MODULE 2

QUEUES: Queues, Circular Queues, Using Dynamic Arrays, Multiple Stacks and queues. **LINKED LISTS :** Singly Linked, Lists and Chains, Representing Chains in C, Linked Stacks and Queues, Polynomials.

QUEUES ABSTRACT DATA TYPE

DEFINITION

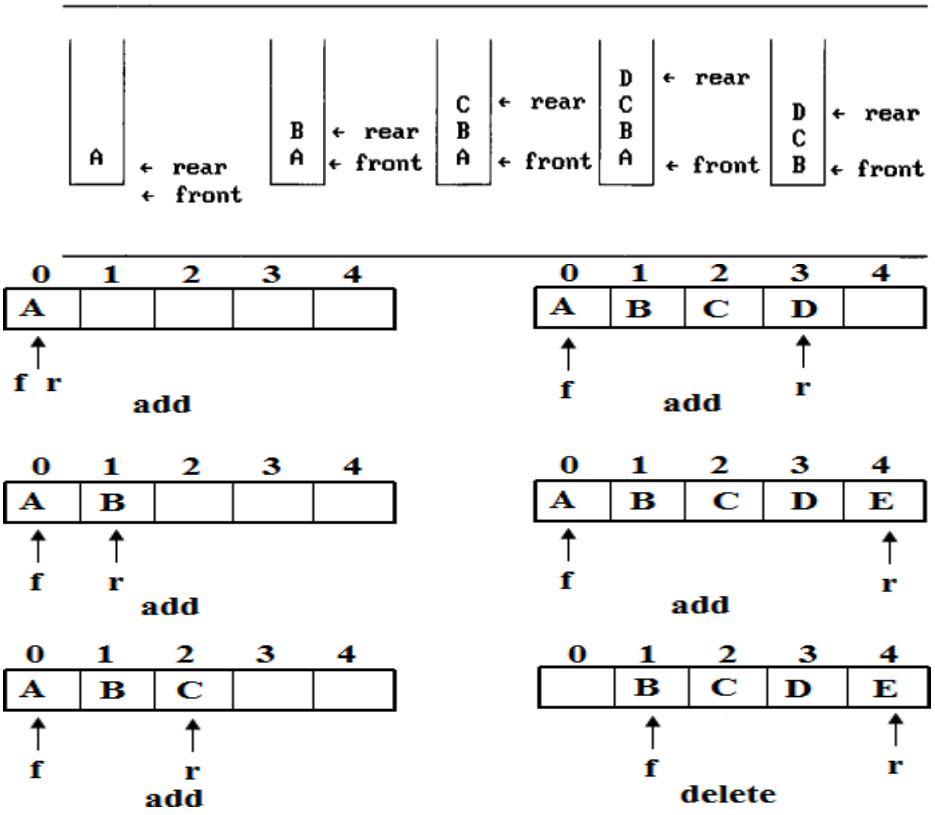
- "A queue is an ordered list in which insertions (additions, pushes) and deletions (removals and pops) take place at different ends."
- The end at which new elements are added is called the rear, and that from which old elements are deleted is called the front.
- Given a queue Q = (a_0, a_1,…… a_{n-1}) , a_0, is the front element a_{n-1} is the rear element, a_{i+1} is behind a_i 0< =i < n-1.

  If the elements are inserted A, B, C, D and E in this order, then A is the first element deleted from the queue. Since the first element inserted into a queue is the first element removed, queues are also known as First-In-First-Out (FIFO) lists.

QUEUE REPRESENTATION USING ARRAY

- Queues may be represented by one-way lists or linear arrays.
- Queues will be maintained by a linear array QUEUE and two pointer variables: FRONT-containing the location of the front element of the queue
- REAR-containing the location of the rear element of the queue.
- The condition FRONT = NULL will indicate that the queue is empty.

- Figure indicates the way elements will be deleted from the queue and the way new elements will be added to the queue.
- Whenever an element is deleted from the queue, the value of FRONT is increased by 1; this can be implemented by the assignment FRONT := FRONT + 1
- When an element is added to the queue, the value of REAR is increased by 1; this can be implemented by the assignment REAR := REAR + 1



structure *Queue* is
objects: a finite ordered list with zero or more elements.
functions:
for all *queue* ∈ *Queue*, *item* ∈ *element*, *max_queue_size* ∈ positive integer
Queue CreateQ(*max_queue_size*) ::=
create an empty queue whose maximum size is *max_queue_size*
Boolean IsFullQ(*queue*, *max_queue_size*) ::=
if (number of elements in *queue* == *max_queue_size*)
return *TRUE*
else return *FALSE*
Queue AddQ(*queue*, *item*) ::=
if (IsFullQ(*queue*)) *queue_full*
else insert *item* at rear of *queue* and return *queue*
Boolean IsEmptyQ(*queue*) ::=
if (*queue* == CreateQ(*max_queue_size*))
return *TRUE*
else return *FALSE*
Element DeleteQ(*queue*) ::=
if (IsEmptyQ(*queue*)) **return**
else remove and return the *item* at front of queue.

Structure 3.2: Abstract data type *Queue*

Implementation of the queue operations as follows.

1. Queue Create

```
Queue CreateQ(maxQueueSize) ::=
        #define MAX_QUEUE_SIZE 100          /* maximum queue size */
        typedef struct
                {
                        int key;
                        /* other fields */
                } element;
        element queue[MAX_QUEUE_SIZE];
        int rear = -1;
        int front = -1;
```

2. Boolean IsEmptyQ(queue) ::= front ==rear

3. Boolean IsFullQ(queue) ::= rear == MAX_QUEUE_SIZE-1

In the queue, two variables are used which are front and rear. The queue increments rear in addq( ) and front in delete( ). The function calls would be addq (item); and item =delete( );

4. addq(item)

```
void addq(int *rear, element item)
  {
  // add an item to the queue
                if (rear == MAX_QUEUE_SIZE-1)
                {
                        queue_Full(
                        ); return;
                }
        queue [++rear] = item;
}
```

Program: Add to a queue

5. deleteq( )

```
element deleteq(int *front, int *rear)
        {       /* remove element at the front of the queue */
                if (front == rear)
                        return queue_Empty( );          /* return an error key
                return queue[++front];
}
```

Program: Delete from a queue

6. **queueFull( )**

The queueFull function which prints an error message and terminates

```
execution void queueFull()
        {
                fprintf(stderr, "Queue is full, cannot add
                element"); exit(EXIT_FAILURE);
        }
```

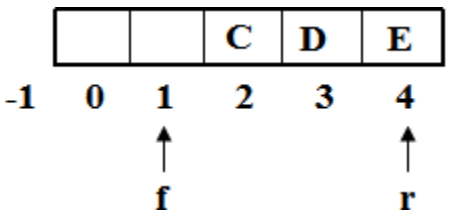
Example: Job scheduling

- Queues are frequently used in creation of a **job queue** by an operating system. If the operating system does not use priorities, then the jobs are processed in the order they enter the system.
- Figure illustrates how an operating system process jobs using a sequential representation for its queue.

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					Queue is empty
-1	0	J1				Job 1 is added
-1	1	J1	J2			Job 2 is added
-1	2	J1	J2	J3		Job 3 is added
0	2		J2	J3		Job 1 is deleted
1	2			J3		Job 2 is deleted

Drawback of Queue

When item enters and deleted from the queue, the queue gradually shifts to the right as shown in figure.

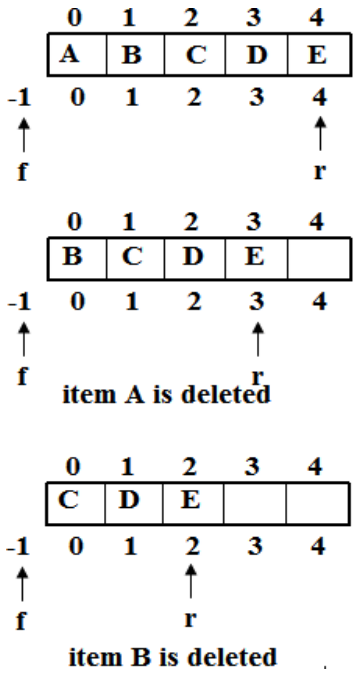


In this above situation, when we try to insert another item, which shows that the **queue is full** . This means that the **rear** index equals to MAX_QUEUE_SIZE -1. But even if the space is available at the front end, rear insertion cannot be done.

Overcome of Drawback using different methods

Method 1:

- When an item is deleted from the queue, move the entire queue to the **left** so that the first element is again at queue[0] and front is at **-1**. It should also recalculate **rear** so that it is correctly positioned.
- Shifting an array is very time-consuming when there are many elements in queue & queueFull has worst case complexity of **O(MAX_QUEUE_SIZE)**



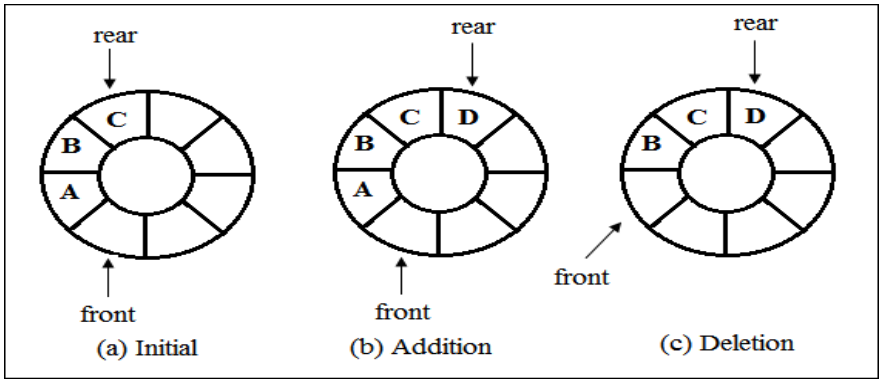
Method 2:

Circular Queue

- It is "The queue which **wrap around** the end of the array." The array positions are arranged in a circle.
- In this convention the variable **front** is changed. **front** variable points one position **counterclockwise** from the location of the front element in the queue. The convention for rear is unchanged.

CIRCULAR QUEUES

- It is "The queue which **wrap around** the end of the array." The array positions are arranged in a circle as shown in figure.
- In this convention the variable **front** is changed. **front** variable points one position **counterclockwise** from the location of the front element in the queue. The convention for rear is unchanged.



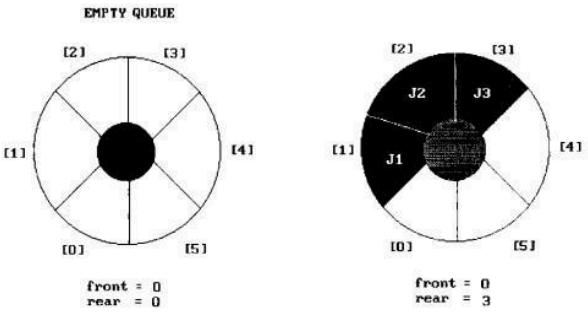


Figure 3.6: Empty and nonempty circular queues

Implementation of Circular Queue Operations

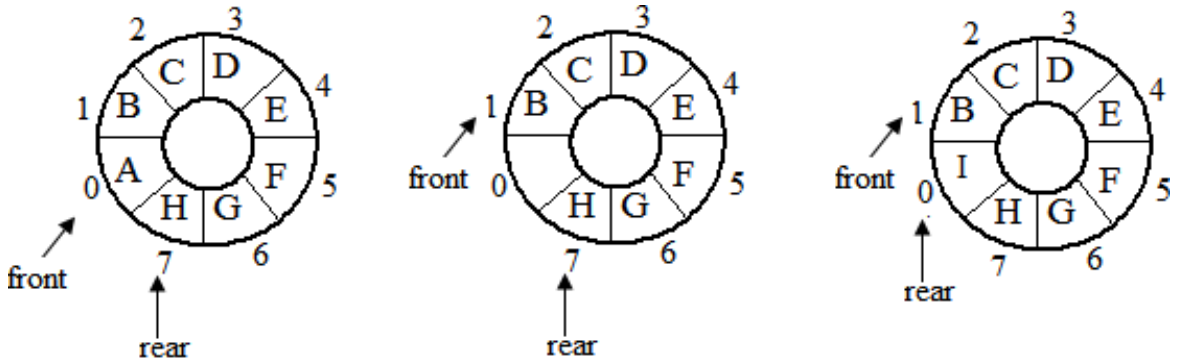
- When the array is viewed as a circle, each array position has a **next** and a **previous** position. The position next to **MAX-QUEUE-SIZE -1** is **0**, and the position that precedes **0** is **MAX-QUEUE-SIZE -1**.
- When the queue **rear** is at MAX_QUEUE_SIZE-1, the next element is inserted at position **0**.
- In circular queue, the variables **front** and **rear** are moved from their current position to the next position in clockwise direction. This may be done using code

```
if (rear = =
        MAX_QUEUE_SIZE-1)
        rear = 0;
else rear++;
```

Addition & Deletion

- To add an element, increment **rear** one position clockwise and insert at the new position. Here the **MAX_QUEUE_SIZE** is 8 and if all 8 elements are added into queue and that can be represented in below figure (a).
- To delete an element, increment **front** one position clockwise. The element **A** is deleted from queue and if we perform 6 deletions from the queue of Figure (b) in this fashion, then queue becomes empty and that **front =rear.**
- If the element **I** is added into the queue as in figure (c), then **rear** needs to increment by 1 and the value of rear is **8**. Since queue is circular, the next position should be 0 instead of 8.

  This can be done by using the modulus operator, which computes remainders.



(a) (b) (c)

```
void addq(element
item)   {               /* add an item to the queue
        rear = (rear +1)*/%
        MAX_QUEUE_SIZE; if (front ==
        rear)
                        queueFull(rear);        /* print error and exit
         */ queue [rear] = item;
        }
```

Program: Add to a circular queue

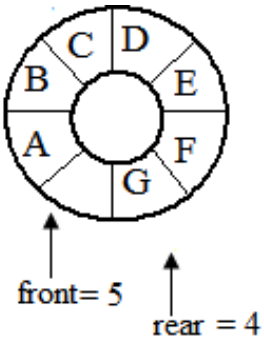
```
element
deleteq()               /* remove front element from the queue
    {           element*item;
                if (front == rear)
                        return queueEmpty( );/* return an error key */ front =
                (front+1)% MAX_QUEUE_SIZE;
                return queue[front];
    }
```

Program: Delete from a circular queue

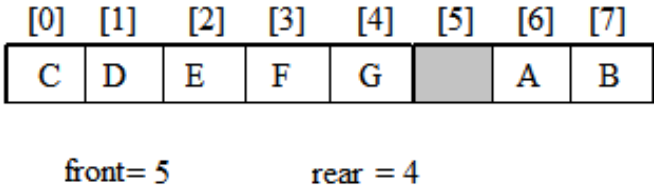
CIRCULAR QUEUES USING DYNAMIC ARRAYS

- A dynamically allocated a
- rray is used to hold the queue elements. Let **capacity** be the number of positions in the array queue.
- To add an element to a **full queue**, first increase the size of this array using a function **realloc**.

Consider the **full queue** of figure (a). This figure shows a queue with seven elements in an array whose capacity is 8. A circular queue is flatten out the array as in Figure (b).

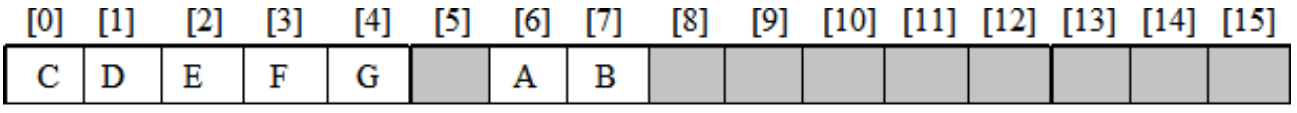


(a) A full circular queue



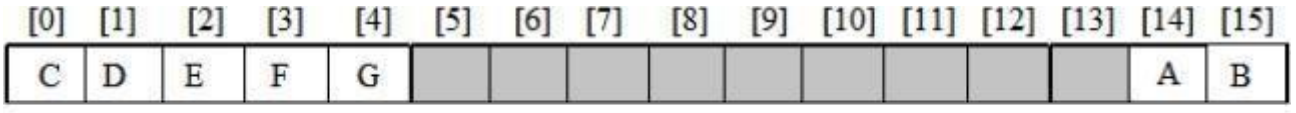
(b) Flattened view of circular full queue

Figure (c) shows the array after array doubling by relloc



(c) After array doubling

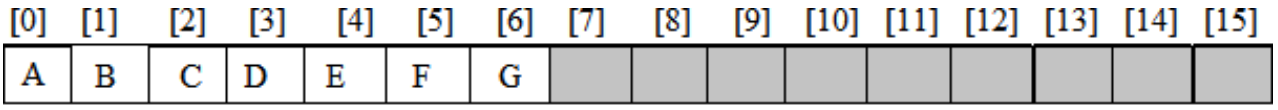
To get a proper circular queue configuration, slide the elements in the right segment (i.e., elements A and B) to the right end of the array as in figure (d)



(d After shifting right segment

To obtain the configuration as shown in figure (e), follow the steps

1) Create a new array **newQueue** of twice the capacity.
2) Copy the second segment (i.e., the elements queue [front +1] through queue [capacity-1]) to positions in **newQueue** beginning at 0.
3) Copy the first segment (i.e., the elements queue [0] through queue [rear]) to positions in newQueue beginning at **capacity – front – 1.**



(e) Alternative configuration

Below program gives the code to add to a circular queue using a dynamically allocated array.

```
void addq( element
  item)  {                  /* add an item to the
                  queue rear = (rear +1) %
                  capacity; if(front == rear)
                            queueFull( );   /* double capacity */
                  queue[rear] = item;
          }
```

Below program obtains the configuration of **figure (e)** and gives the code for queueFull. The function copy (a,b,c) copies elements from locations **a** through **b-1** to locations beginning at **c**.

```
void
queueFull( )
          {         /* allocate an array with twice the
                  capacity */ element *newQueue;
                  MALLOC ( newQueue, 2 * capacity * sizeof(* queue));
                  /* copy from queue to newQueue
                  */ int start = ( front + ) %
                  capacity;
                  if ( start < 2)   /* no wrap around */
                  else
                            copy( queue+start,
                  {                 /* queue wrap around */
                                    queue+start+capacity-1,newQueue);
```

```
copy(queue, queue+capacity, newQueue);
copy(queue, queue+rear+1, newQueue+capacity-start);
}
/* switch to newQueue*/ front =
                    2*capacity – 1; rear =
                    capacity – 2; capacity
                    * =2; free(queue);
queue= newQueue;
}
```

Program: queueFull

MULTIPLE STACKS AND QUEUES

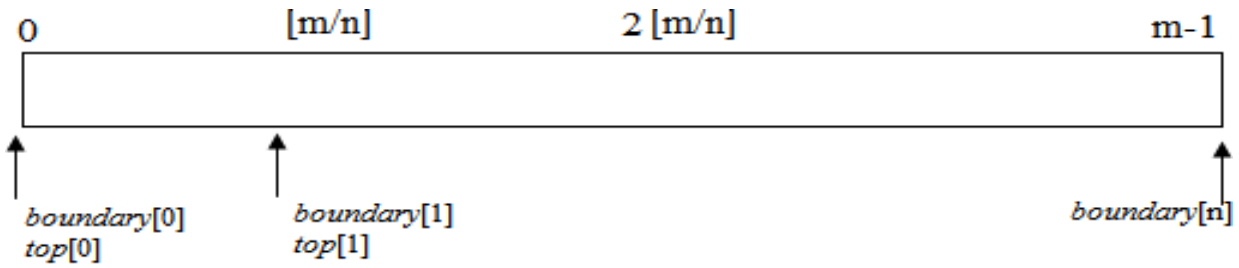
- In multiple stacks, we examine only **sequential mappings** of stacks into an array. The array is one dimensional which is **memory[MEMORY_SIZE]**. Assume ***n*** stacks are needed, and then divide the available memory into *n* segments. The array is divided in proportion if the expected sizes of the various stacks are known. Otherwise, divide the memory into equal segments.
- Assume that *i* refers to the stack number of one of the *n* stacks. To establish this stack, create indices for both the **bottom** and **top** positions of this stack. ***boundary*[i]** points to the position immediately to the left of the bottom element of stack *i,* ***top[i]*** points to the top element. Stack *i* is empty iff *boundary[i]=top[i]*.

The declarations are:

```
#define MEMORY_SIZE 100      /* size of memory */
#define MAX_STACKS 10        /* max number of stacks plus 1 */
element                      /* global memory declaration */
memory[MEMORY_SIZE]; int
top [MAX_STACKS];
int boundary
[MAX_STACKS] ; int n;        /*number of stacks entered by the user */
```

To divide the array into roughly equal segments

```
top[0] = boundary[0] = -1; for (j=
        1;j<n; j++)
                top[j] = boundary[j] = (MEMORY_SIZE / n) *
        j; boundary[n] = MEMORY_SIZE - 1;
```



All stacks are empty and divided into roughly equal segments

Figure: Initial configuration for *n* stacks in *memory [m]*.

In the figure, **n** is the number of stacks entered by the user, n < MAX_STACKS, and m =MEMORY_SIZE. Stack **i** grow from **boundary[i] + 1** to **boundary [i + 1]** before it is

full. A boundary for the last stack is needed, so set **boundary [n]** to **MEMORY_SIZE-1**.

Implementation of the add operation

```
void push(int i, element
item)
{               /* add an item to the ith
        stack */ if (top[i] == boundary[i+l])
                stackFull(i);
        memory[++top[i]
        ] = item;
}
```

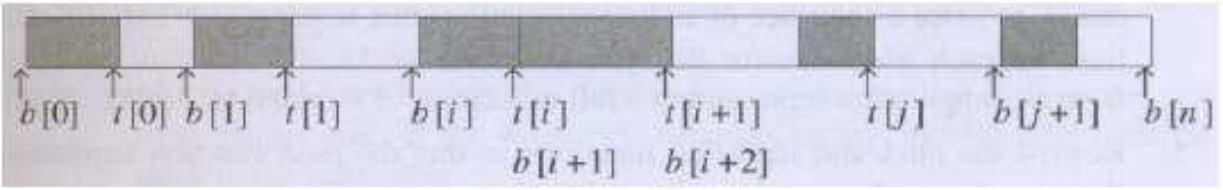
Program: Add an item to the ith stack

Implementation of the delete operation

```
element pop(int i)
        {       /* remove top element from the ith stack */ if
            (top[i] == boundary[i])
                    return stackEmpty(i);
                return memory[top[i]--];
}
```

The top[i] == boundary[i+1] condition in push implies only that a particular stack ran out of memory, not that the entire memory is full. But still there may be a lot of unused space between other stacks in array memory as shown in Figure.

Therefore, create an error recovery function called stackFull , which determines if there is any free space in memory. If there is space available, it should shift the stacks so that space is allocated to the full stack.



b boundary , t=top

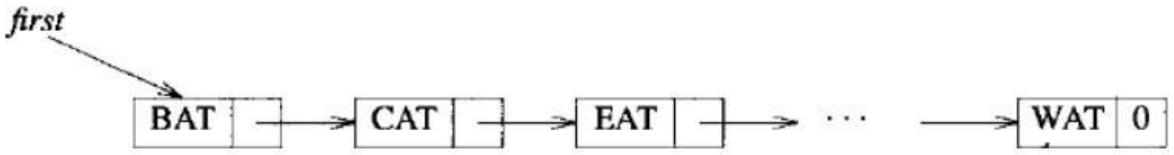
LINKED LIST

DEFINITION

A linked list, or one-way list, is a linear collection of data elements, called nodes, where the linear order is given by means of pointers. That is, each node is divided into two parts:

- The first part contains the information of the element, and
- The second part, called the ***link field*** **or** ***nextpointer*** field, contains the address of the next node in the list.

A linked list is a dynamic data structure where each element (called a node) is made up of two items - the data and a reference (or pointer) which points to the next node. A linked list is a collection of nodes where each node is connected to the next node through a pointer.



Usual way to draw a linked list

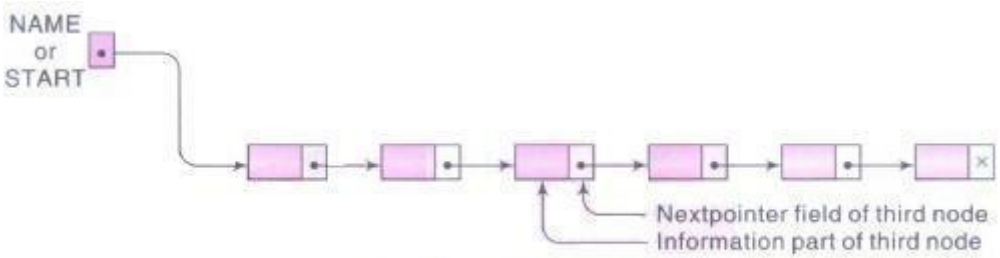


Fig: Linked list with 6 nodes

In the above figure each node is pictured with two parts.

➢ The left part represents the information part of the node, which may contain an entire record of data items.

➢ The right part represents the link field of the node

➢ An arrow drawn from a node to the next node in the list.

➢ The pointer of the last node contains a special value, called the NULL.

A pointer variable called first which contains the address of the first node. A special case is the list that has no nodes; such a list is called the null list or empty list and is denoted by the null pointer in the variable first.

REPRESENTATION OF LINKED LISTS IN MEMORY

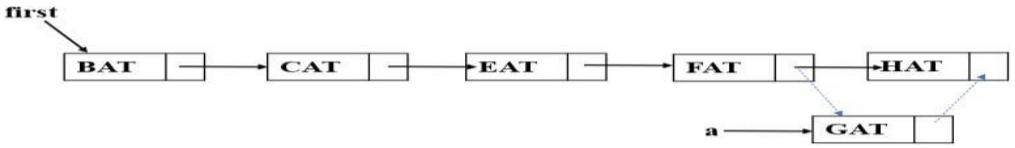
Let LIST be a linked list. Then LIST will be maintained in memory as follows.

1. LIST requires two linear arrays such as DATA and LINK-such that DATA[K] and LINK[K] contains the information part and the nextpointer field of a node of LIST.
2. LIST also requires a variable name such as START which contains the location of the beginning of the list, and a nextpointer sentinel denoted by NULL-which indicates the end of the list.
3. The subscripts of the arrays DATA and LINK will be positive, so choose NULL = 0, unless otherwise stated.

The following examples of linked lists indicate that the nodes of a list need not occupy adjacent elements in the arrays DATA and LINK, and that more than one list may be maintained in the same linear arrays DATA and LINK. However, each list must have its own pointer variable giving the location of its first node.

	DATA	LINK
1	HAT	15
2		
3	CAT	4
4	EAT	9
5	GAT	1
6		
7	WAT	0
8	BAT	3
9	FAT	5
10		
11	VAT	7

Insert GAT to data[5]



Insert node GAT into list

REPRESENTING CHAIN IN C

The following capabilities are needed to make linked representation

1. A mechanism for defining a node's structure, that is, the field it contains. So self-referential structures can be used
2. A way to create new nodes, so MALLOC functions can do this operation
3. A way to remove nodes that no longer needed. The FREE function handles this operation.

1. Defining a node structure

```
typedef struct listNode *listPointer
  typedef struct
{
          char data[4];
          listPointer list;
} listNode;
```

Create a New Empty list

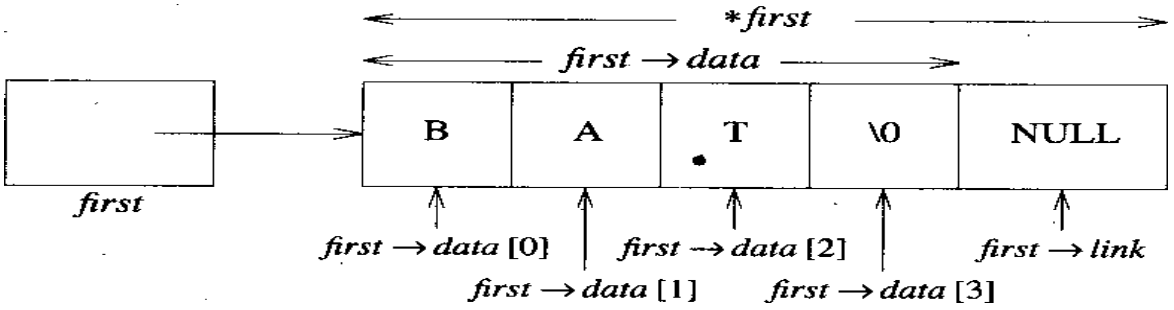
```
listPointer first = NULL
```

To create a New Node

```
MALLOC (first, sizeof(*first));
```

To place the data into NODE

```
strcpy(first→ data,"BAT");
first→ link = NULL
```



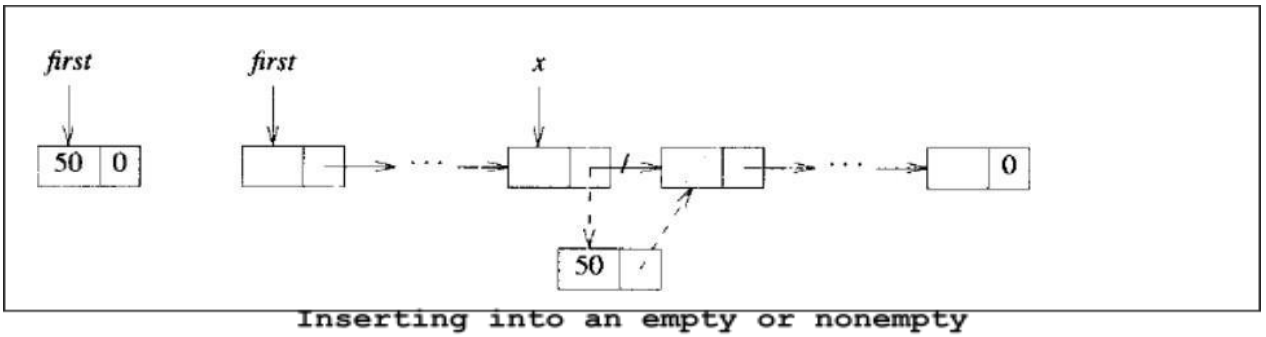
2. Two-node linked list:

```
listPointer
create2()
{       /*create a linked list with two
        nodes*/ listPointer first,second;
        MALLOC(first,sizeof(*first));
        MALLOC(second,sizeof(*second));
        Second->link=NULL
        ;
        Second->data=20;
        First->data=10;
        First->link=secon
        d;
        Return first
}
```

3. List insertion:

Program: create two node list

```
void insert(listPointer *first, listPointer x)
{
listPointer temp;
      malloc(temp,sizeof(*temp);
      temp->data=50;
if(*first)
{
temp->link; x->link;
}
else
{
temp->link;
*first->temp;
}
}
```



Inserting into an empty or nonempty list

4. Deletion from the list:

- Deletion depends on the location of the nodes.
- We have three pointers:
  - **first** points to start of the list,
  - **x** points to the node that we have to delete
  - **trail** points to the node the precedes to x.

```
void delete(listPointer *first, listPointer trail, listPointer x)
{
      if(trail)
            trail->link=x->link;
      else
            *first=(*first)->link;
      free(x);
}
```

Deletion from list

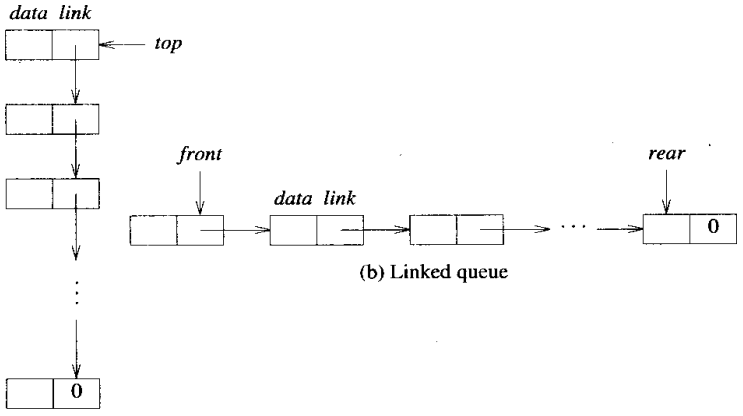
5. Printing out a list

```
void printList(listPointer first)
{
      printf("The list contains");
      for(;first;first=first->link)
            Printf("%4d",first->data);
      printf("\n");
}
```

Printing list

LINKED STACKS AND QUEUES

The below figure shows stacks and queues using linked list. Nodes can easily add or delete a node from the top of the stack. Nodes can easily add a node to the rear of the queue and add or delete a node at the front



(b) Linked queue

(a) Linked stack

Linked Stack

The representation of n ≤ MAX_STACKS

```
#define MAX_STACKS 10  /* maximum number of stacks */
typedef struct {
                    int key;
                    /* other fields */
             }element;

typedef struct stack *stackPointer;
typedef struct {
                    element data; stackPointer
                    link;
             } stack;
stackPointer top[MAX_STACKS];
```

The initial condition for the stacks is:

```
top[i] = NULL, 0 ≤ i < MAX_STACKS
```

The boundary condition is:

```
top [i] = NULL iff the ith stack is empty
```

Functions push and pop add and delete items to/from a stack.

```
void push(int i, element item)
      {/* add item to the ith stack */
            stackPointer temp;
            MALLOC(temp, sizeof(*temp));
            temp→data = item;
            temp→link = top[i];
            top[i] = temp;
      }
```

Add to a linked stack

Function push creates a new node, temp, and places item in the data field and top in the link field. The variable top is then changed to point to temp. A typical function call to add an element to the ith stack would be push (i,item).

```
element pop(int
i)
      {     /* remove top element from the ith stack */
            stackPointer temp = top[i];
            element item;
            if (! temp)
                  return stackEmpty();

            item = temp→data;
            top[i] =
            temp→link; free
            (temp) ;
            return item;
      }
```

Delete from a linked stack

Function pop returns the top element and changes top to point to the address contained in its link field. The removed node is then returned to system memory. A typical function call to delete an element from the ith stack would be item = pop (i);

Linked Queue

The representation of m ≤ MAX_QUEUES queues,

```
#define MAX-QUEUES 10 /* maximum number of queues */
Typedef struct queue *queuePointer;
typedef struct {
                element data;
                queuePointer link;
 } queue;
queuePointer front[MAX_QUEUES], rear[MAX_QUEUES];
```

The initial condition for the queues is:

front[i] = NULL, 0 ≤ i < MAX_QUEUES

The boundary condition is:

front[i] = NULL iff the ith queue is empty

Functions addq and deleteq implement the add and delete operations for multiple queues.

```
void addq(i,
item)
{                   /* add item to the rear of queue i
                 */ queuePointer temp;
              MALLOC(temp, sizeof(*temp));
              temp→data      =
              item; temp→link
              =    NULL;    if
              (front[i])
                    rear[i] →link = temp;
              else
                    front[i] =
     }        temp; rear[i] =
              temp;
```

Program: Add to the rear of a linked queue

Function addq is more complex than push because we must check for an empty queue. If the queue is empty, then change front to point to the new node; otherwise change rear's link field to point to the new node. In either case, we then change rear to point to the new node.

```
element deleteq(int i)
      {/* delete an element from queue i */
           queuePointer temp =
            front[i]; element item;
            if (! temp)
                  return
            queueEmpty(); item =
            temp→data; front[i]=
            temp→link;
            free (temp) ;
            return item;
       }
```

Program: Delete from the front of a linked queue

Function deleteq is similar to pop since nodes are removing that is currently at the start of the list. Typical function calls would be addq (i, item); and item = deleteq (i);

APPLICATIONS OF LINKED LISTS – POLYNOMIALS

1. Representation of the polynomial:

$$A(x) = a_{m-1}x^{e_{m-1}} + \cdots + a_0 x^{e_0}$$

where the ai are nonzero coefficients and the ei are nonnegative integer exponents such that em-l > em-2 > ... > e1 > e0 ≥ 0.

Present each term as a node containing coefficient and exponent fields, as well as a pointer tothe next term.

Assuming that the coefficients are integers, the type declarations are:

```
typedef struct polyNode
*polyPointer; typedef struct {
                int coef; int expon;
                    polyPointer link;
              } polyNode;
polyPointer a,b;
```

We draw polynomial nodes as:

coef	expon	link

Figure shows how we would store the polynomials

$$a = 3x^{14} + 2x^8 + 1$$

and

$$b = 8x^{14} - 3x^{10} + 10x^6$$

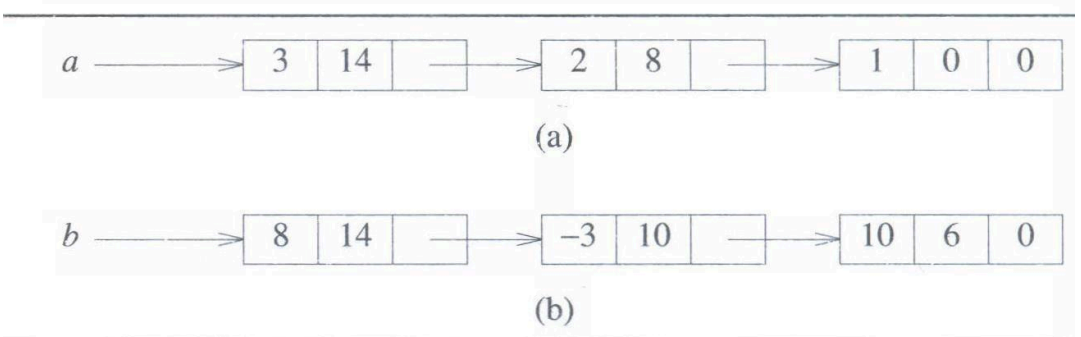


Figure: Representation of $3x^{14}+2x^8+1$ and $8x^{14}-3x^{10}+10x^6$

2. Adding Polynomials

To add two polynomials, examine their terms starting at the nodes pointed to by ***a*** and ***b***.

- If the exponents of the two terms are equal, then add the two coefficients and create a new term for the result, and also move the pointers to the next nodes in ***a*** and ***b***.
- If the exponent of the current term in ***a*** is less than the exponent of the current term in***b***, then create a duplicate term of ***b***, attach this term to the result, called ***c***, and advance the pointer to the next term in ***b***.
- If the exponent of the current term in ***b*** is less than the exponent of the current term in***a***, then create a duplicate term of ***a***, attach this term to the result, called ***c***, and advance the pointer to the next term in ***a***

Below figure illustrates this process for the polynomials addition.

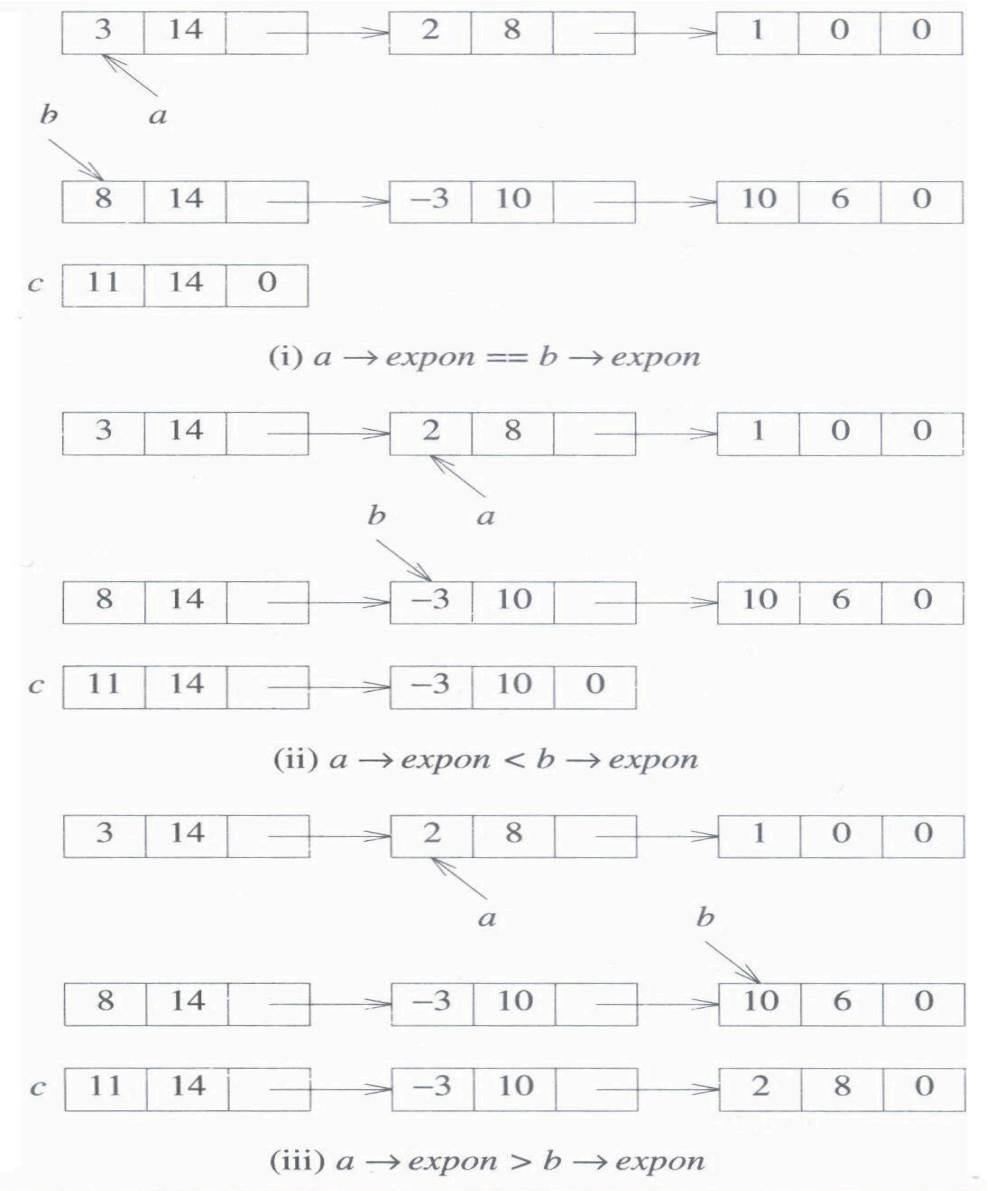


Figure: Generating the first three terms of $c = a + b$

The complete addition algorithm is specified by padd( )

```
polyPointer padd(polyPointer a, polyPointer b)
{/* return a polynomial which is the sum of a and b */
   polyPointer c, rear, temp;
   int sum;
   MALLOC(rear, sizeof(*rear));
   c = rear;
   while (a && b)
      switch (COMPARE(a→expon,b→expon)) {
         case -1: /* a→expon < b→expon */
              attach(b→coef,b→expon,&rear);
              b = b→link;
              break;
         case 0: /* a→expon = b→expon */
              sum = a→coef + b→coef;
              if (sum) attach(sum,a→expon,&rear);
              a = a→link;  b = b→link; break;
         case 1: /* a→expon > b→expon */
              attach(a→coef,a→expon,&rear);
              a = a→link;
      }
   /* copy rest of list a and then list b */
   for (; a; a = a→link) attach(a→coef,a→expon,&rear);
   for (; b; b = b→link) attach(b→coef,b→expon,&rear);
   rear→link = NULL;
   /* delete extra initial node */
   temp = c; c = c→link;  free(temp);
   return c;
}
```

Program : Add two polynomials

```
void attach(float coefficient, int exponent,
            polyPointer *ptr)
{/* create a new node with coef = coefficient and expon =
    exponent, attach it to the node pointed to by ptr.
    ptr is updated to point to this new node */
   polyPointer temp;
   MALLOC(temp, sizeof(*temp));
   temp→coef = coefficient;
   temp→expon = exponent;
   (*ptr)→link = temp;
   *ptr = temp;
}
```

Program : Attach a node to the end of a list

Analysis of padd:

To determine the computing time of padd, first determine which operations contribute to the cost. For this algorithm, there are three cost measures:

(l) Coefficient additions

(2) Exponent comparisons

(3) Creation of new nodes for c

The maximum number of executions of any statement in padd is bounded above by m + n. Therefore, the computing time is O(m+n). This means that if we implement and run the algorithm on a computer, the time it takes will be C1m + C2n + C3, where C1, C2, C3 are constants. Since any algorithm that adds two polynomials must look at each nonzero term at least once, padd is optimal to within a constant factor.

3. Erasing a Polynomial

```
Void erase(polyPointer *ptr)
{
      polyPointer temp;
      while(*ptr)
      {
            Temp=*ptr;
            *ptr=(*ptr)->link;
            Free(temp)
      }
}
```

Program:erasing polynomial

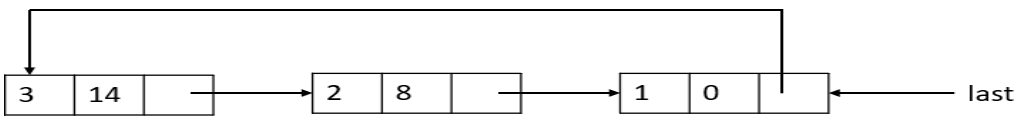
4. Circular representation of polynomials

Circular linked list are one they of liner linked list. In which the link fields of last node of the list contains the address of the first node of the list instead of contains a null pointer. Advantages:- Circular list are frequency used instead of ordinary linked list because in circular list all nodes contain a valid address.

The important feature of circular list is as follows.

(1) In a circular list every node is accessible from a given node.

(2) Certain operations like concatenation and splitting becomes more efficient in circular list.



Circular representation of $3x^{14}+2x^{8}+1$

- We can free the nodes that are no longer used and can reuse the nodes later by maintain a list called freed. When new node is needed we examine the this list. If the list is not empty then we may use one of the nodes. Only when list is empty we need to create a node using malloc.
- Let avial be a variable of type polyPointer that points to first node in our list of freed nodes. We call this list as available space list or avail list.
- Initially set avail to NULL.instead of using malloc or free we use getNode and retNode.
- Erase circular list in a fxed amount of time independent of number of nodes in list using cerase

```
polyPointer getNode(void)
{
polyPointer node;
      if(avail)
{
node=avail;
           avail=avail->link;
}
else
           malloc(node, sizeof(*node));
      return node
}
```

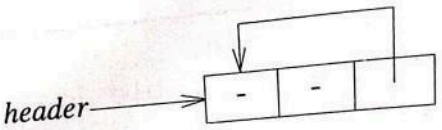
Program:getNode function

```
void retNode(polyPointer node)
 {
      node->link=avail;
       avail=node;
 }
```

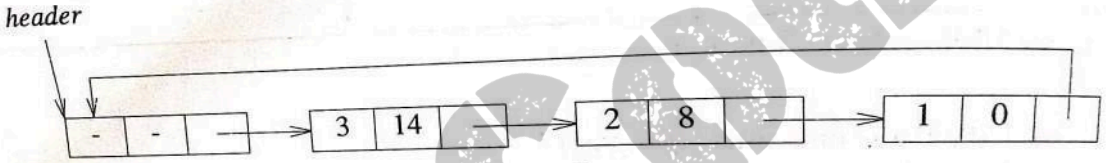
Program: retNode function

```
void cerase(polyPointer *ptr)
{
if(*ptr)
{
temp=(*ptr)->link;
            (*ptr)->link=avai
            l; avail=temp;
*ptr=NULL;
}
}
```

Program:Erasing a circular list



(a) Zero polynomial



(b) $3x^{14} + 2x^8 + 1$

```
polyPointer cpadd(polyPointer a, polyPointer b)
{/* polynomials a and b are singly linked circular lists
   with a header node. Return a polynomial which is
   the sum of a and b */
  polyPointer startA, c, lastC;
  int sum, done = FALSE;
  startA = a;              /* record start of a */
  a = a→link;              /* skip header node for a and b*/
  b = b→link;
  c = getNode();           /* get a header node for sum */
  c→expon = -1; lastC = c;
  do {
    switch (COMPARE(a→expon, b→expon)) {
      case -1: /* a→expon < b→expon */
               attach(b→coef,b→expon,&lastC);
               b = b→link;
               break;
      case 0:  /* a→expon = b→expon */
               if (startA == a)  done = TRUE;
               else {
                  sum = a→coef + b→coef;
                  if (sum) attach(sum,a→expon,&lastC);
                  a = a→link; b = b→link;
               }
               break;
      case 1:  /* a→expon > b→expon */
               attach(a→coef,a→expon,&lastC);
               a = a→link;
    }
  } while (!done);
  lastC→link = c;
  return c;
}
```