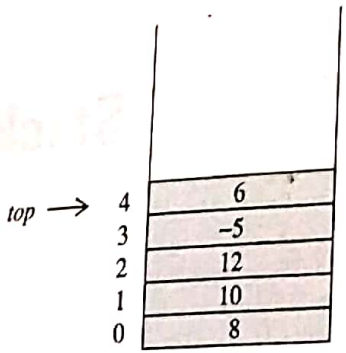
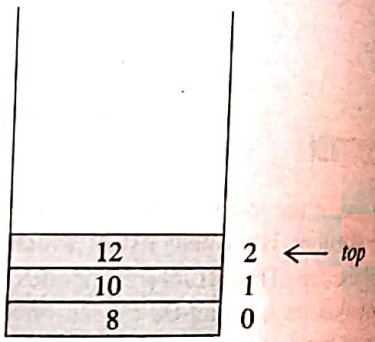
Example 7.1:

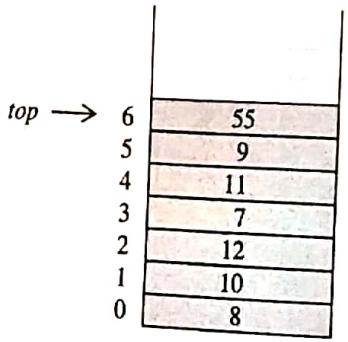
The following figure illustrates a stack, which can accommodate maximum of 10 elements.



(*a*) Stack after pushing elements 8, 10, 12, −5, 6 in turn

(*b*) Stack after poping top two elements, *i.e.*, 6, and −5 in turn





(*c*) Stack after pushing elements 7, 11, 9, 55 in turn

Figure 7.2: Illustration of a stack whose elements are integer values

7.2 OPERATIONS ON STACKS

The following operations are performed on stacks.

Table 6.1: Various operations performed on ~~queues~~ stacks

Operation	Description
create	To create an initial stack. This task will be accomplished using a constructor.
destroy	To remove (delete) stack from memory. This task will be accomplished using a destructor.
push	Insert (push) an element at the top of stack.
pop	Access and remove the top element of the stack.
peek	Access the top element of the stack without removing it from the stack.
isFull	Check whether the stack is full.
isEmpty	Check whether the stack is empty.

All of these operation runs in O(1) time.

Note that if the stack *s* is empty then it is not possible to pop the stack *s*. Similarly, as there is no element in the stack, the *Peek(s)* operations is also valid. Therefore, we must ensure that the stack is not empty before attempting to perform these operations.

Likewise, if the stack *s* is full then it is not possible to push a new element on the stack *s*. Therefore, we must ensure that the stack is not full before attempting to perform push operation.

7.3 REPRESENTATION OF A STACK IN MEMORY

A stack can be represented in memory using a linear array or a linear linked list. Let us first discuss array representation of stacks.

7.3.1 Representing a Stack using an Array

To implement a stack we need a variable, called *top*, that holds the index of the top element of the stack and an array, named *elements*, to hold the elements of the stack.

Suppose that the elements of the stack can be any primitive type or pointer type, the following is the required ADT to represent a generic stack:

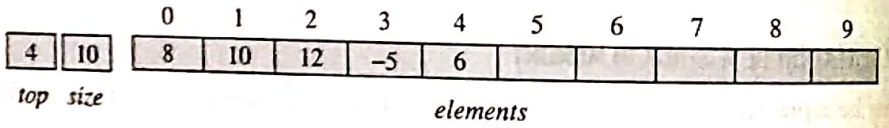
```
template <class T>
class ArrayStack
{
   protected:                  // data members
      int top;
      int size;
      T *elements;
   Public:                     // member functions
      // zero argument constructor,
      // it constructs a stack with default size = 10
```

```
        ArrayStack();
        // parametrized constructor that constructs,
        // a stack with size = n
        ArrayStack(int n);
        // destructor that removes the stack from memory
        ~ArrayStack();
        // Check whether the stack is empty
        int isEmpty();
        // Check whether the stack is full
        int isFull();
        // Insert (push) an element at the top of stack
        void push(T value);
        // Access and remove the top element of the stack
        T pop();
        // Access the top element of the stack
        // without removing it from the stack
        T peek();
};  // -------- End of ArrayStack class --------
```

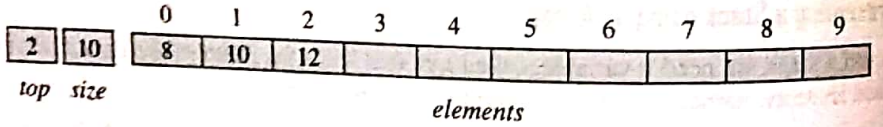
We have defined our own data type named *ArrayStack*, which is class and whose first data member *top* will be used as an index to the top element, second data member *size* will be used to hold the size of the stack, and third data member *elements*, a dynamic array whose elements can be any primitive type or pointer type.

The implementation of the member functions is given in the subsequent sections.

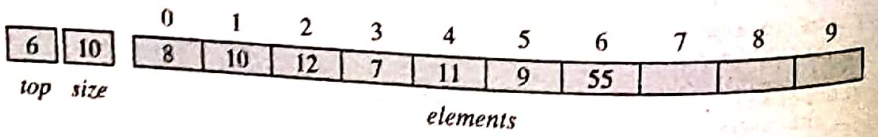
With these declarations, assuming a stack whose elements are of type *int* and size is 10, a stack of Figure 7.2 will be represented in computer memory as



(*a*) Representation of stack of Figure 7.2(*a*) in memory



(*b*) Representation of stack of Figure 7.2(*b*) in memory



(*c*) Representation of stack of Figure 7.2(*c*) in memory

Figure 7.3: Array representation of stack of Figure 7.2

7.3.1.1 Creating an Empty Stack

Before we can use a stack, it is to be initialized. As the index of array elements can take any value in the range 0 to *size*−1, the purpose of initializing the stack is served by assigning value −1 (sentinel value) to the *top* variable. This simple task is accomplished using following constructors, where the zero argument constructors create a stack with default size 10 while the parameterised constructor creates a stack of use-defined size (*n*).

Listing 7.1:

```
// zero argument constructor,
// it constructs a stack with default size = 10
template <class T>
ArrayStack<T>::ArrayStack()
{
  elements = new T[10];
  size = 10;
  top = -1;
}
// parametrized constructor that constructs,
// a stack with size = n
template <class T>
ArrayStack<T>::ArrayStack(int n)
{
  elements = new T[n];
  size = n;
  top = -1;
}
```

7.3.1.2 Testing Stack for Underflow

Before we remove an item from a stack, it is necessary to test whether stack still have some elements, *i.e.*, to test that whether the stack is empty or not. If it is not empty then the pop operation can be performed to remove the top element. This test is performed by comparing the value of *top* with sentinel value −1, as shown in the following function.

Listing 7.2:

```
// Check whether the stack is empty
template <class T>
int ArrayStack<T>::isEmpty()
{
  if ( top == -1 )
    return 1;
  else
    return 0;
}
```

The above function returns value 1, indicating that stack is empty, if the test condition is satisfied; otherwise it returns value 0, indicating that stack is not empty.

The *isEmpty()* function can also be written using conditional operator (?:) as

Listing 7.3:

```
// Check whether the stack is empty
template <class T>
int ArrayStack<T>::isEmpty()
{
    return ( top == -1 ? 1 : 0 );
}
```

7.3.1.3 Testing Stack for Overflow

Before we insert new item onto the stack, it is necessary to test whether stack still have some space to accommodate the incoming element i.e. to test that whether the stack is full or not. If it is not full then the *push* operation can be performed to insert the element at top of the stack. This test is performed by comparing the value of the *top* with value *size*–1, the largest index value that the *top* can take, as shown in the following function.

Listing 7.4:

```
// Check whether the stack is empty
template <class T>
int ArrayStack<T>::isFull()
{
   if ( top == size-1 )
      return 1;
   else
      return 0;
}
```

The above function returns value 1, indicating that stack is full, if the test condition is satisfied otherwise it returns value 0, indicating that stack is not full.

The *isFull()* function can also be written using conditional operator (?:) as

Listing 7.5:

```
// Check whether the stack is full
template <class T>
int ArrayStack<T>::isFull()
{
    return ( top == size-1 ? 1 : 0 );
}
```

7.3.1.4 Push Operation

Before the push operation, if the stack is empty, then the value of the *top* will be –1 (sentinel value) and if the stack is not empty then the value of the *top* will be the index of the element

currently on the top. Therefore, before we place value onto the stack, the value of the *top* is incremented so that it points to the new top of stack, where incoming element is placed. This task is accomplished as shown in the following function.

Listing 7.6:

```
// Insert (push) an element at the top of stack
template <class T>
void ArrayStack<T>::push(T value)
{
   top++;
   elements[top] = value;
}
```

The above function can also be written as

Listing 7.7:

```
// Insert (push) an element at the top of stack
template <class T>
void ArrayStack<T>::push(T value)
{
   elements[++top] = value;
}
```

7.3.1.5 Pop Operation

The element on the top of stack is assigned to a local variable, which later on will be returned via the *return* statement. After assigning the top element to a local variable, the variable *top* is decremented so that it points to the new top.

This task is accomplished as shown in the following function.

Listing 7.8:

```
// Access and remove the top element of the stack
template <class T>
T ArrayStack<T>::pop()
{
   T temp;
   temp = elements[top];
   top--;
   return temp;
}
```

The above function can also be written as

Listing 7.9:

```
// Access and remove the top element of the stack
template <class T>
```

```
T ArrayStack<T>::pop()
{
        return elements[top--];
}
```

In the above two versions of the *pop()* function, the top element is returned via the ret... statement.

7.3.1.6 Accessing Top Element

There may be instances where we want to access the top element of the stack without removing it from the stack, *i.e.*, without popping it. This task is accomplished as shown in the following function:

Listing 7.10:

```
// Access the top element without removing it from the stack
template <class T>
T ArrayStack<T>::peek()
{
   return elements[top];
}
```

7.3.1.7 Removing Stack from Memory

As soon as the *ArrayStack* goes out of scope, it needs to be removed from computer memory. This is done using following destructor:

Listing 7.11:

```
// destructor that removes the stack from memory
template <class T>
T ArrayStack<T>::~ArrayStack()
{
   delete elements;
   top = -1;
   size = 0;
}
```

Following is the complete listing of the source code of the menu driven program to demonstrate the implementation of *ArrayStack*.

Listing 7.12:

```
//    Program to demonstrate the various operations on array based stack
#include<iostream.h>
#include<iomanip.h>
template <class T>
class ArrayStack
{
   protected:
                  // data members
```



```
   int top;
   int size;
   T *elements;
                  // member functions
public:
   // zero argument constructor,
   // it constructs a stack with default size = 10
   ArrayStack();
   // parametrized constructor that constructs,
   // a stack with size = n
   ArrayStack(int n);
   // destructor that removes the stack from memory
   ~ArrayStack();
   // Check whether the stack is empty
   int isEmpty();
   // Check whether the stack is full
   int isFull();
   // Insert (push) an element at the top of stack
   void push(T value);
   // Access and remove the top element of the stack
   T pop();
   // Access the top element of the stack
   // without removing it from the stack
   T peek();
}; // -------- End of ArrayStack class --------

// zero argument constructor,
// it constructs a stack with default size = 10
template <class T>
ArrayStack<T>::ArrayStack() {
  elements = new T[10];
  size = 10;
  top = -1;
}
// parametrized constructor that constructs,
// a stack with size = n
template <class T>
ArrayStack<T>::ArrayStack(int n) {
  elements = new T[n];
  size = n;
  top = -1;
}
// destructor that removes the stack from memory
template <class T>
ArrayStack<T>::~ArrayStack() {
  delete elements;
  top = -1;
  size = 0;
}
// Check whether the stack is empty
template <class T>
int ArrayStack<T>::isEmpty() {
  return ( top == -1 ? 1 : 0 );
}
// Check whether the stack is full
template <class T>
int ArrayStack<T>::isFull() {
  return ( top == size-1 ? 1 : 0 );
```

```
}
// Insert (push) an element at the top of stack
template <class T>
void ArrayStack<T>::push(T value) {
   top++;
   elements[top] = value;
}
// Access and remove the top element of the stack
template <class T>
T ArrayStack<T>::pop() {
   T temp;
   temp = elements[top];
   top--;
   return temp;
}
// Access the top element of the stack
// without removing it from the stack
template <class T>
T ArrayStack<T>::peek() {
   return elements[top];
}
// -------- main function ---------
void main()
{
   int choice, element, m;
   cout << "\nSize of stack you want to use : ";
   cin >> m;
   ArrayStack<int> intStack(m);
   do
   {
      cout << "\n     Options available \n";
      cout << "    ++++++++++++++++++ \n\n";
      cout << "      1. Push  \n";
      cout << "      2. Pop \n";
      cout << "      3. Peek \n";
      cout << "      4. Exit\n\n";
      cout << "Enter your choice ( 1-4 ) : ";
      cin >> choice;
      switch ( choice )
      {
         case 1 : if ( intStack.isFull() ) {
                     cout << "Stack full, press any key to continue";
                  } else {
                     cout  << "Enter value : ";
                     cin >> element;
                     intStack.push(element);
                  }
                  break;
         case 2 : if ( intStack.isEmpty() ) {
                     cout << "Stack empty, press any key to continue";
                  }
                  else
                     cout << "Value poped is " << intStack.pop();
                  break;
         case 3 : if ( intStack.isEmpty() ) {
                     cout << "Stack empty, press any key to continue";
                  }
```



```
                  else
                     cout << "Value at top is " << intStack.peek();
      }
   }
   while ( choice != 4 );
} // ---- end of main function ----
```

7.3.2 Representing a Stack using a Linked List

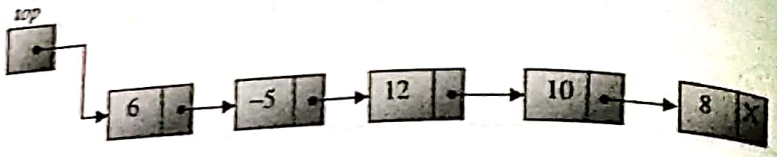
A stack represented using a linked list is also known as *linked stack*. The array-based representation of stacks suffers from following limitations

- Size of the stack must be known in advance.
- We may come across situations when an attempt to push an element causes overflow. However, stack, as an abstract data structure can not be full. Hence, abstractly, it is always possible to push an element onto stack. Therefore, representing stack as an array prohibits the growth of stack beyond the finite number of elements.

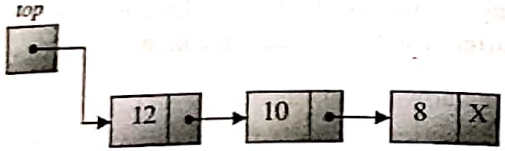
The linked list representation allows a stack to grow to a limit of the computer's memory.

```
template <class T>
class node
{
   public:
      T info;
      node *next;
};
template <class T>
class LinkStack
{
   protected:              // data member
      node<T> *top;
   public:                 // member functions
      // zero argument constructor,
      // it constructs a empty stack
      LinkStack();
      // destructor
      ~LinkStack();
      // Check whether the stack is empty
      int isEmpty();
      // Insert (push) an element at the top of stack
      void push(T value);
      // Access and remove the top element of the stack
      T pop();
      // Access the top element of the stack
      // without removing it from the stack
      T peek();
}; // ************ End of Stack class ****************
```

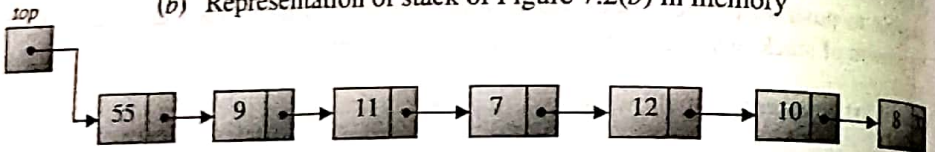
We have defined our own data type named *LinkStack*, which has a pointer to a self-referent class and whose first data member *info* hold the element of the stack and the second d member *next* holds the address of the element under it in the stack.



(*a*) Representation of stack of Figure 7.2(*a*) in memory



(*b*) Representation of stack of Figure 7.2(*b*) in memory



(*c*) Representation of stack of Figure 7.2(*c*) in memory

Figure 7.4: Linked representation of stack of Figure 7.2

With these declarations, we will write functions for various operations to be performed on a stack represented using linked list.

7.3.2.1 Creating an Empty Stack

Before we can use a stack, it is to be initialized. To initialize a stack, we will create an empty linked list. The empty linked list is created by setting pointer variable *top* to value NULL.

This simple task can be accomplished by the following function.

Listing 7.13:

```
// zero argument constructor,
// it constructs a empty stack
template <class T>
LinkStack<T>::LinkStack()
{
   top = NULL;
}
```

7.3.2.2 Testing Stack for Underflow

The stack is tested for underflow condition by checking whether the linked list is empty. The empty status of the linked lists will be indicated by the NULL value of pointer variable *top*. This is shown in the following function.

Listing 7.14:

```
// Check whether the stack is empty
template <class T>
int LinkStack<T>::isEmpty()
{
  return ( top == NULL ? 1 : 0 );
}
```

The above function returns value 1, indicating that stack is empty, if the test condition is satisfied; otherwise it returns value 0, indicating that stack is not empty.

7.3.2.3 Testing Stack for Overflow

Since a stack represented using a linked list can grow to a limit of a computer's memory, there overflow condition never occurs. Hence, this operation is not implemented for linked stacks.

7.3.2.4 Push Operation

To push a new element onto the stack, the element is inserted in the beginning of the linked list. This task is accomplished as shown in the next function.

Listing 7.16:

```
// Insert (push) an element at the top of stack
template <class T>
void LinkStack<T>::push(T value)
{
  node<T> *ptr;
  ptr = new node<T>;
  ptr->info = value;
  ptr->next = top;
  top = ptr;
}
```

7.3.2.5 Pop Operation

To pop an element from the stack, the element is removed from the beginning of the linked list. This task is accomplished as shown in the following function.

Listing 7.17:

```
// Access and remove the top element of the stack
template <class T>
T LinkStack<T>::pop()
{
  T temp;
  node<T> *ptr;
  temp = head->info;
  ptr = top;
  top = head->next;
```

```
template <class T>
class node

{
  public:
    T info;
    node *next;
};
template <class T>
class LinkStack

{
  protected:              // data member
    node<T> *top;
  public:                 // member functions
    // zero argument constructor,
    // it constructs a empty stack
    LinkStack();
    // destructor
    ~LinkStack();
    // Check whether the stack is empty
    int isEmpty();
    // Insert (push) an element at the top of stack
    void push(T value);
    // Access and remove the top element of the stack
    T pop();
    // Access the top element of the stack
    // without removing it from the stack
    T peek();
}; // ************ End of Stack class ****************
// zero argument constructor,
// it constructs a empty stack
template <class T>
LinkStack<T>::LinkStack()
{
  top = NULL;
}
// Check whether the stack is empty
template <class T>
int LinkStack<T>::isEmpty()
{
  return ( top == NULL ? 1 : 0 );
}
// Insert (push) an element at the top of stack
template <class T>
void LinkStack<T>::push(T value)
{
  node<T> *ptr;
  ptr = new node<T>;
  ptr->info = value;
  ptr->next = top;
  top = ptr;
}
// Access and remove the top element of the stack
template <class T>
T LinkStack<T>::pop()
{
  T temp;
  node<T> *ptr;
  temp = head->info;
```

...ement in the beginning of the linked list is assigned to a local variable *temp*, which ...ill be returned via the return statement. And the first node from the linked list is ...

7.6 Accessing Top Element

...top element is accessed as shown in the following function.

Listing 7.18:

```
// Access the top element of the stack
// without removing it from the stack
template <class T>
T LinkStack<T>::peek()
{
  return ( top->info );
}
```

7.3.2.7 Dispose a Stack

Because stack is implemented using linked lists, therefore it is programmer's job to write ... code to release the memory occupied by the stack. The following listing shows the ... steps to be performed.

Listing 7.19:

```
// empty stack from memory
template <class T>
LinkStack<T>::~LinkStack()
{
  node<T> *ptr;
  while ( head != 0 ) {
    ptr = head;
    head = head->next;
    delete ptr;
  }
}
```

This dispose operations is $O(n)$ time.

Following is the complete listing of the source code of the menu driven program ... demonstrate the implementation of *LinkStack*.

Listing 7.20:

```
//    Program to demonstrate the various operations on linked stack
#include<iostream.h>
#include<iomanip.h>
```

7.5 APPLICATIONS OF STACKS

As mentioned earlier, stack is one of the most commonly used data structures. Some of its applications are

1. Stacks are used to pass parameters between functions. On a call to a function, the parameters and local variables are stored on a stack.
2. High-level programming languages, such as Pascal, C, etc., that provides support for recursion use stack for bookkeeping. Remember, in each recursive call, there is need to save the current values of parameters, local variables and the return address (the address where the control has to return from the call).

In addition, stacks are used in solving enormous number of problems. Some of the problems are discussed in this section. In the later chapters, we will also see the use of stack.

7.5.1 Parenthesis Checker

Parenthesis Checker is a program that checks whether a mathematical expression is properly parenthesised. We will consider three sets of grouping symbols: *the standard parentheses* "()", *braces* "{}", and *brackets* "[]". For an input expression, it verifies that for each left parentheses, brace, or bracket, there is a corresponding closing symbol and the symbols are appropriately nested.

Some examples of valid inputs are

Valid Input
( )
(( ) [ ])
([[ ] [ ]])
[{ { ( ) ( ) } [ ] [ ] [ ] ( ) { }]

Some examples of invalid inputs are

Invalid Input
([ ] { }
{) [ ])
({ [[ ] })
[{ { ( ) ( ) [ ] } [ ] ( ) { }]

Inside parentheses (braces or brackets), there can be any valid arithmetic expression. The actual arithmetic expression is of no concern in this problem.

The formal algorithm for parentheses checker is presented below.

Algorithm 7.1:

PranethesisChecker(exp)

Here *exp* is character string representing an arithmetic expression comprising all types of bracketing symbols.

```
Begin
  read: exp
  Create empty stack
  for each character c in exp
      if ( current character is a left symbol) then
          Push the character onto stack
      else if ( current character is a right symbol) then
          if ( stack is empty ) then
              Print: "Error: No matching open symbol"
              Exit
          else
              Pop a symbol s from the stack
              if ( s doesn't correspond to c ) then
                  Print: "Error: Incorrect nesting of symbols"
                  Exit
              endif
          endif
      endif
  endfor
  if ( stack is not empty ) then
      Print: "Error: Missing closing symbol(s)"
  else
      Print: "Input expression is OK"
  endif
End.
```


The following is the implementation of parentheses checker.

Listing 7.21:

```
void pranethesisChecker( char *exp )
{
  char ch;
  LinkStack<char> charStack;
  while ( *exp )
  {
     if ( *exp == '('|| *exp == '{' || *exp == '[' )
        charStack.push(*exp);
     else if ( *exp == ')' || *exp == '}' || *exp == ']' ) {
        if ( charStack.isEmpty() ) {
           cout << "\nError: No matching open symbol" ;
           return;
        } else {
           ch = charStack.pop();
           if ( ch != getMatchingSymbol( *exp ) ) {
              cout << "\nError: Incorrect nesting of symbols";
              return;
           }
        }
     }
     exp++;
  }
  if ( !charStack.isEmpty() ) {
     cout << "\nError: Missing closing symbol(s)";
     return;
  } else {
     cout << "\nInput Expression is OK";
  }
}
```

The *pranethesisChecker()* function uses *getMatchingSymbol()* function whose listing is given below.

Listing 7.22:

```
char getMatchingSymbol( char ch )
{
   char matchingChar;
   switch ( ch )
   {
      case '(' : matchingChar = ')';
                 break;
      case ')' : matchingChar = '(';
                 break;
      case '{' : matchingChar = '}';
                 break;
      case '}' : matchingChar = '{';
                 break;
      case '[' : matchingChar = ']';
                 break;
      case ']' : matchingChar = '[';
   }
   return matchingChar;
}
```

Listing 7.23:

```
// Program to demonstrate the use of stack in checking whether the
// the arithmetic expression is properly parenthesised

#include <iostream.h>
#include <iomanip.h>

// header file containing code for generic stack implemented
// using linear linked list
#include "stlnk.h"

void pranethesisChecker(char *);
char getMatchingSymbol(char);
void main()
{
   char *charExpression = "";
   cout << "\nEnter any expression\n\n" ;
   cin.getline(charExpression,80);
   pranethesisChecker( charExpression );
} /*---- end of main function ----*/
void pranethesisChecker( char *exp )
{
   char ch;
   LinkStack<char> charStack;
   while ( *exp )
   {
      if ( *exp == '('|| *exp == '{' || *exp == '[' )
         charStack.push(*exp);
```



```
      else if ( *exp == ')' || *exp == '}' || *exp == ']' ) {
         if ( charStack.isEmpty() ) {
            cout << "\nError: No matching open symbol" ;
            return;
         } else {
            ch = charStack.pop();
            if ( ch != getMatchingSymbol( *exp ) ) {
               cout << "\nError: Incorrect nesting of symbols";
               return;
            }
         }
      }
      exp++;
   }
   if ( !charStack.isEmpty() ) {
      cout << "\nError: Missing closing symbol(s)";
      return;
   } else {
      cout << "\nInput Expression is OK";
   }
}
char getMatchingSymbol( char ch )
{
   char matchingChar;
   switch ( ch )
   {
      case '(' : matchingChar = ')'; break;
      case ')' : matchingChar = '('; break;
      case '{' : matchingChar = '}'; break;
      case '}' : matchingChar = '{'; break;
      case '[' : matchingChar = ']'; break;
      case ']' : matchingChar = '[';
   }
   return matchingChar;
}
```

7.5.2 Mathematical Notation Translation

Let us first consider various types of notations for writing mathematical expressions. In our discussion, we will consider the following set of operators.

Table 7.1: List of Operators

Symbol Used	Operation Performed	Precedence
* (asterisk)	Multiplication	Highest
/ (slash)	Division	Highest
% (percentage)	Modulus (Remainder)	Highest
+ (plus)	Addition	Lowest
- (hyphen)	Subtraction	Lowest

For simplicity, we will assume that a given expression contains no unary operation. We also assume that the operations on the same level of precedence are performed from left to right.

7.5.2.1 Infix Notation

In this notation, the operator symbol is placed between its two operands. For example

- to add A to B we can write as A + B or B + A
- to subtract D from C we write as C – D, but we can't write D – C as this operation is not commutative

In this notation, we must distinguish between

(A + B) / C and A + (B / C)

7.5.2.2 Polish (Prefix) Notation

In this notation, named after the Polish mathematician ***Jan Lukasiewicz***, the operator symbol is placed before its two operands. For example,

- to add A to B we can write as +AB or +BA
- to subtract D from C we have to write as –CD not as –DC

In order to translate an arithmetic expression in infix notation to polish notation, we do step by step using brackets ([]) to indicate the partial translation.

Consider the following expression in infix notation:

(A – B / C) * (A * K – L)

partial translations may look like:

(A – [/BC]) * ([*AK] – L)
[–A/BC] * [–*AKL]
*–A/BC–*AKL

The fundamental property of polish notation is that the order in which the operations are to perform is completely determined by the positions of the operators and operands in the expression. Accordingly one never needs parentheses when writing expressions in polish notation.

This notation is also called *prefix* notation.

7.5.2.3 Reverse Polish (Postfix) Notation

In this notation the operator symbol is placed after its two operands. For example,

- to add A to B we can write as AB+ or BA+
- to subtract D from C we have to write as CD– not as DC–

In order to translate an arithmetic expression in infix notation to reverse polish notation, we do step by step using brackets ([]) to indicate the partial translation.

Consider the following expression in infix notation:

(A – B / C) * (A * K – L)

the partial translations may look like:

(A – [BC/]) * ([AK/] – L)
[ABC/–] * [AK/L–]
ABC/–AK/L–*

Like polish notation, here too one never needs the use of parentheses when writing expressions in reverse polish notation.

This notation is frequently called *postfix* (or *suffix*) notation.

7.5.2.4 Evaluating Mathematical Expressions

Programs that perform some sort of translation often use stacks. One example of such program is one that translates an arithmetic expression from *infix* notation to *postfix* notation. Although human beings are quite used to work with mathematical expressions in infix notation, which is rather complex. Using this notation, a nontrivial set of rules must be applied to the expression in order to determine the final value. These rules, which are taken for granted, involve concepts such as operator precedence and associativity.

Using infix notation, one cannot tell the order in which the operators should be applied by looking at the expression. Therefore, whenever an expression contains more than one operator, precedence rules are applied to decide which operator and operands are evaluated first. The standard rules for precedence dictate that parentheses are evaluated first, followed by unary minus, multiplication and division, and lastly, addition and subtraction. All these operators are left associative, which means that if there is more than operator of the same precedence, the leftmost operator is applied first.

In comparison, the expression in the postfix notation is much easy to evaluate. As the operands appear before the operator, there is no need for operator precedence or for parentheses to group/order operations. In order to evaluate a postfix expression, it is scanned from left to right. As operands are encountered, they are pushed on a stack. When an operator is encountered, pop top one or two operands depending on the operator, perform the operation and place the result back on the stack. At the end of the expression, the stack contains only one element (if expression is correct) and that is the resulting value of the expression.

The problems addressed in this section are to develop procedures to translate an expression from infix notation to postfix notation and then to evaluate the resultant expression.

For simplicity, we will develop following two separate procedures:

- To translate the expression from infix notation to postfix notation.
- To evaluates the expression in postfix notation.

In each of these procedures, stack is the main tool that is used to accomplish the given task.

7.5.2.5 Infix-to Postfix Procedure

Let *q* be an arithmetic expression written in infix notation. Besides operands and operators, it may also contain left and right parentheses. We assume the given expressions contain only operators listed in Table 7.1, and operations of the same precedence are evaluated from left to right.

Algorithm 7.2:

InfixToPostfix(*q*, *p*)

Where *q* is the arithmetic expression in-infix notation. This algorithm translates this expression into its equivalent postfix expression *p*.

```
Begin
  Create Empty Stack
  while ( not end of expression q ) do
    Scan q from left and pick up the next element, say celement
    if ( celement is left parentheses ) then
      Push celement onto stack
    else if ( celement is right parentheses ) then
      while ( ( stack is not empty )&&( element at top of stack is
                not left parentheses ) ) do
        Pop stack and append poped element to p
      endwhile
      if ( stack is empty ) then
        Print : "Error: Incorrect expression"
        Exit
      endif
      Remove the left parentheses from stack but do not add to P
    else if ( celement is operand ) then
      Append celement to P
    else if ( celement is operator ) then
      while ( ( element at top of stack is operator and its
                precedence ≥ celement ) && ( stack is not empty ) ) do
        Pop stack and append poped element to p
      endwhile
      Push celement onto stack
    else
      Print: "Error: Incorrect element in expression"
      Exit
    endif
  endwhile
  while ( ( stack is not empty ) && ( element at top of stack is not
            left parenthesis ) do
    Pop stack and append poped element to p
  endwhile
  if ( element at top of stack is left parenthesis ) then
    Print: "Error: Incorrect expression"
  endif
End.
```


Example 7.2:

Consider the following infix expression *q*:

(7-5)*(9/2)

The progress of the algorithm to translate *q* to postfix expression *p* is shown:

Character Scanned	Stack	Expression *p*
(	(	
7	(	7
–	(–	7
5	(–	7 5
)	Empty	7 5 –
*	*	7 5 –
(	* (	7 5 –
9	* (	7 5 – 9
/	* (/	7 5 – 9
2	* (/	7 5 – 9 2
)	*	7 5 – 9 2 /
End of expression	Empty	7 5 – 9 2 / *

Before scanning the infix expression stack is initialized to empty status.

Listing 7.24:

```
void infixToPostfix( char *source, char *target )
{
  char *s, *t;
  LinkStack<char> charStack;
  s = source;     t = target;
  while ( *s != '\0' ) {
    if ( ( *s == ' ' ) || ( *s == '\t' ) )  // skip white spaces
    {
      s++;
      continue;
    } else if ( *s == '(' ) {
      charStack.push(*s);
      s++;
    } else if ( *s == ')' ) {
      while ((charStack.isEmpty())&&(charStack.peek()!='(')) {
        *t = charStack.pop();
        t++;
        *t = ' ';    //  add one space
        t++;
      }
      if ( charStack.isEmpty() )
      {
```

```
            cout << "\nError: Incorrect expression\n";
            exit(1);
         }
         charStack.pop(); // remove left parenthesis from stack
         s++;
      } else if ( isdigit( *s ) || isalpha( *s ) ) {
         *t = *s;
         t++;
         *t = ' ';   //  add one space
         t++;
         s++;
      } else if (*s=='-' || *s=='-' || *s=='*' || *s=='/') {
         while ((!charStack.isEmpty())&&( charStack.peek()!='(') &&
                (getPriority(charStack.peek()) >= getPriority(*s)) )
         {
            *t = charStack.pop();
            t++;
            *t = ' ';   //  add one space
            t++;
         }
         charStack.push(*s);
         s++;
      } else {
         cout << "\nError: Incorrect element in expression \n";
         exit(1);
      }
   }
   while ((!charStack.isEmpty())&&(charStack.peek()!='('))
   {
      *t = charStack.pop();
      t++;
      *t = ' ';   //  add one space
      t++;
   }
   if ( charStack.peek() == '(' )
   {
      cout << "\nError: Incorrect expression\n";
      exit(1);
   }
   *t = '\0';   //  terminate the string by null character
}
```

The *infixToPostfix()* function uses *getPriority()* function whose listing is given below.

Listing 7.25:

```
int getPriority( char op ) {
   int priority;
   if ( op == '/' || op == '*' || op == '%' )
      priority = 1;
   else if ( op == '+' || op == '(' )
      priority = 0;
   return priority;
}
```


Listing 7.26:

```
//   Program to demonstrate the use of stack in converting arithmetic
//   expression from infix notation to postfix notation
#include <iostream.h>
#include <stdlib.h>
#include <ctype.h>
// header file containing code for generic stack implemented
// using linear linked list
#include "stlnk.h"
void infixToPostfix( char *source, char *target );
int getPriority( char op );

void main()
{
   char infixExp[80], postfixExp[80];
   cout << "\nEnter arithmetic expression in infix notation\n\n";
   cin.getline(infixExp,80);
   infixToPostfix( infixExp, postfixExp );
   cout << "\nEquivalent expression in postfix notation is\n\n";
   cout << postfixExp;
}/*---- end of main function ----*/

void infixToPostfix( char *source, char *target )
{
   char *s, *t;
   LinkStack<char> charStack;
   s = source;
   t = target;
   while ( *s != '\0' ) {
      if ( ( *s == ' ' ) || ( *s == '\t' ) )  // skip white spaces
      {
         s++;
         continue;
      } else if ( *s == '(' ) {
         charStack.push(*s);
         s++;
      } else if ( *s == ')' ) {
         while ((charStack.isEmpty())&&(charStack.peek()!='(')) {
            *t = charStack.pop();
            t++;
            *t = ' ';   //  add one space
            t++;
         }
         if ( charStack.isEmpty() )
         {
            cout << "\nError: Incorrect expression\n";
            exit(1);
         }
         charStack.pop(); // remove left parenthesis from stack
         s++;
      } else if ( isdigit( *s ) || isalpha( *s ) ) {
         *t = *s;
         t++;
         *t = ' ';   //  add one space
         t++;
         s++;
```

```
    } else if (*s=='+' || *s=='-' || *s=='*' || *s=='/') {
      while ((!charStack.isEmpty())&&( charStack.peek()!='(') &&
            (getPriority(charStack.peek()) >= getPriority(*s)) )
      {
         *t = charStack.pop();
         t++;
         *t = ' ';   //  add one space
         t++;
      }
      charStack.push(*s);
      s++;
    } else {
      cout << "\nError: Incorrect element in expression \n";
      exit(1);
    }
  }
  while ((!charStack.isEmpty())&&(charStack.peek()!='('))
  {
     *t = charStack.pop();
     t++;
     *t = ' ';   //  add one space
     t++;
  }
  if ( charStack.peek() == '(' )
  {
     cout << "\nError: Incorrect expression\n";
     exit(1);
  }
  *t = '\0';    //  terminate the string by null character
}
int getPriority( char op )
{
  int priority;
  if ( op == '/' || op == '*' || op == '%' )
     priority = 1;
  else if ( op == '+' || op == '(' )
     priority = 0;
  return priority;
}
```

7.5.2.6 Evaluating Expression in Postfix Notation

Algorithm 7.3:

EvaluatePostfixExpression(*p*, *result*)

Here *p* is the arithmetic expression in postfix notation. This algorithm evaluates this expression and returns the value of the expression through variable *result*.

```
Begin
  Create Empty Stack
  while ( not end-of expression p )  do
     if ( element is operand ) then
        Push element onto stack
     else
        Pop two elements and let first one is a and the second one is b
        Evaluate b⊗a, let its value be c, where ⊗ is an operator
        Push c onto stack
     endif
  endwhile
  Pop stack and assign this value to parameter result
End.
```


Example 7.3:

Consider the following postfix expression *p*:

7 5 – 9 2 / *

The progress of the algorithm to evaluate postfix expression *p* is shown in the following table.

Character Scanned	Stack
7	7
5	7 5
–	2
9	2 9
2	2 9 2
/	2 4.5
*	9
End of expression	9

Before scanning the infix expression stack is initialized to empty status. The final number in stack, *i.e.*, 9, is the value of postfix expression *p*.

The implementation of the algorithm 7.3 is given below.

Listing 7.27:

```
float evaluatePostfixExpression( char *exp )
{
  char c;
  LinkStack<float> floatStack;
  float op1, op2, value;
  while ( *exp )
  {
     if (*exp == ' ' || *exp == '\t' || *exp == ',')
     {
       exp++;
       continue;
     }
     if ( isdigit( *exp ) )
       //  convert to numerical value
       floatStack.push((float)(*exp - '0'));
     else
     {
       op2 = floatStack.pop();  // second operand
```

```
    op1 = floatStack.pop();  // first operand
    switch ( *exp )
    {
      case '+' : value = op1 + op2;
                 break;
      case '-' : value = op1 - op2;
                 break;
      case '/' : value = (float) op1 / op2;
                 break;
      case '*' : value = op1 * op2;
                 break;
      default  : cout << "\nIllegal operation ... %x7\n";
                 exit(1);
    }
    floatStack.push(value);
  }
  exp++;
}
return ( floatStack.pop() );
}
```

Listing 7.28:

```
//   Program to demonstrate the use of stack in evaluating arithmetic
//   expression in postfix notation
#include<iostream.h>
#include<stdlib.h>
#include<ctype.h>
// header file containing code for generic stack implemented
// using linear linked list
#include "stlnk.h"
float evaluatePostfixExpression( char *exp );
void main()
{
  float value;
  char *postfixExp = "";
  cout << "\nEnter arithmetic expression in postfix notation\n\n";
  cin.getline(postfixExp,80);
  value = evaluatePostfixExpression(postfixExp);
  cout << "\nValue of expression = " << value;
} /*----- end of main function -----*/
float evaluatePostfixExpression( char *exp )
{
  char c;
  LinkStack<float> floatStack;
  float op1, op2, value;
  while ( *exp )
  {
    if (*exp == ' ' || *exp == '\t' || *exp == ',')
    {
      exp++;
      continue;
    }
    if ( isdigit( *exp ) )
```



```
      // convert to numerical value
      floatStack.push((float)(*exp - '0'));
    else
    {
      op2 = floatStack.pop();  // second operand
      op1 = floatStack.pop();  // first operand
      switch ( *exp )
      {
        case '+' : value = op1 + op2;
                   break;
        case '-' : value = op1 - op2;
                   break;
        case '/' : value = (float) op1 / op2;
                   break;
        case '*' : value = op1 * op2;
                   break;
        default  : cout << "\nIllegal operation ... %x7\n";
                   exit(1);
      }
      floatStack.push(value);
    }
    exp++;
  }
  return ( floatStack.pop() );
}
```

7.5.3 Quicksort Algorithm

Quicksort is an algorithm that uses *divide-and-conquer* strategy to sort an array. In the divide-and-conquer strategy, the problem to be solved is divided into two sub problems, which are then solved separately. Then the solutions of these two sub problems are combined to generate the solution of the given problem. Quicksort method reduces the problem of sorting a set of number to the problem of sorting two subsets of smaller size. This reduction step is illustrated below with the help of an example.

Consider the following list of 8 numbers.

20 22 15 10 35 12 44 38

The reduction step of the quicksort algorithm finds the final position of the first number, 20 in case.

This is accomplished as follows.

(20) 22 15 10 35 12 44 (38)

Beginning with the rightmost (last) number, 38, scan the list from right to left, comparing number with 20, and stopping at the first number less than 20. The number is 12. Interch 20 and 12 to obtain the list

(12) 22 15 10 35 (20) 44 38

Now beginning with 12, scan the list from left to right, comparing each number with 20, and stopping at the first number greater than 20. The number is 22. Interchange 22 and 20 to obtain the list

12 (20) 15 10 35 (22) 44 38

Now beginning with 22, scan the list from right to left, comparing each number with 20, and stopping at the first number less than 20. The number is 10. Interchange 10 and 20 to obtain the list

12 (10) 15 (20) 35 22 44 38

Now beginning with 10, scan the list from left to right, comparing each number with 20, and stopping at the first number greater than 20. We do not meet any such number before meeting 20. This means all the numbers have been scanned and compared with 20. Further, all the numbers less than 20 now form the sub list of numbers to the left of 20, and all numbers greater than 20 now form the sub list of numbers to the right of 20 as shown below:

12 10 15 (20) 35 22 44 38

Left sub list Right sub list

Thus element 20 is correctly placed in its final position, and the task os sorting the entire list is now reduced to sorting two sub lists of smaller size. The above reduction step is repeated with each sub list containing two or more elements. Since we can process one list at a time, we must keep track of the other sub list for future processing. This is accomplished by using a stack with integer elements to temporarily hold these sub lists. That is, the index of the first element and the last element of each sub list are pushed onto the stack. The reduction step is applied only once these indices are removed from the stack.

Algorithm 7.4:

Reduction(*a, beg, end, loc*)

Here *a* is an array in memory, and *beg* and *end* represents the lower bound and upper bound of the sub list in question. This algorithm places the first element of given sub list at its final position *loc*.

```
Begin
  Set left = beg, right = end, loc = beg
  Set done = false
  while ( not done ) do
    while ( a[loc] < a[right] ) and ( loc ≠ right ) ) do
      Set right = right - 1
    endwhile
    if ( loc = right ) then
      Set done = true
    else
      Interchange a[loc] and a[right]
      Set loc = right
    endif
    if ( not done ) then
    while ( a[loc] > a[left] ) and ( loc ≠ left ) ) do
      Set left = left + 1
    endwhile
    if ( loc = left ) then
      Set done = true
    else
      Interchange a[loc] and a[left]
      Set loc = left
    endif
  endif
endwhile
End.
```


The implementation of the above algorithm is given below.

Listing 7.29:

```
void reduction( int a[], int beg, int end, int *loc )
{
  int left, right, temp;
  boolean done;
  left = *loc = beg;
  right = end;
  done = false;
  while ( !done ) {
    while ( ( a[*loc] < a[right] ) && ( *loc != right ) )
      right--;
    if ( *loc == right )
      done = true;
    else {
      temp = a[*loc];
       a[*loc] = a[right];
      a[right] = temp;
       *loc = right;
    }
    if ( !done ) {
      while ( ( a[*loc] > a[left] ) && ( *loc != left ) )
        left++;
      if ( *loc == left )
        done = true;
      else {
        temp = a[*loc];
        a[*loc] = a[left];
        a[left] = temp;
        *loc = left;
      }
    }
  }
}
```

The algorithm for iterative version of the quicksort algorithm will look likes as shown below.

Algorithm 7.5:

QuickSortIterative(*a, n*)

Here *a* is an array of size *n* (> 1) in memory. This algorithm sorts this array in ascending order using Quick sort technique.

```
Begin
  Create Empty stack
  Push array bounds (lower bound = 0, and upper bound = n-1) onto STACK
  while ( stack is not empty ) do
     Pop stack and initialize the array bounds of the current sub list
     to variable beg and end
     Call Reduction( a, beg, end, loc)
     if ( beg < loc - 1 ) then
        Push array bounds of the left sub list onto stack
     endif
     if ( end  > loc + 1 ) then
         Push array bounds of the right sub list onto stack
     endif
  endwhile
End.
```

The implementation of the above algorithm is given below.

Listing 7.30:

```
void quickSortIterative(  int a[], int n )
{
   int loc, beg, end;
   LinkStack<int> intStack;
   intStack.push(0);
   intStack.push(n-1);
   while ( ! intStack.isEmpty() {
      end = intStack.pop();
      beg = intStack.pop();
      reduction( a, beg, end, &loc);
      if ( beg < loc - 1 ) {
           intStack.push(beg);
           intStack.push(loc-1);
      }
      if ( end > loc + 1 ) {
           intStack.push(loc+1);
           intStack.push(end);
      }
   }
}
```

The above code uses a linked stack with integer elements. This function will be called as

```
quickSortIterative( a, n );
```

However, the recursive implementation of Quicksort algorithm will look like as shown below

Algorithm 7.6:

QuickSortRecursive(*a, lb, ub*)

Here *a* is an sub list with starting index *lb* and ending index *ub*. This algorithm sorts this sub list in ascending order.

```
Begin
  if ( lb < ub )  then
     Call Reduction( a, lb, ub, loc)
     Call QuickSortRecursive( a, lb, loc - 1 )
     Call QuickSortRecursive( a, loc + 1, ub )
  endif
End.
```

The implementation of the above algorithm is shown in the following listing.

Listing 7.31:

```
void quickSortRecursive( int a[], int lb, int ub )
{
  int loc;
  if ( lb < ub )
  {
      reduction( a, lb, ub, &loc);
      quickSortRecursive( a, lb, loc - 1 );
      quickSortRecursive( a, loc + 1, ub );
  }
}
```

The original call to this function will be

```
quickSortRecursive( a, 0, n -1 );
```

Listing 7.32:

```
// Program to demonstrate the use of stack in implementing Quicksort
// algorithm to sort an array of integers in ascending order
#include<iostream.h>
#include<stdlib.h>
#include<ctype.h>
// header file containing code for generic stack implemented
// using linear linked list
#include "stlnk.h"
#define MAX 20
void reduction( int a[], int beg, int end, int *loc );
void quickSortIterative( int a[], int n );
void main()
{
  int i, n, a[MAX];
  cout << "\nEnter number of elements in array : " ;
```

```
   cin >> n;
   if ( n > MAX ) {
      cout << "\nInput size of array greater than declared size\n";
      exit(1);
   }
   cout << "\nEnter " << n << " elements\n\n";
   for ( i = 0; i < n; i++ )
       cin >> a[i];
   quickSortIterative( a, n );
   cout << "\n\nSorted list of elements\n\n";
   for ( i = 0; i < n; i++ )
       cout << a[i] << "  ";
   cout << endl;
} /*---- end of main function ----*/
void quickSortIterative(  int a[], int n )
{
   int loc, beg, end;
   LinkStack<int> intStack;
   intStack.push(0);
   intStack.push(n-1);
   while ( ! intStack.isEmpty() ) {
      end = intStack.pop();
      beg = intStack.pop();
      reduction( a, beg, end, &loc);
      if ( beg < loc - 1 ) {
         intStack.push(beg);
      intStack.push(loc-1);
      }
      if ( end > loc + 1 ) {
         intStack.push(loc+1);
         intStack.push(end);
      }
   }
}
void reduction( int a[], int beg, int end, int *loc )
{
   int left, right, temp;
   int done = 0;
   left = *loc = beg;
   right = end;
   while ( !done )
   {
      while ( ( a[*loc] < a[right] ) && ( *loc != right ) )
        right--;
      if ( *loc == right )
        done = 1;
      else {
        temp = a[*loc];
      a[*loc] = a[right];
        a[right] = temp;
      *loc = right;
      }
      if ( !done ) {
        while ( ( a[*loc] > a[left] ) && ( *loc != left ) )
           left++;
```



```
      if ( *loc == left )
        done = 1;
      else {
        temp = a[*loc];
        a[*loc] = a[left];
        a[left] = temp;
        *loc = left;
      }
    }
  }
}
```

REVIEW EXERCISES

1. What is a stack? Is stack a linear or non-linear data structure?
2. Stack is a special type of list? Comment.
3. How does an array based stack compares with linked stack?
4. Why an operation to check stack overflow is not implemented on linked stacks?
5. Can we implement a stack in C whose elements can be of different type? If your answer is yes, the how?
6. How *peek()* operation differs from *pop()* operation?
7. While representing a stack using linked lists, we have implemented additional operation *dispose()* to remove the stack from memory. Why such operation is not implemented for stack represented using an array.
8. What are different notations to represent a mathematical expression? Which notation is most suitable for use in computers? Give at least one example in each case.
9. Translate following mathematical expressions into its equivalent postfix expressions:
   (*i*) (A – B) * (D / E)
   (*ii*) (A + B / D) / (E – F) + G
   (*iii*) A * B – (C + D) / (E – F) + G / H
   (*a*) by inspection
   (*a*) by using algorithm 7.2.
10. Given the following arithmetic expression in infix notation as

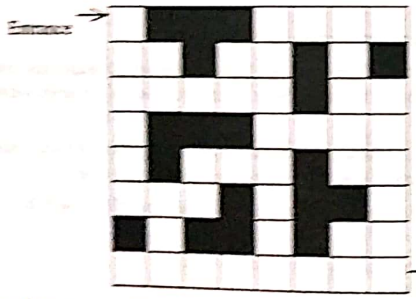
    12 / (7 – 3) + 2 * (3 + 8) – 7

    Translate this expression into postfix notation and then evaluate it.
11. How iterative implementation of quick sort method will process following list of numbers:

    23 12 11 45 30 8 55 22 66

PROGRAMMING EXERCISES

1. Write a menu driven program to implementing the various operations on a ... represented using an array.
2. Write a menu driven program to implementing the various operations on a ... represented using a linked list.
3. Design a data structure to represent two stacks in an array named *elements* of size ... write functions for *push()* and *pop()* operations to insert and delete element from ... $0 \le i \le 1$. Your functions should be able to add elements to the stacks as long as ... less than *n* elements in both stacks together and should run in O(1) time.
4. Design a data structure to represent *m* stacks in an array named *elements* of size ... write functions for *push()* and *pop()* operations to insert and delete element from ... $0 \le i \le m-1$. These operations should run in O(1) time.
5. **Rat in a Maze problem:** A maze is a rectangular area, with *m* rows and *n* columns, ... entrance and an exit. The interior of the maze contains obstacles. The entrance ... upper-left corner, and the exit is at the lower-right corner. One sample maze of size ... shown below:



One such path is

{ (1,1), (2,1), (2,2), (2,3), (3,3), (3,4), (3,5), (4,5), (5,5), (6,5), (7,5), (8,5), (8,6), (8,7), (8,8) }

A rat in a maze problem is to find a path from the entrance to exit of a maze. A path is basically a sequence of positions, none of which is blocked, and such that each position is the first to the north, south, east or west neighbour of the preceding position. Write a program to find such path, if exists, otherwise output message "*No solution for this maze*".

Chapter 8

Queues

8.1 INTRODUCTION

All of us are familiar with queues. In simple terms, a queue is a line of persons waiting for their turn at some service counter/window. A service counter can be a ticketing window of a cinema hall, a ticketing window of a bus stand or railway station, a service counter of a bank, a service counter of a gas agency, a fees deposit counter in a college, etc. Depending on the type of service provided by the service counter and the number of persons interested in this service, there could be queues of varying lengths. The service at the service counter is provided on the First-Come First-Serve (FCFS) basis, i.e., in order of their arrival in the queue.

Figure 8.1: Queue of passengers waiting for a bus at a bus stop

Suppose that at service counter, t_1 units of time is needed to provide a service to a single pe and on average a new person arrives at the service counter in t_2 units of time. The follo possibilities may arise:

1. If $t_1 < t_2$, then the service counter will be free for some time before a new person arri the service counter. Hence, no queue in this case.
2. If $t_1 > t_2$, then service counter will be busy for some time even after the arrival of a person. Therefore, this person has to wait for some time. Similarly, other persons arriv the service counter have to wait. Hence, a queue will be formed.
3. If $t_1 = t_2$, then as a person leaves the service counter, a new persons arrives at the se counter. Hence, no queue in this cases also.

This chapter describes a queue, its representation in computer memory, and the implementa of different operations that can be performed on them. It also describes a deque and a pri queue in brief.

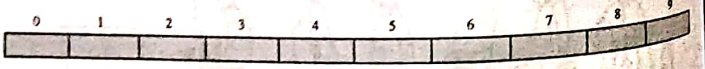
8.2 QUEUE DEFINED

In the context of data structures, a *queue* is a linear list in which insertions can take place at one end of the list, called the *rear* of the list, and deletions can take place only at other end, calle the *front* of the list. The behaviour of a queue is like a First-In-First-Out (FIFO) syst Figure 8.2 shows a queue as black box in which elements enters from one side and leaves fr other side.

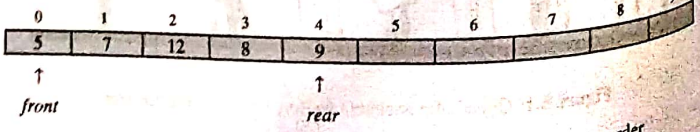
In queue terminology, the insertion and deletion operations are known as *enqueue* and deque operations.

Example 8.1:

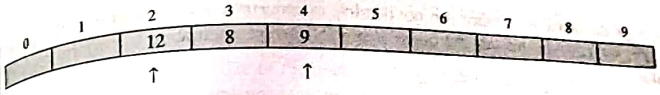
The following figure illustrates a queue, which can accommodate maximum of 10 elements



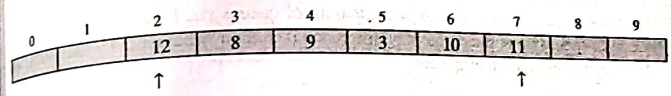
(*a*) An empty queue



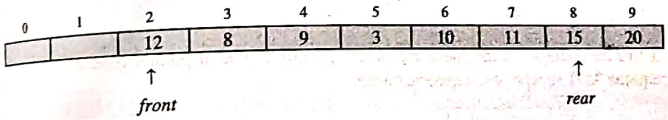
(*b*) Queue after enqueuing elements 5, 7, 12, 8, and 9 in order



(*c*) Queue after dequeuing elements 5, 7 from queue



(*d*) Queue after enqueuing elements 3, 10, 11 in a queue



(*e*) Queue after enqueuing elements 15 and 20 in a queue

Figure 8.2: Illustration of a queue whose elements are integer values

8.3 OPERATIONS ON QUEUES

The following operations are performed on queues:

Table 8.1: Various operations performed on queues

Operation	Description
create	To create an initial queue. This task will be accomplished using a constructor.
destroy	To remove (delete) queue from memory. This task will be accomplished using a destructor.
enqueue	Insert (push) an element at the rear of the queue.
dequeue	Access and remove the front element from the queue.
peek	Access the front element of the queue without removing it from the queue.
isFull	Check whether the queue is full.
isEmpty	Check whether the queue is empty.

All of these operation runs in O(1) time.

Note that if the queue is empty then it is not possible to dequeue the queue. Similarly, if there is no element in the queue, the *peek* operation is also not valid. Therefore, we must ensure that the queue is not empty before attempting to perform these operations.

Likewise, if the queue is full then it is not possible to enqueue a new element in to the queue. Therefore, we must ensure that the queue is not full before attempting to perform *enqueue* operation.

8.4 REPRESENTATION OF QUEUES IN MEMORY

Queues, like stacks, can also be represented in memory using a linear array or a linear linked list. Even using a linear, we have two implementations of queues viz. *linear* and *circular*. The representation of these two is same, the only difference is in their behaviour. To start with, let us consider array representation of queues.

8.4.1 Representing a Queue using an Array

To implement a queue we need two variables, called *front* and *rear*, that holds the index of the first and last element of the queue and an array, named *elements*, to hold the elements of the queue.

Suppose that the elements of the queue can be any primitive type or pointer type, the following is the required ADT to represent a generic queue:

The following are the necessary declarations

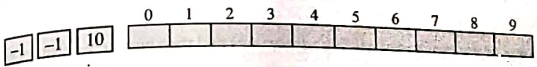
```
template <class T>
class LinearQueue
{
   protected:              // data members
      int front;
      int rear;
      int size;
      T *elements;

   public:                 // member functions
      // zero argument constructor,
      // it constructs a stack with default size = 10
      LinearQueue();
      // parametrized constructor that constructs,
      // a stack with size = n
      LinearQueue(int n);
      // destructor that removes the stack from memory
      ~LinearQueue();
      // Check whether the stack is empty
      int isEmpty();
      // Check whether the stack is full
      int isFull();
      // Insert an element at the end of the queue
      void enqueue(T value);
      // Access and remove the first element of the queue
      T dequeue();
      // Access the first element of the queue
      // without removing it from the queue
      T peek();
}; // -------- End of LinearQueue class --------
```

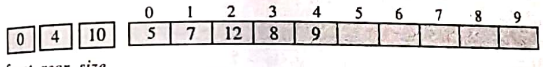

We have defined our own data type named *LinearQueue*, which is class and whose first data member *front* will be used to hold the index of the first element of the queue, second data member *rear* will be used to hold the index of the last element of the queue, third data member *size* will be used to hold the size of the queue, and fourth data member *elements*, a dynamic array whose elements can be any primitive type or pointer type.

The implementation of the member functions is given in the subsequent sections.

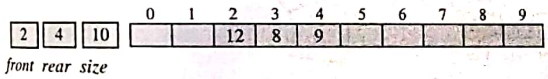
With these declarations, assuming a queue whose elements are of type *int* and size is 10, a queue of Figure 8.2 will be represented in computer memory as



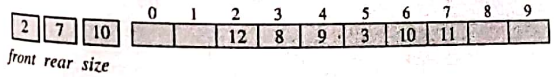
(*a*) Representation of queue of Figure 8.2(*a*) in memory



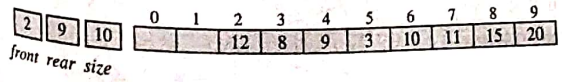
(*b*) Representation of queue of Figure 8.2(*b*) in memory



(*c*) Representation of queue of Figure 8.2(*c*) in memory



(*d*) Representation of queue of Figure 8.2(*d*) in memory



(*e*) Representation of queue of Figure 8.2(*e*) in memory

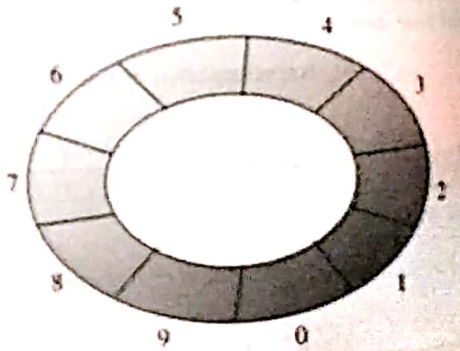
Figure 8.3: Array representation of linear queue of Figure 8.2

You must have observed, as shown in Figure 8.3(e), that it is not possible to enqueue the element as such, though two positions in the linear queue are vacant. To overcome this problem, the elements of the queue are moved forward, so that the vacant positions are shifted towards the rear end of the linear queue. After shifting, *front* and *rear* are adjusted properly and then the element is enqueued in the linear queue as usual. For example, we want to enqueue one more element, say 60, after making adjustments the queue will look as shown in Figure 8.4.

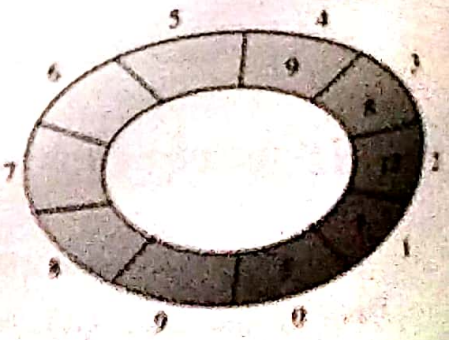
Figure 8.4: State of the queue after making adjustments and then enqueuing element 60

However, this difficulty can be overcome if we treat the queue position with index 0 as the position that comes after position with index 9, *i.e.*, we treat the queue as circular as shown in Figure 8.5.

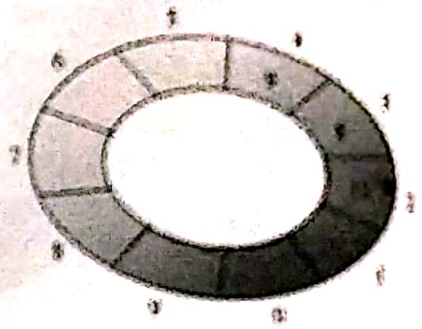
(a) Representation of queue of Figure 8.2(a)



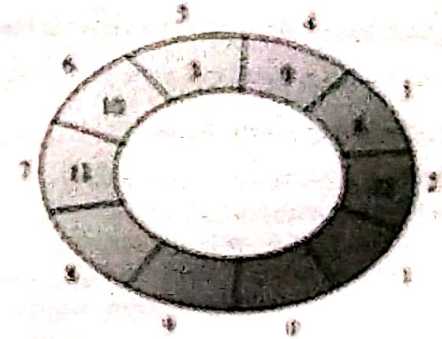
(b) Representation of queue of Figure 8.2(b)



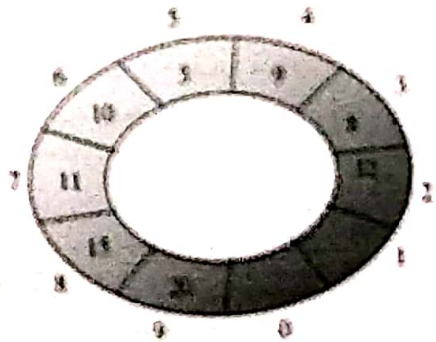
(c) Representation of queue of Figure 8.2(c)



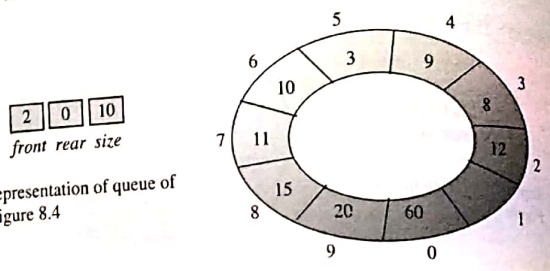
(d) Representation of queue of Figure 8.2(d)



(e) Representation of queue of Figure 8.2(e)



Now to enqueue one more element, say 60, the *rear* is reset to 0, and the element is inserted at the index as shown below.



(f) Representation of queue of Figure 8.4

Figure 8.5: (a) – (f) Illustration of Operations on Circular Queues

8.4.1.1 Implementation of Operations on a Linear Queue

The implementation of different operations on a linear queue is described below.

8.4.1.1.1 Creating an Empty Linear Queue

Before we can use a queue, it is to be initialized. As the index of array elements can take a value in the range 0 to *size*−1, the purpose of initializing the queue is served by assigning val −1 (sentinel value) to *front* and *rear* variables. This simple task is accomplished using following constructors, where the zero argument constructors create a queue with default size 10 while the parameterized constructor creates a queue of use-defined size (*n*).

Listing 8.1:

```
// zero argument constructor,
// it constructs a queue with default size = 10
template <class T>
LinearQueue<T>::LinearQueue()
{
   elements = new T[10];
   size = 10;
   front = rear = -1;
}
// parametrized constructor that constructs,
// a queue with size = n
template <class T>
LinearQueue<T>::LinearQueue(int n)
{
   elements = new T[n];
   size = n;
   front = rear = -1;
}
```


8.4.1.2 Testing a Linear Queue for Underflow

Before we remove an element from a queue, it is necessary to test whether a queue still have some elements, *i.e.*, to test that whether the queue is empty or not. If it is not empty then the dequeue operation can be performed to remove the front element. This test is performed by comparing the value of *front* with sentinel value −1, as shown in the following function.

Listing 8.2:

```
// Check whether the queue is empty
template <class T>
int LinearQueue<T>::isEmpty()
{
  return ( front == -1 ? 1 : 0 );
}
```

The above function returns value 1, indicating that queue is empty, if the test condition is satisfied; otherwise it returns value 0, indicating that queue is not empty.

8.4.1.3 Testing a Linear Queue for Overflow

Before we insert new element in a queue, it is necessary to test whether queue still have some space to accommodate the incoming element i.e. to test that whether the queue is full or not. If it is not full then the *enqueue* operation can be performed to insert the element at rear end of the queue. This test is performed by test condition

```
( front = 0 ) and ( rear = size - 1 )
```

i.e., front element is at position with index 0 and the last element is at position with index *size*−1, which indicates that no position in linear queue is available.

This task is accomplished by the following function

Listing 8.3:

```
// Check whether the queue is full
template <class T>
int LinearQueue<T>::isFull()
{
  return ( ((front==0)&&(rear==size-1)) ? 1 : 0 );
}
```

The above function returns value 1, indicating that queue is full, if the test condition is satisfied; otherwise it returns value 0, indicating that queue is not full.

8.4.1.1.4 Enqueue Operation on Linear Queue

Even if a linear queue is not full, there is another problem that may arise during enqueue operation. That problem is that, the value of the *rear* variable is equal to *size*−1, though the value of the *front* variable is not equal to 0. This possibility implies that the incoming element cannot be enqueued unless the elements are moved forward, and variables *front* and *rear* are adjusted accordingly. Which is, of course, a time consuming process.

There are two conditions, which can occur, even if the queue is not full. These are

- If a linear queue is empty, then the value of the *front* and *rear* variables will be −1 (sentinel value), then both *front* and *rear* are set to 0.
- If a linear queue is not empty, then there are further two possibilities:
  - If the value of the *rear* variable is less than *size*−1, then the *rear* variable is incremented.
  - If the value of the *rear* variable is equal to *size*−1, then the elements of the linear queue are moved forward, and the *front* and *rear* variables are adjusted accordingly.

This task is accomplished as shown in the following function

Listing 8.4:

```
// Insert an element at the end of the queue
template <class T>
void LinearQueue<T>::enqueue(T value)
{
   int i;
   if ( LinearQueue.isEmpty() )
       front = rear = 0;
   else if ( rear == size-1 ) {
       for ( i = front, i <= rear; i++ )
           elements[i - front] = elements[i];
       rear = rear - front + 1;
       front = 0;
   } else {
       rear++;
   }
   elements[rear] = value;
}
```

8.4.1.1.5 Dequeue Operation on a Linear Queue

The front element of a linear queue is assigned to a local variable, which later on will be returned via the *return* statement. After assigning the front element of a linear queue to a local variable, the value of the *front* variable is modified so that it points to the new front.

There are two possibilities:

- If there was only one element in a linear queue, then after dequeue operation queue will become empty. This state of a linear queue is reflected by setting *front* and *rear* variables to −1, the sentinel value.
- Otherwise the value of the *front* variable is incremented.

This task is accomplished as shown in the following function

Listing 8.5:

```
// Access and remove the first element of the queue
template <class T>
T LinearQueue<T>::dequeue()
{
   T temp;
   temp = elements[front];
   if ( front == rear )
       front = rear = -1;
   else
       front++;
   return temp;
}
```

8.4.1.1.6 Accessing Front Element

The element in the front of queue is accessed as shown in the following function

Listing 8.6:

```
// Access the first element of the queue
// without removing it from the queue
template <class T>
T LinearQueue<T>::peek()
{
   return elements[front];
}
```

8.4.1.1.7 Removing Queue from Memory

As soon as the *LinearQueue* goes out of scope, it needs to be removed from computer memory. This is done using following destructor:

Listing 8.7:

```
// destructor that removes the queue from memory
template <class T>
LinearQueue<T>::~LinearQueue()
{
   delete elements;
   front = rear = -1;
   size = 0;
}
```

Listing 8.8:

```
//  Program to demonstration the implementation of various operations
//  on a linear queue represented using a linear array
#include<iostream.h>
#include<iomanip.h>

template <class T>
class LinearQueue
{
   protected:              // data members
      int front;
      int rear;
      int size;
      T *elements;

   public:                 // member functions
      // zero argument constructor,
      // it constructs a stack with default size = 10
      LinearQueue()
      {
         elements = new T[10];
         front = rear = -1;
         size = 10;
      }
      // parametrized constructor that constructs,
      // a stack with size = n
      LinearQueue(int n)
      {
         elements = new T[n];
         front = rear = -1;
         size = n;
      }
      // destructor that removes the stack from memory
      ~LinearQueue()
      {
         delete elements;
         front = rear = -1;
         size = 0;
      }
      // Check whether the stack is empty
      int isEmpty()
      {
         return ( front == -1 ? 1 : 0 );
      }
      // Check whether the stack is full
      int isFull()
      {
         return ( ((front==0)&&(rear==size-1)) ? 1 : 0 );
      }
      // Insert an element at the end of the queue
      void enqueue(T value);
      // Access and remove the first element of the queue
      T dequeue()
      {
         T temp;
         temp = elements[front];
         if ( front == rear )
```



```
            front = rear = -1;
         else
            front++;
         return temp;

      }
      // Access the first element of the queue
      // without removing it from the queue
      T peek()
      {
         return elements[front];

      }
}; // -------- End of LinearQueue class --------

// Insert an element at the end of the queue
template <class T>
void LinearQueue<T>::enqueue(T value)
{
   int i;
   if ( isEmpty() )
      front = rear = 0;
   else if ( rear == size-1 ) {
      for ( i = front; i <= rear; i++ )
       elements[i - front] = elements[i];
      rear = rear - front + 1;
      front = 0;
   } else {
      rear++;
   }
   elements[rear] = value;
}
void main()
{
   int choice, element, m;
   cout << "\nSize of queue you want to use : ";
   cin >> m;
   LinearQueue<int> intQueue(m);
   do
   {
      cout << "\n\nOptions available \n";
      cout << "++++++++++++++++++ \n\n";
      cout << "    1. Enqueue\n";
      cout << "    2. Dequeue\n";
      cout << "    3. Peek\n";
      cout << "    4. Exit\n\n";
      cout << "Enter your choice ( 1-4 ) : ";
      cin >> choice;
      switch ( choice )
      {
        case 1: if ( intQueue.isFull() )
                   cout << "\nQueue full...\n";
                else {
                   cout << "Enter value : " ;
                   cin >> element;
                   intQueue.enqueue(element);
                }
                break;
        case 2: if ( intQueue.isEmpty() )
                  cout << "\nQueue empty...\n";
```

```
                else
                   cout << "Value dequeued is " << intQueue.dequeue();
                break;
        case 3: if ( intQueue.isEmpty() )
                   cout << "\nQueue empty...\n";
                else
                   cout << "First Element of queue is "
                        << intQueue.peek();

      }
   }
   while ( choice != 4 );
} //---- end of main function ---
```

8.4.1.2 Limitations of a Linear Queue

The only limitation of linear queue is that —

If the last position of the queue is occupied, it is not possible to enqueue any more elements even though some positions are vacant towards the front positions of the queue.

However, this limitation can be overcome by moving the elements forward, such that the first element of the queue goes to position with index 0, and the rest of the elements move accordingly. And finally the *front* and *rear* variables are adjusted appropriately.

But this operation may be very time consuming if the length of the linear queue is very long. As mentioned earlier, this limitation can be overcome if we treat that the queue position with index 0 comes immediately after the last queue position with index *size*–1. The resulting queue is known as *circular queue*.

8.4.1.3 Implementation of Operations on a Circular Queue

The operations of initialization, testing for underflow for circular queue is similar to that of a linear queue. However, other operations are slightly different. These are described below.

8.4.1.3.1 Testing a Circular Queue for Overflow

Before we insert new element in a circular queue, it is necessary to test whether a circular queue still have some space to accommodate the incoming element i.e. to test that whether a circular queue is full or not. If it is not full then the enqueue operation can be performed to insert the element at rear of a circular queue. This test is performed by test conditions

- `( front = 0 )` and `( rear = size-1)`
- `front = rear + 1`

If any one of these two conditions is satisfied, it means circular queue is full.

This task is accomplished by the following function

Listing 8.9:

```
// Check whether the queue is full

template <class T>
int CircularQueue::isFull()
{
  if ( ((front==0)&&(rear==size-1))||(front==rear+1) )
    return 1;
  else
    return 0;
}
```

The above function returns value 1, indicating that queue is full, if the test condition is satisfied; otherwise it returns value 0, indicating that queue is not full.

8.4.1.3.2 Enqueue Operation on a Circular Queue

Before the enqueue operation, there are three conditions, which can occur, even if a circular queue is not full. These are

- If the queue is empty, then the value of the *front* and *rear* will be –1 (sentinel value), then both *front* and *rear* are set to 0.
- If the queue is not empty then the value of the *rear* will be the index of the last element of the queue, then the *rear* variable is incremented.
- If the queue is not full and the value of *rear* variable is equal to *size*–1, then *rear* variable is set to 0.

This task is accomplished as shown in the following function

Listing 8.10:

```
// Insert an element at the end of the queue

template <class T>
void CircularQueue::enqueue( T value )
{
  if ( front == -1 )
    front = rear = 0;
  else if ( rear == size-1 )
    rear = 0;
  else
    rear++;
  elements[rear] = value;
}
```

The *rear* index can also be adjusted as shown in the following code.

Listing 8.11:

```
// Insert an element at the end of the queue
template <class T>
void CircularQueue::enqueue( T value )
{
   if ( front == -1 )
      front = rear = 0;
   else
      rear = ( rear + 1 ) % size+;
   elements[rear] = value;
}
```

8.4.1.3.3 Dequeue Operation on a Circular Queue

The front element of a circular queue is assigned to a local variable, which later on will be returned via the *return* statement. After assigning the front element of a circular queue, the value of the *front* variable is modified so that it points to the new front.

There are two possibilities:

- If there was only one element in a circular queue, then after dequeue operation the circular queue will become empty. This state of a circular queue is reflected by setting *front* and *rear* variables to –1, the sentinel value.
- If value of the front variable is equal to *size*–1, then set *front* variable to 0.

If none of the above conditions hold, then the *front* variable is incremented.

This task is accomplished as shown in the following function.

Listing 8.12:

```
// Access and remove the first element of the queue
template <class T>
T LinearQueue<T>::dequeue()
{
    T temp;
    temp = elements[front];
    if ( front == rear )
       front = rear = -1;
    else if ( front == size-1 )
       front = 0;
    else
       front++;
    return temp;
}
```

The *front* index can also be adjusted as shown in the following code.

Listing 8.13:

```
// Access and remove the first element of the queue
template <class T>
T LinearQueue<T>::dequeue()
{
   T temp;
   temp = elements[front];
   if ( front == rear )
      front = rear = -1;
   else
      front = ( front + 1 ) % size;
   return temp;
}
```

Listing 8.14:

```
// Program to demonstration the implementation of various operations
// on a circular queue represented using a linear array
#include<iostream.h>
#include<iomanip.h>

template <class T>
class CircularQueue
{
  protected:            // data members
     int front;
     int rear;
     int size;
     T *elements;

  public:               // member functions
     // zero argument constructor,
     // it constructs a stack with default size = 10
     CircularQueue()
     {
        elements = new T[10];
        front = rear = -1;
        size = 10;
     }
     // Parametrized constructor that constructs,
     // a stack with size = n
     CircularQueue(int n)
     {
        elements = new T[n];
        front = rear = -1;
        size = n;
     }
     // destructor that removes the stack from memory
     ~CircularQueue()
     {
        delete elements;
        front = rear = -1;
        size = 0;
     }
     // Check whether the stack is empty
```

```
    int isEmpty()
    {
       return ( front == -1 ? 1 : 0 );
    }
    // Check whether the stack is full
    int isFull()
    {
       if ( ((front==0)&&(rear==size-1))||(front==rear+1) )
          return 1;
       else
          return 0;
    }
    // Insert an element at the end of the queue
    void enqueue(T value)
    {
       if ( front == -1 )
          front = rear = 0;
       else
          rear = ( rear + 1 ) % size;
       elements[rear] = value;
    }
    // Access and remove the first element of the queue
    T dequeue()
    {
       T temp;
       temp = elements[front];
       if ( front == rear )
          front = rear = -1;
       else
          front = ( front + 1 ) % size;
       return temp;
    }
    // Access the first element of the queue
    // without removing it from the queue
    T peek()
    {
       return elements[front];
    }
}; // -------- End of CircularQueue class --------

void main()
{
   int choice, element, m;
   cout << "\nSize of queue you want to use : ";
   cin >> m;
   CircularQueue<int> intQueue(m);
   do
   {
      cout << "\n\nOptions available \n";
      cout << "*****************  \n\n";
      cout << "    1. Enqueue\n";
      cout << "    2. Dequeue\n";
      cout << "    3. Peek\n";
      cout << "    4. Exit\n\n";
      cout << "Enter your choice ( 1-4 ) : ";
      cin >> choice;
```



```
      switch ( choice )
      {
       case 1: if ( intQueue.isFull() )
                  cout << "\nQueue full...\n";
               else {
                  cout << "Enter value : " ;
                  cin >> element;
                  intQueue.enqueue(element);
               }
               break;
       case 2: if ( intQueue.isEmpty() )
                  cout << "\nQueue empty...\n";
               else
                  cout << "Value dequeued is " << intQueue.dequeue();
               break;
       case 3: if ( intQueue.isEmpty() )
                  cout << "\nQueue empty...\n";
               else
                  cout << "First Element of queue is "
                       << intQueue.peek();
      }
   }
   while ( choice != 4 );
   //---- end of main function ---
```

Note that we created *CircularQueue* from scratch. However, since the only difference in the implementation of *LinearQueue* and *CircularQueue* class is the implementation of the *isFull()*, *enqueue()*, and *dequeue()* member functions, we can create *CircularQueue* by inheriting (deriving) it from *LinearQueue* class and overriding the *isFull()*, *enqueue()*, and *dequeue()* member functions of the *LinearQueue* class, as shown below:

```
template <class T>
class CircularQueue : public LinearQueue<T>
{
   public:              // member functions
      // zero argument constructor,
      // it constructs a stack with default size = 10
      CircularQueue() : LinearQueue<T>() { }

      // parametrized constructor that constructs
      // a stack with size = n
      CircularQueue(int n) : LinearQueue<T>(n) { }
      // destructor that removes the stack from memory

      ~CircularQueue() { }

      // Override method of LinearQueue class
      int isFull()
      {
       if ( ((front==0)&&(rear==size-1))||(front==rear+1) )
          return 1;
       else
          return 0;
       }
```

```
// Override method of LinearQueue class
// Insert an element at the end of the queue
void enqueue(T value)
{
 if ( front == -1 )
    front = rear = 0;
 else
    rear = ( rear + 1 ) % size;
 elements[rear] = value;
}

// Override method of LinearQueue class
// Access and remove the first element of the queue
T dequeue()
{
 T temp;
 temp = elements[front];
 if ( front == rear )
    front = rear = -1;
 else
    front = ( front + 1 ) % size;
 return temp;
}
}; // -------- End of CircularQueue class --------
```

8.4.2 Representing a Queue using a Linked List

A queue represented using a linked list is also known as a *linked queue*. The array-based representation of queues suffers from following limitations

- Size of the queue must be known in advance.
- We may come across situations when an attempt to enqueue an element causes overflow. However, queue, as an abstract data structure cannot be full. Hence, abstractly, it is always possible to enqueue an element in a queue. Therefore, implementing queue as an array prohibits the growth of queue beyond the finite number of elements.

The linked list representation allows a queue to grow to a limit of the computer's memory.

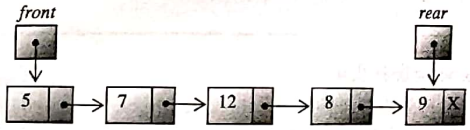
```
template <class T>
class node {
   public:
      T info;
      node *next;
};
template <class T>
class LinkQueue {
   protected:                  // data member
      node<T> *front;
      node<T> *rear;
   public:                     // member functions
      // zero argument constructor,
      // it constructs an empty queue
```



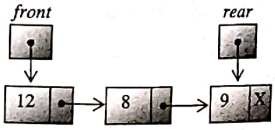
```
      LinkQueue();
      // destructor that removes queue from memory
      ~LinkQueue();
      // Check whether the stack is empty
      int isEmpty();
      // Insert an element at the end of the queue
      void enqueue(T value);
      // Access and remove the first element of the queue
      T dequeue();
      // Access the first element of the queue
      // without removing it from the queue
      T peek();
};  // -------- End of LinkQueue class --------
```

Here we have defined two data types, classes, named *node* and *Linkqueue*. The first class *node* a self-referential class, whose first element *info* hold the element of the queue and the second element *next* holds the address of the element after it in the queue. The second class is *Linkqueue*, a class, whose first element *front* holds the address of the first element of the queue, and the second element *rear* holds the address of the last element of the queue.

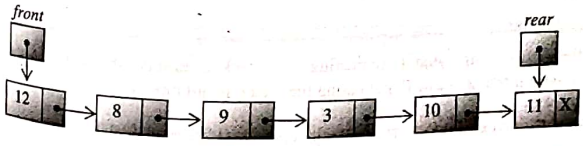
With these declarations, we will write functions for various operations to be performed on a queue represented using linked list.



(*a*) Representation of queue of Figure 8.2(*a*) in memory



(*b*) Representation of queue of Figure 8.2(*b*) in memory



(*c*) Representation of queue of Figure 8.2(*c*) in memory

Figure 8.6: Linked representation of queue of Figure 8.2

8.4.2.1 Implementation of Operations on a Linear Queue

The implementation of different operations on a queue represented using linear linked list is described in this section.

8.4.2.1.1 Creating an Empty Queue

Before we can use a queue, it is to be created/initialized. To initialize a queue, we will create an empty linked list. Setting pointer variables *front* and *rear* to NULL value creates an empty linear linked, hence an empty queue.

This simple task can be accomplished by the following constructor.

Listing 8.15:

```
// zero argument constructor,
// it constructs an empty queue
template <class T>
LinkQueue<T>::LinkQueue()
{
    front = rear = NULL;
}
```

8.4.2.1.2 Testing Queue for Underflow

The queue is tested for underflow condition by checking whether the linked list is empty. The empty status of the linked lists will be indicated by the NULL value of pointer variable *front*. This is shown in the following member function

Listing 8.16:

```
// Check whether the stack is empty
template <class T>
int LinkQueue<T>::isEmpty()
{
   if ( front == NULL )
      return 1;
   else
      return 0;
}
```

The above function returns value 1, indicating that stack is empty, if the test condition is satisfied; otherwise it returns value 0, indicating that stack is not empty.

The *isEmpty()* function can also be written using conditional operator (?:) as

Listing 8.17:

```
// Check whether the stack is empty
template <class T>
int LinkQueue<T>::isEmpty()
{
  return ( front == NULL ? 1 : 0 );
}
```


8.4.2.1.3 Testing Queue for Overflow

Since a queue represented using a linked list can grow to a limit of a computer's memory, therefore overflow condition never occurs. Hence, this operation is not implemented for linked queues.

8.4.2.1.4 Enqueue Operation

To enqueue a new element onto the queue, the element is inserted in the end of the linked list. This task is accomplished as shown in the following function

Listing 8.18:

```
// Insert an element at the end of the queue
template <class T>
void LinkQueue<T>::enqueue(T value)
{
  node<T> *ptr;
  ptr = new node<T>;
  ptr->info = value;
  ptr->next = NULL;
  if ( rear == NULL ) //  queue initially empty
    front = rear = ptr;
  else
  {
    rear->next = ptr;
    rear = ptr;
  }
}
```

8.4.2.1.5 Dequeue Operation

To dequeue/remove an element from the queue, the element is removed from the beginning of the linked list. This task is accomplished as shown in the following function

Listing 8.19:

```
// Access and remove the first element of the queue
template <class T>
T LinkQueue<T>::dequeue()
{
  T temp;
  Node<T> *ptr;
  temp = front)->info;
  ptr = front;
  if ( front == rear )          // only one element
    front = rear = NULL;
```

```
    else
       front = front->next;
    delete ptr;
    return temp;
}
```

The element in the beginning of the linked list is assigned to a local variable *temp*, which later on will be returned via the *return* statement. And the first node from the linked list is removed.

8.4.2.1.6 Accessing Front Element

The element in the front of queue is accessed as shown in the following function

Listing 8.20:

```
// Access the first element of the queue
// without removing it from the queue
template <class T>
T LinkQueue<T>::peek()
{
    return front->info;
}
```

8.4.2.1.7 Disposing a Queue

Because queue is implemented using linked lists, therefore it is programmer's job to write the code to release the memory occupied by the queue. The following listing shows the different steps to be performed.

Listing 8.21:

```
// destructor that removes queue from memory
template <class T>
LinkQueue<T>::~LinkQueue()
 {
    node<T> *ptr;
    while ( front != NULL )
    {
       ptr = front;
       front = front->next;
       delete ptr;
    }
    rear = NULL;
}
```

The complete code for *LinkQueue* class will look like

Listing 8.22:

```
// Program to demonstration the implementation of various operations
// on a queue represented using a linear linked list
#include <iostream.h>
#include <iomanip.h>
template <class T>
class node {
  public:
     T info;
     node *next;
};
template <class T>
class LinkQueue
{
  protected:              // data member
     node<T> *front;
     node<T> *rear;
  public:                 // member functions
     // zero argument constructor,
     // it constructs an empty queue
LinkQueue()
{
   front = rear = NULL;
}
     // destructor that removes queue from memory
     ~LinkQueue();
     // Check whether the stack is empty
int isEmpty()
{
  return ( front == NULL ? 1 : 0 );
}
// Insert an element at the end of the queue
void enqueue(T value)
{
  node<T> *ptr;
  ptr = new node<T>;
  ptr->info = value;
  ptr->next = NULL;
  if ( rear == NULL ) //  queue initially empty
     front = rear = ptr;
  else
  {
     rear->next = ptr;
     rear = ptr;
  }
}
// Access and remove the first element of the queue
T dequeue()
{
  T temp;
  node<T> *ptr;
```

```
   temp = front->info;
   ptr = front;
   if ( front == rear )          // only one element
      front = rear = NULL;
   else
      front = front->next;
   delete ptr;
   return temp;
}
// Access the first element of the queue
// without removing it from the queue
T peek()
{
    return front->info;
}
}; // -------- End of LinkQueue class --------
// destructor that removes queue from memory
template <class T>
LinkQueue<T>::~LinkQueue()
{
   node<T> *ptr;
   while ( front != NULL )
   {
      ptr = front;
      front = front->next;
      delete ptr;
   }
   rear = NULL;
}
void main()
{
   int choice, element;
   LinkQueue<int> intQueue;
   do
   {
      cout << "\n\nOptions available \n";
      cout << "****************** \n\n";
      cout << "   1. Enqueue\n";
      cout << "   2. Dequeue\n";
      cout << "   3. Peek\n";
      cout << "   4. Exit\n\n";
      cout << "Enter your choice ( 1-4 ) : ";
      cin >> choice;
      switch ( choice )
      {
        case 1: cout << "Enter value : " ;
                cin >> element;
                intQueue.enqueue(element);
                break;
        case 2: if ( intQueue.isEmpty() )
                   cout << "\nQueue empty...\n";
                else
                   cout << "Value dequeued is " << intQueue.dequeue();
                break;
        case 3: if ( intQueue.isEmpty() )
                   cout << "\nQueue empty...\n";
                else
```



```
                cout << "First Element of queue is "
                     << intQueue.peek();
      }
   }
   while ( choice != 4 );
} //---- end of main function ---
```

8.5 MULTIPLE QUEUES

We have seen that if we represent a queue using an array, we have to allocate sufficient space for the queue. If the space allocated to an array is less, it may result in frequent overflows and we may have to modify the code and reallocate more space for array. On other side, if we attempt to reduce the number of overflows by allocating large space for array, it may result in sheer wastage of space if the typical size of the queue is very small as compared to actual size of the array. In this case, we may say that there exists a trade-off between the number of overflows and the space.

One possibility to reduce this trade-off is to represent more than one queue in the same array of sufficient size. Figure 8.7 shows the possible mapping of five queues to same array. Each queue is allocated a same amount of space.

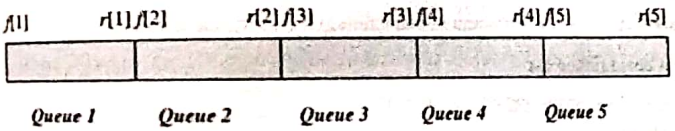


Figure 8.7: Representation of five queues by a single array

Implementations of different operations on these queues are left as an exercise for the students.

8.6 DEQUE

A *deque* is a kind of queue in which elements can be added or removed from at either end but not in the middle. The term deque is a contraction of the name *double-ended queue*.

There are two variations of a deque, which are intermediate between a deque and a queue. These are:

- ***Input-restricted deque*** – which allows insertions only at one end but allows deletions at both ends.
- ***Output-restricted deque*** – which allows insertions at both ends but allows deletions only at one end.

Deques are introduced here just for academic interest of the students their use in practical situations is almost insignificant.

8.7 PRIORITY QUEUE

A *priorities queue* is a kind of queue in which each element is assigned a priority and the order in which elements are deleted and processed comes from the following rules:

- An element with highest priority is processed first before any element of lower priority.
- Two or more elements with the same priority are processed according to the order in which they were added to the queue.

There can be different criteria's for determining the priority. Some of them are summarized below:

- A shortest job is given higher priority over the longer one.
- An important job is given the higher priority over a routine type job. For example, a transaction for on-line booking of an order is given preference over payroll processing.
- In a commercial computer center, the amount you pay for the job can determine priority for your job. Pay more to get higher priority for your job.

8.7.1 Representing a Priority Queue in Memory

There are various ways of representing (maintaining) a priority queue in memory. These are:

- Using a linear linked list
- Using multiple queues, one for each priority
- Using a heap

8.7.1.1 Linear Linked List Representation

In this representation, each node of the linked list will have three fields:

- An information field *info* that hold the element of the queue,
- A priority filed *prn* that holds the priority number of the element, and
- A next pointer filed *link* that holds the pointer to the next element of the queue.

Further, a node X precedes a node Y in the linear linked list if either node X has higher priority that Y or both nodes have same priority but X was added to the list before Y.

In a given system, we can assign higher priority to lower number or higher number. Therefore, if higher priority is for lower number, we have to maintain linear linked lists sorted on ascending order of the priority. However, if higher priority is for higher number, we have to maintain linear linked lists sorted on descending order of the priority. This way, the element to be processed next is available as the first element of the linked list.

8.7.1.2 Multiple Queue Representation

In this representation, one queue is maintained for each priority number. In order to process an element of the priority queue, element from the first non-empty highest priority number queue is accessed. In order to add a new element to the priority queue, the element is inserted in an appropriate queue for given priority number.

8.7.1.3 Heap Representation of a Priority Queue

A *heap* is a complete binary tree and with the additional property – the root element is either smallest or largest from its children. If root element of the heap is smallest from its children, it is known as *min heap*. If root element of the heap is largest from its children, it is known as *max heap*.

A priority queue having highest priority for lower number can be represented using min heap, and a priority queue having highest priority for higher number can be represented using max heap.

Hence, the element of the priority queue to be processed next is in the root node of the heap. Heaps are discussed in detail in *Chapter 10*.

If we compare the effort in adding or removing elements from the priority queue, we find that heap representation is the best.

8.8 APPLICATIONS OF QUEUES

Some of the applications of queues are listed below:

- There are several algorithms that use queues to solve problems efficiently. For example, we have used queue for performing level-order traversal of a binary tree in *Chapter 9*, and for performing breadth-first search on a graph in *Chapter 11*. Also in most of the simulation-related problems, queues are heavily used.
- When the jobs are submitted to a networked printer, they are arranged in order of arrival. Thus, essentially, jobs sent to a printer are placed on a queue.
- There is a kind of computer network where disk is attached to one computer, known as *file server*. Users on other computers are given access to files on a first-come first-served basis, so the data structure is a queue.
- Virtually every real-life line (supposed to be) a queue. For example, lines at ticket counters at cinema halls, railway stations, bus stands, etc., are queues, because the service, *i.e.*, ticket, is provided on first-come first-served basis.

REVIEW EXERCISES

1. What is a queue? Is queue a linear or non-linear data structure?
2. Queue is a special type of list? Comment.
3. How does an array based queue compares with linked queue?
4. Why an operation to check queue overflow is not implemented on linked queues?
5. Can we implement a queue in C whose elements can be of different type? If your answer is yes, then show how?
6. How *peek()* operation differs from *dequeue()* operation?
7. While representing a queue using linked lists, we have implemented additional operation *dispose()* to remove the queue from memory. Why such operation is not implemented for queue represented using an array.
8. How does a linear queue compare with circular queue?

PROGRAMMING EXERCISES

1. Write a menu driven program to implementing the various operations on a linear queue.
2. Write a menu driven program to implementing the various operations on a circular queue.
3. Write a menu driven program to implementing the various operations on a queue represented using a linked list.
4. Design a data structure to represent m circular queues in an array named *elements* of size n. Also write functions for *enqueue()* and *dequeue()* operations to insert and delete elements from queue i, $0 \le i \le m-1$. These operations should run in O(1) time.
5. Using the data structure designed in question 4, write functions for *isFull()* and *isEmpty()* operations to check overflow and underflow of queue i, $0 \le i \le m-1$. These operations should also run in O(1) time.
6. **Kitty Fishing - A Simple Game of Cards:** There is a simple two-player card game called 'Kitty fishing'. When the game begins player A and B has the same number of cards. Then each gives out one card in turn. Each card given out on the table should be laid overlapped one by one. When the card newly given out finds a card which has the same value on the table, the player who gives out the card will take the cards between the two same cards following the order the cards on the table, and put them to the back of his cards. The same player continues to give out net cards. Player giving a card is called a turn.

   Note: Do not change the order of your cards.

Chapter 9

Trees

9.1 INTRODUCTION

So far in the text we have discussed data structures such as arrays, linked lists, stacks and queues for representing linear and tabular data. These structures in general are not suitable for representing hierarchical data. In hierarchical data we have ancestor-descendant, superior-subordinate, whole-part, or similar relationship among data elements. The following examples will further illustrate the situations make such hierarchical data. In hierarchical data we have ancestor-descendant, superior-subordinate, whole-part, or similar relationship among data elements.

Example 9.1: *Family Structure*

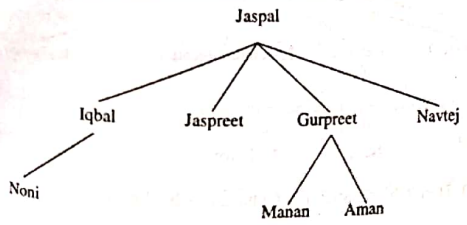


Figure 9.1: Hierarchical structure showing Jaspal's descendant

Figure 9.1 shows descendants of Jaspal arranged in hierarchical manner, beginning with Jaspal at the top of the hierarchy. Jaspal's children (Iqbal, Jaspreet, Gurpreet, and Navtej) are shown next in the hierarchy, and a line or an edge joins Jaspal and his children. Iqbal has one child,

Jaspreet has none, Gurpreet has two, and Navtej has none. Iqbal's only child is shown below him, and Gurpreet's children are shown below him. Note that there is an edge between each parent and his/her child(ren). From this hierarchical representation it is easy to identify Iqbal's siblings, Jaspal's descendants, Noni's ancestors, and so on.

Example 9.2: *Business Corporate Structure*

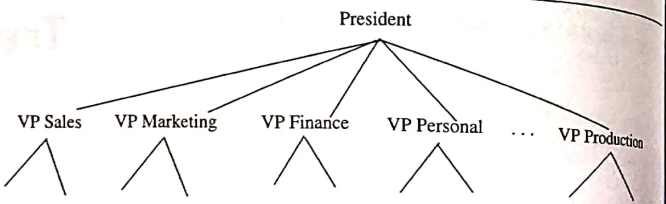


Figure 9.2: Hierarchical administrative structure of a typical business organization

Figure 9.2 shows administrative setup of a typical business organization. The person highest in the hierarchy (president in this case) appears at the top. Those who are next in the hierarchy (vice presidents (VPs) in this case) are shown next in the hierarchy below the president. The vice presidents are the president's immediate subordinates, and the present is their superior. Each vice president, in turn, has his/her subordinates who may themselves have subordinates.

Example 9.3: *Federal Government Structure*

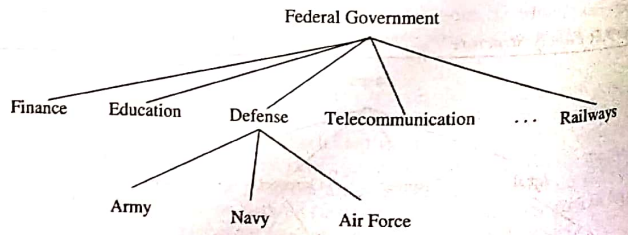


Figure 9.3: Hierarchical structure of a typical federal government

Figure 9.3 shows hierarchical subdivisions of a federal government. At the top, we have entire federal government. At the next level; we have its major subdivisions, *i.e.*, different departments. Each subdivision may have further subdivisions, which will be shown, at the next level. For example, Department of Defense has been subdivided into Army, Navy, and Air Force.

Example 9.4: *Modular Structure of a Computer Program*

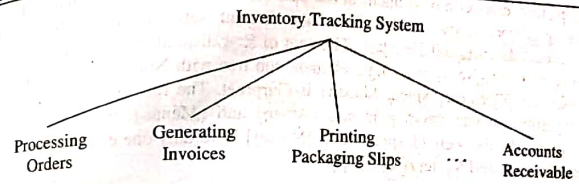


Figure 9.4: Module hierarchy for inventory tracking system

For another example of hierarchical data, consider the popular software development technique referred to as *modularization*. The objective of modularization is to divide the large and complex software system into functionally independent modules (parts) so that each can be developed independently. If necessary, each module may be further divided. It reduces the overall software develop time, as it is much easier to solve several problems than one large one. In addition, different programmers can develop different modules at the same time.

Figure 9.4 shows possible modular decomposition of computer based inventory-tracking system. The inventory tracking system is divided into several modules. Only four are shown in the figure. The first module is used to book the orders. If the orders can be met, then the second module is used to generate the invoice for the booked order. After generating the invoice, the third module is used to print the packaging slip, which will be sent to the godown for packing the goods ordered, and so on. The fourth module is used when the payments from the customers are received.

A tree is an ideal data structure for representing such kind of hierarchical data. As there are many types of trees in a forest, likewise there are many types of trees in data structures — *(general) trees*, *binary trees*, *expression trees*, *tournament trees*, *binary search trees*, *threaded trees*, *AVL trees* and *B-trees*.

In this chapter, we will introduce these trees while laying major stress on *binary search trees*.

9.2 TREE DEFINED

A *(general) tree* T is a finite nonempty set of elements. One of these elements is called the root, and the remaining elements, if any, are partitioned into trees, which are called the *sub trees* of T.

Let us see how this definition relates to our examples of hierarchical data. The element at the highest level of the hierarchy is root. The elements at the next level are the roots of the sub trees formed by partitioning of the remaining elements.

Consider Example 9.1. The data set is {Jaspal, Iqbal, Jaspreet, Navtej, Gurpreet, Noni, Aman, Manan}. Therefore, number n of element in the data set = 8. The root of the data set is Jaspal. The remaining elements are divided into four disjoint sets {Iqbal, Noni}, {Jaspreet}, {Gurpreet, Aman, Manan}, and {Navtej}. The root of first disjoint set {Iqbal, Noni} is Iqbal. The remaining element {Noni} is a single element sub tree with Noni as its root. The root of second disjoint set {Gurpreet, Aman, Manan} is Gurpreet. The remaining elements {Aman, Manan} are partitioned into the disjoint sets {Aman} and {Manan}, which are both single element sub trees. The data sets {Jaspreet} and {Navtej} have only one element, and therefore their roots are Jaspreet and Navtej respectively.

9.3 TREE TERMINOLOGY

Level of an Element – Very commonly used tree term is level. By definition, the root of the tree is at level 1; its children, if any, are at level 2; their children, if any, are at level 3; and so on. In Figure 9.1, Jaspal is at level 1; Iqbal, Jaspreet, Gurpreet, and Navtej are at level 2; Noni, Aman, and Manan are at level 3.

Degree of an Element – The degree of an element is the number of children it has. The degree of a leaf is zero. In Figure 9.1, Jaspal has degree 4.

Degree of a Tree – The degree of a tree is the maximum of its element degrees. The degree of tree in Figure 9.1 is 4.

Height/Depth of a Tree – If level of the root is denoted by 1, then the maximum level number of the tree is known as its *height* or *depth*.

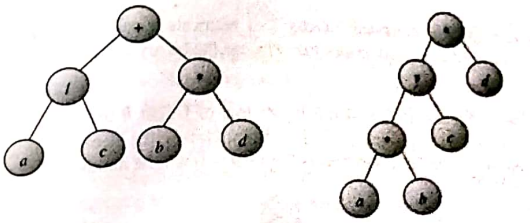
9.4 BINARY TREES

A *binary tree T* is either empty or has a finite collections of elements. When the binary tree is not empty, one of its elements is called the *root* and the remaining elements, if any, are partitioned into two binary trees, which are called the left & right sub trees of *T*.

The essential differences between a binary tree and a tree are

- A binary tree can be empty whereas a tree cannot.
- Each element in binary tree has at most two sub trees (one or both of these sub trees may be empty). Each element in a tree can have any number of sub trees.
- The sub trees of each element in a binary tree are ordered. That is, we distinguish between the left and right sub trees. The sub trees in a tree are unordered.

Like a tree, a binary tree is drawn with its root at the top. The elements in the left (right) sub tree of the root are drawn below and to the left (right) of the root. Between each element and its children is a line or edge.



(a) $(a/c)+(b*d)$ (b) $(((a*b)+c)*d)$

Figure 9.5: Expression trees

Figure 9.5 shows some binary trees that represent arithmetic expressions. Each operator ($+, -, *, /$) may have one or two operands. The left operand, if any, is the left sub tree of the operator and the right operand is the right sub tree. The leaf elements in an expression tree are either constants or variables. Note that an expression tree contains no parentheses.

9.4.1 Some Properties of Binary Trees

Property 1:

A binary tree with n elements, $n > 0$, has exactly $n-1$ edges.

Proof: Every element in a binary tree, except the root, has exactly one parent. There is exactly one edge between each child and its parent. So the number of edges is $n-1$.

Property 2:

A binary tree of height h, $h > 0$, has at least h and at most 2^h-1 elements in it.

Proof: Since there must be at least one element at each level, the number of elements is at least h. As each element can have at most two children, the number of elements at level k is at most 2^{k-1} for $k > 0$. For $h = 0$, the total number of elements is 0, which equals 2^0-1. For $h > 0$, the total number of elements cannot exceed $\sum_{k=1}^{h} 2^{k-1} = 2^h - 1$.

Property 3:

The height of the binary tree that contains n elements, $n \geq 0$, is at most n and at least $\lceil \log_2(n+1) \rceil$.

Proof: Since there must be at least one element at each level, the height cannot exceed n. From property 2, we know that a binary tree of height h can have no more than 2^h-1 elements. So $n \leq 2^h-1$. Hence $h \geq \log_2(n+1)$. Since h is an integer, we get $h \geq \lceil \log_2(n+1) \rceil$.

A binary tree of height h that contains exactly 2^h-1 elements is called a *full binary tree*. The binary tree of Figure 9.5(a) is a full binary tree whereas the binary tree of Figure 9.5(b) is not a full binary tree.

Suppose we number the elements in a full binary tree of height h using numbers 1 through 2^h-1. We begin at level 1 and go down to level h. Within levels, the elements are numbered left to right. The elements of the full binary tree of Figure 9.5(a) will be numbered as shown below.

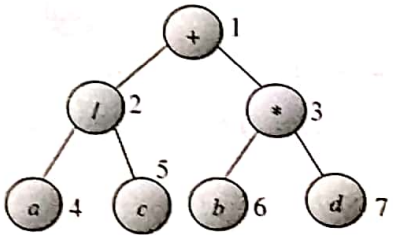


Figure 9.6: Full binary tree of height 3

Now suppose we delete the k, $k \geq 0$, elements numbered 2^h-i, $1 \leq i \leq k$ for any k. The resulting binary tree is called a *complete binary tree*. Figure 9.7 shows some examples of complete trees.

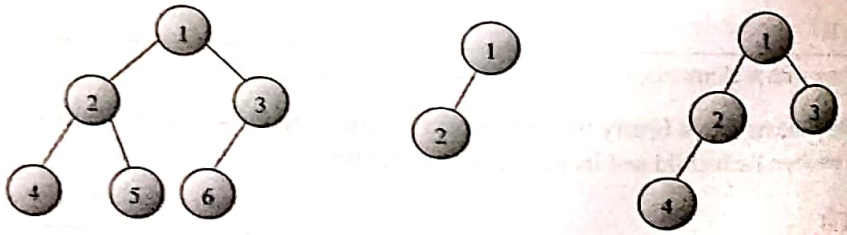


Figure 9.7: Complete Binary trees

Property 4:

Let i, $1 \leq i \leq n$, be the number assigned to an element of a complete binary tree. The following statements are true.

1. If $i = 1$, then this element is the root of the binary tree. If $i > 1$, then the parent of this element has been assigned the number $\lfloor i/2 \rfloor$.
2. If $2i > n$, then this element has no left child. Otherwise, its left child has been assigned the number $2i$.
3. If $2i + 1 > n$, then this element has no right child. Otherwise, its right child has been assigned the number $2i + 1$.

Its proof can be established by induction on i. This is left as an exercise for the readers.

9.5 TOURNAMENT TREES

Suppose n players enter in a chess tournament and further suppose that the tournament is played on knock-out basis. By knock-out basis means a player is eliminated from the tournament upon losing a match. Pairs of players play each other until only one remains undefeated. The undefeated player is declared winner of the tournament. Figure 9.8 shows a possible tournament involving eight players named a through h. The tournament is described by a binary tree in which each external node (leaf node) represents a player and each internal node represents a match played between players designated by the children of the node.

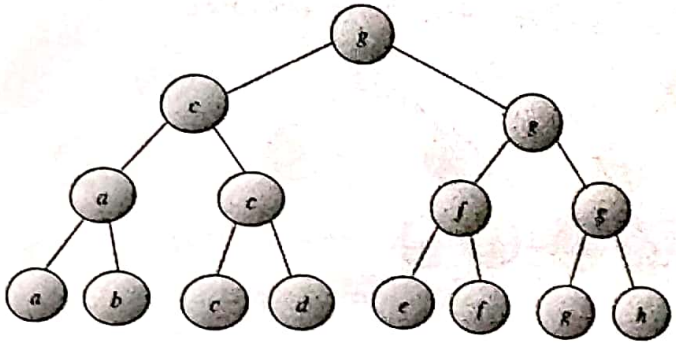


Figure 9.8: Eight Player Tournament tree

Each level of internal nodes defines a *round* of matches that can be played in parallel. In the first round player a and b, c and d, e and f, g and h play. The winner of each match is recorded at the internal node that represents the match. In our example, the four winners are a, c, f and g. In the next round of matches, players a and c, f and g play against each other. The winners are c and g. Now in the final round, players c and g play each other, and player g emerges winner of the final match. Hence player g is the winner of the tournament.

Figure 9.9 show a possible tournament that is played between 5 players. Players c, d and e are given byes in the first round. The winner in this case is player e.

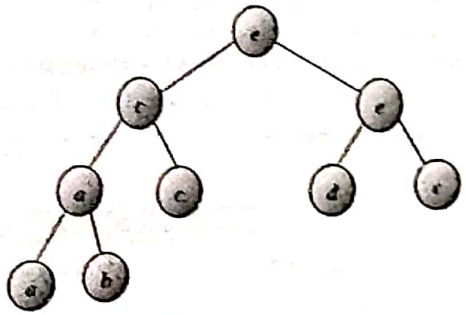


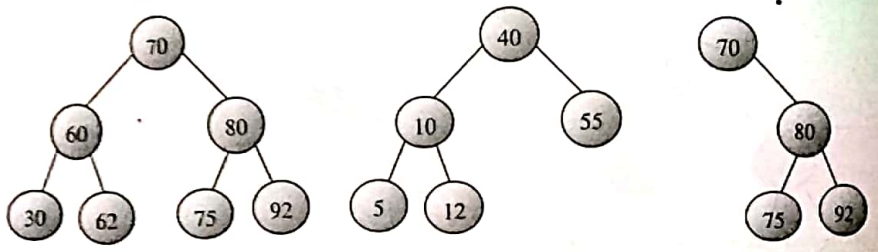
Figure 9.9: 5 Player Tournament tree

The tournament trees of Figure 9.8 and 9.9 are called *winner trees* because at each internal node, we record the winner of the match played at that node. In contrast, if we record the loser of the match played at that node, the tournament tree is called a *loser tree*. Tournament trees are also called *selection trees*. Winner trees can be adapted for computer use.

9.6 BINARY SEARCH TREES

A *binary search tree* T is a binary tree that may be empty. A non-empty binary search tree satisfies the following properties:

1. Every element has a key (or value) and no two elements have the same key i.e. all keys are unique.
2. The keys, if any, in the left sub tree of the root are smaller than the key in the node.
3. The keys, if any, in the right sub tree of the root are larger than the key in the node.
4. The left and right sub trees of the root are also binary search trees.



(*a*) (*b*) (*c*)

Figure 9.10: Binary Search trees

9.6.1 Representation of a Binary and Binary Search Tree

A binary tree and a binary search tree is represented in an identical manner. These can be represented using a linear array (array representation) or linked lists (linked representation). We will discuss these representations one by one and their merits and demerits.

9.6.1.1 Array Representation

The array representation utilizes *Property 4*. In this representation, the binary (search) tree is represented by storing each element at the array position corresponding to the number assigned to each number (node). In this representation, a binary (search) tree of height h requires an array of size $(2^h - 1)$, in the worst case. For simplicity, we consider that array is indexed with an index set beginning from 1 not from 0.

1	2	3	4	5	6	7
70	60	80	30	62	75	92

(*a*) Array representation of binary search trees of Figure 9.10(*a*)

1	2	3	4	5	6	7
40	10	55	5	12		

(*b*) Array representation of binary search trees of Figure 9.10(*b*)

1	2	3	4	5	6	7
70		80			75	92

(*c*) Array representation of binary search trees of Figure 9.10(*c*)

Figure 9.11: Array representation of binary search trees of Figure 9.10

Can you imagine how the following binary search tree will stored in array representation?

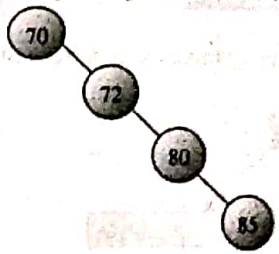


Figure 9.12: A right-skewed binary search tree

The array representation of binary search tree of Figure 9.12 will be as shown in Figure 9.13.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
70		72				80								85

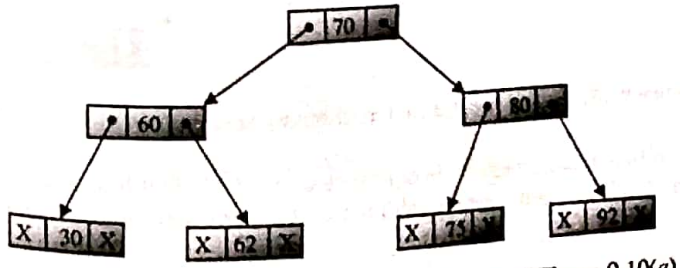
Figure 9.13: Array representation of binary search tree of Figure 9.12

This scheme of representation is quite wasteful of space when the binary (search) tree is skewed or number of elements is small as compared to its height.

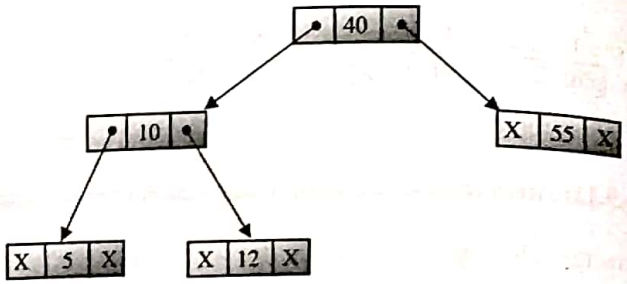
Hence, array representation is useful only when the binary (search) tree is full or complete.

9.6.1.2 Linked Representation

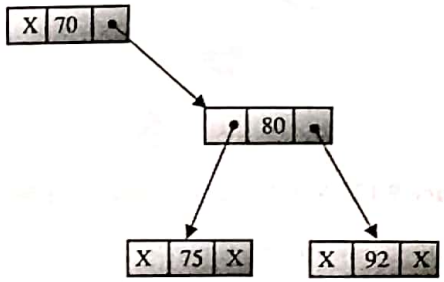
The most popular and practical way of representing a binary (search) tree is using links (pointers).



(*a*) Linked representation of binary search trees of Figure 9.10(*a*)



(*b*) Linked representation of binary search trees of Figure 9.10(*b*)



(*c*) Linked representation of binary search trees of Figure 9.10(*c*)

Figure 9.14: Linked representation of binary search trees of Figure 9.10

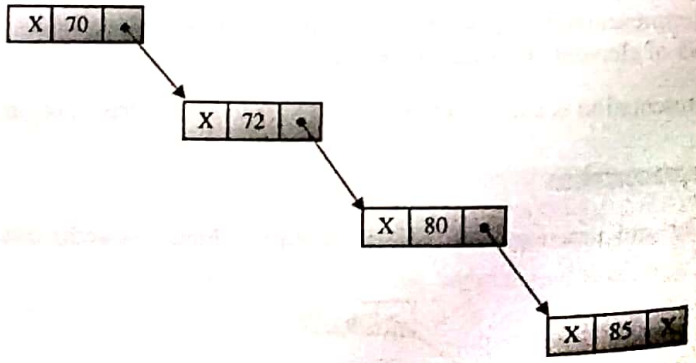


Figure 9.15: Linked representation of binary search trees of Figure 9.12

In linked representation, each element is represented by a node that has exactly two link fields. Let us call these fields *left* and *right*. In addition to these two link fields, each node has a data field called *info*.

An arrow represents each edge in the drawing of a binary tree from the parent node to the child node. This pointer is placed in the appropriate link field of the parent node. Since an n-element binary tree has exactly $n-1$ edges, there will be $2n-(n-1) = n+1$ link fields that have NULL values (pictured as X).

Suppose that the elements of the binary can be any primitive type or pointer type, the following is the required ADT to represent a binary tree:

```
// class definition to represent a node of a binary search tree
template <class T>
class BSTNode
{
   public:
      BSTNode *left;
      T info;
      BSTNode *right;
};
// class definition for binary search tree
template <class T>
class BSTree
{
   private:
      BSTNode<T> *root;
   public:
      // constructor to initialize binary search tree
      // to empty state
      BSTree() {
         root = NULL;
      }

      // destructor, that calles recursive function deleteTree()
      // to remove binary search tree from memory
      ~BSTree() {
          deleteTree(&root);
      }

      // member function to insert an element in binary search
      // tree, it calls recursive function insert() to
      // actually perform the insertion
      void insertElement(T);
      // member function to perform preorder traversal of binary
      // search tree, it calls recursive function
      // traversePreorder() to perform the actual traversal
      void traversePre();
      // member function to perform inorder traversal of binary
      // search tree, it calls recursive function
      // traverseInorder() to perform the actual traversal
      void traverseIn();
      // member function to perform postorder traversal of binary
      // search tree, it calls recursive function
      // traversePostorder() to perform the actual traversal
      void traversePost();
      // member function to perform the levelorder traversal of
      // binary search tree, it use linked queue
      void levelorderTraversal();
```

```
// member function to search a given element, it calls
// recursive function search() to actually perform the
// search
BSTNode<T> *searchElement(T);
// member function to delete a given element from binary
// search tree, it calls recursive function deleteNode()
// to actually delete the element from binary search tree
void deleteElement(int);
// member function that returns the location of the largest
// element of the binary search tree, it calls recursive
// function findMaximum() to find the largest element
BSTNode<T> *findLargest();
// member function that returns the location of the smallest
// element of the binary search tree, it calls recursive
// function findMinimum() to find the smallest element
BSTNode<T> *findSmallest();
// member function that returns the internal (non-leaf)
// nodes of the binary search tree, it calls recursive
// function internalNodes() to actually count the
// internal nodes
int countInternalNodes();
// member function that returns the external (leaf) nodes
// of the binary search tree, it calls recursive
// function externalNodes() to actually count the
// external nodes
int countExternalNodes();
// member function that returns the total number of nodes
// of the binary search tree, it calls recursive function
// totalNodes() to actually count the total nodes
int countTotalNodes();
// member function that returns the height of the binary
// search tree, it calls recursive function findHeight()
// to actually find the height of the binary search tree
int heightTree();
// member function to find the mirror image of the binary
// search tree, it calls recursive function reflect()
// to actually find the image
void mirrorImage();

// ------ RECURSIVE MEMBER FUNCTIONS -------

// recursive member function to insert an element into
// binary search tree
static void insert(BSTNode<T> **, T);
// recursive member to perform preorder traversal of
// binary search tree
static void traversePreorder(BSTNode<T> *);
// recursive member to perform inorder traversal of
// binary search tree
static void traverseInorder(BSTNode<T> *);
// recursive member to perform postorder traversal of
// binary search tree
static void traversePostorder(BSTNode<T> *);
// recursive member function to search a given element,
// it return the location of the element if present,
// else returns NULL indicating that the element is
// not present in binary search tree
static BSTNode<T> *search(BSTNode<T> *, T);
```



```
// recursive member function that returns the location of
// the largest element of the binary search tree.
// It returns NULL if tree is empty a given element or
// it has no right child
static BSTNode<T> *findMaximum(BSTNode<T> *);
// recursive member function that returns the location
// of the smallest element of the binary search tree.
// It returns NULL if tree is empty a given element or
// it has no left child
static BSTNode<T> *findMinimum(BSTNode<T> *);
// recursive member function that delete an element from
// binary search tree. It calls recursive member function to
// search the element to be deleted
static void deleteEle (BSTNode<T> **, T) {
// recursive member function to remove the binary search
// tree from memory
static void deleteTree(BSTNode<T> **);
// recursive member function that returns the number of
// internal (non-leaf) nodes of the binary search tree
static int internalNodes(BSTNode<T> *);
// recursive member function that returns the number of
// external (leaf) nodes in a binary search tree
static int externalNodes(BSTNode<T> *);
// recursive member function that returns the total
// number of nodes in a binary search tree
static int totalNodes(BSTNode<T> *);
// recursive member function that returns the height of the
// binary search tree
static void findHeight(BSTNode<T> *, int *);
// recursive member function that reflects (mirror image)
// a binary search tree about an axis passing through
// its root
static void reflect(BSTNode<T> *);
}; // -------- End of BSTree class --------
```

We have defined our own data type named *BSTree*, which is class and whose first data member is a pointer variable *left* will be used to hold the address of the left child (left subtree), second data member is *info* that can be any primitive type, will be used to hold the data of the node, and third data member is a pointer variable *right* will be used to hold the address of the right child (right subtree).

9.6.2 Common Operations on Binary and Binary Search Trees

Some of the operations that are commonly performed on binary as well as binary search trees are:

1. Create an empty tree.
2. Traverse it.
3. Determine its height.
4. Determine the number of elements in it.
5. Determine the number of internal nodes i.e. non-leaf nodes.
6. Determine the number of external nodes i.e. leaf nodes.
7. Determine its mirror image.
8. Remove it from memory.

9.6.2.2.1 Pre-Order Traversal of a Binary (Search) Tree

The listing on the next page shows the recursive implementation of pre-order traversal algorithm.

Listing 9.2:

```
// member function to perform preorder traversal of binary
// search tree, it calls recursive function
// traversePreorder() to perform the actual traversal
template <class T>
void BSTree<T> :: traversePre()
{
   traversePreorder(root);
}

// recursive member to perform preorder traversal of
// binary search tree
template <class T>
void BSTree<T> :: traversePreorder ( BSTNode<T> *h )
{
   if ( h != NULL ) {
      cout << h->info << "  ";
      traversePreorder( h->left );
      traversePreorder( h->right );
   }
}
```

As you can see, the recursive implementation is more elegant and preferred one. However, we can also perform pre-order traversal using iterative procedure. The iterative implementation requires the explicit use of stack whose elements will be pointers of type *BSTNode*.

The following algorithm lists the various steps to be performed.

Algorithm 9.1:

PreorderTraversalIterative(*root*)

Here root is a pointer variable that holds the address of the root node of the given tree. It uses a stack *s* whose elements are pointers to type *BST*. It uses a temporary pointer variable *ptr* to hold the address of current node.

```
Begin
  CreateEmptyStack(s)          // call procedure to create an empty stack
  Push( s, NULL )              // push NULL value as sentinel value
  Set ptr = root
  while ( ptr ≠ NULL ) do
     Print: ptr->info
     if ( ptr->right ≠ NULL ) then
        Push( s, ptr->right )
     endif
     if ( ptr->left ≠ NULL ) then
        Set ptr = ptr->left
     else
        Set ptr = Pop(s)       // pop value from stack and assign to ptr
     endif
  endwhile
End.
```


The implementation of the algorithm 9.1 is left as an exercise for the readers.

9.6.2.2.2 In-Order Traversal of a Binary (Search) Tree

The following listings show the recursive implementation of in-order traversal algorithm.

Listing 9.3:

```
// member function to perform inorder traversal of binary
// search tree, it calls recursive function
// traverseInorder() to perform the actual traversal
template <class T>
void BSTree<T> :: traverseIn()
{
   traverseInorder(root);
}

// recursive member to perform inorder traversal of
// binary search tree
template <class T>
void BSTree<T> :: traverseInorder ( BSTNode<T> *h )
{
   if ( h != NULL ) {
      traverseInorder( h->left );
      cout << h->info << "  ";
      traverseInorder( h->right );
   }
}
```

Various steps required for iterative implementation of in-order traversal are shown in the following algorithm.

Algorithm 9.2:

InorderTraversalIterative(root)

Here root is a pointer variable that holds the address of the root node of the given tree. It uses a stack *s* whose elements are pointers to type *BST*. It uses a temporary pointer variable *ptr* to hold the address of current node. It also uses a flag variable *done* to control the iteration.

```
Begin
  CreateEmptyStack(s)          // call procedure to create an empty stack
  Push( s, NULL )              // push NULL value as sentinel value
  Set ptr = root
```

```
Set done = false
while ( not done ) do
   while ( ptr ≠ NULL ) do
      Push( s, ptr )
      Set ptr = ptr->left
   endwhile
   Set ptr = Pop(s)    // pop up value from stack and assign to ptr
   Set flag = true
   while ( ( ptr ≠ NULL ) and ( flag = true ) ) do
      Print: ptr->info
      if ( ptr->right ≠ NULL ) then
          Set ptr = ptr->right
          Set flag = false
      else
          Set ptr = Pop(s)
      endif
   endwhile
   if ( ptr  = NULL ) then
      Set done = true
   endif
endwhile
End.
```

The implementation of the above algorithm is also left as an exercise for the readers.

9.6.2.2.3 Post-Order Traversal of a Binary (Search) Tree

The following listing shows the recursive implementation of post-order traversal algorithm.

Listing 9.4:

```
// member function to perform postorder traversal of binary
// search tree, it calls recursive function
// traversePostorder() to perform the actual traversal
template <class T>
void BSTree<T> :: traversePost()
{
   traversePostorder(root);
}
// recursive member to perform postorder traversal of
// binary search tree
template <class T>
void BSTree<T> :: traversePostorder ( BSTNode<T> *h )
{
   if ( h != NULL ) {
      traversePostorder( h->left );
      traversePostorder( h->right );
      cout << h->info << "  ";
   }
}
```

The iterative implementation of post-order traversal is bit complex as compared to pre-order and in-order traversals.

9.6.2.2.4 Level-Order Traversal of a Binary (Search) Tree

The following listing shows the implementation of level-order traversal algorithm.

Listing 9.5:

```
// member function to perform the levelorder traversal of binary
// search tree, it use linked queue
template <class T>
void BSTree<T>::levelorderTraversal()
{
   LinkQueue<BSTNode<T>*> q;
   BSTNode<T> *ptr;
   q.enqueue(root);
   while( ! q.isEmpty()  ) {
      ptr = q.dequeue();
      if ( ptr->left != NULL )
          q.enqueue(ptr->left);
      if ( ptr->right != NULL )
          q.enqueue(ptr->right);
      cout << ptr->item << "  ";
   }
}
```

In the above listing we have considered a queue named *q* represented using linked list whose implementation is given in *Chapter 7*. The elements of the queue *q* will be pointers to type *BSTNode<T>*.

Binary tree has one very interesting property – *property is that if we are given in-order traversal and either of pre-order or post-order traversals, we can construct the binary tree from these traversals.*

Example 9.6:

Consider that a binary tree T has 8 nodes. The pre-order and in-order traversals of binary tree T yield the following sequence of nodes:

Pre-order : A B D E G H C F

In-order : D B G E H A C F

Draw the binary tree T.

Solution: The binary tree T is drawn as follows:

The first node in the pre-order traversal is always a root node of the binary tree. Thus the root node of the binary tree T is node A.

After determining the root node of the binary tree, our aim is to find the nodes that comprise the left sub tree and right sub tree of node A. We know from the in-order traversal that first left sub tree is traversed, then root node, and finally the right sub tree. Thus all the nodes to the left

of node A in in-order traversal comprise its left sub tree. Similarly, all the nodes to the right of node A in in-order traversal comprise its right sub tree.

The above procedure partitions the pre-order and in-order traversal sequences as shown below:

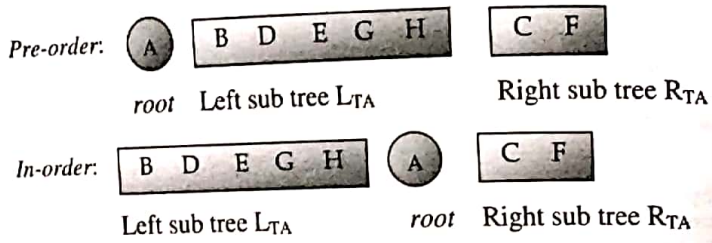


Figure 9.18 shows the partial binary tree constructed so far.

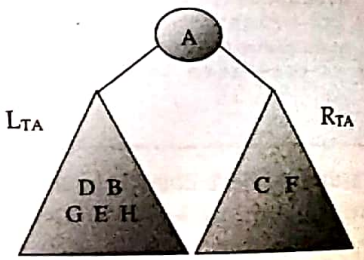


Figure 9.18

In the same way, we can construct the left sub tree L_{TA} and right sub tree R_{TB}. Let us first consider the left sub tree L_{TA}. The pre-order and in-order traversal sequences of the left sub tree L_{TA} are

Pre-order: B D E G H
In-order: D B G E H

From these traversal sequences we find that the root of the sub tree is node B, and its left sub tree consists of a single node D and the right sub tree consists of nodes G, E, and H.

The above procedure partitions the pre-order and in-order sequences as shown below:

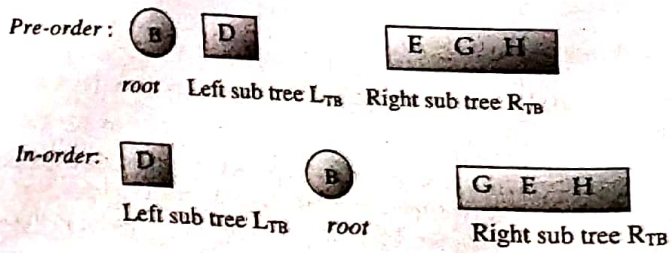


Figure 9.19 shows the partial binary tree constructed so far.

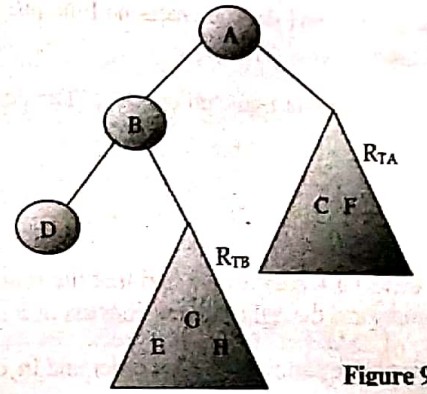


Figure 9.19

Now consider the right sub tree R_{TB}. The pre-order and in-order traversal sequences of R_{TB} are

Pre-order: E G H
In-order: G E H

From these traversal sequences we find that the root of the sub tree R_{TB} is node E, and its left sub tree consists of a single node G and the right sub tree also consists of a single node G.

The above procedure partitions the pre-order and in-order traversal sequences of R_{TB} as shown below:

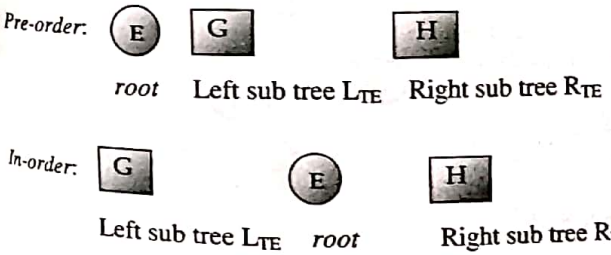


Figure 9.20 shows the partial binary tree constructed so far.

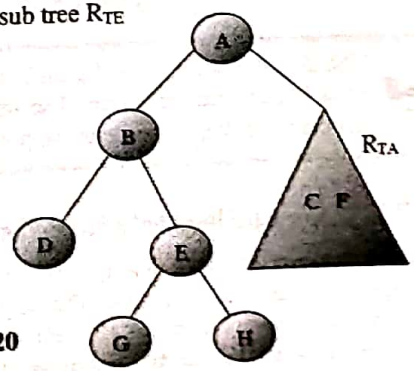


Figure 9.20

Thus we have completed the construction of the left sub tree L_{TA} of node A, which was the

node of the binary tree.

Now we are left with the right sub tree R_{TA}. The pre-order and in-order traversal sequences

R_{TA} are

Pre-order: C F

In-order: C F

From these traversal sequences we find that the root of the sub tree R_{TA} is node C, and its

sub tree is empty and the right sub tree consists of a single node F.

The above procedure partitions the pre-order and in-order traversal sequences of R_{TA} as shown

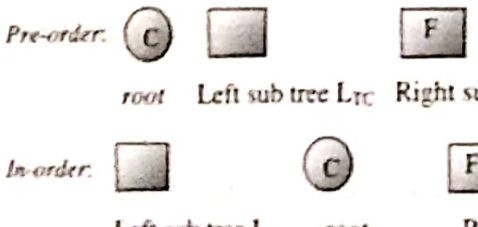


Figure 9.21 shows the binary tree after this step, and this is the final binary tree.

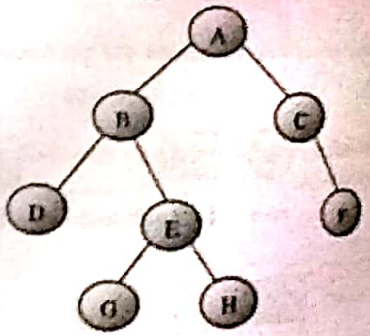


Figure 9.21

The above procedure can be easily modified if the in-order and the post-order traversal of a binary tree are given. The only difference is that the root of the tree will be appearing as the last node in the post-order traversal.

9.6.2.3 Determining Height of a Binary (Search) Tree

In order to determine the height of a binary (search) tree, we find the height of the left sub tree and right sub tree of a given node. The height of the binary (search) tree at a given node will be equal to maximum height of the left and right sub trees plus 1. Listing 9.6 shows the implementation of the various steps required.

Listing 9.6:

```
// member function that returns the height of the binary
// search tree, it calls recursive function findHeight()
// to actually find the height of the binary search tree
template <class T>
int BSTree<T>:: heightTree()
{
  int h;
  findHeight(root, &h);
  return h;
}
// recursive member function that returns the height of the
// binary search tree
template <class T>
void BSTree<T>:: findHeight ( BSTNode<T> *tree, int *h )
{
  int hl, hr;
  if ( tree == NULL ) {
    *h = 0;
  } else {
    findHeight ( tree->left, &hl );
    findHeight ( tree->right, &hr );
    if ( hl > hr )
    *h = hl + 1;
    else
    *h = hr + 1;
  }
}
```

In the *findHeight()* function, the height of the binary (search) tree is returned via the formal argument *h* which is a pointer to *int* type. The following listing shows another implementation to find the height of the tree.

Listing 9.7:

```
// member function that returns the height of the binary
// search tree, it calls recursive function findHeight()
// to actually find the height of the binary search tree
template <class T>
int BSTree<T>:: heightTree()
{
  return findHeight(root);
}
// recursive member function that returns the height of the
// binary search tree
template <class T>
int BSTree<T>:: findHeight ( BSTNode<T> *tree)
{
  int leftHeight, rightHeight;
  if ( tree == NULL )
    return 0;
  else
  {
    leftHeight = findHeight( tree->left );
```

```
    rightHeight = findHeight( tree->right );
    if ( leftHeight > rightHeight )
       return ++leftHeight;
    else
       return ++rightHeight;
  }
}
```

9.6.2.4 Determining Number of Nodes/Elements

Any of the traversal schemes can be used to determine the number of elements in a binary (search) tree. However, we will use a different approach. Note that the number of elements/nodes in a binary (search) tree is equal to the number of nodes in the left sub tree plus number of nodes in the right sub tree of a given node plus 1. Note that if the binary (search) tree is empty, then the number of nodes is equal to 0.

Listing 9.8 shows the implementation of the various steps required for this approach.

Listing 9.8:

```
// member function that returns the total number of nodes
// of the binary search tree, it calls recursive function
// totalNodes() to actually count the total nodes
template <class T>
int BSTree<T> :: countTotalNodes()
{
   return totalNodes(root);
}

// recursive member function that returns the total
// number of nodes in a binary search tree
template <class T>
int BSTree<T> :: totalNodes( BSTNode<T> *tree )
{
   if ( tree == NULL )
      return 0;
   else
      return (totalNodes(tree->left)+totalNodes(tree->right)+1);
}
```

9.6.2.5 Determining Number of Internal/Non-leaf Nodes

The number of internal/non-leaf nodes in a binary (search) tree is equal to the number of internal/non-leaf nodes in the left sub tree plus number of non-leaf nodes in the right sub tree of a given node plus 1. Note that if the binary (search) tree is empty or there is only one node then the number of internal/non-leaf nodes is equal to 0.

Listing 9.9 shows the implementation of the various steps required.

Listing 9.9:

```
// member function that returns the internal (non-leaf)
// nodes of the binary search tree, it calls recursive
// function internalNodes() to actually count the internal nodes
template <class T>
int BSTree<T> :: countInternalNodes() {
  return internalNodes(root);
}

// recursive member function that returns the number of
// internal (non-leaf) nodes of the binary search tree
template <class T>
int BSTree<T> :: internalNodes( BSTNode<T> *tree ) {
  if ( (tree == NULL) || ((tree->left == NULL) &&
       (tree->right == NULL) ) )
    return 0;
  else
    return ( internalNodes( tree->left ) +
             internalNodes( tree->right ) + 1);
}
```

9.6.2.6 Determining Number of External/Leaf Nodes

The number of external/leaf nodes in a tree is equal to the sum of the external/leaf nodes in the left sub tree and the external/leaf nodes in the right sub tree of a given node. Note that if the binary (search) tree is empty then the number of external/non-leaf nodes is 0 and if there is only one node in the binary (search) tree empty then the number of external/leaf nodes is equal to 1.

Listing 9.10 shows the implementation of the various steps required.

Listing 9.10:

```
// member function that returns the external (leaf) nodes
// of the binary search tree, it calls recursive
// function externalNodes() to actually count the
// external nodes
template <class T>
int BSTree<T> :: countExternalNodes() {
  return externalNodes(root);
}

// recursive member function that returns the number of
// external (leaf) nodes in a binary search tree
template <class T>
int BSTree<T> :: externalNodes( BSTNode<T> *tree ) {
  if ( tree == NULL )
    return 0;
  else if ( (tree->left == NULL )&&(tree->right == NULL ) )
    return 1;
  else
    return ( externalNodes(tree->left) + externalNodes(tree->right));
}
```

9.6.2.7 Determining Mirror Image

The mirror image of a binary (search) tree is obtained by interchanging left and right ch recursively. Listing 9.11 shows the implementation of the various steps required.

Listing 9.11:

```
// member function to find the mirror image of the binary
// search tree, it calls recursive function reflect()
// to actually find the image
template <class T>
void BSTree<T>:: mirrorImage()
{
   reflect(root);
}
// recursive member function that reflects (mirror image)
// a binary search tree about an axis passing through
// its root
template <class T>
void BSTree<T>:: reflect( BSTNode<T> *tree )
{
   BSTNode<T>*  temp;
   if ( tree != NULL )
   {
      reflect( tree->left );
      reflect( tree->right );
      temp = tree->left;
      tree->left = tree->right;
      tree->right = temp;
   }
}
```

9.6.2.8 Removing Binary (Search) Tree from Memory

Before a program using binary (search) tree represented using linked representation termin the binary (search) tree must be removed from memory. To accomplish this task, first we del the left sub tree and then right sub tree and then node, recursively. The following listing sho the implementation of the various steps required.

Listing 9.12:

```
// destructor, that calls recursive function deleteTree()
// to remove binary search tree from memory
~BSTree()
{
   deleteTree(&root);
}

// recursive member function to remove the binary search
// tree from memory
template <class T>
void BSTree<T>::deleteTree ( BSTNode<T> **tree )
{
   if ( *tree != NULL ) {
      deleteTree( &((*tree)->left) );
      deleteTree( &((*tree)->right) );
      delete tree;
   }
}
```


The operations discussed next are relevant to binary search trees only.

9.6.2.9 Inserting a New Element

If the binary search tree is initially empty, then the element is inserted as root node; otherwise the new element is inserted as a terminal node. If the element is less than the element in the root node, then the new element is inserted in the left sub tree else right sub tree. Listing 9.13 shows the iterative implementation of the various steps required.

Listing 9.13:

```
// iterative member function that inserts a new element into
// binary search tree
template <class T>
void BSTree<T> :: insert( BSTNode<T> **tree, T element )
{
   BSTNode<T> *ptr, *nodePtr, *parentPtr;
   ptr = new BSTNode<T>;
   ptr->info = element;
   ptr->left = ptr->right = NULL;
   if ( *tree == NULL )
      *tree = ptr;
   else
   {
      parentPtr = NULL;
      nodePtr = *tree;
      while ( nodePtr != NULL ) {
         parentPtr = nodePtr;
         if ( element < nodePtr->info )
            nodePtr = nodePtr->left;
         else
            nodePtr = nodePtr->right;
      }
      if ( element < parentPtr->info )
         parentPtr->left = ptr;
      else
         parentPtr->right = ptr;
   }
}
```

The recursive implementation of insert operation will look like

Listing 9.14:

```
// member function to insert an element in binary search
// tree. It calls recursive function insert() to
// actually perform the insertion
template <class T>
void BSTree<T> :: insertElement( T element )
{
   insert (&root, element);
}

// recursive member function to insert an element into
// binary search tree
template <class T>
void BSTree<T> :: insert( BSTNode<T> **h, T element )
{
   if ( *h == NULL ) {
      *h = new BSTNode<T>;
      (*h)->info = element;
      (*h)->left = (*h)->right = NULL;
   } else if ( element < (*h)->info )
      insert( &((*h)->left), element);
   else
      insert( &((*h)->right), element);
}
```

9.6.2.10 Searching an Element

The element in a binary search tree can be searched very quickly. The search operation in binary search tree is similar to applying binary search technique to a sorted linear array. The element to be searched is compared with the root node. If it matches with the root node, the search terminates here; otherwise the search is continued in the left sub tree of the node if the element is less than the root node or in the right sub tree if the element is greater than the root node. Listings 9.15 show the recursive implementation of the various steps required.

Listing 9.15:

```
// member function to search a given element, it calls
// recursive function search() to actually perform the
// search
BSTNode<T> *searchElement(T value)
{
   return searchRecursive(root, value);
}

// recursive member function to search a given element,
// it return the location of the element if present,
// else returns NULL indicating that the element is
// not present in binary search tree
BSTNode<T> *searchRecursive( BSTNode<T> *tree, T value )
{
   if ( ( tree->info == value ) || ( tree == NULL ) )
      return tree;
   else if ( value < tree->info )
      return searchRecursive( tree->left, value );
   else
      return searchRecursive( tree->right, value );
}
```


The iterative implementation of search operations is left as an exercise for the readers.

9.6.2.11 Finding Smallest Node

Because of the order property of binary search tree, we know that the smallest node in the tree will be one of the nodes in the left sub tree, if it exists; otherwise the node itself will be the smallest node. The following listings show the recursive implementation of the various steps required. Its iterative implementation is left as an exercise for the readers.

Listing 9.16:

```
// member function that returns the location of the smallest
// element of the binary search tree, it calls recursive
// function findMinimum() to find the snallest element
template <class T>
BSTNode<T>* BSTree<T> :: findSmallest()
{
   return findMinimum(root);
}
// recursive member function that returns the location of
// the smallest element of the binary search tree.
// It returns NULL if tree is empty a given element or
// it has no left child
template <class T>
BSTNode<T>* BSTree<T> :: findMinimum ( BSTNode<T> *t )
{
   if ( ( t == NULL ) || ( t->left == NULL ) )
      return t;
   else
      return findMinimum (t->left);
}
```

9.6.2.12 Finding Largest Node

Because of the order property of BST, we know that the largest node in the tree will be one of the nodes in the right sub tree if it exists otherwise the node itself will be the largest node. The following listings show the recursive implementation of the various steps required. Its iterative implementation is left as an exercise for the readers.

Listing 9.17:

```
// member function that returns the location of the largest
// element of the binary search tree, it calls recursive
// function findMaximum() to find the largest element
template <class T>
```

```
BSTNode<T>* BSTree<T> :: findLargest()
{
   return findMaximum(root);
}

// recursive member function that returns the location
// of the largest element of the binary search tree.
// It returns NULL if tree is empty a given element or
// it has no right child
template <class T>
BSTNode<T>* BSTree<T>::findMaximum(BSTNode<T> *t)
{
   if ( ( t == NULL ) || ( t->right == NULL ) )
      return t;
   else
      return findMaximum(t->right);
}
```

9.6.2.13 Deleting a Node

To delete a node, the following possibilities may arise:

- ***node is terminal node*** — in this case, if the node is left child of its parent, then the left pointer of its parent is set to NULL otherwise it will be right child of its parent and accordingly the pointer of its parent is set to NULL.
- ***node having only one child*** — in this case, the appropriate pointer of its parent is set to its child, thus by passing it.
- ***node having two children*** — there are variety of ways in which this case can be handled. We will consider the way where its predecessor replaces the node value, and then the predecessor of the node is deleted.

Combining these three cases, the entire procedure for deleting a node can be implemented recursively as shown in the following listing.

Listing 9.18:

```
// member function to delete a given element from binary
// search tree, it calls recursive function deleteNode()
// to actually delete the element from binary search tree
template <class T>
void BSTree<T>::deleteElement(T element)
{
   deleteEle(&root,element);
}

// recursive member function that delete an element from
// binary search tree. It calls recursive member function to
// search the element to be deleted
template <class T>
void BSTree<T>::deleteNode(BSTNode<T> **tree, T element)
{
   BSTNode<T> *temp;
```



```
if ( *tree == NULL )
{
  cout << endl
       << "Element " << element << " not found in BST"
       << endl;
} else if ( element < (*tree)->info )
  deleteNode( &((*tree)->left), element );
else if ( element > (*tree)->info )
  deleteNode( &((*tree)->right), element );
else if ( (*tree)->left && (*tree)->right ) {
  temp = findMaximum ( (*tree)->left );
  (*tree)->info = temp->info;
  deleteNode( &((*tree)->left), temp->info );
} else {
  temp = *tree;
  if (((*tree)->left==NULL)&&((*tree)->right==NULL))
*tree = NULL;
  else if ( (*tree)->right == NULL )
*tree = (*tree)->left;
  else
*tree = (*tree)->right;
  delete temp;
}
}
```

From this onward, we will assume that the entire code of the class for implementing a binary search tree is stored in disk file named *btree.h*.

Listing 9.19:

```
// Program to illustrate the implementation of Binary Search Tree

#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include "btree.h"
void main()
{
  int choice, element;
  BSTNode<int> *loc;
  BSTree<int> tree;
  while ( 1 )
  {
     cout <<  "\n\n     Options available \n" ;
     cout <<  "++++++++++++++++++++++++++++++ \n\n" ;
     cout <<  "    1. Insert node\n" ;
     cout <<  "    2. Pre-order traversal\n" ;
     cout <<  "    3. In-order traversal\n" ;
     cout <<  "    4. Post-order traversal\n" ;
     cout <<  "    5. Level-order traversal\n" ;
     cout <<  "    6. Delete node\n" ;
     cout <<  "    7. Total nodes\n" ;
     cout <<  "    8. Total external nodes\n" ;
     cout <<  "    9. Total internal nodes\n" ;
     cout <<  "   10. Find height\n" ;
     cout <<  "   11. Smallest node\n" ;
```

```
        cout <<  "  12. Largest node\n" ;
        cout <<  "  13. Mirror Image\n" ;
        cout <<  "  14. Exit\n\n";
        cout <<  "Enter your choice ( 1-14 ) : " ;
        cin >> choice;

        switch ( choice )
        {
           case 1 : cout << "\nElement to be inserted into tree : ";
                    cin >> element;
                    tree.insertElement(element);
                    break;
           case 2 : cout << "\nPre-order Traversal of BST is\n" ;
                    tree.traversePre();
                    break;
           case 3 : cout << "\nIn-order Traversal of BST is\n" ;
                    tree.traverseIn();
                    break;
           case 4 : cout << "\nPost-order Traversal of BST is\n";
                    tree.traversePost();
                    break;
           case 5 : cout << "\nLevel-order Traversal of BST is\n";
                    tree.levelorderTraverse();
                    break;
           case 6 : cout << "\nElement to delete from tree : ";
                    cin >> element;
                    tree.deleteElement(element);
                    break;
           case 7 : cout << "\nNumber of nodes in BST = "
                         << tree.countTotalNodes();
                    break;
           case 8 : cout << "\nNumber of leaf nodes in BST = "
                         << tree.countExternalNodes();
                    break;
           case 9 : cout << "\nNumber of non-leaf nodes in BST = "
                         << tree.countInternalNodes();
                    break;
           case 10: cout <<  "\nHeight of BST = " <<
                    tree.heightTree();
                    break;
           case 11: loc = tree.findSmallest();
                    cout <<  "\nSmallest node in tree = " << loc->info ;
                    break;
           case 12: loc = tree.findLargest();
                    cout <<  "\nLargest node in tree = " << loc->info ;
                    break;
           case 13: tree.mirrorImage();
                    break;
           case 14: return;
        }
    }
}
```


9.7 AVL TREES

We can guarantee $O(\log_2 n)$ performance for each search tree operation by ensuring that the search tree height is always $O(\log_2 n)$. Trees with a worst-case height of $O(\log_2 n)$ are called *balanced trees*. One of the popular balanced trees is *AVL tree*, which was introduced by *Adelson-Velskii* and *Landis*.

If T is a nonempty binary tree with T_L and T_R as its left and right sub tree, then T is an AVL tree if and only if

1. $|h_L - h_R| \leq 1$, where h_L and h_R are the heights of T_L and T_R, respectively, and
2. T_L and T_R are AVL trees.

An AVL search tree is a binary search tree that is also an AVL tree. The height of an empty (null) AVL tree is taken as -1.

9.7.1 Representation of an AVL Tree

The node in an AVL tree will have an additional field *bf* (for balance factor) in addition to the structure of a node in a binary search tree.

Suppose that the elements of the AVL tree can be any primitive type or pointer type, the following is the required ADT to represent an AVL tree:

```
// class definition to represent a node of AVL tree
template <class T>
class AVLNode {
   public:
      AVLNode *left;
      T info;
      int bf;
      AVLNode *right;
};

// class definition for AVL tree
template <class T>
class AVLTree {
   private:
      AVLNode<T> *root;

   public:

      // member functions
}; // -------- End of AVLTree class --------
```

The value for field *bf* will be chosen as

$$bf = \begin{cases} -1 & \text{if } h_L < h_R \\ 0 & \text{if } h_L = h_R \\ +1 & \text{if } h_L > h_R \end{cases}$$

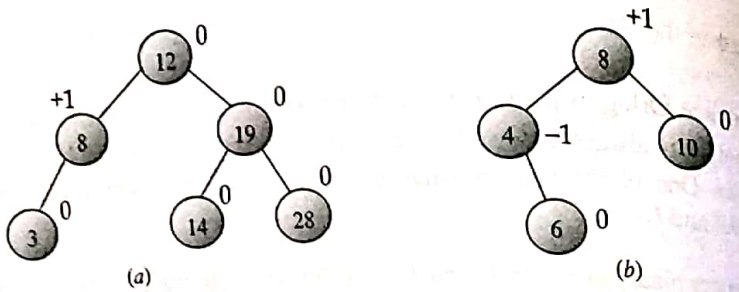


Figure 9.22: AVL trees

9.7.2 Height of an AVL Tree

Let N_h be the minimum number of nodes in an AVL tree of height h. In the worst case, the height of one of the sub trees is h–1, and the height of the other is h–2. Both these sub trees are also AVL trees. Hence

$N_h = N_{h-1} + N_{h-2} + 1$, $N_0 = 0$, and $N_1 = 1$

Notice the similarity between the definition for N_h and the definition of the Fibbonacci numbers

$F_n = F_{n-1} + F_{n-2}$, $F_0 = 0$, and $F_1 = 1$

It can be shown that $N_h = F_{h+2} - 1$ for $h \geq 0$. Fibbonacci number theory, we know that $F_n \approx \phi^n / \sqrt{5}$, where $\phi = (1+\sqrt{5})/2$. Hence, $N_h \approx \phi^{h+2} / \sqrt{5} - 1$. If there are n nodes in the tree then its height h is at most $\log_\phi(\sqrt{5}(n+1) - 2 \approx 1.44\log_2(n+2) = O(\log_2 n)$.

If an insertion and deletion causes an AVL tree to become unbalanced, then the tree must be restructured to return it to a balanced state.

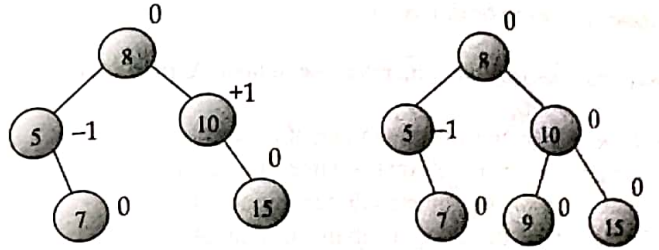
9.7.3 Operations on an AVL Tree

The operations on AVL tree that differ from a binary search tree are insertion and deletion only. These operations can disturb the balance property of an AVL tree. These operations are considered next.

9.7.3.1 Insertion of a Node

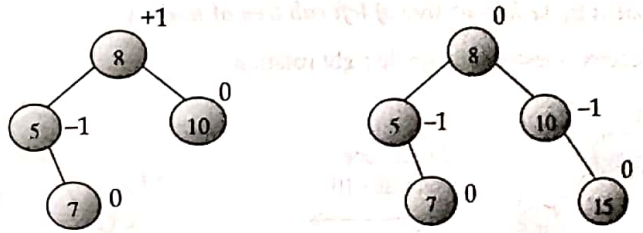
The new node is inserted using the usual binary search tree insert procedure i.e. comparing the key of the new node with that in the root, and inserting the new node into the left or right sub tree as appropriate. In some cases this insertion may not change the height of the sub tree, in which case neither the height nor the balance factor at the root will be changed. Even if it increases the height of the sub tree, there may be the case that this sub tree was shorter earlier, thus only the balance factor at the root will change where as the height of the tree remains unchanged. The only case that causes problems occurs when the node is inserted in a sub tree

that is taller than the other sub tree. This would cause one sub tree to have height more by a factor of 2 than the other sub tree whereas the AVL condition does not permit a difference of more than 1.



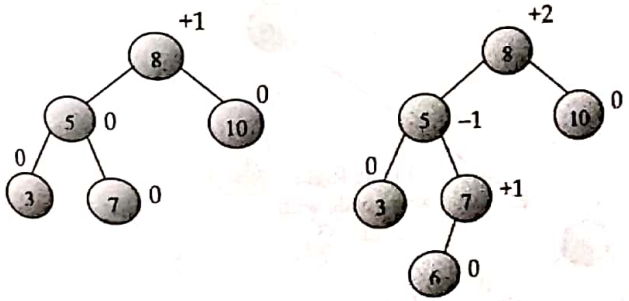
(a) Original AVL tree (b) After inserting value 9

Figure 9.23: Neither the balance factor of the root nor the height of the AVL tree is affected.



(a) Original AVL tree (b) After inserting value 15

Figure 9.24: Height remains unchanged but the balance factor of the root gets changed.



(a) Original AVL tree (b) After inserting value 6

Figure 9.25: Height as well as balance factor gets changed. It needs re-balancing about root node.

In order to restore the balance property, we use the tree rotations, which are described, in the next section.

To perform rotation it is necessary to identify a specific node, say node A, whose balance factor is neither 0, 1 or −1, and is the nearest ancestor to the inserted node on the path from the inserted node to the root. This implies that all nodes on the path from the inserted node to node A will have their balance factors to be 0, 1 or −1.

Based on the position of the inserted node with reference to node A, the following cases will arise:

- Inserted node is in the left sub tree of left sub tree of node A.
- Inserted node is in the right sub tree of right sub tree of node A.
- Inserted node is in the right sub tree of left sub tree of node A.
- Inserted node is in the left sub tree of right sub tree of node A.

These cases are demonstrated below through examples.

Case I: ***Inserted node is in the left sub tree of left sub tree of node A***

Here the balance property is restored by single right rotation.

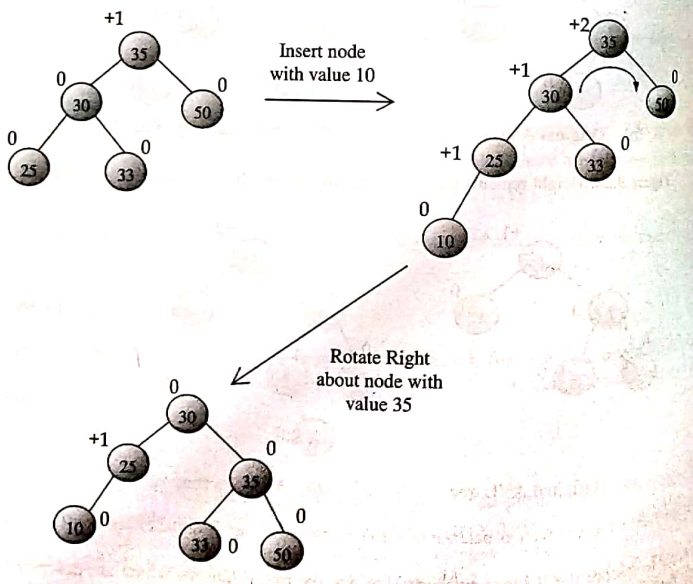


Figure 9.26: Restoring balance by right rotation

Case II: ***Inserted node is in the right sub tree of right sub tree of node A***

Here the balance property is restored by single left rotation.

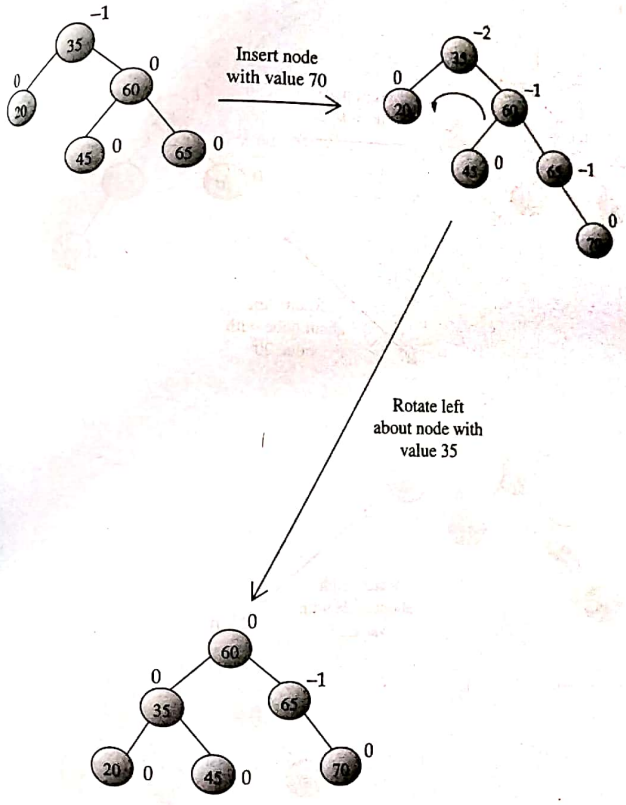


Figure 9.27: Restoring balance by left rotation

Case III: ***Inserted node is in the right sub tree of left sub tree of node A***

Here the balance property is restored by double rotation - left rotation followed by the right rotation.

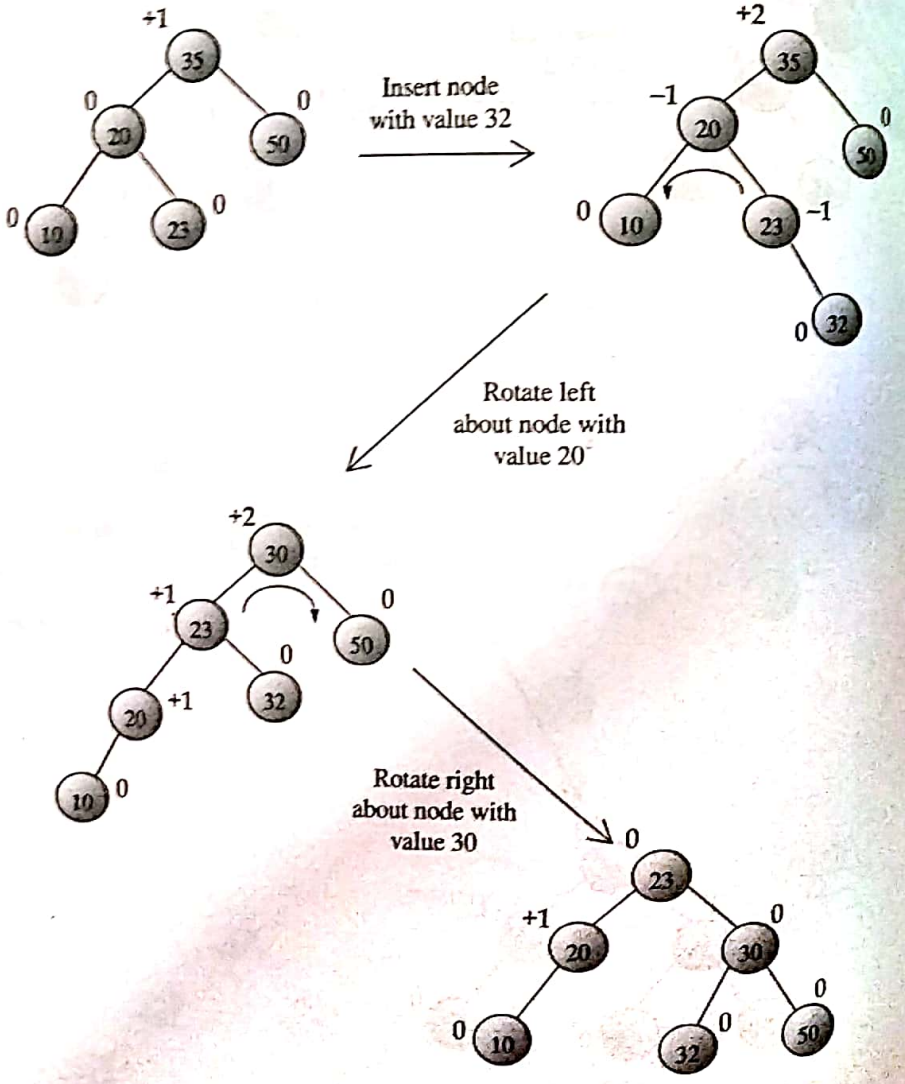


Figure 9.28: Restoring balance by double rotation
(left rotation followed by right rotation)

Case IV: ***Inserted node is in the left sub tree of right sub tree of node A***

Here the balance property is restored by double rotation - right rotation followed by the left rotation.

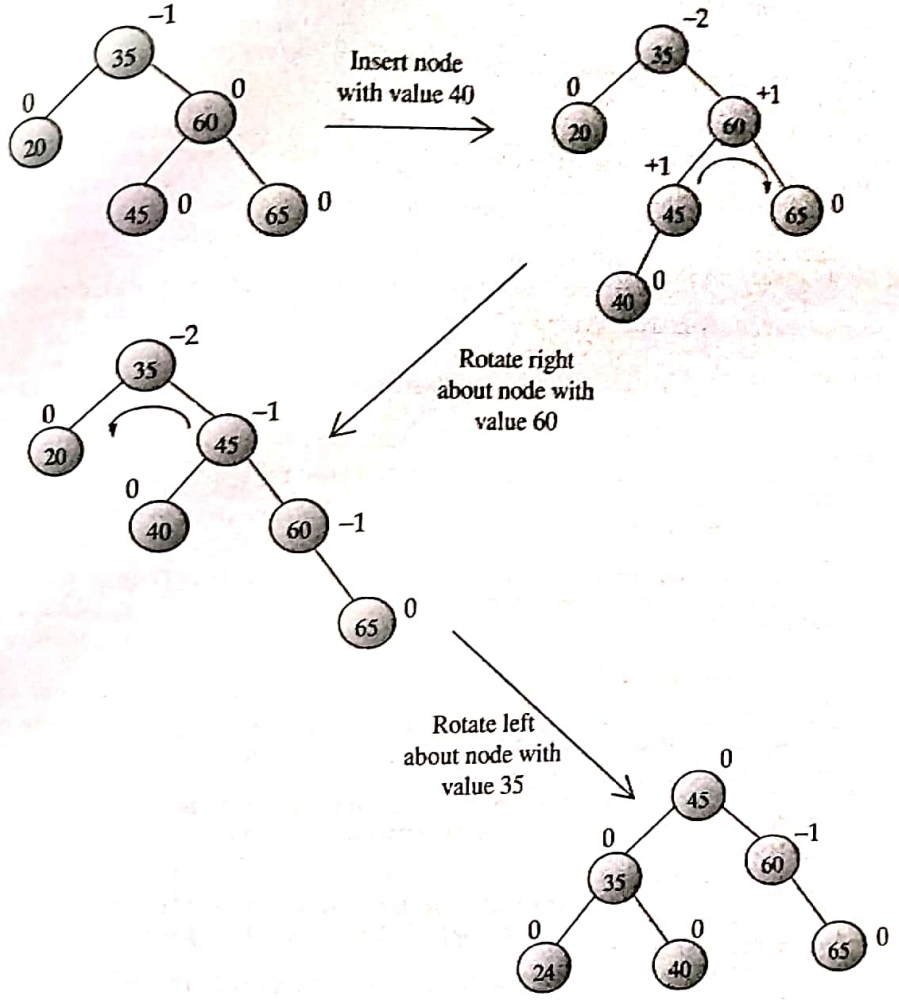


Figure 9.29: Restoring balance by double rotation
(right rotation followed by left rotation)

9.7.3.2 Deletion of a Node

Deletion of a node from an *AVL* tree requires the same basic ideas including single and double rotations, which are used for insertion.

Listing 9.20

```
// Program to illustrate the implementation of AVL Tree
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>

// class definition for node of an AVL tree
template <class T>
class AVLNode
{
   public:
      AVLNode *left;
      T item;
      int bf;
      AVLNode *right;
};
// class definition for AVL tree
template <class T>
class AVLTree {
   private:
      AVLNode<T> *root;
   public:
      // constructor that initializes AVL tree to empty state
      AVLTree() {
        root = NULL;
      }
      // destructor, it calls recursive function deleteTree() to
      // remove the AVL tree from memory
      ~AVLTree()
      {
        deleteTree(&root);
      }
// function to insert an element into AVL tree, it calls recursive
// function insertNode() that actually performs the insertion
      void insertElement(int);
// member function to perform inorder traversal of AVL tree, it
// calls recursive function traverseInorder() to actually perform
// the inorder traversal
      void traverseIn();
// function to delete an element from AVL tree, it calls recursive
// function deleteNode() that actually performs the deletion
      void deleteElement(int);
// function that returns the height of  an AVL tree, it calls
// recursive function findheight() that actually find the height
// of the AVL tree
      int heightTree();

      // ------ RECURSIVE MEMBER FUNCTIONS -------

// recursive function that inserts a new element into AVL tree,
// and readjusts it with suitable rotations so that it remains
// AVL tree after insertion
      static void insertNode(AVLNode<T> **, int);
// recursive function that performs inorder traversal
// of AVL tree
      static void traverseInorder(AVLNode<T> *);
```



```
// recursive function that return the largest element in AVL tree
      static AVLNode<T> *findMaximum(AVLNode<T> *);
// recursive function that deletes given element from AVL tree
      static void deleteNode(AVLNode<T> **, int);
// recursive function that removes AVL tree from memory
      static void deleteTree(AVLNode<T> **);
// recursive function that find the height of the AVL tree
      static void findHeight(AVLNode<T> *, int *);

};

// function to insert an element into AVL tree, it calls recursive
// function insertNode() that actually performs the insertion
template <class T>
void AVLTree<T> :: insertElement( T element ) {
  insertNode (&root, element);
}
// recursive function that inserts a new element into AVL tree, and
// readjusts it with suitable rotations so that it remains
// AVL tree after insertion
template <class T>
void AVLTree<T> :: insertNode( AVLNode<T> **h, int element ) {
  AVLNode<T> *ptr1, *ptr2;
  if ( *h == NULL ) {
     *h = new AVLNode<T>;
     (*h)->item = element;
     (*h)->left = (*h)->right = NULL;
     (*h)->bf = 0;

  } else if ( element < (*h)->item ) {
     insertNode( &((*h)->left), element);
     switch ( (*h)->bf )
     {
       case 1: ptr1 = (*h)->left;
               if ( ptr1->bf == 1 ) {
                  (*h)->left = ptr1->right;     // right rotation
                  ptr1->right = *h;
                  (*h)->bf = 0;
                  *h = ptr1;
               } else {
                  ptr2 = ptr1->right; // double rotation, left-right
                  ptr1->right = ptr2->left;
                  ptr2->left = ptr1;
                  (*h)->left = ptr2->right;
                  ptr2->right = *h;

                  (*h)->bf = ptr2->bf == 1 ? -1 : 0;
                  ptr1->bf = ptr2->bf == -1 ? 1 : 0;

                  *h = ptr2;
               }
               break;
       case 0: (*h)->bf = 1;
               break;
       case -1: (*h)->bf = 0;
     }
  } else {
```

```
      insertNode( &((*h)->right), element);
      switch ( (*h)->bf )
      {
        case 1: (*h)->bf = 0;
                break;
        case 0: (*h)->bf = -1;
                break;
        case -1: ptr1 = (*h)->right;
                 if ( ptr1->bf == -1 ) {
                    (*h)->right = ptr1->left;      // left rotation
                    ptr1->left = *h;
                    (*h)->bf = 0;
                    *h = ptr1;
                 } else {
                    ptr2 = ptr1->left; // double rotation, right-left
                    ptr1->left = ptr2->right;
                    ptr2->right = ptr1;
                    (*h)->right = ptr2->left;
                    ptr2->left = *h;

                    (*h)->bf = ptr2->bf == -1 ? 1 : 0;
                    ptr1->bf = ptr2->bf == 1 ? -1 : 0;

                    *h = ptr2;
                 }
                 break;
      }
   }
}
// member function to perform inorder traversal of AVL tree, it calls
// recursive function traverseInorder() to actually perform the
// inorder traversal
template <class T>
void AVLTree<T> :: traverseIn()
{
   traverseInorder(root);
}
// recursive function that performs inorder traversal of AVL tree
template <class T>
void AVLTree<T> :: traverseInorder ( AVLNode<T> *h ) {
   if ( h != NULL ) {
      traverseInorder( h->left );
      cout << h->item << "  ";
      traverseInorder( h->right );
   }
}
// recursive function that return the largest element in AVL tree
template <class T>
AVLNode<T>* AVLTree<T>::findMaximum(AVLNode<T> *t) {
   if ( ( t == NULL ) || ( t->right == NULL ) )
      return t;
   else
      return findMaximum(t->right);
}
// function to delete an element from AVL tree, it calls recursive
// function deleteNode() that actually performs the deletion
template <class T>
```



```
void AVLTree<T>::deleteElement(T element) {
   deleteNode(&root,element);
}
// recursive function that deletes given element from AVL tree
template <class T>
void AVLTree<T>::deleteNode (AVLNode<T> **h, T element) {
   AVLNode<T> *temp, *ptr1, *ptr2;
   if ( *h == NULL ) {
      cout << endl
           << "Element " << element << " not found in AVL Tree";
      cout << endl
           << "Press any key to continue ...";
      getch();
   } else if ( element < (*h)->item ) {
      deleteNode( &((*h)->left), element );
      switch ( (*h)->bf )
      {
        case 1: (*h)->bf = 0;
                break;
        case 0: (*h)->bf = -1;
                break;
        case -1: ptr1 = (*h)->right;
                 if ( ptr1->bf == -1 ) {
                    (*h)->right = ptr1->left;      // left rotation
                    ptr1->left = *h;
                    (*h)->bf = 0;
                    *h = ptr1;
                 } else {
                    ptr2 = ptr1->left; // double rotation, right-left
                    ptr1->left = ptr2->right;
                    ptr2->right = ptr1;
                    (*h)->right = ptr2->left;
                    ptr2->left = *h;

                    (*h)->bf = ptr2->bf == -1 ? 1 : 0;
                    ptr1->bf = ptr2->bf == 1 ? -1 : 0;

                    *h = ptr2;
                 }
      }
   } else if ( element > (*h)->item ) {
      deleteNode( &((*h)->right), element );
      switch ( (*h)->bf )
      {
        case 1: ptr1 = (*h)->left;
                if ( ptr1->bf == 1 ) {
                   (*h)->left = ptr1->right;      // right rotation
                   ptr1->right = *h;
                   (*h)->bf = 0;
                   *h = ptr1;
                } else {
                   ptr2 = ptr1->right; // double rotation, left-right
                   ptr1->right = ptr2->left;
                   ptr2->left = ptr1;
                   (*h)->left = ptr2->right;
                   ptr2->right = *h;
                   (*h)->bf = ptr2->bf == 1 ? -1 : 0;
```

```
                ptr1->bf = ptr2->bf == -1 ? 1 : 0;

                *h = ptr2;
              }
              break;
      case 0: (*h)->bf = 1;
              break;
      case -1: (*h)->bf = 0;
    }
  } else if ( (*h)->left && (*h)->right ) {
    temp = findMaximum ( (*h)->left );        // find predecessor
    (*h)->item = temp->item;   // replace node by predecessor
    deleteNode( &((*h)->left), temp->item );  // delete predecessor
  } else {
    temp = *h;
    if (((*h)->left==NULL)&&((*h)->right==NULL)) // terminal node
      *h = NULL;
    else if ( (*h)->right == NULL ) // left child only
      *h = (*h)->left;
    else                            // right child only
      *h = (*h)->right;
    delete temp;
  }
}
// function that returns the height of  an AVL tree, it calls
// recursive function findheight() that actually find the height
// of the AVL tree
template <class T>
int AVLTree<T>:: heightTree() {
  int h;
  findHeight(root, &h);
  return h;
}
// recursive function to find the height of the AVL tree
template <class T>
void AVLTree<T>:: findHeight ( AVLNode<T> *tree, int *h ) {
  int hl, hr;
  if ( tree == NULL ) {
    *h = 0;
  } else {
    findHeight ( tree->left, &hl );
    findHeight ( tree->right, &hr );
    if ( hl > hr )
      *h = hl + 1;
    else
      *h = hr + 1;
  }
}
// recursive function that removes AVL tree from memory
template <class T>
void AVLTree<T>::deleteTree ( AVLNode<T> **tree ) {
  if ( *tree != NULL ) {
    deleteTree( &((*tree)->left) );
    deleteTree( &((*tree)->right) );
    delete tree;
  }
}
```



```
                // ------- main function -------
void main()
{
  int choice, element;
  AVLNode<int> *loc;
  AVLTree<int> tree;
  while ( 1 )
  {
    cout <<   "\n\n      Options available \n" ;
    cout <<   "++++++++++++++++++++++++++++++++ \n\n" ;
    cout <<   "    1. Insert node\n" ;
    cout <<   "    2. In-order traversal\n" ;
    cout <<   "    3. Delete node\n" ;
    cout <<   "    4. Find height\n" ;
    cout <<   "    5. Exit\n\n";
    cout <<   "Enter your choice ( 1-5 ) : " ;
    cin >> choice;
    switch ( choice )
    {
      case 1 : cout << "\n\nElement to be inserted into tree : ";
               cin >> element;
               tree.insertElement(element);
               break;
      case 2 : cout  << "\n\nIn-order Traversal of AVLT is\n" ;
               tree.traverseIn();
               break;
      case 3 : cout << "Element to delete from tree : ";
               cin >> element;
               tree.deleteElement(element);
               break;
      case 4 : cout <<  "\n\nHeight of AVL Tree = " <<
               tree.heightTree() ;
               break;
      case 5 : return;
    }
  }
}
```

9.8 THREADED BINARY TREES

On carefully examining the linked representation of a binary tree *T*, you will find that approximately half of the pointer fields (*left* and *right* pointer fields that hold the address of the left child and right child, respectively) contain NULL entries (pictured as X) in these fields. The space occupied by these NULL entries can be utilized to store some kind of valuable information.

One possible way to utilize this space is that we can store special pointer that point to nodes higher in the tree, *i.e.*, ancestors. These special pointers are called *threads*, and the binary tree having such pointers is called *a threaded binary tree*.

Threads in a binary tree must be distinguished from normal pointers. In the graphical representation of a threaded binary tree, the threads are shown by dotted lines. In computer memory, an extra field, called *tag* or *flag* is used to distinguish a thread from a normal pointer.

There are many ways to thread a binary tree. These are

- The right NULL pointer of each node can be replaced by a thread to the successor of the node under in-order traversal called a *right thread*, and the tree will be called a *right threaded tree*.
- The left NULL pointer of each node can be replaced by a thread to the predecessor of the node under in-order traversal called a *left thread*, and the tree will be called a *left-threaded tree*.
- Both left and right NULL pointers can be used to point to predecessor and successor of the node, respectively, under in-order traversal. Such a tree is called a *fully threaded tree*.

A threaded binary tree where only one thread is used is also known as *one-way threaded* tree and where both the threads are used is also known as *two-way* threaded tree.

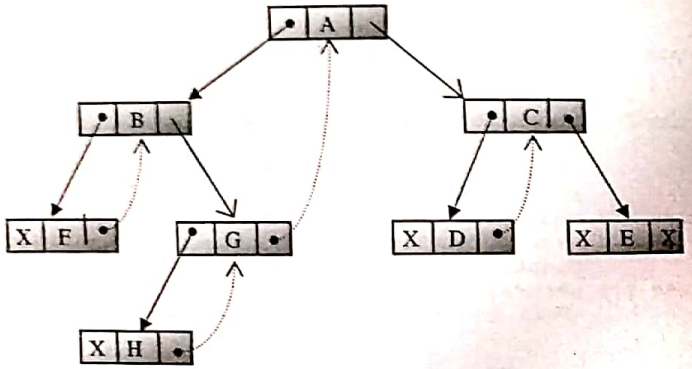


Figure 9.30: Right-threaded binary tree (*one-way threading*)

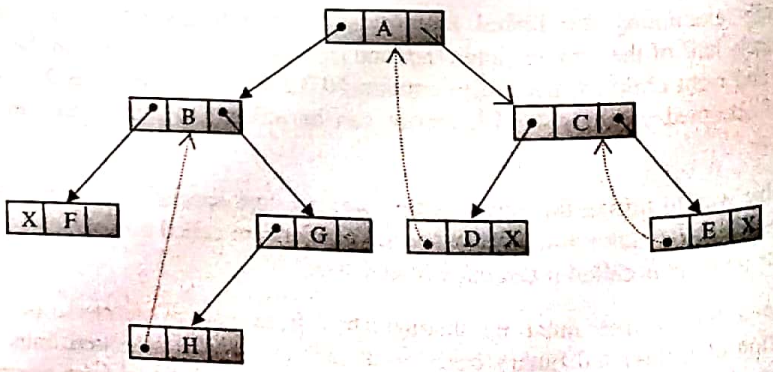


Figure 9.31: Left-threaded binary tree (*one-way threaded*)

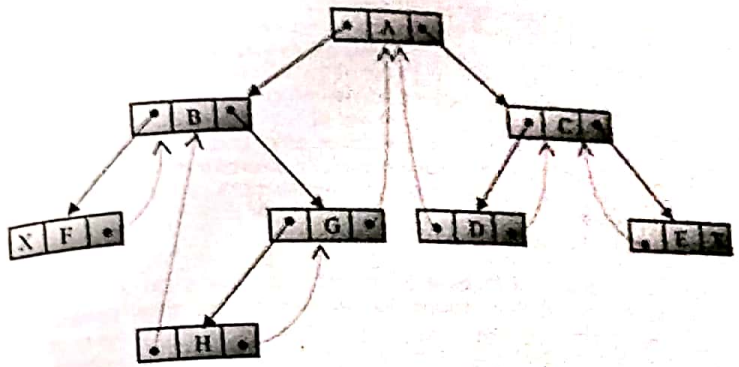


Figure 9.32: A fully threaded binary tree (*two-way threading*)

We will discuss the representation of right-threaded binary and the some operations that differ from a non-threaded binary tree (insertion, deletion, traversal, removal).

9.8.1 Representing a Threaded Binary Tree in Memory

To represent the above threaded binary tree, we need following kind of representation for the node

```
class TBSTNode
{
   public:
      int info;
      char thread;
      struct TBSTNode *left;
      struct TBSTNode *right;
};
class ThreadedBST
{
   protected:
      TBSTNode *root;
   public:
      // constructor to initialize threaded binary search tree
      // to empty state
      ThreadedBST()
      {
         root = NULL;
      }
      // destructor, that calls recursive function removeTBSTree()
      // to remove binary search tree from memory
      ~ThreadedBST()
      {
         removeTBSTree(&root);
      }
      // member function to perform inorder traversal of
      // threaded binary search tree
      void inorderTraversal();
      // member function to insert an element in threaded
      // binary search tree
```

```
        void insertElement(int element);
        // member function to delete a given element from threaded
        // binary search tree, it calls recursive function
        // deleteNode() to actually delete the element from
        // threaded binary search tree
        void deleteElement(int element);
        // member function that returns the location of the largest
        // element of the threaded binary search tree, it calls
        // recursive function findLargestNode() to find the largest
        // element
        TBSTNode *largestNode();

        // ------ RECURSIVE MEMBER FUNCTIONS -------
        // recursive member function that returns the location of
        // the largest element of the threaded binary search tree.
        // It returns NULL if tree is empty a given element.
        static TBSTNode *findLargestNode(TBSTNode *ptr);
        // recursive member function to delete a node containing
        // the given element from threaded binary search tree
        static void deleteNode(TBSTNode **ptr, int element);
        // recursive member function to remove a threaded
        // binary search tree from memory
        static void removeTBSTree( TBSTNode **tree );
};
```

In this representation, we have used *char* field *thread* as a tag. The character '0' will used a normal right pointer and character '1' will be used for thread.

9.8.2 Operations on a Threaded Binary Tree

The only operations that differ from a binary search tree are traversal, insertion of a new element, deletion of an element and deletion of the tree itself. These operations are described in the next section.

9.8.2.1 In-order Traversal of Threaded Binary Tree

Using the above representation, the iterative version of the in-order traversal can be expressed as shown in the following code.

Listing 9.21:

```
// member function to perform inorder traversal of
// threaded binary search tree
void ThreadedBST::inorderTraversal()
{
   TBSTNode *pPtr, *qPtr;
   pPtr = root;
   do
   {
      qPtr = NULL;
      while ( pPtr != NULL )   //  traverse left branch
      {
         qPtr = pPtr;
         pPtr = pPtr->left;
      }
```



```
      if ( qPtr != NULL )      //  tree is not empty
      {
         cout << qPtr->info << "   ";
         pPtr = qPtr->right;
         while ( ( qPtr->thread == '1' )&&( pPtr != NULL ) )
         {
            cout << pPtr->info << "   ";
            qPtr = pPtr;
            pPtr = pPtr->right;
         }
      }
   }
   while ( pPtr != NULL );
}
```

If you compare this implementation with the recursive implementation or earlier iterative implementation, you will find that this implementation if more efficient. It is more efficient in the sense that it uses minimum resources of the system and runs faster as no stack is used here. Note that the recursive algorithms use stack implicitly, where as we have to maintain our own stack in the iterative implementation. This adds extra overhead on the system and reduces the efficiency.

9.8.2.2 Insertion in a Threaded Binary Search Tree

The following code shows the steps required to insert an element in a threaded binary search tree.

Listing 9.22:

```
// member function to insert an element in threaded
// binary search tree
void ThreadedBST::insertElement(int element )
{
  TBSTNode *ptr, *nodePtr, *parentPtr;
  ptr = new TBSTNode;
  ptr->info = element;
  ptr->left = NULL;
  if ( root == NULL )  // Intially tree empty
  {
     root = ptr;
     ptr->right = NULL;
     ptr->thread = '0';
  }
  else
  {
     parentPtr = NULL;
     nodePtr = root;
     while ( nodePtr != NULL )
     {
        parentPtr = nodePtr;
        if ( element < nodePtr->info )
           nodePtr = nodePtr->left;
        else
        {
           if ( nodePtr->thread == '1' )
```

```
            nodePtr = NULL;
         else
            nodePtr = nodePtr->right;
      }
   }
   if ( element < parentPtr->info )
   {
      parentPtr->left = ptr;
      ptr->right = parentPtr;
      ptr->thread = '1';
   } else {
      if ( parentPtr->thread == '1' ) {
         parentPtr->thread = '0';
         ptr->thread = '1';
         ptr->right = parentPtr->right;
         parentPtr->right = ptr;
      } else {
         ptr->thread = '0';
         ptr->right = NULL;
         parentPtr->right = ptr;
      }
   }
}
```

If the inserted element becomes left terminal node, then its right pointer field is a thread and points to its parent node.

If the inserted element becomes right terminal node, then its right pointer field is a thread and points to its parent node, the there two possibilities:

1. If its parent's right pointer field is NULL i.e. right pointer field is not a thread, then the right pointer field of new node will contain value NULL.
2. If its parent's right pointer field is a thread, then the right pointer field of new node will contain thread of its parent.

9.8.2.3 Finding Largest Element a Threaded Binary Search Tree

The following code shows the steps required finding the location of the largest element in a threaded binary search tree.

Listing 9.23:

```
// member function that returns the location of the largest
// element of the threaded binary search tree, it calls
// recursive function findLargestNode() to find the largest
// element
TBSTNode * ThreadedBST::largestNode()
{
   return findLargestNode(root);
}
// recursive member function that returns the location of
// the largest element of the threaded binary search tree.
// It returns NULL if tree is empty a given element.
```



```
TBSTNode * ThreadedBST::findLargestNode(TBSTNode *ptr)
{
   if ( ptr == NULL )
      return NULL;
   else if ( ( ptr->thread == '1') || ( ptr->right == NULL ) )
      return ptr;
   else
      return findLargestNode(ptr->right);
}
```

9.8.2.4 Deleting an Element From a Threaded Binary Search Tree

The following code shows the steps required deleting an element from a threaded binary search tree.

Listing 9.24:

```
// member function to delete a given element from threaded
// binary search tree, it calls recursive function
// deleteNode() to actually delete the element from
// threaded binary search tree
void ThreadedBST::deleteElement(int element) {
   deleteNode(&root,element);
}
// recursive member function to delete a node containing
// the given element from threaded binary search tree
void ThreadedBST::deleteNode(TBSTNode **ptr, int element) {
   TBSTNode *tempPtr1, *tempPtr2;
   char tempThread;
   if ( *ptr == NULL ) {
      cout << "\nElement " << element << " not found in TBST\n";
      return;
   }
   if ( element < (*ptr)->info )
      deleteNode( &((*ptr)->left), element );
   else if ( element > (*ptr)->info )
      deleteNode( &((*ptr)->right), element );
   else if ((*ptr)->left&&(*ptr)->right&&(*ptr)->thread=='0') {
      tempPtr1 = findLargestNode( (*ptr)->left );
      (*ptr)->info = tempPtr1->info;
      deleteNode( &((*ptr)->left), tempPtr1->info );
   } else {
      tempPtr1 = *ptr;
      // terminal node
      if ( ((*ptr)->left == NULL) &&
           ( ((*ptr)->right == NULL) || ((*ptr)->thread=='1') ) )
         *ptr = NULL;    // tree contains root node only
      else if ( (*ptr)->left != NULL )  // left child only
      {
         tempThread = (*ptr)->thread;
         tempPtr2 = (*ptr)->right;
         *ptr = (*ptr)->left;
         (*ptr)->thread = tempThread;
         if ( tempThread == '1' )
            (*ptr)->right = tempPtr2;
      }
```

```
        else                                    // right child only
            *ptr = (*ptr)->right;
        delete tempPtr1;
    }
}
```

9.8.2.5 Deleting a Threaded Binary Tree

The following code shows the steps required to delete a threaded binary or binary search tree.

Listing 9.25:

```
// destructor, that calls recursive function removeTBSTree()
// to remove binary search tree from memory
~ThreadedBST()
{
   removeTBSTree(&root);
}
// recursive member function to remove a threaded
// binary search tree from memory
void ThreadedBST::removeTBSTree( TBSTNode **tree )
{
   if ( *tree != NULL ) {
      removeTBSTree( &(*tree)->left );
      if ( (*tree)->thread == '0' )
          removeTBSTree( &(*tree)->right );
      delete *tree;
   }
}
```

Listing 9.26:

```
//  Program to illustrate the implementation of different operations
//  discussed above on a Threaded Binary Search Tree
# include <iostream.h>
# include <iomanip.h>
# include <stdlib.h>
class TBSTNode
{
   public:
      int info;
      char thread;
      struct TBSTNode *left;
      struct TBSTNode *right;
};
class ThreadedBST
{
   protected:
      TBSTNode *root;
   public:
      ThreadedBST()
      {
        root = NULL;
      }
      ~ThreadedBST()
```



```
      {
        removeTBSTree(&root);
      }
      void inorderTraversal();
      void insertElement(int element);
      void deleteElement(int element);
      void deleteNode(TBSTNode **ptr, int element);
      TBSTNode *largestNode();
      TBSTNode *findLargestNode(TBSTNode *ptr);
      static void removeTBSTree( TBSTNode **tree );
};
void ThreadedBST::inorderTraversal()
{
  TBSTNode *pPtr, *qPtr;
  pPtr = root;
  do
  {
    qPtr = NULL;
    while ( pPtr != NULL )    //  traverse left branch
    {
      qPtr = pPtr;
      pPtr = pPtr->left;
    }
    if ( qPtr != NULL )       //   tree is not empty
    {
      cout << qPtr->info << "   ";
      pPtr = qPtr->right;
      while ( ( qPtr->thread == '1' )&&( pPtr != NULL ) )
      {
         cout << pPtr->info << "   ";
         qPtr = pPtr;
         pPtr = pPtr->right;
      }
    }
  }
  while ( pPtr != NULL );
}
void ThreadedBST::insertElement(int element )
{
  TBSTNode *ptr, *nodePtr, *parentPtr;
  ptr = new TBSTNode;
  ptr->info = element;
  ptr->left = NULL;
  if ( root == NULL )  // Intially tree empty
  {
    root = ptr;
    ptr->right = NULL;
    ptr->thread = '0';
  }
  else
  {
    parentPtr = NULL;
    nodePtr = root;
    while ( nodePtr != NULL )
    {
      parentPtr = nodePtr;
      if ( element < nodePtr->info )
        nodePtr = nodePtr->left;
```

```
        else
        {
          if ( nodePtr->thread == '1' )
             nodePtr = NULL;
          else
             nodePtr = nodePtr->right;
        }
      }
      if ( element < parentPtr->info )
      {
        parentPtr->left = ptr;
        ptr->right = parentPtr;
        ptr->thread = '1';
      } else {
        if ( parentPtr->thread == '1' ) {
            parentPtr->thread = '0';
            ptr->thread = '1';
            ptr->right = parentPtr->right;
            parentPtr->right = ptr;
        } else {
            ptr->thread = '0';
            ptr->right = NULL;
            parentPtr->right = ptr;
        }
      }
   }
}
TBSTNode * ThreadedBST::largestNode()
{
   return findLargestNode(root);
}
TBSTNode * ThreadedBST::findLargestNode(TBSTNode *ptr)
{
   if ( ptr == NULL )
       return NULL;
   else if ( ( ptr->thread == '1') || ( ptr->right == NULL ) )
       return ptr;
   else
       return findLargestNode(ptr->right);
}
void ThreadedBST::deleteElement(int element)
{
   deleteNode(&root,element);
}
void ThreadedBST::deleteNode(TBSTNode **ptr, int element)
{
    TBSTNode *tempPtr1, *tempPtr2;
    char tempThread;
    if ( *ptr == NULL ) {
        cout << "\nElement " << element << " not found in TBST\n";
        return;
    }
    if ( element < (*ptr)->info )
       deleteNode( &((*ptr)->left), element );
    else if ( element > (*ptr)->info )
       deleteNode( &((*ptr)->right), element );
    else if ( (*ptr)->left && (*ptr)->right && (*ptr)->thread == '0' ) {
       tempPtr1 = findLargestNode( (*ptr)->left );
```



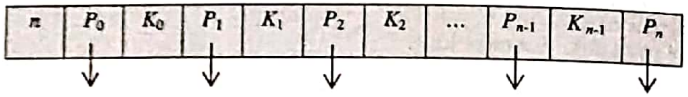
```
       (*ptr)->info = tempPtr1->info;
       deleteNode( &((*ptr)->left), tempPtr1->info );
    } else {
       tempPtr1 = *ptr;
       // terminal node
       if ( ((*ptr)->left == NULL) &&
            ( ((*ptr)->right == NULL) || ((*ptr)->thread=='1') ) )
          *ptr = NULL;    // tree contains root node only
       else if ( (*ptr)->left != NULL )  // left child only
       {
          tempThread = (*ptr)->thread;
          tempPtr2 = (*ptr)->right;
          *ptr = (*ptr)->left;
          (*ptr)->thread = tempThread;
          if ( tempThread == '1' )
             (*ptr)->right = tempPtr2;
       }                                  // right child only
       else
          *ptr = (*ptr)->right;
       delete tempPtr1;
    }
}
void ThreadedBST::removeTBSTree( TBSTNode **tree )
{
   if ( *tree != NULL ) {
      removeTBSTree( &(*tree)->left );
      if ( (*tree)->thread == '0' )
        removeTBSTree( &(*tree)->right );
      delete *tree;
   }
}
void main()       //    main function
{
   int choice, element, height;
   ThreadedBST tbtree;
   while ( 1 )
   {
      cout << "\n\n      Options available \n";
      cout << "+++++++++++++++++++++++++++++++ \n\n";
      cout << "    1. Insert node\n";
      cout << "    2. In-order traversal\n";
      cout << "    3. Delete node\n";
      cout << "    4. Exit\n\n";
      cout << "Enter your choice ( 1-4 ) : ";
      cin >> choice;
      switch( choice )
      {
         case 1 : cout << "Element to be inserted into tree : ";
                  cin >> element;
                  tbtree.insertElement(element);
                  break;
         case 2 : cout << "\nIn-order Traversal of TBST is\n";
                  tbtree.inorderTraversal();
                  break;
         case 3 : cout << "Element to delete from tree : ";
                  cin >> element;
                  tbtree.deleteElement(element);
                  break;
```

```
            case 4 : exit(1);
        }
    }
}
```

9.9 M-WAY SEARCH TREES

An *m-way search tree* is a tree in which each node has out-degree ≤ m. When m-way search tree is not empty, it has the following properties:

- Each node has the following structure



  where P_0, P_1, P_2,..., P_n are pointers to the node's sub trees and K_0, K_1, K_2,..., K_{n-1} are the key values. The requirement that each node have out-degree ≤ m forces $n \le m-1$.

- The key values in a node are in ascending order, *i.e.*

  $$K_i < K_{i+1} \quad \text{for } i = 0, 1, 2, \ldots, n-2$$

- All key values in nodes of the sub tree pointed to by P_i are less than the key value K_i for $i = 0, 1, 2, \ldots, n-1$.
- All key values in nodes of the sub tree pointed to by P_n are greater than the key value K_{n-1}.
- The sub trees pointed to by P_i for $i = 0, 1, 2, \ldots, n$ are also m-way search trees.

We can define the node type for a 7-way search tree as follows:

```
class Node {
   public:
      int key[6];
      Node *nodePtr[7];
};
```

9.9.1 M-Way Search Tree as Indexes

When an m-way search tree is used an index, each key-pointer pair (K_i, P_i) becomes a triplet (K_i, P_i, A_i), where A_i is the address of record associated with key value K_i. Thus, each node not only points to its child nodes in the tree, but also to associated records. We can define the node type for a 7-way search tree index as follows:

```
class Node {
   public:
      int key[6];
      recordPtr *recPtr[6];
      Node *nodePtr[7];
};
```

where *recordPtr* is assumed to be a type that holds an address of the associated record with a given key.

9.9.2 Searching an M-Way Search Tree

Searching for a key in an m-way search tree is an extension of searching a key in a binary search tree. The procedure is as follows:

- If the key is less than K_0, then the search is continued in m-way search tree pointed to by P_0.
- If the key lies between K_0 and K_1, then the search is continued in m-way search tree pointed to by P_1.
- If the key lies between K_1 and K_2, then the search is continued in m-way search tree pointed to by P_2.

  :

  :

- If the key is greater than K_{n-1}, then the search is continued in m-way search tree pointed to by P_n.

9.10 B-TREE

The B-tree structure was discovered by *R. Bayer* and *E. McCreight* of Boeing Scientific Research Labs and has become one of the most popular techniques for organizing an index structure. A *B-tree of order m* is an m-way search tree with the following properties:

- Each node of the tree, except the root node and leaves, has at least $m/2$ sub trees and no more than m sub trees.
- Root of the tree has at least two sub trees unless it is a leaf node.
- All leaves of the tree are on the same level.

The first condition ensures that each node of the tree is at least half full, the second one forces the tree to branch early while the third one keeps the tree balanced.

Several variations of the basic B-tree have been developed, two of them, that have become popular are:

- *B⁺-Tree* — it is a B-tree in which the leave nodes are linked to each other from left to right to form a linked list of the keys in sequential order.
- *B*-Tree* — it is a B-tree in which each interior node is at least two-third full rather than just half full.

9.10.1 Searching a B-tree

The procedure for searching a key in a B-tree is same as searching a key in an m-way search tree. Then what is performance of search in B-tree? To find the answer, let us first find the minimum number of keys in a B-tree of order m with height h. By definition, the root of B-tree has at least two children. The internal nodes, that are upto level h-2, has at least $m/2$ children.

Therefore, the B-tree must contain at least $\left[1 + 2 + \sum_{i=3}^{h-2} 2\lceil m/2 \rceil^{i-2}\right]$ internal nodes and at least $\left[2\lceil m/2 \rceil^{h-2}\right]$ leaves.

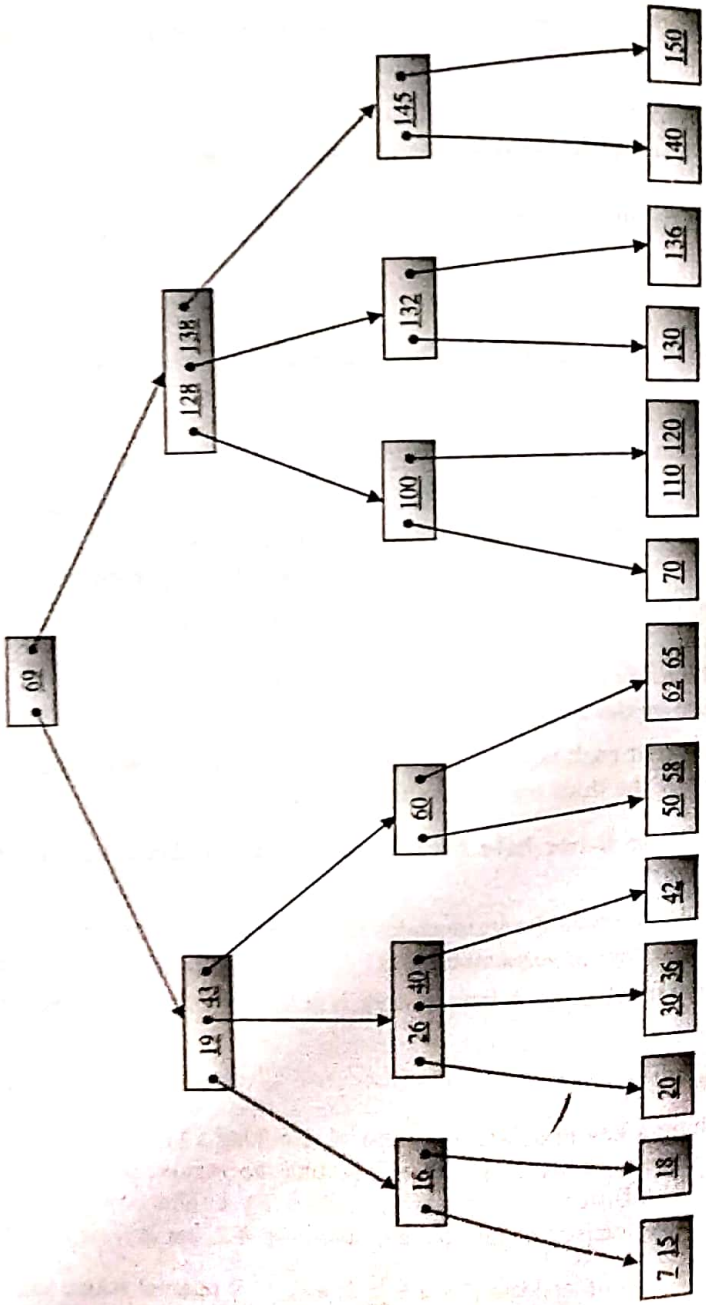


Figure 9.33 B-tree of order 3

Further, each internal node, except the root, must contain at least $\lceil m/2 \rceil - 1$ keys, and the root at each leaf must contain 1 element. Thus, the minimum number of keys in a B-tree is

$$n \geq 1 + \left(2 + \sum_{i=1}^{h-2} 2\lceil m/2 \rceil^{i-1}\right)\left(\lceil m/2 \rceil - 1\right) + 2\lceil m/2 \rceil^{h-1} \quad \text{which reduces to} \quad n + 1 \geq 2\lceil m/2 \rceil^{h-1}$$

That is, the maximum height of the tree is

$$h \leq 2 + \log_{\lceil m/2 \rceil}((n+1)/2) \qquad \text{for } m > 2$$

Recall that the leaves will appear at level h-1. Since the B-tree's height determines the maximum search length, we now know that the B-tree very well for direct searches.

9.10.2 Inserting into a B-tree

In order to insert a key into a B-tree, first a B-tree search is performed to find the correct location for the key. Then the key is inserted. Note that a node usually may have a space available for another key and pointer. However, the node may be already full and cannot accommodate another key. In that case, the node is split into two nodes, with half the key values and pointer going into one node while the other half going into another node.

These twin nodes have the same parent, which is modified for the additional key value and pointer. In the worst case, the splitting procedure may need to be carried all the way up the root of the tree. In that case, the height of the tree increases by one level. This insertion procedure is implemented as a recursion since the splitting procedure is the same, irrespective of the level.

Note that the probability of node splitting decreases as the order of the B-tree increases.

9.10.3 Deleting from a B-tree

Deletion of a key from a B-tree is slightly complicated than insertion. In order to keep the B-tree and B-tree, two nodes may need to be merged when one node contains less number of keys and pointers than the required keys and pointers.

As with insertion, first a B-tree search is performed to find the node containing the key to be deleted. If the key to be deleted is in the interior node, the tree is transformed to move this key into a leaf node because it is easier to delete a key from a leaf node. Once the deletion is made, the other keys may need to be shifted or leaves need to be merged to meet the minimum key and minimum pointer requirement for a B-tree. This shifting and merging may be needed for several levels.

If the merge percolates (bubbles) all the way to the root (highest level), a new root is formed and the height of the B-tree decreases by a factor of 1.

As with splitting, the probability, the probability of node merging decreases as the order of the B-tree increases.

9.11 B⁺-TREE

The B⁺-tree data structure is another variation on the basic B-tree. It is one of the popular techniques for implementing indexed file organization.

As you can see in Figure 9.33, the leaves have been connected to form a linked list of keys in sequential order. The B⁺-tree has two parts — the first part is the *index set* that constitutes interior nodes and the second part is the *sequence set* that constitutes leaves. The linked leaves are an excellent aspect of a B⁺-tree since using it keys can be accessed sequentially in addition to accessing them directly.

9.11.1 Inserting into a B⁺-tree

The procedure to insert a new key value into a B⁺-tree is almost same as for B-tree. When a leaf node is split into two nodes, a copy of the low-order key from the rightmost node is promoted to the separator key value in the parent node. The new node also must be inserted in the linked list of the sequence set.

9.11.2 Deleting from a B⁺-tree

Deletion of a key from a B⁺-tree is easier than B-tree. When a key value is deleted from a leaf there is no need to delete that key from the index set of the key. That key value still can direct searches to the proper leaves.

9.11.3 Searching a B⁺-tree

Note that a direct search must terminate in a node in the sequence set. If there is a key in the index set that match the sought key, the preceding pointer is followed until the correct leaf is reached. It is not necessary that every key in the index set must also appear in the sequence set since a key may have been deleted from the sequence set. A deleted key is retained in the index set for purposes of guiding access to the sequence set.

A B⁺-tree provides as good as performance for direct searching as does a B-tree of same order.

The maximum height of a B⁺-tree of order m with n key values is $\left(\log_{\lceil m/2 \rceil} \frac{n+1}{2}\right)$.

Sequential access to the data indexed by a B⁺-tree is provided by scanning the sequence set. Typically, as an indexed sequential file, the sequence set may contain data records in addition to keys. However, if data records are of variable length or relatively large, then the file is can only be structured so that sequence set contains pairs of key values and pointers (address) to the data records stored in secondary storage.

9.12 B*-TREE

The B*-tree data structure was identified by *Knuth* as one of the several possible variations of the basic B-tree. The main motivation behind the B*-tree was to reduce the overhead involved in insertions and deletions. And the other primary motive was to improve the search performance. The B*-tree is a B-tree in which each node is at least two-thirds full instead of

least half full. The basic idea of B*-tree is to reduce the node splitting. When a node becomes full, rather than being split, a redistribution scheme takes place and pointers from the full node and places them into a sibling node. The operations on B*-tree are performed in a manner similar to that of B-tree.

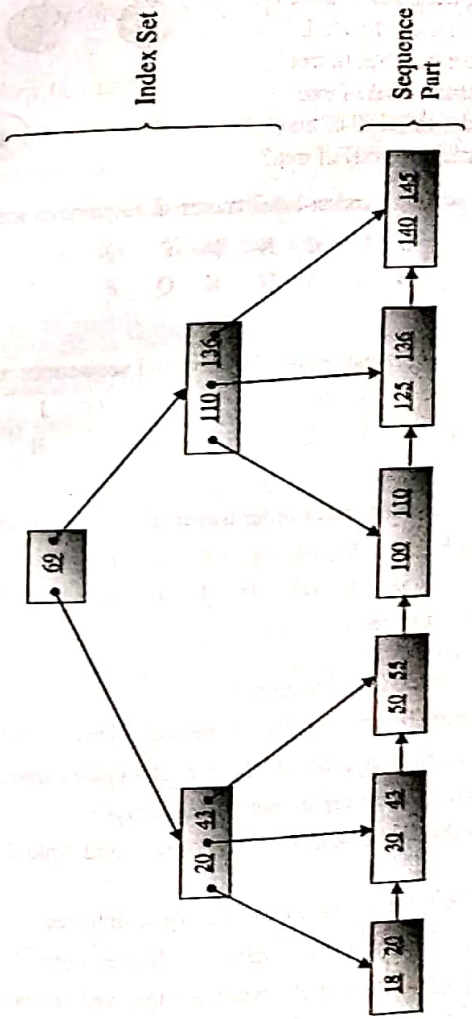
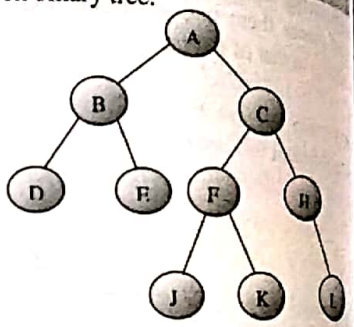


Figure 9.34 B⁺-tree of order 3

REVIEW EXERCISES

1. Answer the following questions in the context of given binary tree.



   (*a*) Which nodes are leave nodes?
   (*b*) Which is the root node?
   (*c*) What is the height of the tree?
   (*d*) Which nodes are non-leave nodes?
   (*e*) Which nodes are descendents of node C?
   (*f*) Which nodes are ancestors of node L?
   (*g*) What is the pre-order traversal of tree?
   (*h*) What is the in-order traversal of tree?
   (*i*) What is the post-order traversal of tree?
   (*j*) What is the level-order traversal of tree?

2. For a binary tree T, the pre-order and in-order traversal sequences are as follows:

   Pre-order : A B L M K N P Q
   In-order : L B M A N K Q P

   Draw the binary tree T.

3. For a binary tree T, the in-order and post-order traversal sequences are as follows:

   In-order : D C K E A H B Q J I
   Post-order : D K E C H Q J I B A

   Draw the binary tree T.

4. For a binary tree T, the pre-order and in-order traversal sequences are as follows:

   Pre-order : A B E H Q R C D K L M
   In-order : B Q R H E A D L K M C

   (*a*) What is the height of the tree?
   (*b*) What are the internal nodes?
   (*c*) What is its post-order traversal sequence?

5. What do you by a complete binary tree? Draw one such binary tree?
6. What is the maximum number of nodes at *k*th level of a binary tree?
7. Why binary trees, usually, are not represented using arrays?
8. Binary search tree combines the best worlds of arrays and linked lists into one structure. Comment.
9. Compare and contrast between a binary and a binary search tree.
10. Name the operations that are specific to binary search trees not to binary trees.
11. If same set of elements but in different order are inserted in two different empty binary search trees, will you get identical binary search trees?
12. When we insert a new node in a binary search tree, will it become an internal node or terminal node?

13. Can we insert a new node as an internal node?
14. How the height of a binary search tree effects its performance?
15. If there are 27 nodes in a complete binary tree, what will be its height? Also tell how many nodes will be in the last level.
16. What are the different operations that modify a tree?
17. Which of the operations on binary search tree is more complex and why?
18. How an AVL tree differs from a binary search tree? How AVL trees are represented in computer memory?
19. What is a thread? How is it useful?
20. What are the advantages of threading a binary (search) tree?
21. What is an *M-way* search tree?
22. What is the difference between a B-tree and a B^*-tree? How that difference matters?
23. What is the difference between a B-tree and a B^+-tree? When you might prefer to use B^+-tree instead of a B-tree?

PROGRAMMING EXERCISES

1. Write an iterative version of a function to search an element in a binary search tree.
2. We want to have only unique elements in a binary search tree. Give the modified version of *insert()* function.
3. Give the modified version of *search()* function so that it not only returns the location of the element but also return location of its parent.
4. Write a function to perform the level order traversal of a binary search tree.
5. Write a function to find the predecessor of a given element.
6. Write a function to find the successor of a given element.
7. Write iterative versions of functions to find the smallest and largest element in a binary search tree.
8. If the node structure of a binary search tree is modified to include an additional, say *father*, pointer to hold the address of a parent node as given below:

```
class Node {
  public:
     int info;
     Node *father, *left, *right;
};
```

   What are the operations that are required to be modified? Also show the modified version of these operations.
9. Write functions to implement the iterative versions of pre-order, in-order, and post-order traversals of a binary tree.

Heaps

10.1 INTRODUCTION

A *heap* is a binary tree that satisfies the following properties:

- Shape property
- Order property

By the *shape property* we mean that heap must be a complete binary tree, whereas by *order property* we mean that for every node in the heap, the value stored in that node is greater than or equal to the value in each of its children.

A heap that satisfies these properties is known as *max heap*.

However, if the order property is such that for every node in the heap, the value stored in that node is less than or equal to the value in each of its children, that heap is known as *min heap*.

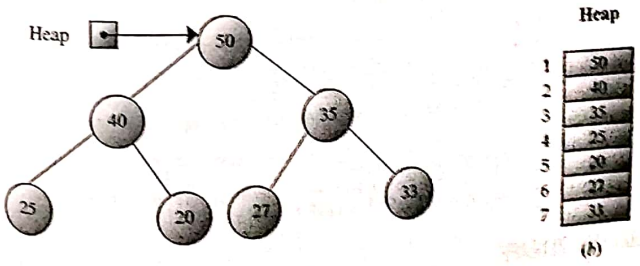


Figure 10.1: (*a*) Max heap of with 7 elements. (*b*) Sequential representation of Max heap

A heap is represented in memory using linear arrays, *i.e.*, by *sequential representation*. This chapter describes various operations on heaps, and its application in implementing priority queues and sorting a linear array. In our discussion, we will be considering max heaps.

10.2 REPRESENTING A HEAP IN MEMORY

Since a heap is a complete or nearly complete binary tree, therefore a heap of size n is represented in memory using a linear array of size n. This array representation is described in *Chapter 9*.

Following are required declarations

```
template <class T>
class MaxHeap
{
   private:
      T *heap;
      int n;
      int size;
   public:
      // constructor
      MaxHeap(int m)
      {
         heap = new T[m+1];
         n = 0;
         size = m;
      }
      // destructor
      ~MaxHeap()
      {
         delete heap;
      }
      // member function to build initial heap
      void heapify();
      // member function to insert an element into heap
      void insertElement(T item);
      // member function to delete element from root of heap
      T deleteElement();
      // member function to display the contents of heap
      void show();
      // member function to test whether the heap is empty
      int isEmpty();
      // member function to test whether the heap is full
      int isFull();
      // member function to readjust the heap after deletion
      void reheapifyDownward(int start, int finish);
      // member function to readjust the heap after insertion
      void reheapifyUpward(int start);
};
```

10.3 OPERATIONS ON HEAPS

On heaps, only two operations are performed. These are — *deleting an element from the heap* and *inserting a new element into the heap*. These are described below.

10.3.1 Deleting an Element From Heap

Element is always deleted from the root of the heap. If we delete the element from the root node, it creates a hole, *i.e.*, vacant space, in the root position. Because the heap must be complete, we fill the hole with the last (bottom) element of the heap.

Although the heap becomes complete, *i.e.*, satisfies shape property, the order property of the heaps is violated, as the values that come from the bottom are small. This situation suggests one of the basic operations:

"Given a complete binary tree whose elements satisfy the heap order property except in the root position, re-adjust the structure so that again it will be heap".

This operation involves moving the elements down from the root position until either it ends up in a position where the root property is satisfied or it hits the leaf node.

This operation in the text will be referred to as *reheapify downward* operation and works as follows:

Interchange the value of the root node with the value of the child, which is the largest among its children. For example, if the value of the left child is the largest, the root node is interchanged with its left child, and then the down heap property is repeated from its left child onward. Similar is the case if the right child is largest, start reheapify downward operation from that child with which the value of the root node was interchanged. This operation is recursive.

The various steps required for the reheapify downward operation are summarized in algorithm 10.1.

Algorithm 10.1:

ReheapifyDownward(heap, start, finish)

Here *heap* is a linear array, *start* is the index of the element from where reheapify downward operation is to start, and *finish* is index of the last (bottom) element of the heap. The variable *index* is used to keep track of the index of the largest child.

```
Begin
  if heap[start] is not a leaf node then
     Set index = index of the child with largest value
     if ( heap[start] < heap[index] ) then
        Swap heap[start] and heap[index]
        Call ReheapifyDownward( heap, index, finish )
     endif
  endif
End.
```

The implementation of the above algorithm is given in the following listing.

Listing 10.1:

```
// member function to readjust the heap after deletion
template <class T>
void MaxHeap<T>::reheapifyDownward(int start, int finish) {
   int  index, lchild, rchild;
   T maximum, temp;
   lchild = 2*start;          //  index of left child
   rchild = lchild + 1;             //  index of right child
   if ( lchild <= finish ) {
      maximum = heap[lchild];
      index = lchild;
      if ( rchild <= finish ) {
         if ( heap[rchild] > maximum ) {
            maximum = heap[rchild];
            index = rchild;
         }
      }
      if ( heap[start] < heap[index] ) {
         temp = heap[start];
         heap[start] = heap[index];
         heap[index] = temp;
         reheapifyDownward(index, finish);
      }
   }
}
```

Thus, deletion of an element from heap is done using the following steps:

- Assign the value of the root node to the temporary variable, which may be required for further processing,
- Bring the last element of the heap to the root node position,
- Reduce the size of the heap by a factor of one, and
- Apply reheapify downward operation from root node.

Algorithm 10.2:

DeleteElement(*heap*, *n*, *item*)

Here *heap* is the linear array with size *n*. This algorithm deletes the element from the root of *heap* and assign to *item* through which it is returned to the calling program. It also decreases its size by a factor of one, *i.e.*, size becomes *n*–1.

```
Begin
  Set item =  heap[1]
  Set heap[1] = heap[n]
  Set n = n-1
  Call ReheapifyDownward( heap, 1, n )
End.
```

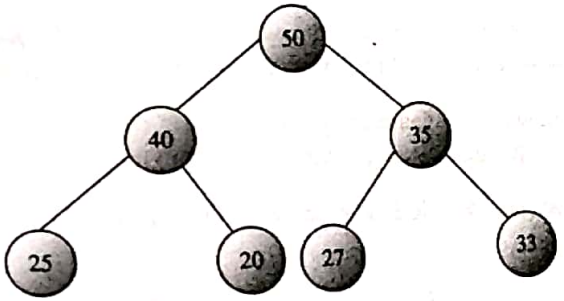

The implementation of the above algorithm is given in the following listing.

Listing 10.2:

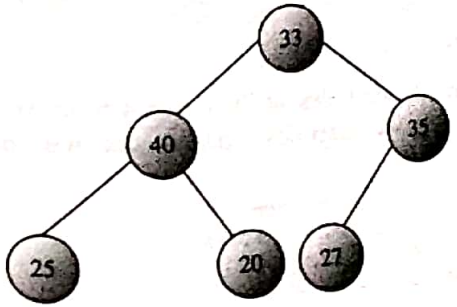
```
// member function to delete element from root of heap
template <class T>
int MaxHeap<T>::deleteElement()
{
  T temp;
  temp = heap[1];
  heap[1] = heap[n];
  n--;
  reheapifyDownward(1,n);
  return temp;
}
```

Example 10.1:

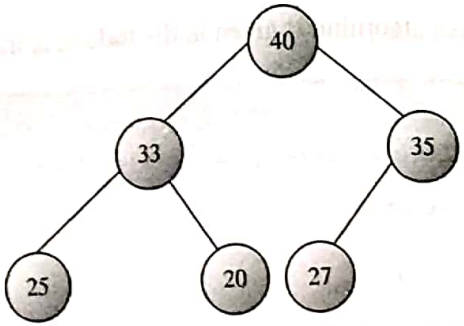
Following figure illustrates the deletion operation.



(*a*) Starting heap



(*b*) After Removing 50 and replacing with 33



(c) Reheapify downward from root node (swap 33 with 40)

Figure 10.2: Illustration of deletion and reheapify downward operation

10.3.2 Inserting an Element into Heap

Element is always inserted as last (bottom) child of the original heap. Although, with this insertion, the heap remains complete, but the order property may be violated if a larger element is inserted. This situation suggests another basic operation:

> *"Given a complete binary tree whose elements satisfy the heap order property except in the last (bottom) position, re-adjust the structure so that again it will be heap".*

This operation involves moving the elements up from the last (bottom) position until either it ends up in a position where the root property is satisfied or we hit the root node.

This operation in the text will be referred to as *reheapify upward* operation and works as follows:

> *If the value of the last node is greater than its parent, exchange it value with its parent, and then repeat the same process from the parent node, and so on until the order property is satisfied or we hit the root node. This operation is also recursive.*

The various steps required for *reheapify upward* operation are summarized in algorithm 10.3.

Algorithm 10.3:

ReheapifyUpward(***heap, start*** **)**

Here *heap* is a linear array, *start* is the index of the element from where reheapify upward operation is to start. It use *parent* as the index of the parent node in the heap.

```
Begin
   if heap[start] is not a root node then
      if ( heap[parent] < heap[start] ) then
         Swap heap[parent] and heap[start]
         Call ReheapifyUpward(heap, parent )
      endif
   endif
End.
```


The implementation of the above algorithm is given in the following listing.

Listing 10.3:

```
// member function to readjust the heap after insertion
template <class T>
void MaxHeap<T>::reheapifyUpward(int start)
{
  T temp, parent;
  if ( start > 1 ) {
     parent = start / 2;
     if ( heap[parent] < heap[start] ) {
        temp = heap[start];
        heap[start] = heap[parent];
        heap[parent] = temp;
        reheapifyUpward(parent);
     }
  }
}
```

Thus, insertion of an element into heap is done using the following steps:

- Increase the size of the heap by a factor of one,
- Insert the element as the last (bottom) element of the heap, and
- Apply *reheapify upward* operation from last node.

These steps summarized in algorithm 10.4.

Algorithm 10.4:

InsertElement(***heap, n, item*** **)**

Here *heap* is a linear array with size *n*. This algorithm inserts an element *item* into the *heap* and also increases its size by a factor of one, *i.e.*, size becomes *n*+1.

```
Begin
  Set n = n + 1
  Set heap[n] = item
  Call ReheapifyUpward(heap, n )
End.
```

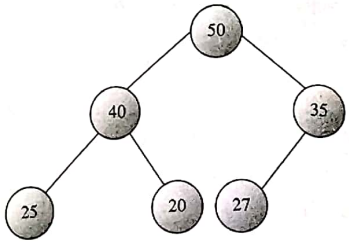
The implementation of the above algorithm is given in the following listing.

Listing 10.4:

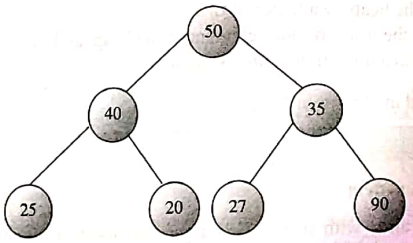
```
// member function to insert an element into heap
template <class T>
void MaxHeap<T>::insertElement(T item)
{
  n++;
  heap[n] = item;
  reheapifyUpward(n);
}
```

Example 10.2:

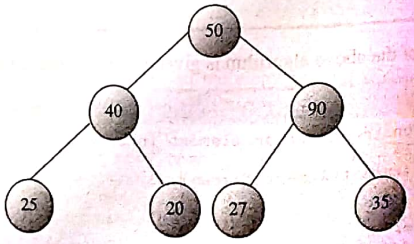
Following figure illustrates the insertion operation.



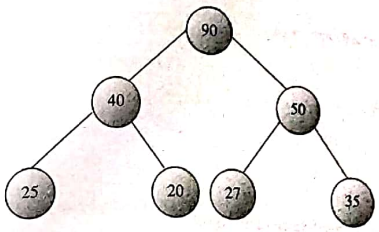
(*a*) Starting heap



(*b*) Add value 90



(*c*) Reheapify upward from last node with value 90



(*d*) Further reheapify upward from node with value 90

Figure 10.3: Illustration of insertion and reheapify upward operation

Listing 10.5:

```
// Program to demonstrate the implementation of operations on a
// Max Heap with element of type 'int'
#include <iostream.h>
#include <iomanip.h>
template <class T>
class MaxHeap
{
  private:
     T *heap;
     int n;
     int size;
  public:
     MaxHeap(int m) {
       heap = new T[m+1];
       n = 0;
       size = m;
     }
     ~MaxHeap() {
       delete heap;
     }
     void heapify();
     void insertElement(T item);
     T deleteElement();
     void show();
     int isEmpty();
     int isFull();
     void reheapifyDownward(int start, int finish);
     void reheapifyUpward(int start);
};
template <class T>
void MaxHeap<T>::insertElement(T item)
{
  n++;
  heap[n] = item;
  reheapifyUpward(n);
}
```

```
template <class T>
int MaxHeap<T>::deleteElement()
{
   T temp;
   temp = heap[1];
   heap[1] = heap[n];
   n--;
   reheapifyDownward(1,n);
   return temp;
}

template <class T>
T MaxHeap<T>::deleteElementFrom(int pos)
{
   T temp;
   temp = heap[pos];
   heap[pos] = heap[n];
   n--;
   reheapifyDownward(pos,n);
   return temp;
}

template <class T>
void MaxHeap<T>::show()
{
   int i;
   cout << "\n\nElements of Max Heap are \n\n";
   for ( i = 1; i <= n; i++ )
      cout << heap[i] << "   ";
   cout << endl;
}

template <class T>
int MaxHeap<T>::isEmpty()
{
   return ( ( n==0 ) ? 1 : 0 );
}

template <class T>
int MaxHeap<T>::isFull()
{
   return ( ( n == size ) ? 1 : 0 );
}

template <class T>
void MaxHeap<T>::reheapifyDownward(int start, int finish) {
   int  index, lchild, rchild;
   T maximum, temp;
   lchild = 2*start;          //  index of left child
   rchild = lchild + 1;              //  index of right child
   if ( lchild <= finish ) {
      maximum = heap[lchild];
      index = lchild;
      if ( rchild <= finish ) {
       if ( heap[rchild] > maximum ) {
          maximum = heap[rchild];
          index = rchild;
       }
      }
      if ( heap[start] < heap[index] ) {
       temp = heap[start];
       heap[start] = heap[index];
       heap[index] = temp;
       reheapifyDownward(index, finish);
      }
   }
}

template <class T>
void MaxHeap<T>::reheapifyUpward(int start)
{
   T temp, parent;
   if ( start > 1 ) {
      parent = start / 2;
      if ( heap[parent] < heap[start] ) {
        temp = heap[start];
        heap[start] = heap[parent];
        heap[parent] = temp;
        reheapifyUpward(parent);
      }
   }
}

template <class T>
void MaxHeap<T>::heapify()
{
   int i, index;
   index = n / 2;
   for ( i = index; i >= 1; i-- )
      reheapifyUpward(i);
}

void main()
{
   int choice, element, m;
   cout << "\nEnter size of the heap : ";
   cin >> m;

   MaxHeap<int> maxHeap(m);

   do
   {
      cout << "\n\n      Options available \n";
      cout << "     ++++++++++++++++++ \n\n";
      cout << "       1. Add Element  \n";
      cout << "       2. Delete Element from root \n";
      cout << "       3. Delete Element from given position \n";
      cout << "       4. Show heap \n";
      cout << "       5. Exit\n\n";
      cout << "Enter your choice ( 1-5 ) : ";
      cin >> choice;
      switch ( choice )
      {
        case 1 : if ( maxHeap.isFull() )
                 cout << "\nMax Heap already full ..";
```

```
                else {
                   cout << "\nEnter Element : ";
                   cin >> element;
                   maxHeap.insertElement(element);
                }
                break;
        case 2 : if ( maxHeap.isEmpty() )
                   cout << "\nMax Heap already empty ..";
                else {
                   cout << "Element removed is "
                        << maxHeap.deleteElement();
                }
                break;
        case 3 : cout << "\nEnter position from where to delete : ";
                cin >> m;
                if ( maxHeap.isEmpty() )
                   cout << "\nMax Heap already empty ..";
                else {
                   cout << "Element removed is "
                        << maxHeap.deleteElementFrom(m);
                }
                break;
        case 4 : maxHeap.show();
      }
   }
   while ( choice != 5 );
} //----  end of main function ----
```

10.4 APPLICATIONS OF HEAPS

The main applications of heaps are:

- Implementing a priority queue.
- Sorting an array using efficient technique known as *heapsort*.

The *heapsort* technique is described in *Chapter* 13.

10.4.1 Priority Queues

A priority queue is a structure with an interesting accessing function: Only the *highest-priority* element can be accessed. By highest priority we mean different things depending on the application. To illustrate this consider the following situations:

- A small company has only one secretary. When other employees leave work on the secretary's desk, which get job done first? The jobs are processed in order of the employee's importance in the company. Suppose, secretary has on his table work given by president and vice president, secretary completes president's work first. This example shows that the priority of each job relates to the level of the employee who initiated it.
- In a telephone answering system, calls are answered in the order that they are received. That is, the highest-priority call is the one that has been waiting the longest. Thus a FIFO queue can be considered a priority queue whose highest-priority element is the one that has been queued the longest time.

In this section, we will describe a priority queue whose *highest-priority* element is the one with the largest key value. We will assume that as the jobs enter the system, they are assigned a value according to its importance. The more important jobs are assigned larger values.

The operations defined for the priority queue are very much similar to the operation specified for FIFO queue.

Suppose priority queue *pq* with storage capacity *MAX* is defined, the various operations are implemented as described below.

Algorithm 10.5:

CreateEmptyPQ(*pq*, *n*)

Here *pq* is a linear array with size *n* representing a priority queue. This algorithm sets size equal to zero to indicate that there is no element in it.

```
Begin
  Set n = 0
End.
```

Algorithm 10.6:

IsEmptyPQ(*pq*, *n*, *status*)

Here *pq* is a linear array with size *n* representing a priority queue. This algorithm sets the variable *status* to value *true* if the size of the *pq* is zero, *i.e.*, it is empty; otherwise it sets *status* to value *false*.

```
Begin
  if ( n = 0 ) then
     Set status = true
  else
     Set status = false
  endif
End.
```

Algorithm 10.7:

IsFullPQ(*pq*, *n*, *status*)

Here *pq* is a linear array with size *n* representing a priority queue. This algorithm sets the variable *status* to value *true* if the size of the *pq* is MAX, *i.e.*, it is full; otherwise it sets *status* to value *false*.

```
Begin
```

```
  if ( n = MAX ) then
     Set status = true
  else
     Set status = false
  endif
End.
```

Algorithm 10.8:

EnqueuePQ(*pq, n, element*)

Here *pq* is a linear array with size *n* representing a priority queue. This algorithm inserts *element* into it if it is not full.

```
Begin
  Call IsFullPQ( pq, n, status )
  if ( status = true ) then
     Print: "Overflow"
  else
     Call InsertElement( pq, n, element )
  endif
End.
```

Algorithm 10.9:

DequeuePQ(*pq, n, element*)

Here *pq* is a linear array with size *n* representing a priority queue. This algorithm deletes the front element and stores in variable *element* into it if it is not empty.

```
Begin
  Call IsEmptyPQ( pq, n, status )
  if ( status = true ) then
     Print: "Underflow"
  else
     Call DeleteElement( pq, n, element )
  endif
End.
```

The following declarations, that use the *MaxHeap* class, can be used to implement a priority queue:

```
template <class T>
class PriorityQueue : MaxHeap<T>
{
    public:
       // constructor
       PriorityQueue(int m) : MaxHeap<T>(m) { }
       // destructor
       ~PriorityQueue() { }
       // member function that inserts a new element into priority
       // queue
       void enqueue(T item);
       // member function that removes an element from a queue
       T dequeue();
       // member function to test whether the priority
       // queue is empty
       int isEmpty();
       // member function to test whether the priority
       // queue is full
       int isFull();
       // member function to show the contents of the priority
       // queue
       void show();
};
// member function that inserts a new element into priority queue
template <class T>
void PriorityQueue<T>::enqueue(T item) {
   if ( isFull() )
       cout << "\nPriority Queue is full. . .\n";
   else
       insertElement(item);
}
// member function that removes an element from a queue
template <class T>
T PriorityQueue<T>::dequeue() {
   T temp;
   if ( isEmpty() )
       cout << "\nPriority Queue is empty. . .\n";
   else
       temp = MaxHeap<T>::deleteElement();
   return temp;
}
// member function to test whether the priority
// queue is empty
template <class T>
int PriorityQueue<T>::isEmpty() {
   return MaxHeap<T>::isEmpty();
}
// member function to test whether the priority
// queue is full
template <class T>
int PriorityQueue<T>::isFull()
{
   return MaxHeap<T>::isFull();
}
// member function to show the contents of priority queue
template <class T>
void PriorityQueue<T>::show()
{
   if ( isEmpty() )
      cout << "\nPriority Queue is empty ...";
   else {
      cout << "\nElements of Priority Queue are : ";
      MaxHeap<T>::show();
   }
}
```

Listing 10.6:

```
//  Program to demonstrate the implementation of priority queue
#include <iostream.h>
#include <iomanip.h>
template <class T>
class MaxHeap {
   private:
      T *heap;
      int n;
      int size;
   public:
      MaxHeap(int m) {
        heap = new T[m+1];
        n = 0;
        size = m;
      }
      ~MaxHeap() {
        delete heap;
      }
      void heapify();
      void insertElement(T item);
      T deleteElement();
      void show();
      int isEmpty();
      int isFull();
      void reheapifyDownward(int start, int finish);
      void reheapifyUpward(int start);
};
template <class T>
void MaxHeap<T>::insertElement(T item) {
   n++;
   heap[n] = item;
   reheapifyUpward(n);
}
template <class T>
int MaxHeap<T>::deleteElement() {
   T temp;
   temp = heap[1];
   heap[1] = heap[n];
   n--;
   reheapifyDownward(1,n);
   return temp;
}
template <class T>
void MaxHeap<T>::show() {
   int i;
   for ( i = 1; i <= n; i++ )
      cout << heap[i] << "   ";
   cout << endl;
}
template <class T>
int MaxHeap<T>::isEmpty() {
   return ( ( n==0 ) ? 1 : 0 );
}
template <class T>
int MaxHeap<T>::isFull() {
   return ( ( n == size ) ? 1 : 0 );
}
template <class T>
void MaxHeap<T>::reheapifyDownward(int start, int finish) {
  int  index, lchild, rchild;
  T maximum, temp;
  lchild = 2*start;          //  index of left child
  rchild = lchild + 1;             //  index of right child
  if ( lchild <= finish ) {
     maximum = heap[lchild];
     index = lchild;
     if ( rchild <= finish ) {
        if ( heap[rchild] > maximum ) {
           maximum = heap[rchild];
           index = rchild;
        }
     }
     if ( heap[start] < heap[index] ) {
        temp = heap[start];
        heap[start] = heap[index];
        heap[index] = temp;
        reheapifyDownward(index, finish);
     }
  }
}
template <class T>
void MaxHeap<T>::reheapifyUpward(int start)
{
  T temp, parent;
  if ( start > 1 ) {
      parent = start / 2;
      if ( heap[parent] < heap[start] ) {
        temp = heap[start];
        heap[start] = heap[parent];
        heap[parent] = temp;
        reheapifyUpward(parent);
      }
  }
}
template <class T>
void MaxHeap<T>::heapify() {
  int i, index;
  index = n / 2;
  for ( i = index; i >= 1; i-- )
      reheapifyUpward(i);
}
template <class T>
class PriorityQueue : MaxHeap<T>
{
   public:
      // constructor
      PriorityQueue(int m) : MaxHeap<T>(m) { }
     // destructor
      ~PriorityQueue() { }
    // member function that inserts a new element into priority queue
      void enqueue(T item);
    // member function that removes an element from a queue
      T dequeue();
```

```
    // member function to test whether the priority
     // queue is empty
     int isEmpty();
    // member function to test whether the priority
     // queue is full
     int isFull();
     // member function to show the contents of the priority queue
     void show();
};
// member function that inserts a new element into priority
// queue
template <class T>
void PriorityQueue<T>::enqueue(T item) {
   if ( isFull() )
       cout << "\nPriority Queue is full. . .\n";
   else
       insertElement(item);
}
// member function that removes an element from a queue
template <class T>
T PriorityQueue<T>::dequeue() {
   T temp;
   if ( isEmpty() )
       cout << "\nPriority Queue is empty. . .\n";
   else
       temp = MaxHeap<T>::deleteElement();
   return temp;
}
// member function to test whether the priority
// queue is empty
template <class T>
int PriorityQueue<T>::isEmpty()
{
   return MaxHeap<T>::isEmpty();
}
// member function to test whether the priority
// queue is full
template <class T>
int PriorityQueue<T>::isFull() {
   return MaxHeap<T>::isFull();
}
// member function to show the contents of priority queue
template <class T>
void PriorityQueue<T>::show() {
   if ( isEmpty() )
      cout << "\nPriority Queue is empty ...";
   else {
      cout << "\nElements of Priority Queue are : ";
      MaxHeap<T>::show();
   }
}
void main()
{
   int choice, element, m;
   cout << "\nEnter size of the Priority Queue : ";
   cin >> m;
   PriorityQueue<int> pq(m);
   do
```



```
{
   cout << "\n\n      Options available \n";
   cout << "     ++++++++++++++++++ \n\n";
   cout << "       1. Insert Element  \n";
   cout << "       2. Remove Element \n";
   cout << "       3. Show Priority Queue \n";
   cout << "       4. Exit\n\n";
   cout << "Enter your choice ( 1-4 ) : ";
   cin >> choice;
   switch ( choice )
   {
      case 1 : if ( pq.isFull() )
                  cout << "\nPriority Queue is already full ..";
               else {
                  cout << "\nEnter Element : ";
                  cin >> element;
                  pq.enqueue(element);
               }
               break;
      case 2 : if ( pq.isEmpty() )
                  cout << "\nPriority Queue is already empty ..";
               else {
                  cout << "Element removed is "
                       << pq.dequeue();
               }
               break;
      case 3 : pq.show();
   }
}
while ( choice != 4 );
//----  end of main function ----
```

10.4.2 Sorting an Array Using Heapsort

Because of the order property, the maximum value of a *max heap* is in the root node. We can take advantage of this situation by using a heap to help us to sort the given array a having n elements.

The general approach of heap sort is as follows:

1. From the given array, build the initial max heap.
2. Interchange the root (maximum) element with the last element.
3. Use *reheapifydownward* operation from root node to rebuild the heap of size one less than the starting size.
4. Repeat steps 1 and 2 until there are no more elements.

Step 1 looks very much similar to selection sort, but the step that makes the heap sort fast is step 2 – *finding the next largest element*. As the shape property of a heap guarantees a binary tree of minimum height, therefore step 2 makes only $O(\log_2 n)$ comparisons in each iteration.

10.4.2.1 Building a Heap

To have a look at how the heap relates to array of unsorted elements. The Figure 10.4 shows unsorted array and its equivalent binary tree.

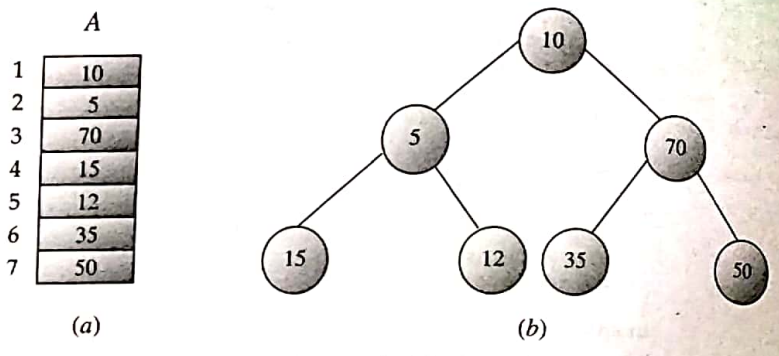
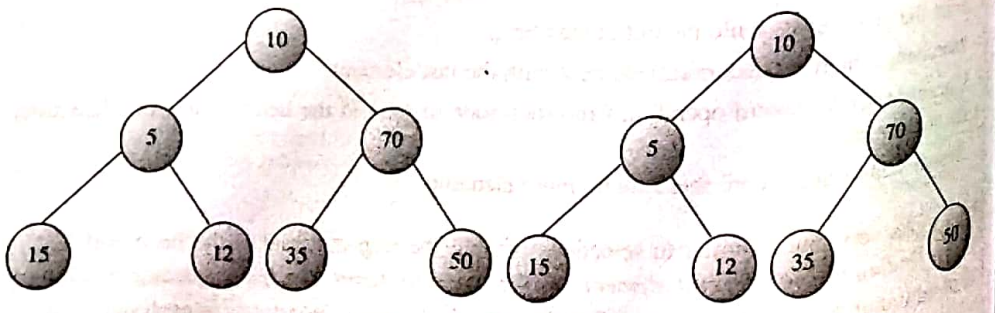


Figure 10.4: (*a*) Unsorted array *A*, (*b*) Equivalent binary tree

We want that the elements of the unsorted array must satisfy the order property of max heap. Let us first have a look at the binary tree to see that is there any part of the binary tree that already satisfies the order property. Observe that all the leaf nodes (subtrees with only single node) are heaps. In Figure 10.4(*b*), the subtrees whose roots contain values 15, 12, 35, and 50 are heaps because they are root nodes.

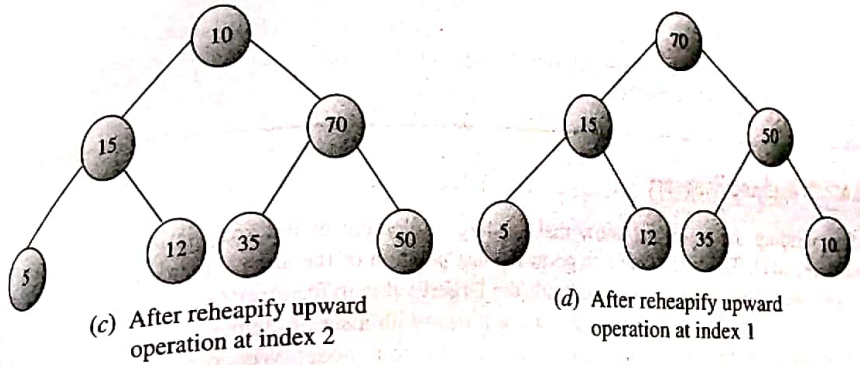
Now let us look at the first leftmost non-leaf node, the one containing value 5. The subtree rooted at this node is not a heap, and it is the root of the subtree that is heap. And we have already seen how to fix this problem using reheapify upward operation from this node. We apply reheapify upward operation to all the subtrees on this level, then we move up a level in the tree and continue until we reach the root node. After the reheapify upward operation is applied to the root node, the whole tree will satisfy the order property.

The heap building process, also popularly known as *heapify*, is illustrated for the given unsorted array *A* in Figure 10.4 is illustrated below:



(*a*) Starting binary tree

(*b*) After reheapify upward operation at index 3



(*c*) After reheapify upward operation at index 2

(*d*) After reheapify upward operation at index 1

Figure 10.5: Illustration of heap building process

The binary tree of Figure 10.5(*d*) represents a *max* heap.

	a[1]	*a*[2]	*a*[3]	*a*[4]	*a*[5]	*a*[6]	*a*[7]
Original array A	10	5	70	15	12	35	50
After reheapify upward							
From index 3	10	5	70	15	12	35	50
From index 2	10	15	70	5	12	35	50
From index 1	70	15	50	5	12	35	10

Figure 10.6: Illustration of changing contents of array *A*

The various steps required are summarized in the following algorithm shown below.

Algorithm 10.10:

Heapify(*a*, *n*)

Here *a* is the linear array with size *n*. This algorithm builds the max heap using the procedure described above.

```
Begin
  Set index = Parent of node with index n
  for i = index  to 1 by -1 do
    Call ReheapifyUpward( a, i, n )
  endfor
End.
```

The implementation of the above algorithm is given in the following listing.

Listing 10.7:

```
template <class T>
void MaxHeap<T>::heapify()
{
```

```
int i, index;
index = 2 - 2;
for ( i = index; i >= 1; i-- )
    reheapifyDpward(i);
```

10.4.2.2 Applying Heapsort

After building the heap, we know that the largest element of the array is in root node of the heap, *i.e.*, $a[1]$. This value have to go to the last position of the array, *i.e.*, $a[n]$, so these values are swapped. Because $a[n]$ now contains the largest value in the array (its correct position), this position is not touched further. Now, we will deal with a set of elements from $a[1]$ to $a[n-1]$, which satisfies the order property except at the root node. We correct this by applying *reheapifydownward* operation.

At this point, we know that the next largest element in the array is in the root node of the heap. To put this element in the correct position in the array, we swap it with the element $a[n-1]$. Now two largest elements are in their final position in the array, and we are left dealing with elements from $a[1]$ to $a[n-2]$. This process is repeated until all the elements are in their correct positions, *i.e.*, until heap contains only single element, which is the smallest element in the array.

The steps required to sort an array a of unsorted element in ascending order are summarized in the following algorithm.

Algorithm 10.11:

HeapSort(*a*, *n*)

Here a is a linear array of size n in memory. This algorithm sorts this array in ascending order using Heap sort method.

```
Begin
  Call Heapify( a, n )
  for i = n to 2 by -1 do
      Interchange elements a[1] and a[i]
      Call Reheapifydownward( a, 1, i-1 )
  endfor
End.
```

Implementation of the above algorithm is given below:

Listing 10.8:

```
template <class T>
void Array<T>::heapSort()
{
   int i, temp;
   heapify();
```



```
for ( i = n; i > 1; i-- ) {
   temp = a[1];
   a[1] = a[i];
   a[i] = temp;
   reheapifydownward( a, 1, i-1 );
}
```

Example 10.3:

Let us consider a working example to illustrate the working of heapsort algorithm. For this purpose consider the following array

10, 5, 70, 15, 12, 35, 50

Solution:

	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$	$a[7]$
After heapify operation	70	15	50	5	12	35	10
Swap	10	15	50	5	12	35	70
Reheapifydownward	50	15	35	5	12	10	70
Swap	10	15	35	5	12	50	70
Reheapifydownward	35	15	10	5	12	50	70
Swap	12	15	10	5	35	50	70
Reheapifydownward	15	12	10	5	35	50	70
Swap	5	12	10	15	35	50	70
Reheapifydownward	12	5	10	15	35	50	70
Swap	10	5	12	15	35	50	70
Reheapifydownward	10	5	12	15	35	50	70
Swap	5	10	12	15	35	50	70
Final sorted array	5	10	12	15	35	50	70

Figure 10.7: Illustration of effect of heapsort on array a

Analysis of Heap Sort

In the heapsort algorithm, we call function *heapify()* to build the initial heap, which is $O(\log n)$ operation, and inside the loop we call utility function *reheapifydownward()*. The loop is executed $(n-1)$ times, and in each iteration, the element in the root node of the heap is swapped

with the last element of the reduced size heap and heap is rebuild. Swapping two elements takes constant time. A complete binary tree with n nodes has $O(\log_2(n-1))$ levels. In the worst case, if the root element is bumped down to a leaf position, the *reheapifydownward* operation would perform $O(\log_2 n)$ swaps. So the swap plus *reheapifydownward* is $O(\log_2 n)$. Multiplying for activity by $(n-1)$ iteration shows that the sorting loop is $O(n\log_2 n)$.

Combining the heap build, which is $O(n)$, and the sorting loop, we can see that complexity of *heapSort* is $O(n\log_2 n)$.

Listing 10.8:

```
//   Program to sort an array of integers in ascending
//   order using heap sort method

#include<iostream.h>
#include<iomanip.h>

template <class T>
class Array
{
   private:
      T *a;
      int size;
   public:
      Array(int n) {
         size = n;
         a = new T[n+1];
      }
      ~Array() {
         delete a;
      }
      void inputArray();
      void heapify();
      void reheapifydownward(int, int);
      void heapSort();
      void outputArray();
};
template <class T>
void Array<T>::inputArray()
{
   int i;
   cout << "\nEnter " << size << " elements of array\n\n";
   for ( i = 1; i <= size; i++ ) {
      cin >> a[i];
   }
}
template <class T>
void Array<T>::heapify()
{
   int i, index;
   index = size / 2;
   for ( i = index; i >= 1; i-- )
       reheapifydownward(i, size );
}
```



```
template <class T>
void Array<T>::reheapifydownward(int start, int finish)
{
   int index, lchild, rchild, maximum, temp;
   lchild = 2*start;
   rchild = lchild + 1;
   while ( lchild <= finish ) {
      maximum = a[lchild];
      index = lchild;
      if ( rchild <= finish ) {
         if ( a[rchild] > maximum ) {
            maximum = a[rchild];
            index = rchild;
         }
      }
      if ( a[start] < a[index] ) {
         temp = a[start];
         a[start] = a[index];
         a[index] = temp;
         reheapifydownward(index, finish);
      }
   }
}
template <class T>
void Array<T>::heapSort()
{
   int i, temp;
   heapify();
   for ( i = size; i > 1; i-- ) {
      temp = a[1];
      a[1] = a[i];
      a[i] = temp;
      reheapifydownward(1, i-1);
   }
}
template <class T>
void Array<T>::outputArray()
{
   int i;
   for ( i = 1; i <= size; i++ ) {
      cout << a[i] << "\t";
   }
   cout << endl;
}
void main()
{
   int n;
   cout << "Enter size of array : ";
   cin >> n;
   Array<int> a(n);
   a.inputArray();
   cout << "\n\nInput array is : ";
   a.outputArray();
   a.heapSort();
   cout << "\n\nSorted array is : ";
   a.outputArray();
}
```

REVIEW EXERCISES

1. What is heap? What is the difference between a min heap and max heap?
2. Show the implementation of *heapify()* operation for a *min heap*.
3. Show the implementation of delete operation that can delete an element from a heap from an arbitrary index, not necessarily from root of the heap.
4. Does the sequence <23, 17, 14, 6, 13, 10, 1, 5, 7, 12> represents a max heap?
5. Does the sequence <3, 7, 4, 6, 13, 10, 1, 5, 7, 1> represents a min heap?
6. The sequence <70, 15, 50, 5, 12, 35, 10> represents a max heap. Show the heap after the following operations:
   (*a*) Insertion of element 25
   (*b*) Deletion from root
   (*c*) Insertion of element 75

   Also give the number of exchanges performed in each case.
7. Illustrate the operation *heapify()* on the sequence <13, 10, 5, 7, 22, 11, 9, 18> in order to build a *min heap*.
8. How a heap differs from a binary tree?
9. How a heap can be used to implement a priority queue?

PROGRAMMING EXERCISES

1. Write functions for implementing *reheapify upward()* and *reheapify upward()* operations on a *min* heap.
2. Write a function to delete an item at index *i* from a heap.

Chapter 11

Graphs

11.1 INTRODUCTION

Graph is another important non-linear data structure. This data structure is used to represent relationship between pairs of elements, which are not necessarily hierarchical in nature. A *graph* is defines as

"Graph G is a ordered set (V, E), where V(G) represent the set of elements, called vertices, and E(G) represents the edges between these vertices".

Figure 11.1 shows a sample graph, for which $V(G) = \{ v_1, v_2, v_3, v_4, v_5\}$, and $E(G) = \{e_1, e_2, e_3, e_4, e_5\}$ i.e. there are five vertices and five edges in the graph.

A graph can be either *directed* or *undirected*. In an undirected graph, there is no specific direction associated with the edges. In general, in an undirected graph, an edges is represented by an *unordered* pair $[u, v]$ (i.e. $e = [u, v]$) , and that can be traversed from u toward v or vice versa. Figure 11.1 depicts an undirected graph. Whereas in a directed graph, an edge is represented by an *ordered* pair (u, v) (i.e. $e = (u, v)$), and that can be traversed only from u toward v. Figure 11.2 depicts a directed graph.

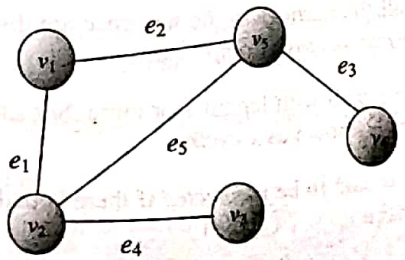


Figure 11.1: Undirected graph

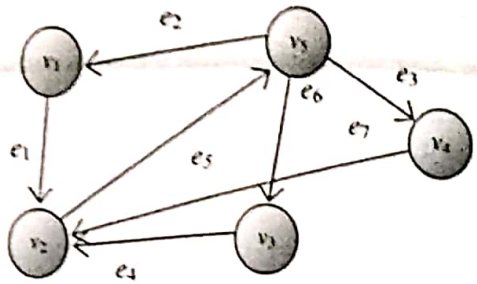


Figure 11.2: Directed graph

Before we move on to describe the storage of graphs in computer memory, and subsequent operation on these graphs, let us first review important terms in the graph terminology, which will be used, in remaining text.

11.2 GRAPH TERMINOLOGY

First we will review the terminology applicable to undirected graphs.

Adjacent Vertices – As an edge e is represented by pair of vertices denoted by $[u, v]$. The vertices u and v are called endpoints of e. These vertices are also called adjacent vertices or neighbors.

Degree of a Vertex – The degree of vertex u, written as $deg(u)$, is the number of edges containing u. If $deg(u)=0$, this means that vertex u does not belong to any edge, then vertex u is called an *isolated* vertex.

Path – A *path* P of length n from a vertex u to vertex v is defined as sequence of $(n+1)$ vertices, *i.e.*

$$P = (v_1, v_2, v_3, \ldots v_{n+1})$$

such that $u = v_1$, $v = v_{n+1}$, and v_{i-1} is adjacent to v_i for $i = 2, 3, \ldots, (n+1)$.

The path is said to be *closed* if the endpoints of the path are same i.e. $v_1 = v_{n+1}$.

The path is said to be *simple* if all the vertices in the sequence are distinct, with the exception that $v_1 = v_{n+1}$. In that case it is known as *closed simple* path.

Cycle – A *cycle* is a closed simple path with length 2 or more. Sometimes, a cycle of length k (i.e. k distinct vertices in the path) is known as *k-cycle*.

Connected Graph – A graph G is said to be *connected* if there is path between any two of its vertices, *i.e.*, there is no isolated vertex.

A connected graph without any cycles is called a *tree*. Thus we can say that tree is a special graph.

Complete Graph – A graph G is said to *complete* or *fully connected* if there is path from every vertex to every other vertex. A complete graph with n vertices will have $n(n-1)/2$ edges.

Weighted Graph – A graph is said to *weighted graph* if every edge in the graph is assigned some data. The weight of the, denoted by $w(e)$, is a non-negative value that may be representing the cost of moving along that edge or distance between the vertices.

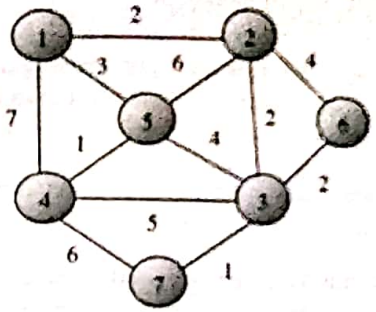


Figure 11.3: Weighted undirected graph

Multiple Edges – Distinct edges e and e' are called *multiple edges* if they connect the same endpoints, *i.e.*, if $e = [u, v]$ and $e' = [u, v]$.

Multigraph – A graph containing multiple edges.

Loop – An edge is a *loop* if it has identical endpoints, *i.e.*, if $e = [u, u]$.

Standard definition of graph usually does not permit either multiple edges or loops.

The following terms have relevance to directed graphs only.

A *directed graph* G is a graph in which each edge is assigned a certain direction, *i.e.*, each edge is an ordered pair of (u,v) of vertices rather than an unordered pair $[u, v]$.

Suppose G is a directed graph with $e = (u, v)$ as one of the edges, then

- e *begins* at u and *ends* at v,
- u is the *origin* (also known as *initial point*) of e, and v is the *destination* (also known as *terminal point*) of e,
- u is the *predecessor* of v, and v is the *successor* of u,

Outdegree and Indegree of a Vertex – The *outdegree* of a vertex u, denoted by $outdeg(u)$, is the number of edges originating at u. Similarly, The *indegree* of a vertex u, denoted by $indeg(u)$, is the number of edges terminating at u.

Source and Sink – A Vertex u is called a *source* if it has a outdegree greater than zero, but zero indegree. Similarly, a vertex u is called *sink* if it has indegree greater than zero, but zero outdegree.

Reachability – A vertex v is said to be reachable from u if there exists a path from vertex u to vertex v.

Strongly Connected – A directed graph G is said to be *strongly connected*, if for each pair u, v of vertices in G, if there exists a path from u to v, there must exist a path from v to u; otherwise the directed graph is *unilaterally connected*.

Parallel Edges – Distinct edges e and e' are called *parallel edges* if they connect the same source and terminal vertices, i.e. if $e = (u, v)$ and $e' = (u, v)$.

Simple Directed Graph – A directed graph G is said to be simple, if there are no parallel edges. A simple directed graph may have loops, but it can't have more than one loop at a given vertex.

Directed Acyclic Graph – A directed acyclic graph (DAG) G is a directed graph without cycle(s).

11.3 REPRESENTATION OF GRAPHS

There are two ways to represent a graph $G = (V, E)$:

- using an adjacency matrix
- using an adjacency list

The adjacency list representation is usually preferred because it provides a compact way to represent *sparse* graphs. A sparse graph is that for which $|E|$ is much less than $|V|^2$. When the graph is *dense* graph, we may prefer adjacency matrix representation. A dense graph is that for which $|E|$ is close to $|V|^2$ and we need to tell very quickly if there is an edge connecting given two vertices.

11.3.1 Adjacency Matrix Representation

Consider a directed graph $G = (V, E)$. We will assume that the vertices are numbered 1, 2, 3, ..., $|V|$, in some arbitrary manner. The *adjacency matrix* representation of a graph G then consists of $|V| \times |V|$ matrix $A = (a_{ij})$, such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

For undirected graph $G = (V, E)$, the adjacency matrix representation is also consists of $|V| \times |V|$ matrix $A = (a_{ij})$, but its elements are as follows:

$$a_{ij} = \begin{cases} 1 & \text{if either } [i, j] \in E \text{ or } [j, i] \in E \\ 0 & \text{otherwise} \end{cases}$$

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	0	1
3	0	1	0	0	0
4	0	0	0	0	1
5	1	1	0	1	0

(*a*) Adjacency matrix for undirected graph of Figure 11.1

	1	2	3	4	5
1	0	1	0	0	0
2	0	0	0	0	1
3	0	1	0	0	0
4	0	1	0	0	0
5	1	0	1	1	0

(*b*) Adjacency matrix for directed graph of Figure 11.2

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

(*c*) Adjacency matrix for weighted graph of Figure 11.3

Figure 11.4: Adjacency matrix representation of graphs

A matrix, like adjacency matrix for non-weighted graphs that contains entries of only 0 and 1 is called *bit matrix* or a *boolean matrix*.

The adjacency matrix representation of a graph requires $O(V^2)$ memory locations irrespective of the number of edges in the graph.

Observe the symmetry of the adjacency matrix of directed graph in figure 11.4(a), as it is store only the entries on or above of the diagonal of the adjacency matrix, we can reduce the memory requirements to store the graph to almost half.

To represent weighted graphs using adjacency matrix, the weight of edge (u, v) is simply stored as the entry in row u and column v of the adjacency matrix. In case, zero can also be a possible weight of the edge, then we have to store some sentinel value for non-existent edges which can be a negative value since the weight of the edge is always a positive number.

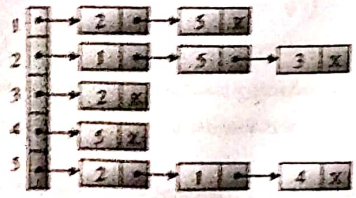
11.3.2 Adjacency List Representation

The adjacency-list representation of a graph $G = (V, E)$ consists of an array Adj of $|V|$ lists, one for each vertex in V. For each $u \in V$, the adjacency list $Adj[u]$ contains all the vertices v such that there is an edge $(u, v) \in E$ i.e. $Adj[u]$ consists of all the vertices adjacent to u in G. The vertices in each adjacency list are stored in an arbitrary order.

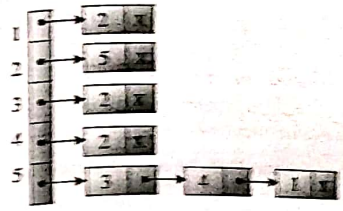
If G is an undirected graph, the sum of the lengths of all the adjacency lists is $2|E|$, since for every edge (u, v), u appears in v's adjacency list (i.e. $u \in Adj[v]$), and v appears in u's adjacency list (i.e. $v \in Adj[u]$). If G is a directed graph, the sum of the lengths of all the adjacency lists is $|E|$, since edges of the form (u, v) is represented by having v appearing in u's adjacency list (i.e. $v \in Adj[u]$).

Adjacency lists can be easily modified to represent weighted graphs. The weight of edge $e = (u, v)$, denoted by $w(e)$ or $w(u, v)$, is simply stored with the vertex v in $Adj[u]$ i.e. the information part of the vertex will have two fields – *one to store vertex and the second to store the weight of the edge.*

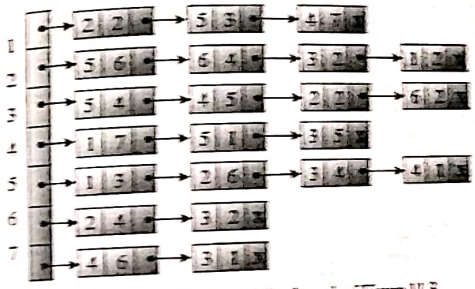
Although the adjacency list representation requires very less memory as compared to the adjacency matrix, the simplicity of adjacency matrix makes it preferable when graphs are reasonably small. Moreover, if the graph is not weighted one, there is additional advantage to the adjacency matrix representation. Rather than using one byte/word (in case of PC, to store an integer value in the range of 1–127 we need one byte, in range 1–65535, we need two bytes etc.) for each entry of the matrix, the adjacency matrix uses only one bit per entry.



(a) Adjacency-list for undirected graph of Figure 11.1



(b) Adjacency-list for undirected graph of Figure 11.2



(c) Adjacency-list for weighted graph of Figure 11.3

Figure 11.5: Adjacency-list representation of graphs

11.3.3 Implementation of Adjacency List in C++ Language

Let us first consider non-weighted graph with number of vertices MAX, where MAX is defined equal to 50, say, then the following declaration will be required

```
#define  MAX   20
struct NWNode {                    // ---- node declaration ----
   int vertex;
   NWNode *next;
};
class NonWeightedGraph     // class modelling weighted graph
{
   protected:
      NWNode *adj[MAX];
      int size;
   public:
      NonWeightedGraph(int n);
      ~NonWeightedGraph();
      void addEdge(int i, int j, int wt);
      void deleteEdge(int i, int j);
      void inputGraph();
      void printGraph();
```

```
        void BFS(int s);
        void DFSIterative(int s);
        void DFSRecursive(int s);
        void DFSVisit(int u);
        void DFSRecursiveModified();
        void DFSVisitModified(int u);
        void topologicalSort();
};  //    ---- end of class template ----
```

For weighted graph with number of vertices MAX, following declaration will be required:

```
#define  MAX   20
struct Wnode {          // --- node declaration ---
   int vertex;
   int vertex;
   WNode *next;
};
class WeightedGraph {   // --- class modelling weighted graph
   protected:
      WNode *adj[MAX];
      int size;
   public:
      WeightedGraph(int n);
      ~WeightedGraph();
      void addEdge(int i, int j, int w);
      void deleteEdge(int i, int j);
      void inputGraph();
      void printGraph();
      void BFS(int s);
      void DFSIterative(int s);
      void DFSRecursive(int s);
      void DFSVisit(int u);
      void DFSRecursiveModified();
      void DFSVisitModified(int u);
      void topologicalSort();
      void MinimumSpanningTreePrim();
      void shortestPathDijkstra(int s, int d);
};  //    ---- end of class template ----
```

11.4 OPERATIONS ON GRAPHS

Unless or otherwise stated, most of the algorithms developed assumes that the input graph is represented in adjacency-list form.

11.4.1 Creating an Empty Graph

To create an empty graph, the entire adjacency list is set to NULL. This task is accomplished by the constructor of the appropriate class as shown below:

Listing 11.1:

```
//  parameterized constructor to create non-weighted graph
NonWeightedGraph::NonWeightedGraph(int n)
{
   int i;
   for ( i = 1; i <= n; i++ )
```



```
      adj[i] = NULL;
   size = n;
}
```

Listing 11.2 :

```
// parameterized constructor to create weighted graph
WeightedGraph::WeightedGraph(int n) {
   int i;
   for ( i = 1; i <= n; i++ )
      adj[i] = NULL;
   size = n;
}
```

11.4.2 Entering Graph Information

The graph information is entered as shown in the following listing.

Listing 11.3:

```
// member function to input a non-weighted graph
void NonWeightedGraph::inputGraph()
{
   NWNode *ptr, *last;
   int i, j, m, val;
   for ( i = 1; i <= size; i++ ) {
      last = NULL;
      cout << "\nNumber of nodes in the adjacency"
           << " list of node #" << i << " : ";
      cin >> m;
      for ( j = 1; j <= m; j++ ) {
         cout << "Enter node #" << j << " : ";
         cin >> val;
         ptr = new NWNode;
         ptr->vertex = val;
         ptr->next = NULL;
         if ( adj[i] == NULL )
            adj[i] = last = ptr;
         else {
            last->next = ptr;
            last = ptr;
         }
      }
   }
}
```

Listing 11.4:

```
// member function to input a weighted graph
void WeightedGraph::inputGraph()
{
   WNode *ptr, *last;
   int i, j, m, val, wt;
   for ( i = 1; i <= size; i++ ) {
```

```
        last = NULL;
        cout << "\nNumber of nodes in the adjacency"
             << " list of node #" << i << " : ";
        cin >> m;
        for ( j = 1; j <= m; j++ ) {
           cout << "Enter node #" << j << " : ";
           cin >> val;
           cout << "Enter weight for edge ("
                << i << "," << val << ") : ";
           cin >> wt;
           ptr = new WNode;
           ptr->vertex = val;
           ptr->weight = wt;
           ptr->next = NULL;
           if ( adj[i] == NULL )
               adj[i] = last = ptr;
           else {
               last->next = ptr;
               last = ptr;
           }
        }
     }
}
```

11.4.3 Outputting a Graph

The following listing shows the way graph information can be outputted.

Listing 11.5:

```
// member function to output a non-weighted graph
void NonWeightedGraph::printGraph() {
  NWNode *ptr;
  int i;
  for ( i = 1; i <= size; i++ ) {
     ptr = adj[i];
     cout << "\n " << i;
     while ( ptr != NULL ) {
        cout << "->" << ptr->vertex
        ptr = ptr->next;
     }
     cout << endl;
  }
}
```

Listing 11.6:

```
// member function to output a weighted graph
void WeightedGraph::printGraph() {
  WNode *ptr;
  int i;
  for ( i = 1; i <= size; i++ ) {
     ptr = adj[i];
     cout << "\n " << i;
```



```
     while ( ptr != NULL ) {
        cout << "->(" << ptr->vertex
             << "," << ptr->weight << ")";
        ptr = ptr->next;
     }
     cout << endl;
  }
}
```

11.4.4 Deleting a Graph

This task is accomplished by the destructor of the respective classes as shown below:

Listing 11.7:

```
// destructor to remove non-weighted graph from memory
NonWeightedGraph::~NonWeightedGraph()
{
  int i;
  NWNode *temp, *ptr;
  for ( i = 1; i <= size; i++ )
  {
     ptr = adj[i];
     while ( ptr != NULL )
     {
        temp = ptr;
        ptr  = ptr->next;
        delete temp;
     }
     adj[i] = NULL;
  }
}
```

Listing 11.8:

```
// destructor to remove weighted graph from memory
WeightedGraph::~WeightedGraph()
{
  int i;
  WNode *temp, *ptr;
  for ( i = 1; i <= size; i++ )
  {
     ptr = adj[i];
     while ( ptr != NULL )
     {
        temp = ptr;
        ptr  = ptr->next;
        delete temp;
     }
     adj[i] = NULL;
  }
}
```

```
//  Program to illustrate the input and output of a non-weighted graph
#include <iostream.h>
#include <iomanip.h>
#define   MAX    20
struct NWNode {          // --- node declaration ---
   int vertex;
   NWNode *next;
};
class NonWeightedGraph  // class modelling weighted graph
{
   protected:
      NWNode *adj[MAX];
      int size;
   public:
      NonWeightedGraph(int n);
      ~NonWeightedGraph();
      void addEdge(int i, int j, int w);
      void deleteEdge(int i, int j);
      void inputGraph();
      void printGraph();
      void BFS(int s);
      void DFSIterative(int s);
      void DFSRecursive(int s);
      void DFSVisit(int u);
      void DFSRecursiveModified();
      void DFSVisitModified(int u);
      void topologicalSort();
};  //    ---- end of class template ----
//  parameterized constructor
NonWeightedGraph::NonWeightedGraph(int n) {
   int i;
   for ( i = 1; i <= n; i++ )
      adj[i] = NULL;
   size = n;
}
// member function to input a weighted graph
void NonWeightedGraph::inputGraph()
{
   NWNode *ptr, *last;
   int i, j, m, val;
   for ( i = 1; i <= size; i++ ) {
       last = NULL;
       cout << "\nNumber of nodes in the adjacency"
            << " list of node #" << i << " : ";
       cin >> m;
       for ( j = 1; j <= m; j++ ) {
          cout << "Enter node #" << j << " : ";
          cin >> val;
          ptr = new NWNode;
          ptr->vertex = val;
          ptr->next = NULL;
          if ( adj[i] == NULL )
            adj[i] = last = ptr;
          else {
            last->next = ptr;
```

```
            last = ptr;
          }
       }
   }
}
// member function to input a weighted graph
void NonWeightedGraph::printGraph()
{
  NWNode *ptr;
  int i;
  for ( i = 1; i <= size; i++ ) {
     ptr = adj[i];
     cout << "\n " << i;
     while ( ptr != NULL ) {
        cout << "->" << ptr->vertex;
        ptr = ptr->next;
     }
     cout << endl;
  }
}
// destructor
NonWeightedGraph::~NonWeightedGraph()
{
  int i;
  NWNode *temp, *ptr;
  for ( i = 1; i <= size; i++ ) {
      ptr = adj[i];
      while ( ptr != NULL ) {
         temp = ptr;
         ptr  = ptr->next;
         delete temp;
      }
      adj[i] = NULL;
  }
}
void main()
{
  int n;
  cout << "\nEnter number of nodes[<=20] in the graph : ";
  cin >> n;
  NonWeightedGraph nwgraph(n);
  nwgraph.inputGraph();
  cout << "\nADJACENCY LIST FOR INPUT GRAPH \n\n";
  nwgraph.printGraph();
}
```

Listing 11.10:

```
//  Program to illustrate the input and output of a weighted graph
#include <iostream.h>
#include <iomanip.h>
#define  MAX    20
struct WNode              // --- node declaration ---
{
  int vertex;
  int weight;
```

```
   WNode *next;
};
class WeightedGraph {  // --- class modelling weighted graph
   protected:
      WNode *adj[MAX];
      int size;
   public:
      WeightedGraph(int n);
      ~WeightedGraph();
      void addEdge(int i, int j, int w);
      void deleteEdge(int i, int j);
      void inputGraph();
      void printGraph();
      void BFS(int s);
      void DFSIterative(int s);
      void DFSRecursive(int s);
      void DFSVisit(int u);
      void DFSRecursiveModified();
      void DFSVisitModified(int u);
      void topologicalSort();
      void MinimumSpanningTreePrim();
      void shortestPathDijkstra(int s, int d);
};  //    ---- end of class template ----
//  parameterized constructor
WeightedGraph::WeightedGraph(int n) {
   int i;
   for ( i = 1; i <= n; i++ )
      adj[i] = NULL;
   size = n;
}
// member function to input a weighted graph
void WeightedGraph::inputGraph()
{
   WNode *ptr, *last;
   int i, j, m, val, wt;
   for ( i = 1; i <= size; i++ ) {
       last = NULL;
       cout << "\nNumber of nodes in the adjacency"
            << " list of node #" << i << " : ";
       cin >> m;
       for ( j = 1; j <= m; j++ ) {
          cout << "Enter node #" << j << " : ";
          cin >> val;
          cout << "Enter weight for edge ("
               << i << "," << val << ") : ";
          cin >> wt;
          ptr = new WNode;
          ptr->vertex = val;
          ptr->weight = wt;
          ptr->next = NULL;
          if ( adj[i] == NULL )
              adj[i] = last = ptr;
          else {
              last->next = ptr;
              last = ptr;
          }
       }
   }
```



```
// member function to input a weighted graph
void WeightedGraph::printGraph()
{
 WNode *ptr;
 int i;
 for ( i = 1; i <= size; i++ )
 {
   ptr = adj[i];
   cout << "\n " << i;
   while ( ptr != NULL )
   {
      cout << "->(" << ptr->vertex
           << "," << ptr->weight << ")";
      ptr = ptr->next;
   }
   cout << endl;
 }
}
// destructor
WeightedGraph::~WeightedGraph()
{
  int i;
  WNode *temp, *ptr;
  for ( i = 1; i <= size; i++ )
  {
     ptr = adj[i];
     while ( ptr != NULL )
     {
        temp = ptr;
        ptr  = ptr->next;
        delete temp;
     }
     adj[i] = NULL;
  }
}
void main()
{
  int n;
  cout << "\nEnter number of nodes[<=20] in the graph : ";
  cin >> n;
  WeightedGraph wgraph(n);
  wgraph.inputGraph();
  cout << "\nADJACENCY LIST FOR INPUT GRAPH \n\n";
  wgraph.printGraph();
}
```

11.5 Traversal

Many applications of graphs require examining the vertices and edges of a graph *G*. There are two standard ways for graph traversal:

- Breadth-first search
- Depth-first search

11.4.5.1 Breadth-First Search

Given an input graph $G = (V, E)$ and a source vertex s, from where to begin. The breadth first search (BFS) systematically explores the edges of G to discover every vertex that is reachable from s. It produces a breadth-first tree with root s that contains all such vertices that are reachable form s. For every vertex v reachable from s, the path in the breadth-first tree from s to v corresponds to a shortest path. The algorithm works on both directed as well as undirected graphs.

To keep track of the progress, BFS colors each vertex WHITE, GRAY, or BLACK, where vertex with WHITE color indicates that the vertex is still undiscovered, vertex with GRAY color indicates that the vertex is discovered but not yet processed, and the vertex with BLACK color indicates that the vertex has been processed. These three states can also be represented by using some integer variable, say *status*. The above mentioned three states during the progress of the BFS can be indicated by assigning value 1, 2, and 3 to *status*. In our discussion, we will consider coloring scheme.

All the vertices are initially WHITE (non-discovered), and later on become GRAY (discovered) and then BLACK (finished). As soon as the vertex is discovered for the first time, it becomes non-WHITE. Therefore, GRAY and BLACK vertices are have been discovered, but the breadth-first search distinguishes between them to ensure that the search proceeds in a breadth-first manner. If $(u, v) \in E$ and vertex u is BLACK, then vertex v is either GRAY or BLACK; that is, all vertices adjacent to BLACK vertices have been discovered. GRAY vertices may have some adjacent WHITE vertices; they represent the frontier between discovered and undiscovered vertices.

The algorithm uses a FIFO queue structure to maintain the order in which the vertices are to be processed.

Algorithm 11.1:

BreadthFirstSearch(***adj, n, s*** **)**

Here *adj* is the adjacency-list of the graph with n vertices, and vertex s is the source vertex. This algorithm uses a FIFO queue q, and linear array *color* of size n, to keep track of the progress of BFS. It also uses a local variable u.

```
Begin
   for i = 1 to n by 1 do
     Set color[i] = WHITE
   endfor
   Set color[s] = GRAY
   Call CreateQueue( q );
   Call Enqueue(q, s)
   while ( q is not empty ) do
     Set u = Peek( q )
     Set ptr = adj[u]
     while ( ptr ≠ NULL ) do
       if ( color[ptr->info] = WHITE ) then
         Set color[s] = GRAY
         Call Enqueue( q, ptr->info)
       endif
     endwhile
     Print: u
     Set color[u] = BLACK
   endwhile
End.
```


Example 11.1:

Consider the following graph, and source vertex 2.

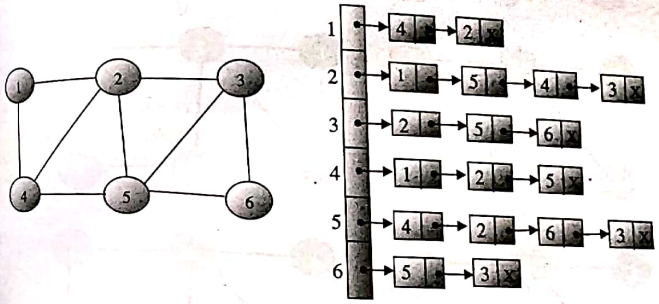
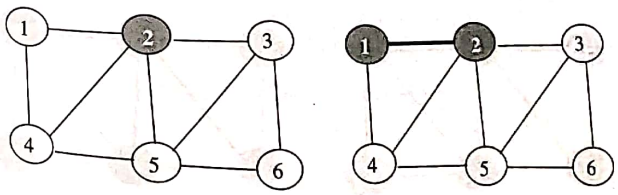


Figure 11.6: Undirected graph and its adjacency-list

Solution:



(*a*) (*b*)

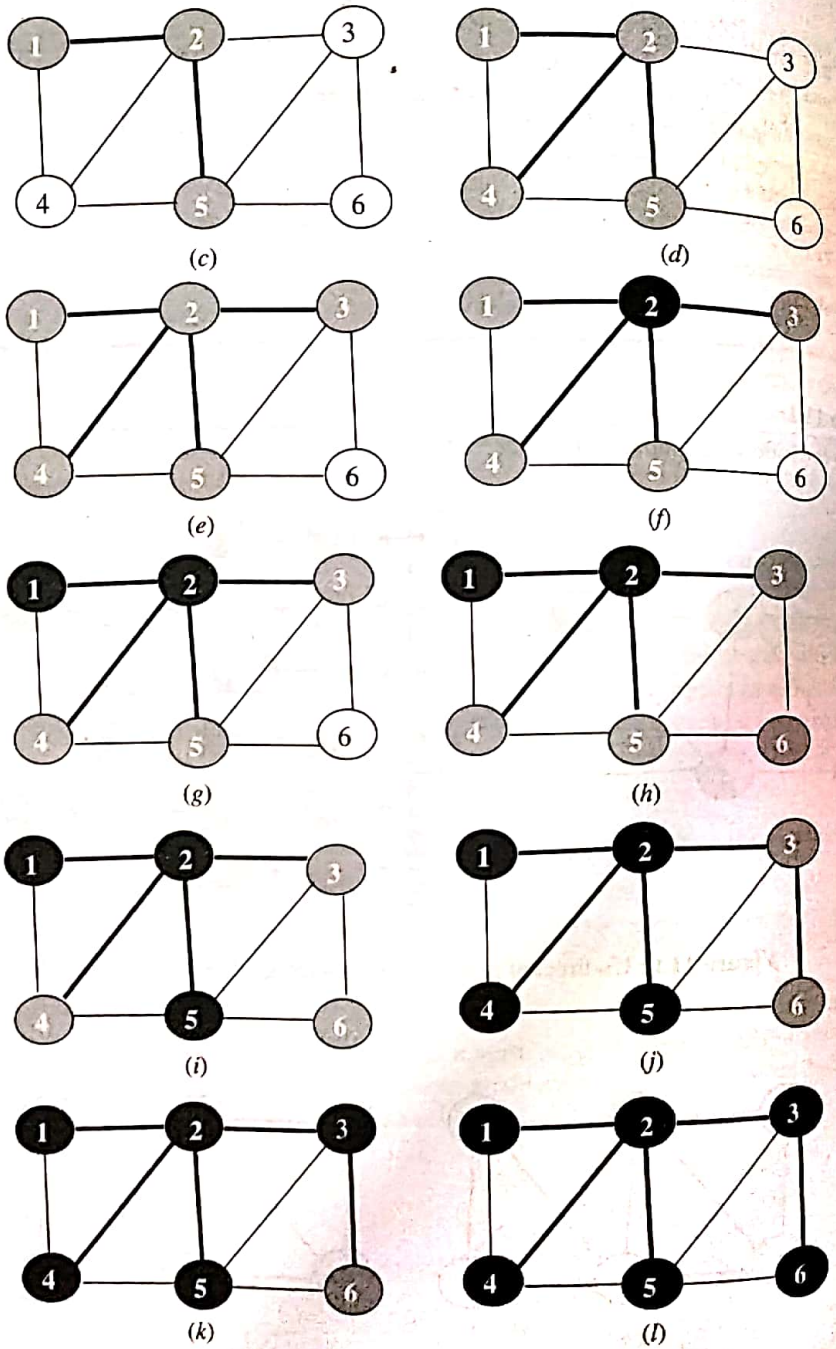


Figure 11.7: (*a*) – (*l*) illustrates BFS on undirected graph.

The vertices are traversed in following order

2 1 5 4 3 6

The following algorithm summarizes various steps required to implement the BFS.

Listing 11.11:

```
// Program to illustrate the BFS on a non-weighted graph
//
#include <iostream.h>
#include <iomanip.h>
#include "graph.h"
#include "linkq.h"
#define WHITE 0
#define GRAY  1
#define BLACK 2
void NonWeightedGraph::BFS(int s)
{
  NWNode *ptr;
  LinkQueue<int> q;
  int i, u, v, color[MAX];
  for ( i = 1; i <= size; i++ )
     color[i] = WHITE;
  color[s] = GRAY;
  q.enqueue(s);
  while ( !q.isEmpty() )
  {
     u = q.peek();
     ptr = adj[u];
     while ( ptr != NULL )
     {
        v = ptr->vertex;
        if ( color[v] == WHITE )
        {
            color[v] = GRAY;
            q.enqueue(v);
        }
        ptr = ptr->next;
     }
     u = q.dequeue();
     cout << u << "    ";
     color[u] = BLACK;
  }
}
void main()
{
  int n, s;
  cout << "\nEnter number of nodes in the graph : ";
  cin >> n;
  NonWeightedGraph nwgraph(n);
  nwgraph.inputGraph();
  cout << "\nEnter source vertex : ";
  cin >> s;
  cout << "\n\nBFS from vertex " << s << " is\n\n";
  nwgraph.BFS(s);
}
```

11.4.5.2 Depth-First Search

The depth-first search (DFS), as its name implies, is to search *deeper* in the graph, whenever possible. The edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it. When all of v's edges have been explored, the search *backtracks* to explore edges leaving the vertex from which v was discovered. This process continues until we have discovered all the vertices that are reachable from the source vertex. This algorithm also works on both directed as well as on undirected graphs.

To keep track of the progress, the DFS also uses the same coloring scheme as described for BFS. In addition, the DFS makes uses of a stack structure to maintain the order in which the vertices are to be processed, instead of queue.

The algorithm 11.2 summarizes various steps required to implement the DFS.

Algorithm 11.2:

DepthfirstSearchIterative(*adj*, *n*, *s*)

Here *adj* is the adjacency-list of the graph with n vertices, and vertex s is the source vertex. This algorithm uses a stack STACK, and linear array *color* of size n, to keep track of the progress of DFS. It also uses local variables u and *ptr*.

```
Begin
  for i = 1 to n by 1 do
     Set color[i] = WHITE
  endfor
  Set color[s] = GRAY
  CALL createStack( stack )
  Call push( stack, s )
  while ( stack is not empty ) do
     Set u = peek(stack)
     Set ptr = adj[u]
     while (ptr ≠ NULL ) do
        if (color[ptr->info] = WHITE ) then
           Set color[ptr->info] = GRAY
           Call push(&stack, ptr->info)
           Set u = ptr->info
           Set ptr = adj[u]
        else
           Set ptr = ptr ->next
        endif
     endwhile
     Call pop(&stack, u)
     Print: u
     Set color[u] = BLACK
  endwhile
End.
```


Example 11.2:
Consider the following graph, and source vertex 2

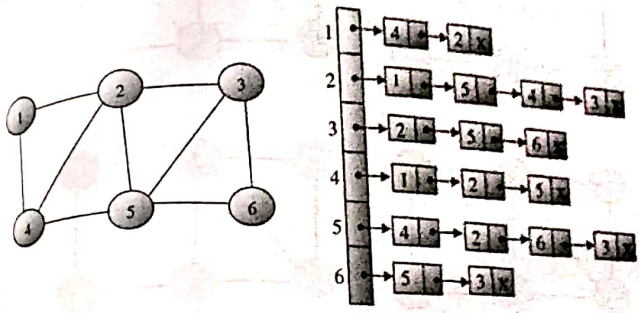
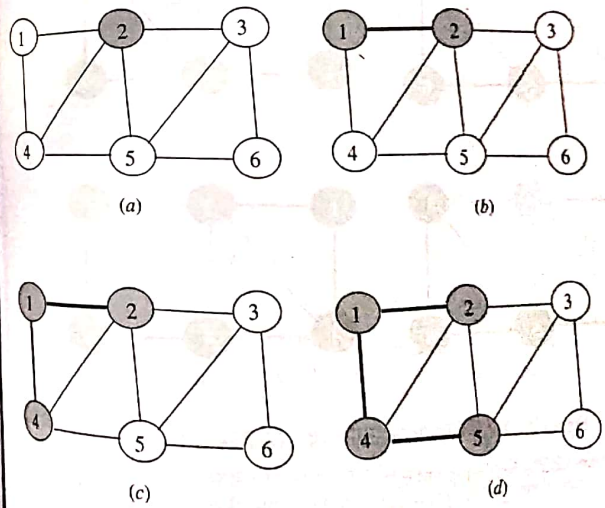


Figure 11.8: Undirected graph and its adjacency-list

Solution:



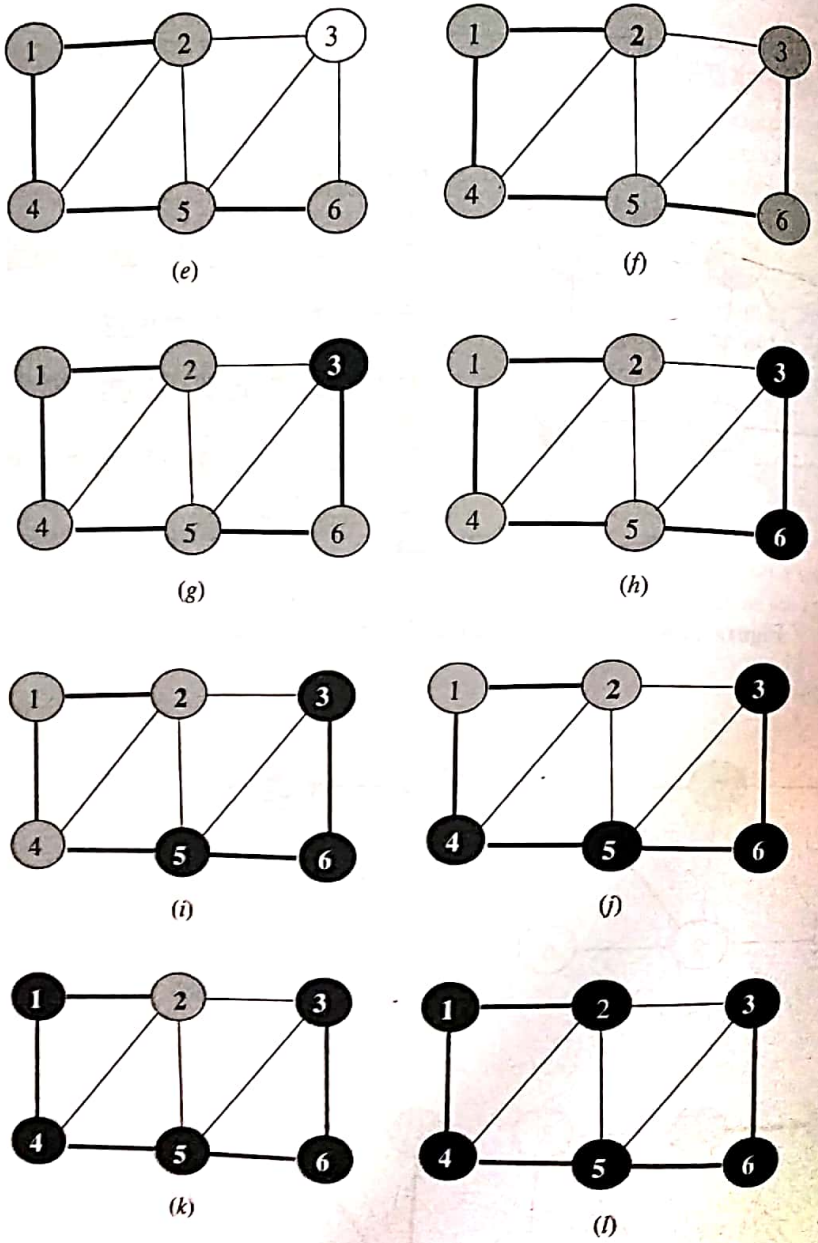


Figure 11.9: *(a) – (l)* illustrates DFS on undirected graph.

The vertices are traversed in following order

3 6 5 4 1 2

Listing 11.12:

```
//  Program to illustrate the DFS iterative on a non-weighted graph
//
#include <iostream.h>
#include <iomanip.h>
#include "graph.h"
#include "stack.h"
#define WHITE 0
#define GRAY  1
#define BLACK 2
void NonWeightedGraph::DFS(int s)
{
  NWNode *ptr;
  LinkStack<int> intStack;
  int i, u, color[MAX];
  for ( i = 1; i <= size; i++ )
     color[i] = WHITE;
  color[s] = GRAY;
  intStack.push(s);
  while ( !intStack.isEmpty() ) {
     u = intStack.peek();
     ptr = adj[u];
     while ( ptr != NULL ) {
       if (color[ptr->vertex] == WHITE ) {
           color[ptr->vertex] = GRAY;
           intStack.push(ptr->vertex);
           u = ptr->vertex;
           ptr = adj[u];
       } else {
           ptr = ptr->next;
       }
     }
     u = intStack.pop();
     cout << u << "   ";
     color[u] = BLACK;
  }
}
void main()
{
  int n, s;
  cout << "\nEnter number of nodes in the graph : ";
  cin >> n;
  NonWeightedGraph nwgraph(n);
  nwgraph.inputGraph();
  cout << "\nEnter source vertex : ";
  cin >> s;
  cout << "\n\nDFS from vertex " << s << " is\n\n";
  nwgraph.DFS(s);
}
```

In example 10.2, you must have observed that DFS is a recursive procedure. *i.e.*, traverses as deep as possible exploring each edge on the way, and backtracks once there is no further edge coming out of the recently discovered vertex.

The algorithm 11.3 summarizes the various steps to perform recursive visit to each vertex u.

Algorithm 11.3:

DfsVisit(*adj*, *n*, *u*)

Here *adj* is the adjacency-list of the graph with *n* vertices, and vertex *u* is the vertex to be visited. This algorithm uses recursion.

```
Begin
  Set color[u] = GRAY
  Set ptr = adj[u]
  while (ptr  ≠ NULL ) do
     Set v = ptr->info
     if (color[v] = WHITE ) then
        Set color[v] = GRAY
        Call DfsVisit(adj, n, v )
     endif
  endwhile
  Print: u
  Set color[u] = BLACK
End.
```

Algorithm 11.4 summarizes the various steps to perform initialization, and call the procedure of algorithm 11.3 to perform DFS visit to undiscovered vertices.

Algorithm 11.4:

DFRecursive(*adj*, *n*, *s*)

Here *adj* is the adjacency-list of the graph with *n* vertices, and vertex *s* is the source vertex. This algorithm uses linear array *color* of size *n*, to keep track of the progress of DFS. The variables *i* is used as index variable.

```
Begin
  for i = 1 to n by 1 do
     Set color[i] = WHITE
  endfor
  Call DfsVisit( adj, n, s )
End.
```

Listing 11.13:

```
//  Program to illustrate the DFS recursive on a non-weighted graph
#include <iostream.h>
#include <iomanip.h>
#include "graph.h"
#define WHITE 0
```



```
#define GRAY  1
#define BLACK 2
int color[MAX];

void NonWeightedGraph::DFSVisit(int u)
{
  int v;
  NWNode *ptr;
  color[u] = GRAY;
  ptr = adj[u];
  while ( ptr != NULL ) {
     v = ptr->vertex;
     if ( color[v] == WHITE )
      DFSVisit(v);
     ptr = ptr->next;
  }
  cout << u << "    ";
  color[u] = BLACK;
}

void NonWeightedGraph::DFSRecursive(int s)
{
  int i;
  for ( i = 1; i <= size; i++ )
     color[i] = WHITE;
  DFSVisit(s);
}

void main()
{
  int n, s;
  cout << "\nEnter number of nodes in the graph : ";
  cin >> n;
  NonWeightedGraph nwgraph(n);
  nwgraph.inputGraph();
  cout << "\nEnter source vertex : ";
  cin >> s;
  cout << "\n\nBFS from vertex " << s << " is\n\n";
  nwgraph.DFSRecursive(s);
}
```

11.5 APPLICATIONS OF GRAPHS

In this section, we will discuss some of the applications of graphs.

11.5.1 Topological Sort

A topological sort of a directed graph without cycles, also known as directed acyclic graph or simple DAG, $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering.

If the graph is contains cycle(s), *i.e.*, graph is not DAG, then no linear ordering is possible.

A topological sort of a graph can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.

Directed acyclic graphs are used in many applications to indicate precedence among events.

To find topological sort of a DAG, the general structure of the algorithm can be described as follows:

```
Begin
   Perform DFS on G for each vertex
   As each vertex is finished, insert at front of a linked list
   Print the elements of linked list in order
End.
```

To find the topological sort of a DAG, the procedure to perform DFS visit to a vertex is slightly modified as given in the algorithm 11.5, where instead of printing the vertex, it is inserted in the beginning of the linked list.

Algorithm 11.5:

DfsVisitModified(adj, n, u)

Here *adj* is the adjacency-list of the graph with *n* vertices, and vertex *u* is the vertex to be visited. This algorithm uses recursion.

```
Begin
  Set color[u] = GRAY
  Set ptr = adj[u]
  while (ptr  ≠ NULL ) do
     Set v = ptr->info
     if ( color[v] = WHITE ) then
        Call DfsVisitModified(adj, n, v )
     endif
     Set ptr  = ptr->next
  endwhile
  Insert vertex u in beginning of linear linked list LIST
  Set color[u] = BLACK
End.
```

The algorithm 11.6 summarizes the various steps to perform the topological sort of a DAG.

Algorithm 11.6:

TopologicalSort(*adj*, *n*)

Here *adj* is the adjacency-list of the graph with *n* vertices. This algorithm finds the topological sort of the graph G. It uses linear linked list LIST to store the visited by DFS-VISIT procedure in order of their traversal.

```
Begin
```



```
Create linear linked list LIST
for i = 1 to n by 1 do
   Set color[i] = WHITE
endfor
for i = 1 to n by 1 do
   if ( color[i] = WHITE ) then
      Call DfsVisitModified(adj, n, i )
   endif
endfor
Set ptr = LIST
while (ptr  ≠ NULL ) do
   Print ptr->info
   Set ptr = ptr ->next
endwhile
End.
```

Example 11.3:

Consider the following graph

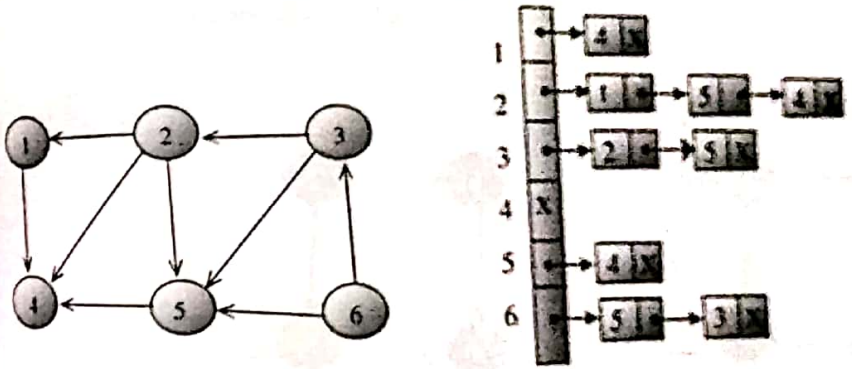
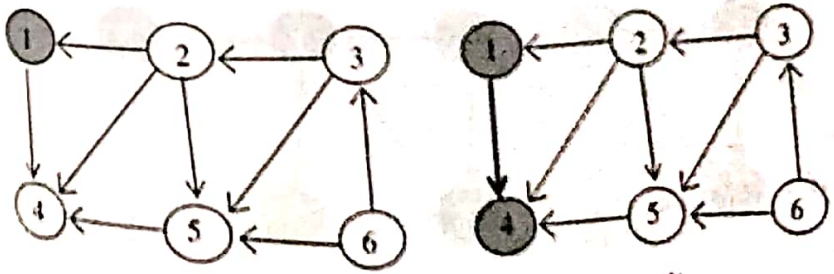
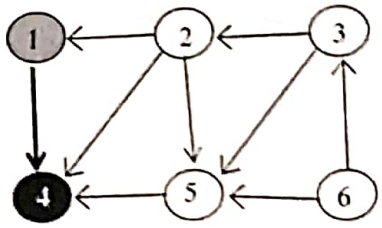


Figure 11.10: Given directed acyclic graph and its adjacency list

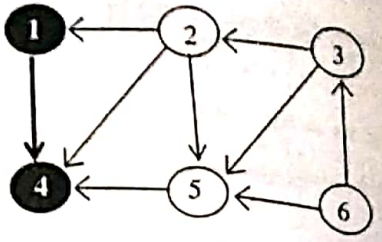
Solution:



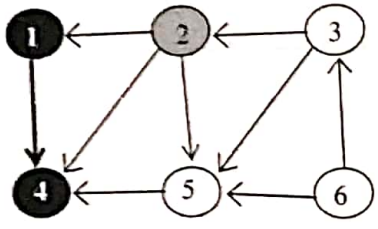
(a) (b)



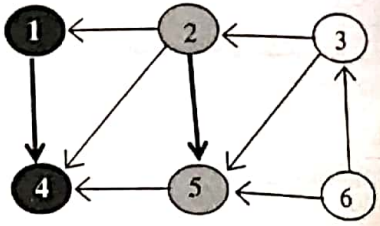
(c)



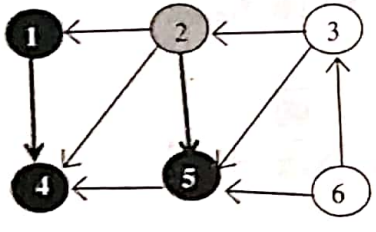
(d)



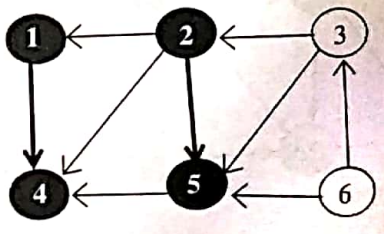
(e)



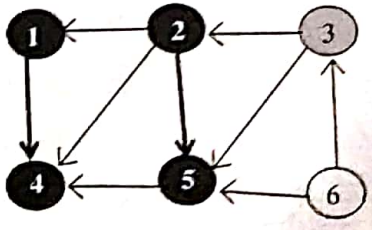
(f)



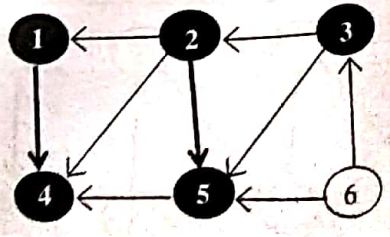
(g)



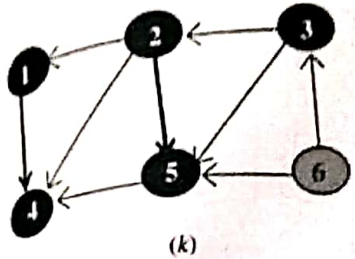
(h)



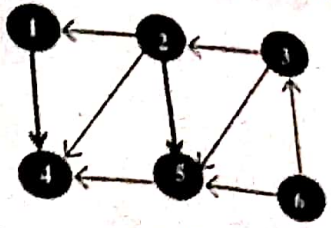
(i)



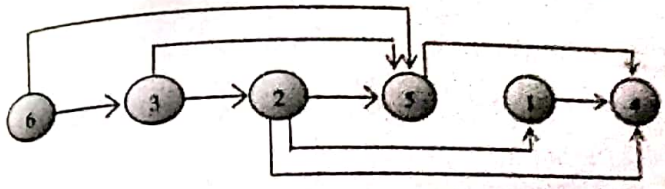
(j)



(k)



(l)



(m)

Figure 11.11: (*a*) – (*l*) illustrates DFS on undirected graph. Tree edges are shown thick as they are produced by DFS. (*m*) shows topological sort of the given graph.

Listing 11.14:

```
// Program to illustrate the topological sort on DAG
#include <iostream.h>
#include <iomanip.h>
#include "llist.h"
#include "graph.h"
#define WHITE 0
#define GRAY  1
#define BLACK 2

int color[MAX];
llist<int> list;

void NonWeightedGraph::topologicalSort()
{
  cout << "\nTopological Sort of given graph\n";
  DFSRecursiveModified();
  list.traverseInOrder();
}

void NonWeightedGraph::DFSRecursiveModified() {
  int i;
  for ( i = 1; i <= size; i++ )
     color[i] = WHITE;
  for ( i = 1; i <= size; i++ ) {
     if ( color[i] == WHITE )
        DFSVisitModified(i);
  }
```

```
void NonWeightedGraph::DFSVisitModified(int u)
{
   int v;
   NWNode *ptr;
   color[u] = GRAY;
   ptr = adj[u];
   while ( ptr != NULL )
   {
      v = ptr->vertex;
      if ( color[v] == WHITE )
       DFSVisitModified(v);
      ptr = ptr->next;
   }
   list.insertAtBeginning(u);
   color[u] = BLACK;
}
void main()
{
   int n;
   cout << "\nEnter number of nodes in the graph : ";
   cin >> n;
   NonWeightedGraph nwgraph(n);
   nwgraph.inputGraph();
   nwgraph.topologicalSort();
}
```

11.5.2 Minimum Spanning Tree

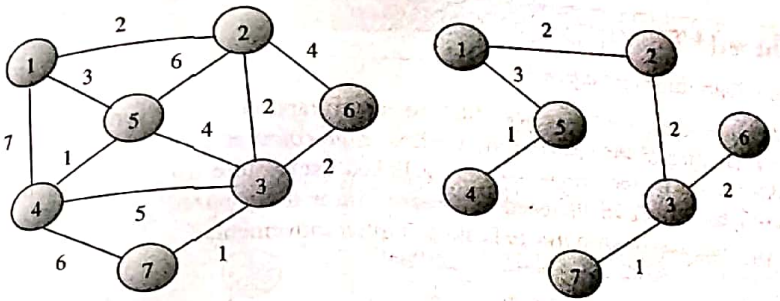
A *spanning tree* for a graph, $G = (V, E)$, is a subgraph of G that is a tree and contains all the vertices of G. In a weighted graph, the weight of a graph is the sum of the weights of the edges of the graph. A *minimum spanning tree* (MST) for a weighted graph is a spanning tree with minimum weight. If graph G with n vertices, then the MST will have $(n-1)$ edges, assuming that the graph is connected. In general, a weighted graph may have more than one MST.

If G is not connected, then it cannot have any spanning tree. In this case it will have a spanning forest. For simplicity, we will assume that the given graph G is connected.

There are many situations in which MST must be found. For example;

- We want to find cheapest way to connect a set of terminals, where terminals may represent cities, electrical/electronic components of a circuit, computers or factories, by using say roads, wires, or telephone lines. The solution to this is a MST, which has an edge for each possible connection weighted by the cost of that connection.
- It is also an important sub problem in various routing algorithms. By routing means finding efficient paths through a graph that visit every vertex (or every edge).

If G is not connected, then it cannot have any spanning tree. In this case it will have a spanning forest. To simplify our discussion, we will assume that the given graph G is connected.



(*a*) Given weighted graph (*b*) Minimum Spanning Tree

Figure 11.12: Given graph and its corresponding Minimum Spanning Tree

The Prim algorithm begins by selecting an arbitrary starting vertex, and then branches out from the part of the tree constructed so far by choosing a new vertex and edge at each iteration. During the course of the algorithm, the vertices may be thought of as divided into three disjoint categories as follows:

- **Tree vertices** – Those in the tree constructed so far.
- **Fringe vertices** – Those vertices that are not in tree, but are adjacent to some vertex in the tree.
- **Unseen vertices** – Remaining vertices of the graph.

The key step in the algorithm is the selection of a vertex from the fringe vertices and an incident edge. Actually, since the weights are on the edges, the focus of the choice is on the edge, not the vertex. The Prim's algorithm always chooses an edge from a tree vertex to a fringe vertex of minimum weight. The general structure of the algorithm can be described as follows:

```
Begin
  Select an arbitrary vertex to start the tree
  while there are fringe vertices do
     Select an edge of minimum weight between a tree and
     a fringe vertex
     Add the selected edge and the fringe vertex to the tree
  endwhile
End.
```

After each iteration of the algorithm's loop, there may be new fringe vertices, and the set of edges from which the next selection is made. For each fringe vertex, we need to keep track of only one edge to it from the tree – *the one with lowest weight*. We will call such edges candidate edges.

The various steps required are summarized in the algorithm 11.7.

Algorithm 11.7:

MinimumSpanningTree(*adj*, *n* **)**

Here *adj* is the adjacency-list representing the input graph G with n vertices. It uses linear *status* to keep track of the status of the vertices, *edge-count* as counter to keep track of the edges added so far to tree, *fringe-list* a linear linked list to store fringe vertices, linear array *fringe-wt* to keep track of the candidate edges, linear array *parent* to store parent of the vertex, and a flag *stuck*, when *true* indicate no further movement.

```
Begin
  for i = 2 to n by 1 do
      Set status[i] = UNSEEN
  endfor
  Set x = 1
  Set status[x] = INTREE
  Set edge-count = 0
  Set stuck = false
  Create fringe-list

  while ( ( edge-count <  n -1 ) and ( not stuck ) ) do
     Set ptr = adj[x]

     while (ptr  ≠ NULL ) do
        Set y = ptr->vertex
        if ((status[y] = fringe)and(ptr->weight < fringe-wt[y])) then
           Set parent[y] = x
           Set fringe-wt[y] = ptr->weight
        else if  (status[y] = UNSEEN ) then
           Set status[y] = FRINGE
           Insert vertex y into fringe-list
           Set parent[y] = x
           Set fringe-wt[y] = ptr->weight
        endif
        Set ptr = ptr->next
     endwhile

     if ( fringe-list is empty ) then
        Set stuck = true
     else
        Traverse the fringe list to find a vertex with minimum weight
        Set x = the fringe vertex incident with the edge
        Remove x from the fringe-list
        Set status[x] = INTREE
        Set edge-count = edge-count + 1
     endif
  endwhile
  for x = 2 to n by 1 do
     Print x, parent[x]
  endfor
End.
```


Example 11.4:

Consider the following graph

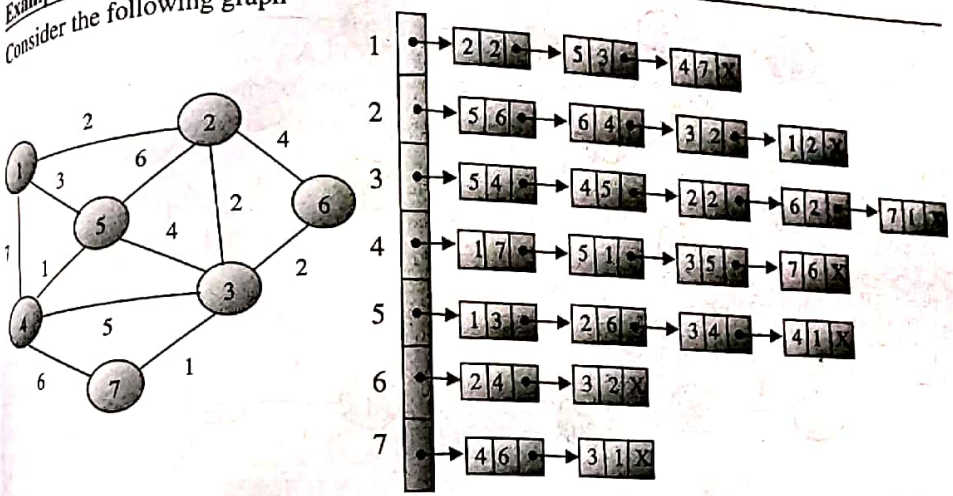
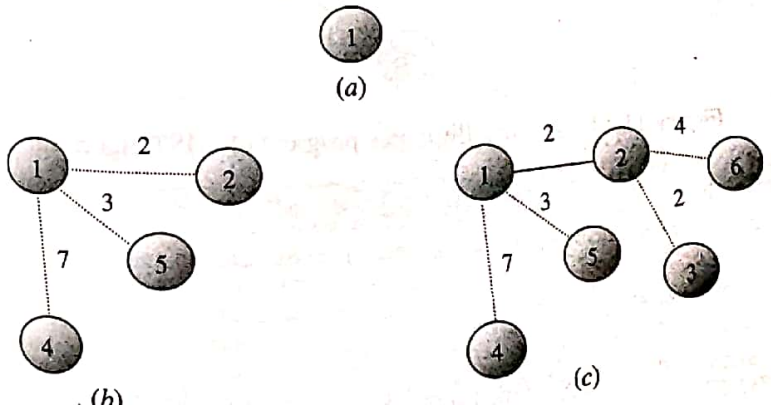


Figure 11.13: Given graph and its adjacency-list

Solution:

Following notations for tree edges and fringe edges will be used.

The procedure starts with vertex 1 as progresses as shown below.



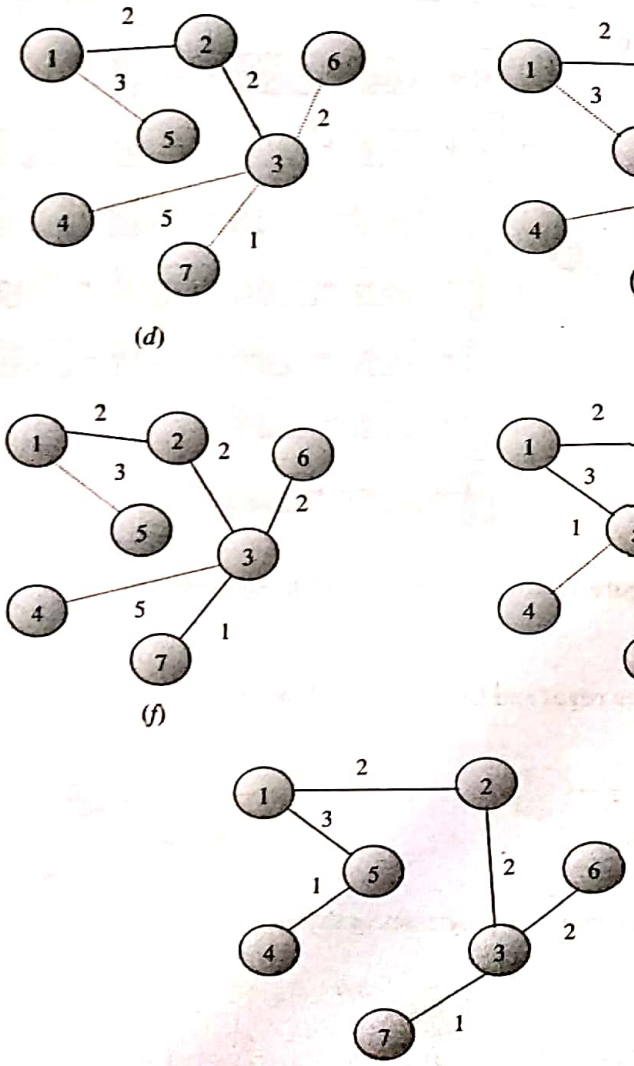


Figure 11.14: *(a)* – *(h)* Illustrates progress of MST algorithm

Listing 11.15:

```
//  Program to construct minimum spanning tree using Prim's algorithm
#include <iostream.h>
#include <iomanip.h>
#include "graph.h"
#define UNSEEN   0
#define FRINGE   1
#define INTREE   2
```



```
struct Node
{
  int vertex;
  Node *next;
};
void insertAtBeginning( Node **fringeList, int d )
{
  Node *ptr;
  ptr = new Node;
  ptr->vertex = d;
  ptr->next = *fringeList;
  *fringeList = ptr;
}
void deleteElement(Node **fringeList,  int d )
{
  Node *loc, *ploc=NULL;
  ploc = NULL;
  loc = *fringeList;
  while ( loc->vertex != d )
  {
     ploc = loc;
     loc = loc->next;
  }
  if ( ploc == NULL )
     *fringeList = (*fringeList)->next;
  else
     ploc->next = loc->next;
  delete loc;
}
int isEmpty ( Node *ptr  )
{
  return ( ptr == NULL ) ? 1 : 0;
}

void deleteList(Node **fringeList)
{
  Node *ptr;
  while ( *fringeList != NULL )
  {
     ptr = *fringeList;
     *fringeList = (*fringeList)->next;
     delete ptr;
  }
}

void WeightedGraph::MinimumSpanningTreePrim()
{
  Node *fringeList=NULL, *ptr2;
  WNode *ptr1;
  int i, x, stuck, w, edgeCount, minWt, y, weight;
  int parent[MAX], status[MAX], fringeWt[MAX];
  for ( i = 1; i <= size; i++ )
     status[i] = UNSEEN;
  x = 1;
  status[x] = INTREE;
  edgeCount = 0;
  stuck = 0;
```

```
   while ( ( edgeCount < size ) && ( !stuck ) )
   {
      ptr1 = adj[x];
      while ( ptr1 != NULL ) {
        y = ptr1->vertex;
        weight = ptr1->weight;
        if (( status[y] == FRINGE )&&( weight < fringeWt[y] ))
        {
          parent[y] = x;
          fringeWt[y] = weight;
        } else if ( status[y] == UNSEEN ) {
          status[y] = FRINGE;
          parent[y] = x;
          fringeWt[y] = weight;
          insertAtBeginning(&fringeList, y);
        }
        ptr1 = ptr1->next;
      }
      if ( isEmpty(fringeList) )
        stuck = 1;
      else
      {
        x = fringeList->vertex;
        minWt = fringeWt[x];
        ptr2 = fringeList->next;
        while ( ptr2 != NULL )
        {
           w = ptr2->vertex;
           if ( fringeWt[w] < minWt )
           {
              x = w;
              minWt = fringeWt[w];
           }
           ptr2 = ptr2->next;
        }
        deleteElement(&fringeList,x);
        status[x] = INTREE;
        edgeCount++;
      }
   }
   for ( x = 2; x <= size; x++ )
      cout << "(" << x << "," << parent[x] << ")";
   deleteList(&fringeList);
}

void main()
{
   int n;
   cout << "\nEnter number of nodes in the graph : ";
   cin >> n;
   WeightedGraph wgraph(n);
   wgraph.inputGraph();
   cout << "\n\nMinimum Spanning Tree";
   cout << " constitutes following edges\n\n";
   wgraph.MinimumSpanningTreePrim();
}
```


11.5.3 Finding Shortest Paths

A *path* from source vertex v to w is *shortest path* from v to w if there is no path from v to w with lower weights. The shortest paths are not necessarily unique.

There are many instances where we are interested in finding the most convenient paths for traveling from one place to another place. For example, suppose we have with us an airline map, which gives the cities having direct flights and the total time of flight, and air fair. We are interested to find that which route we should take so that we can reach as quickly as possible or a route for which the air fair is the minimum.

We may be interested in find a shortest path from a given city to another city, shortest paths from a given city to any other city reachable from the given city, and finally, the shortest paths from each city to the other one.

11.5.3.1 Shortest Path for Given Source and Destination

In order to find a shortest path from a given source to given destination, we discuss Dijkstra's shortest-path algorithm, which is very similar to Prim's algorithm for finding the MST.

The distance from a vertex x to a vertex y, denoted by $d(x,y)$, is the weight of a shortest path from vertex x to vertex y. The Dijkstra's shortest-path algorithm finds the shortest paths in order of increasing distance from a vertex, say v. Like MST, it branches out by selecting certain edges that lead to new edges.

Like MST, we keep track of candidate edge (the best found so far) for each fringe vertex. For each fringe vertex z, there is at least one path $v(=v_0), v_1, v_2, \ldots, v_k, z$ such that all the vertices except z are already in the tree. The candidate edge for z is the edge v_kz from a shortest path of this form. This information is stored in linear array DIST. The various steps required are summarized in the algorithm 11.8.

Algorithm 11.8:

ShortestPath(*adj*, *n*, *s*, *d*)

Here *adj* is the adjacency-list representing the input graph G with n vertices, s is the source vertex and d is the destination vertex. It uses linear *status* to keep track of the status of the vertices, *fringe-list* a linear linked list to store fringe vertices, linear array *dist* to denote weight of the path to the vertex, linear array *parent* to store parent of the vertex, and a flag *stuck*, when *true* indicate no further movement.

```
Begin
  for i = 1 to n by 1 do
     Set status[i] = UNSEEN, parent[i] = 0, dist[s] = 0
  endfor
  Set status[s] = INTREE
  Create fringe-list
  Set x = s
```

```
  Set stuck = false
  while ( ( x ≠ d ) and ( not stuck ) ) do
     Set ptr = adj[x]
     while ( ptr ≠ NULL ) do
        Set y = ptr->vertex
        if ((status[y]=fringe)and(dist[x] + ptr->weight < dist[y])) then
           Set parent[y] = x
           Set dist[y] = dist[x] + ptr->weight
        else if  ( status[y] = UNSEEN ) then
           Set status[y] = FRINGE
           Insert vertex y into fringe-list
           Set parent[y] = x
           Set dist[y] = dist[x] + ptr->weight
        endif
        Set ptr = ptr->next
     endwhile
     if ( fringe-list is empty ) then
        Set stuck = true
     else
        Traverse the fringe-list to find a vertex with minimum distance
        Set x =  vertex with minimum distance from s
        Remove x from the fringe-list
        Set status[x] = INTREE
     endif
  endwhile
  if ( parent[d] = 0 ) then
     Print "No path from  ", s, " to  ", d, " exists"
  else
     Call print-path( s, d, parent )
  endif
End.
```

The steps required for outputting the path recursively are summarized in the following algorithm.

Algorithm 11.9:

PrintPath(*s*, *d*, *parent*)

Here *s* is the source vertex, *d* the destination vertex, and *parent* is a linear array with *n* elements that maintains the parent child relationship.

```
Begin
  if ( s = d ) then
     Print: s
  else
     Call PrintPath( s, parent[d], parent )
  endif
End.
```


Example 11.5:

Consider the following graph

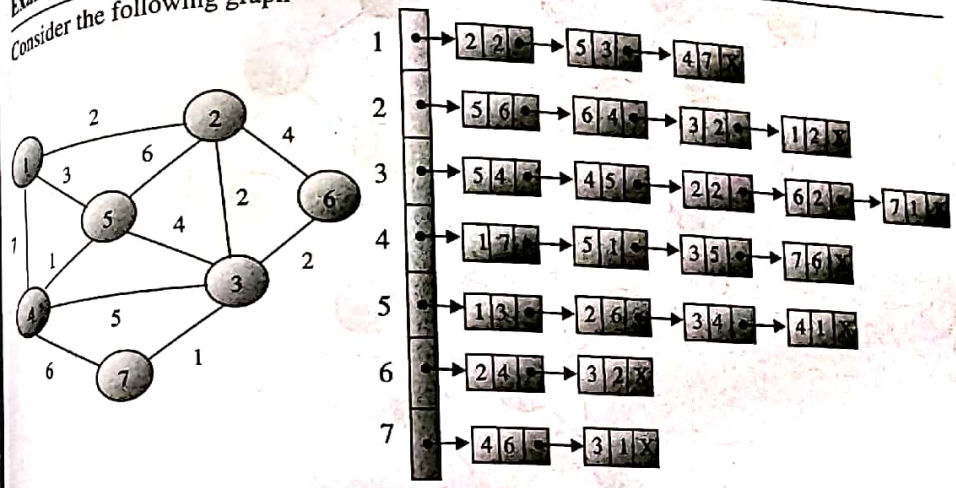
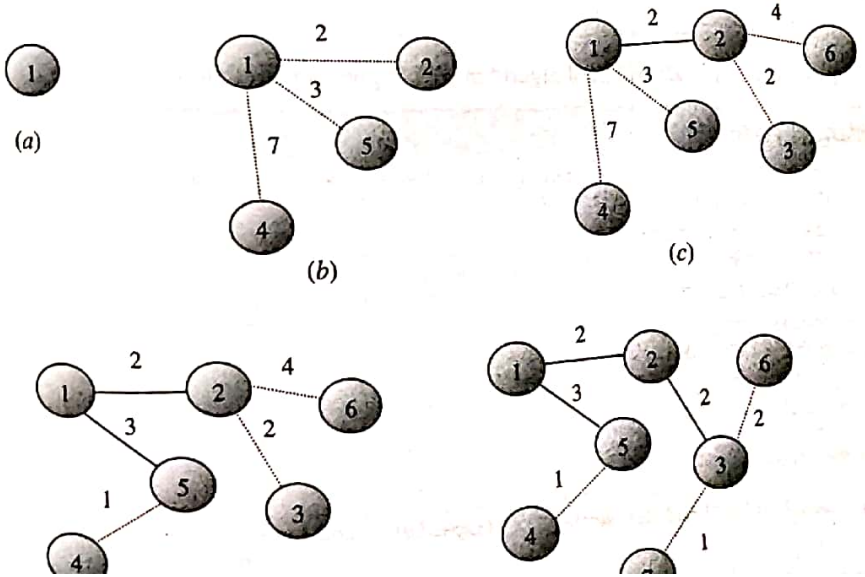


Figure 11.15: Given graph and its adjacency-list

Solution:

The following figure illustrates the working of the algorithm 11.6 by taking vertex 1 as source and vertex 6 as destination.



shortest paths, i.e., predecessor matrix P can be computed on-line as it ... matrices $D^{(k)}$. Specifically, the sequence of matrices $P^{(0)}$, $P^{(1)}$, $P^{(2)}$, ..., $P^{(n)}$ ... The element $p_{ij}^{(k)}$ is defined to be predecessor of vertex j on a shortest path from ... all the intermediate vertices in the set $\{1, 2, 3, \ldots, k\}$.

A recursive definition for $p_{ij}^{(k)}$ is as given below:

When $k = 0$, a shortest path from i to j has no intermediate vertices at all. Thus

$$p_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w(i,j) = 0 \\ i & \text{if } i \neq j \text{ or } w(i,j) \neq 0 \end{cases}$$

For $k \geq 1$, if we take a path from i to k and from k to j, then the predecessor of j we choose ... the same as the predecessor of j we choose on a shortest path from k with all ... vertices in the set $\{1, 2, 3, \ldots, k-1\}$. Otherwise, we choose the same predecessor of j that ... choose on a shortest path from i with all intermediate vertices in the set $\{1, 2, 3, \ldots, k-1\}$.

Formerly, for $k \geq 1$

$$p_{ij}^{(k)} = \begin{cases} p_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ p_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Based on the above recurrences, the procedure given in algorithm 11.10 can be used ... compute the values $d_{ij}^{(k)}$ and $p_{ij}^{(k)}$ in order of increasing values of k.

Algorithm 11.9:

FloydWarshall(w, n)

Here w is $n \times n$ adjacency-matrix representing directed input graph G. This uses a function ... MIN that returns the minimum elements among the two. The constant INFINITY represents ... a very large value which is used as a sentinel value to indicate that there is no edge between ... vertex i and j.

```
Begin
  for i = 1 to n by 1 do
    for i = 1 to n by 1 do
      if ( w[i,j] = 0 ) then
        Set d[i,j] = INFINITY, p[i,j] = 0
      else
        Set d[i,j] = w[i,j], p[i,j] = i
      endif
    endfor
  endfor
  for k = 1 to n by 1 do
    for i = 1 to n by 1 do
      for j = 1 to n by 1 do
        if ( d[i,j] <= d[i,k] + d[k,j] ) then
          Set p[i,j] = p[i,j]
        else
          Set p[i,j] = p[k,j]
        endif
        Set d[i,j] = MIN(d[i,j], d[i,k] + d[k,j])
      endfor
    endfor
  endfor
  Print "Shortest path matrix is"
  for i = 1 to n by 1 do
    for j = 1 to n by 1 do
      Print d[i,j]
    endfor
  endfor
  Print "Shortest paths are"
  for i = 1 to n by 1 do
    for j = 1 to n by 1 do
      if ( i ≠ j ) then
        Call print-paths( p, i, j )
      endif
    endfor
  endfor
End.
```


...e various steps required outputting all the paths are summarized in algorithm 11.10.

Algorithm 11.10:

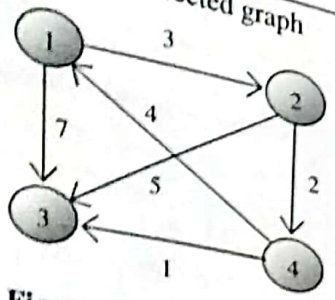
PrintPaths(p, i, j)

Here i is the source vertex, j the destination vertex, and p is a two-dimensional array of size ... that stores the parent child relationship.

```
Begin
  if ( i = j ) then
    Print i
  else if ( p[i,j] = NIL ) then
    Print "No path from", i, "to", j, "exists"
  else
    Call PrintPaths( p, i, p[i,j] )
    Print j
  endif
End.
```

...mple 11.6:

Consider the following directed graph



$$W = \begin{pmatrix} 0 & 3 & 7 & 0 \\ 0 & 0 & 5 & 2 \\ 0 & 0 & 0 & 0 \\ 4 & 0 & 1 & 0 \end{pmatrix}$$

Figure 11.17: Directed graph and its adjacency matrix representation

The following matrices are the sequence of matrices D and P computed by the algorithm 11...

$$D^{(0)} = \begin{pmatrix} \alpha & 3 & 7 & \alpha \\ \alpha & \alpha & 5 & 2 \\ \alpha & \alpha & \alpha & \alpha \\ 4 & \alpha & 1 & \alpha \end{pmatrix} \qquad P^{(0)} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 0 \\ 4 & 0 & 4 & 0 \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} \alpha & 3 & 7 & \alpha \\ \alpha & \alpha & 5 & 2 \\ \alpha & \alpha & \alpha & \alpha \\ 4 & 7 & 1 & \alpha \end{pmatrix} \qquad P^{(1)} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 0 \\ 4 & 1 & 4 & 0 \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} \alpha & 3 & 7 & 5 \\ \alpha & \alpha & 5 & 2 \\ \alpha & \alpha & \alpha & \alpha \\ 4 & 7 & 1 & \alpha \end{pmatrix} \qquad P^{(2)} = \begin{pmatrix} 0 & 1 & 1 & 2 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 0 \\ 4 & 1 & 4 & 0 \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} \alpha & 3 & 7 & 5 \\ \alpha & \alpha & 5 & 2 \\ \alpha & \alpha & \alpha & \alpha \\ 4 & 7 & 1 & \alpha \end{pmatrix} \qquad P^{(3)} = \begin{pmatrix} 0 & 1 & 1 & 2 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 0 \\ 4 & 1 & 4 & 0 \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} \alpha & 3 & 6 & 5 \\ \alpha & \alpha & 5 & 2 \\ \alpha & \alpha & \alpha & \alpha \\ 4 & 7 & 1 & \alpha \end{pmatrix} \qquad P^{(4)} = \begin{pmatrix} 0 & 1 & 4 & 2 \\ 4 & 0 & 4 & 2 \\ 0 & 0 & 0 & 0 \\ 4 & 1 & 4 & 0 \end{pmatrix}$$

Figure 11.18: Sequences of matrices computed by Algorithm 11.9

...e interpretation of the above matrices is as follows:

Paths	Its weight
$1 \rightarrow 2$	3
$1 \rightarrow 2 \rightarrow 4 \rightarrow 3$	6
$1 \rightarrow 2 \rightarrow 4$	5
$2 \rightarrow 4 \rightarrow 1$	6
$2 \rightarrow 4 \rightarrow 3$	3
$2 \rightarrow 4$	2
No path from 3 to 1 exists	–
No path from 3 to 2 exists	–
No path from 3 to 4 exists	–
$4 \rightarrow 1$	4
$4 \rightarrow 1 \rightarrow 2$	7
$4 \rightarrow 3$	1

...e implementation of the algorithm 11.10 and 11.9 is given in listing 11.19.

Listing 11.19:

```
#include <iostream.h>
#include <iomanip.h>
#define MAX 8
#define NIL 0
#define INFINITY  3200
void printPaths(int p[][MAX], int i, int j);
void FloydWarshall(int w[][MAX], int n);
void main()
{
  int w[MAX][MAX], i, j, n;
  cout << "\nEnter number of nodes in the graph : ";
  cin >> n;
  cout << "\nEnter weight of the edge "
       << "(0 in case there is no edge)\n\n";
  for ( i = 1; i <= n; i++ ) {
    for ( j = 1; j <= n; j++ ) {
      cout << "Enter weight for edge ("
```

```
            cin >> w[i][j];
        }
    }
    FloydWarshall(w, n);
}
void FloydWarshall( int w[][MAX], int n )
{
    int d[MAX][MAX], p[MAX][MAX], i, j, k;
    for ( i = 1; i <= n; ++i )
        for ( j = 1; j <= n; ++j ) {
            if ( w[i][j] == 0 ) {
                d[i][j] = INFINITY;
                p[i][j] = NIL;
            } else {
                d[i][j] = w[i][j];
                p[i][j] = i;
            }
        }
    for ( k = 1; k <= n; k++ ) {
        for ( i = 1; i <= n; i++ ) {
            for ( j = 1; j <= n; j++ ) {
                if ( d[i][j] <= ( d[i][k] + d[k][j] ) )
                    p[i][j] = p[i][j];
                else
                    p[i][j] = p[k][j];
                if ( d[i][j] <= ( d[i][k] + d[k][j] ) )
                    d[i][j] = d[i][j];
                else
                    d[i][j] = d[i][k] + d[k][j];
            }
        }
    }
    cout << "\n\nShortest paths are\n";
    for ( i = 1; i <= n; i++ ) {
        cout << endl;
        for ( j = 1; j <= n; j++ ) {
            if ( i != j ) {
                cout << endl;
                printPaths( p, i, j );
            }
        }
    }
}
void printPaths( int p[][MAX], int i, int j )
{
    if ( i == j )
        cout << i;
    else if ( p[i][j] == NIL )
        cout << "No path from " << i << " to " << j << " exists";
    else {
        printPaths( p, i, p[i][j] );
        cout << " -> " << j;
    }
}
```


11.6 Transitive Closure of a Directed Graph

Given a directed graph $G = (V, E)$ with vertex set $V = \{1, 2, 3, \ldots, n\}$, we may wish to find out whether there is a path in G from vertex i to vertex j for all pairs $i, j \in V$. The answer of the above question can be found by finding the transitive closure of given graph.

Transitivity says that if there exists a path from vertex i to vertex k and vertex k to vertex j, then there exists a path from vertex i to vertex j as shown in the following figure with a dashed line.

Figure 11.19: Illustration of transitivity

The *transitive closure* of a graph $G = (V, E)$ is defined as graph $G^* = (V, E^*)$, where

$E^* = \{ (i, j) :$ there exists a path from vertex i to vertex j in G$\}$

One way to compute the transitive closure is to assign a weight of 1 to each edge of E and run the Floyd-Warshall algorithm. If there is a path from vertex i to vertex j, we get $d_{ij} < n$. Otherwise, we get $d_{ij} = \infty$.

There is another way, though quite similar, that save time and space in practice. This method involves the substitution of the logical operations $\vee$ (OR) and $\wedge$ (AND) for arithmetic operations *min* and + in the Floyd-Warshall algorithm.

For $i, j, k = 1, 2, 3, \ldots, n$, we define $t_{ij}^k = 1$, if there exists a path in graph G from vertex i to vertex j with intermediate vertices in the set $\{1, 2, 3, \ldots, k\}$, and 0 otherwise. We construct the transitive closure $G^* = (V, E^*)$ by putting edge (i, j) into E^* iff $t_{ij}^n = 1$.

A recursive definition for $t_{ij}^{(k)}$ is as given below.

When $k = 0$

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } (i,j) \notin E \\ 1 & \text{if } (i,j) \in E \end{cases}$$

And for $k \geq 1$

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$$

We compute the matrices $T^{(k)} = \left(t_{ij}^{(k)}\right)$ in the increasing order of k.

Algorithm

TransitiveClosure(a, t, n)

Here a is $n \times n$ adjacency-matrix representing directed input graph G. This algorithm computes the transitive closure and stores in matrix t also of size $n \times n$.

```
Begin
  for i = 1 to n by 1 do
     for i = 1 to n by 1 do
        if ( a[i,j] = 1 ) then
           Set t[i,j] = 1
        else
           Set t[i,j] = 0
        endif
     endfor
  endfor
  for k = 1 to n by 1 do
     for i = 1 to n by 1 do
        for j = 1 to n by 1 do
           Set t[i,j] = t[i,j] ∨ ( t[i,k] ∧ t[k,j] )
        endfor
     endfor
  endfor
End.
```

Implementation of this is left as an exercise for the readers.

11.6 ARTICULATION POINTS, BRIDGES, AND BICONNECTED COMPONENTS

Let $G = (V, E)$ be a connected, undirected graph. An *articulation point*, also called a *cut point*, of G is a vertex whose removal disconnects G. A *bridge* of G is an edge whose removal disconnects G. A biconnected component of G is a maximal set of edges such that any two edges in the set lie on a common simple cycle.

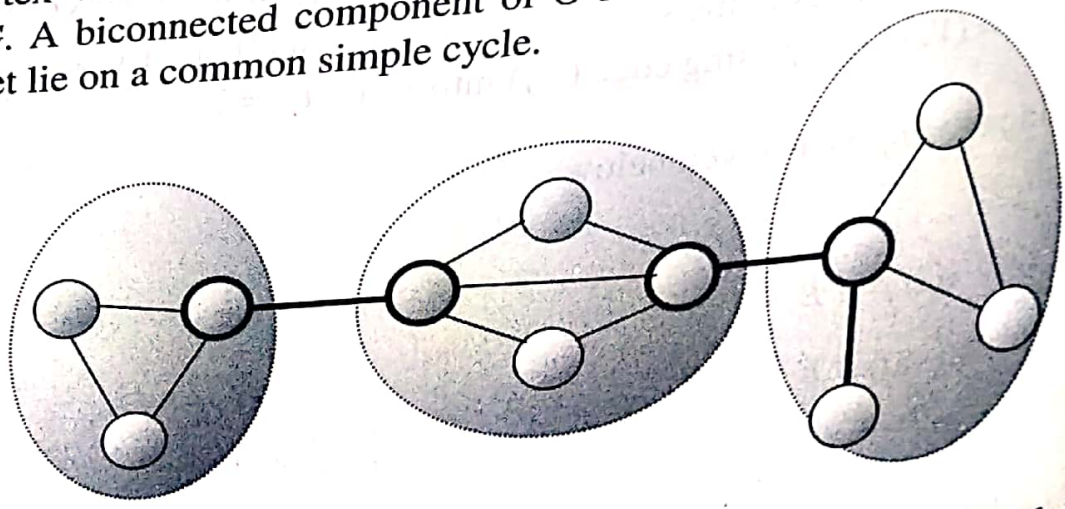
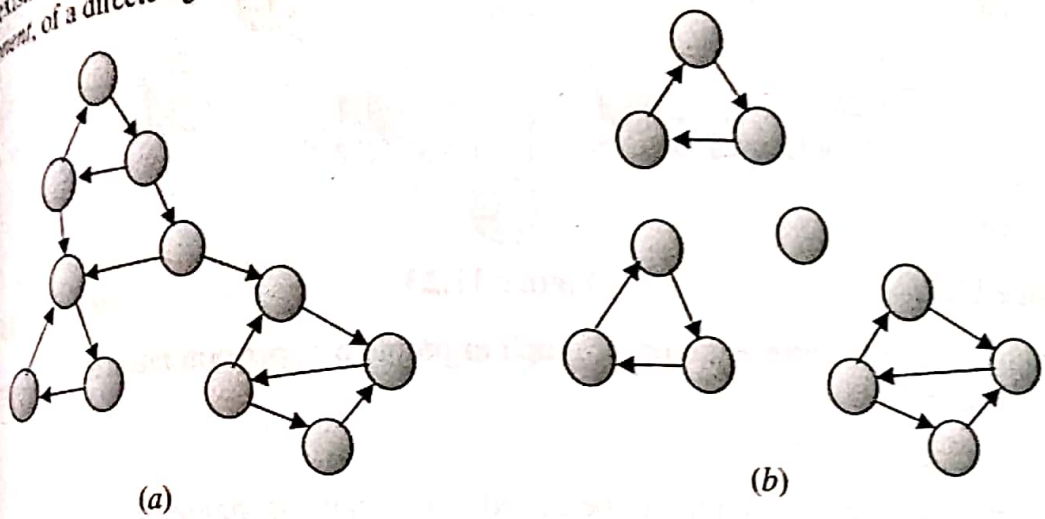


Figure 11.20 The articulation points and bridges are shown with thick lines, while biconnected components are shown inside dashed circles.

We can find articulation points, bridges, and biconnected components using depth-first search

STRONGLY CONNECTED COMPONENTS

...ected graph $G = (V, E)$ is *strongly connected* if and only if for each pair of vertices i and j ... exists a path from i to j and j to i. A *strongly connected component*, or simply *strong* ...ponent, of a directed graph is a maximal strongly connected subgraph.



(*a*) (*b*)

Figure 11.21: (*a*) A directed graph, (*b*) Its strong components

EULERIAN TOUR

...*Eulerian tour* through an undirected graph is a path whose edge list contains each edge of ...graph exactly once. An *Eulerian graph* is a graph that possesses an Eulerian tour. It has ...proved that an undirected graph is Eulerian graph if and only if it is connected and has ...er zero or two vertices with an odd degree.

HAMILTONIAN TOUR

...*Hamiltonian tour* through an undirected graph is a path whose vertex list contains each ...ex of the graph exactly once. A *Hamiltonian graph* is a graph that possesses a Hamiltonian ...No simple condition exists that can be used to characterize Hamiltonian graph.

REVIEW EXERCISES

1. Is graph a linear or non-linear data structure?
2. How graphs can be represented in computer memory? Give relative merits and de-merits of each representation scheme.
3. We have an application using which a railway passenger can know whether there exists a path from a given city to another city, *i.e.*, does there exists a railway link direct or indirect. Which way graph should be represented in memory?
4. How breadth-first search differs from depth-first search? Can we perform these searches on weighted graphs? Justify your answer.
5. Show the possible adjacency matrix and adjacency list representation of graph of figure 11.22.

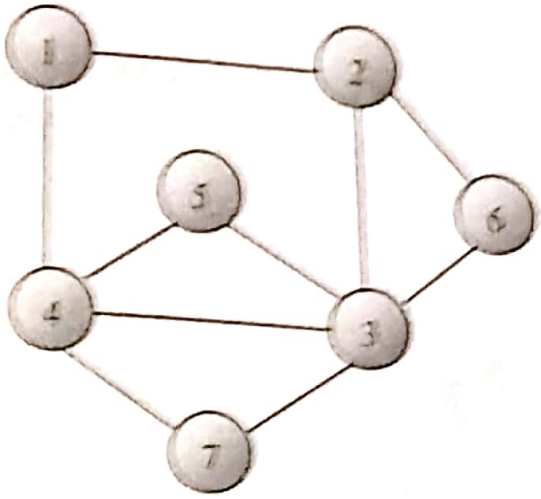


Figure 11.22

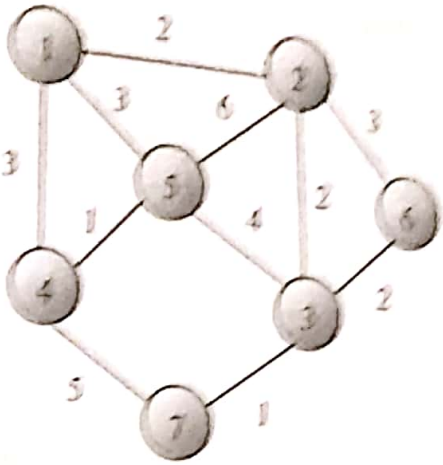


Figure 11.23

Figure 11.24

6. Using adjacency list representation of graph in problem 5, perform breadth-first search and depth-first search.
7. What is topological sort?
8. What is spanning tree? When it can be called a minimum spanning tree? Find all possible spanning trees and a minimum spanning tree of graph in figure 11.23.
9. Show the progress of Prim's algorithm to find the minimum-spanning tree for a graph shown in exercise 7.
10. Show the progress of Dijkstra's algorithm to find the shortest path from vertex 4 to vertex 1 in a graph shown in exercise 7.
11. Show the progress of Floyd-Warshall's algorithm to find the shortest paths between all pairs of vertices of graph in figure 11.24.

PROGRAMMING EXERCISES

1. The *degree* of a vertex in an undirected graph is defined as the number of edges incident on it. Write a program to find the *degree* of each vertex.
2. The *indegree* of a vertex in a directed graphs is defined the number of edges in which it appears as a destination vertex. Likewise, the *outdegree* of a vertex in a directed graph is defined the number of edges in which it appears as a source vertex. Write a program to find the *indegree* and *outdegree* of each vertex.
3. An undirected graph is said to be fully connected, if every vertex is connected to every other vertex. Write a function that takes an undirected graph as its parameter and returns value 1 if graph is fully connected else returns value 0.
4. Write a program to insert a new edge in an undirected graph.
5. Write a program to find the transitive closure of an undirected as well as directed graph.
6. Write a program to find the vertices connected by edge with maximum weight in a directed graph.