Rashtreeya Sikshana Samithi Trust

**RV Institute of Technology and Management®**

(Affiliated to VTU, Belagavi)

**JP Nagar, Bengaluru - 560076**

**Department Computer Science and Engineering**

**&**

**Department Information Science and Engineering**

# Course Name : DATA STRUCTURES AND APPLICATIONS

**Course Code : BCS304**

**III Semester**

**2022 Scheme**

## Module-2

**QUEUES:** Queues, Circular Queues, Using Dynamic Arrays, Multiple Stacks and Queues.
**LINKKED LISTS:** Singly Linked List, Lists and Chains, Representing Chains in C,Linked Stacks and Queues, Polynomial

## QUEUES

### 2.1 Queue 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.

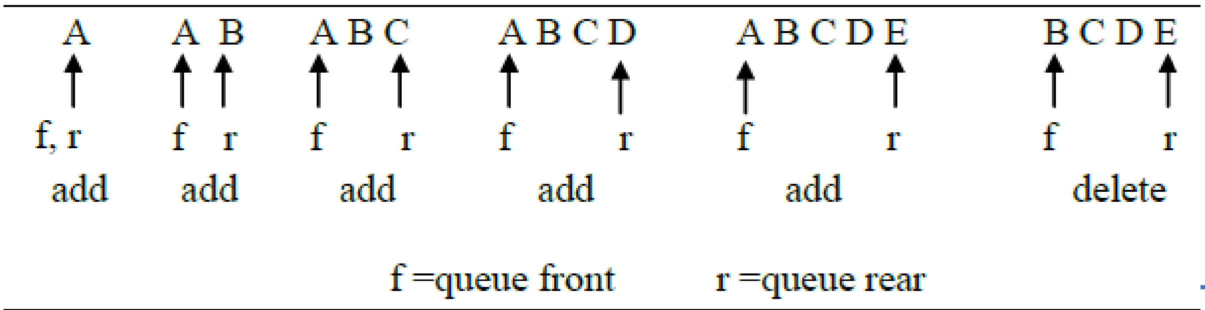
If the elements are inserted A, B, C, D and E in this order, then A is the first element deleted from the queue. Since the first element inserted into a queue is the first element removed, queues are also known as First-In-First-Out (FIFO) lists.

#### 2.1.1 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 2.1, indicates the way elements will be deleted from the queue and the way new elementswill 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; thiscan be implemented by the assignment REAR := REAR + 1



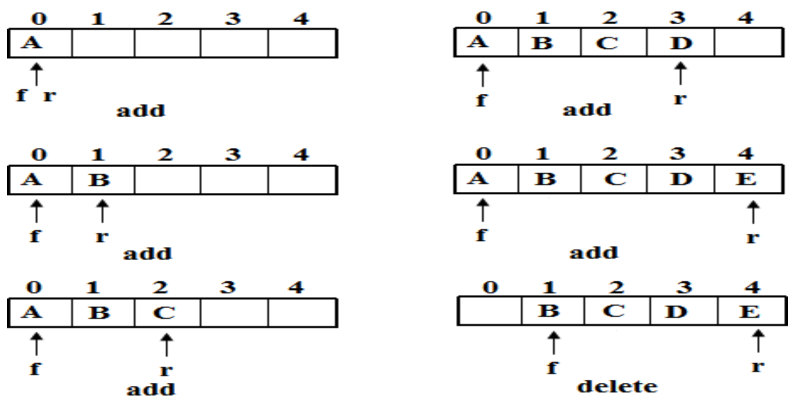


Figure 2.1 Queue Operations

## 2.1.2 Queue Operations

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 inaddq( ) and front in delete( ). The function calls would be addq (item); and item =delete( );

**4.** addq(item)

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

**Program: Add to a queue**

**5.** deleteq( )

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

**Program: Delete from a queue**

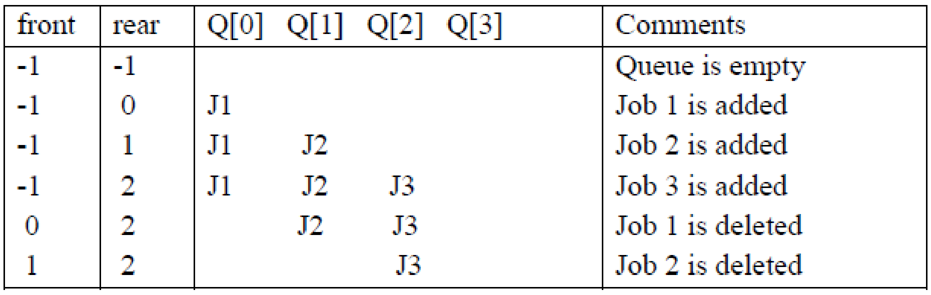
**6.** queueFull( )

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 2.2 illustrates how an operating system process jobs using a sequential representation for its queue.

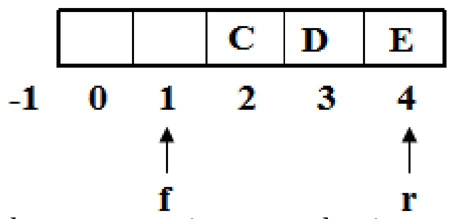


| front | rear | Q[0] | Q[1] | Q[2] | Q[3] | Comments |
|---|---|---|---|---|---|---|
| -1 | -1 | | | | | Queue is empty |
| -1 | 0 | J1 | | | | Job 1 is added |
| -1 | 1 | J1 | J2 | | | Job 2 is added |
| -1 | 2 | J1 | J2 | J3 | | Job 3 is added |
| 0 | 2 | | J2 | J3 | | Job 1 is deleted |
| 1 | 2 |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

**Figure 2.2: Insertion and deletion from a sequential queue**

## Drawback of Queue

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

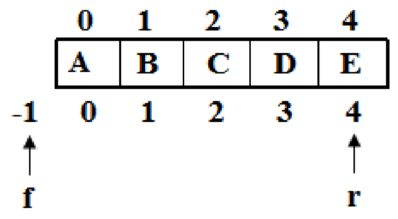


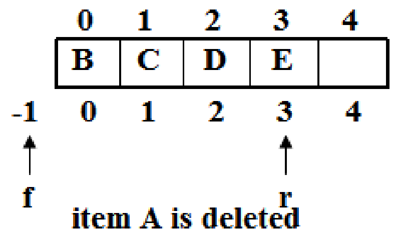
In this above situation, when we try to insert another item, which shows that the queue is full. This means that the rear index equals to MAX_QUEUE_SIZE -1. But even if the space is available at the front end, rear insertion cannot be done.

## Overcome of Drawback using different methods

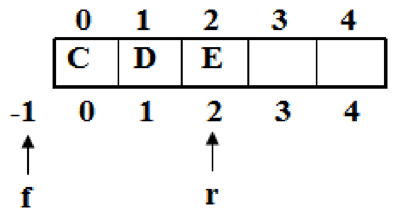
Method 1:

- 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)





item A is deleted



item B is deleted

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 counter clockwise from the location of the front element in the queue. The convention for rear is unchanged.

## 2.2 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 counter clockwise from the location of the front element in the queue. The convention for rear is unchanged.

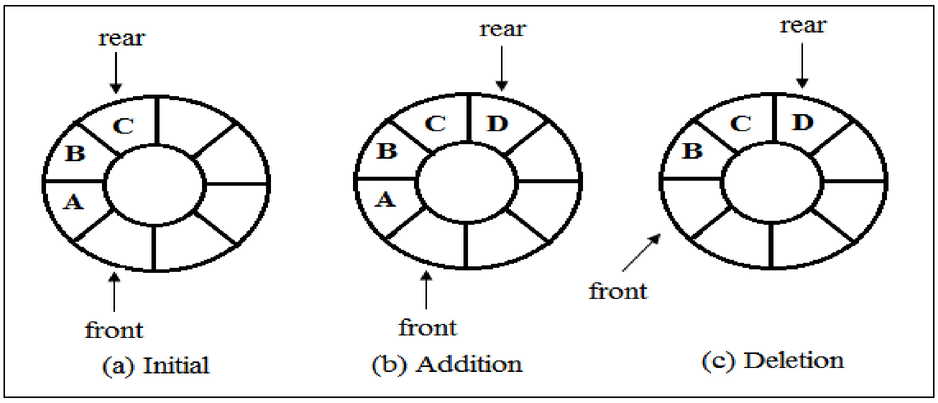


**Figure 2.3: Circular Queue**

**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.

(rear +1) % MAX_QUEUE_SIZE

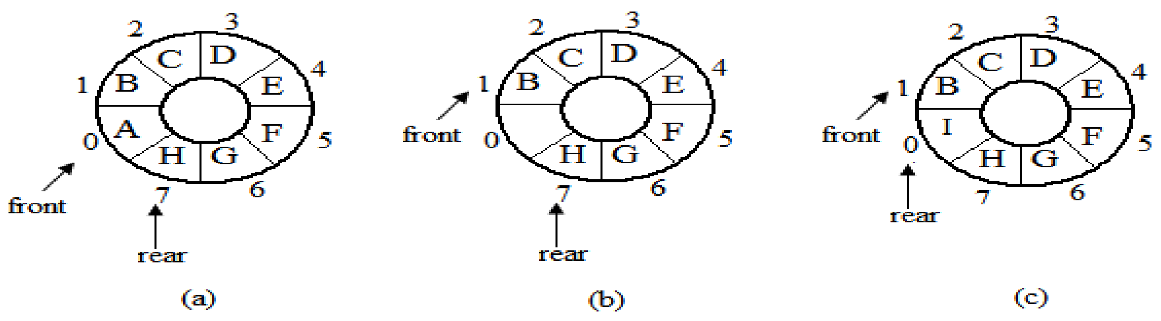


**Figure 2.4: Circular Queue operations**

**Program: Add to a circular queue**

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

```
element deleteq()
{ /* remove front element from the queue */ element item;
          if (front == rear)
                    return queueEmpty(  );          /* return an error key */
```

```
                front = (font+1)% MAX_QUEUE_SIZE;
        return queue[front];
}
```

**Program: Delete from a circular queue**

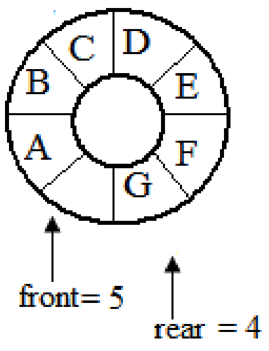
Note:

- When queue becomes empty, then front =rear. When the queue becomes full and front=rear. It is difficult to distinguish between an empty and a full queue.
- To avoid the resulting confusion, increase the capacity of a queue just before it becomes full.

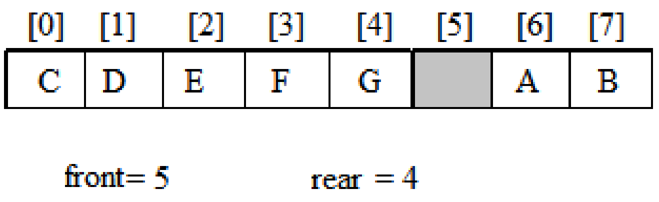
### 2.2.1 Circular Queues Using Dynamic Arrays

- A dynamically allocated array 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. As with dynamically allocated stacks, array doubling is used.

Consider the full queue of figure2.5 (a). This figure shows a queue with seven elements in an arraywhose capacity is 8. A circular queue is flatten out the array as in Figure 2.5 (b).

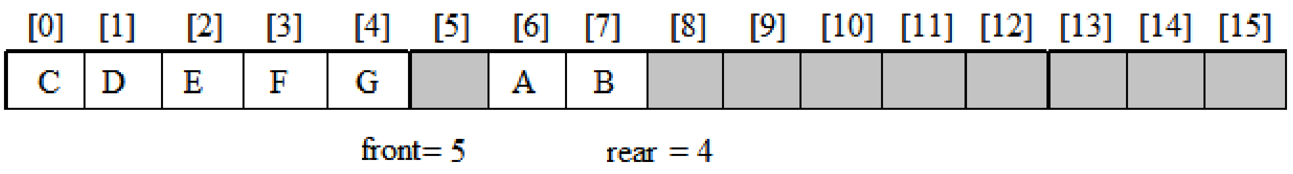


(a) A full circular queue



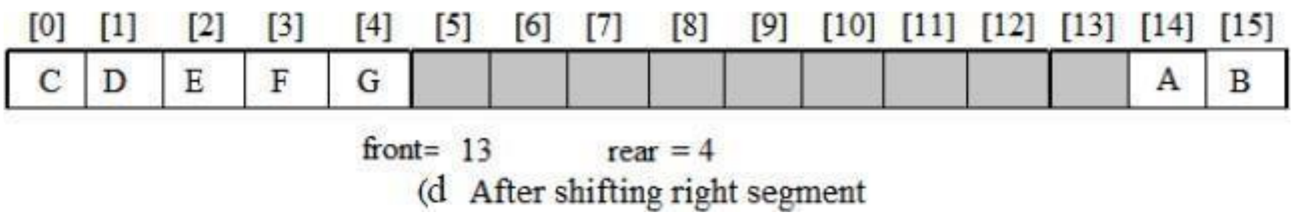
(b) Flattened view of circular full queue

Figure 2.5 (c) shows the array after array doubling by realloc



(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 2.5 (d)



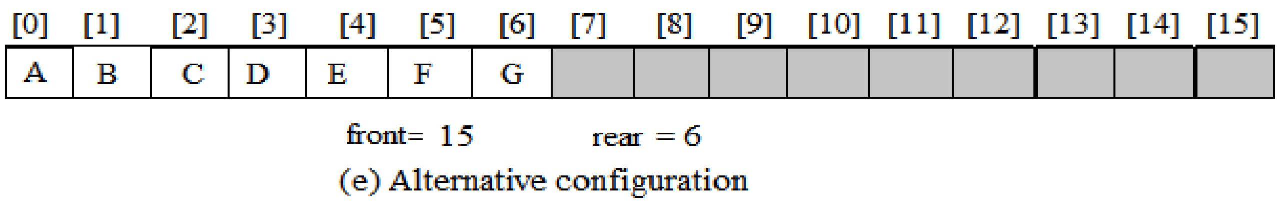


**Figure 2.5: Circular Queue using Dynamic Arrays**

To obtain the configuration as shown in figure 2.5 (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.

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 + 1 ) % capacity;
        if ( start < 2)    /* no wrap around */
```

```
          copy( queue+start, queue+start+capacity-1,newQueue);
     else
     {    /* queue wrap around */
          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**

## 2.3 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 memory[MEMORY_SIZE];        /* global memory declaration */ 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;
```

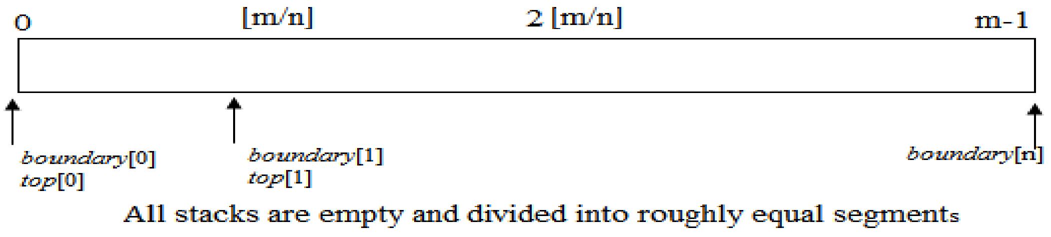


**Figure 2.6: Initial configuration for n stacks in memory [m].**

In the figure 2.6, 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**

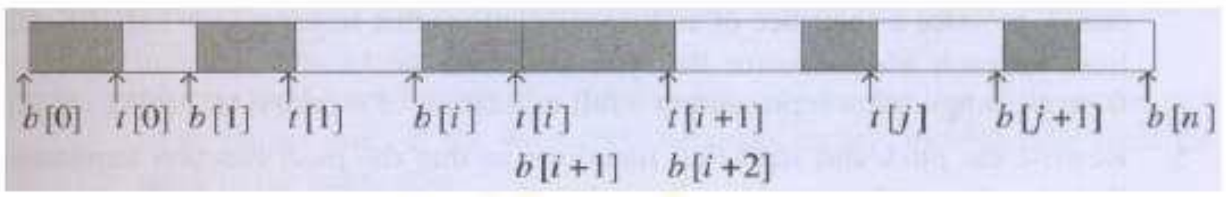
**Implementation of the delete operation**

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

**Program: Delete an item from the ith stack**

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

Method to design stackFull

- Determine the least, j, i < j < n, such that there is free space between stacks j and j+1. Thatis, top[j] < boundary[j+l]. If there is a j, then move stacks i+l,i+2, .., j one position to the right (treating memory[O] as leftmost and memory[MEMORY_SIZE - 1] as rightmost). This creates a space between stacks i and i+1.
- If there is no j as in (1), then look to the left of stack i. Find the largest j such that 0 ≤ j≤ i and there is space between stacks j and j+ 1 ie, top[j] < boundary[j+l]. If there is a j, thenmove stacks j+l, j+2, ... , i one space to the left. This also creates space between stacks i andi+1.
- If there is no j satisfying either condition (1) or condition (2), then all MEMORY_SIZE spaces of memory are utilized and there is no free space. In this case stackFull terminates with an error message.

## 2.4 LINKED LISTS

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.

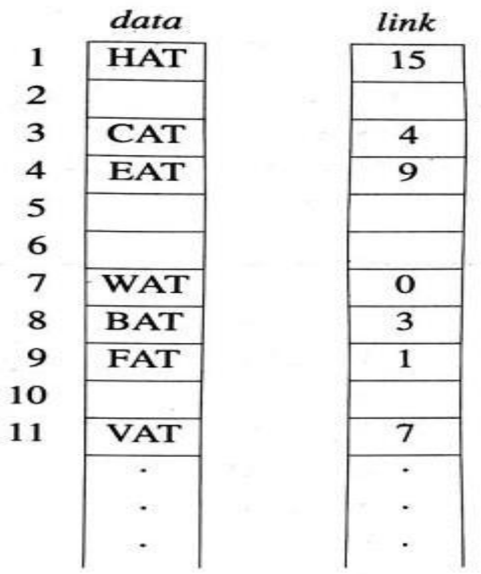


**Figure 2.7(a) Non Sequential List-representation**

Figure 2.7(a) shows how some of the elements in list of three words may be represented in memory by using pointers. The elements of the list are stored in a one-dimensional array called *data,* but elements are no longer occur in a sequential order.

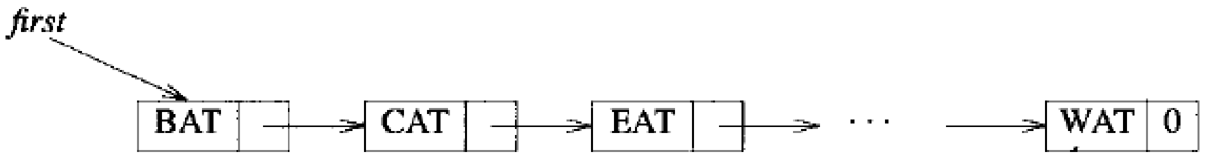


**Figure 2.7(b) Usual way to draw a linked list**

Figure 2.7(b) represents the ordered sequence of nodes with links being represented by arrows. The linkedstructures of figures 2.7 a and b are called singly links or chains. In singly linked list, each node has exactly one pointer field.

A chain is a singly linked list that is comprised of zero or more nodes. When the number of nodes is zero,the chain is empty.The nodes of non eempty chain are ordered so that the first node links to the second node, the second node links to third and so on. The last node of a chain has 0 link.

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 recordof 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***.

## 2.5 REPRESENTING CHAINS IN C

The following capabilities are required for linked list representation:

1. A mechanism for defining a node's structure, that is, the fields it contains. We use self referential structure.
2. A way to create new nodes when we need them. The *malloc* macrons handles this operation.
3. A way to remove nodes that we no longer need. The *free* function handles this operation.

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

*listPointer first = NULL;*

This statement indicates that we have a new list called *first*. Remember that list contain the address of the start of the lst. Since the new list is initially empty, its starting address is zero. Therefore we use reserved word NULL to signify this condition. We can use an IS_EMPTY macro to test for an empty list.

*#define IS_EMPTY(first) (!first)*

*To create a New Node*

MALLOC (first, sizeof(*first));

*To place the data into Node*

strcpy(first→data,"BAT");

first→ link = NULL

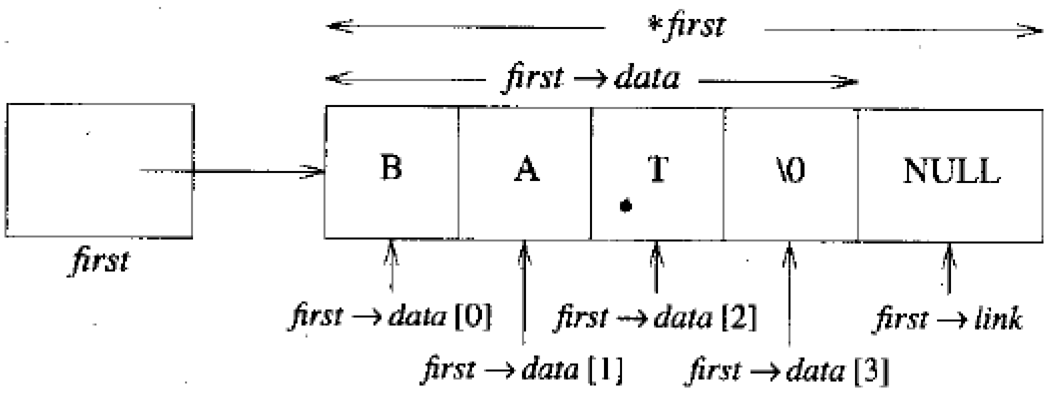
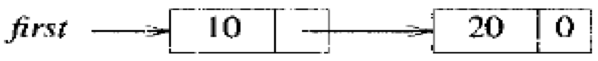


**Figure: 2.8 Referencing the fields of a node**

A linked lsit with two nodes is created by using create2 function as shown below.

**<u>Creating two node lists:</u>**

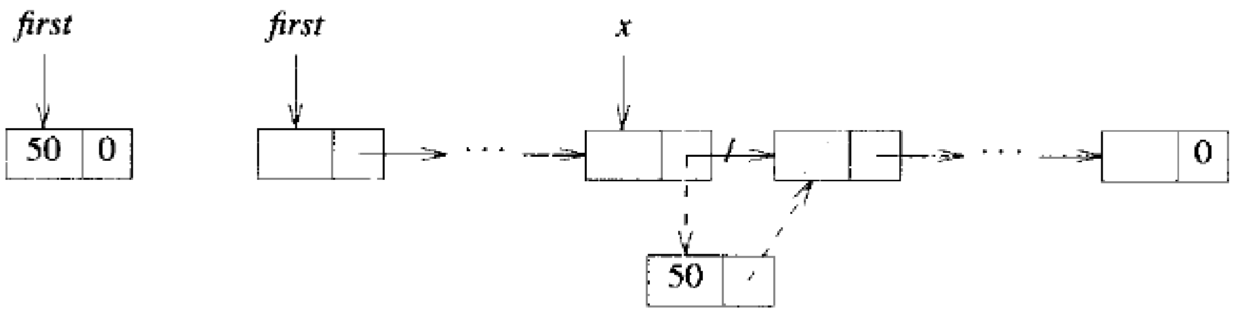
```
listPointer create2()
{/* create a linked list with two nodes */
   listPointer first, second;
   MALLOC(first, sizeof(*first));
   MALLOC(second, sizeof(*second));
   second→link = NULL;
   second→data = 20;
   first→data = 10;
   first→link = second;
   return first;
}
```



A two-node list

**Insertion into front of list**

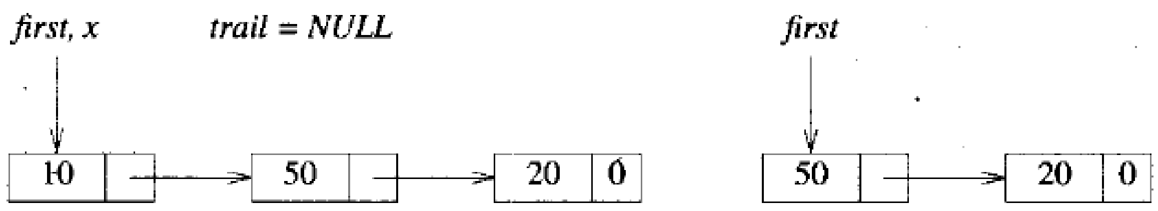
```
void insert(listPointer *first, listPointer x)
{/* insert a new node with data = 50 into the chain
    first after node x */
   listPointer temp;
   MALLOC(temp, sizeof(*temp));
   temp→data = 50;
   if (*first) {
      temp→link = x→link;
      x→link = temp;
   }
   else {
      temp→link = NULL;
      *first = temp;
   }
}
```



(a) (b)

Inserting into an empty and nonempty list

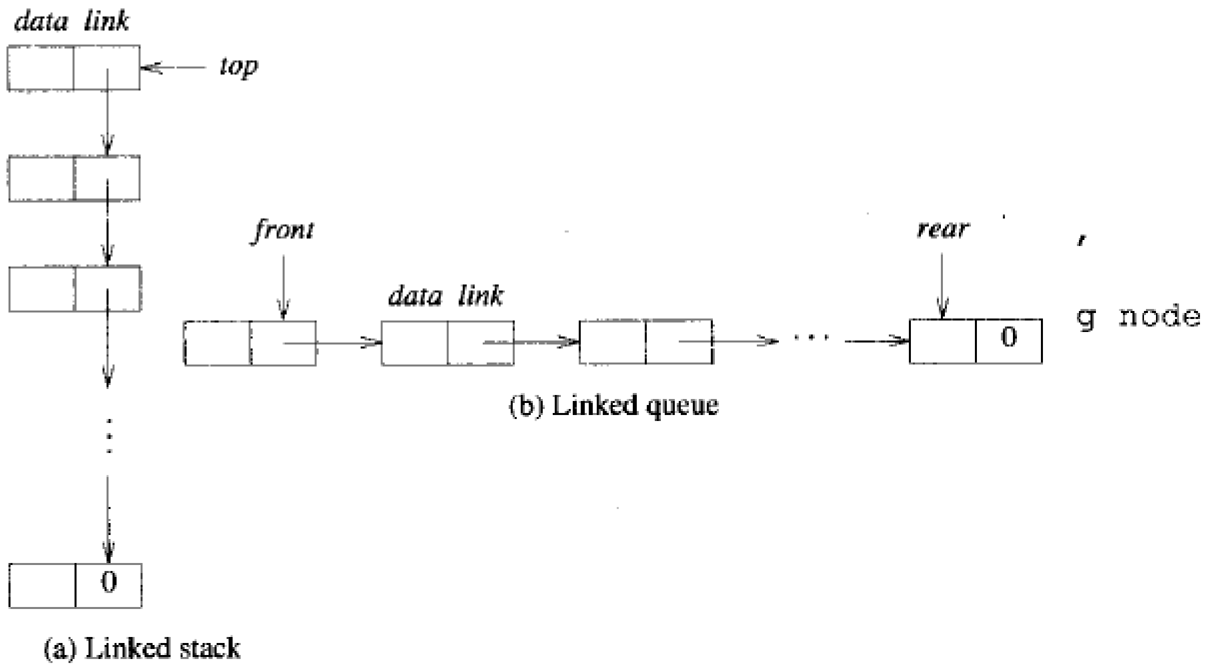
**Deletion from the list:**



(a) Before deletion (b) After deletion

List before and after the function call *delete(&first, NULL, first);*

## 2.6 LINKED STACKS



Linked stack and queue

The above 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

If we wish to represent $n \le MAX\_STACKS$ stacks simultaneously, we begin with the declarations:

```
#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];
```

We assume that the initial condition for the stacks is:

$$top[i] = NULL,\ 0 \le i < MAX\_STACKS$$

and the boundary condition is:

*top* [*i* ] = *NULL* iff the *i*th stack is empty

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).

```
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 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 i$^{th}$ stack would be item = pop (i);

```
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

## 2.7 LINKED QUEUES

To represent $m \leq MAX\_QUEUES$ queues simultaneously, we begin with the declarations:

```
#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];
```

We assume that the initial condition for the queues is:

$$front[i] = NULL, 0 \leq i < MAX\_QUEUES$$

and the boundary condition is:

$$front[i] = \text{NULL iff the } i\text{th queue is empty}$$

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.

```
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;
}
```

Add to the rear 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);

```
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;
}
```

Delete from the front of a linked queue

## 2.7 POLYNOMIALS

### 2.7.1 Representation of the polynomial:

$$A(x) = a_{m-1}x^{e_{m-1}} + \cdots + a_0x^{e_0}$$

where the $a_i$ are nonzero coefficients and the $e_i$ are nonnegative integer exponents such that $e_{m-1} > e_{m-2} > \cdots > e_1 > e_0 \geq 0$. We represent each term as a node containing coefficient and exponent fields, as well as a pointer to the 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;
```

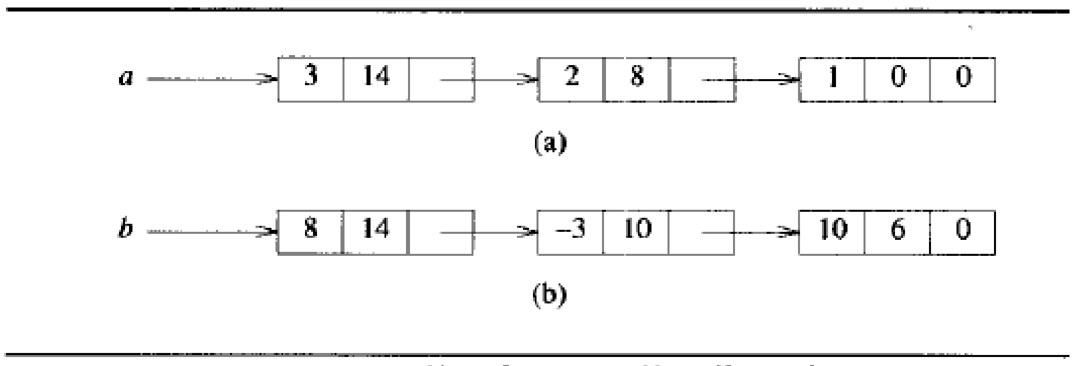
We draw *polyNodes* as:

| coef | expon | link |
|---|---|---|

and

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

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



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

**Figure 2.9 Representation of polynomial**

### 2.7.2 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 polynomial s addition.

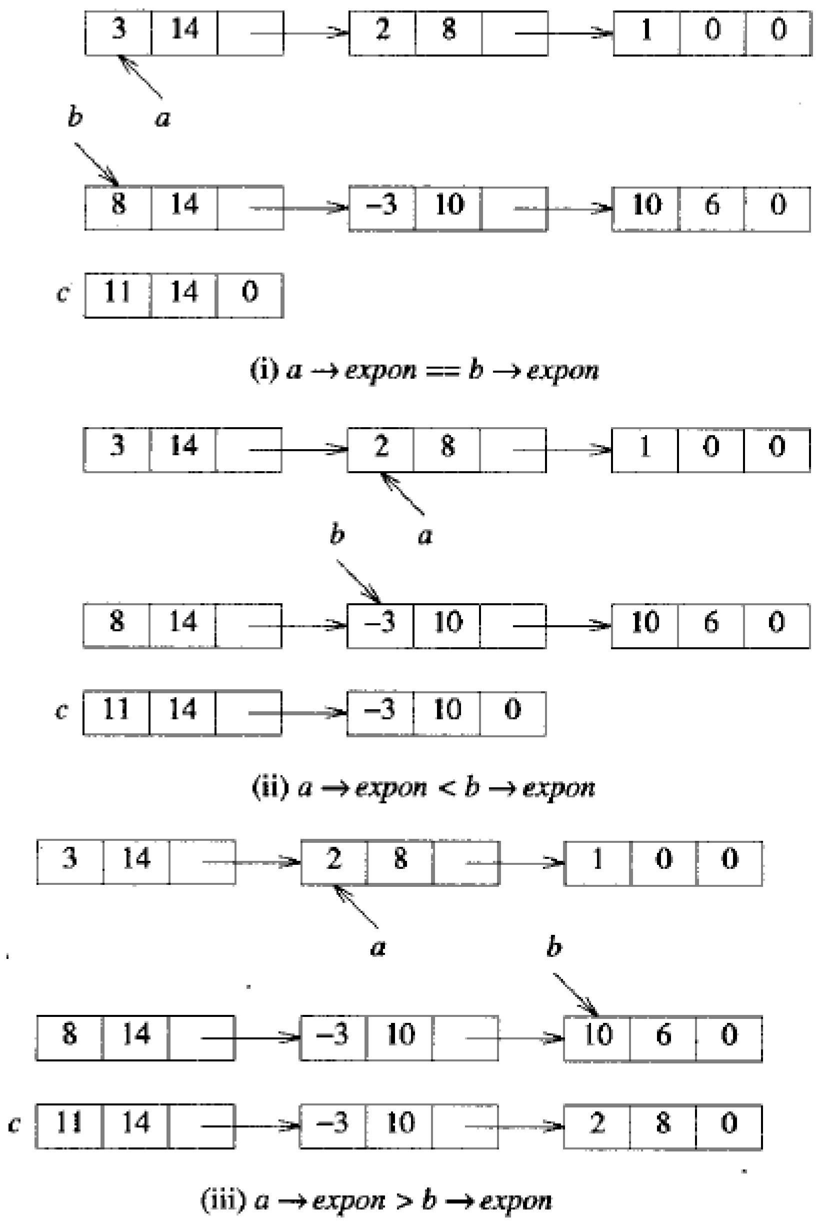


**Figure2.10 Generating the first three terms of c=a+b**

```
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;
}
```

Add two polynomials

```
void attach(float coefficient, int exponent, poly_pointer
                                                  *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 */
   poly_pointer temp;
   temp = (poly_pointer)malloc(sizeof(poly_node));
   if (IS_FULL(temp)) {
      fprintf(stderr, "The memory is full\n");
      exit(1);
   }
   temp->coef = coefficient;
   temp->expon = exponent;
   (*ptr)->link = temp;
   *ptr = temp;
}
```

**Program: Attaching node at the end the list**

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

1. coefficient additions
2. exponent comparisons
3. creation of new nodes for d

If we assume that each of these operations takes a single unit of time if done once, then the number of times that we perform these operations determines the total time taken by padd.

This number clearly depends on how many terms are present in the polynomials a and b. Assume that a and b have m and n terms, respectively:
achieved when the exponents of one polynomial are a subset of the exponents of the other.

$$A(x) = a_{m-1}x^{e_{m-1}} + \cdots + a_0x^{e_0}$$

$$B(x) = b_{n-1}x^{f_{n-1}} + \cdots + b_0x^{f_0}$$

where $a_i, b_i \neq 0$ and $e_{m-1} > \cdots > e_0 \geq 0$, $f_{n-1} > \cdots > f_0 \geq 0$. Then clearly the number of coefficient additions varies as:

$$0 \leq \text{number of coefficient additions} \leq \min\{m,n\}$$

The lower bound is achieved when none of the exponents are equal, while the upper is achieved when the exponents of one polynomial are a subset of the exponents of the other.

### 2.7.3 Erasing Polynomials

The use of linked lists is well suited to polynomial operations. We can easily imagine writing a collection of functions for input, output, addition, subtraction, and multiplication of polynomials using linked lists as the means of representation. A hypothetical user who wishes to read in polynomials $a(x)$, $b(x)$, and $d(x)$ and then compute $e(x) = a(x) * b(x) + d(x)$ would write his or her main function as:

```
poly_pointer a, b, d, e
.
.
.
a = read_poly();
b = read_poly();
d = read_poly();
temp = pmult(a,b);
e = padd(temp,d);
print_poly(e);
```

If user wishes to compute more polynomials, it would be useful to reclaim the nodes that are being used to represent temp (r) since we created temp (x) only to hold a partial result for d {x}. By returning the nodes of temp {x\ we may use them to hold other polynomials. One by one, erase frees the nodes in temp

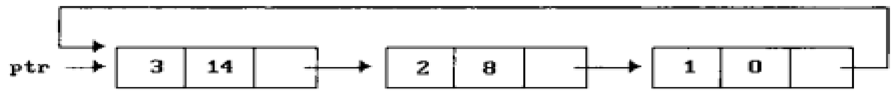
```
void erase(polyPointer *ptr)
{/* erase the polynomial pointed to by ptr */
   polyPointer temp;
   while (*ptr) {
      temp = *ptr;
      *ptr = (*ptr)→link;
      free(temp);
   }
}
```

Erasing a polynomial

of nodes in the list using *cerase*

## 2.7.4 Circular list representation of Polynomial

We can free all the nodes of a polynomial more efficiently if we modify our list structure so that the link field of the last node points to the first node in the list . We call this a circular list. A singly linked list in which the last node has a null link is called a chain.



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

**Figure 2.11 Circular Representation of polynomial**

As we indicated earlier, we free nodes that are no longer in use so that we may reuse these nodes later. We can meet this objective, and obtain an efficient erase algorithm for circular lists, by maintaining our own list (as a chain) of nodes that have been "freed." When we need a new node, we examine this list. If the list is not empty, then we may use one of its nodes.

Only when the list is empty do we need to use malloc to create a new node. Let avail be a variable of type poly-pointer that points to the first node in our list of freed nodes. Henceforth, we call this list the available space list or avail list. Initially, we set avail to NULL. Instead of using malloc and free, we now use get-node and ret-node .

```
void cerase(polyPointer *ptr)
{/* erase the circular list pointed to by ptr */
   polyPointer temp;
   if (*ptr) {
      temp = (*ptr)→link;
      (*ptr)→link = avail;
      avail = temp;
      *ptr = NULL;
   }
}
```

Erasing a circular list

We may erase a circular list in a fixed amount of time independent of the number of nodes in the list using cerase Figure 2.11 shows the changes involved in erasing a circular list.

**get Node Function**

```
polyPointer getNode(void)
{/* provide a node for use */
   polyPointer node;
   if (avail) {
      node = avail;
      avail = avail→link;
   }
   else
      MALLOC(node, sizeof(*node));
   return node;
}
```

A direct changeover to the structure of Figure 2.10 creates problems when we implement the other polynomial operations since we must handle the zero polynomial as a special case.

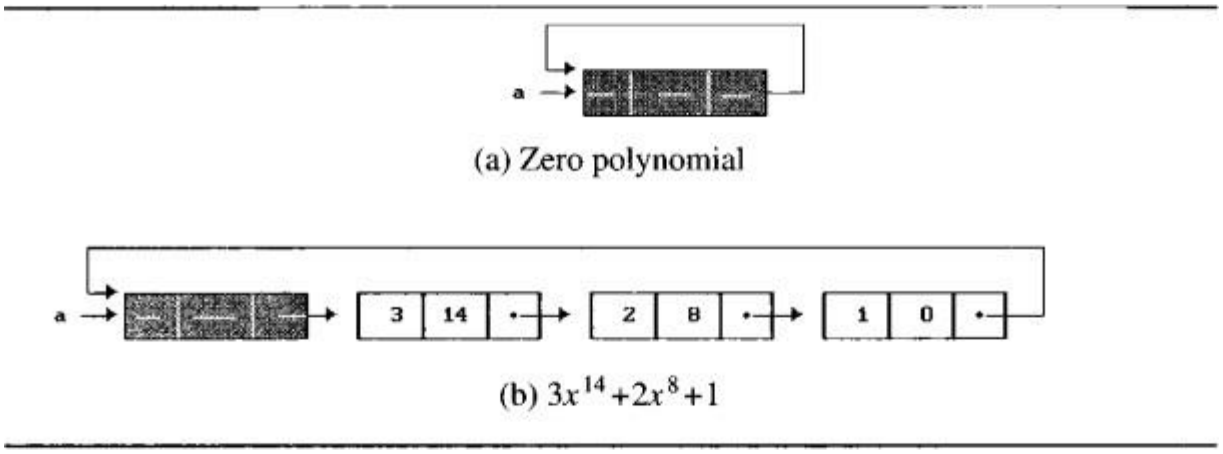


**Figure 2.12 (a) and (b) Polynomial with header nodes**

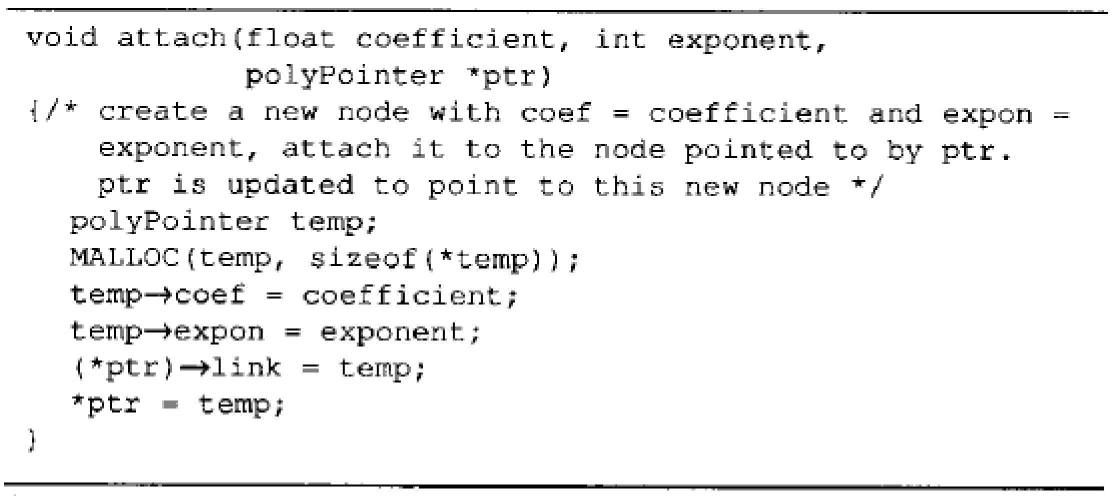
To avoid this special case, we introduce a head node into each polynomial, that is, each polynomial, zero or nonzero, contains one additional node. The exponent and coef fields of this node are irrelevant. Thus, the zero polynomial has the representation of Figure 2.12(a), while a(x) = 2.12(b).

**retNode Functon**

```
void retNode(polyPointer node)
{/* return a node to the available list *
   node→link = avail;
   avail = node;
}
```

**Adding two polynomial represented as circular lists with header nodes**

```
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;
}
```



```
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;
}
```

Attach a node to the end of a list

## Question Bank

1. Define Queue. Implement the operations of queue using arrays. Apply the same on job scheduling.
2. Show how queues are represented using arrays?
3. Explain queues operations using dynamic arrays.
4. Give the disadvantage of ordinary queue and how it is solved in circular queue. Explain with suitable example how you would implement circular queue using dynamically allocated array
5. What is circular queue? Explain how it is differ from linear queue. Write a C program for primitive operations of circular queue.
6. Explain multiple stacks and queues.
7. What is linked list? Explain the different types of linked list with examples.
8. Give a node structure to create a linked list of integers and write a C function to performthe following.
   a. Create a three-node list with data 10, 20 and 30
   b. Inert a node with data value 15 in between the nodes having data values 10 and 20
   c. Delete the node which is followed by a node whose data value is 20
   d. Display the resulting singly linked list.
9. With node structure show how would you store the polynomials in linked lists? Write Cfunction for adding two polynomials represented as circular lists.
10. Write a note on:
    a. Linked representation of sparse matrix
    b. Doubly linked list.
11. Write a function to insert a node at front and rear end in a circular linked list. Write downsequence of steps to be followed.

12. Define linked list. Write a C program to implement the insert and delete operation on aqueue using linked list.
13. Write a C-function to add two polynomials using linked list representation. Explain withsuitable example.
14. Explain how a chain can be used to implement a queue. Write the functions to insert anddelete elements from such a queue.