence example



INPUT
Initial
0≡4
3≡1
6≡10
8≡9
7≡4
6≡8
3≡5
2≡11
11≡0
[-1]
[-2]
[-3]
[-4]
[-5]