STACK:

A stack is one of the most important and useful non-primitive linear data structure in computer science. It is an ordered collection of items into which new data items may be added/inserted and from which items may be deleted at only one end, called the top of the stack. As all the addition and deletion in a stack is done from the top of the stack, the last added element will be first removed from the stack. That is why the stack is also called Last-in-First-out (LIFO). Note that the most frequently accessible element in the stack is the top most elements, whereas the least accessible element is the bottom of the stack.
The operation of the stack can be illustrated as in Fig. 3.1.

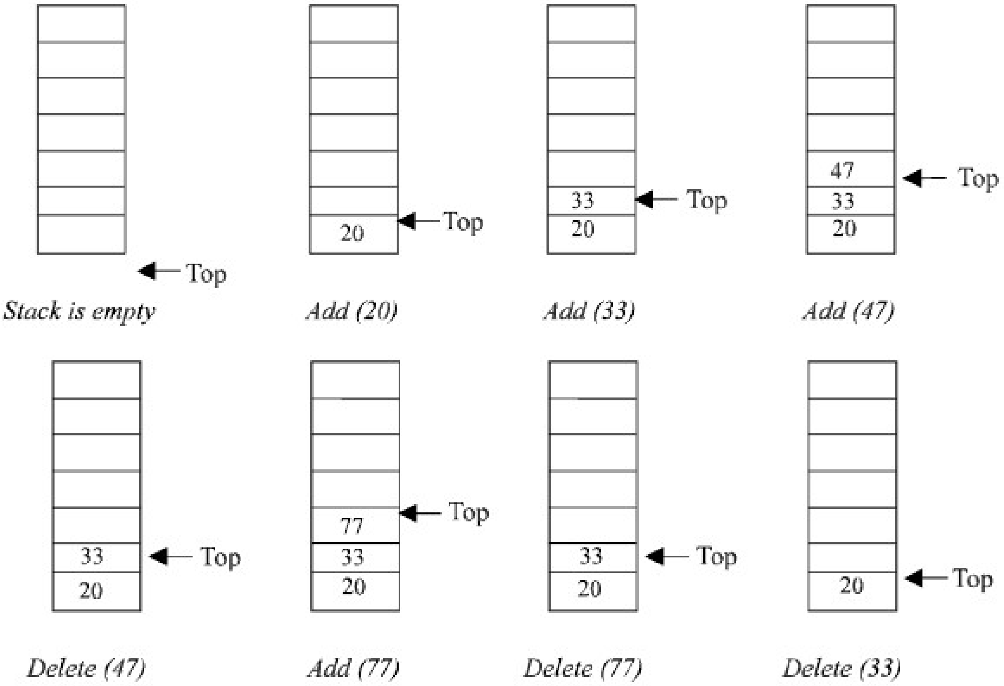


Fig. 3.1. Stack operation.

The insertion (or addition) operation is referred to as push, and the deletion (or remove) operation as pop. A stack is said to be empty or underflow, if the stack contains no elements. At this point the top of the stack is present at the bottom of the stack. And it is overflow when the stack becomes full, i.e., no other elements can be pushed onto the stack. At this point the top pointer is at the highest location of the stack.

OPERATIONS PERFORMED ON STACK

The primitive operations performed on the stack are as follows:

PUSH: The process of adding (or inserting) a new element to the top of the stack is called PUSH operation. Pushing an element to a stack will add the new element at the top. After every push operation the top is incremented by one. If the array is full and no new element can be accommodated, then the stack overflow condition occurs.

POP: The process of deleting (or removing) an element from the top of stack is called POP operation. After every pop operation the stack is decremented by one. If there is no element in the stack and the pop operation is performed then the stack underflow condition occurs.

STACK IMPLEMENTATION

Stack can be implemented in two ways:

1. Static implementation (using arrays)
2. Dynamic implementation (using pointers)

Static implementation uses arrays to create stack. Static implementation using arrays is a very simple technique but is not a flexible way, as the size of the stack has to be declared during the program design, because after that, the size cannot be varied (*i.e.*, increased or decreased). Moreover static implementation is not an efficient method when resource optimization is concerned (*i.e.*, memory utilization). For example a stack is implemented with array size 50. That is before the stack operation begins, memory is allocated for the array of size 50. Now if there are only few elements (say 30) to be stored in the stack, then rest of the statically allocated memory (in this case 20) will be wasted, on the other hand if there are more number of elements to be stored in the stack (say 60) then we cannot change the size array to increase its capacity.

The above said limitations can be overcome by dynamically implementing (is also called linked list representation) the stack using pointers.

STACK USING ARRAYS

Implementation of stack using arrays is a very simple technique. Algorithm for pushing (or add or insert) a new element at the top of the stack and popping (or delete) an element from the stack is given below.

Algorithm for push

Suppose STACK[SIZE] is a one dimensional array for implementing the stack, which will hold the data items. TOP is the pointer that points to the top most element of the stack. Let DATA is the data item to be pushed.

```
1. If TOP = SIZE – 1, then:
    (a) Display “The stack is in overflow condition”
    (b) Exit
2. TOP = TOP + 1
3. STACK [TOP] = ITEM
4. Exit
```

Algorithm for pop

Suppose STACK[SIZE] is a one dimensional array for implementing the stack, which will hold the data items. TOP is the pointer that points to the top most element of the stack. DATA is the popped (or deleted) data item from the top of the stack.

```
1. If TOP < 0, then
    (a) Display “The Stack is empty”
    (b) Exit
2. Else remove the Top most element
3. DATA = STACK[TOP]
4. TOP = TOP – 1
5.Exit.
```

```
//THIS PROGRAM IS TO DEMONSTRATE THE OPERATIONS PERFORMED ON THE STACK AND IT IS
//IMPLEMENTATION USING ARRAYS
#include<stdio.h>
#include<conio.h>
//Defining the maximum size of the stack
#define MAXSIZE 100
//Declaring the stack array and top variables in a structure
struct stack
{
int stack[MAXSIZE];
int Top;
};
//type definition allows the user to define an identifier that would represent an existing data type. The user-defined data type
//identifier can later be used to declare variables.
typedef struct stack NODE;

//This function will add/insert an element to Top of the stack
void push(NODE *pu)
```

```
{
int item;
//if the top pointer already reached the maximum allowed size then we can say that the stack is full or overflow
if (pu->Top == MAXSIZE–1)
{
printf(“\nThe Stack Is Full”);
getch();
}
//Otherwise an element can be added or inserted by incrementing the stack pointer Top as follows
else
{
printf(“\nEnter The Element To Be Inserted = ”);
scanf(“%d”,&item);
pu->stack[++pu->Top]=item;
}
}
//This function will delete an element from the Top of the stack
void pop(NODE *po)
{
int item;
//If the Top pointer points to NULL, then the stack is empty That is NO element is there to delete or pop
if (po->Top == -1)
printf(“\nThe Stack Is Empty”);
//Otherwise the top most element in the stack is popped or deleted by decrementing the Top pointer
else
{
item=po->stack[po->Top--];
printf(“\nThe Deleted Element Is = %d”,item);
}
}
//This function to print all the existing elements in the stack
void traverse(NODE *pt)
{
int i;
//If the Top pointer points to NULL, then the stack is empty. That is NO element is there to delete or pop
if (pt->Top == -1)
printf(“\nThe Stack is Empty”);
//Otherwise all the elements in the stack is printed
else
{
printf(“\n\nThe Element(s) In The Stack(s) is/are...”);
for(i=pt->Top; i>=0; i--)
printf (“\n %d”,pt->stack[i]);
}
}
void main( )
{
int choice;
char ch;
//Declaring an pointer variable to the structure
NODE *ps;
//Initializing the Top pointer to NULL
ps->Top=–1;
do
{
clrscr();
//A menu for the stack operations
printf(“\n1. PUSH”);
printf(“\n2. POP”);
printf(“\n3. TRAVERSE”);
printf(“\nEnter Your Choice = ”);
scanf (“%d”, &choice);
switch(choice)
{
case 1://Calling push() function by passing
//the structure pointer to the function
push(ps);
break;
case 2://calling pop() function
pop(ps);
break;
case 3://calling traverse() function
traverse(ps);
break;
default:
printf("\nYou Entered Wrong Choice") ;
}
printf("\n\nPress (Y/y) To Continue = ");
//Removing all characters in the input buffer for fresh input(s), especially <<Enter>> key
fflush(stdin);
scanf("%c",&ch);
}while(ch == 'Y' || ch == 'y');
}
```

APPLICATIONS OF STACKS

There are a number of applications of stacks; three of them are discussed briefly in the preceding sections. Stack is internally used by compiler when we implement (or execute) any recursive function. If we want to implement a recursive function non-recursively, stack is programmed explicitly. Stack is also used to evaluate a mathematical expression and to check the parentheses in an expression.

RECURSION

Recursion occurs when a function is called by itself repeatedly; the function is called recursive function. The general algorithm model for any recursive function contains the following steps:

1. *Prologue:* Save the parameters, local variables, and return address.
2. *Body:* If the base criterion has been reached, then perform the final computation and go to step 3; otherwise, perform the partial computation and go to step 1 (initiate a recursive call).
3. *Epilogue:* Restore the most recently saved parameters, local variables, and return address.

RECURSION vs ITERATION

Recursion of course is an elegant programming technique, but not the best way to solve a problem, even if it is recursive in nature. This is due to the following reasons:

1. It requires stack implementation.
2. It makes inefficient utilization of memory, as every time a new recursive call is made a new set of local variables is allocated to function.
3. Moreover it also slows down execution speed, as function calls require jumps, and saving the current state of program onto stack before jump.

Though inefficient way to solve general problems, it is too handy in several problems as discussed in the starting of this chapter. It provides a programmer with certain pitfalls, and quite sharp concepts about programming. Moreover recursive functions are often easier to implement d maintain, particularly in case of data structures which are by nature recursive. Such data structures are queues, trees, and linked lists. Given below are some of the important points, which differentiate iteration from recursion.

No	*Iteration*	*Recursion*
1	It is a process of executing a statement or a set of statements repeatedly, until some specified condition is specified.	Recursion is the technique of defining anything in terms of itself.

2	Iteration involves four clear-cut Steps like initialization, condition, execution, and updating.	There must be an exclusive if statement inside the recursive function, specifying stopping condition.
3	Any recursive problem can be solved iteratively.	Not all problems have recursive solution.
4	Iterative counterpart of a problem is more efficient in terms of memory utilization and execution speed.	Recursion is generally a worse option to go for simple problems, or problems not recursive in nature.

DISADVANTAGES OF RECURSION

1. It consumes more storage space because the recursive calls along with automatic variables are stored on the stack.
2. The computer may run out of memory if the recursive calls are not checked.
3. It is not more efficient in terms of speed and execution time.
4. According to some computer professionals, recursion does not offer any concrete advantage over non-recursive procedures/functions.
5. If proper precautions are not taken, recursion may result in non-terminating iterations.
6. Recursion is not advocated when the problem can be through iteration. Recursion may be treated as a software tool to be applied carefully and selectively.

POLISH NOTATION

The process of writing the operators of an expression either before their operands or after their operands is called Polish notation. It was given by Polish mathematician JanLuksiewicz. The fundamental property of polish notation is that the order in which the operations are to be performed is completely determined by the positions of the operators and operands in the expression.

There are basically three types of notation for an expression (mathematical expression; An expression is defined as the number of operands or data items combined with several operators.)

1. Infix notation
2. Prefix notation
3. Postfix notation

The *infix notation* is what we come across in our general mathematics, where the operator is written in-between the operands. For example: The expression to add two numbers A and B is written in infix notation as:

A + B

Note that the operator '+' is written in between the operands A and B.

The *prefix notation* is a notation in which the operator(s) is written before the operands,

The same expression when written in prefix notation looks like:

+ A B

As the operator '+' is written before the operands A and B, this notation is called prefix (pre means before).

In the *postfix notation* the operator(s) are written after the operands, so it is called the postfix notation (post means after), it is also known as *suffix notation* or *reverse polish notation*. The above expression if written in postfix expression looks like:

A B +

The prefix and postfix notations are not really as awkward to use as they might look.

For example, a C function to return the sum of two variables A and B (passed as argument) is called or invoked by the instruction: add(A, B)

Note that the operator *add* (name of the function) precedes the operands A and B. Because the postfix notation is most suitable for a computer to calculate any expression (due to its reverse characteristic), and is the universally accepted notation for designing Arithmetic and Logical Unit (ALU) of the CPU (processor). Therefore it is necessary to study the postfix notation. Moreover the postfix notation is the way computer looks towards arithmetic expression, any expression entered into the computer is first converted into postfix notation, stored in stack and then calculated.

AdvantagesF of using postfix notation

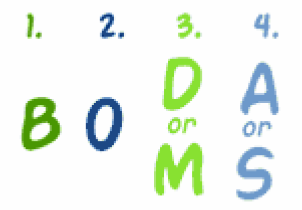
Human beings are quite used to work with mathematical expressions in *infix* notation, which is rather complex. One has to remember a set of nontrivial rules while using this notation and it must be applied to expressions in order to determine the final value. These rules include precedence, BODMAS , and associativity. Using infix notation, one cannot tell the order in which operators should be applied. Whenever an infix expression consists of more than one operator, the precedence rules (BODMAS) should be applied to decide which operator (and operand associated with that operator) is evaluated first. But in a postfix expression operands appear before the operator, so there is no need for operator precedence and other rules. As soon as an operator appears in the postfix expression during scanning of postfix expression the topmost operands are popped off and are calculated by applying the encountered operator. Place the result back onto the stack; likewise at the end of the whole operation the final result will be there in the stack.

Meaning of BODOMAS

B	**B**rackets first
O	**O**rders (ie Powers and Square Roots, etc.)
DM	**D**ivision and **M**ultiplication (left-to-right)
AS	**A**ddition and **S**ubtraction (left-to-right)

Divide and Multiply rank equally (and go left to right).

Add and Subtract rank equally (and go left to right)



After you have done "B" and "O", just go from left to right doing any "D" ***or*** "M" as you find them.

Then go from left to right doing any "A" ***or*** "S" as you find them.

Examples

Example: How do you work out **3 + 6 × 2** ?
Multiplication before **A**ddition:
First **6 × 2 = 12**, then **3 + 12 = 15**

Example: How do you work out **(3 + 6) × 2** ?
Brackets first: First **(3 + 6) = 9**, then **9 × 2 = 18**

Example: How do you work out **12 / 6 × 3 / 2** ?
Multiplication and **D**ivision rank equally, so just go left to right:
First **12 / 6 = 2**, then **2 × 3 = 6**, then **6 / 2 = 3**

Oh, yes, and what about 7 + (6 × 5^2 + 3) ?

7 + (6 × 5^2 + 3)	
7 + (6 × 25 + 3)	Start inside Brackets, and then use "Orders" First
7 + (150 + 3)	Then Multiply
7 + (153)	Then Add
7 + 153	Brackets completed, last operation is add
160	DONE

Notation Conversions

Let A + B * C be the given expression, which is an infix notation. To calculate this expression for values 4, 3, 7 for A, B, C respectively we must follow certain rule (called BODMAS in general mathematics) in order to have the right result. For example:

A + B * C = 4 + 3 * 7 = 7 * 7 = 49

The answer is not correct; multiplication is to be done before the addition, because multiplication has higher precedence over addition. This means that an expression is calculated according to the operator's precedence not the order as they look like. The error in the above calculation occurred, since there were no braces to define the precedence of the operators. Thus expression A + B * C can be interpreted as A + (B * C). Using this alternative method we can convey to the computer that multiplication has higher precedence over addition.

Operator precedence

Exponential operator	^	Highest precedence
Multiplication/Division	*, /	Next precedence
Addition/Subtraction	+, -	Least precedence

CONVERTING INFIX TO POSTFIX EXPRESSION

The method of converting infix expression A + B * C to postfix form is:

A + B * C Infix Form
A + (B * C) Parenthesized expression
A + (B C *) Convert the multiplication
A (B C *) + Convert the addition
A B C * + Postfix form

The rules to be remembered during infix to postfix conversion are:

1. Parenthesize the expression starting from left to light.
2. During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized. For example in the above expression
B * C is parenthesized first before A + B.
3. The sub-expression (part of expression), which has been converted into postfix, is to be treated as single operand.
4. Once the expression is converted to postfix form, remove the parenthesis.

Give postfix form for A + [(B + C) + (D + E) * F] / G

Solution. Evaluation order is

A + { [(BC +) + (DE +) * F] / G}
A + { [(BC +) + (DE + F *] / G}
A + { [(BC + (DE + F * +] / G} .
A + [BC + DE + F *+ G /]
ABC + DE + F * + G / + Postfix Form

Give postfix form for (A + B) * C / D + E ^ A / B

Solution. Evaluation order is

[(AB +) * C / D] + [(EA ^) / B]
[(AB +) * C / D] + [(EA ^) B /]

[(AB +) C * D /] + [(EA ^) B /]

(AB +) C * D / (EA ^) B / +

AB + C * D / EA ^ B / + Postfix Form

Algorithm

Suppose P is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Q. Besides operands and operators, P (infix notation) may also contain left and right parentheses. We assume that the operators in P consists of only exponential (^), multiplication (*), division (/), addition (+) and subtraction (-).

The algorithm uses a stack to temporarily hold the operators and left parentheses. The postfix expression Q will be constructed from left to right using the operands from P and operators, which are removed from stack. We begin by pushing a left parenthesis onto stack and adding a right parenthesis at the end of P. the algorithm is completed when the stack is empty.

```
1. Push “(” onto stack, and add“)” to the end of P.
2. Scan P from left to right and repeat Steps 3 to 6 for each element of P until the stack is empty.
3. If an operand is encountered, add it to Q.
4. If a left parenthesis is encountered, push it onto stack.
5. If an operator is encountered, then:
(a) Repeatedly pop from stack and add P each operator (on the top of stack), which has the same precedence as, or higher precedence than operator
(b) Add operator to stack.
6. If a right parenthesis is encountered, then:
(a) Repeatedly pop from stack and add to P (on the top of stack until a left parenthesis is encountered.
(b) Remove the left parenthesis. [Do not add the left parenthesis to P.]
7. Exit.
```

EVALUATING POSTFIX EXPRESSION

Following algorithm finds the RESULT of an arithmetic expression P written in postfix notation. The following algorithm, which uses a STACK to hold operands, evaluates P.

Algorithm

```
1. Add a right parenthesis “)” at the end of P. [This acts as a sentinel.]
2. Scan P from left to right and repeat Steps 3 and 4 for each element of P until the sentinel “)” is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator  is encountered, then:
(a) Remove the two top elements of STACK, where A is the top element and B is the next-to-top element.
(b) Evaluate B  operator A.
(c) Place the result on to the STACK.
5. Result equal to the top element on STACK.
6. Exit.
```

QUEUE

A queue is logically a *first in first out (FIFO or first come first serve)* linear data structure. The concept of queue can be understood by our real life problems. For example a customer come and join in a queue to take the train ticket at the end (rear) and the ticket is issued from the front end of queue. That is, the customer who arrived first will receive the ticket first. It means the customers are serviced in the order in which they arrive at the service centre. It is a homogeneous collection of elements in which new elements are added at one end called *rear*, and the existing elements are deleted from other end called *front*.

The basic operations that can be performed on queue are

1. Insert (or add) an element to the queue (push)
2. Delete (or remove) an element from a queue (pop)

Push operation will insert (or add) an element to queue, at the rear end, by incrementing the array index. Pop operation will delete (or remove) from the front end by decrementing the array index and will assign the deleted value to a variable. Total number of elements present in the queue is front-rear+1, when implemented using arrays. Following figure will illustrate the basic operations on queue.

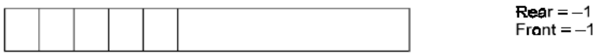


Fig. 4.1. Queue is empty.

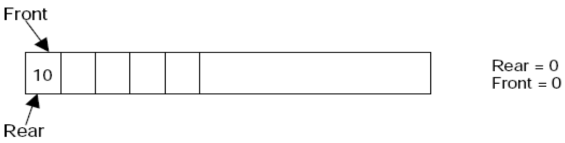
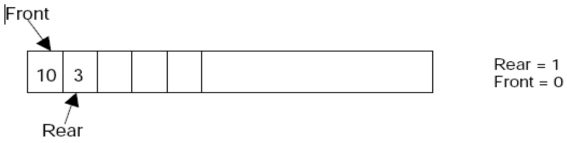


Fig. 4.2. push(10)



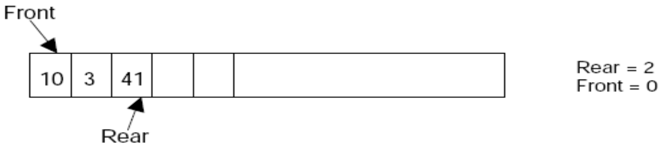


Fig. 4.4. push(41)

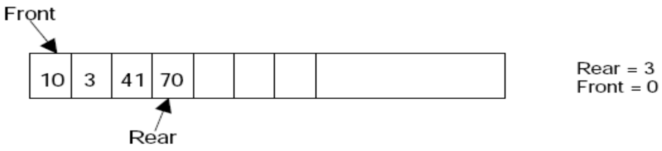


Fig. 4.5. push(70)

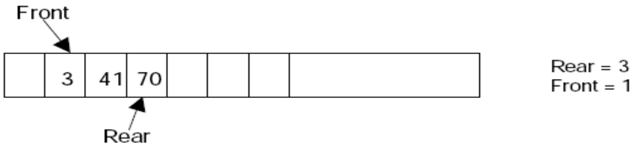


Fig. 4.6. x = pop() (*i.e.*; x = 10)

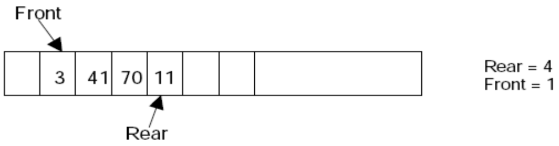


Fig. 4.7. push(11)

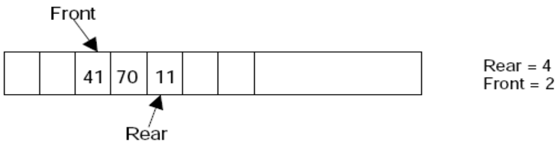


Fig. 4.8. x = pop() (*i.e.*; x = 3)

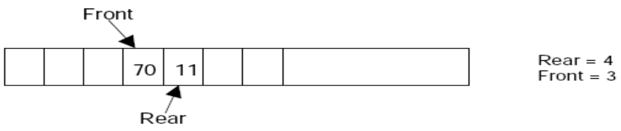


Fig. 4.9. *x* = pop() (*i.e.*, *x* = 41)

Queue can be implemented in two ways:

1. Using arrays (static)
2. Using pointers (dynamic)

Implementation of queue using pointers will be discussed in chapter 5. Let us discuss underflow and overflow conditions when a queue is implemented using arrays.

If we try to pop (or delete or remove) an element from queue when it is empty, underflow occurs. It is not possible to delete (or take out) any element when there is no element in the queue.

Suppose maximum size of the queue (when it is implemented using arrays) is 50. If we try to push (or insert or add) an element to queue, overflow occurs. When queue is full it is naturally not possible to insert any more elements

ALGORITHM FOR QUEUE OPERATIONS

Let Q be the array of some specified size say SIZE

INSERTING AN ELEMENT INTO THE QUEUE

```
1. Initialize front=0 rear = –1
2. Input the value to be inserted and assign to variable “data”
3. If (rear >= SIZE)
(a) Display “Queue overflow”
(b) Exit
4. Else
(a) Rear = rear +1
5. Q[rear] = data
6. Exit
```

DELETING AN ELEMENT FROM QUEUE

```
1. If (rear< front)
(a) Front = 0, rear = –1
(b) Display “The queue is empty”
(c) Exit
```

```
2. Else
(a) Data = Q[front]
3. Front = front +1
4. Exit
```

Program for queue implementation in C

```
#include<conio.h>
#include<stdio.h>
#include<process.h>

#define MAX 50

int queue_arr[MAX];
int rear = –1;
int front = –1;

//This function will insert an element to the queue
void insert ()
{
            int added_item;
            if (rear==MAX-1)
            {
                   printf("\nQueue Overflow\n");
                   getch();
                   return;
            }
            else
            {
                   if (front==–1)  /*If queue is initially empty */
                          front=0;
                   printf(“\nInput the element for adding in queue: ”);
                    scanf(“%d”, &added_item);
                    rear=rear+1;
                    //Inserting the element
                    queue_arr[rear] = added_item ;
      }
 }/*End of insert()*/

//This function will delete (or pop) an element from the queue
void del()
{
            if (front == –1 || front > rear)
            {
                   printf ("\nQueue Underflow\n");
                   return;
            }
            else
            {
                   //deleteing the element
      printf ("\nElement deleted from queue is : %d\n",
      queue_arr[front]);
                   front=front+1;
            }
}/*End of del()*/

//Displaying all the elements of the queue
void display()
{
            int i;
            //Checking whether the queue is empty or not
            if (front == –1 || front > rear)
            {
                   printf (“\nQueue is empty\n”);
                   return;
```

```
          }
          else
          {
                printf("\nQueue is :\n");
                for(i=front;i<= rear;i++)
                      printf("%d  ",queue_arr[i]);
                printf("\n");
          }
}/*End of display() */

void main()
{
          int choice;
          while (1)
          {
                clrscr();


 //Menu options
 printf("\n1.Insert\n");
 printf("2.Delete\n");
 printf("3.Display\n");
 printf("4.Quit\n");
 printf("\nEnter your choice:");
 scanf("%d", & choice);
 switch(choice)
 {
 case 1 :
        insert();
        break;
 case 2:
        del();
        getch();
        break;
 case 3:
        display();
        getch();
        break;
 case 4:
        exit(1);
 default:
        printf ("\n Wrong choice\n");
        getch();
 }/*End of switch*/

}/*End of while*/

}/* End of main() */
```

Suppose a queue Q has maximum size 5, say 5 elements pushed and 2 elements popped.

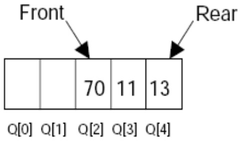


Fig. 4.10

Now if we attempt to add more elements, even though 2 queue cells are free, the elements cannot be pushed. Because in a queue, elements are always inserted at the *rear* end and hence *rear* points to last location of the queue array Q[4]. That is queue is full (overflow condition) though it is empty. This limitation can be overcome if we use circular queue.

OTHER QUEUES

There are three major variations in a simple queue. They are
1. Circular queue
2. Double ended queue (de-queue)
3. Priority queue
Priority queue is generally implemented using linked list
.

CIRCULAR QUEUE

In circular queues the elements Q[0],Q[1],Q[2] .... Q[*n* – 1] is represented in a circular fashion with Q[1] following Q[n]. A circular queue is one in which the insertion of a new element is done at the very first location of the queue if the last location at the queue is full.

Suppose Q is a queue array of 6 elements. Push and pop operation can be performed on circular. The following figures will illustrate the same.

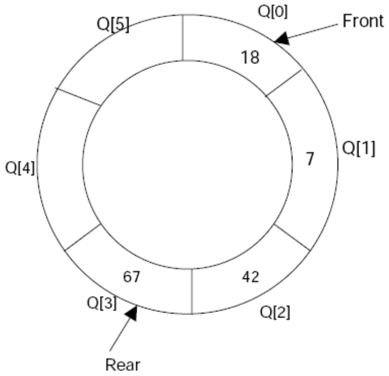


Fig. 4.11. A circular queue after inserting 18, 7, 42, 67.

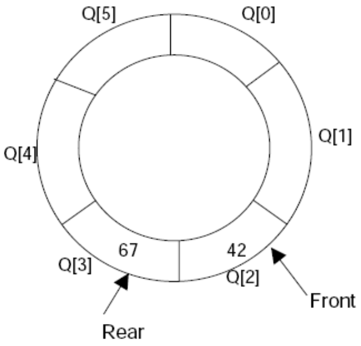


Fig. 4.12. A circular queue after popping 18, 7

After inserting an element at last location Q[5], the next element will be inserted at the very first location (*i.e.*, Q[0]) that is circular queue is one in which the first element comes just after the last element.

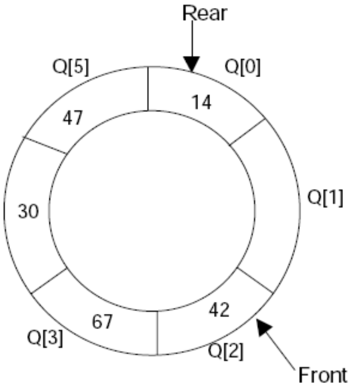


Fig. 4.13. A circular queue after pushing 30, 47, 14

At any time the position of the element to be inserted will be calculated by the relation Rear = (Rear + 1) % SIZE
After deleting an element from circular queue the position of the front end is calculated by the relation Front= (Front + 1) % SIZE
After locating the position of the new element to be inserted, *rear*, compare it with *front*. If (rear = front), the queue is full and cannot be inserted anymore

ALGORITHMS

Let Q be the array of some specified size say SIZE. FRONT and REAR are two pointers where the elements are deleted and inserted at two ends of the circular queue. DATA is the element to be inserted.

Inserting an element to circular Queue

```
1. Initialize FRONT = – 1; REAR = 1
2. REAR = (REAR + 1) % SIZE
3. If (FRONT is equal to REAR)
    (a) Display “Queue is full”
    (b) Exit
4. Else
    (a) Input the value to be inserted and assign to variable “DATA”
5. If (FRONT is equal to – 1)
    (a) FRONT = 0
    (b) REAR = 0
6. Q[REAR] = DATA
7. Repeat steps 2 to 5 if we want to insert more elements
8. Exit
```

Deleting an element from a circular queue

```
1. If (FRONT is equal to – 1)
    (a) Display “Queue is empty”
    (b) Exit
2. Else
    (a) DATA = Q[FRONT]
3. If (REAR is equal to FRONT)
    (a) FRONT = –1
    (b) REAR = –1
4. Else
    (a) FRONT = (FRONT +1) % SIZE
5. Repeat the steps 1, 2 and 3 if we want to delete more elements
6. Exit
```

PROGRAM

```
/// PROGRAM TO IMPLEMENT CIRCULAR QUEUE USING ARRAY IN C++
#include<conio.h>
#include<process.h>
#include<iostream.h>
#define MAX 50
//A class is created for the circular queue
class circular_queue
{
int cqueue_arr[MAX];
int front,rear;
public:
//a constructor is created to initialize the variables
circular_queue()
{
front = –1;
rear = –1;
}
//public function declarations
void insert();
void del();
void display();
};
//Function to insert an element to the circular queue
void circular_queue::insert()
{
int added_item;

//Checking for overflow condition
if ((front == 0 && rear == MAX-1) || (front == rear +1))
{
cout<<“\nQueue Overflow \n”;
getch();
return;
}
if (front == –1) /*If queue is empty */
{
front = 0;
rear = 0;
}
else
if (rear == MAX-1)/*rear is at last position of queue */
rear = 0;
else
rear = rear + 1;
cout<<“\nInput the element for insertion in queue:”;
```

```
cin>>added_item;
cqueue_arr[rear] = added_item;
}/*End of insert()*/
//This function will delete an element from the queue
void circular_queue::del()
{
//Checking for queue underflow
if (front == –1)
{
cout<<“\nQueue Underflow\n”;
return;
}
cout<<“\nElement deleted from queue is:”<<cqueue_arr[front]<<“\n”;
if (front == rear) /* queue has only one element */
{
front = –1;
rear = –1;
}
else
if(front == MAX-1)
front = 0;
else
front = front + 1;
}/*End of del()*/
//Function to display the elements in the queue
void circular_queue::display()
{
int front_pos = front,rear_pos = rear;
//Checking whether the circular queue is empty or not
if (front == –1)
{
cout<<“\nQueue is empty\n”;
return;
}

//Displaying the queue elements
cout<<“\nQueue elements:\n”;
if(front_pos <= rear_pos )
while(front_pos <= rear_pos)
{
cout<<cqueue_arr[front_pos]<<“, ”;
front_pos++;
}
else
{
while(front_pos <= MAX-1)
{
cout<<cqueue_arr[front_pos]<<“, ”;
front_pos++;
}
front_pos = 0;
while(front_pos <= rear_pos)
{
cout<<cqueue_arr[front_pos]<<“, ”;
front_pos++;
}
}/*End of else*/
cout<<“\n”;
}/*End of display() */
void main()
{
int choice;
```

```
//Creating the objects for the class
circular_queue co;
while(1)
{
clrscr();
//Menu options
cout <<“\n1.Insert\n”;
cout <<“2.Delete\n”;
cout <<“3.Display\n";
cout <<“4.Quit\n”;
cout <<“\nEnter your choice: ”;
cin>>choice;
switch(choice)
{
case 1:
co.insert();
break;
case 2 :
co.del();
getch();
break;
case 3:
co.display();
getch();
break;
case 4:
exit(1);
default:
cout<<“\nWrong choice\n”;
getch();
}/*End of switch*/
}/*End of while*/
}/*End of main()*/
```

DEQUES

A deque is a homogeneous list in which elements can be added or inserted (called push operation) and deleted or removed from both the ends (which is called pop operation). ie; we can add a new element at the rear or front end and also we can remove an element from both front and rear end. Hence it is called Double Ended Queue.

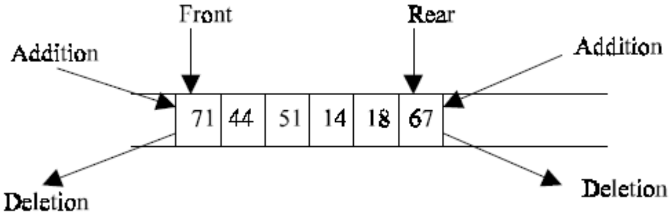


Fig. 4.14. A deque

There are two types of deque depending upon the restriction to perform insertion or deletion operations at the two ends. They are

1. Input restricted deque
2. Output restricted deque

An input restricted deque is a deque, which allows insertion at only 1 end, rear end, but allows deletion at both ends, rear and front end of the lists.
An output-restricted deque is a deque, which allows deletion at only one end, front end, but allows insertion at both ends, rear and front ends, of the lists.
The possible operation performed on deque is

1. Add an element at the rear end
2. Add an element at the front end
3. Delete an element from the front end
4. Delete an element from the rear end

Only 1st, 3rd and 4th operations are performed by input-restricted deque and 1st, 2nd and 3rd operations are performed by output-restricted deque.

ALGORITHMS FOR INSERTING AN ELEMENT

Let Q be the array of MAX elements. *front* (or left) and *rear* (or *right*) are two array index (pointers), where the addition and deletion of elements occurred. Let DATA be the element to be inserted. Before inserting any element to the queue *left* and *right* pointer will point to the – 1.

INSERT AN ELEMENT AT THE RIGHT SIDE or REAR END or RIGHT MOST OF THE DE-QUEUE

```
1. Input the DATA to be inserted
2. If ((left == 0 && right == MAX–1) || (left == right + 1))
    (a) Display “Queue Overflow”
    (b) Exit
3. If (left == –1)
    (a) left = 0
    (b) right = 0
4. Else
    (a) if (right == MAX –1)
        (i) left = 0
    (b) else
        (i) right = right+1
5. Q[right] = DATA
6. Exit
```

INSERT AN ELEMENT AT THE LEFT SIDE or FRONT END or LEFT MOST OF THE DE-QUEUE

```
1. Input the DATA to be inserted
2. If ((left == 0 && right == MAX–1) || (left == right+1))
    (a) Display “Queue Overflow”
    (b) Exit
```

```
3. If (left == – 1)
    (a) Left = 0
    (b) Right = 0
4. Else
    (a) if (left == 0)
        (i) left = MAX – 1
    (b) else
        (i) left = left – 1
5. Q[left] = DATA
6. Exit
```

ALGORITHMS FOR DELETING AN ELEMENT

Let Q be the array of MAX elements. *front* (or *left*) and *rear* (or *right*) are two array index (pointers), where the addition and deletion of elements occurred. DATA will contain the element just deleted.

DELETE AN ELEMENT FROM THE RIGHT SIDE or REAR END or RIGHT MOST OF THE DE-QUEUE

```
1. If (left == – 1)
    (a) Display “Queue Underflow”
    (b) Exit
2. DATA = Q [right]
3. If (left == right)
    (a) left = – 1
    (b) right = – 1
```

```
4. Else
    (a) if(right == 0)
        (i) right = MAX-1
    (b) else
        (i) right = right-1
5. Exit
```

DELETE AN ELEMENT FROM THE LEFT SIDE or FRONT END or LEFT MOST OF THE DE-QUEUE

```
1. If (left == – 1)
    (a) Display “Queue Underflow”
    (b) Exit
2. DATA = Q [left]
3. If(left == right)
    (a) left = – 1
    (b) right = – 1
4. Else
    (a) if (left == MAX-1)
        (i) left = 0
    (b) Else
        (i) left = left +1
5. Exit
```

```
//PROGRAM TO IMPLEMENT INPUT AND OUTPUT ESTRICTED DE-QUEUE USING ARRAYS IN C
#include<conio.h>
#include<stdio.h>
#include<process.h>
#define MAX 50
int deque_arr[MAX];
int left = –1;
int right = –1;
//This function will insert an element at the right side of the de-queue
void insert_right()
{
int added_item;
if ((left == 0 && right == MAX-1) || (left == right+1))
{
printf (“\nQueue Overflow\n”);
getch();
return;
}
if (left == –1) /* if queue is initially empty */
{
left = 0;
right = 0;
}
else
if(right == MAX-1) /*right is at last position of queue */
right = 0;
else
right = right+1;
printf("\n Input the element for adding in queue: ");
scanf (“%d”, &added_item);
//Inputting the element at the right
deque_arr[right] = added_item ;
}/*End of insert_right()*/
//Function to insert an element at the left position of the de-queue
void insert_left()
{
```

```
int added_item;
//Checking for queue overflow
if ((left == 0 && right == MAX-1) || (left == right+1))
{
printf ("\nQueue Overflow \n");
getch();
return;
}
if (left == –1)/*If queue is initially empty*/
{
left = 0;
right = 0;
}
else
if (left== 0)
left = MAX –1;
else
left = left-1;
printf("\nInput the element for adding in queue:");
scanf ("%d", &added_item);
//inputting at the left side of the queue
deque_arr[left] = added_item ;
}/*End of insert_left()*/
//This function will delete an element from the queue from the left side
void delete_left()
{
//Checking for queue underflow
if (left == –1)
{
printf("\nQueue Underflow\n");
return;
}
//deleting the element from the left side
printf ("\nElement deleted from queue is: %d\n",deque_arr[left]);
if(left == right) /*Queue has only one element */
{
left = –1;
right=–1;
}
else
if (left == MAX-1)
left = 0;
else
left = left+1;
}/*End of delete_left()*/
//Function to delete an element from the right hand
//side of the de-queue
void delete_right()
{
//Checking for underflow conditions
if (left == –1)
{
printf(“\nQueue Underflow\n”);
return;
}
printf(“\nElement deleted from queue is : %d\n”,deque_arr[right]);
if(left == right) /*queue has only one element*/
{
left = –1;
right=–1;
}
else
```

```
if (right == 0)
right=MAX-1;
else
right=right-1;
}/*End of delete_right() */
//Displaying all the contents of the queue
void display_queue()
{
int front_pos = left, rear_pos = right;
//Checking whether the queue is empty or not
if (left == –1)
{
printf (“\nQueue is empty\n”);
return;
}
//displaying the queue elements
printf ("\nQueue elements :\n");
if ( front_pos <= rear_pos )
{
while(front_pos <= rear_pos)
{
printf (“%d ”,deque_arr[front_pos]);
front_pos++;
}
}
else
{
while(front_pos <= MAX-1)
{
printf(“%d ”,deque_arr[front_pos]);
front_pos++;
}
front_pos = 0;
while(front_pos <= rear_pos)
{
printf (“%d ”,deque_arr[front_pos]);
front_pos++;
}
}/*End of else */
printf (“\n”);
}/*End of display_queue() */
//Function to implement all the operation of the input restricted queue
void input_que()
{
int choice;
while(1)
{
clrscr();
//menu options to input restricted queue
printf ("\n1.Insert at right\n");
printf ("2.Delete from left\n");
printf ("3.Delete from right\n");
printf ("4.Display\n");
printf ("5.Quit\n");
printf ("\nEnter your choice : ");
scanf ("%d",&choice);
switch(choice)
{
case 1:
insert_right();
break;
case 2:
```

```
delete_left();
getch();
break;
case 3:
delete_right();
getch();
break;
case 4:
display_queue();
getch();
break;
case 5:
exit(0);
default:
printf("\nWrong choice\n");
getch();
}/*End of switch*/
}/*End of while*/
}/*End of input_que() */
//This function will implement all the operation of the output restricted queue
void output_que()
{
int choice;
while(1)
{
clrscr();
//menu options for output restricted queue
printf (“\n1.Insert at right\n”);
printf (“2.Insert at left\n”);
printf (“3.Delete from left\n”);
printf (“4.Display\n”);
printf (“5.Quit\n”);
printf (“\nEnter your choice:”);
scanf (“%d”,&choice);
switch(choice)
{
case 1:
insert_right();
break;
case 2:
insert_left();
break;
case 3:
delete_left();
getch();
break;
case 4:
display_queue();
getch();
break;
case 5:
exit(0);
default:
printf(“\nWrong choice\n”);
getch();
}/*End of switch*/
}/*End of while*/
}/*End of output_que() */
void main()
{
int choice;
clrscr();
```

```
//Main menu options
printf (“\n1.Input restricted dequeue\n”);
```

```
printf (“2.Output restricted dequeue\n”);
printf (“Enter your choice:”);
scanf (“%d”,&choice);
switch(choice)
{
case 1:
input_que();
break;
case 2:
output_que();
break;
default:
printf(“\nWrong choice\n”);
}/*End of switch*/
}/*End of main()*/
```

APPLICATIONS OF QUEUE

1. Round robin techniques for processor scheduling is implemented using queue.
2. Printer server routines (in drivers) are designed using queues.
3. All types of customer service software (like Railway/Air ticket reservation) are designed using queue to give proper service to the customers.

Sorting

The operation of sorting is the most common task performed by computers today. Sorting is used to arrange names and numbers in meaningful ways. For example; it is easy to look in the dictionary for a word if it is arranged (or sorted) in alphabetic order . Let A be a list of *n* elements A1, A2, ... An in memory. Sorting of list A refers to the operation of rearranging the contents of A so that they are in increasing (or decreasing) order (numerically or lexicographically); A1 < A2 < A3 < ...... < An. Since A has *n* elements, the contents in A can appear in *n*! ways. These ways correspond precisely to the *n*! permutations of 1,2,3, ... *n*. Each sorting algorithm must take care of these *n*! possibilities.

For example Suppose an array A contains 7 elements, 42, 33, 23, 74, 44, 67, 49. After sorting, the array A contains the elements as follows 23, 33, 42, 44, 49, 67, 74. Since A consists of 7 elements, there are 7! =.5040 ways that the elements can appear in A. The elements of an array can be sorted in any specified order *i.e*., either in ascending order or descending order. For example, consider an array A of size 7 elements 42, 33, 23, 74, 44, 67, 49. If they are arranged in ascending order, then sorted array is 23, 33, 42, 44, 49, 67, 74 and if the array is arranged in descending order then the sorted array is 74, 67, 49, 44, 42, 33, 23. In this chapter all the sorting techniques are discussed to arrange in ascending order.

It is very difficult to select a sorting algorithm over another. And there is no sorting algorithm better than all others in all circumstances. Some sorting algorithm will perform well in some situations, so it is important to have a selection of sorting algorithms. Some factors that play an important role in selection processes are the time complexity of the algorithm (use of computer time), the size of the data structures (for Eg: an array) to be sorted (use of storage space), and the time it takes for a programmer to implement the algorithms (programming effort).

For example, a small business that manages a list of employee names and salary could easily use an algorithm such as bubble sort since the algorithm is simple to implement and the data to be sorted is relatively small. However a large public limited with ten thousands of employees experience horrible delay, if we try to sort it with bubble sort algorithm. More efficient algorithm, like Heap sort is advisable.

COMPLEXITY OF SORTING ALGORITHMS

The complexity of sorting algorithm measures the running time of n items to be sorted. The operations in the sorting algorithm, where A1, A2 ….. An contains the items to be sorted and B is an auxiliary location, can be generalized as:

(*a*) Comparisons- which tests whether A*i* < A*j* or test whether A*i* < B

(*b*) Interchange- which switches the contents of A*i* and A*j* or of A*i* and B

(*c*) Assignments- which set B = A and then set A*j* = B or A*j* = A*i*

Normally, the complexity functions measure only the number of comparisons

.

BUBBLE SORT

In bubble sort, each element is compared with its adjacent element. If the first element is larger than the second one, then the positions of the elements are interchanged, otherwise it is not changed. Then next element is compared with its adjacent element and the same process is repeated for all the elements in the array until we get a sorted array.

Let A be a linear array of *n* numbers. Sorting of A means rearranging the elements of A so that they are in order. Here we are dealing with ascending order. *i.e.*, A[1] < A[2] < A[3] < ...... A[*n*].

Suppose the list of numbers A[1], A[2], ………… A[*n*] is an element of array A. The bubble sort algorithm works as follows:

Step 1: Compare A[1] and A[2] and arrange them in the (or desired) ascending order, so that A[1] < A[2].that is if A[1] is greater than A[2] then interchange the position of data by swap = A[1]; A[1] = A[2]; A[2] = swap. Then compare A[2] and A[3] and arrange them so that A[2] < A[3]. Continue the process until we compare A[N – 1] with A[N].

Note: Step1 contains *n* – 1 comparisons *i.e.*, the largest element is "bubbled up" to the *n*th position or "sinks" to the *n*th position. When step 1 is completed A[N] will contain the largest element.

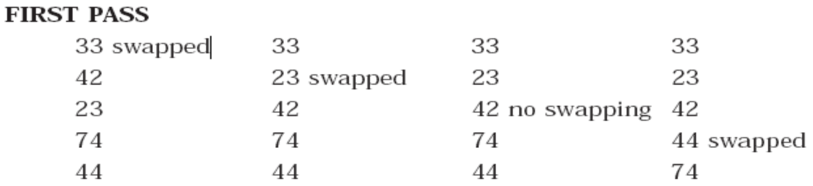
Step 2: Repeat step 1 with one less comparisons that is, now stop comparison at A [*n* – 1] and possibly rearrange A[N – 2] and A[N – 1] and so on.

Note: in the first pass, step 2 involves n–2 comparisons and the second largest element will occupy A[*n*-1]. And in the second pass, step 2 involves *n* – 3 comparisons and the 3rd largest element will occupy A[*n* – 2] and so on.

Step *n* – 1: compare A[1]with A[2] and arrange them so that A[1] < A[2]

After *n* – 1 steps, the array will be a sorted array in increasing (or ascending) order.

The following figures will depict the various steps (or PASS) involved in the sorting of an array of 5 elements. The elements of an array A to be sorted are: 42, 33, 23, 74, 44



FIRST PASS

33 swapped	33	33	33
42	23 swapped	23	23
23	42	42 no swapping	42
74	74	74	44 swapped
44	44	44	74

ALGORITHM

Let A be a linear array of *n* numbers. Swap is a temporary variable for swapping (or interchange) the position of the numbers.

```
1. Input n numbers of an array A
2. Initialise i = 0 and repeat through step 4 if (i < n)
3. Initialize j = 0 and repeat through step 4 if (j < n – i – 1)
4. If (A[j] > A[j + 1])
(a) Swap = A[j]
(b) A[j] = A[j + 1]
(c) A[j + 1] = Swap
5. Display the sorted numbers of array A
6. Exit.
```

```
//PROGRAM TO IMPLEMENT BUBBLE SORT USING ARRAYS IN C
#include<conio.h>
#include<stdio.h>
#define MAX 20
void main()
{
int arr[MAX],i,j,k,temp,n,xchanges;
clrscr();
printf (“\nEnter the number of elements : ”);
scanf (“%d”,&n);
for (i = 0; i < n; i++)
{
printf (“E\nnter element %d : ”,i+1);
scanf (“%d”,&arr[i]);
}
printf (“\nUnsorted list is :\n”);
for (i = 0; i < n; i++)
printf (“%d ”, arr[i]);
printf (“\n”);
/* Bubble sort*/
for (i = 0; i < n–1 ; i++)
{
xchanges=0;
for (j = 0; j <n–1–i; j++)
{
if (arr[j] > arr[j+1])
{
temp = arr[j];
arr[j] = arr[j+1];
arr[j+1] = temp;
xchanges++;
}/*End of if*/
}/*End of inner for loop*/
if (xchanges==0) /*If list is sorted*/
break;
printf(“\nAfter Pass %d elements are : ”,i+1);
for (k = 0; k < n; k++)
printf(“%d ”, arr[k]);
printf("\n");
}/*End of outer for loop*/
printf("\nSorted list is :\n");
for (i = 0; i < n; i++)
printf("%d ", arr[i]);
getch();
}/*End of main()*/
```

TIME COMPLEXITY

The time complexity for bubble sort is calculated in terms of the number of comparisons *f (n)* (or of number of loops); here two loops (outer loop and inner loop) iterates (or repeated) the comparisons. The number of times the outer loop iterates is determined by the number of elements in the list which is asked to sort (say it is *n*). The inner loop is iterated one less than the number of elements in the list (*i.e*., *n*-1 times) and is reiterated upon every iteration of the outer loop $f(n) = (n - 1) + (n - 2) + ... + 2 + 1 = n(n - 1) = O(n^2)$.

BEST CASE

In this case you are asked to sort a sorted array by bubble sort algorithm. The inner loop will iterate with the 'if' condition evaluating time that is the swap procedure is never called. In best case outer loop will terminate after one iteration, *i.e.*, it involves performing one pass, which requires n–1 comparisons $f(n) = O(n)$

WORST CASE

In this case the array will be an inverted list (*i.e.*, 5, 4, 3, 2, 1, 0). Here to move first element to the end of the array, *n*–1 times the swapping procedure is to be called. Every other element in the list will also move one location towards the start or end of the loop on every iteration. Thus n times the outer loop will iterate and n (n-1) times the inner loop will iterate to sort an inverted array $f(n) = (n(n - 1))/2 = O(n^2)$.

AVERAGE CASE

Average case is very difficult to analyse than the other cases. In this case the input data(s) are randomly placed in the list. The exact time complexity can be calculated only if we know the number of iterations, comparisons and swapping. In general, the complexity of average case is: $f(n) = (n(n–1))/2 = O(n^2)$.

SELECTION SORT

Selection sort algorithm finds the smallest element of the array and interchanges it with the element in the first position of the array. Then it finds the second smallest element from the remaining elements in the array and places it in the second position of the array and so on. Let A be a linear array of '*n*' numbers, A [1], A [2], A [3],...... A [*n*].

Step 1: Find the smallest element in the array of *n* numbers A[1], A[2], ... A[*n*]. Let

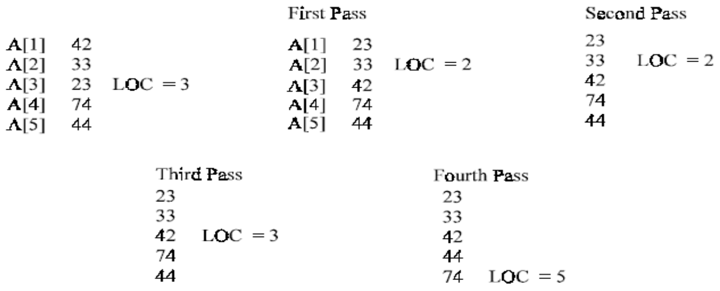
LOC is the location of the smallest number in the array. Then interchange A[LOC] and A[1] by swap = A[LOC]; A[LOC] = A[1]; A[1] = Swap.

Step 2: Find the second smallest number in the sub list of *n* – 1 elements A [2] A [3] ...... A [*n* – 1] (first element is already sorted). Now we concentrate on the rest of the elements in the array. Again A [LOC] is the smallest element in the remaining array and LOC the corresponding location then interchange A [LOC] and A [2].Now A [1] and A [2] is sorted, since A [1] less than or equal to A [2].

Step 3: Repeat the process by reducing one element each from the array

Step n – 1: Find the *n* – 1 smallest number in the sub array of 2 elements (*i.e*., A(*n*–1), A (*n*)). Consider A [LOC] is the smallest element and LOC is its corresponding position. Then interchange A [LOC] and A(*n* – 1). Now the array A [1], A [2], A [3], A [4],………..A [*n*] will be a sorted array.

Following figure is generated during the iterations of the algorithm to sort 5 numbers 42, 33, 23, 74, 44 :



A[1]	42	
A[2]	33	
A[3]	23	LOC = 3
A[4]	74	
A[5]	44	

First Pass

A[1]	23	
A[2]	33	LOC = 2
A[3]	42	
A[4]	74	
A[5]	44	

Second Pass

23	
33	LOC = 2
42	
74	
44	

Third Pass

23	
33	
42	LOC = 3
74	
44	

Fourth Pass

23	
33	
42	
44	
74	LOC = 5

ALGORITHM

Let A be a linear array of *n* numbers A [1], A [2], A [3], ……… A [k], A [k+1], …….. A [n]. *Swap* be a temporary variable for swapping (or interchanging) the position of the numbers. *Min* is the variable to store smallest number and *Loc* is the location of the smallest element.

```
1. Input n numbers of an array A
2. Initialize i = 0 and repeat through step5 if (i < n – 1)
    (a) min = a[i]
    (b) loc = i
```

```
3. Initialize j = i + 1 and repeat through step 4 if (j < n – 1)
4. if (a[j] < min)
    (a) min = a[j]
    (b) loc = j
5. if (loc ! = i)
    (a) swap = a[i]
    (b) a[i] = a[loc]
    (c) a[loc] = swap
6. display “the sorted numbers of array A”
7. Exit
```

```
//PROGRAM TO IMPLEMENT SELECTION SORT USING ARRAYS IN C
#include<conio.h>
#include<stdio.h>
#define MAX 20
void main()
{
int arr[MAX], i,j,k,n,temp,smallest;
clrscr();
printf (“\nEnter the number of elements : ”);
scanf (“%d”, & n);
for (i = 0; i < n; i++)
{
printf (“\nEnter element %d : ”,i+1);
scanf (“%d”, &arr[i]);
}
printf (“\nUnsorted list is : \n”);
for (i = 0; i < n; i++)
printf (“%d ”, arr[i]);
printf (“\n”);
/*Selection sort*/
for (i = 0; i < n – 1 ; i++)
{
/*Find the smallest element*/
smallest = i;
for(k = i + 1; k < n ; k++)
{
if (arr[smallest] > arr[k])
smallest = k ;
}
if ( i != smallest )
{
temp = arr [i];
arr[i] = arr[smallest];
arr[smallest] = temp ;
}
printf (“\nAfter Pass %d elements are : ”,i+1);
for (j = 0; j < n; j++)
printf (“%d ”, arr[ j]);
printf (“\n”);
}/*End of for*/
printf (“\nSorted list is : \n”);
for (i = 0; i < n; i++)
printf (“%d ”, arr[i]);
getch();
}/*End of main()*/
```

TIME COMPLEXITY

Time complexity of a selection sort is calculated in terms of the number of comparisons $f(n)$. In the first pass it makes $n - 1$ comparisons; the second pass makes $n - 2$ comparisons and so on. The outer *for loop* iterates for (n - 1) times. But the inner loop iterates for $n*(n - 1)$ times to complete the sorting.

$f(n) = (n - 1) + (n - 2) + ... + 2 + 1$

$= (n(n - 1))/2$

$= O(n^2)$.

Best case	*Worst case*	*Average case*
$n - 1 = O(n)$	$\frac{n(n-1)}{2} = O(n^2)$	$\frac{n(n-1)}{2} = O(n)$

INSERTION SORT

Insertion sort algorithm sorts a set of values by inserting values into an existing sorted file. Compare the second element with first, if the first element is greater than second, place it before the first one. Otherwise place is just after the first one. Compare the third value with second. If the third value is greater than the second value then place it just after the second. Otherwise place the second value to the third place. And compare third value with the first value. If the third value is greater than the first value place the third value to second place, otherwise place the first value to second place. And place the third value to first place and so on.

Let A be a linear array of n numbers A [1], A [2], A [3], ... A[n]. The algorithm scan the array A from A [1] to A [n], by inserting each element A[k], into the proper position of the previously sorted sub list. A [1], A [2], A [3], ... A [$k - 1$]

Step 1: As the single element A [1] by itself is sorted array.

Step 2: A [2] is inserted either before or after A [1] by comparing it so that A[1], A[2] is sorted array.

Step 3: A [3] is inserted into the proper place in A [1], A [2], that is A [3] will be compared with A [1] and A [2] and placed before A [1], between A [1] and A[2], or after A [2] so that A [1], A [2], A [3] is a sorted array.

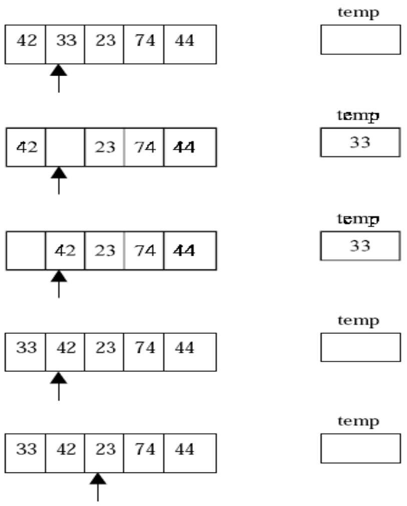
Step 4: A [4] is inserted in to a proper place in A [1], A [2], A [3] by comparing it; so that A [1], A [2], A [3], A [4] is a sorted array.

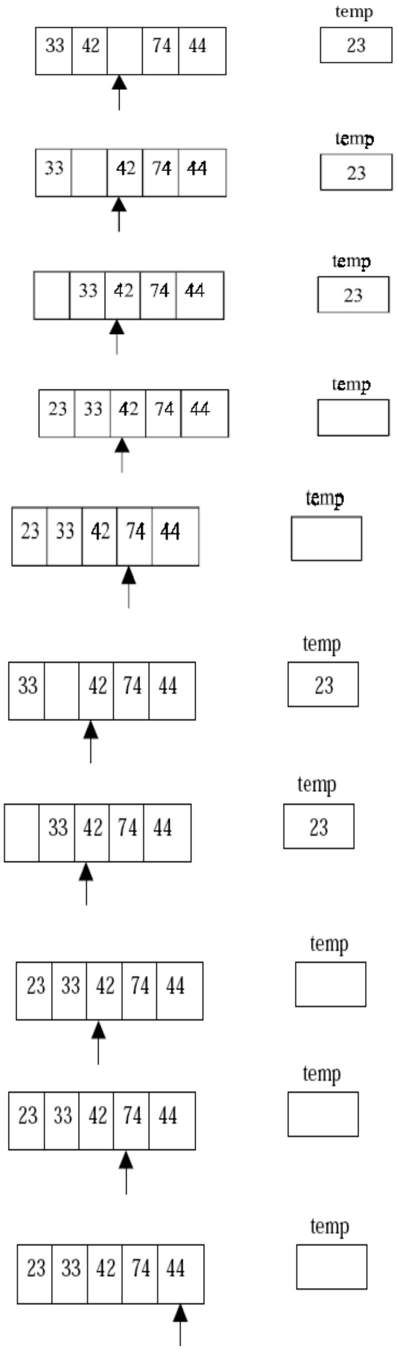
Step 5: Repeat the process by inserting the element in the proper place in array

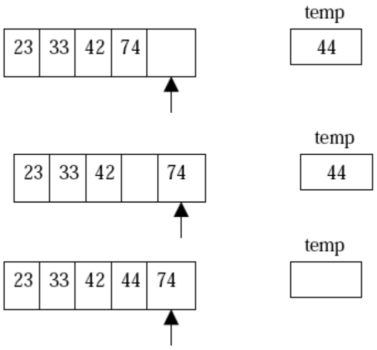
Step n : A [n] is inserted into its proper place in an array A [1], A [2], A [3], ... A [n –1] so that A [1], A [2], A [3], ... ,A [n] is a sorted array.

To illustrate the insertion sort methods, consider a following array with five elements

42, 33, 23, 74, 44 :




33 42 74 44
temp 23
33 42 74 44
temp 23
33 42 74 44
temp 23
23 33 42 74 44
temp
23 33 42 74 44
temp
33 42 74 44
temp 23
33 42 74 44
temp 23
23 33 42 74 44
temp
23 33 42 74 44
temp
23 33 42 74 44
temp



ALGORITHM

Let A be a linear array of n numbers A [1], A [2], A [3], ... ,A [n]......Swap be a temporary variable to interchange the two values. Pos is the control variable to hold the position of each pass.

```
1. Input an array A of n numbers
2. Initialize i = 1 and repeat through steps 4 by incrementing i by one.
(a) If (i < = n – 1)
(b) Swap = A [I],
(c) Pos = i – 1
3. Repeat the step 3 if (Swap < A[Pos] and (Pos >= 0))
(a) A [Pos+1] = A [Pos]
(b) Pos = Pos-1
4. A [Pos +1] = Swap
5. Exit
```

```
//PROGRAM TO IMPLEMENT INSERTION SORT USING ARRAYS IN C
#include<conio.h>
#include<stdio.h>
#define MAX 20
void main()
{
int arr[MAX],i,j,k,n;
clrscr();
printf (“\nEnter the number of elements : ”);
scanf (“%d”,&n);
for (i = 0; i < n; i++)
{
printf (“\nEnter element %d : ”,i+1);
scanf (“%d”, &arr[i]);
}
printf (“\nUnsorted list is :\n”);
for (i = 0; i < n; i++)
printf (“%d ”, arr[i]);
printf (“\n”);

/*Insertion sort*/
for(j=1;j < n;j++)
{
k=arr[j]; /*k is to be inserted at proper place*/
for(i=j–1;i>=0 && k<arr[i];i--)
arr[i+1]=arr[i];
arr[i+1]=k;
printf (“\nPass %d, Element inserted in proper place: %d\n”,j,k);
for (i = 0; i < n; i++)
printf (“%d ”, arr[i]);
printf (“\n”);
}
printf (“\nSorted list is :\n”);
for (i = 0; i < n; i++)
printf (“%d ”, arr[i]);
getch();
}/*End of main()*/
```

TIME COMPLEXITY

In the insertion sort algorithm (n – 1) times the loop will execute for comparisons and interchanging the numbers. The inner while loop iterates maximum of ((n – 1) · (n – 1))/2 times to computing the sorting.

WORST CASE

The worst case occurs when the array A is in reverse order and the inner while loop must use the maximum number (n – 1) of comparisons. Hence $f(n) = (n – 1) + ... 2 + 1 = (n (n – 1))/2 = O(n^2)$.

AVERAGE CASE

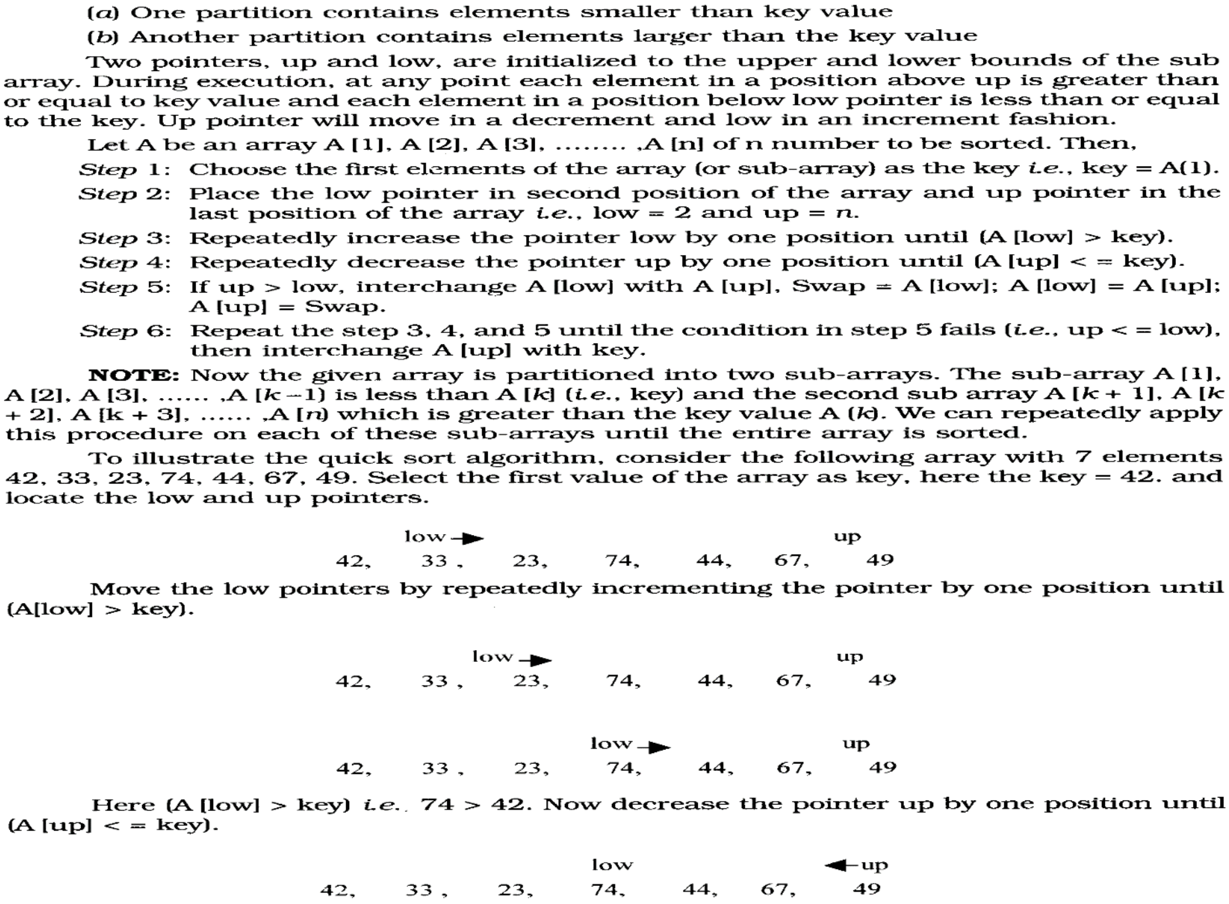
On the average case there will be approximately (n – 1)/2 comparisons in the inner while loop. Hence the average case $f(n) = (n – 1)/2 + ... + 2/2 +1/2 = n (n – 1)/4 = O(n^2)$.

BEST CASE

The best case occurs when the array A is in sorted order and the outer for loop will iterate for (n – 1) times. And the inner while loop will not execute because the given array is a sorted array *i.e.*, $f(n) = O(n)$

QUICK SORT

It is one of the widely used sorting techniques and it is also called the partitionexchange sort. Quick sort is an efficient algorithm and it passes a very good time complexity in average case. This is developed by C.A.R. Hoare. It is an algorithm of the divide-and conquer type. The quick sort algorithm works by partitioning the array to be sorted. And each partitions are internally sorted recursively. In partition the first element of an array is chosen as a key value. This key value can be the first element of an array. That is, if A is an array then key = A (0), and rest of the elements are grouped into two portions such that,



(*a*) One partition contains elements smaller than key value

(*b*) Another partition contains elements larger than the key value

Two pointers, up and low, are initialized to the upper and lower bounds of the sub array. During execution, at any point each element in a position above up is greater than or equal to key value and each element in a position below low pointer is less than or equal to the key. Up pointer will move in a decrement and low in an increment fashion.

Let A be an array A [1], A [2], A [3], ... ,A [n] of n number to be sorted. Then,

Step 1: Choose the first elements of the array (or sub-array) as the key *i.e.*, key = A(1).

Step 2: Place the low pointer in second position of the array and up pointer in the last position of the array *i.e.*, low = 2 and up = *n*.

Step 3: Repeatedly increase the pointer low by one position until (A [low] > key).

Step 4: Repeatedly decrease the pointer up by one position until (A [up] < = key).

Step 5: If up > low, interchange A [low] with A [up], Swap = A [low]; A [low] = A [up]; A [up] = Swap.

Step 6: Repeat the step 3, 4, and 5 until the condition in step 5 fails (*i.e.*, up < = low), then interchange A [up] with key.

NOTE: Now the given array is partitioned into two sub-arrays. The sub-array A [1], A [2], A [3], ... ,A [*k* –1] is less than A [*k*] (*i.e.*, key) and the second sub array A [*k* + 1], A [*k* + 2], A [k + 3], ... ,A [*n*) which is greater than the key value A (*k*). We can repeatedly apply this procedure on each of these sub-arrays until the entire array is sorted.

To illustrate the quick sort algorithm, consider the following array with 7 elements 42, 33, 23, 74, 44, 67, 49. Select the first value of the array as key, here the key = 42. and locate the low and up pointers.

```
          low →                               up
    42,    33 ,    23,     74,     44,    67,     49
```

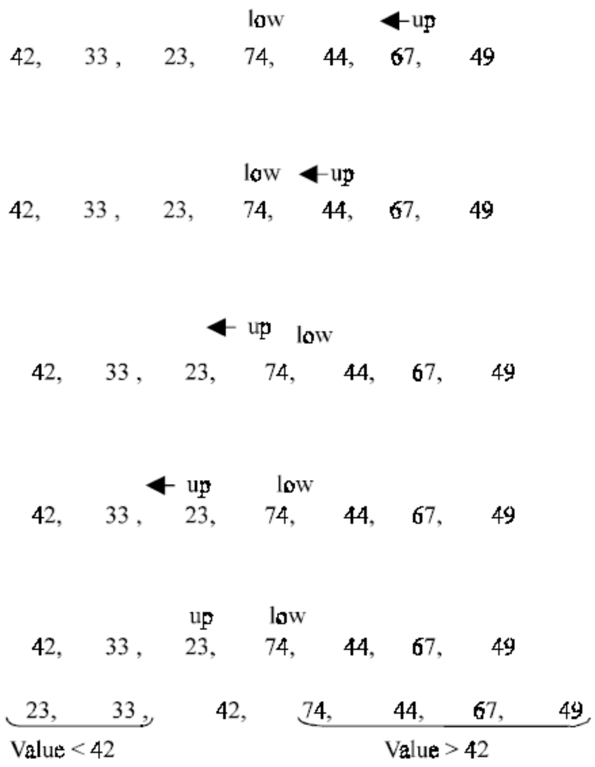
Move the low pointers by repeatedly incrementing the pointer by one position until (A[low] > key).

```
                  low →                       up
    42,    33 ,    23,     74,     44,    67,     49
```

```
                          low →               up
    42,    33 ,    23,     74,     44,    67,     49
```

Here (A [low] > key) *i.e.*, 74 > 42. Now decrease the pointer up by one position until (A [up] < = key).

```
                           low              ← up
    42,    33 ,    23,     74,     44,    67,     49
```



ALGORITHM

Let A be a linear array of *n* elements A (1), A (2), A (3)......A (*n*), low represents the lower bound pointer and up represents the upper bound pointer. Key represents the first element of the array, which is going to become the middle element of the sub-arrays.

```
1. Input n number of elements in an array A
2. Initialize low = 2, up = n , key = A[(low + up)/2]
3. Repeat through step 8 while (low < = up)
4. Repeat step 5 while(A [low] > key)
5. low = low + 1
6. Repeat step 7 while(A [up] < key)
7. up = up–1
8. If (low < = up)
(a) Swap = A [low]
(b) A [low] = A [up]
(c) A [up] = swap
(d) low=low+1
```

```
(e) up=up–1
9. If (1 < up) Quick sort (A, 1, up)
10. If (low < n) Quick sort (A, low, n)
11. Exit
```

```
//PROGRAM TO IMPLEMENT QUICK SORT USING ARRAYS RECURSIVELY IN C
#include<conio.h>
#include<stdio.h>
#define MAX 30
enum bool {FALSE,TRUE};
//Function display the array
void display(int arr[],int low,int up)
{
int i;
for(i=low;i<=up;i++)
```

```
printf (“%d ”,arr[i]);
}
//This function will sort the array using Quick sort algorithm
void quick(int arr[],int low,int up)
{
int piv,temp,left,right;
enum bool pivot_placed=FALSE;
//setting the pointers
left=low;
right=up;
piv=low; /*Take the first element of sublist as piv */
if (low>=up)
return;
printf (“\nSublist : ”);
display(arr,low,up);
/*Loop till pivot is placed at proper place in the sublist*/
while(pivot_placed==FALSE)
{
/*Compare from right to left */
while( arr[piv]<=arr[right] && piv!=right )
right=right–1;
if ( piv==right )
pivot_placed=TRUE;
if ( arr[piv] > arr[right] )
{
temp=arr[piv];
arr[piv]=arr[right];
arr[right]=temp;
piv=right;
}
/*Compare from left to right */
while( arr[piv]>=arr[left] && left!=piv )
left=left+1;
if (piv==left)
pivot_placed=TRUE;
if ( arr[piv] < arr[left] )
{
temp=arr[piv];
arr[piv]=arr[left];
arr[left]=temp;
piv=left;
}
}/*End of while */
printf (“ Pivot Placed is %d  ”,arr[piv]);
display(arr,low,up);
printf ("\n");
quick(arr,low,piv–1);
quick(arr,piv+1,up);
}/*End of quick()*/
void main()
{
int array[MAX],n,i;
clrscr();
printf (“\nEnter the number of elements : ”);
scanf (“%d”,&n);
for (i=0;i<n;i++)
{
printf (“\nEnter element %d : ”,i+1);
scanf (“%d”,&array[i]);
}
printf (“\nUnsorted list is :\n”);
display(array,0,n–1);
```

```
printf (“\n”);
quick (array,0,n–1);
printf (“\nSorted list is :\n”);
display(array,0,n–1);
getch();
}/*End of main() */
```

TIME COMPLEXITY

The time complexity of quick sort can be calculated for any of the following case. It is usually measured by the number $f(n)$ of comparisons required to sort n elements.

WORST CASE

The worst case occurs when the list is already sorted. In this case the given array is partitioned into two sub arrays. One of them is an empty array and another one is an array. After partition, when the first element is checked with other element, it will take n comparison to recognize that it remain in the position so as $(n - 1)$ comparisons for the second position.

$$\begin{aligned} f(n) &= n + (n - 1) + \ldots\ldots + 2 + 1 \\ &= (n (n + 1))/2 \\ &= O(n^2). \end{aligned}$$

AVERAGE CASE

In this case each reduction step of the algorithm produces two sub arrays. Accordingly
:

(*a*) Reducing the array by placing one element and produces two sub arrays.

(*b*) Reducing the two sub-arrays by placing two elements and produces four subarrays.

(*c*) Reducing the four sub-arrays by placing four elements and produces eight subarrays. And so on. Reduction step in the kth level finds the location at 2^{k-1} elements; however there will be approximately $\log_2 n$ levels at reduction steps. Furthermore each level uses at most n comparisons, so $f(n) = O(n \log n)$

BEST CASE

The base case analysis occurs when the array is always partitioned in half, That key
= A [(low+up)/2]
$f(n) = Cn + f(n/2) + f(n/2)$
$= Cn + 2f(n/2)$
$= O(n)$ where C is a constant.

MERGE SORT

Merge sort is based on the **divide-and-conquer** paradigm. Its worst-case running time has a lower order of growth than insertion sort. Since we are dealing with subproblems, we state each subproblem as sorting a subarray $A[p .. r]$. Initially, $p = 1$ and $r = n$, but these values change as we recurse through subproblems.

To sort $A[p .. r]$:

1. **Divide Step**

   If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split $A[p .. r]$ into two subarrays $A[p .. q]$ and $A[q + 1 .. r]$, each containing about half of the elements of $A[p .. r]$. That is, q is the halfway point of $A[p .. r]$.

2. **Conquer Step**

   Conquer by recursively sorting the two subarrays $A[p .. q]$ and $A[q + 1 .. r]$.

3. **Combine Step**

   Combine the elements back in $A[p .. r]$ by merging the two sorted subarrays $A[p .. q]$ and $A[q + 1 .. r]$ into a sorted sequence. To accomplish this step, we will define a procedure MERGE (A, p, q, r).

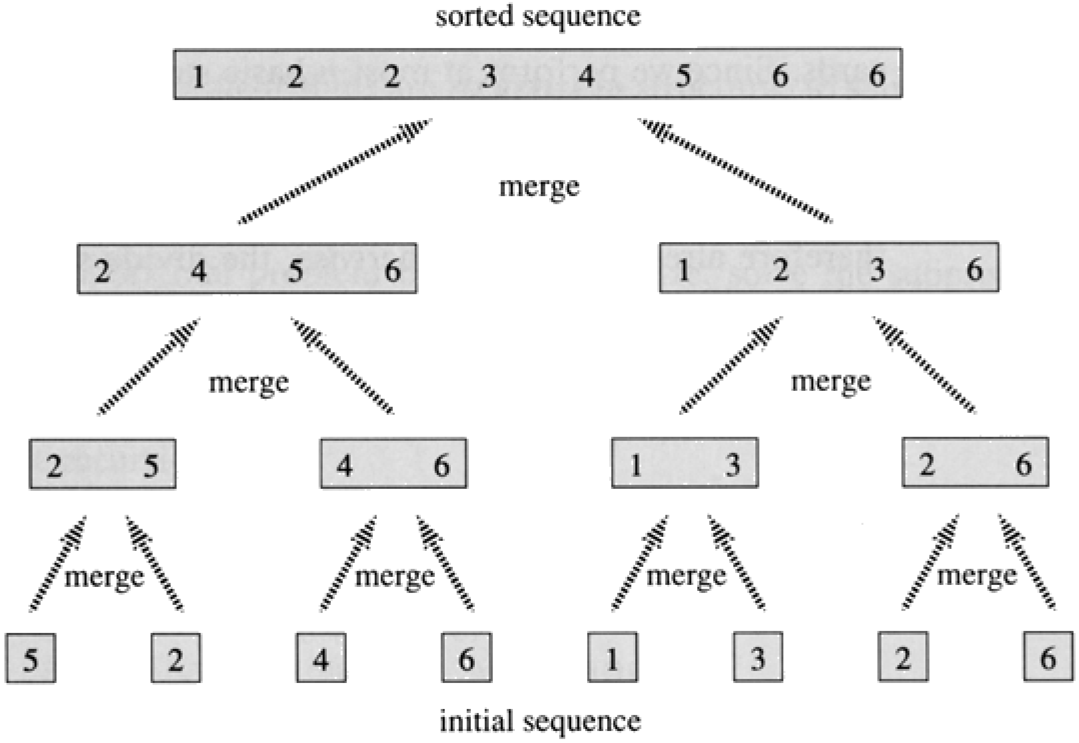
Note that the recursion bottoms out when the subarray has just one element, so that it is trivially sorted.

Algorithm: Merge Sort

To sort the entire sequence A[1 .. n], make the initial call to the procedure MERGE-SORT (A, 1, n).

```
MERGE-SORT (A, p, r)
IF p < r                              // Check for base case
    THEN q = FLOOR[(p + r)/2]            // Divide step
        MERGE (A, p, q)                 // Conquer step.
        MERGE (A, $q$ + 1, $r$)          // Conquer step.
        MERGE (A, $p$, $q$, $r$)          // Conquer step.
```

Example: Bottom-up view of the above procedure for *n* = 8.



Merging

What remains is the MERGE procedure. The following is the input and output of the MERGE procedure.

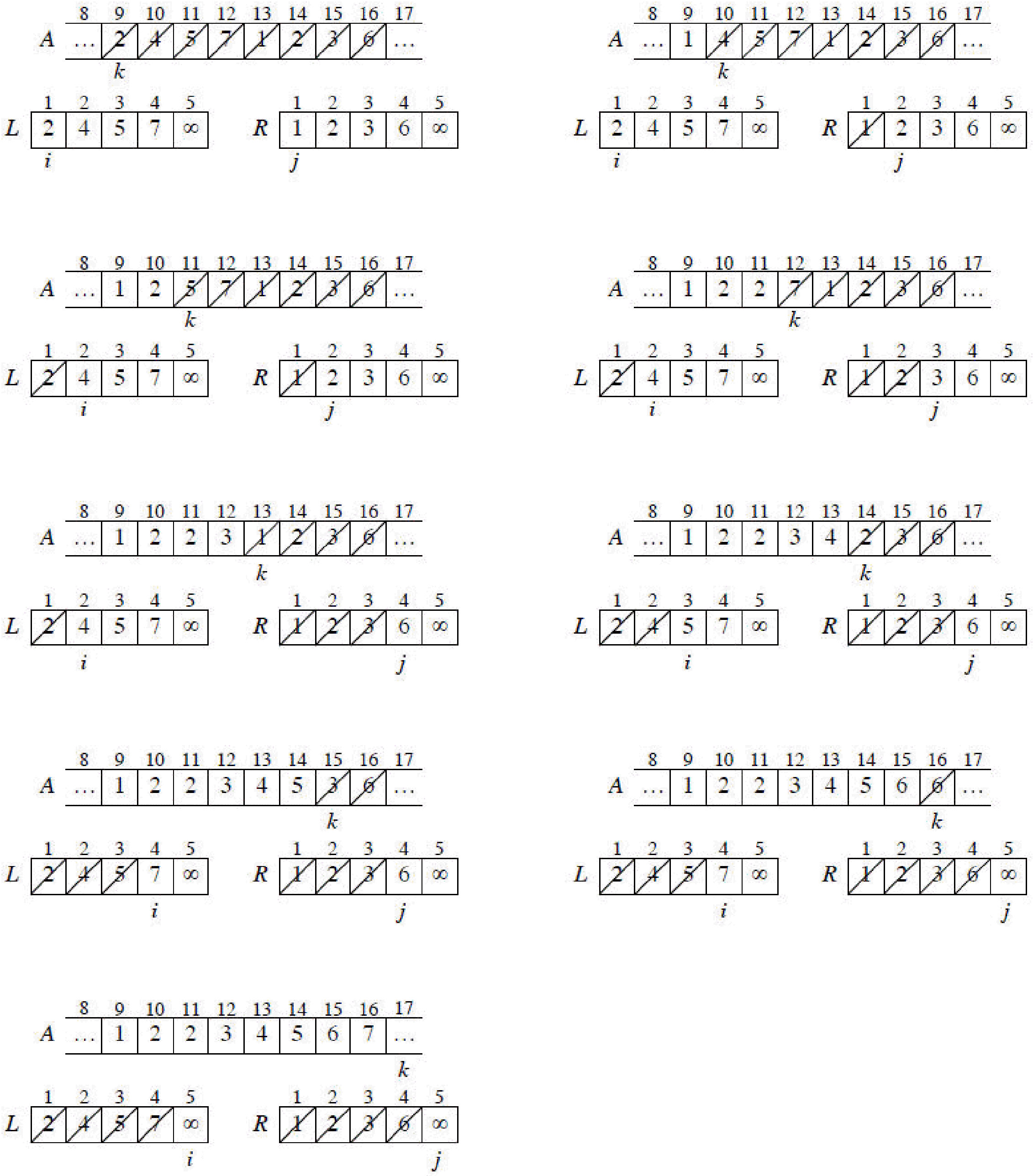
INPUT: Array *A* and indices *p*, *q*, *r* such that $p \le q \le$ r and subarray *A*[*p* .. *q*] is sorted and subarray *A*[*q* + 1 .. *r*] is sorted. By restrictions on *p*, *q*, *r*, neither subarray is empty.

OUTPUT: The two subarrays are merged into a single sorted subarray in *A*[*p* .. *r*].

MERGE procedure is as follow:

```
MERGE ($A$, $p$, $q$, $r$ )
$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
Create arrays L[1 . . $n_1$ + 1] and R[1 . . $n_2$ + 1]
FOR $i \leftarrow 1$ TO $n_1$
    DO L[$i$] ← A[$p + i - 1$]
FOR $j \leftarrow 1$ TO $n_2$
    DO R[$j$] ← A[$q + j$ ]
L[$n_1$ + 1] ← ∞
R[$n_2$ + 1] ← ∞
$i \leftarrow 1$
$j \leftarrow 1$
FOR $k \leftarrow p$ TO $r$
    DO IF L[$i$ ] ≤ R[ $j$]
          THEN A[$k$] ← L[$i$]
                 $i \leftarrow i + 1$
          ELSE A[k] ← R[j]
                 $j \leftarrow j + 1$
```

- The first part shows the arrays at the start of the "for $k \leftarrow p$ to r" loop, where A[p . . q] is copied into L[1 . . n_1] and A[q + 1 . . r] is
copied into R[1 . . n_2].
- Succeeding parts show the situation at the start of successive iterations.
- Entries in A with slashes have had their values copied to either L or R and have not had a value copied back in yet. Entries in L and R with slashes have been copied back into A.
- The last part shows that the subarrays are merged back into A[p . . r], which is now sorted, and that only the sentinels (∞) are exposed in the arrays L and R.]



Implementation

```
void mergeSort(int numbers[], int temp[], int array_size)
        {
             m_sort(numbers, temp, 0, array_size - 1);
```

```
}

void m_sort(int numbers[], int temp[], int left, int right)
{
        int mid;
        if (right > left)
        {
            mid = (right + left) / 2;
            m_sort(numbers, temp, left, mid);
            m_sort(numbers, temp, mid+1, right);

            merge(numbers, temp, left, mid+1, right);
        }
}
void merge(int numbers[], int temp[], int left, int mid, int right)
        {
            int i, left_end, num_elements, tmp_pos;

            left_end = mid - 1;
            tmp_pos = left;
            num_elements = right - left + 1;
while ((left <= left_end) && (mid <= right))
        {
                if (numbers[left] <= numbers[mid])
                {
                        temp[tmp_pos] = numbers[left];
                        tmp_pos = tmp_pos + 1;
                        left = left +1;
                }
```

```
    else
    {
        temp[tmp_pos] = numbers[mid];
        tmp_pos = tmp_pos + 1;
        mid = mid + 1;
    }
}
while (left <= left_end)
    {
        temp[tmp_pos] = numbers[left];
        left = left + 1;
        tmp_pos = tmp_pos + 1;
    }
    while (mid <= right)
    {
        temp[tmp_pos] = numbers[mid];
        mid = mid + 1;
        tmp_pos = tmp_pos + 1;
    }

    for (i = 0; i <= num_elements; i++)
    {
        numbers[right] = temp[right];
        right = right - 1;
    }
}
```

TIME COMPLEXITY

Let $f(n)$ denote the number of comparisons needed to sort an n element array A using the merge sort algorithm. The algorithm requires almost log n passes, each involving n or fewer comparisons. In average and worst case the merge sort requires $O(n \log n)$ comparisons. The main drawback of merge sort is that it requires $O(n)$ additional space for the auxiliary array

```
//PROGRAM TO IMPLEMENT MERGE SORT WITHOUT RECURSION IN C
#include<stdio.h>
#include<conio.h>
#define MAX 30
void main()
{
int arr[MAX],temp[MAX],i,j,k,n,size,l1,h1,l2,h2;
clrscr();
printf (“\nEnter the number of elements : ”);
scanf (“%d”,&n);
for (i=0;i<n;i++)
{
printf (“\nEnter element %d : ”,i+1);
scanf (“%d”,&arr[i]);
}
printf (“\nUnsorted list is : ”);
for ( i = 0 ; i<n ; i++)
printf(“%d ”, arr[i]);
/*l1 lower bound of first pair and so on*/
for (size=1; size < n; size=size*2 )
{
l1 = 0;
k = 0; /*Index for temp array*/
while( l1+size < n)
{
h1=l1+size–1;
l2=h1+1;
h2=l2+size–1;
if ( h2>=n ) /* h2 exceeds the limlt of arr */
h2=n-1;
/*Merge the two pairs with lower limits l1 and l2*/
i=l1;
j=l2;
while(i<=h1 && j<=h2 )
{
if ( arr[i] <= arr[ j] )
temp[k++]=arr[i++];
else
temp[k++]=arr[ j++];
}
while(i<=h1)
temp[k++]=arr[i++];
while( j<=h2)
temp[k++]=arr[ j++];
/**Merging completed**/
l1 = h2+1; /*Take the next two pairs for merging */
}/*End of while*/
for (i=l1; k<n; i++) /*any pair left */
temp[k++]=arr[i];
for(i=0;i<n;i++)
arr[i]=temp[i];
printf (“\nSize=%d \nElements are : ”,size);
for ( i = 0 ; i<n ; i++)
printf (“%d ”, arr[i]);
}/*End of for loop */
printf (“\nSorted list is :\n”);
for ( i = 0 ; i<n ; i++)
printf (“%d ”, arr[i]);
```

```
getch();
}/*End of main()*/
```

```
//PROGRAM TO IMPLEMENT MERGE SORT THROUGH RECURSION IN  C
#include<stdio.h>
#include<conio.h>
#define MAX 20
int array[MAX];
//Function to merge the sub files or arrays
void merge(int low, int mid, int high )
{
int temp[MAX];
int i = low;
int j = mid +1 ;
int k = low ;
while( (i < = mid) && (j < =high) )
{
if (array[i] < = array[ j])
temp[k++] = array[i++] ;
else
temp[k++] = array[ j++] ;
}/*End of while*/
while( i <= mid )
temp[k++]=array[i++];
while( j <= high )
temp[k++]=array[j++];
for (i= low; i < = high ; i++)
array[i]=temp[i];
}/*End of merge()*/
//Function which call itself to sort an array
void merge_sort(int low, int high )
{
int mid;
if ( low ! = high )
{
mid = (low+high)/2;
merge_sort( low , mid );
merge_sort( mid+1, high );
merge(low, mid, high );
}
}/*End of merge_sort*/
void main()
{
int i,n;
clrscr();
printf (“\nEnter the number of elements : ”);
scanf (“%d”,&n);
for (i=0;i<n;i++)
{
printf (“\nEnter element %d : ”,i+1);
scanf (“%d”,&array[i]);
}
printf (“\nUnsorted list is :\n”);
for ( i = 0 ; i<n ; i++)
printf (“%d ”, array[i]);
merge_sort( 0, n–1);
printf (“\nSorted list is :\n”);
for ( i = 0 ; i<n ; i++)
printf (“%d ”, array[i]);
getch();
}/*End of main()*/
```

HEAP

A heap is defined as an almost complete binary tree of n nodes such that the value of each node is less than or equal to the value of the father. It can be sequentially represented as

A[*j*] <= A[(*j* – 1)/2]

for 0 <= [(*j* – 1)/2] < *j* <= *n* – 1

The root of the binary tree (*i.e.*, the first array element) holds the largest key in the heap. This type of heap is usually called descending heap or mere heap, as the path from the root node to a terminal node forms an ordered list of elements arranged in descending order. Fig. 6.1 shows a heap.

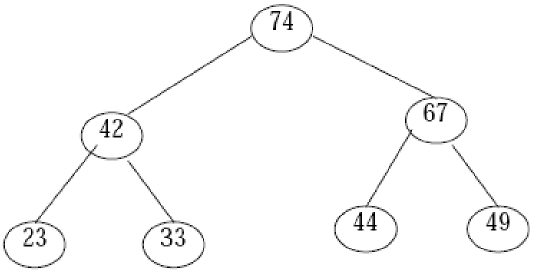
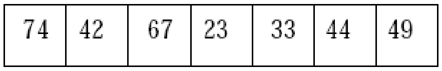


Fig. 6.1. Heap representation



74	42	67	23	33	44	49

Fig. 6.2. Sequential representation

We can also define an ascending heap as an almost complete binary tree in which the value of each node is greater than or equal to the value of its father. This root node has the smallest element of the heap. This type of heap is also called min heap.

HEAP AS A PRIORITY QUEUE

A heap is very useful in implementing priority queue. The priority queue is a data structure in which the intrinsic ordering of the data items determines the result of its basic operations. Primarily queues can be classified into two types:

1. Ascending priority queue
2. Descending priority queue

An ascending priority queue can be defined as a group of elements to which new elements are inserted arbitrarily but only the smallest element is deleted from it. On the other hand a descending priority queue can be defined as a group of elements to which new elements are inserted arbitrarily but only the largest element is deleted from it.

HEAP SORT

A heap can be used to sort a set of elements. Let H be a heap with *n* elements and it can be sequentially represented by an array A. Inset an element *data* into the heap H as follows:

1. First place *data* at the end of H so that H is still a complete tree, but not necessarily a heap.
2. Then the data be raised to its appropriate place in H so that H is finally a heap.

To understand the concept of insertion of data into a heap is illustrated with following two examples:

INSERTING AN ELEMENT TO A HEAP

Consider the heap H in Fig. 6.1. Say we want to add a data = 55 to H.

Step 1: First we adjoin 55 as the next element in the complete tree as shown in Fig.6.2. Now we have to find the appropriate place for 55 in the heap by rearranging it.

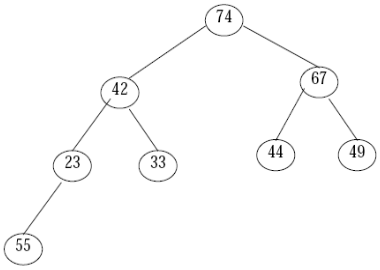


Fig. 6.2.

Step 2: Compare 55 with its parent 23. Since 55 is greater than 23, interchange 23 and 55. Now the heap will look like as in Fig. 6.3.

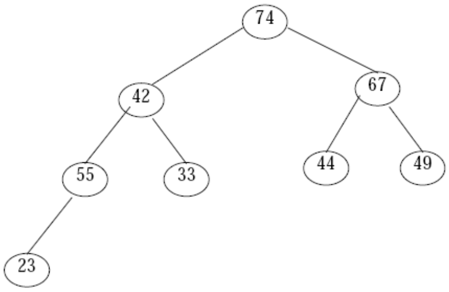


Fig. 6.3

Step 3: Compare 55 with its parent 42. Since 55 is greater than 42, interchange 55 and 42. Now the heap will look like as in Fig. 6.4.

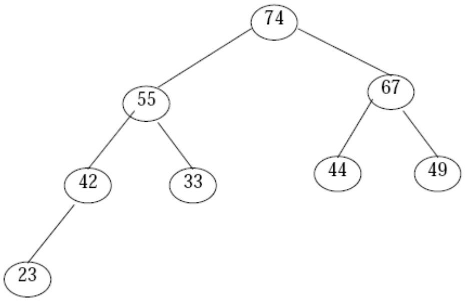


Fig. 6.4

Step 4: Compare 55 with its new parent 74. Since 55 is less than 74, it is the appropriate place of node 55 in the heap H. Fig. 6.4 shows the final heap tree.

CREATING A HEAP

A heap H can be created from the following list of numbers 33, 42, 67, 23, 44, 49, 74 as illustrated below :

Step 1: Create a node to insert the first number (*i.e.*, 33) as shown Fig 6:5

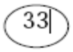


Fig. 6.5

Step 2: Read the second element and add as the left child of 33 as shown Fig. 6.6. Then restructure the heap if necessary.

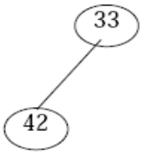


Fig. 6.6

Compare the 42 with its parent 33, since newly added node (*i.e.*, 42) is greater than 33 interchange node information as shown Fig. 6.7.

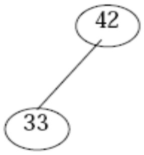


Fig. 6.7

Step 3: Read the 3rd element and add as the right child of 42 as shown Fig. 6.8. Then restructure the heap if necessary.

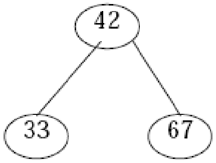


Fig. 6.7

Compare the 67 with its parent 42, since newly added node (*i.e.*, 67) is greater than 42 interchange node information as shown Fig. 6.8.

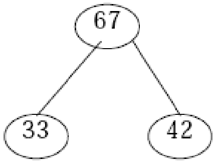


Fig. 6.8

Step 4: Read the 4th element and add as the left child of 33 as shown Fig. 6.9. Then restructure the heap if necessary.

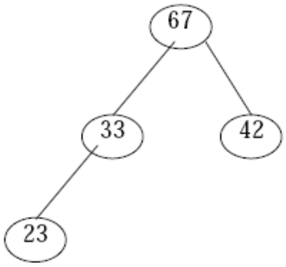


Fig. 6.9

Since newly added node (*i.e.*, 23) is less than its parent 33, no interchange.

Step 5: Read the 5th element and add as the right child of 33 as shown Fig. 6.10. Then restructure the heap if necessary.

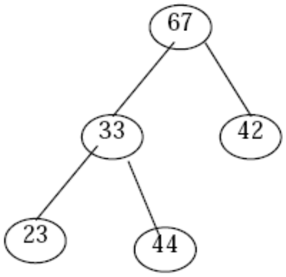


Fig. 6.10

Compare the 44 with its parent 33, since newly added node (*i.e.*, 44) is greater than 33 interchange node information as shown Fig. 6.11.

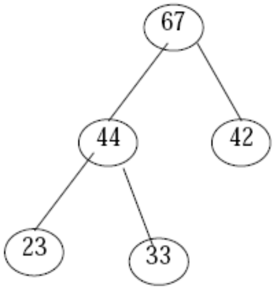


Fig. 6.11

Step 6: Read the 6th element and add as the left child of 42 as shown Fig. 6.12. Then restructure the heap if necessary.

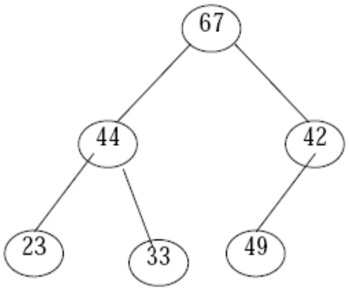


Fig. 6.12

Compare the (newly added node) 49 with its parent 42, since newly added node (*i.e.*, 49) is greater than 42 interchange node information as shown Fig. 6.13.

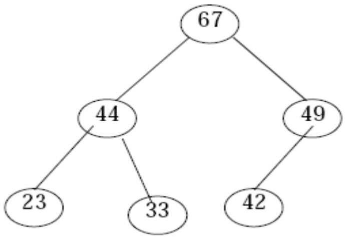


Fig. 6.13

Step 7: Read the 7th element and add as the left child of 49 as shown Fig. 6.14. Then restructure the heap if necessary.

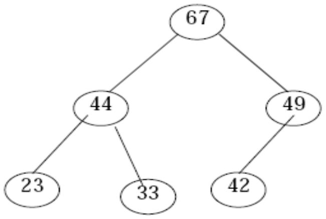


Fig. 6.14

Compare the (newly added node) 74 with its parent 49, since newly added node (*i.e.*, 74) is greater than 49 interchange node information as shown Fig. 6.15.

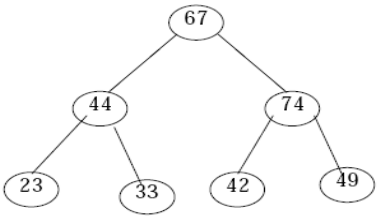


Fig. 6.15

Compare the recently changed node 74 with its parent 67, since it is greater than 67 interchange node information as shown Fig. 6.16.

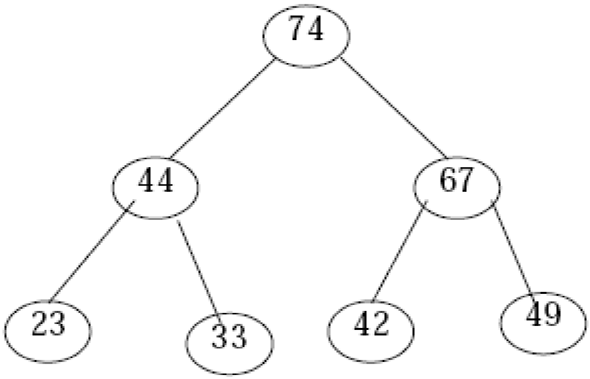


Fig. 6.16

ALGORITHM

Let H be a heap with *n* elements stored in the array HA. This procedure will insert a new element *data* in H. LOC is the present location of the newly added node. And PAR denotes the location of the parent of the newly added node.

```
1. Input $n$ elements in the heap H.
2. Add new node by incrementing the size of the heap H: $n = n + 1$ and LOC = $n$
3. Repeat step 4 to 7 while (LOC > 1)
4. PAR = LOC/2
5. If (data <= HA[PAR])
   (a) HA[LOC] = data
   (b) Exit
6. HA[LOC] = HA[PAR]
7. LOC = PAR
8. HA[1] = data
9. Exit
```

DELETING THE ROOT OF A HEAP

Let H be a heap with *n* elements. The root R of H can be deleted as follows:

(*a*) Assign the root R to some variable data.

(*b*) Replace the deleted node R by the last node (or recently added node) L of H so that H is still a complete tree, but not necessarily a heap.

(*c*) Now rearrange H in such a way by placing L (new root) at the appropriate place, so that H is finally a heap.

Consider the heap H in Fig. 6.4 where R = 74 is the root and L = 23 is the last node (or recently added node) of the tree. Suppose we want to delete the root node R = 74. Apply the above rules to delete the root. Delete the root node R and assign it to *data* (*i.e.*, data = 74) as shown in Fig. 6.17.

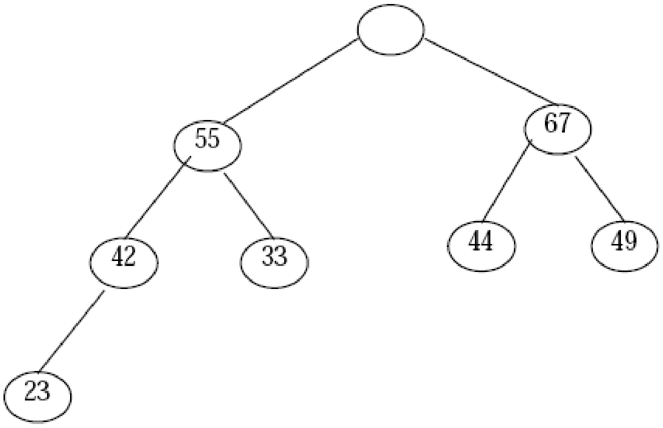


Fig. 6.17

Replace the deleted root node R by the last node L as shown in the Fig. 6.18.

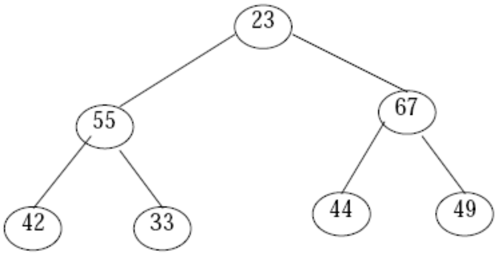


Fig. 6.18

Compare 23 with its new two children 55 and 67. Since 23 is less than the largest child 67, interchange 23 and 67. The new tree looks like as in Fig. 6.19.

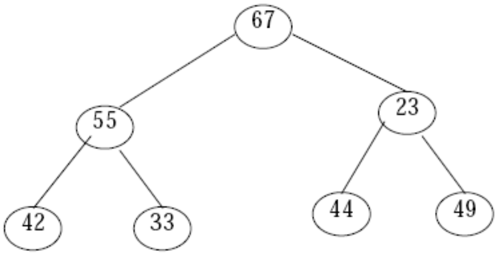


Fig. 6.19

Again compare 23 with its new two children, 44 and 49. Since 23 is less than the largest child 49, interchange 23 and 49as shown in Fig. 6.20.

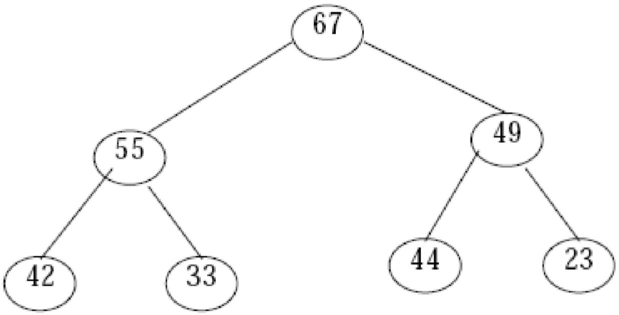


Fig. 6.20

The Fig. 6.20 is the required heap H without its original root R.

ALGORITHM

Let H be a heap with *n* elements stored in the array HA. *data* is the item of the node to be removed. *Last* gives the information about last node of H. The LOC, left, right gives the location of Last and its left and right children as the Last rearranges in the appropriate place in the tree.

```
1. Input n elements in the heap H
2. Data = HA[1]; last = HA[n] and n = n – 1
3. LOC = 1, left = 2 and right = 3
4. Repeat the steps 5, 6 and 7 while (right <= n)
5. If (last >= HA[left]) and (last >= HA[right])
(a) HA[LOC] = last
(b) Exit
6. If (HA[right] <= HA[left])
```

```
        (i) HA[LOC] = HA[left]
        (ii) LOC = left
    (b) Else
        (i) HA[LOC] = HA[right]
        (ii) LOC = right
7. left = 2 × LOC; right = left +1
8. If (left = n ) and (last < HA[left])
    (a) LOC = left
9. HA[LOC] = last
10. Exit
```

```
//PROGRAM TO IMPLEMENT HEAP SORT USING ARRAYS IN C
#include<conio.h>
#include<stdio.h>
int arr[20],n;
//Function to display the elements in the array
void display()
{ int i;
for(i=0;i<n;i++)
printf (“%d ”,arr[i]);
printf (“\n”);
}/*End of display()*/
//Function to insert an element to the heap
void insert(int num,int loc)
{
int par;
while(loc>0)
{
par=(loc–1)/2;
if (num<=arr[par])
{
arr[loc]=num;
return;
}
arr[loc]=arr[par];
loc=par;
}/*End of while*/
arr[0]=num;
}/*End of insert()*/
//This function to create a heap
void create_heap()
{
int i;
for(i=0;i<n;i++)
insert(arr[i],i);
}/*End of create_heap()*/
//Function to delete the root node of the tree
void del_root(int last)
{
int left,right,i,temp;
i=0; /*Since every time we have to replace root with last*/
/*Exchange last element with the root */
temp=arr[i];
arr[i]=arr[last];
arr[last]=temp;
left=2*i+1; /*left child of root*/
right=2*i+2;/*right child of root*/
while( right < last)
{
if ( arr[i]>=arr[left] && arr[i]>=arr[right] )
return;
if ( arr[right]<=arr[left] )
{
temp=arr[i];
arr[i]=arr[left];
arr[left]=temp;
i=left;
}
else
{
temp=arr[i];
arr[i]=arr[right];
arr[right]=temp;
i=right;
}
left=2*i+1;
right=2*i+2;
}/*End of while*/
if ( left==last–1 && arr[i]<arr[left] )/*right==last*/
{
temp=arr[i];
arr[i]=arr[left];
arr[left]=temp;
}
}/*End of del_root*/
//Function to sort an element in the heap
void heap_sort()
{
int last;
for(last=n–1; last>0; last--)
del_root(last);
}/*End of del_root*/
void main()
{
int i;
clrscr();
printf (“\nEnter number of elements : ”);
scanf (“%d”,&n);
for(i=0;i<n;i++)
{
printf (“\nEnter element %d : ”,i+1);
scanf (“%d”,&arr[i]);
}
printf (“\nEntered list is :\n”);
display();
create_heap();
printf (“\nHeap is :\n”);
display();
heap_sort();
printf (“\nSorted list is :\n”);
display();
getch();
}/*End of main()*/
```

TIME COMPLEXITY

When we calculate the time complexity of the heap sort algorithm, we need to analyse the two phases separately.

Phase 1: Let H be a heap and suppose you want to insert a new element data in H. Then few comparisons are required to locate the appropriate place, and it cannot exceed the depth of H. Since H is a complete tree, its depth is bounded by log m where m is the number of elements in H. Then

$f(n) = \text{O}(n \log n)$

Note that the number of comparison in the worst case is $\text{O}(n \log n)$

In the second phase we analyse the complexity of the algorithm to delete a root element from the heap H with n elements.

Phase 2: Suppose H is a complete tree with $n - 1 = m$ elements, and suppose the left and right sub tree of H are heaps and L is the root of H. Rearranging the node L will take four comparisons to move one step down in the tree H. Since the depth of H does not exceed $\log_2 m$, rearranging will take at most $4 \log_2 m$ comparisons to find the appropriate place of L in the tree H.

$$f(n) = 4n \log_2 n$$