Assignment-II(16 marks answer)

1.Explain with example the insertion & deletion in Singly linked list

Single Linked List

A singly linked list is a list in which each node contains only one link field pointing to the next node in the list.

A node in this type of linked list contains two types of fields.

Data – This holds the list element

Next – Pointer to the next node in the list

Data

E.g

Next Ptr

The actual representation of the above list is shown below.

The list contains five structures. The structures are stored at memory locations 800, 1600, 1800, 1985, 2600 respectively. The Next pointer in the first structure contains the value 1600.

Basic linked List Operations

The basic operations to be performed on linked lists are as

Creation - Create a linked list

Insertion - insert a new node at the specified position

Deletion - delete the specified node

Traversing - to display every node information

Find - Search a particular data

Implementation

For easy implementation of all linked list operation a sentinel node is maintained to point the beginning of the list. This node is sometimes referred to as a header or dummy node. To access the list, we must know the address of the header Node.

To insert a new node at the beginning of the list, we have to change the pointer of the head node. If we miss to do this we can lose the list. Likewise deleting a node from the front of the list is also a special case, because it changes the head of the list.

To solve the above mentioned problem, we will keep a sentinel node.

A linked list with header representation is shown below.

Type Declaration

```
typedef struct Node *PtrToNode;
typedef PtrToNode Position;
typedef PtrToNode List;
struct Node
{
ElementType Element;
Position Next;
}
```

Creating Linked List

The malloc( ) function is used to allocate a block of memory to a node in a linked list. The create function is used to create a dummy header node. It is shown in the below

```
List Create( )
{
List L;
L=(struct Node *)malloc(sizeof(struct Node));
L->Element=0;
L->Next=NULL;
return L;
}
```

Routine to check whether the list is empty

The empty list is shown below.

This function return true if the list is empty otherwise return fals.

```
int IsEmpty (List L) /*Returns 1 if L is empty */
{
return L->Next = = NULL;
}
```

Routine to check whether the current position is last

This function returns true if P is the last position in List L.

```
int IsLast (Position P, List L) /* Returns 1 is P is the last position in L */
{
return P->Next = = NULL;
}
```

Find Routine

This function returns the position of the element X in the List L.

```
Position Find (int X, List L)
{
/*Returns the position of X in L; NULL if X is not found */
Position P;
P = L ->next;
while (P! = NULL && P->Element ! = X)
P = P->Next;
return P;
}
```

Routine to Insert an Element in an Linked List

This function is used to insert an element X into the list after the position X. The insert action can be shown below

```
void Insert (ElementType X, List L, Position P)
/* Insert after the position P*/
{
Position Tmpcell;
Tmpcell = (struct Node*)malloc (size of (Struct Node));
If (Tmpcell == NULL)
Printf("Error! No Space in memory");
else
{
Tmpcell->Element = X;
Tmpcell->Next = P->Next;
```

```
P->Next = Tmpcell;

}

}
```

Find Previous Routine

The FindPrevious routine returns the position of the predecessor node.

```
Position FindPrevious (int x, List L)

{


/* Returns the position of the predecessor */

Position P;

P = L;

while (P->Next ! = NULL && P->Next ->Element ! = x)

P = P->next;

return P;

}
```

Find Next Routine

This function returns the position of the successor node

```
Position FindNext (int x, List L)

{

/*Returns the position of its successor */

P = L->next;

while (P->Next! = NULL && P->Element ! = x)

P = P->next;

return P->next;

}
```

Routine to Delete an Element from the List

This function deletes the first occurrence of an element from the linked list.This can be shown in below figure.

```
void Delete(int x, List L)

 Delete the first occurence of x from the List */
```

```
position P, temp;

P = Findprevious (x,L);

If (!IsLast(P,L))

{

temp = P->Next;

P ->Next = temp->Next;

Free (temp);

}

}
```

Routine to Delete the list

This function is used to release the memory allocated for the linked list.

```
void DeleteList (List L)

{

nextptr P, Temp;

P = L ->Next;

L->Next = NULL;

while (P! = NULL)

{

temp = P->next

free (P);

P = temp;
```

2.Explain Doubly Linked List with an example.

A Doubly linked list is a linked list in which each node has three fields namely data field, forward link (FLINK) and Backward Link (BLINK). FLINK points to the successor node in the list whereas BLINK points to the predecessor node.

STRUCTURE DECLARATION : -

```
Struct Node
{
int Element;
Struct Node *FLINK;
Struct Node *BLINK
};
```

ROUTINE TO INSERT AN ELEMENT IN A DOUBLY LINKED LIST

```
void Insert (int X, list L, position P)
{
Struct Node * Newnode;
Newnode = malloc (size of (Struct Node));
If (Newnode ! = NULL)
{
```

```
Newnode ->Element = X;
Newnode ->Flink = P      Flink;
P ->Flink ->Blink = Newnode;
P ->Flink = Newnode ;
Newnode ->Blink = P;
}
}
```

ROUTINE TO DELETE AN ELEMENT

```
void Delete (int X, List L)
{
position P;
P = Find (X, L);
If ( IsLast (P, L))
{
Temp = P;
P ->Blink ->Flink = NULL;
free (Temp);
}
else
{
Temp = P;
P ->Blink ->Flink = P->Flink;
P ->Flink ->Blink = P->Blink;
free (Temp);
}
}
```

Advantage

* Deletion operation is easier.
* Finding the predecessor & Successor of a node is easier.

Disadvantage

* More Memory Space is required since it has two pointers.

3.Explain with example the insertion & deletion in circularly linked list

(Refer singly LL)

Last->next=head;

4.Write a C Program in polynomial addition.

```
void polyadd(struct link *poly1,struct link *poly2,struct link *poly)
{
   while(poly1->next &&  poly2->next)
   {
       if(poly1->pow>poly2->pow)
       {
           poly->pow=poly1->pow;
           poly->coeff=poly1->coeff;
           poly1=poly1->next;
       }
       else if(poly1->pow<poly2->pow)
       {
           poly->pow=poly2->pow;
           poly->coeff=poly2->coeff;
           poly2=poly2->next;
       }
       else
       {
           poly->pow=poly1->pow;
           poly->coeff=poly1->coeff+poly2->coeff;
           poly1=poly1->next;
```

```
            poly2=poly2->next;
        }
        poly->next=(struct link *)malloc(sizeof(struct link));
        poly=poly->next;
        poly->next=NULL;
    }
    while(poly1->next || poly2->next)
    {
        if(poly1->next)
        {
            poly->pow=poly1->pow;
            poly->coeff=poly1->coeff;
            poly1=poly1->next;
        }
        if(poly2->next)
        {
            poly->pow=poly2->pow;
            poly->coeff=poly2->coeff;
            poly2=poly2->next;
        }
        poly->next=(struct link *)malloc(sizeof(struct link));
        poly=poly->next;
        poly->next=NULL;
    }
      }
```

5.Explain about polynomial subtraction.

```
void polyadd(struct link *poly1,struct link *poly2,struct link *poly)
{
    while(poly1->next &&  poly2->next)
    {
        if(poly1->pow>poly2->pow)
        {
            poly->pow=poly1->pow;
            poly->coeff=poly1->coeff;
            poly1=poly1->next;
        }
        else if(poly1->pow<poly2->pow)
        {
            poly->pow=poly2->pow;
            poly->coeff=poly2->coeff;
            poly2=poly2->next;
        }
        else
        {
            poly->pow=poly1->pow;
            poly->coeff=poly1->coeff-poly2->coeff;
            poly1=poly1->next;
            poly2=poly2->next;
        }
        poly->next=(struct link *)malloc(sizeof(struct link));
        poly=poly->next;
        poly->next=NULL;
    }
    while(poly1->next || poly2->next)
    {
        if(poly1->next)
        {
            poly->pow=poly1->pow;
            poly->coeff=poly1->coeff;
            poly1=poly1->next;
        }
```

```
    if(poly2->next)
    {
        poly->pow=poly2->pow;
        poly->coeff=poly2->coeff;
        poly2=poly2->next;
    }
    poly->next=(struct link *)malloc(sizeof(struct link));
    poly=poly->next;
    poly->next=NULL;
}
  }
```

6.Explain Array implementation of list ADT.

Array-based Implementation

In this implementation, the list elements are stored in contiguous cells of an array. All the list operation can be implemented by using the array.

Insertion

Insertion refers to the operation of adding another element to the list at the specified position.

If an element is inserted at the end of the array, if there is a space to add the element then insertion can be done easily. If we want to insert an element in the middle of the array then half of the elements must be moved downwards to new location to accommodate the new element and retain the order of the element. If an element is inserted at the beginning of the array then the entire array elements can be moved downward one step to make space for new element.

Routine to insert an element at the specified position

```
void insert(int *a , int pos, int num)
{
int i;
for(i=max-1;i>=pos;i--)
a[i]=a[i-1];
a[i]=num;
}
```

The running time for insertion operation as O(n).

Deletion

Deletion refers to the operation of removing an element from the array.

Deleting the element from the end of the array can be done easily. Deleting the first element of the array requires shifting all elements in the list up one. Deleting the other elements requires half of the list needs to be moved.

Routine for Delete an element from the specified position

```
void del(int *a ,int pos)
{
int i;
for(i=pos;i<max;i++)
a[i-1]=a[i];
a[i]=0;
```

```
}
```
Delete the first occurrence of the specified element
```
void del(int *a,int d)
{
int i,pos;
pos=-1;
for(i=0;i<max;i++)
{
if(a[i] = =d)
{pos=i;
break;
}
}
if(pos==-1)
printf("\n Element not found in the list");
else
{
for(i=pos;i<max;i++)
a[i-1]=a[i];
a[i-1]=0;
}
}
```
Search

This operation is used to check whether the given element is present in the list or not
```
/*Search a given element */
void search(int *a,int d);
{
int i,pos;
pos =-1;
for(i=0;i<max;i++)
{
if(a[i]==d)
{
pos =i;
break;
}
}
if(pos==-1)
printf("\n Element not found in the list");
else
printf("\n Element found at position %d",pos);
```

}

Display

Display all elements in the array.

/* display the list elements */

void display(int *a)

{

int i;

printf("\n List……");

for(i=0;i<max;i++)

printf("%d\t",a[i]);

}

Disadvantages of Array Implementation'

Even if the array is dynamically allocated, an estimate of the maximum size of the list is required. Usually this requires a high overestimate, which waste considerable

7.Write a procedure to insert &delete a element in the array implementation of stack

Array Implementation of Stacks

If we use an array implementation, the implementation is trivial. Associated with each stack is the top of stack, tos, which is -1 for an empty stack (this is how an empty stack is initialized).

To push some element x onto the stack, we increment tos and then set STACK[tos] = x, where STACK is the array representing the actual stack.

To pop, we set the return value to STACK[tos] and then decrement tos. Of course, since there are potentially several stacks, the STACK array and tos are part of one structure representing a stack.

struct stack_record

{

unsigned int stack_size;

int top_of_stack;

element_type *stack_array;

};

typedef struct stack_record *STACK;

#define EMPTY_TOS (-1) /* Signifies an empty stack */

STACK create_stack(unsigned int max_elements)

```
{
STACK S;
if( max_elements < MIN_STACK_SIZE )
error("Stack size is too small");
S = (STACK) malloc( sizeof( struct stack_record ) );
if( S == NULL )
fatal_error("Out of space!!!");
S->stack_array = (element_type *)
malloc( sizeof( element_type ) * max_elements );
if( S->stack_array == NULL )
fatal_error("Out of space!!!");
S->top_of_stack = EMPTY_TOS;
/*S->stack_size = max_elements;
return( S );
}
void dispose_stack( STACK S )
{
if( S != NULL )
{
free( S->stack_array );
free( S );
}
}
int is_empty( STACK S )
{
return( S->top_of_stack == EMPTY_TOS );
}
Void make_null( STACK S )
{
S->top_of_stack = EMPTY_TOS;
```

```
    }
    Void push( element_type x, STACK S )
    {
    if( is_full( S ) )
    error("Full stack");
    else
    S->stack_array[ ++S->top_of_stack ] = x;
    }
    element_type top( STACK S )
    {
    if( is_empty( S ) )
    error("Empty stack");
    else
    return S->stack_array[ S->top_of_stack ];
    }
    void pop( STACK S )
    {
    if( is_empty( S ) )
    error("Empty stack");
    else
    S->top_of_stack--;
    }
    element_type pop( STACK S )
    {
    if( is_empty( S ) )
    error("Empty stack");
    else
    return S->stack_array[ S->top_of_stack-- ];
}
```

8.Write a procedure to insert & delete a element in the linked list implementation of stack.

Stack ADT

A stack is a list with the restriction that inserts and deletes can be performed in

only one position, namely the end of the list called the top.

The fundamental operations on a stack are

Stack Model

push, which is equivalent to an insert,

pop, which deletes the most recently inserted element.

Example

Implementation of Stack

Linked List Implementation of Stacks

We perform a push by inserting at the front of the list. We perform a pop by

deleting the element at the front of the list. A top operation merely examines the element

at the front of the list, returning its value.

pop on an empty stack or a push on a full stack will overflow the array bounds

and cause a crash.

```
typedef struct node *node_ptr;
struct node
{
element_type element;
node_ptr next;
};
typedef node_ptr STACK;
int is_empty( STACK S )
{
return( S->next == NULL );
}
STACK create_stack( void )
{
STACK S;
```

```
S = (STACK) malloc( sizeof( struct node ) );
if( S == NULL )
fatal_error("Out of space!!!");
return S;
}
void make_null( STACK S )
{
if( S != NULL )
S->next = NULL;
else
error("Must use create_stack first");
}
void push( element_type x, STACK S )
{
node_ptr tmp_cell;
tmp_cell = (node_ptr) malloc( sizeof ( struct node ) );
if( tmp_cell == NULL )
fatal_error("Out of space!!!");
else
{
tmp_cell->element = x;
tmp_cell->next = S->next;
S->next = tmp_cell;
}
}
element_type top( STACK S )
{
if( is_empty( S ) )
error("Empty stack");
else
```

```
return S->next->element;

}

Void pop( STACK S )

{

node_ptr first_cell;

if( is_empty( S ) )

error("Empty stack");

else

{

first_cell = S->next;

S->next = S->next->next;

free( first_cell );

}

}
```

9.Write a procedure to insert & delete a element in the array implementation of queue

Array implementation

A Queue is a linear data structure which follows First In First Out (FIFO) principle, in which insertion is performed at rear end and deletion is performed at front end.

Example : Waiting Line in Reservation Counter,

Operations on Queue

The fundamental operations performed on queue are

1. Enqueue
2. Dequeue

Enqueue :

The process of inserting an element in the queue.

Dequeue :

The process of deleting an element from the queue.

Exception Conditions

Overflow : Attempt to insert an element, when the queue is full is said to be overflow condition.

Underflow : Attempt to delete an element from the queue, when the queue is empty is said to be underflow.

Implementation of Queue

Queue can be implemented using arrays and pointers.

Array Implementation

In this implementation queue Q is associated with two pointers namely rear pointer and front pointer.

To insert an element X onto the Queue Q, the rear pointer is incremented by 1 and then set

Queue [Rear] = X

To delete an element, the Queue [Front] is returned and the Front Pointer is incremented by 1.

ROUTINE TO ENQUEUE

```
voidEnqueue (int X)
{
if (rear > = max _ Arraysize)
```

```
print (" Queue overflow");
else
{
Rear = Rear + 1;
Queue [Rear] = X;
}
}
```

ROUTINE FOR DEQUEUE

```
void delete ( )
{
if (Front < 0)
print (" Queue Underflow");
else
{
X = Queue [Front];
if (Front = = Rear)
{
Front = 0;
Rear = -1;
}
else
Front = Front + 1 ;
}
}
```

10.Write a procedure to insert & delete a element in the linked list implementation of queue

Linked implementation

Enqueue operation is performed at the end of the list.

Dequeue operation is performed at the front of the list.

```
void create()
{
printf("\nENTER THE FIRST ELEMENT: ");
cur=(struct node *)malloc(sizeof(struct node));
scanf("%d",&cur->data);
cur->link=NULL;
first=cur;
}

void display()
{
cur=first;
printf("\n");
while(cur!=NULL)
{
printf("%d\n",cur->data);
cur=cur->link;
}
}
void push()
{
printf("\nENTER THE NEXT ELEMENT: ");
cur=(struct node *)malloc(sizeof(struct node));
scanf("%d",&cur->data);
cur->link=first;
first=cur;
}
```

```
void pop()
{
if(first==NULL)
{
printf("\nSTACK IS EMPTY\n");
}
else
{
cur=first;
printf("\nDELETED ELEMENT IS %d\n",first->data);
first=first->link;
free(cur);
}
}
```

11.Explain Insertion sort with example

One of the simplest sorting algorithms is the insertion sort. Insertion sort consists of n - 1 passes. For pass p = 2 through n, insertion sort ensures that the elements in positions 1 through p are in sorted order. Insertion sort makes use of the fact that elements in positions 1 through p - 1 are already known to be in sorted order. Figure shows a sample file after each pass of insertion sort. Figure shows the general strategy. In pass p, we move the pth element left until its correct place is found among the first p elements. The code in Figure implements this strategy. The sentinel in a[0] terminates the while loop in the event that in some pass an element is moved all the way to the front. Lines 3 through 6 implement that data movement without the explicit use of swaps. The element in position p is saved in tmp, and all larger elements (prior to position p) are moved one spot to the right. Then tmp is placed in the correct spot.

```
void
insertion_sort( input_type a[ ], unsigned int n )
{
unsigned int j, p;
input_type tmp;
a[0] = MIN_DATA; /* sentinel */
 for( p=2; p <= n; p++ )
{
```

```
 tmp = a[p];
 for( j = p; tmp < a[j-1]; j-- )
a[j] = a[j-1];

a[j] = tmp;
}
}
```

Original 34 8 64 51 32 21 Positions Moved

After p = 2 8 34 64 51 32 21 1

After p = 3 8 34 64 51 32 21 0

After p = 4 8 34 51 64 32 21 1

After p = 5 8 32 34 51 64 21 3

After p = 6 8 21 32 34 51 64 4

Implementation of insertion sort:

```
#include <stdio.h>
 int main()
{
  int n, array[1000], c, d, t;
```

```
printf("Enter number of elements\n");
scanf("%d", &n);

printf("Enter %d integers\n", n);

for (c = 0; c < n; c++) {
  scanf("%d", &array[c]);
}

for (c = 1 ; c <= n - 1; c++) {
  d = c;

  while ( d > 0 && array[d] < array[d-1]) {
    t          = array[d];
    array[d]   = array[d-1];
    array[d-1] = t;

    d--;
  }
}

printf("Sorted list in ascending order:\n");

for (c = 0; c <= n - 1; c++) {
  printf("%d\n", array[c]);
}

 return 0;
}
```

12. Write a note on shell sort with example.

Routine for Shell sort

```
shellsort( input_type a[ ], unsigned int n )
{
unsigned int i, j, increment;
input_type tmp;
for( increment = n/2; increment > 0; increment /= 2 )
 for( i = increment+1; i<=n; i++ )
{
 tmp = a[i];
 for( j = i; j > increment; j -= increment )
 if( tmp < a[j-increment] )
a[j] = a[j-increment];
else
break;
 a[j] = tmp;
}
}
```

Example:

Original 81 94 11 93 12 35 17 95 28 58 41 75 15

After 5-sort 35 17 11 28 12 41 75 15 96 58 81 94 95

After 3-sort 28 12 11 35 15 41 58 17 94 75 81 96 95

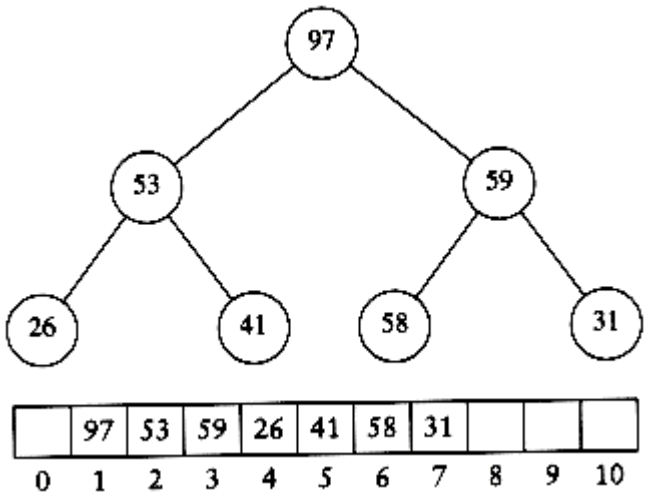
After 1-sort 11 12 15 17 28 35 41 58 75 81 94 95 96

13. Discuss briefly about heap sort with example

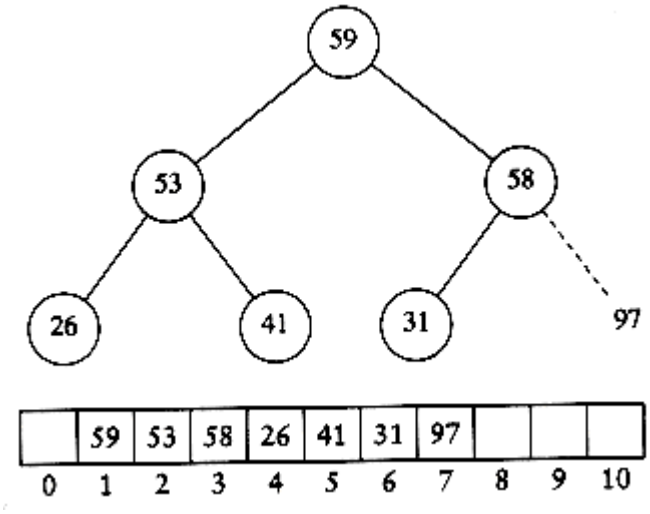
Routine for Heap sort

```
Void  heapsort( input_type a[], unsigned int n )
{
int i;
 for( i=n/2; i>0; i-- ) /* build_heap */
perc_down (a, i, n );
for( i=n; i>=2; i-- )
{
swap( &a[1], &a[i] ); /* delete_max */
perc_down( a, 1, i-1 );
}
}
void
perc_down( input_type a[], unsigned int i, unsigned int n )
{
unsigned int child;
input_type tmp;
for( tmp=a[i]; i*2<=n; i=child )
{
 child = i*2;
if( ( child != n ) && ( a[child+1] > a[child] ) )
child++;
if( tmp < a[child] )
a[i] = a[child];
```

```
else
break;
}
 a[i] = tmp;
}
```



(Max) heap after build_heap phase

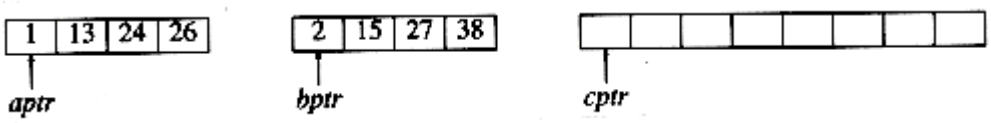


Heap after first delete_max

14.Explain Merge sort with Example.

It is a fine example of a recursive algorithm. The fundamental operation in this algorithm is merging two sorted lists. Because the lists are sorted, this can be done in one pass through the input, if the output is put in a third list. The basic merging algorithm takes two input arrays a and b, an output array c, and three counters, aptr, bptr, and cptr, which are initially set to the beginning of their respective arrays. The smaller of a[aptr] and b[bptr] is copied to the next entry in c, and the appropriate counters are advanced. When either input list is exhausted, the remainder of the other list is copied to c. An example of how the merge routine works is provided for the

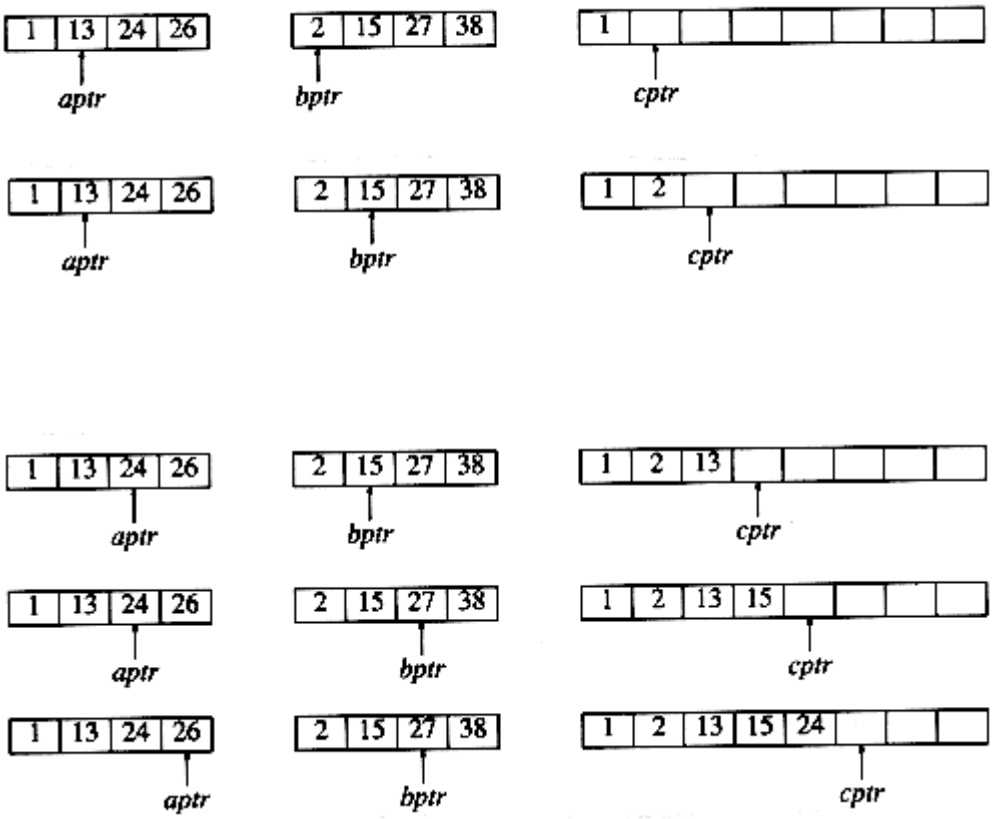
following input.

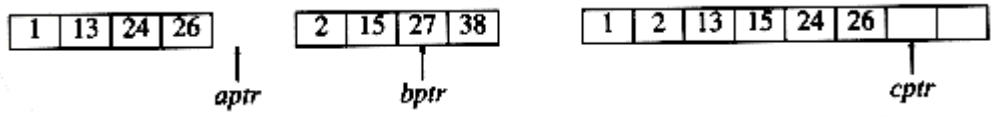


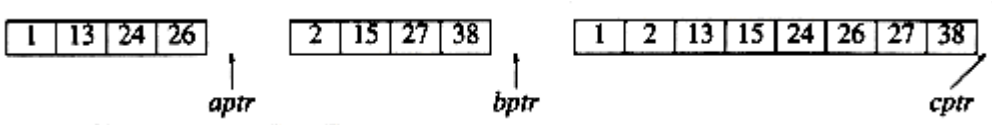
If the array a contains 1, 13, 24, 26, and b contains 2, 15, 27, 38, then the

algorithm proceeds as follows: First, a comparison is done between 1 and 2. 1 is

added to c, and then 13 and 2 are compared.


1 13 24 26
aptr
2 15 27 38
bptr
1
cptr
1 13 24 26
aptr
2 15 27 38
bptr
1 2
cptr
1 13 24 26
aptr
2 15 27 38
bptr
1 2 13
cptr
1 13 24 26
aptr
2 15 27 38
bptr
1 2 13 15
cptr
1 13 24 26
aptr
2 15 27 38
bptr
1 2 13 15 24
cptr


1 13 24 26
aptr
2 15 27 38
bptr
1 2 13 15 24 26
cptr



Routine for Merge sort Algorithm:

```
Void  mergesort( input_type a[], unsigned int n )
{
input_type *tmp_array;
tmp_array = (input_type *) malloc
( (n+1) * sizeof (input_type) );
if( tmp_array != NULL )
{
m_sort( a, tmp_array, 1, n );
free( tmp_array );
}
else
fatal_error("No space for tmp array!!!");
}
void
m_sort( input_type a[], input_type tmp_array[ ],
int left, int right )
{
int center;
if( left < right )
{
center = (left + right) / 2;
m_sort( a, tmp_array, left, center );
m_sort( a, tmp_array, center+1, right );
merge( a, tmp_array, left, center+1, right );
```

```
}
}
```

Implementation of Merge sort:

```
#include<stdio.h>
#define MAX 50
void mergeSort(int arr[],int low,int mid,int high);
void partition(int arr[],int low,int high);
int main()
{
    int merge[MAX],i,n;
    printf("Enter the total number of elements: ");
    scanf("%d",&n);
    printf("Enter the elements which to be sort: ");
            for(i=0;i<n;i++)
            {
             scanf("%d",&merge[i]);
            }
    partition(merge,0,n-1);
    printf("After merge sorting elements are: ");
            for(i=0;i<n;i++)
            {
            printf("%d ",merge[i]);
            }
  return 0;
}
void partition(int arr[],int low,int high)
{
 int mid;
```

```
        if(low<high)
        {
        mid=(low+high)/2;
        partition(arr,low,mid);
        partition(arr,mid+1,high);
        mergeSort(arr,low,mid,high);
        }
}
void mergeSort(int arr[],int low,int mid,int high)
{
   int i,m,k,l,temp[MAX];
   l=low;
   i=low;
   m=mid+1;
                while((l<=mid)&&(m<=high))
                {
                        if(arr[l]<=arr[m])
                        {
                temp[i]=arr[l];
                l++;
                        }
                        else
                        {
                temp[i]=arr[m];
                m++;
                        }
                i++;
                }

                if(l>mid)
```

```
{
        for(k=m;k<=high;k++)
        {
temp[i]=arr[k];
 i++;
        }
}
else
{
        for(k=l;k<=mid;k++)
        {
 temp[i]=arr[k];
 i++;
        }
}
for(k=low;k<=high;k++)
{
arr[k]=temp[k];
}
}
```

15.Explain Quick sort in detail with an eg.

As its name implies, quicksort is the fastest known sorting algorithm in practice. Its average running time is O(n log n). It is very fast, mainly due to a very tight and highly optimized inner loop. It has O(n2) worst-case performance, but this can be made exponentially unlikely with a little effort. The quicksort algorithm is simple to understand and prove correct, although for many years it had the reputation of being an algorithm that could in theory be highly optimized but in practice was impossible to code correctly (no doubt because of FORTRAN). Like mergesort, quicksort is a divide-and-conquer

recursive algorithm.

Routine for Quick sort

```
Void  q_sort( input_type a[], int left, int right )
{
int i, j;
input_type pivot;
if( left + CUTOFF <= right )
{
pivot = median3( a, left, right );
i=left; j=right-1;

{
 while( a[++i] < pivot );
 while( a[--j] > pivot );
 if( i < j )
swap( &a[i], &a[j] );
else
 break;
}
swap( &a[i], &a[right-1] ); /*restore pivot*/
q_sort( a, left, i-1 );
q_sort( a, i+1, right );
}
}
```

Figure 7.14 Main quicksort routine

```
i=left+1; j=right-2;

{
while( a[i] < pivot ) i++;
 while( a[j] > pivot ) j--;
```

```
 if( i < j )
swap( &a[i], &a[j] );
else
 break;
}
```

Implementation of Quick Sort:

```
#include<conio.h>
#include<stdio.h>
int qsort[25];
void sort(int low,int high);

void sort(int low,int high)
{
int temp;
        if(low<high)
        {
                int i=low+1;
                int j=high;
                 int t=qsort[low];
                        while(1)
                        {
                                while(qsort[i]<t) i++;
                                 while(qsort[j]>t) j--;
                                        if(i<j)
                                        {
                                                temp=qsort[i];
                                                qsort[i]=qsort[j];
                                                 qsort[j]=temp;
                                                 i++;j--;
                                        }
```

```
                                    else break;
                                        }
            qsort[low]=qsort[j];
             qsort[j]=t;
            sort(low,j-1);
             sort(j+1,high);
            }
else return;
}
void main()
{
int n;
clrscr();
printf("\nEnter no of elements:");
scanf("%d",&n);
printf("\nEnter elements to be sorted");
for(int i=1;i<=n;i++)
scanf("%d",&qsort[i]);
sort(1,n);
printf("\nSORTED ELEMENT :");
for(i=1;i<=n;i++)
printf("%d\t",qsort[i]);
getch();
}
```

Example

8 1 4 9 0 3 5 2 7 6

i j

We then swap the elements pointed to by i and j and repeat the process until i

and j cross.

After First Swap

---

2 1 4 9 0 3 5 8 7 6

i j

Before Second Swap

---

2 1 4 9 0 3 5 8 7 6

i j

After Second Swap

---

2 1 4 5 0 3 9 8 7 6

i j

Before Third Swap

---

2 1 4 5 0 3 9 8 7 6

j i

At this stage, i and j have crossed, so no swap is performed. The final part of

the partitioning is to swap the pivot element with the element pointed to by i.

After Swap with Pivot

---

2 1 4 5 0 3 6 8 7 9

i pivot

16.Define Hash function. Write routines to find and insert an element in separate chaining.

The implementation of hash tables is frequently called hashing. Hashing is a

technique used for performing insertions, deletions and finds in constant average

time.

The ideal hash table data structure is merely an array of some fixed size,

containing the keys. Typically, a key is a string with an associated value (for

instance, salary information). We will refer to the table size as H_SIZE, with

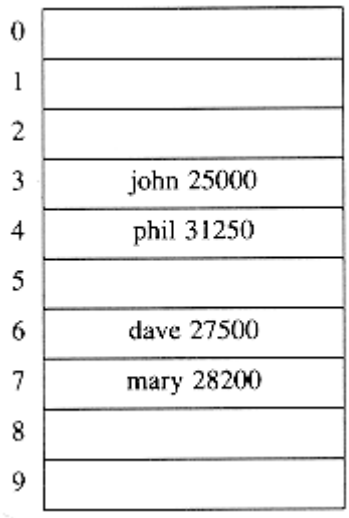
the understanding that this is part of a hash data structure and not merely some

variable floating around globally. The common convention is to have the table run

from 0 to H_SIZE-1; we will see why shortly. Each key is mapped into some number in the range 0 to H_SIZE - 1 and placed in the appropriate cell. The mapping is called a hash function, which ideally should be simple to compute and should ensure that any two distinct keys get different cells. Figure 5.1 is typical of a perfect

situation. In this example, john hashes to 3, phil hashes to 4, dave hashes to 6,

and mary hashes to 7.



Open Hashing (Separate Chaining)

The first strategy, commonly known as either open hashing, or separate chaining,

is to keep a list of all elements that hash to the same value. For convenience,

our lists have headers. This makes the list implementation the same as in Chapter

3. If space is tight, it might be preferable to avoid their use. We assume for

this section that the keys are the first 10 perfect squares and that the hashing

function is simply hash(x) = x mod 10. (The table size is not prime, but is used

here for simplicity.) Figure 5.6 should make this clear. To perform a find, we use the hash function to determine which list to traverse. We then traverse this list in the normal manner, returning the position where the item is found. To perform an insert, we traverse down the appropriate list to check whether the element is already in place (if duplicates are expected, an extra field is usually kept, and this field would be incremented in the event of a match). If the element turns out to be new, it is inserted either

at the front of the list or at the end of the list, whichever is easiest. This is an issue most easily addressed while the code is being written. Sometimes new elements are inserted at the front of the list, since it is convenient and also because

frequently it happens that recently inserted elements are the most likely to be

accessed in the near future

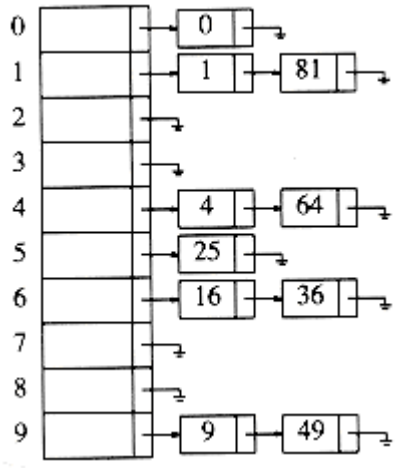


Figure 5.6 An open hash table

Type declaration for open hash table

```
typedef struct list_node *node_ptr;

struct list_node

{

element_type element;

node_ptr next;

};

typedef node_ptr LIST;

typedef node_ptr position;


/* LIST *the_list will be an array of lists, allocated later */
```

```
/* The lists will use headers, allocated later */

struct hash_tbl
{

unsigned int table_size;
LIST *the_lists;
};
```

Initialization routine for open hash table

```
HASH_TABLE
initialize_table( unsigned int table_size )
{
HASH_TABLE H;
int i;
if( table size < MIN_TABLE_SIZE )
{
error("Table size too small");
 return NULL;
}
/* Allocate table */
H = (HASH_TABLE) malloc ( sizeof (struct hash_tbl) );
if( H == NULL )
fatal_error("Out of space!!!");
H->table_size = next_prime( table_size );
/* Allocate list pointers */
H->the_lists = (position *)
malloc( sizeof (LIST) * H->table_size );
 if( H->the_lists == NULL )
```

```
 fatal_error("Out of space!!!");
/* Allocate list headers */
 for(i=0; i<H->table_size; i++ )
{

H->the_lists[i] = (LIST) malloc
( sizeof (struct list_node) );
if( H->the_lists[i] == NULL )
fatal_error("Out of space!!!");
else
H->the_lists[i]->next = NULL;
}
return H;
```

Find routine for open hash table

```
position
find( element_type key, HASH_TABLE H )
{
position p;
LIST L;
 L = H->the_lists[ hash( key, H->table_size) ];
p = L->next;
while( (p != NULL) && (p->element != key) )



/* Probably need strcmp!! */
 p = p->next;
return p;
```

Insert routine for open hash table

```
Void insert( element_type key, HASH_TABLE H )

{

position pos, new_cell;

LIST L;

pos = find( key, H );

if( pos == NULL )

{

new_cell = (position) malloc(sizeof(struct list_node));

if( new_cell == NULL )

fatal_error("Out of space!!!");

else

{

L = H->the_lists[ hash( key, H->table size ) ];

new_cell->next = L->next;

new_cell->element = key; /* Probably need strcpy!! */

L->next = new_cell;

}
```

18. Explain Linear search with example?

The following code implements linear search (Searching algorithm) which is used to find whether a given number is present in an array and if it is present then at what location it occurs. It is also known as sequential search. It is very simple and works as follows: We keep on comparing each element with the element to search until the desired element is found or list ends. Linear search in c language for multiple occurrences and using function.

```
#include <stdio.h>
 int main()
{
   int array[100], search, c, n;
   printf("Enter the number of elements \n");
   scanf("%d",&n);
```

```
  printf("/nEnter the elements");
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
  printf("Enter the number to search\n");
  scanf("%d", &search);
  for (c = 0; c < n; c++)
  {
    if (array[c] == search)
    {
      printf("%d is present at location %d.\n", search, c+1);
      break;
    }
  }
  if (c == n)
    printf("%d is not present in array.\n", search);
    return 0;
}
```

Figure %: The array we're searching

Lets search for the number 3. We start at the beginning and check the first element in the array. Is it 3?

Figure %: Is the first value 3?

No, not it. Is it the next element?

Figure %: Is the second value 3?

Not there either. The next element?

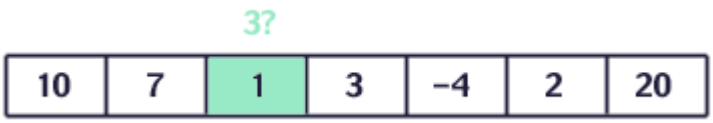


Figure %: Is the third value 3?

Not there either. Next?

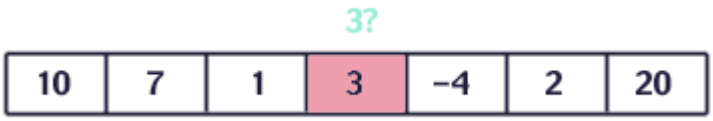


Figure %: Is the fourth value 3?

.

Hence the value 3 is found

19. Explain binary search with an example?

This code implements binary search in c language. It can only be used for sorted arrays, but it's fast as compared to linear search. If you wish to use binary search on an array which is not sorted then you must sort it using some sorting technique say merge sort and then use binary search algorithm to find the desired element in the list. If the element to be searched is found then its position is printed.

```
#include<stdio.h>
int BinarySearch(int *array, int n, int key)
{
	int low = 0, high = n-1, mid;
	while(low <= high)
	{
		mid = (low + high)/2;
		   if(array[mid] < key)
		   {
				low = mid + 1;
		   }
		   else if(array[mid] == key)
		   {
				return mid;
		   }
		   else if(array[mid] > key)
		   {
				high = mid-1;
		   }
		   }
		    return -1;
}
int main()
{
	int n;
	int array[n];
	int i,key,index;
	printf(“\nEnter the number of elements”);
	scanf("%d",&n);
	printf(“\nEnter the elements”);
		    for(i = 0;i < n;i++)
		   {
		scanf("%d",&array[i]);
		   }
		    for(i = 1;i < n;i++)
		   {
				if(array[i] < array[i - 1])
		   {
```

```
                        printf("Given input is not sorted\n");
                        return 0;
                }
                }
printf("Enter the key to be searched");
scanf("%d",&key);
        index = BinarySearch(array,n,key);
        if(index==-1)
        {
                printf("Element not found\n");




   }


 else
        {
                printf("Element is at index %d\n",index);
        }
        return 0;
}
```

Example

The list to be searched: L = 1 3 4 6 8 9 11. The value to be found: X = 4.

Compare X to 6. It's smaller. Repeat with L = 1 3 4.

Compare X to 3. It's bigger. Repeat with L = 4.

Compare X to 4. It's equal. We're done, we found X.

Explain Radix sort with an example?

Radix Sort puts the elements in order by comparing the **digits of the numbers** Radix sort is one of the linear sorting algorithms for integers. It functions by sorting the input numbers on each digit, for each of the digits in the numbers. However, the process adopted by this sort method is somewhat counterintuitive, in the sense that the numbers are sorted on the least-significant digit first, followed by the second-least significant digit and so on till the most significant digit**.**

Consider the following 9 numbers:

493 812 715 710 195 437 582 340 385

We should start sorting by comparing and ordering the **one's** digits

Digit	Sublist
0	340 710
1	
2	812 582
3	493
4	
5	715 195 385
6	
7	437
8	
9	

Notice that the numbers were added onto the list in the order that they were found, which is why the numbers appear to be unsorted in each of the sublists above. Now, we gather the sublists (in order from the 0 sublist to the 9 sublist) into the main list again: 340 710 812 582 493 715 195 385 43

Note: The **order** in which we divide and reassemble the list is **extremely important** as this is one of the foundations of this algorithm.Now, the sublists are created again, this time based on the **ten's** digit:

Digit	Sublist
0	
1	710 812 715
2	
3	437
4	340
5	
6	
7	
8	582 385
9	493 195

Now the sublists are gathered in order from 0 to 9:

710 812 715 437 340 582 385 493 195

Finally, the sublists are created according to the **hundred's** digit:

Digit	Sublist
0	
1	195
2	
3	340 385
4	437 493
5	582
6	
7	710 715
8	812
9	

At last, the list is gathered up again:

195 340 385 437 493 582 710 715 812

Implementation of Radix Sort:

```
#include<stdio.h>

#include<conio.h>

Int getMax(int arr[],int n)

{
```

```
Int mx=arr[0];
For(int i=1;i<n;i++)
if(arr[i]>mx)
mx=arr[i];
return mx;
}
Int countSort(int arr[],int n,int exp)
{
Int output[100];
Int I,count[10]= {0}
For(i=0;i<n;i++)
Count[(arr[i]/exp)%10]++;
For(i=1;i<10;i++)
For(i=n-1;i>=0;i--)
{
Output[count[(arr[i]/exp)%10]-1]=arr[i];

Count[(arr[i/exp)%10]--;
}
For(i=0;i<n;i++)
Arr[i]=output[i];
Void radix sort(int arr[],int n)
{
Int m=getMax(arr,n);
For(int exp=1;m/exp>0;exp*=10)
countSort(arr,n,exp);
}
Void print(int arr[],int n)
{
For(int i=0;i<n;i++)
```

```
Printf(“%d\t”,arr[i]);

}
```