

4.7 SPARSE MATRIX

- In Chapter 2, we considered a sequential representation of sparse matrices and implemented matrix operations.
- However we found that the sequential representation of sparse matrices suffered from the same inadequacies as the similar representation of polynomials.
- As we have seen previously, linked lists allow us to efficiently represent structures that vary in size, a benefit that also applies to sparse matrices.
- In our data representation, we represent each column of a sparse matrix as a circularly linked list with a head node. We use a similar representation for each row of a sparse matrix.

[Figure 4.19]

down	head	right
next		

(a) head node

down	entry	row	col	right
value				

(b) entry node

	entry	i	j	
a_{ij}				

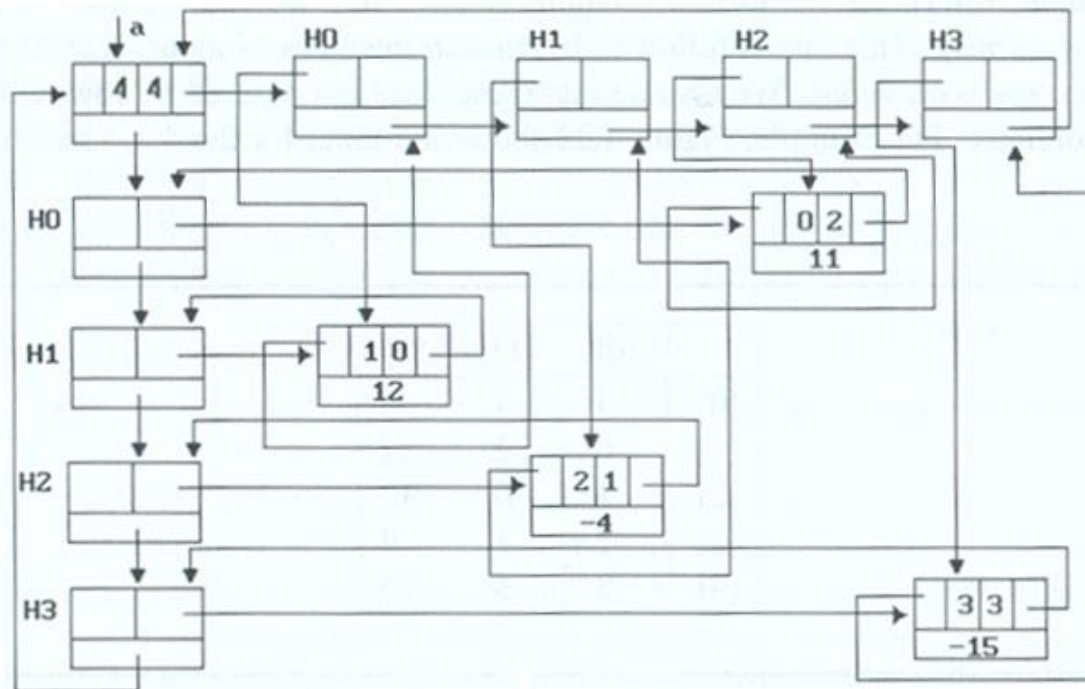
(c) set up for a_{ij}

[Figure 4.20]

$$\begin{bmatrix} 0 & 0 & 11 & 0 \\ 12 & 0 & 0 & 0 \\ 0 & -4 & 0 & 0 \\ 0 & 0 & 0 & -15 \end{bmatrix}$$

■ [Figure 4.21]

- Each head node is in three lists:
a list of rows, a list of columns, and a list of head nodes.
The list of head nodes also has a head node
that has the same structure as an entry node.



NOTE: The tag field of a node is not shown; its value for each node should be clear from the node structure.

Figure 4.21: Linked representation of the sparse matrix a

```
#define MAX_SIZE 50

typedef enum {head, entry} tagfield;
typedef struct matrix_node *matrix_pointer;
typedef struct entry_node {
    int row;
    int col;
    int value;
};

typedef struct matrix_node {
    matrix_pointer down;
    matrix_pointer right;
    tagfield tag;
    union {
        matrix_pointer next;
        entry_node entry;
    } u;
    matrix_pointer hdnnode[MAX_SIZE];
};
```

```
matrix_pointer mread()
{
    int num_rows, num_cols, num_terms, num_heads, i;
    int row, col, value, current_row;
    matrix_pointer temp, last, node;

    scanf(&num_rows, &num_cols, &num_terms);
    num_heads = (num_cols > num_rows) ? num_cols : num_rows;
    node = new_node(); node_tag = entry;
    node->u.entry.row = num_rows;
    node->u.entry.col = num_cols;
```

```

if (!num_heads) node->right = node;
else {
    for (i=0; i<num_heads; i++) {
        temp = new_node();
        hdnode[i] = temp; hdnode[i]->tag = head;
        hdnode[i]->right = temp; hdnode[i]->u.next=temp;
    }
    current_row = 0; last = hdnode[0];
    for (i=0; i<num_terms; i++) {
        scanf(&row, &col, &value);
        if (row > current_row) {
            last->right = hdnode[current_row];
            current_row = row; last = hdnode[row];
        }
        temp = new_node(); temp->tag = entry;
        temp->u.entry.row = row; temp->u.entry.col = col;
        temp->u.entry.value = value; last->right = temp; last = temp;
        hdnode[col]->u.next->down = temp;
        hdnode[col]->u.next = temp;
    }
}

```

```
// close last row
last->right = hdnode[current_row];
// close all column lists
for (i=0; i<num_cols; i++)
    hdnode[i]->u.next->down = hdnode[i];
// link all head nodes together
for (i=0; i<num_heads-1; i++)
    hdnode[i]->u.next = hdnode[i+1];
hdnode[num_heads-1]->u.next = node;
node->right = hdnode[0];
}
return node;
}
```

```
// print out the matrix in row major form
void mwrite(matrix_pointer node)
{
    int i;
    matrix_pointer temp, head = node->right;

    for (i=0; i<node->u.entry.row; i++) {
        for (temp = head->right; temp != head;
             temp = temp->right)
            printf(temp->u.entry.row, temp->u.entry.col,
                  temp->u.entry.value);
        head = head->u.next;
    }
}
```



```

void merase(matrix_pointer *node)
{
    int i, num_heads;
    matrix_pointer x,y, head = (*node)->right;

    for (i=0; i<(*node)->u.entry.row; i++) {
        y = head->right;
        while (y != head) {
            x = y; y = y->right; free(x);
        }
        x = head; head = head->u.next; free(x);
    }
    // free remaining head nodes
    y = head;
    while (y != *node) {
        x = y; y = y->u.next; free(x);
    }
    free(*node); *node = NULL;
}

```

- **Analysis of *mread* : [Program 4.24]**
 $O(\max\{num_rows, num_cols\} + num_terms)$
 $= O(num_rows + num_cols + num_terms).$
- **Analysis of *mwrite* : [Program 4.26]**
 $O(num_rows + num_terms).$
- **Analysis of *merase* : [Program 4.27]**
 $O(num_rows + num_cols + num_terms).$

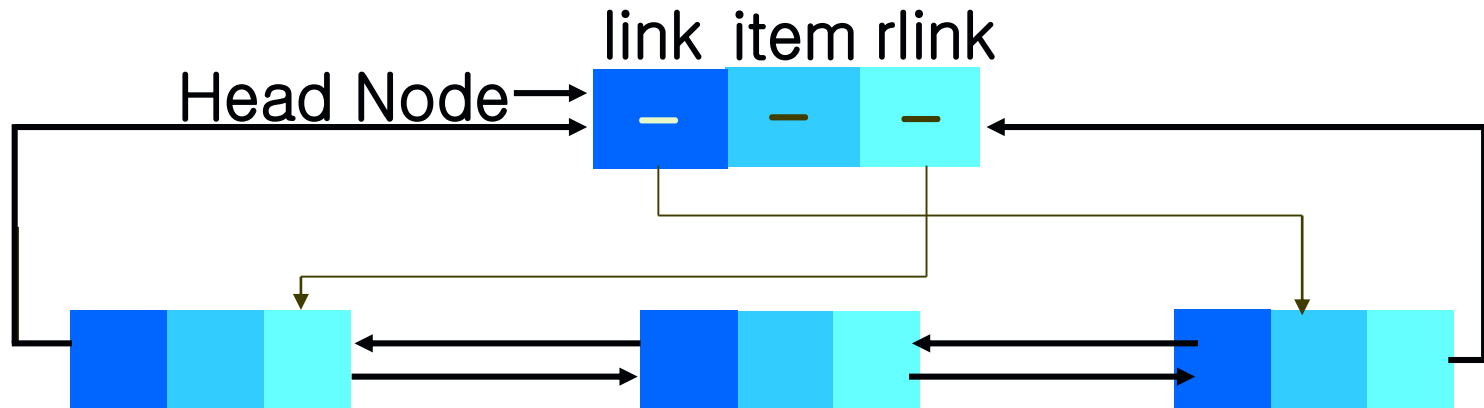
4.8 DOUBLY LINKED LISTS

- Singly linked lists pose problems because we can move easily only in the direction of the links.
- Whenever we have a problem that requires us to move in either direction, it is useful to have doubly linked lists.

- The necessary declarations are :

```
typedef struct node *node_pointer;  
typedef struct node {  
    node_pointer llink;  
    element item;  
    node_pointer rlink;  
};
```

- A doubly linked list may or may not be circular.
- **[Figure 4.23] Doubly linked circular list with head node**



- **[Figure 4.24] Empty doubly linked circular list with head node**



- Now suppose that *ptr* points to any node in a doubly linked list.

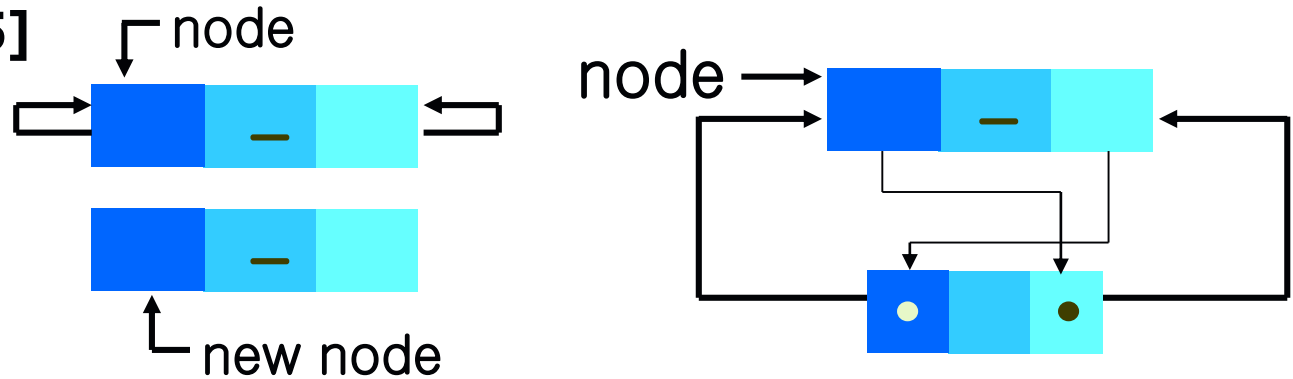
Then :

$$ptr == ptr \rightarrow llink \rightarrow rlink == ptr \rightarrow rlink \rightarrow llink$$

- **Insertion into a doubly linked circular list :**
- **[Program 4.28]**

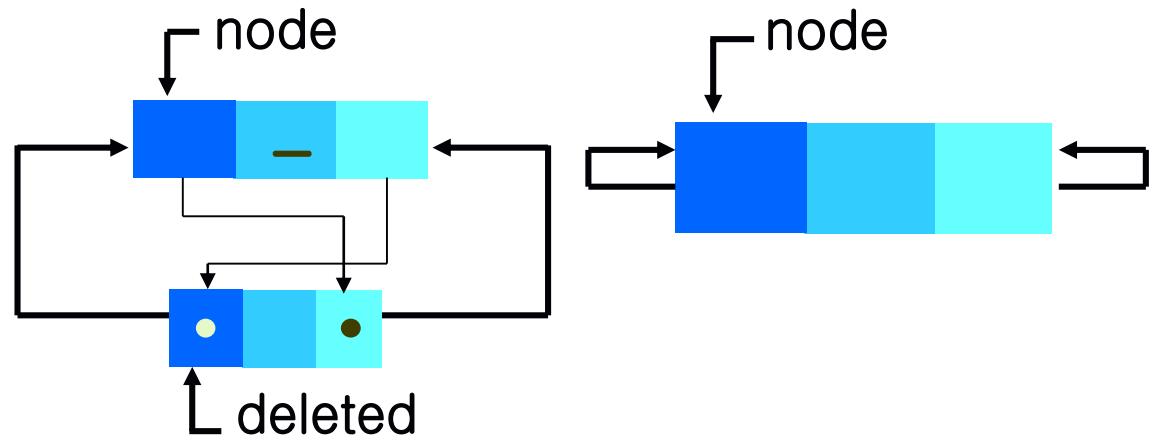
```
void dinsert(node_pointer node, node_pointer newnode)
{
    /* insert newnode to the right of node */
    newnode->llink = node;
    newnode->rlink = node->rlink;
    node->rlink->llink = newnode;
    node->rlink = newnode;
}
```

[Figure 4.25]



- **Deletion from a doubly linked circular list :**
- **[Program 4.29]**

```
void ddelete(node_pointer node, node_pointer deleted) {
    /* delete from the doubly linked list */
    if (node == deleted)
        printf("Deletion of head node not permitted.\n");
    else {
        deleted->llink->rlink = deleted->rlink;
        deleted->rlink->llink = deleted->llink;
        free(deleted);
    }
}
```



[Figure 4.26]

Chapter 5 : TREES

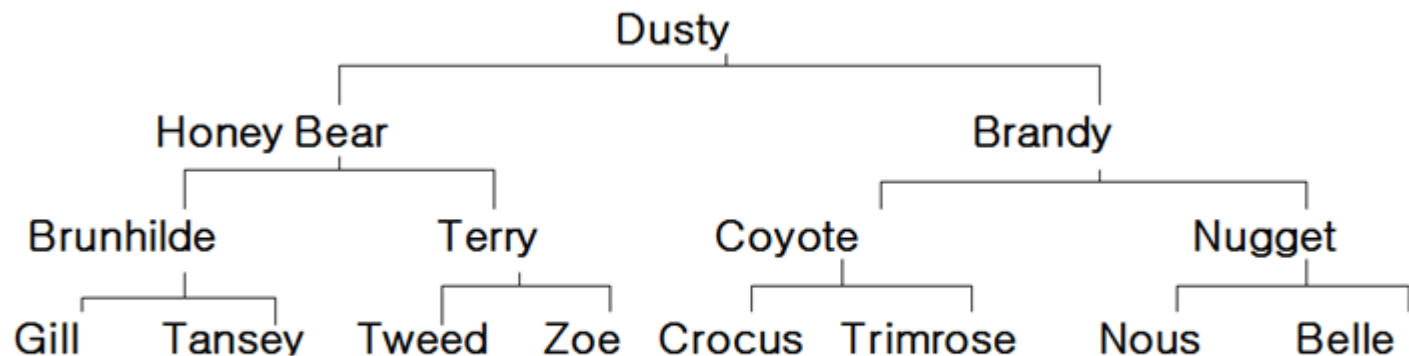
Data Structures Lecture Note
Prof. Jihoon Yang
Data Mining Research Laboratory

5.1 INTRODUCTION

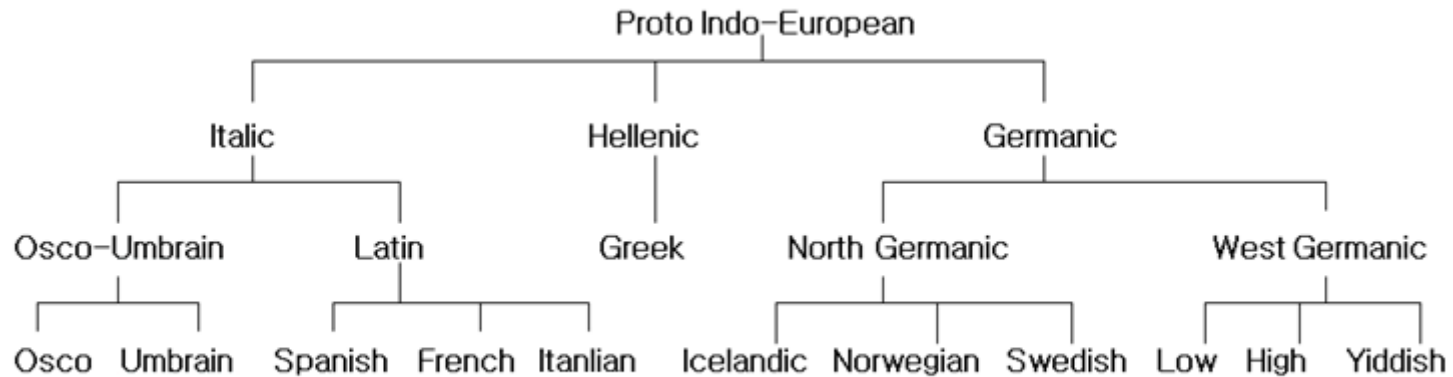
5.1.1 Terminology

The intuitive concept of a tree implies that we organize the data

[Figure 5.1] Two types of genealogical charts



(a) Pedigree



(b) Lineal

Definition : 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, \dots, T_n where each of these sets is a tree
- We call T_1, \dots, T_n the subtrees of the root.

Terms used when referring to trees:

- 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.
- 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.

Terms used when referring to trees:

- 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)))

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

<i>data</i>	<i>link 1</i>	<i>link 2</i>	<i>. . .</i>	<i>link n</i>
-------------	---------------	---------------	--------------	---------------

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

5.1.2 Representation Of Trees

Left Child-Right Sibling Representation

The representations we consider require exactly two link or pointer fields per node.

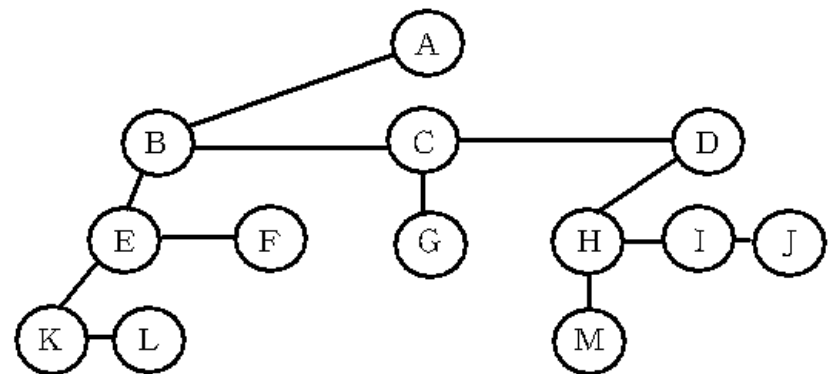
Note that every node has only one leftmost child and one closest right sibling.

(* Strictly speaking, the order of children in a tree is not important. *)

[Figure 5.4]

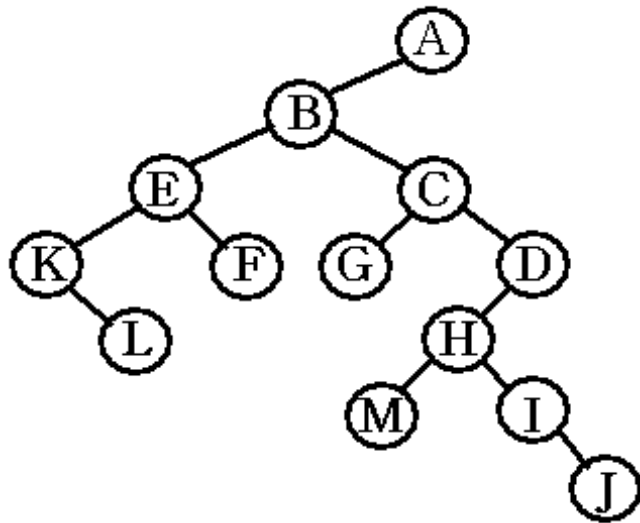
data	
left child	right sibling

[Figure 5.5]

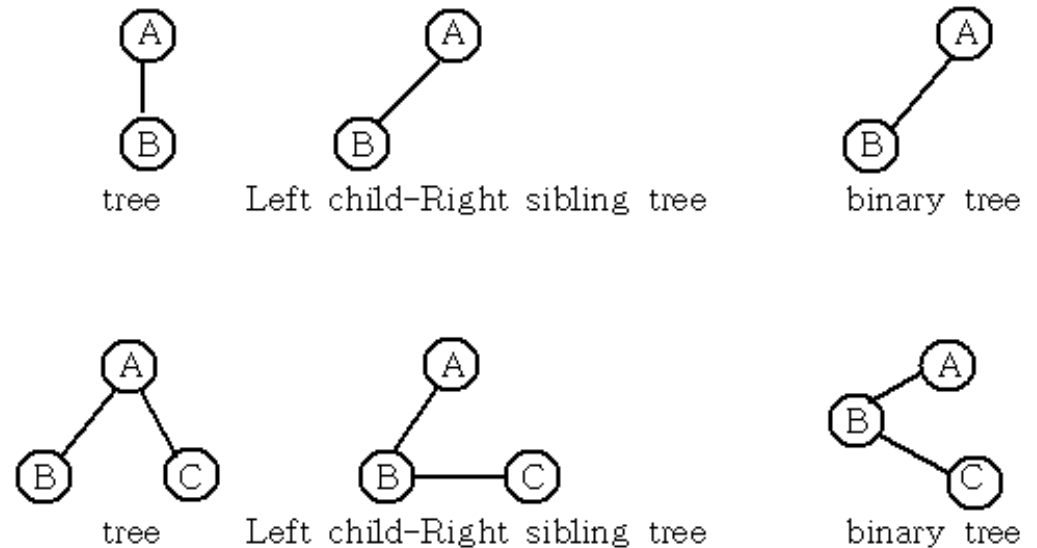


Representation As A Degree Two Tree

[Figure 5.6]



[Figure 5.7]



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.

[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.

Example : [Figure 5.8] Two different binary trees

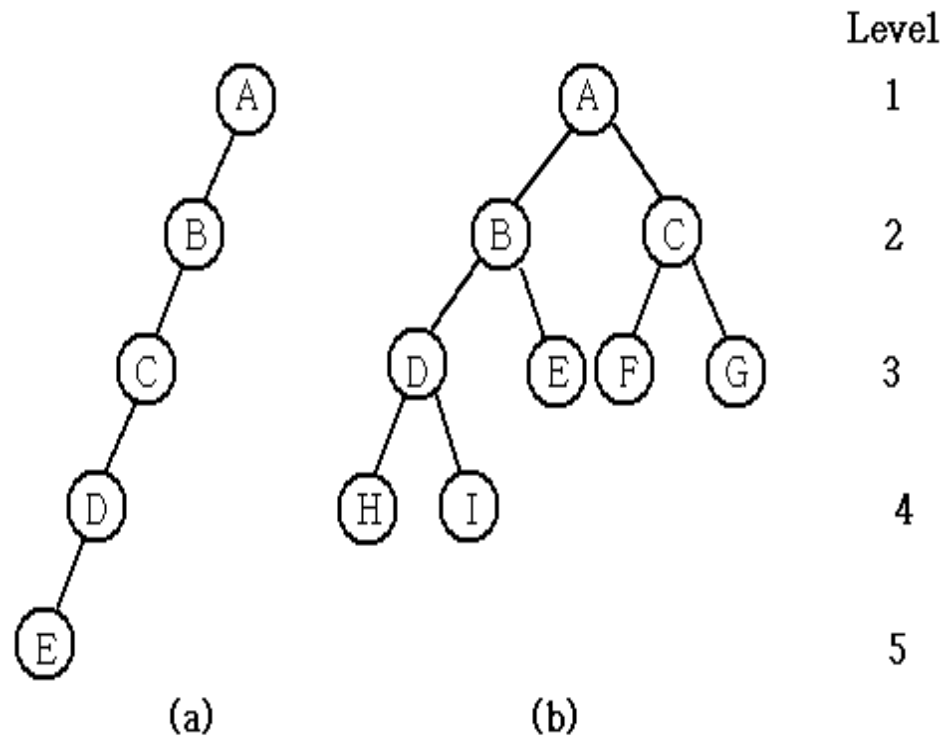


Special Types of binary trees

Skewed tree, Complete binary tree

(Figure 5.9)

[Figure 5.6]



The same terminology we used to describe trees applies to binary trees.

- node,
- degree of a node, degree of a tree,
- leaf or terminal,
- parent, children (left child, right child), sibling,
- ancestor, descendant,
- level of a node, height or depth,

Structure 5.1 : Abstract data type Binary_Tree.

- Structure 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 $bt, bt1, bt2 \in \text{BinTree}$, $item \in \text{element}$

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

Boolean IsEmpty(bt) ::= **if** ($bt == \text{empty binary tree}$) **return** *TRUE*
else return *FALSE*

BinTree MakeBT($bt1, item, bt2$) ::= **return** a binary tree whose
left subtree is $bt1$, whose right
subtree is $bt2$, 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

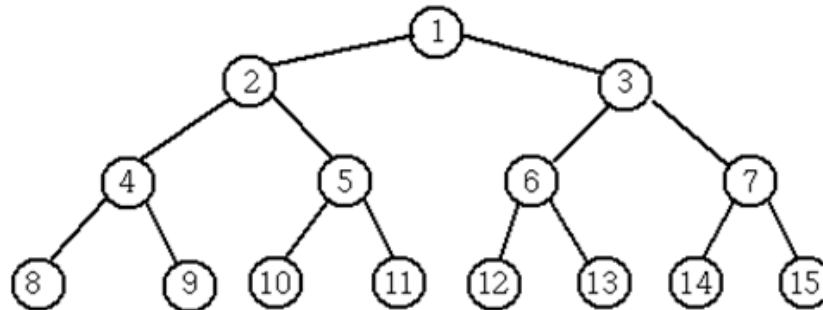
Lemma 5.1 [Maximum number of nodes]:

- (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 $2^{k+1} - 1$ 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.

[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 , $1 \leq i \leq n$, we have :

(1) $\text{parent}(i)$ is at $\lfloor i/2 \rfloor$ if $i \neq 1$.

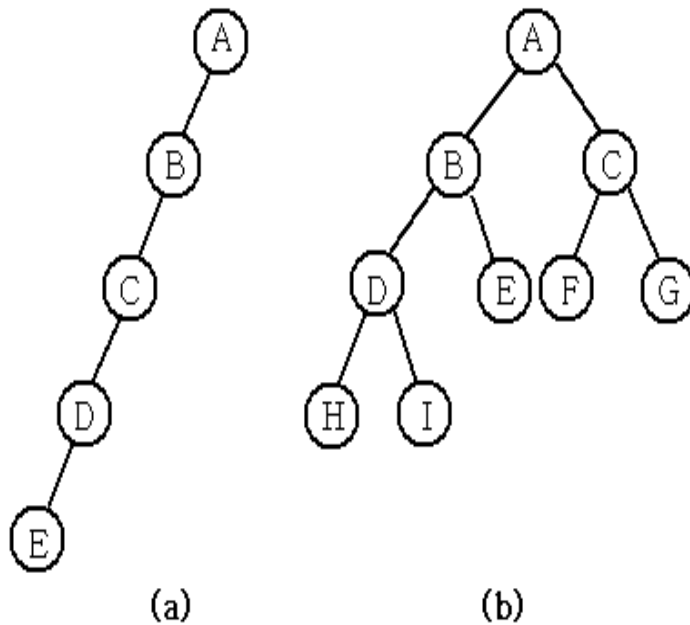
If $i = 1$, i is at the root and has no parent.

(2) $\text{left_child}(i)$ is at $2i$ if $2i \leq n$. If $2i > n$, then i has no left child.

(3) $\text{right_child}(i)$ is at $2i+1$ if $2i+1 \leq n$.

If $2i+1 > n$, then i has no right child.

[Figure 5.9]



[Figure 5.11]

Level

1

2

3

4

5

[1]	A
[2]	B
[3]	-
[4]	C
[5]	-
[6]	-
[7]	-
[8]	D
[9]	-
.	.
.	.
.	.
[16]	E

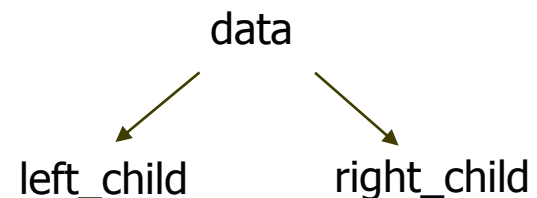
[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

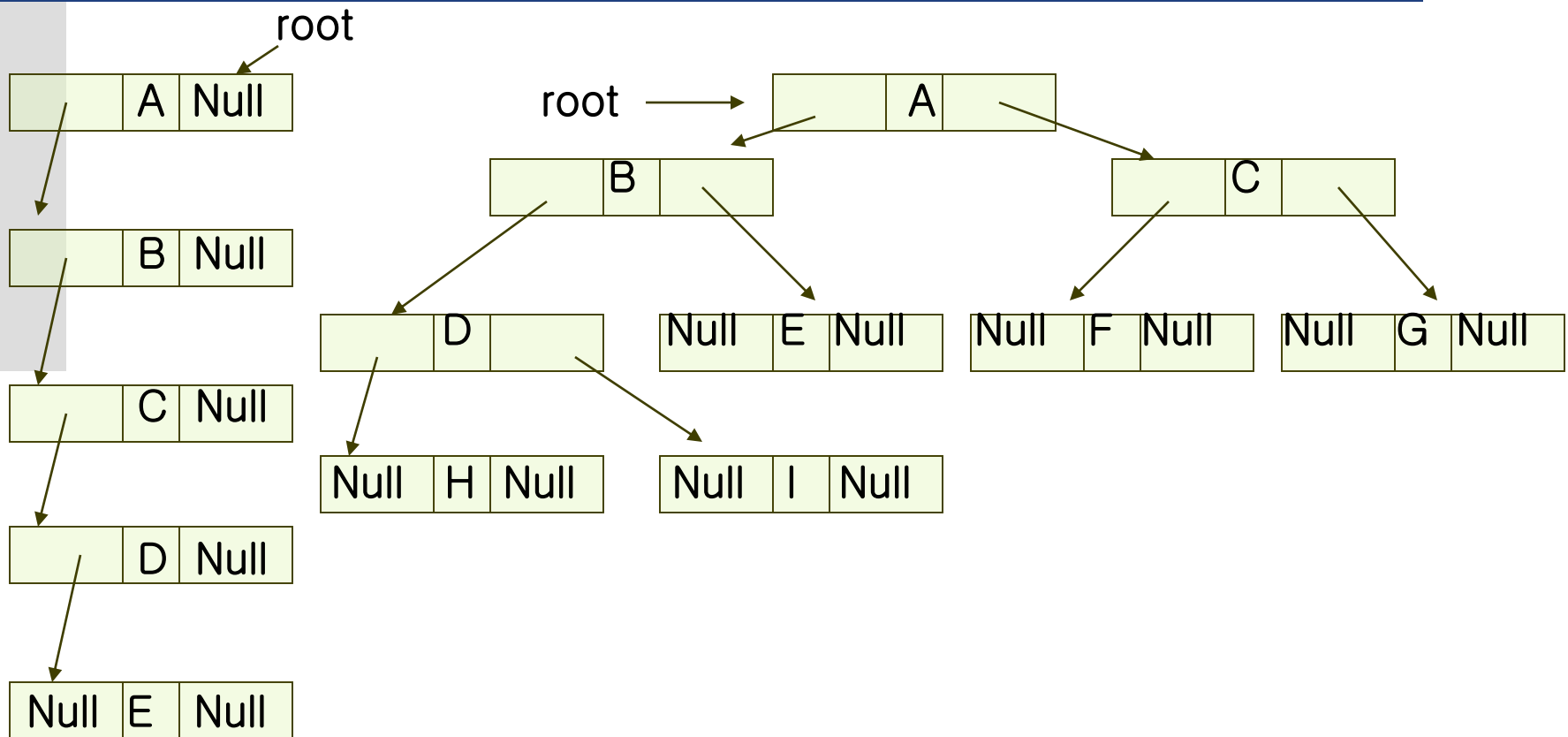
Linked Representation

- Node structure :

```
typedef struct node *tree_pointer;  
typedef struct node {  
    int data;  
    tree_pointer left_child, right_child;  
};
```

[Figure 5.12] Node representation for binary trees



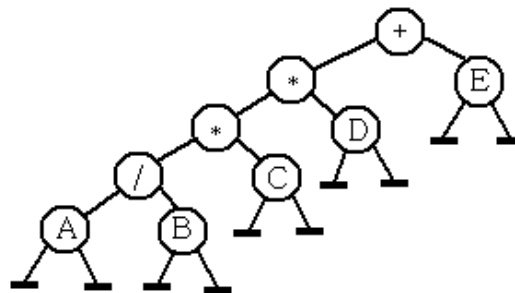


[Figure 5.13] The linked representation for the trees in Figure 5.9.

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.

- 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

```
void inorder (tree_pointer ptr)
/* inorder tree traversal */
{
    if (ptr) {
        inorder (ptr -> left_child);
        printf ("%d", ptr -> data);
        inorder (ptr -> right_child);
    }
}
```

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

[Figure 5.16]

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

■ Preorder Traversal

```
void preorder (tree_pointer ptr)
/* preorder tree traversal */
{
    if (ptr) {
        printf ("%d", ptr -> data);
        preorder (ptr -> left_child);
        preorder (ptr -> right_child);
    }
}
```

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

■ Postorder Traversal

```
void postorder (tree_pointer ptr)
/* postorder tree traversal */
{
    if (ptr) {
        postorder (ptr -> left_child);
        postorder (ptr -> right_child);
        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.


```
void iter_inorder(tree_pointer node)
{
    int top = -1; /* initialize stack */
    tree_pointer stack[MAX_STACK_SIZE];
    for ( ; ; ) {
        for ( ; node; node = node -> left_child)
            add(&top, node); /* add to stack */
        node = delete(&top); /* delete from stack */
        if (!node) break; /* empty stack */
        printf ("%d", node -> data);
        node = node -> right_child;
    }
}
```

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 5.10.

```
void level_order(tree_pointer ptr)
/* level order tree traversal */
```

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

```
{
    int front = rear = 0;
    tree_pointer queue[MAX_QUEUE_SIZE];
    if (!ptr) return; /* empty tree */
    addq(front, &rear, ptr);
    for ( ; ; ) {
        ptr = deleteq(&front, rear); /*empty list returns NULL*/
        if (ptr) {
            printf("%d", ptr->data);
            if (ptr->left_child)
                addq(front, &rear, ptr->left_child);
            if (ptr->right_child)
                addq(front, &rear, ptr->right_child);
        }
        else break;
    }
}
```

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)

■ [Program 5.6]

```
tree_pointer copy(tree_pointer original)
/* this function returns a tree_pointer to an exact copy
   of the original tree */
{
    tree_pointer temp;
    if (original) {
        temp = (tree_pointer) malloc(sizeof(node));
        if (IS_FULL(temp)) {
            fprintf(stderr, "The memory is full\n");
            exit(1);
        }
        temp->left_child = copy(original->left_child);
        temp->right_child = copy(original->right_child);
        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(tree_pointer first, tree_pointer 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->left_child, second->left_child) &&
        equal(first->right_child, second->right_child)))
}
```

■ The Satisfiability Problem

Consider the formulas constructed by taking variables

x_1, x_2, \dots, 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 .

Consider an expression :

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

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

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

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

A first version of satisfiability algorithm:

[Program 5.8]

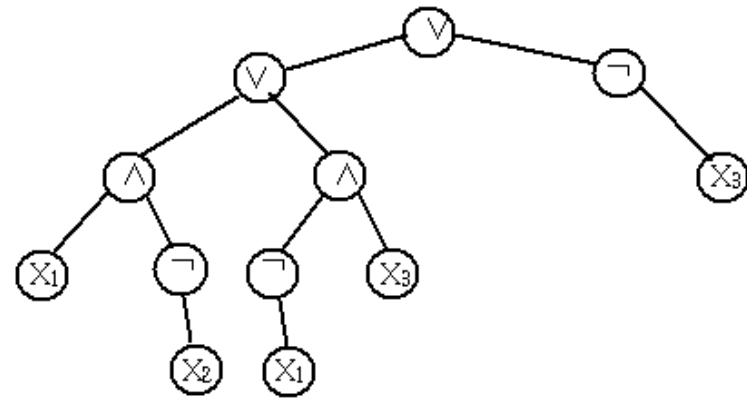
```
for (all  $2^n$  possible combinations) {  
    generate the next combination;  
    replace the variables by their values;  
    evaluate the expression;  
    if (its value is true) {  
        printf(<combination>);  
        return;  
    }  
}  
printf("No satisfiable combination□n");
```

<Evaluating an propositional formula>

Assume that our formula is already in a binary tree.

For a formula :

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



[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.

[Figure 5.19]

left_child	data	value	right_child
------------	------	-------	-------------

```
typedef enum {not, and, or, true, false} logical;  
typedef struct node *tree_pointer;  
typedef struct node {  
    tree_pointer left_child;  
    logical      data;  
    short int    value;  
    tree_pointer right_child;  
};
```

[Program 5.9]

```
void post_order_eval(tree_pointer node)
{
    /* modified postorder traversal to evaluate
    a propositional calculus tree */
    if (node) {
        post_order_eval(node->left_child);
        post_order_eval(node->right_child);
        switch(node->data) {
            case not : node->value =
                        !node->right_child->value;
                        break;
            case and : node->value =
                        node->right_child->value && node->left_child->value;
                        break;
            case or : node->value =
                        node->right_child->value || node->left_child->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 $2n$ links and
among them, $(n+1)$ are NULL links.

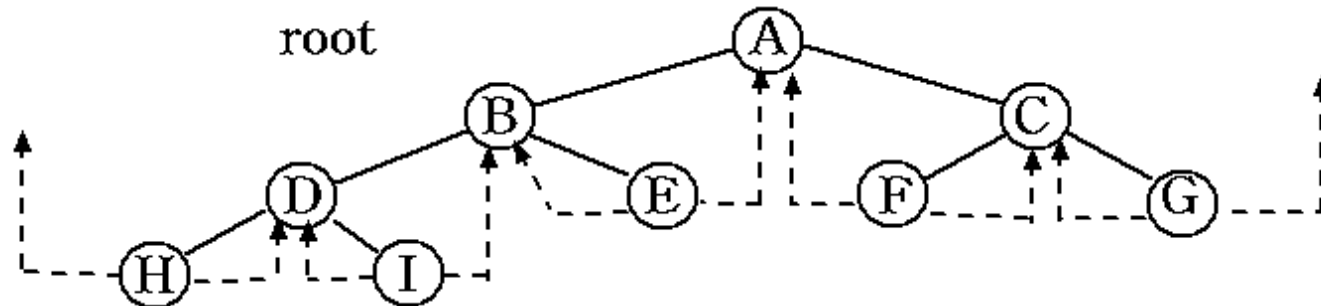
A.J. Perlis and C. Thornton have devised a clever way
to make use of these null links.

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 $ptr \rightarrow left_child$ is null, replace $ptr \rightarrow left_child$ 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 $ptr \rightarrow right_child$ is null, replace $ptr \rightarrow right_child$ 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 .

[Figure 5.21]

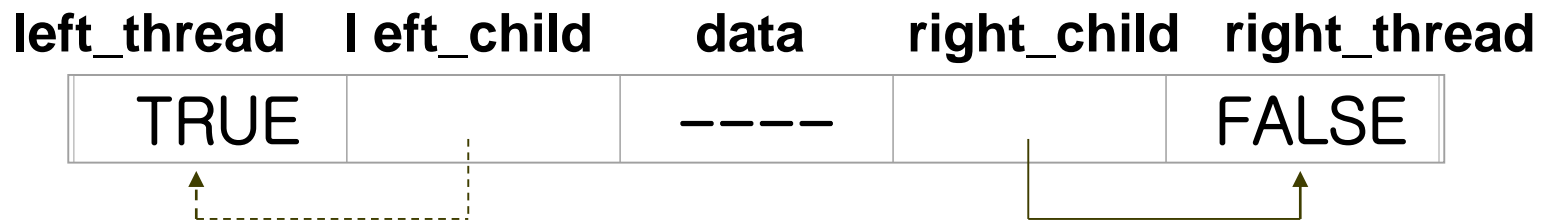


When we represent the tree in memory,
we must be able to distinguish between threads and normal pointers.
This is done by adding two additional fields
to the node structure, *left_thread* and *right_thread*.

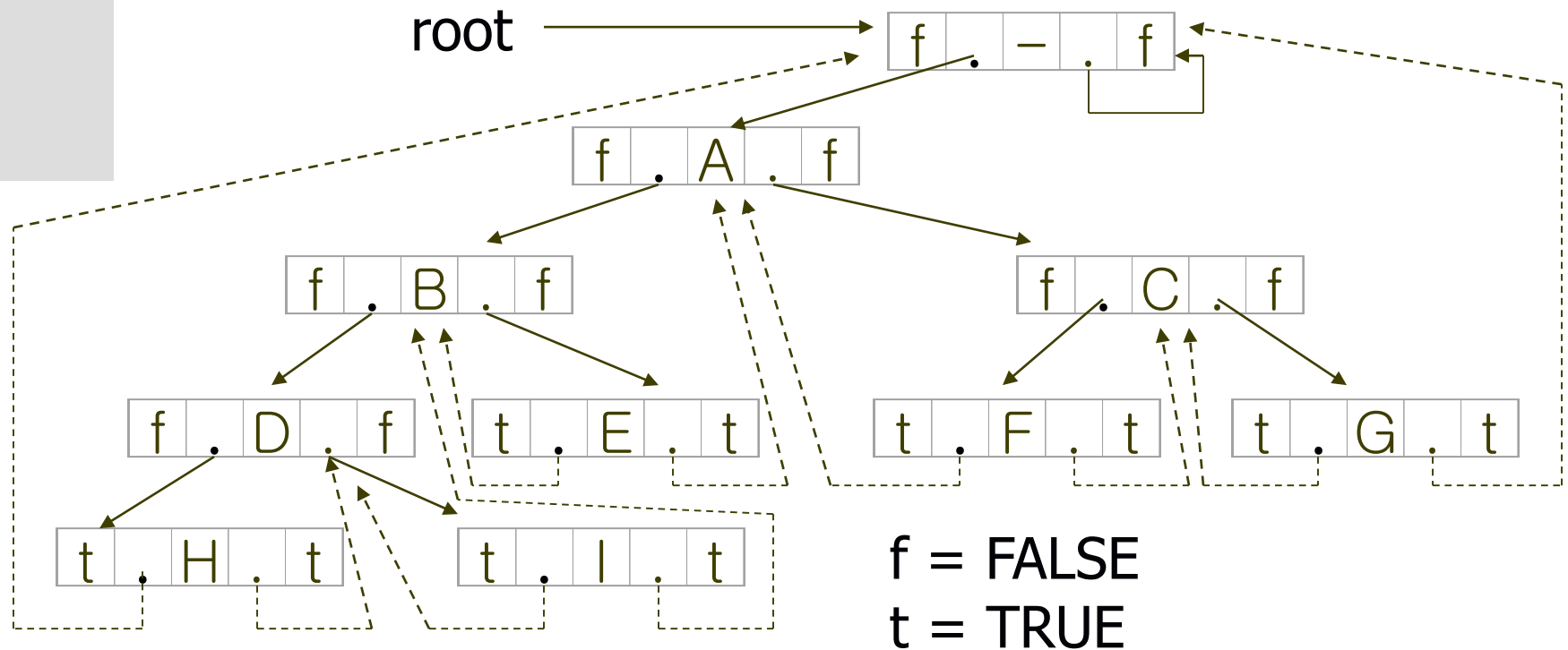
```
typedef struct threaded_tree *threaded_pointer;
typedef struct threaded_tree {
    short int left_thread;
    threaded_pointer left_child;
    char data;
    threaded_pointer right_child;
    short int right_thread;
};
```

We assume that all threaded binary trees have a head node.

[Figure 5.22] An empty threaded tree.



[Figure 5.23]



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;
}
```

To perform an inorder traversal we make repeated calls to insucc.

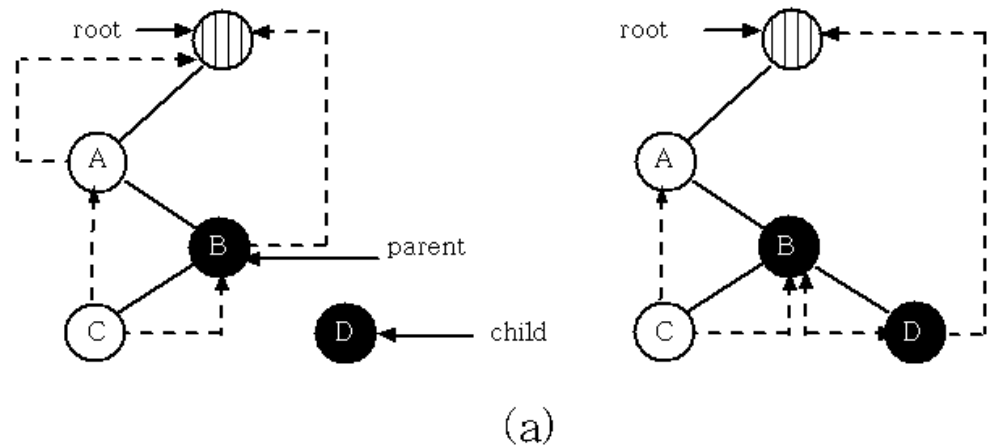
[Program 5.11] : Inorder traversal of a threaded binary tree.

```
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*.

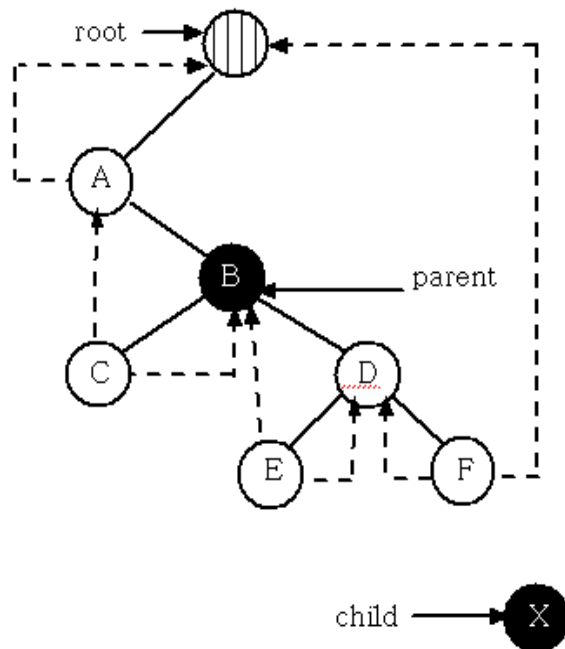
[Figure 5.24] (a)



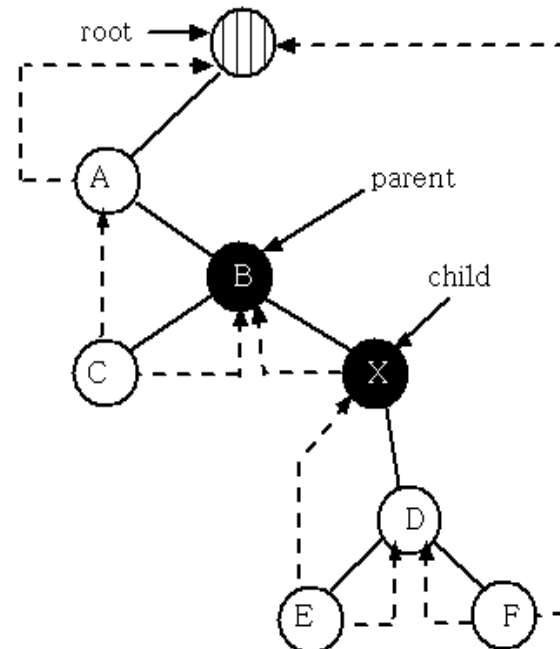
To do this we must :

- (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*

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



before



after

(b)

C code which handles both cases.

[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

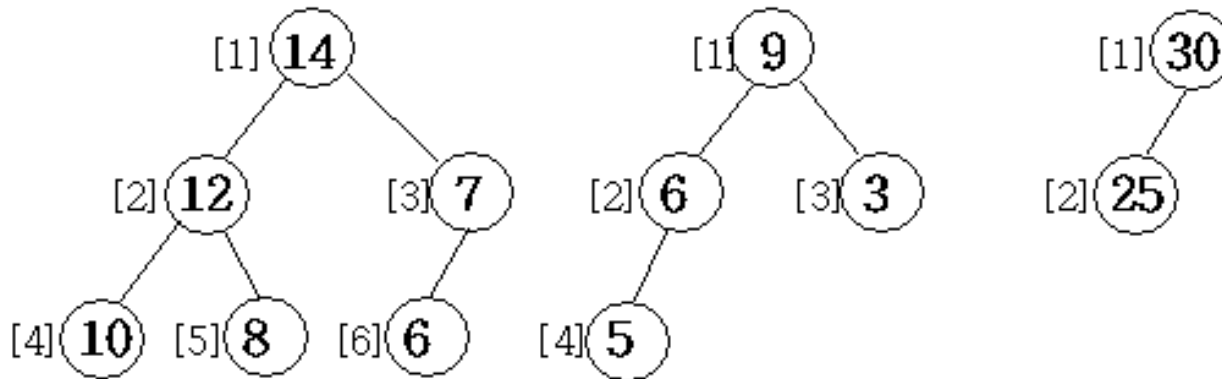
Definition : A *max tree* is a tree in which the key value in each node is no smaller than the key values in its children (if any).

A *max heap* is a complete binary tree that is also a max tree.

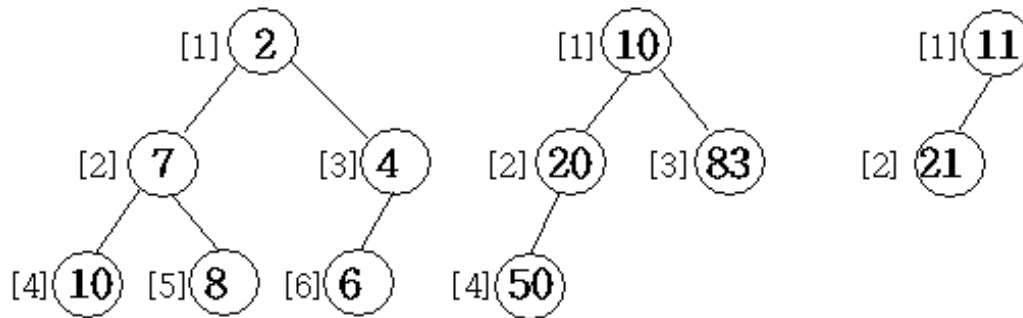
Definition : A *min tree* is a tree in which the key value in each node is no larger than the key values in its children (if any).

A *min heap* is a complete binary tree that is also a min tree.

[Figure 5.25] sample max heaps



[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

[Structure 5.2] : Abstract data type MaxHeap.

Structure MaxHeap is

object: a complete binary tree of $n \geq 0$ 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 ($n == \text{max_size}$) 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 ($n \leq 0$) 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.

Implementing priority queues :

Heaps are used as an efficient implementation of the priority queues. To examine some of the other representations see Figure 5.27.

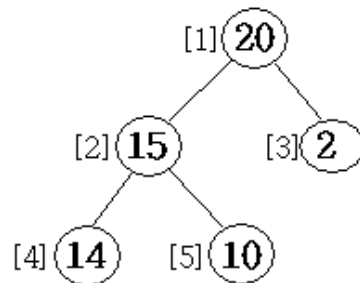
Representation	Insertion	Deletion
Unordered array	$\Theta(1)$	$\Theta(n)$
Unordered linked list	$\Theta(1)$	$\Theta(n)$
Sorted array	$O(n)$	$\Theta(1)$
Sorted linked list	$O(n)$	$\Theta(1)$
Max heap	$O(\log_2 n)$	$O(\log_2 n)$

[Figure 5.27]

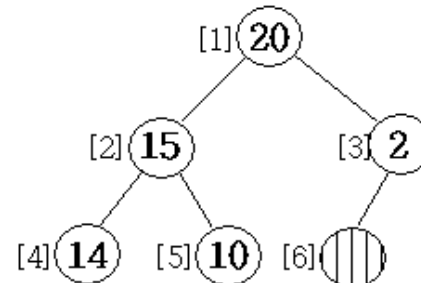
5.6.3 Insertion Into A Max Heap

To illustrate the insertion operation, See Figure 5.28

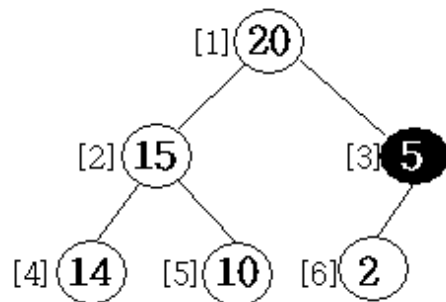
[Figure 5.28]



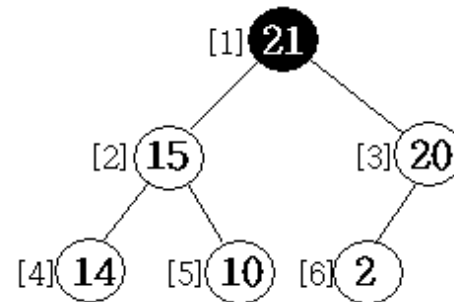
(a) before heap insertion



(b) initial location of new node



(c) insertion 5 into heap(a)



(d) insertion 21 into heap(a)

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.

[Program 5.13] Insertion into a max heap

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

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.

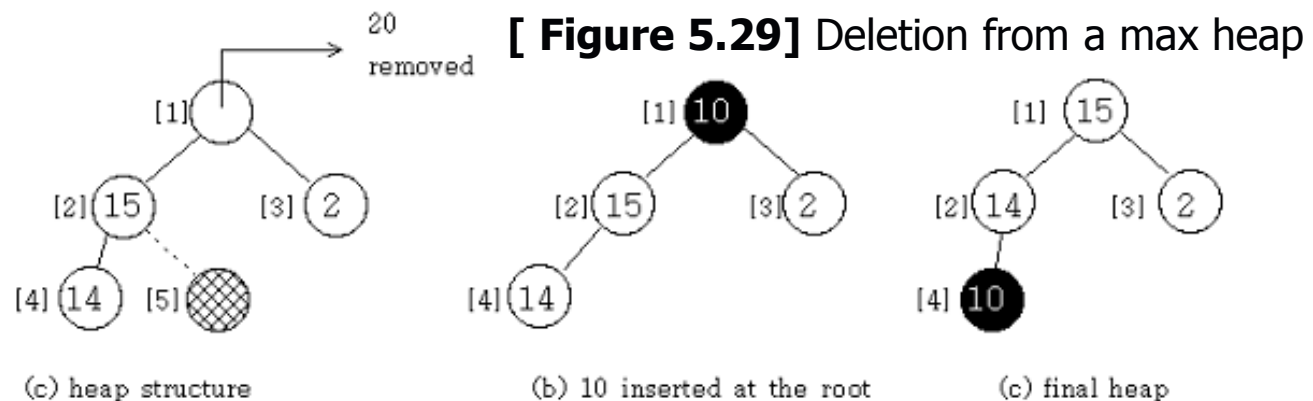
Hence the time complexity is $O(\log_2 n)$.

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.



[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;  
}
```

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 $\lceil \log_2(n+1) \rceil$,
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.

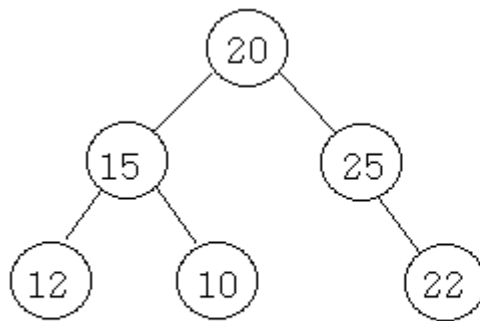
In fact, with a binary search tree we can perform these operations by both key value (e.g., delete the element with key x) and by rank (e.g., delete the fifth smallest element).

Definition: A *binary search tree* is a binary tree. It may be empty.

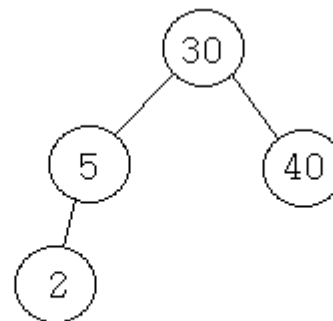
If it is not empty, it satisfies the following properties :

- (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. □

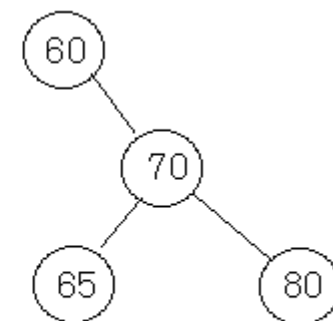
[Figure 5.30] some sample binary trees



(a)



(b)



(c)

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);
}
```

[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;
}
```


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

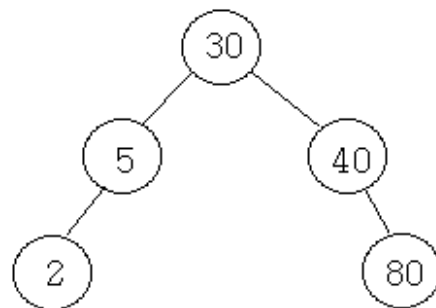
To insert a new element, key :

First, we verify that the key is different from those of existing elements by searching the tree.

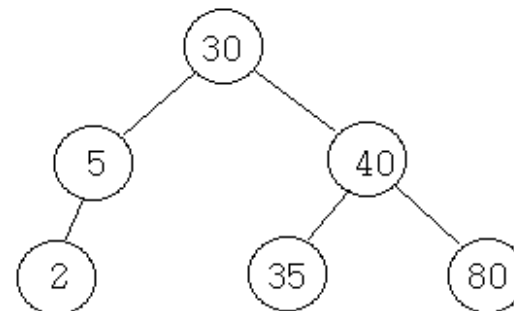
If the search is unsuccessful,

then we insert the element at the point the search terminated.

[Figure 5.31]



(a) Insert 80



(b) Insert 35

[Program 5.17] : Inserting an element 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;
}
}
```

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

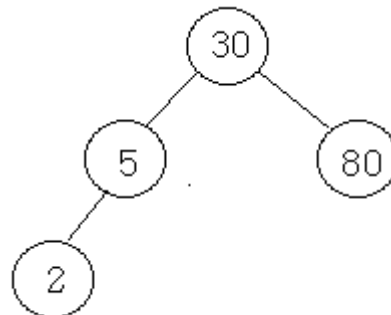
• Deletion of a leaf node :

Set the corresponding child field of its parent to NULL
and free the node.

• Deletion of a nonleaf node with single child :

Erase the node and
then place the single child in the place of the erased node.

[Figure 5.32] Deletion from a binary tree



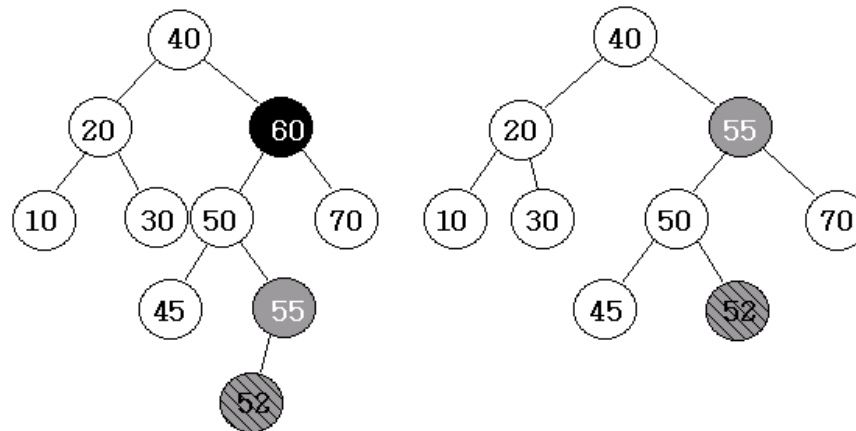
· **Deletion of a nonleaf node with two children :**

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.

5.10 SET REPRESENTATION

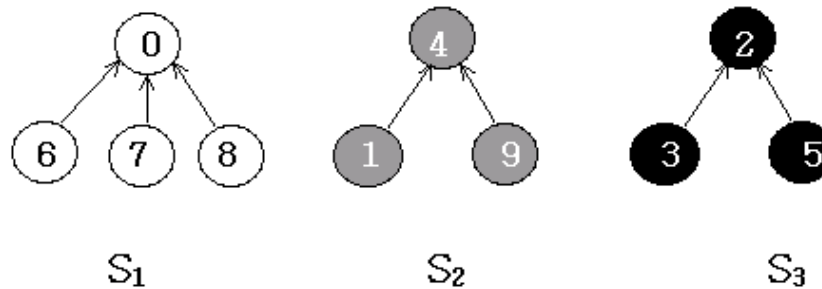
We study the use of trees in the representation of sets.

For simplicity, we assume that the elements of the sets

are the numbers $0, 1, 2, \dots, n-1$.

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.

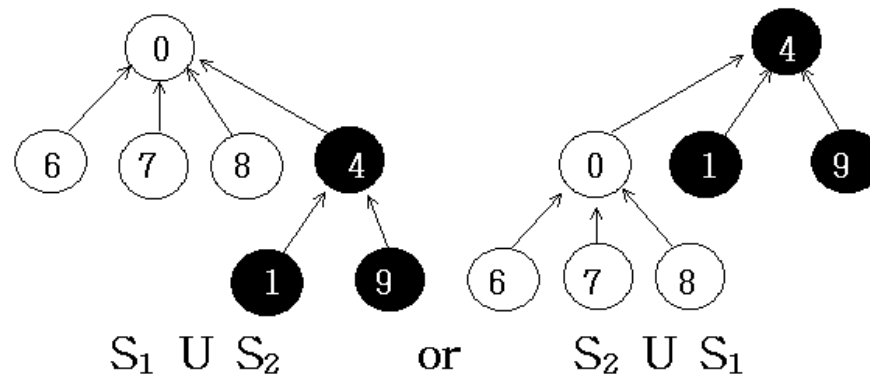
The operations to perform on these sets are:

- (1) *Disjoint set union*. If we wish to get the union of two disjoint sets S_i and S_j , replace S_i and S_j by $S_i \cup S_j$.
- (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 S_1 and S_2 . We simply make one of the trees a subtree of the other. $S_1 \cup S_2$ could have either of the representations of Figure 5.40

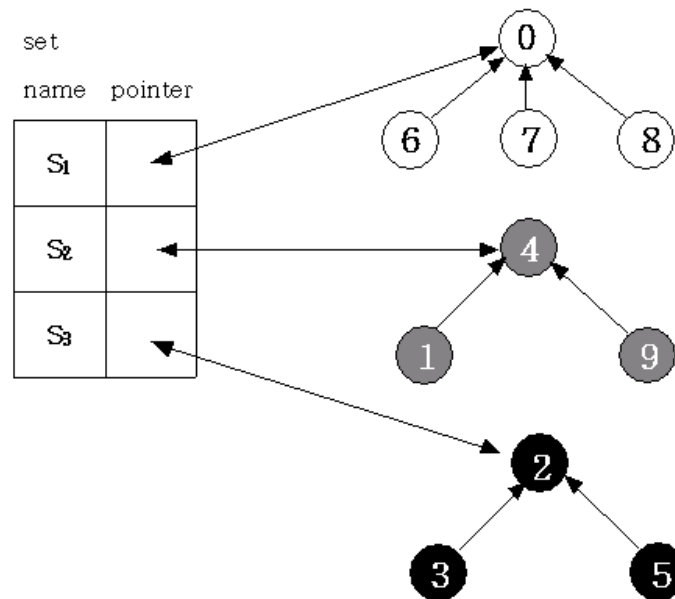
[Figure 5.40] Possible representation of $S_1 \cup S_2$



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



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 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]
<i>parent</i>	-1	4	-1	2	-1	2	0	0	0	4

Notice that root nodes have a parent of -1 .

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;
}
```

Analysis of *union1* and *find1* :

Let us process the following sequence of union-find operations:

$\text{union}(0, 1), \text{find}(0)$

$\text{union}(1, 2), \text{find}(0)$

.

.

.

$\text{union}(n-2, n-1), \text{find}(0)$

This sequence produces the degenerate tree of Figure 5.43.



[Figure 5.43] Degenerate tree

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 :

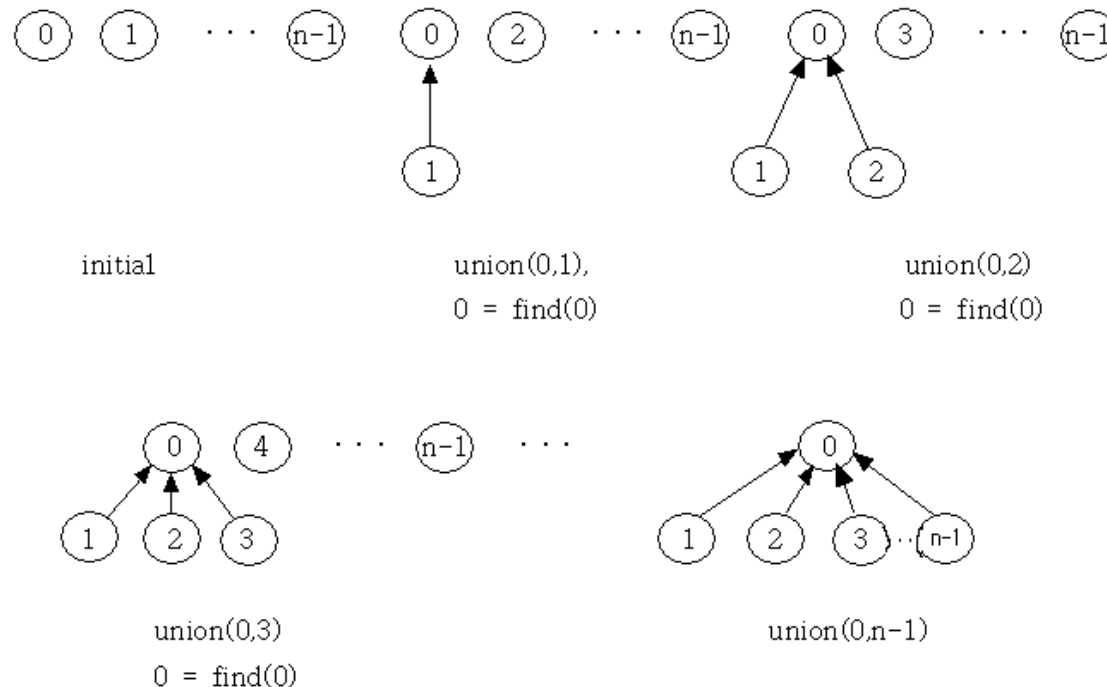
$$\sum_{i=1}^{n-1} i = O(n^2)$$

By avoiding the creation of degenerate trees,
we can attain far more efficient implementations
of the 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 . \square

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



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.

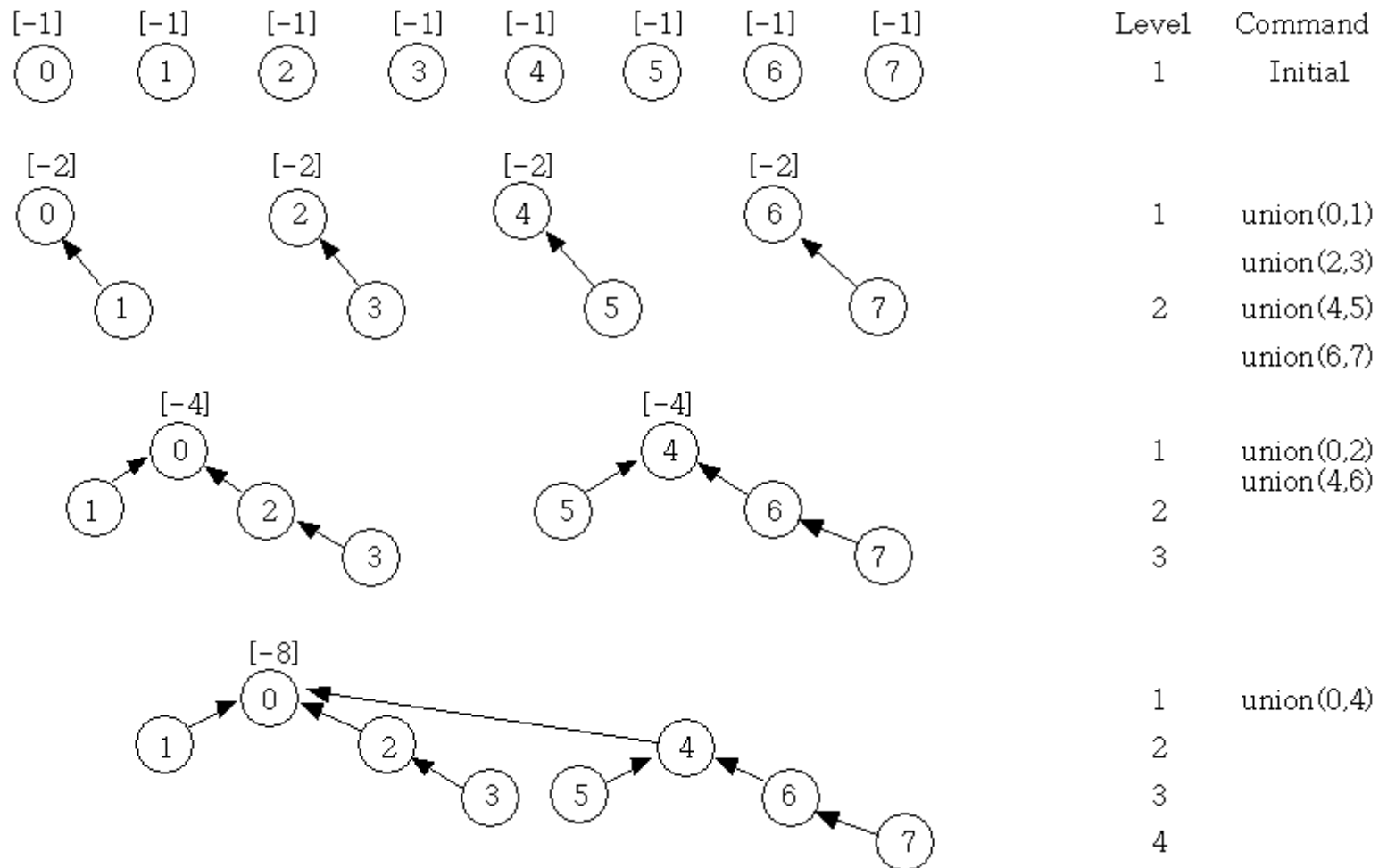
```
void union2(int i, int j)
{
    /* parent[i] = -count[i] and parent[j] = -count[j] */
    int temp = parent[i] + parent[j];
    if (parent[i] > parent[j]) {
        parent[i] = j; /* make j the new root */
        parent[j] = temp;
    }
    else {
        parent[j] = i; /* make i the new root */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

Lemma 5.4 : Let T be a tree with n nodes created as a result of *union2*.
Then the depth of $T \leq \lfloor \log_2 n \rfloor + 1$

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 5.43 shows the result.



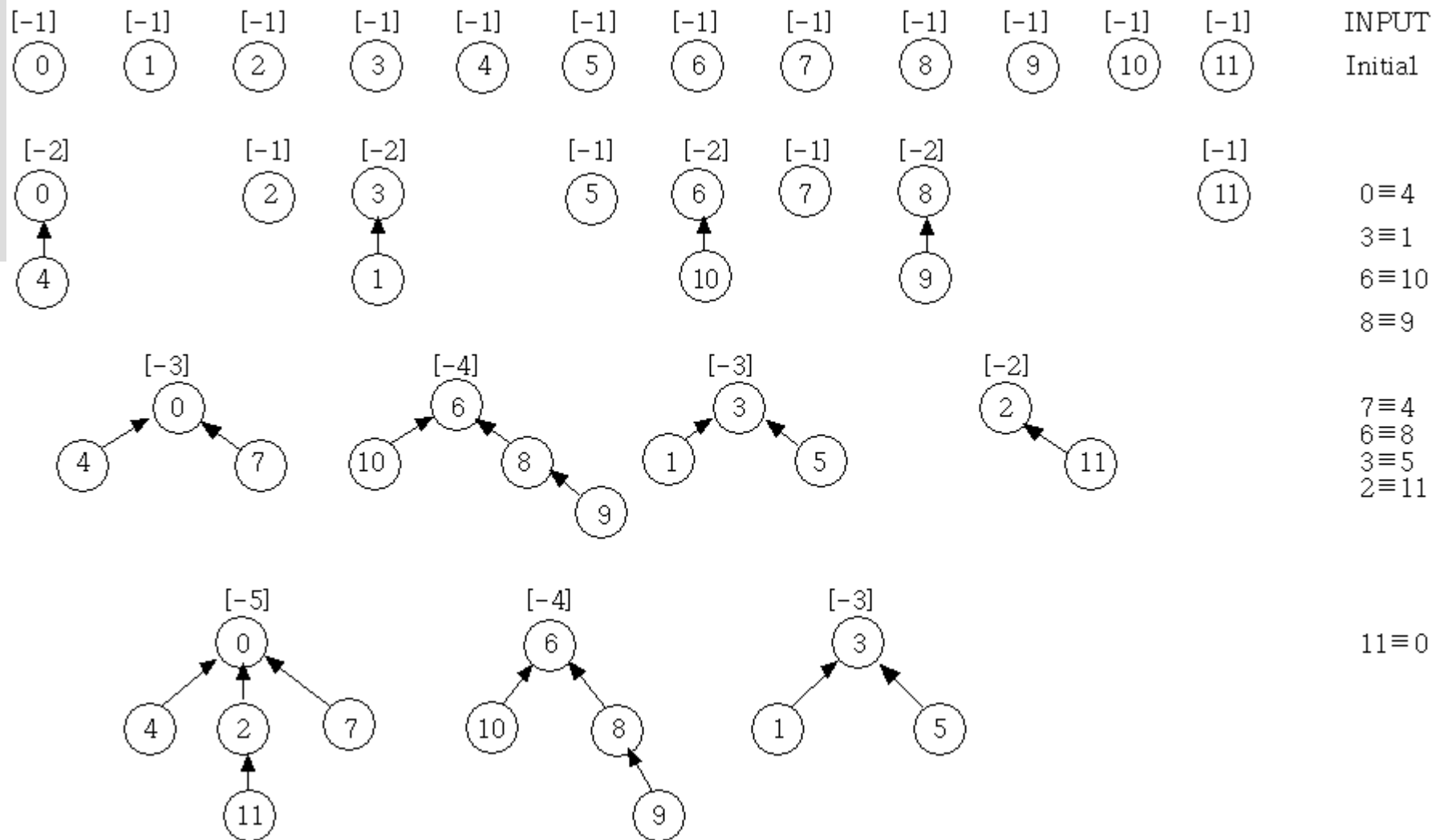
As is evident from this example, in the general case, the depth can be $\lfloor \log_2 n \rfloor + 1$ if the tree has n nodes. \square

As a result of Lemma 5.4,
the time to process a *find* in an n element tree is $O(\log_2 n)$.

If we process an intermixed sequence
of $n-1$ *union* and m *find* operations,
then the time becomes $O(n + m \log_2 n)$.

5.10.2 Equivalence Classes

[Figure 5.44] Trees for equivalence example



Chapter 6 GRAPHS

Data Structures Lecture Note
Prof. Jihoon Yang
Data Mining Research Laboratory

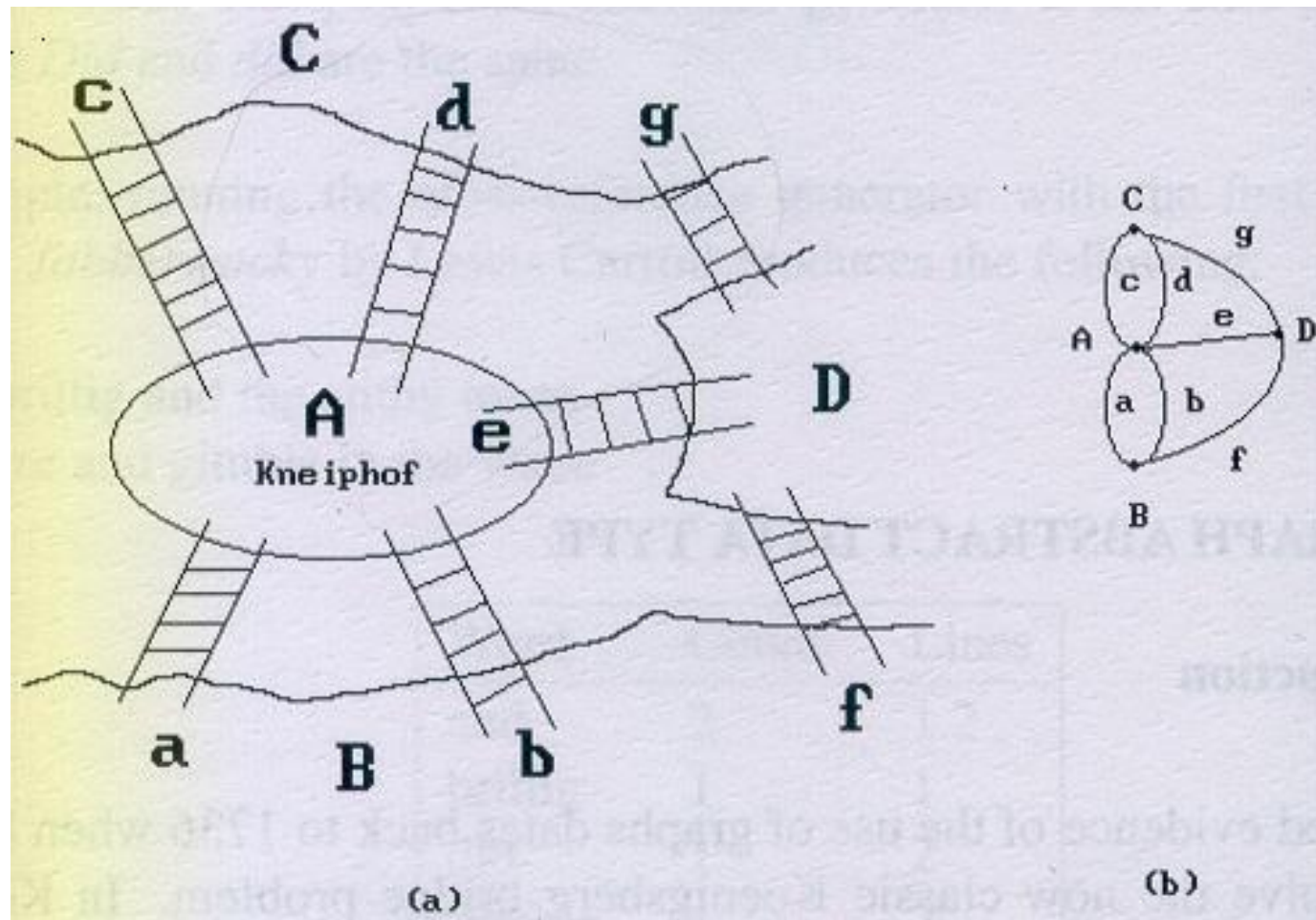
6.1 THE GRAPH ABSTRACT DATA TYPE

6.1.1 Introduction

The first recorded evidence of the use of graphs
dates back to 1736 when *Leonhard Euler* used them
to solve the classic *Koenigsberg* bridge problem.

See Figure 6.1.
(i.e., determining an Eulerian circuit(walk))

Euler proved that a graph has an *Eulerian walk*
iff the degree of each vertex is even.



Graphs are the most widely used mathematical structure

Applications

- Electrical circuit analysis
- Finding shortest routes
- Project planning
- Identification of chemical compounds
- Network flow design
- Gene/protein interactions, etc.

6.1.2 Definitions

a graph, G , consists of two sets :

$V(G)$, a finite, nonempty set of vertices, and

$E(G)$, a finite, possibly empty set of edges.

We may write $G=(V,E)$.

An undirected graph -- each edge is represented
as an unordered pair of vertices.

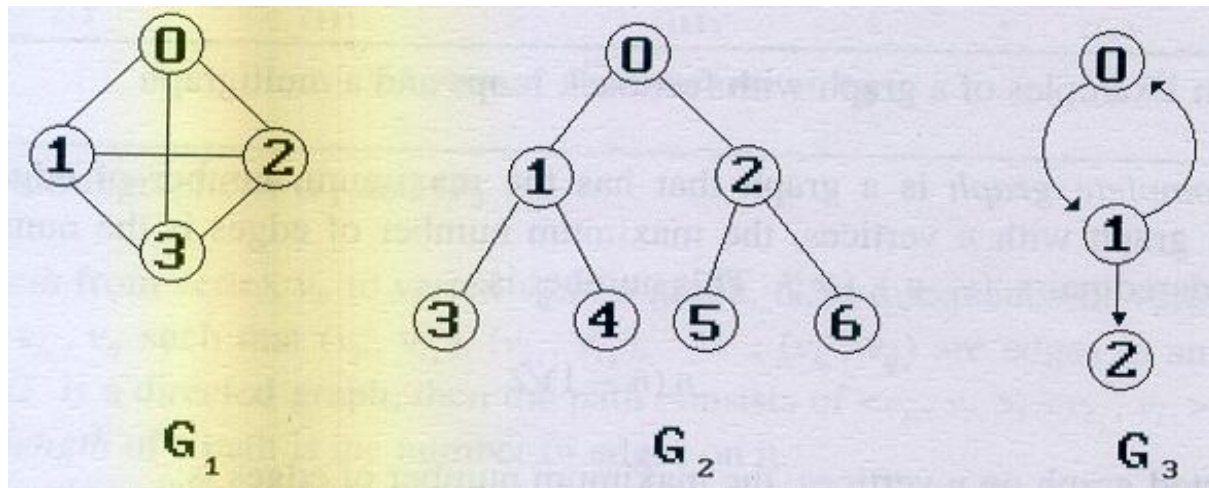
e.g., pairs (v_0, v_1) and (v_1, v_0) represent the same edge.

A directed graph -- each edge is represented
as a directed pair of vertices.

e.g., the pair $\langle v_0, v_1 \rangle$ represents an edge
in which v_0 is the tail and v_1 is the head.

Therefore $\langle v_0, v_1 \rangle$ and $\langle v_1, v_0 \rangle$ represent two different edges.

[Figure 6.2]



The set representation of each of these graphs is :

$$V(G_1) = \{0, 1, 2, 3\}$$

$$E(G_1) = \{(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)\}$$

$$V(G_2) = \{0, 1, 2, 3, 4, 5, 6\}$$

$$E(G_2) = \{(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)\}$$

$$V(G_3) = \{0, 1, 2\}$$

$$E(G_3) = \{<0, 1>, <1, 0>, <1, 2>\}$$

Notice that G_2 is a tree.

We can define trees as a special case of graphs.

Restrictions imposed on graphs :

1. No *self* loops - A graph may not have an edge from a vertex, i , back to itself.
2. No multiple edges - A graph may not have multiple occurrences of the same edge.

We refer a graph with multiple edge as a *multigraph*.

Complete graph --

For an undirected graph,
a complete graph with n vertices has
 $n(n-1)/2$ different edges.

For a directed graph,
a complete graph with n vertices has
 $n(n-1)$ different edges.

If (v_0, v_1) is an edge in an undirected graph,
 then the vertices v_0 and v_1 are *adjacent*
 and the edge (v_0, v_1) is *incident on* vertices v_0 and v_1 .

If $\langle v_0, v_1 \rangle$ is a directed edge,
 then vertex v_0 is *adjacent to* vertex v_1 ,
 while v_1 is *adjacent from* vertex v_0 .

The edge $\langle v_0, v_1 \rangle$ is *incident on* v_0 and v_1 .

A *subgraph* of G is a graph G'
 such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$.

A *path* from vertex v_p to vertex v_q in a graph, G ,
is a sequence of vertices, $v_p, v_{i_1}, v_{i_2}, \dots, v_{i_n}, v_q$,
such that $(v_p, v_{i_1}), (v_{i_1}, v_{i_2}), \dots, (v_{i_n}, v_q)$
are edges in an undirected graph.

If G' is a directed graph,
then the path consists of $\langle v_p, v_{i_1} \rangle, \langle v_{i_1}, v_{i_2} \rangle, \dots, \langle v_{i_n}, v_q \rangle$.

The *length* of a path is the number of edges on it.
simple path simple directed path

A *cycle* is a simple path
in which the first and the last vertices are the same.
directed cycle

**In an undirected graph G ,
two vertices, v_0 and v_1 are *connected*
if there is a path in G from v_0 to v_1 .**

**An undirected graph G is *connected*
if, for every pair of distinct vertices v_i , v_j ,
there is a path from v_i to v_j in G ,
i.e. they are connected.**

**A *connected component*, or simply a *component*, of an undirected graph
is a maximal connected subgraph.**

A *tree* is a graph that is connected and acyclic (it has no cycles).

A directed graph is *strongly* connected

if, for every pair of distinct vertices v_i, v_j in $V(G)$,
there is a directed path from v_i to v_j
and from v_j to v_i .

A *strongly connected component* is a maximal subgraph
which is strongly connected.

A *degree* of a vertex is

the number of edges incident to that vertex.

For a directed graph, we define

the *in-degree* of a vertex v

as the number of edges that have v as the head,

and the *out-degree* of a vertex v

as the number of edges that have v as the tail.

If d_i is the degree of a vertex i in a graph G with n vertices and e edges,
then the number of edges is :

$$e = \frac{1}{2} \sum_{i=0}^{n-1} d_i .$$

[Structure 6.1] Abstract data type Graph

structure Graph is

object : a nonempty set of vertices and a set of undirected edges,
where each edges is a pair of vertices.

functions : for all $graph \in \text{Graph}$, $v, v_1, v_2 \in \text{Vertices}$

Graph Create() ::= **return** an empty graph.

Graph InsertVertex(graph, v) ::= **return** a graph with v inserted.
v has no incident edges.

Graph InsertEdge(graph, v_1, v_2) ::= **return** a graph with a new edge
between v_1 and v_2 .

Graph DeleteVertex(graph, v) ::= **return** a graph in which v and all
edges incident to it are removed.

Graph DeleteEdge(graph, v_1, v_2) ::= **return** a graph in which the edge
(v_1, v_2) is removed. Leave the
incident vertices in the graph.

Boolean IsEmpty(graph) ::= **if** (graph == empty graph) **return** TRUE
else return FALSE.

List Adjacent(graph, v) ::= **return** a list of all vertices that are
adjacent to v.

6.1.3 Graph Representations

Adjacency Matrix

Let $G = (V, E)$ be a graph with n vertices, $n \geq 1$.

Adjacency matrix of G is

a two-dimensional $n \times n$ array, say *adj_mat*, defined as:

$$\begin{aligned} \text{adj_mat}[i][j] &= 1 && \text{if the edge } (V_i, V_j) \text{ is in } E(G) \\ &= 0 && \text{if there is no such edge.} \end{aligned}$$

The same definition can be used for a directed graph

except that the edge $\langle V_i, V_j \rangle$ is directed.

The adjacency matrix for an undirected graph is symmetric,

since the edge (V_i, V_j) is in $E(G)$

iff the edge (V_j, V_i) is also in $E(G)$.

For undirected graphs,

we can save space by storing only the upper or lower triangle
of the matrix.

From the adjacency matrix, we can determine :

- if there is an edge connecting any two vertices.
- the degree of a vertex.

For an undirected graph,

the degree of any vertex i , is its row sum:

$$\sum_{j=0}^{n-1} adj_mat[i][j]$$

For a directed graph,

the row sum is the out-degree,

while the column sum is the in-degree.

Some questions or tasks require us
to examine (potentially) all edges of the graph,
e.g. How many edges are there in G ?
or, Is G connected?

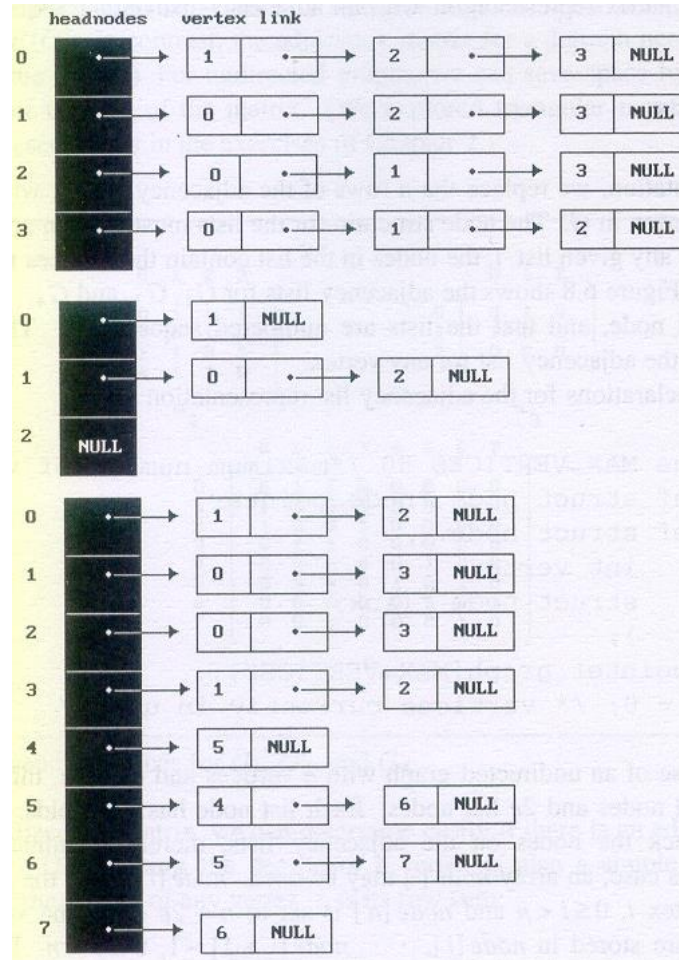
Using adjacency matrices,
all algorithms that answer these questions require at least $O(n^2)$ time
since we must examine $n^2 - n$ entries of the matrix
to determine the edges of the graph.

For a sparse graph, its adjacency matrix becomes sparse.
We might expect that the above questions would be answerable
in significantly less time, say $O(e+n)$ time.

Adjacency Lists

In this representation,
we replace the n rows of the adjacency matrix with n linked lists,
one for each vertex in G .

See Figure 6.8



The C declaration for the adjacency list representation :

```
#define MAX_VERTICES 50 /*maximum number of vertices*/
typedef struct node *node_pointer;
typedef struct node {
    int vertex;
    node_pointer link;
};
node_pointer graph[MAX_VERTICES];
int n=0; /* vertices currently in use */
```

In case of an undirected graph with n vertices and e edges,
this representation requires n head nodes and $2e$ list nodes.

Each list node has two fields.

Questions :

Determine the degree of any vertex, $0 \leq i < n$
equivalently, the number of edges incident on the vertex.

We can determine the total number of edges in G
in $O(n+e)$ time. $0 \leq i < n$

For a directed graph, we can determine the out-degree easily,
but finding the in-degree is more complex.

Maintaining the second list called inverse adjacency list.
See [Figure 6.10]

Weighted Edges

Need to modify our representations.

6.2 ELEMENTARY GRAPH OPERATIONS

Given an undirected graph, $G = (V, E)$, and a vertex, v , in $V(G)$, we wish to visit all vertices in G that are reachable from v , that is, all vertices that are connected to v .

Depth First Search

Breadth First Search

6.2.1 Depth First Search

We begin by visiting the start vertex, v .

Next, we select an unvisited vertex, w , from v 's adjacency list and carry out a depth first search on w .

We preserve our current position in v 's adjacency list by placing it on a stack.

Eventually our search reaches a vertex, u , that has no unvisited vertices on its adjacency list.

At this point, we remove a vertex from the stack and continue processing its adjacency list.

Previously visited vertices are discarded;
unvisited vertices are visited and placed on the stack.

This search terminates when the stack is empty.

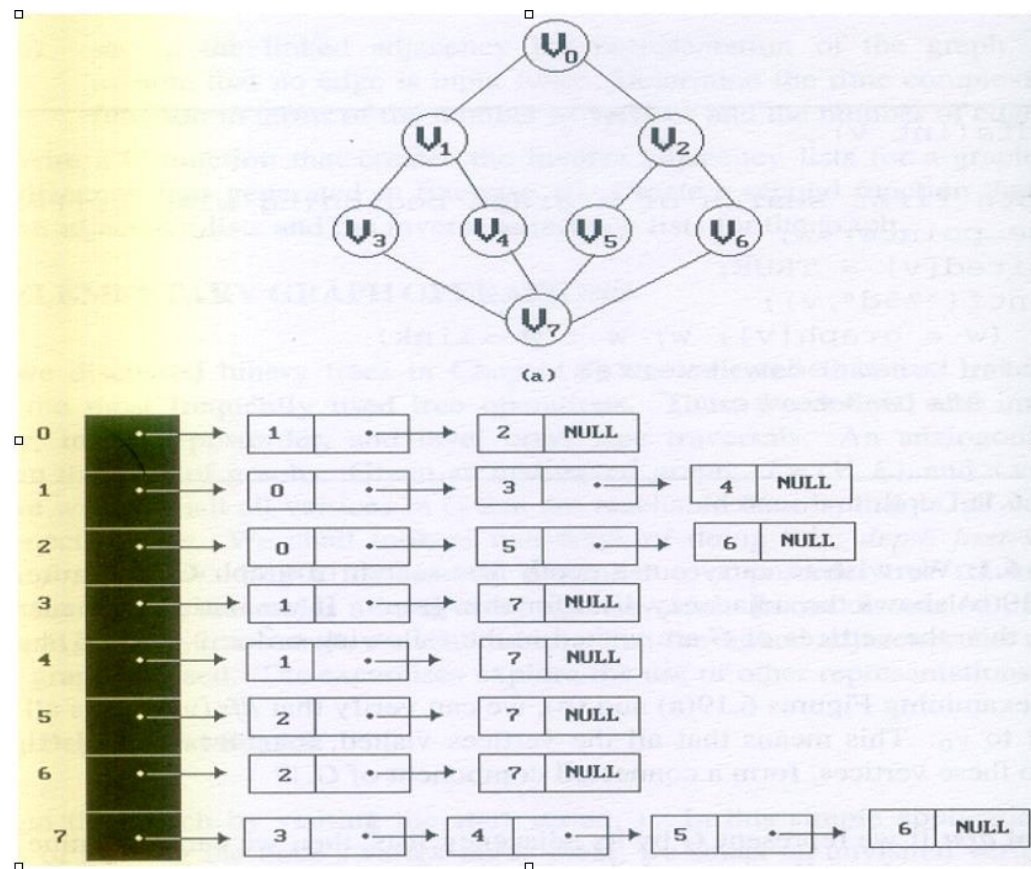
Declaration needed :

```
#define FALSE 0
#define TRUE 1
short int visited[MAX_VERTICES];
```

[Program 6.1] : Depth first search

```
void dfs(int v)
{
    /* depth first search of a graph beginning with vertex v. */
    node_pointer w;
    visited[v] = TRUE;
    printf("%5d", v);
    for (w=graph[v]; w; w=w->link)
        if (!visited[w->vertex])
            dfs(w->vertex);
}
```

Example 6.1 : See Figure 6.19.



Analysis of *dfs* :

If we use adjacency list,
since *dfs* examines each node in adjacency list at most once,
the time to complete the search is $O(e)$.

If we use adjacency matrix,
then determining all vertices adjacent to v requires $O(n)$ time.
Since we visit at most n vertices,
the total time is $O(n^2)$.

6.2.2 Breadth First Search

Starts at vertex and marks it as visited.

It then visits each of the vertices on v's adjacency list.

When we have visited all the vertices on v's adjacency list,
visit all the unvisited vertices that are adjacent to the first vertex
on v's adjacency list.

To implement this scheme,
as we visit each vertex we place the vertex in a queue.

When we have exhausted an adjacency list,
we remove a vertex from the queue
and proceed by examining each of the vertices on its adjacency list.

Unvisited vertices are visited and then placed on the queue;
visited vertices are ignored.

When the queue is empty, the search is finished.

To implement breadth first search,
we use a dynamically linked queue.

Necessary declarations:

```
typedef struct queue *queue_pointer;  
typedef struct queue {  
    int vertex;  
    queue_pointer link;  
};  
void addq(queue_pointer *, queue_pointer *, int);  
int deleteq(queue_pointer *);
```

[program 6.2] : Breadth first search

```
void bfs(int v)
{
    /* breadth first traversal of a graph, starting with node v.
    the global array visited is initialized to 0, the queue
    operations are similar to those described in Chapter 4. */
    node_pointer w;
    queue_pointer front, rear;
    front = rear = NULL; /* initialize queue */
    printf("%5d",v);
    visited[v] = TRUE;
    addq(&front, &rear, v);
    while (front) {
        v = deleteq(&front);
        for (w=graph[v]; w; w=w->link)
            if (!visited[w->vertex]) {
                printf("%5d",w->vertex);
                addq(&front, &rear, w->vertex);
                visited[w->vertex] = TRUE;
            }
    }
}
```


Analysis of *bfs* :

Since each vertex is placed on the queue exactly once,
the while loop is iterated at most n times.

For the adjacency list representation,
this loop has a total cost of $d_0 + d_1 + \dots + d_{n-1} = O(e)$,
where $d_i = \text{degree}(v_i)$.

For the adjacency matrix representation,
the while loop takes $O(n)$ times for each vertex visited.
Therefore, the total time is $O(n^2)$.

6.2.3 Connected Components

Note that in both depth first search and breadth first search,
all vertices visited, together with all edges incident to them,
form a connected component of G .

This property allows us to determine
whether or not an directed graph is connected.

Simply calling either $\text{dfs}(0)$ or $\text{bfs}(0)$
and then determining if there are any unvisited vertices.

This takes $O(n+e)$ time if adjacency lists are used.

A closely related problem is
that of listing the connected components.

[Program 6.3] : Connected components

```
void connected(void)
{
    /* determine the connected components
    of a graph */
    int i;
    for (i=0; i<n; i++)
        if (!visited[i]) {
            dfs(i);
            printf("□n");
        }
}
```

Analysis of *connected* :

If G is represented by its adjacency lists,
then the total time taken by dfs is $O(e)$.

Since the for loop takes $O(n)$ time,
the total time needed to generate
all the connected components is $O(n+e)$.

If G is represented by its adjacency matrix,
the time needed to determine the connected components is $O(n^2)$.

6.2.4 Spanning Trees

When G is connected,
a depth first search or breadth first search starting at any vertex visits
all the vertices in G .

The search implicitly partitions the edges in G into two sets :
 T (for tree edges) is the set of edges used or traversed
 during the search
 and N (for nontree edges) is the set of remaining edges.

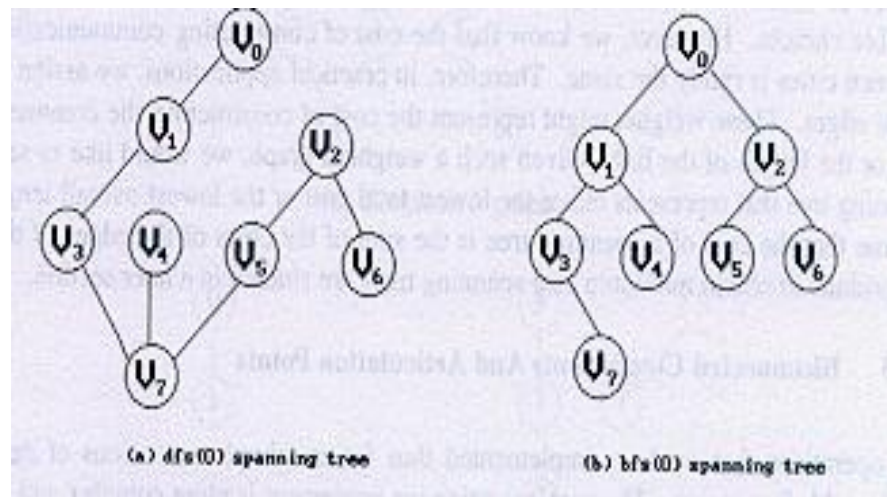
The edges in T form a tree that include all vertices of G .

A spanning tree is any tree that consists solely of edges in G
and that includes all the vertices in G .

See Figure 6.17 (page 284)

We may use either *dfs* or *bfs* to create a spanning tree;
depth first spanning tree
breadth first spanning tree

[Figure 6.18]



If we add a nontree edge (v,w) into any spanning tree T ,
it creates a cycle that consists of the edge (v,w)
and all edges on the path from w to v in T .

Another property of spanning tree:

A spanning tree is a minimal subgraph G' of G
such that $V(G') = V(G)$ and G' is connected.

Any connected graph with n vertices must have at least $n-1$ edges,
and all connected graph with $n-1$ edges are trees.

Therefore, the spanning tree has $n-1$ edges.

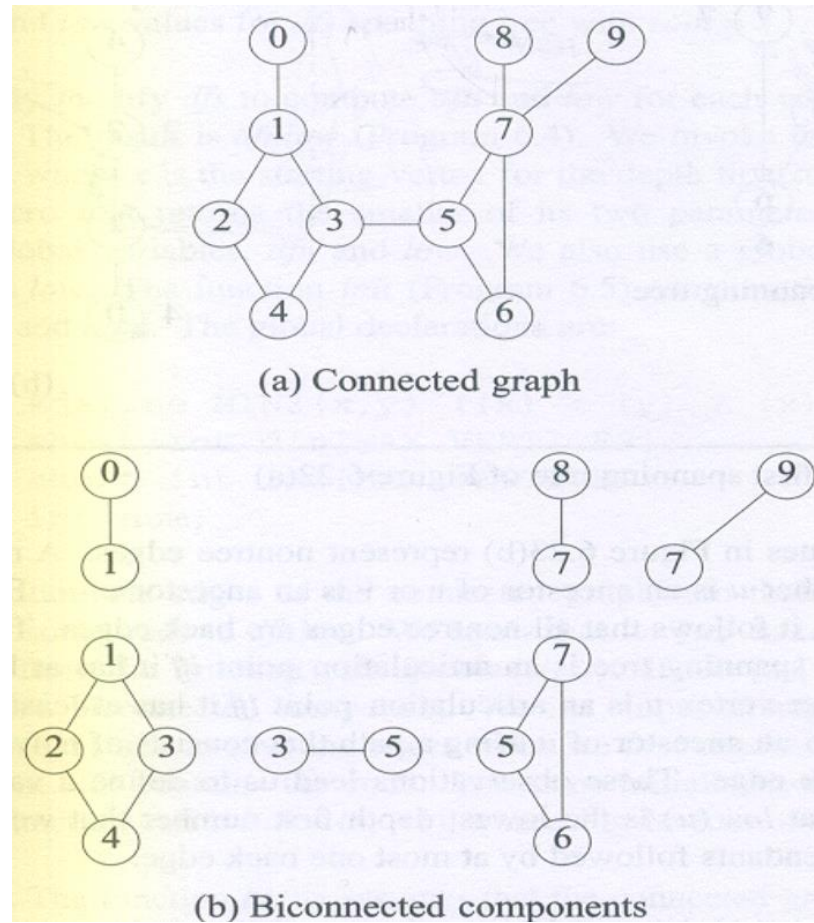
Constructing minimal subgraph finds frequent application
in the design of communication networks.

6.2.5 Biconnected Components and Articulation Points

For an undirected connected graph G ,
an articulation point is a vertex v of G such that
the deletion of v , together with all edges incident on v ,
produces a graph $G - v$
that has at least two connected components.

A biconnected graph is a connected graph
that has no articulation points.
See Figure 6.22 for an example

[Figure 6.22]



In many graph application,
articulation points are undesirable.

A *biconnected* component of a connected
undirected graph is a
maximal *biconnected subgraph* H of G .

It is easy to verify that
two biconnected components of the same graph
have no more than one vertex in common.

This means no edge can be
in two or more biconnected components
of a graph.

Hence, the biconnected components of G partition
the edges of G .

Finding the biconnected components of a connected undirected graph G by using a depth first spanning tree of G :

We number the vertices in the order in which the vertices are visited during the depth first search.

We call this number the *depth first number*, or *dfn*, of the vertex.

If u and v are two vertices, and u is an ancestor of v in the depth first spanning tree, then $dfn(u) < dfn(v)$.

A nontree edge (u, v) is a back edge
iff either u is an ancestor of v or v is an ancestor of u .

From the definition of depth first search,
it follows that all nontree edges are back edges.

This means that
the root of a depth first spanning tree is an articulation point
iff it has at least two children.

In addition,
any other vertex u is an articulation point
iff it has at least one child w such that we cannot reach
an ancestor of u using a path that consists of only w ,
descendants of w , and a single back edge.

These observations lead us to define a value, low , for each vertex of G
such that $low(u)$ is the lowest dfn that we can reach from u
using a path of descendants followed by at most one back edge:

$$low(u) = \min \{dfn(u), \min\{low(w) \mid w \text{ is a child of } u\}, \\ \min\{dfn(w) \mid (u,w) \text{ is a back edge}\}\}$$

Therefore, we can say that u is an articulation point

iff u is either the root of the spanning tree and has two or more children,
or u is not the root and has a child w
such that $low(w) \geq dfn(u)$.

[Figure 6.24]

Vertex	0	1	2	3	4	5	6	7	8	9
dfn	4	3	2	0	1	5	6	7	9	8
low	4	0	0	0	0	5	5	5	9	8

We can easily modify *dfs* to compute *dfn* and *low* for each vertex of a connected undirected graph. See Program 6.4.

Its initial call is *dfnlow*(*x*, -1), where *x* is the starting vertex for the depth first search.

In that program, we use a macro, *MIN2*, and global variables, *dfn*, *low* and *num*.

The function *init* (Program 6.5) contains the code to correctly initialize *dfn*, *low*, and *num*.

Declaration needed:

```
#define MIN2(x,y) ((x) < (y) ? (x) : (y))  
short int dfn[MAX_VERTICES];  
short int low[MAX_VERTICES];  
int num;
```

[Program 6.5] Initialization of dfn and low.

```
void init(void)  
{  
    int i;  
    for (i=0; i<n; i++) {  
        dfn[i] = low[i] = -1;  
    }  
    num = 0;  
}
```

[Program 6.4]

```
void dfnlow(int u, int v)
{
    /* compute dfn and low while performing a dfs search
       beginning at vertex u, v is the parent of u (if any) */
    node_pointer ptr;
    int w;
    dfn[u] = low[u] = num++;
    for (ptr = graph[u]; ptr; ptr = ptr->link) {
        w = ptr->vertex;
        if (dfn[w] < 0) { /* w is an unvisited vertex */
            dfnlow(w, u);
            low[u] = MIN2(low[u], low[w]);
        }
        else if (w != v)
            low[u] = MIN2(low[u], dfn[w]);
    }
}
```


If $\text{low}[w] \geq \text{dfn}[u]$,
then we have identified a new biconnected component.

We can output all edges in a biconnected component
if we use a stack to save the edges when we first encounter them.

See Program 6.6.

Its initial call is *bicon*(*x*, -1),
where *x* is the root of the spanning tree.

The same initialization function (Program 6.5) is used.

[Program 6.6] Biconnected components of a graph

```
void bicon(int u, int v)
{
    node_pointer ptr;
    int w, x, y;
    dfn[u] = low[u] = num++;
    for (ptr=graph[u]; ptr; ptr=ptr->link) {
        w = ptr->vertex;
        if (v != w && dfn[w] < dfn[u])
            add(&top, u, w); /* add edge to stack */
        if (dfn[w] < 0) { /* w is an unvisited vertex */
            bicon(w, u);
            low[u] = MIN2(low[u], low[w]);
            if (low[w] >= dfn[u]) {
                printf("New biconnected component: ");
                do { /* delete edge from stack */
                    delete(&top, &x, &y);
                    printf("<%d, %d>", x, y);
                } while (!(x == u) && (y == w));
                printf("□\n");
            }
        }
        else if (w != v) low[u] = MIN2(low[u], dfn[w]);
    }
}
```

Analysis of *bicon* :

The function *bicon* assumes

that the connected graph has at least two vertices.

The time complexity of *bicon* is $O(n+e)$.

6.3 MINIMUM COST SPANNING TREES

The *cost* of a spanning tree of a weighted undirected graph is the sum of the costs (weights) of the edges in the spanning tree.

A *minimum cost spanning tree* is a spanning tree of least cost.

Three greedy algorithms :
Kruskal's algorithm,
Prim's algorithm,
Sollin's algorithm.

In the greedy method,
we construct an optimal solution in stages.

At each stage, we make a decision
that is the best decision (using some criterion) at this time.

Since we cannot change this decision later,
we make sure that the decision will result in a feasible solution.

Typically, the selection of an item at each stage is based on either a least cost or a highest profit criterion. A feasible solution is one which works within the constraints specified by the problem.

For spanning trees, we use a least cost criterion.

Our solution must satisfy the following constraints:

- (1) we must use only edges within the graph
- (2) we must use exactly $n-1$ edges
- (3) we may not use edges that would produce a cycle

Kruskal's Algorithm

Kruskal's algorithm builds a minimum cost spanning tree T by adding edges to T one at a time.

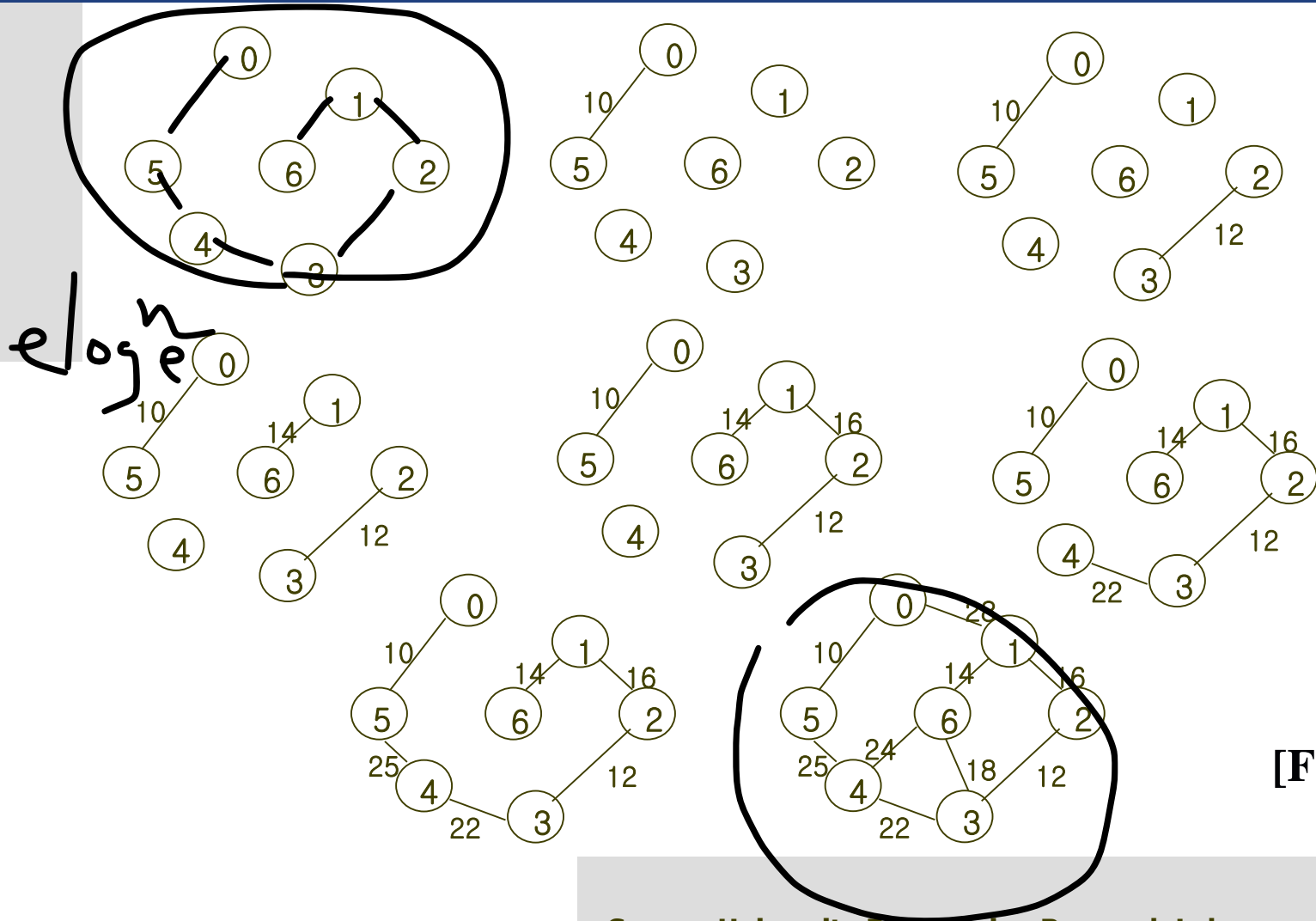
The algorithm selects the edges for inclusion in T in nondecreasing order of their cost.

An edge is added to T

if it does not form a cycle with edges that are already in T .

If G is connected and has vertices $n > 0$,

exactly $n-1$ edges will be selected for inclusion in T .



[Figure 6.25]

Edge	Weight	Result	Figure
----- (0,5)	----- 10	Initial Added to tree	Figure 6.25(b) Figure 6.25(c)
(2,3)	12	Added	Figure 6.25(d)
(1,6)	14	Added	Figure 6.25(e)
(1,2)	16	Added	Figure 6.25(f)
(3,6)	18	Discarded	
(3,4)	22	Added	Figure 6.25(g)
(4,6)	24	Discarded	
(4,5)	25	Added	Figure 6.25(h)
(0,1)	28	Not considered	

[Figure 6.26] Summary of Kruskal's algorithm applied to figure 6.25(a)

[Program 6.7] Kruskal's algorithm.

```
T = {};  
while (T contains less than n-1 edges && E is not empty) {  
    choose a least cost edge (v,w) from E;  
    delete (v,w) from E;  
    if ((v,w) does not create a cycle in T)  
        add (v,w) to T;  
    else  
        discard (v,w);  
}  
if (T contains fewer than n-1 edges)  
    printf("No spanning tree□n");
```

Implementation :

How to determine an edge with minimum cost and delete that edge?
Sorting or Using a min heap.

How to check the new edge, (v,w) , does not form a cycle in T ?
We may use the union-find operations in Section 5.10.

The computing time of Kruskal's algorithm is $O(e \log e)$.

[Theorem 6.1] : Let G be an undirected connected graph,
Kruskal's algorithm generates a minimum cost spanning tree.

Prim's Algorithm

Prim's algorithm, like Kruskal's algorithm,
constructs the minimum cost spanning tree one edge at a time.

However, at each stage, the set of selected edges forms a tree.

Prim's algorithm begins with a tree, T ,
that contains a single vertex.

This may be any of the vertices.

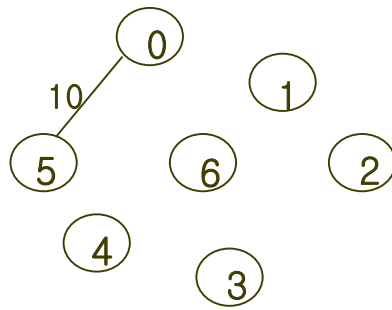
Next, we add a least cost edge (u,v) to T
such that $T \cup \{(u,v)\}$ is also a tree.

We repeat this edge addition step
until T contains $n-1$ edges.

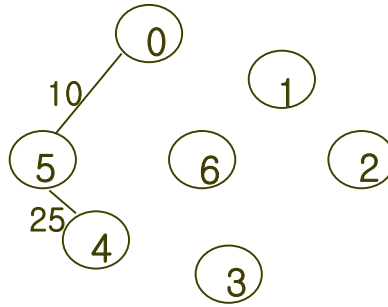
To make sure that the added edge does not form a cycle,
at each step we choose the edge (u,v) such that
exactly one of u or v in T .

[Program 6.8]: Prim's algorithm

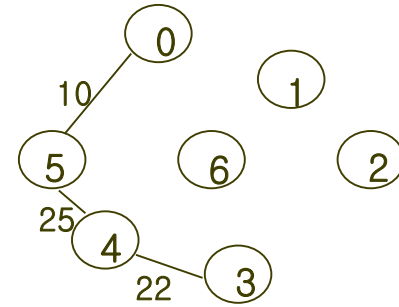
```
T = {};  
TV = {0}; /* start with vertex 0 and no edges */  
while (T contains fewer than n-1 edges) {  
    let (u,v) be a least cost edge such that  
        u ∈ TV and v ∉ TV;  
    if (there is no such edge)  
        break;  
    add v to TV;  
    add (u,v) to T;  
}  
if (T contains fewer than n-1 edges)  
    printf("No spanning tree□n");
```



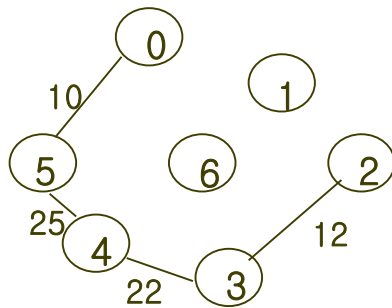
(a)



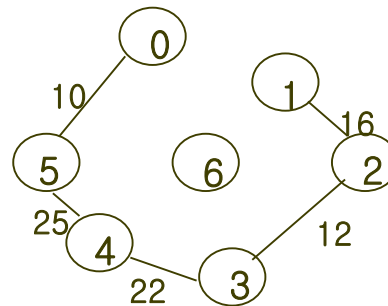
(b)



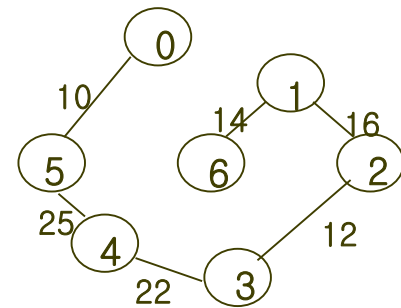
(c)



(d)



(e)



(f)

[Figure 6.27]

Implementation :

For each vertex v not in T ,
we keep a companion vertex, $\text{near}(v)$,
such that $\text{near}(v) \in T$ and
 $\text{cost}(\text{near}(v), v)$ is minimum over all such choices for $\text{near}(v)$.

Computing time is $O(n^2)$, where n is the number of vertices in G .

Sollin's Algorithm

Unlike Kruskal's and Prim's algorithms,

Sollin's algorithm selects several edges for inclusion in T at each stage.

At the start of a stage, the selected edges, together with all n graph vertices, form a spanning forest.

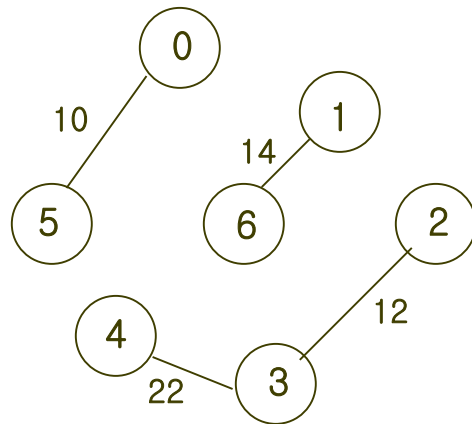
During a stage we select one edge for each tree in the forest.

This edge is a minimum cost edge

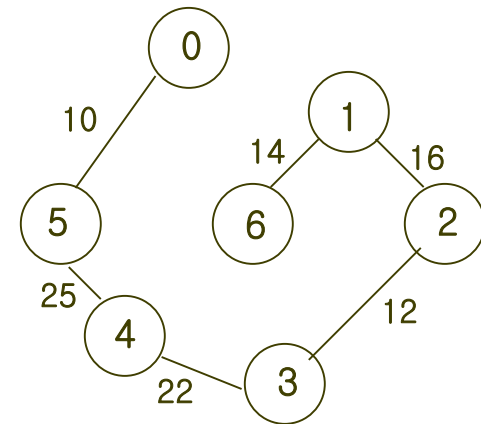
that has exactly one vertex in the tree.

At the start of the first stage the set of selected edges is empty.

The algorithm terminates when there is only one tree at the end of a stage
or no edges remain for selection.



(a)



(b)

[Figure 6.28] Stages in Sollin's algorithm

Shortest Paths and Transitive Closure

Suppose we have a graph that represents the highway system.
In this graph, the vertices represent cities and
edges represent sections of the highway.
Each edge has a weight representing the distance
between the two cities connected by the edge.

Questions (from a motorist who wish to drive from city A to B):

- (1) Is there a path from A to B?
- (2) If there is more than one path from A to B,
which path is the shortest?

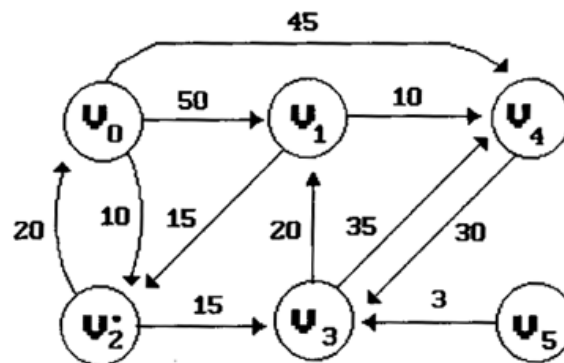
We define the length of a path
as the sum of the weights of the edges on that path.

We assume a directed graph.

Single Source All Destinations: Nonnegative Edge Costs

Given a directed graph $G=(V, E)$,
 a weighting function, $w(e)$, $w(e) > 0$, for the edges of G ,
 and a source vertex, v_0 .

We wish to determine a shortest path from v_0
 to each of the remaining vertices of G .



(a)

	path	length
1)	$v_0 \ v_2$	10
2)	$v_0 \ v_2 \ v_3$	25
3)	$v_0 \ v_2 \ v_3 \ v_1$	45
4)	$v_0 \ v_4$	45

(b)

We may use a greedy algorithm to generate the shortest paths
in nondecreasing order of their lengths.

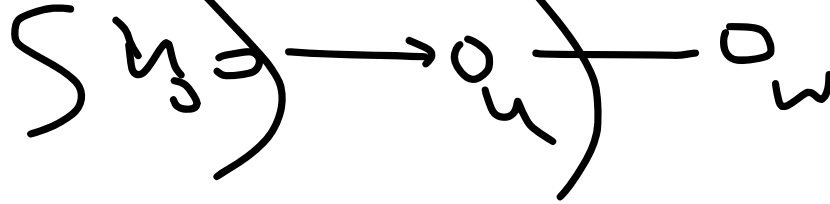
Let S be the set of vertices, including v_0 ,
whose shortest paths have been found.

For w not in S ,

let $\text{distance}[w]$ be the length of the shortest path starting from v_0 ,
going through vertices only in S , and ending in w .

Observations:

- (1) If the next shortest path is to vertex u , then the path from v_0 to u
goes through only those vertices that are in S .
- (2) Vertex u is chosen so that it has the minimum distance,
 $\text{distance}[u]$, among all the vertices not in S .
- (3) Once we have selected u and generated the shortest path from v_0
to u , u becomes a member of S . Adding u to S can change the
distance of shortest paths starting at v_0 , going through vertices only in
 S , and ending at a vertex w , that is not currently in S .



Implementing Dijkstra's algorithm

Assume that the n vertices are numbered from 0 to $n-1$.

Maintain the set S as an array, found,

found[i] = FALSE if vertex i is not in S and

found[i] = TRUE if vertex i is in S .

Graph is represented by its cost adjacency matrix,

with cost[i][j] being the weight of edge $\langle i, j \rangle$.

If the edge $\langle i, j \rangle$ is not in G ,

we set cost[i][j] to some large number.

The choice of this number is arbitrary, but we make two stipulations:

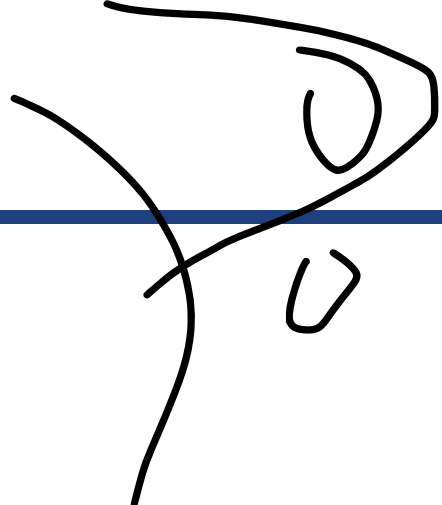
- (1) The number must be larger than any of the value
in the cost matrix.
- (2) The number must be chosen so that distance[u] + cost[u][w] does not produce
an overflow into the sign bit.

Program 6.9 : Declarations for the shortest path algorithm

```
#define MAX_VERTICES 6 /* maximum number of vertices*/
int cost[][MAX_VERTICES] =
    {{ 0, 50, 10, 1000, 45, 1000},
     { 1000, 0, 15, 1000, 10, 1000},
     { 20, 1000, 0, 15, 1000, 1000},
     { 1000, 20, 1000, 0, 35, 1000},
     { 1000, 1000, 30, 1000, 0, 1000},
     { 1000, 1000, 1000, 3, 1000, 0}};
int distance[MAX_VERTICES];
short int found[MAX_VERTICES];
int n = MAX_VERTICES;
```

Program 6.10 : Single source shortest paths

```
void shortestpath(int v, int cost[][MAX_VERTICES],
                 int distance[], int n, short int found[])
{
    /* distance[i] represents the shortest path from vertex v to i,
    found[i] holds a 0 if the shortest path from vertex i has not
    been found and a 1 if it has. cost is the adjacency matrix */
    int i, u, w;
    for (i=0; i<n; i++) {
        found[i] = FALSE;
        distance[i] = cost[v][i];
    }
}
```



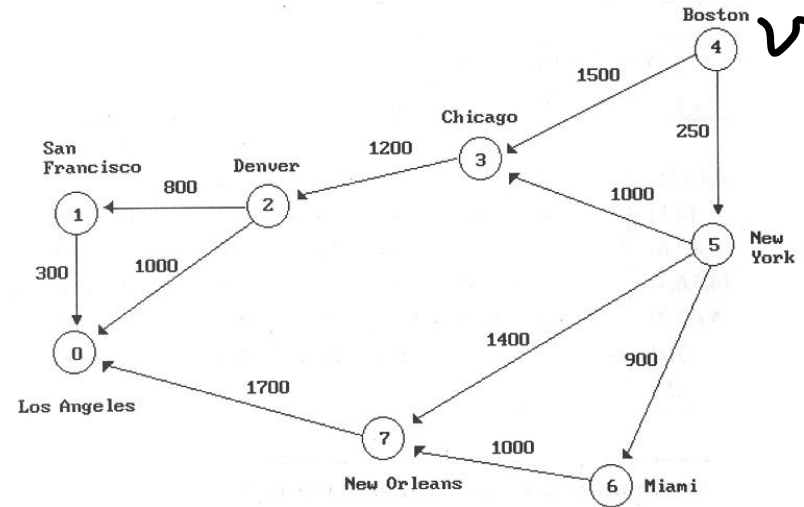
```
found[v] = TRUE;
distance[v] = 0;
for (i=0; i<n-2; i++) {
    u = choose(distance, n, found);
    found[u] = TRUE;
    for (w=0; w<n; w++)
        if (!found[w])
            if (distance[u] + cost[u][w] < distance[w])
                distance[w] = distance[u] + cost[u][w];
}
}
```

Program 6.10 : Single source shortest paths

```
int choose(int distance[], int n, short int found[])
{
    /* find the smallest distance not yet checked */
    int i, min, minpos;
    min = INT_MAX;
    minpos = -1;
    for (i=0; i<n; i++)
        if (distance[i] < min && !found[i]) {
            min = distance[i];
            minpos = i;
        }
    return minpos;
}
```

Analysis of *shortestpath* : $O(n^2)$

Example 6.4 :



(a) Digraph of hypothetical airline routes

	0	1	2	3	4	5	6	7
0	0							
1	300	0						
2	1000	800	0					
3			1200	0				
4				1500	0	250		
5				1000		0	900	1400
6							0	1000
7	1700							0

(b) Cost adjacency matrix

Iteration	S	Vertex selected	Distance (4)							
			LA [0]	SF [1]	DEN [2]	CHI [3]	BOST [4]	NY [5]	MIA [6]	NO [7]
Initial	--	----	$+\infty$	$+\infty$	$+\infty$	1500	0	250	$+\infty$	$+\infty$
1	{4}	5	$+\infty$	$+\infty$	$+\infty$	1250	0	250	1150	1650
2	<u>{4,5}</u>	<u>6</u>	$+\infty$	$+\infty$	$+\infty$	1250	0	250	1150	1650
3	{4,5,6}	3	$+\infty$	$+\infty$	2450	1250	0	250	1150	1650
4	{4,5,6,3}	7	3350	$+\infty$	2450	1250	0	250	1150	1650
5	{4,5,6,3,7}	2	3350	3250	2450	1250	0	250	1150	1650
6	{4,5,6,3,7,2}	1	3350	3250	2450	1250	0	250	1150	1650
	{4,5,6,3,7,2,1}									

Figure 6.31: Action of *shortestpath* on the digraph of Figure 6.30

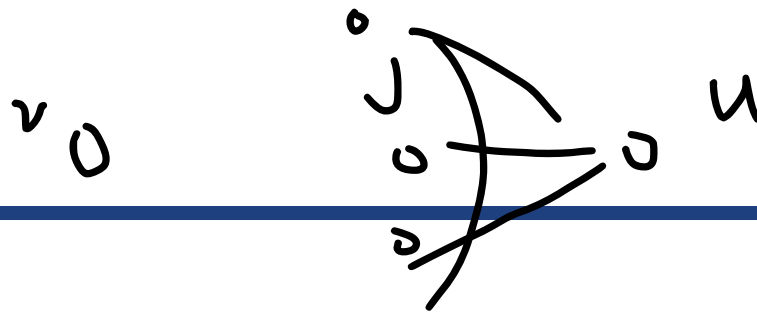
Single Source All Destinations: General Weights

Dijkstra's algorithm does not work for graphs with negative weights (e.g. Figure 6.29)

Bellman-Ford algorithm solves this problem (without cycles of negative length)

$dist^l[u]$: length of a shortest path from the source vertex v to vertex u
under the constraint that the shortest path contains at most l edges

Goal: compute $dist^{n-1}[u]$ for all u



Observations:

1. If the shortest path from v to u with at most k , $k > 1$, edges has no more than $k-1$ edges, then $dist^k[u] = dist^{k-1}[u]$
2. If the shortest path from v to u with at most k , $k > 1$, edges has exactly k edges, then it is comprised of a shortest path from v to some vertex j followed by the edge $\langle j, u \rangle$. The path from v to j has $k-1$ edges, and its length is $dist^{k-1}[j]$. All vertices i such that the edge $\langle i, u \rangle$ is in the graph are candidates for j . Since we are interested in a shortest path, the i that minimizes is the correct value for j

$$dist^k[u] = \min \{ dist^{k-1}[u], \min_i \{ dist^{k-1}[i] + length[i][u] \} \}$$

Bellman-Ford Algorithm

```
void BellmanFord(int n, int v)
{ // single source all destination shortest paths with
  // negative edge lengths
  for (int i=0; i<n; i++)
    dist[i] = length[v][i]; // initialize dist
  for (int k=2; k<=n-1; k++)
    for (each u such that u != v and u has at least one
         incoming edge)
      for (each <i, u> in the graph)
        if (dist[u] > dist[i] + length[i][u])
          dist[u] = dist[i] + length[i][u];
}
```

Time complexity: $O(n^3)$ with adjacency matrix, $O(ne)$ with list

All Pair Shortest Paths

We wish to find the shortest paths between all pairs of vertices, $v_i, v_j, i \neq j$.

We could solve this problem using shortest path
with each of the vertices in $V(G)$ as the source.

The total time required would be $O(n^3)$.

However, we can obtain a conceptually simpler algorithm that works correctly even if some edges in G have negative weights.

(We do require that G has no cycles with a negative length.)

Dynamic programming method.

Although this algorithm still has a computing time of $O(n^3)$,
it has a smaller constant factor.

We represent the graph G by its cost adjacency matrix.

Let $A^k[i][j]$ be the cost of the shortest path from i to j , using only those intermediate vertices with an index $\leq k$.

$A^{n-1}[i][j]$ be the cost of the shortest path from i to j .

$$A^{-1}[i][j] = \text{cost}[i][j].$$

The basic idea in the all pairs algorithm is to begin with the matrix A^{-1} and successively generate the matrices $A^0, A^1, A^2, \dots, A^{n-1}$.

If we have already generated A^{k-1} , then we may generate A^k by realizing that for any pair of vertices i, j one of the two rules below applies.

- (1) The shortest path from i to j going through no vertex with index greater than k does not go through the vertex with index k and so its cost is $A^{k-1}[i][j]$.
- (2) The shortest path go through the vertex k . Such a path consists of a path from i to k followed by one from k to j .

These rules yield the following formulas for $A^k[i][j]$:

$$A^k[i][j] = \min\{A^{k-1}[i][j], A^{k-1}[i][k] + A^{k-1}[k][j]\}, \quad k \geq 0$$

and

$$A^1[i][j] = \text{cost}[i][j].$$

Example 6.5: Look at $A^1[0][2]$.

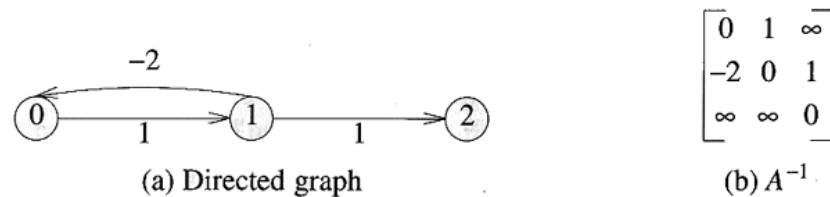


Figure 6.32: Graph with negative cycle

The function *allcosts* computes $A^{n-1}[i][j]$.

The computations are done in place using the array *distance*.

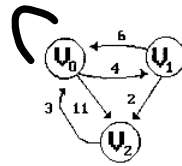
Note that $A^k[i][k] = A^{k-1}[i][k]$ and $A^k[k][j] = A^{k-1}[k][j]$.

Program 6.12 : All pairs, shortest paths function

```
void allcosts(int cost[][MAX_VERTICES],
              int distance[][MAX_VERTICES], int n)
{ /* determine the distances from each vertex to every other vertex, cost is
   the adjacency matrix, distance is the matrix of the distances. */
  int i, j, k;
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      distance[i][j] = cost[i][j];
  — for (k=0; k<n; k++)
    for (i=0; i<n; i++)
      — for (j=0; j<n; j++)
        — if (distance[i][k] + distance[k][j] < distance[i][j])
          distance[i][j] = distance[i][k] + distance[k][j];
}
```


Analysis of *allcosts* : The total time for *allcosts* is $O(n^3)$.

Example 6.6:



	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

(a) Digraph G

(b) Cost adjacency matrix for G

Figure 6.33: Directed graph and its cost matrix

A^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

A^0	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

A^1	0	1	2
0	0	4	6
1	6	0	2
2	3	7	0

A^2	0	1	2
0	0	4	6
1	5	0	2
2	3	7	0