Module-2	8Hours
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 Text Book: Chapter-3: 3.3, 3.4, 3.7 Chapter-4: 4.1 to 4.4	

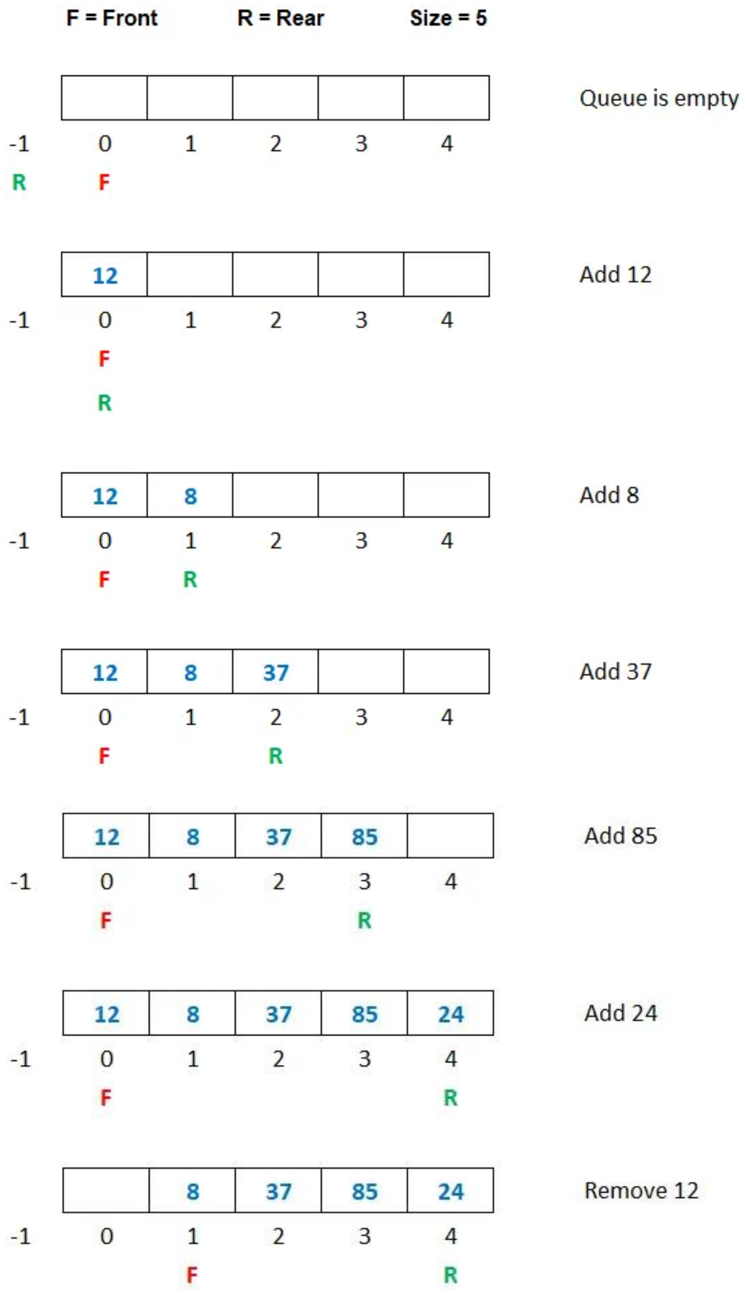
Queue

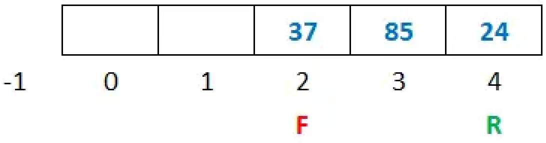
A Queue is a data structure in which we can add element only at one end, called the rear of the queue, and delete element only at the other end, called the front of the queue.

There are two operations possible on the queue.

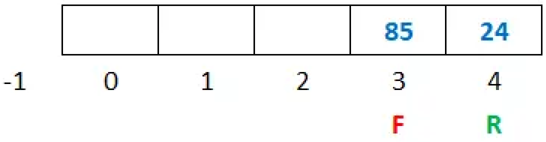
- Add an element to the queue.
- Delete an element from the queue.

To understand how the above operations work on a queue. See the example given below.

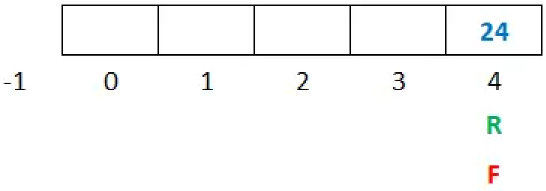




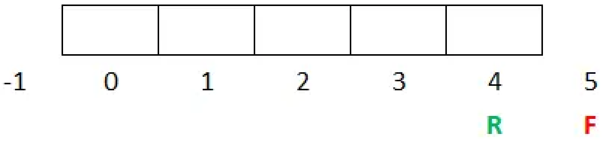
Remove 8



Remove 37



Remove 85



Remove 24

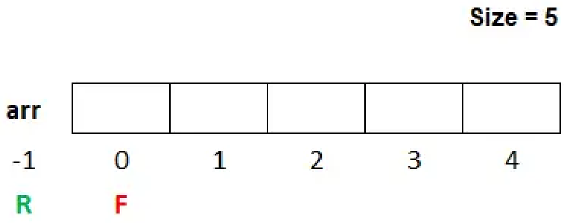
From the above image, we can see that when we add a new element in the queue, the variable R is increased by 1, and the new element is added at the new position of R. Similarly, when we delete an element from the queue, the variable F is increased by 1.

The queue behaves like a first in first out manner. It means that the elements that are added first to the queue, are removed first from the queue.

So a queue is also known as FIFO (First In First Out) data structure.

Array Implementation of Queue

Since a queue is a collection of the same type of elements, so we can implement the queue using an array.



In the above image, we can see an array named arr whose size is 5. We take two variables R and F, The variable R stands for rear and the default value is -1. The variable F stands for front and the default value is 0.

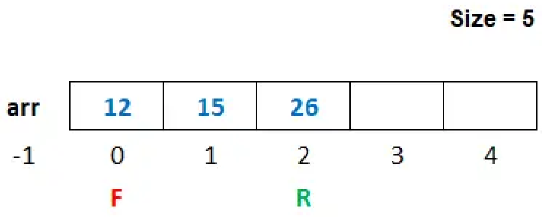
Add Operation in Queue

For add operation in the queue first, we check if the value of R is equal to the value of size-1 then, we will display a message Queue is full, else we will increase the value of R by 1 and add the element in the array at the new location of R.

Example

```
if(R==size-1)
{
        printf("Queue is full\n");
}
else
{
        R=R+1;
        arr[R]=new_item;
}
```

If we add three elements, say 12, 15 and 26 in the queue, then the queue will look like as shown in the image below.



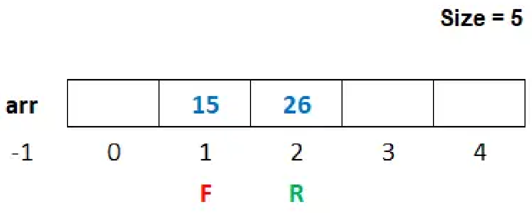
Delete Operation in Queue

For delete operation in the queue first, we check if the value of F is greater than the value of R then, we will display a message Queue is empty, else we will display the deleted element on the screen and then increase the value of F by 1.

Example

```
if(F>R)
{
        printf("Queue is empty\n");
}
else
{
        printf("Element Deleted = %d",arr[F]);
        F=F+1;
}
```

If we delete the first elements 12 from the queue, then the queue will look like as shown in the image below.



Program of Queue using Array

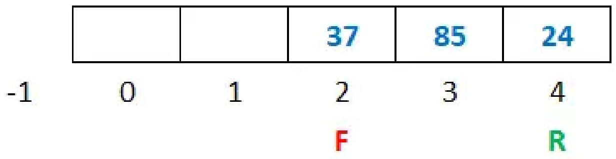
```
#include <stdio.h>
#include <stdlib.h>

#define size 5
```

```
int main()
{
    int arr[size],R=-1,F=0,ch,n,i;

    for(;;)         // An infinite loop
    {
        printf("1. Add\n");
        printf("2. Delete\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter Choice: ");
        scanf("%d",&ch);

        switch(ch)
        {
            case 1:
                if(R==size-1)
                    printf("Queue is full");
                else
                {
                    printf("Enter a number ");
                    scanf("%d",&n);
                    R++;
                    arr[R]=n;
                }
                break;

            case 2:
                if(F>R)
                    printf("Queue is empty");
                else
                {
                    printf("Number Deleted = %d",arr[F]);
                    F++;
                }
                break;

            case 3:
                if(F>R)
                    printf("Queue is empty");
                else
                {
                    for(i=F; i<=R; i++)
                        printf("%d ",arr[i]);
                }
                break;

            case 4:  exit(0);

            default: printf("Wrong Choice");
        }
    }
   return 0;
}
```

Circular Queue

A Circular Queue is a data structure in which elements are stored in a circular manner. In Circular Queue, after the last element, the first element occurs.

A Circular Queue is used to overcome the limitation we face in the array implementation of a Queue. The problem is that when the rear reaches the end and if we delete some elements from the front and then try to add a new element in the queue, it says "Queue is full", but still there are spaces available in the queue. See the example given below.



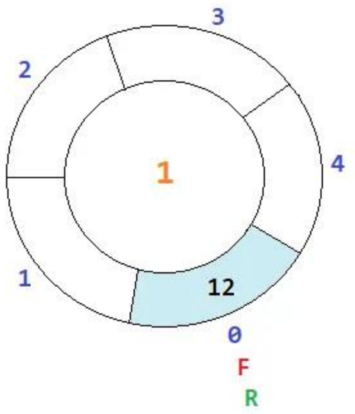
In the above image, the queue is full because the rear R reached the end of the queue. We have deleted two elements from the queue, so the front F is at index 2. We can see that there are spaces available in the queue, but we can't add a new element because the rear can't go back to index 0.

Operation on Circular Queue

There are two operations possible on the circular queue.

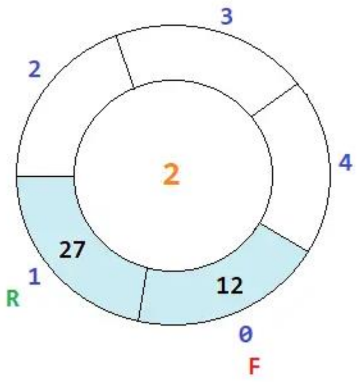
- **Add** an element in the circular queue.
- **Delete** an element from the circular queue.

To understand how the above operations work on a circular queue. See the example given below.



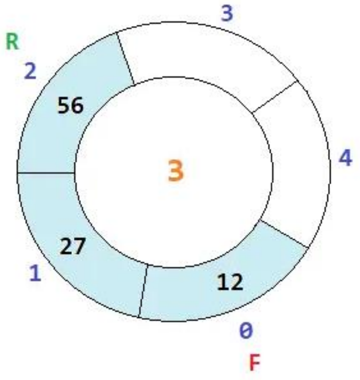
Size = 5
F = 0
R = -1
te = 0 (Total Elements)

Add 12
R = (R+1) % Size = 0
te = 1



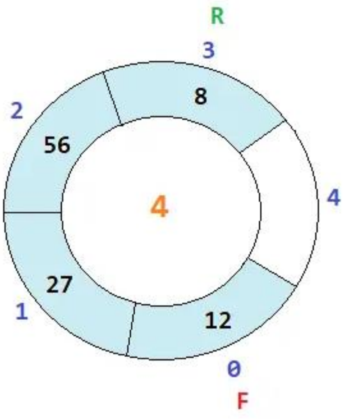
Size = 5
F = 0
R = 0
te = 1 (Total Elements)

Add 27
R = (R+1) % Size = 1
te = 2



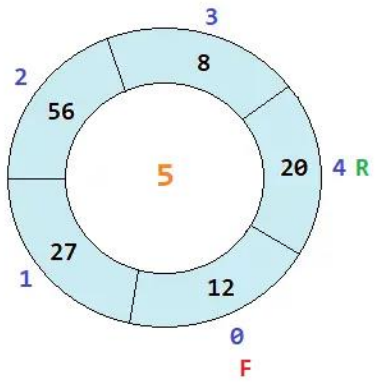
Size = 5
F = 0
R = 1
te = 2 (Total Elements)

Add 56
R = (R+1) % Size = 2
te = 3



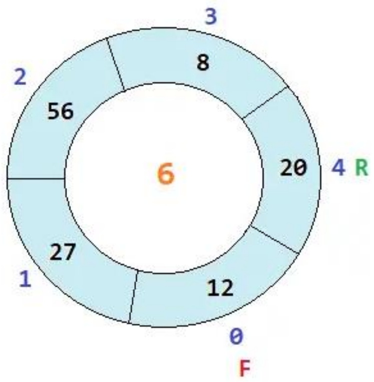
Size = 5
F = 0
R = 2
te = 3 (Total Elements)

Add 8
R = (R+1) % Size = 3
te = 4



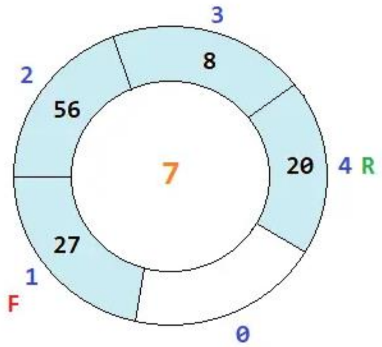
Size = 5
F = 0
R = 3
te = 4 (Total Elements)

Add 20
R = (R+1) % Size = 4
te = 5



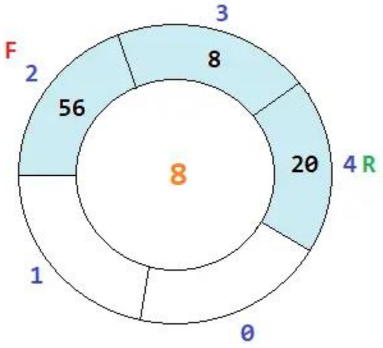
Size = 5
F = 0
R = 4
te = 5 (Total Elements)

Add 15
te == Size
Queue is full



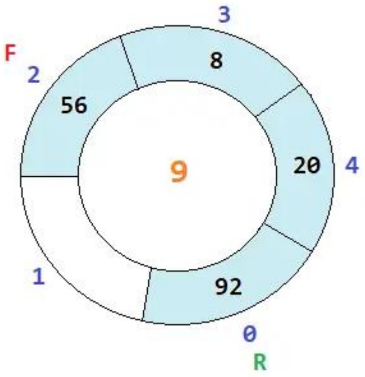
Size = 5
F = 0
R = 4
te = 5 (Total Elements)

Delete 12
F = (F+1) % Size = 1
te = 4



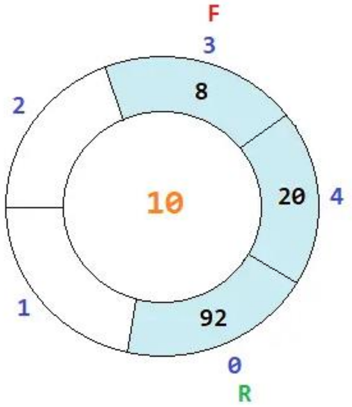
Size = 5
F = 1
R = 4
te = 4 (Total Elements)

Delete 27
F = (F+1) % Size = 2
te = 3



Size = 5
F = 2
R = 4
te = 3 (Total Elements)

Add 92
R = (R+1) % Size = 0
te = 4



Size = 5
F = 2
R = 0
te = 4 (Total Elements)

Delete 56
F = (F+1) % Size = 3
te = 3

From the above image, we can see that when we add a new element in the circular queue, the variable R is increased by R=(R+1)%Size, and the new element is added at the new position of R and te is increased by 1. Similarly, when we delete an element from the circular queue, the variable F is increased by F=(F+1)%Size and te is decreased by 1.

Add Operation in Circular Queue

For add operation in the circular queue first, we check if the value of te is equal to the value of size then, we will display a message Queue is full, else we will increase the value of R by R=(R+1)%Size and add the element in the array at the new location of R and then increased the value of te by 1.

```
if(te==size)
{
    printf("Queue is full\n");
}
```

```
else
{
    R=(R+1)%size;
    arr[R]=new_item;
   te=te+1;
}
```

Delete Operation in Circular Queue

For delete operation in the circular queue first, we check if the value of te is 0 then, we will display a message Queue is empty, else we will display the deleted element on the screen and then increase the value of F by F=(F+1)%Size and then decrease the value of te by 1.

```
if(te==0)
{
    printf("Queue is empty\n");
}
else
{
    printf("Element Deleted = %d",arr[F]);
    F=(F+1)%size;
   te=te-1;
}
```

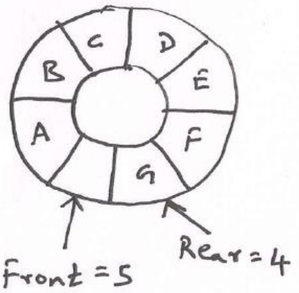
Program of Circular Queue using Array

Below is the complete program of circular queue in C using an array having size 5.

```
#include <stdio.h>
#include <stdlib.h>

#define size 5

int main()
{

        int arr[size],R=-1,F=0,te=0,ch,n,i,x;

        for(;;)             // An infinite loop
        {
                printf("1. Add\n");
                printf("2. Delete\n");
                printf("3. Display\n");
                printf("4. Exit\n");

                printf("Enter Choice: ");
                scanf("%d",&ch);
        switch(ch)
        {
            case 1:
                if(te==size)
                    printf("Queue is full");
                else
                {
                    printf("Enter a number ");
                    scanf("%d",&n);
                    R=(R+1)%size;
                    arr[R]=n;
                    te=te+1;
                }
             break;

            case 2:
                if(te==0)
                    printf("Queue is empty");
                else
                {
                    printf("Number Deleted = %d",arr[F]);
                    F=(F+1)%size;
                    te=te-1;
                }
            break;

            case 3:
                if(te==0)
                    printf("Queue is empty");
                else
                {
                    x=F;
                    for(i=1; i<=te; i++)
                    {
                        printf("%d ",arr[x]);
                        x=(x+1)%size;
                    }
                }
            break;

            case 4:
                exit(0);

            default:
                printf("Wrong Choice");
        }
    }
    return 0;
}
```

Circular Queues using Dynamic Arrays:

By using a dynamically allocated array for the elements, we can increase the size of the array as needed. Let capacity be the number of positions in the array queue. To add an element to a full queue, we use array doubling. Consider the full queue in the following figure, shows a queue with seven elements in an array whose capacity is 8.



(a) Full circular view

queue

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
C	D	E	F	G		A	B

front = 5, rear = 4

(b) Flattened view of circular full queue

The number of elements copied can be limited to capacity - 1 by customizing the array doubling code so as to obtain the configuration of following figure.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
C	D	E	F	G	A	B									

front = 15, rear = 6

This configuration may be obtained as follows:

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.

Program below 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;
}
```

The program below 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 * size of (*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 wraps around */
          copy (queue + start, queue + capacity, new Queue);
          copy (queue, queue + rear +1, new Queue + capacity - start);
     }
     /* switch to new Queue */
     front = 2 * capacity -1;
     rear = capacity -2;
     capacity *= 2;
     free (queue);
     queue = newQueue;
}
```

MULTIPLE STACKS AND QUEUES

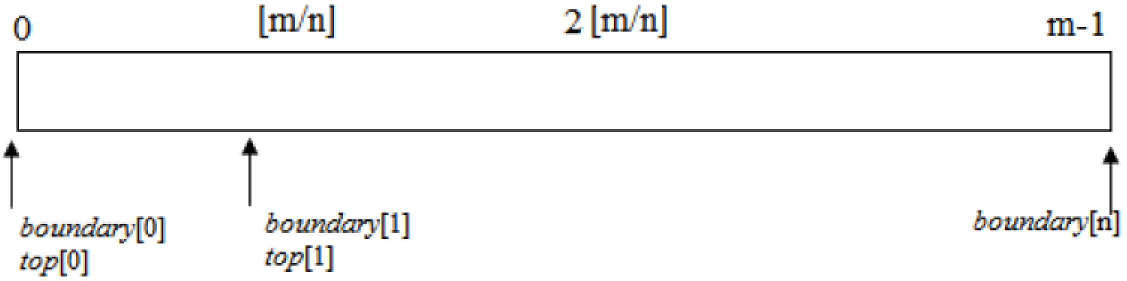
- In multiple stacks, we examine only **sequential mappings** of stacks into an array. The array is one dimensional which is **memory[MEMORY_SIZE]**. Assume ***n*** stacks are needed, and then divide the available memory into *n* segments. The array is divided in proportion if the expected sizes of the 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;
```



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+1])
                        stackFull(i);
                memory[++top[i]]  = item;
        }
```

Program: Add an item to the ith stack

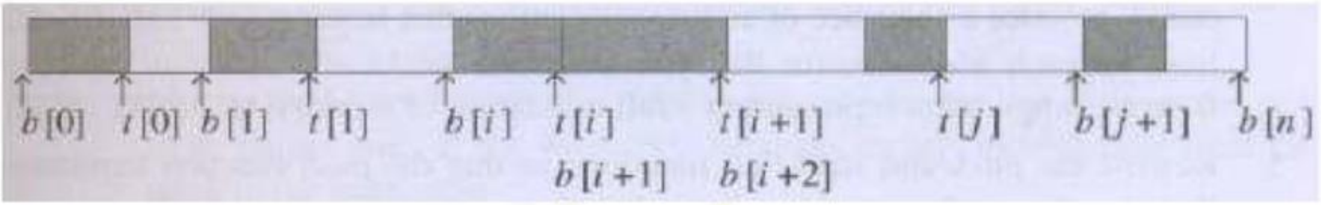
Implementation of the delete operation

```
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.

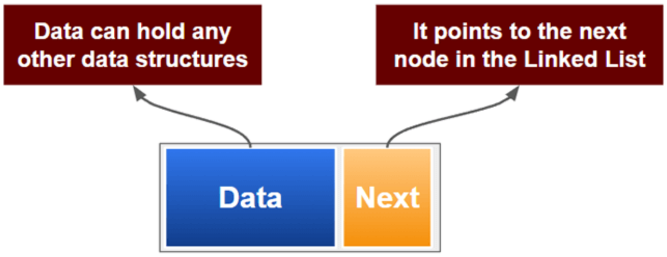


b boundary , t=top

Singly Linked List

Singly Linked List is a linear and unidirectional data structure, where data is saved on the nodes, and each node is connected via a link to its next node. Each node contains a data field and a link to the next node. Singly Linked Lists can be traversed in only one direction.

Here's a node structure of a Singly Linked List:


Data can hold any other data structures
It points to the next node in the Linked List
Data
Next

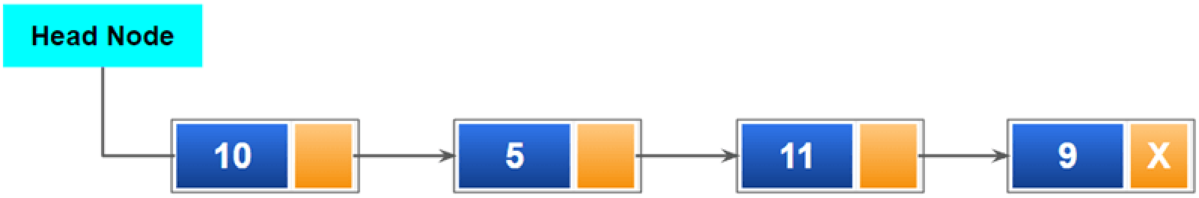
Node structure defined in C:

```
struct Node {
   int data;
   struct Node* next;
};
```

Operations of Singly Linked List

- Inserting at head
- Inserting at tail
- Inserting after a node
- Delete the head node
- Delete the tail node
- Search and Delete a node
- Traversing the Linked List

Here's an example of a linked list with four nodes.



Insertion at the head of a Singly Linked List

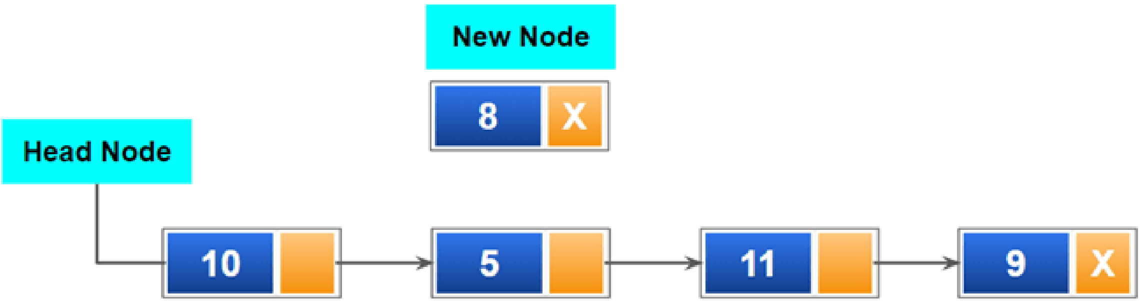
To perform this operation, we need to follow two important conditions. They're

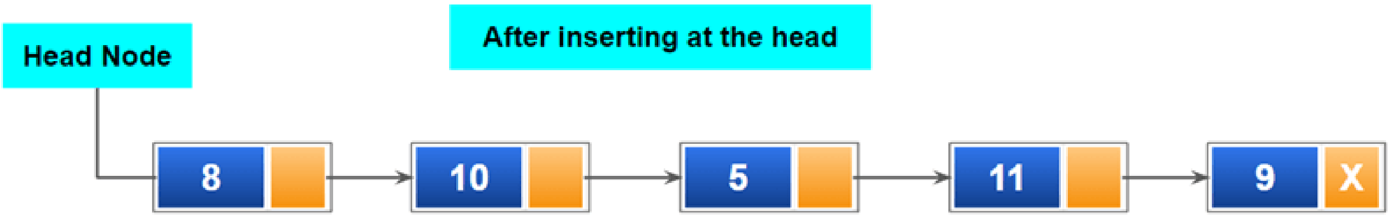
1. If the list is empty, then the newly created node will be the head node, and the next node of the head will be "NULL".
2. If the list is not empty, the new node will be the head node, and the next will point to the previous head node.

Here's the C code for inserting a node at the head of a linked list:

```
struct Node* insertAtHead(struct Node* head, int value) {

   // Create a new node with the given value
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
```

```
    // If the list is empty, make the new node the head
    if (head == NULL) {
        newNode->next = NULL;
        head= newNode;
        return head;
    }
else {
        // Otherwise, insert the new node at the head
        newNode->next = head;
        head= newNode;
        return head;
    }
}
```



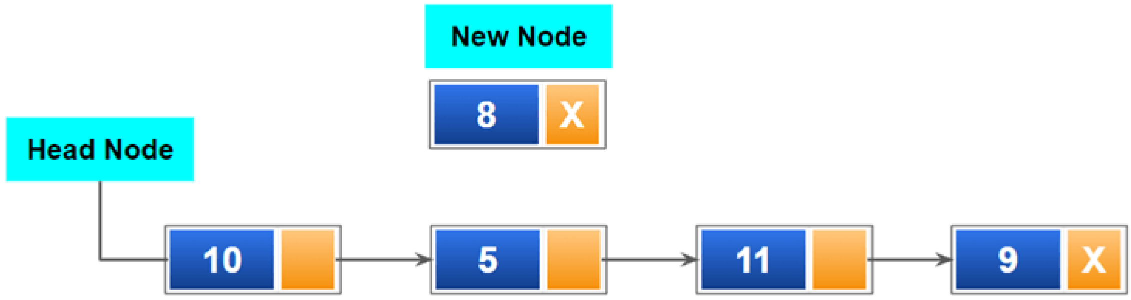


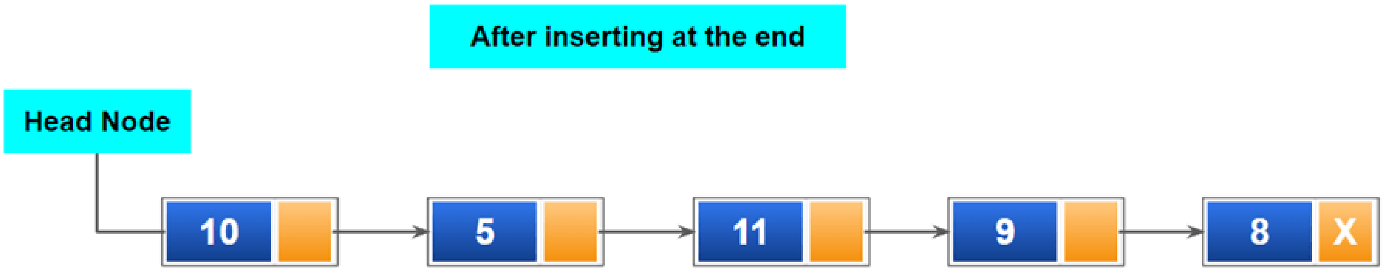
Insertion at the end of a Singly Linked List

Step 1) Traverse until the “next” node of the current node becomes null.
Step 2) Create a new node with the specified value.
Step 3) Assign the new node as the next node of the tail node.

C code for inserting a node at the tail of a singly list:

```
struct Node* insertAtEnd(struct Node* head, int value) {

    // Create a new node with the given value
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;

    // If the list is empty, make the new node the head
    if (head == NULL) {
        return newNode;
    }
    else {
        // Traverse the list to the end
        struct Node* current = head;
        while (current->next != NULL) {
            current = current->next;
        }
        // Insert the new node at the end
        current->next = newNode;
        return head;
    }
}
```



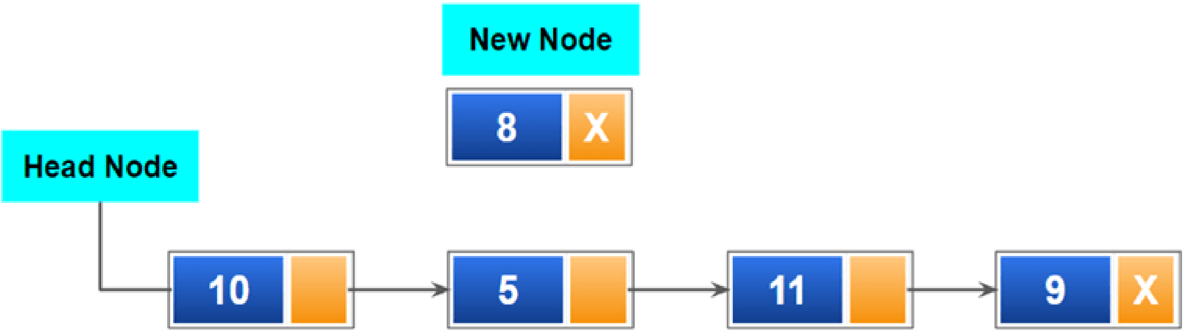


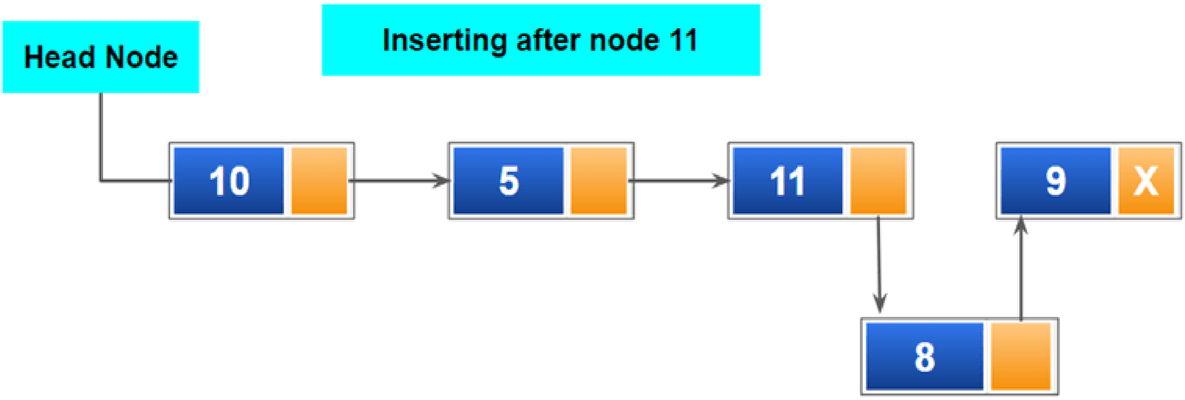
Insertion after a node in a Singly Linked List

Step 1) Traverse the next node until the value of the current node equals the search item.
Step 2) New node’s next pointer will be the current node’s next pointer.
Step 3) Current node’s next node will be the new node.

Here’s the C code for inserting a node after a node:

```
struct Node* insertAfter(struct Node* head, int value, int searchItem) {

    // Create a new node with the given value
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;

    // Traverse the list to find the node with the specified searchItem
    struct Node* current = head;
    while (current != NULL && current->data != searchItem) {
        current = current->next;
    }

    // If the searchItem is not found, return the original list
    if (current == NULL) {
        return head;
    }

    // Insert the new node after the node with searchItem
    newNode->next = current->next;
    current->next = newNode;

    return head;
}
```



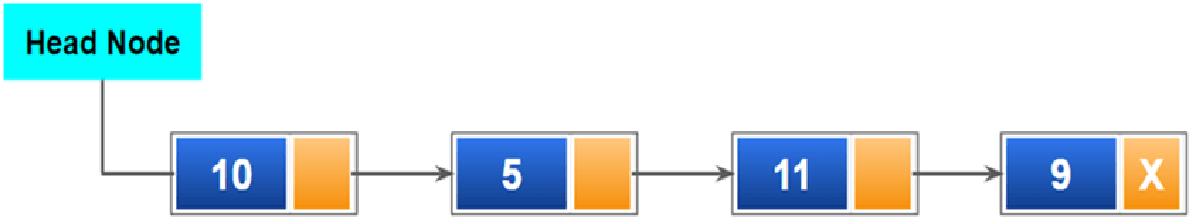


Delete the head node of the singly linked list

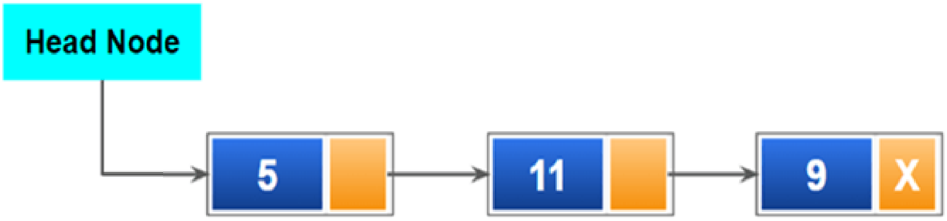
Step 1) Assign the next node of the head node as the new head.
Step 2) Free the allocated memory by the previous head node.
Step 3) Return the new head node.

The C code for deleting the head node:

```
struct Node* deleteHead(struct Node* head) {

    // If the list is empty, return NULL
    if (head == NULL) {
        return NULL;
    }

    // Store the current head in a temporary variable
    struct Node* temp = head;

    // Update the head to the next node
    head = head->next;

    // Free the memory of the original head
    free(temp);

    return head;
}
```



After deleting the head

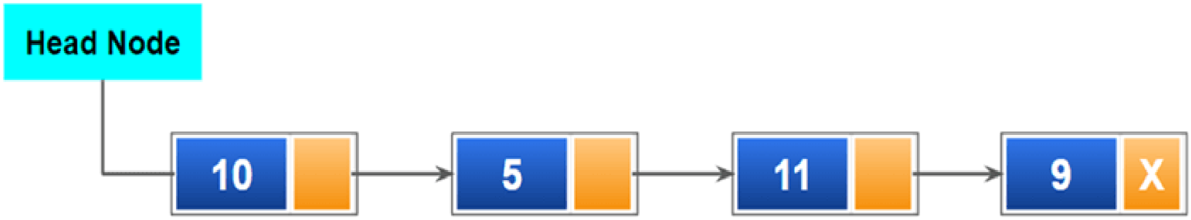


Delete the tail node of the singly linked list

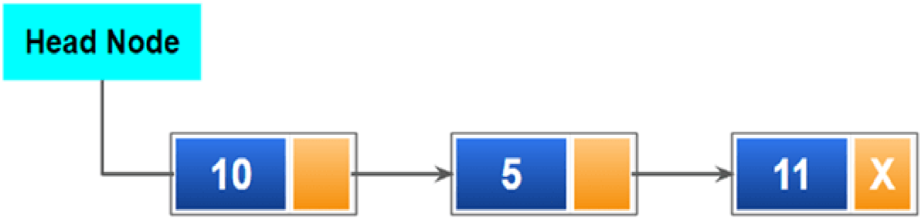
Step 1) Traverse before the tail node. Save the current node.
Step 2) Free the memory of the next node of the current node.
Step 3) Set the next node of the current node as NULL.

Here's the C code for deleting the tail node:

```
struct Node* deleteTail(struct Node* head) {

    // If the list is empty or has only one element, free the head and return NULL
    if (head == NULL || head->next == NULL) {
        free(head);
        return NULL;
    }

    // Traverse the list to the second-to-last node
    struct Node* current = head;
    while (current->next->next != NULL) {
        current = current->next;
    }

    // Free the last node and update the next pointer of the second-to-last node
    free(current->next);
    current->next = NULL;

    return head;
}
```



After deleting the tail

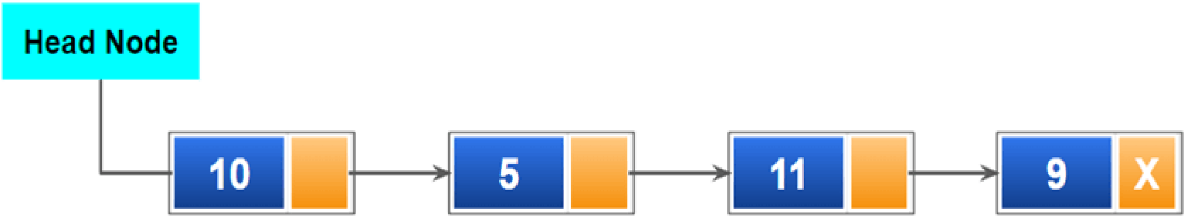


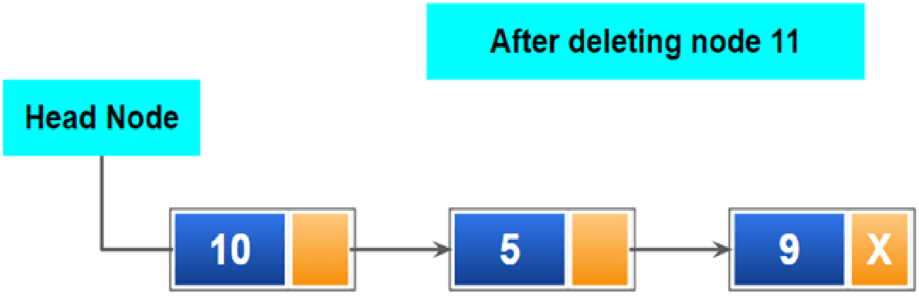
Search and delete a node from a singly linked list

Step 1) Traverse until the end of the linked list. Check if the current node is equal to the search node or not.
Step 2) If any node matches, store the node pointer to the current node.
Step 3) The "next" of the previous node will be the next node of the current node.
Step 4) Delete and free the memory of the current node.

C code for search and delete a node from a singly linked list:

```
struct Node* searchAndDelete(struct Node* head, int searchItem) {

    // If the list is empty, return NULL
    if (head == NULL) {
        return NULL;
    }

    // Traverse the list to find the node with the specified searchItem
    struct Node* current = head;
    while (current->next != NULL && current->next->data != searchItem) {
        current = current->next;
    }

    // If the searchItem is not found, return the original list
    if (current->next == NULL) {
        return head;
    }

    // Delete the node with the specified searchItem
    struct Node* temp = current->next;
    current->next = temp->next;
    free(temp);

    return head;
}
```





Traverse a singly linked list

Step 1) Traverse each node until we get null as the next node.
Step 2) Print the value of the current node.

C code for traversing a singly linked list:

```
void traverse(struct Node* head) {
    // Traverse the list and print the values
    while (head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}
```

Linked Stacks

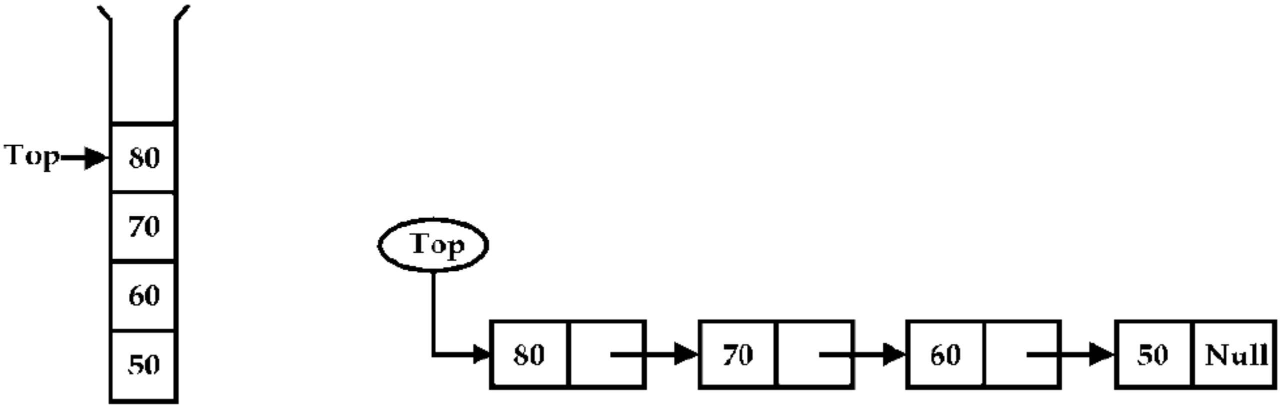
Stack is a linear data structure which follows the property Last In First Out (LIFO). This means, the last inserted element will be the first to be removed. Because of this nature, stack has only one end indicated by variable Top. Top take care of the element present at the top of the stack. Two basic operations that can be applied to stack are

1. Insertion, usually termed as Push and
2. Deletion, usually termed as Pop.

Two conditions are checked while push or pop operation on the linked list.

1. Overflow: the stack is full; element cannot be inserted
2. Underflow: the stack is empty; cannot remove an element

Linked list representation of the stack allows it to grow or shrink without any prior fixed limit due to the dynamic nature of the linked list. Diagram shows how a stack data structure can be represented using linked list.

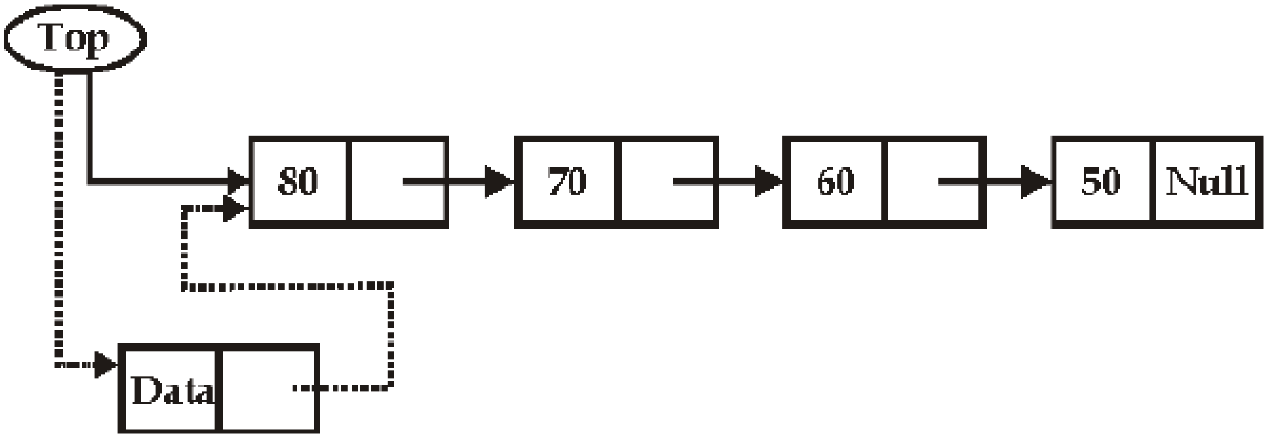


This is the linked list representation of a stack having four elements. The top most element is at the beginning of the list. Here, the top most element is 80. And the oldest element is at the end of the list. Here, the oldest element is 50. The first element of the linked list here is represented by a pointer variable Top.

Push Operation

Push operation is the insertion of an element at the top of the stack. In case of linked list to represent the stack, the push operation can be performed by inserting an element at the beginning of the linked list. Here is the C code depicting the push operation on stack where, the element Data is inserted into the stack using linked list. A stack pointer variable Top points to the top most element or node of the stack.

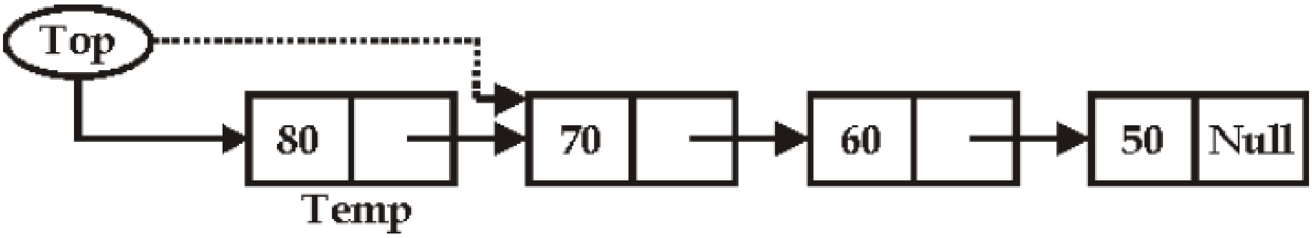
```
// Define a structure for the node
struct Node
{
        int info;               // Data of the node
        struct Node* next;      // Pointer to the next node
};

struct Node* top = NULL;

// Function to push a new element onto the stack
void push(int data)
{

        struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->info = data;
        newNode->next = top;
        top = newNode;
}
```



Pop Operation

Pop operation is the deletion of an element from the stack that can be performed by deleting an element at the beginning of the linked list. While performing pop operation, underflow condition must be checked. Here, the underflow condition occurs when the stack is empty and we try to delete an element from the stack. Here is an algorithm depicting the pop operation on stack, where, the first element or node pointed by stack pointer variable Top is deleted.

```
// Function to pop an element from the stack
int pop()
{

        if (top == NULL)
        {
                printf("Stack is Empty, Underflow Condition\n");
        }

        int data = top->info;
        struct Node* temp = top;
        top = top->next;

        free(temp);
        return data;
}
```



Linked Queues

Queue is a linear data structure which follows the property, First In first Out (FIFO) or First Come First Serve (FCFS). The first inserted element will be the first to be removed. Queue has two ends, one is Front and other is Rear. Front variable indicates the oldest element inserted into the queue and Rear variable indicates the last element inserted into the queue.

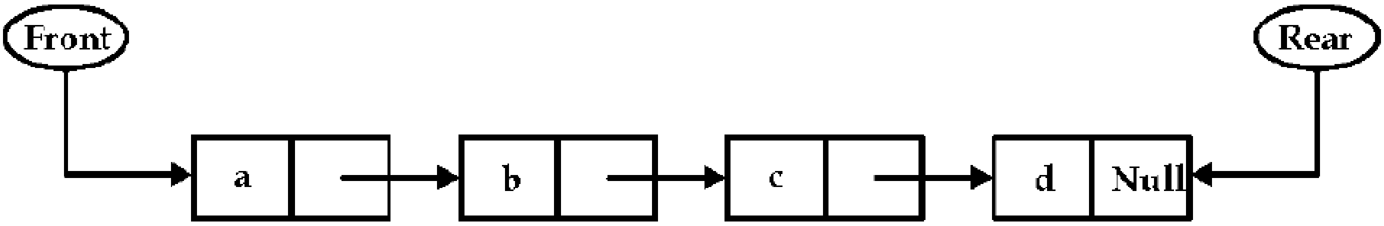
Two basic operations that can be applied to queue are

1. Insertion operation, that takes place at Rear end
2. Deletion operation, that takes place at Front end.

Two conditions are checked while insertion or deletion operation on the linked list.

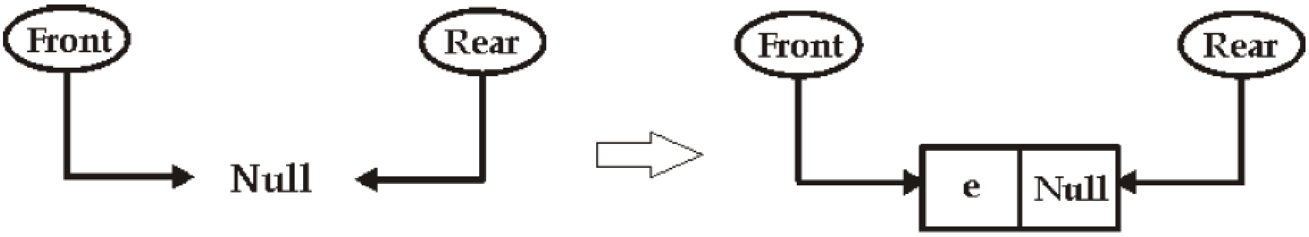
1. Overflow: the queue is full; element cannot be inserted
2. Underflow: the queue is empty; cannot remove an element

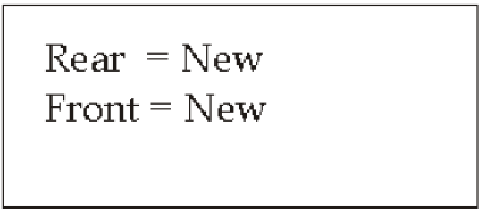
Linked list representation of the queue allows it to grow or shrink without any prior fixed limit due to the dynamic nature of the linked list. Diagram shows how a queue data structure can be represented using linked list.



Insertion Operation

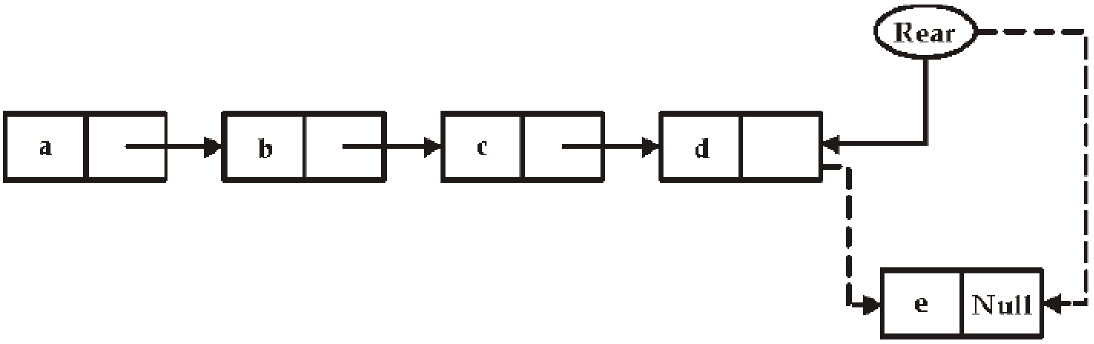
The insertion of new element takes place at the rear end of the queue. Below is the diagram along with algorithm for insertion of an element ‘e’ into the queue. The element ‘e’ is inserted at the Rear end of the list.





```
Rear  = New
Front = New
```

This insertion of a New element ‘*e*’ in the empty queue



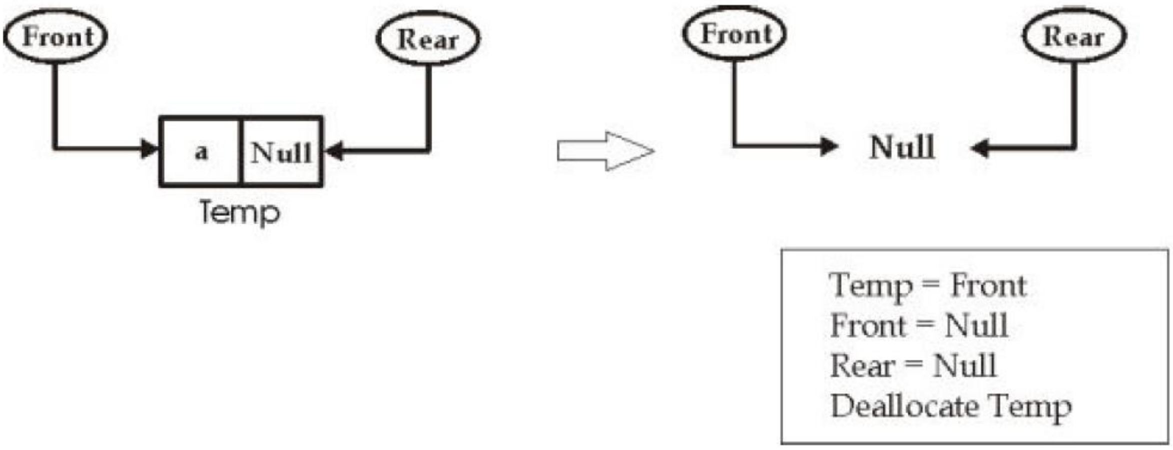
Rear ->Next = New
Rear = New

This insertion of a New element ‘*e*’ in the queue

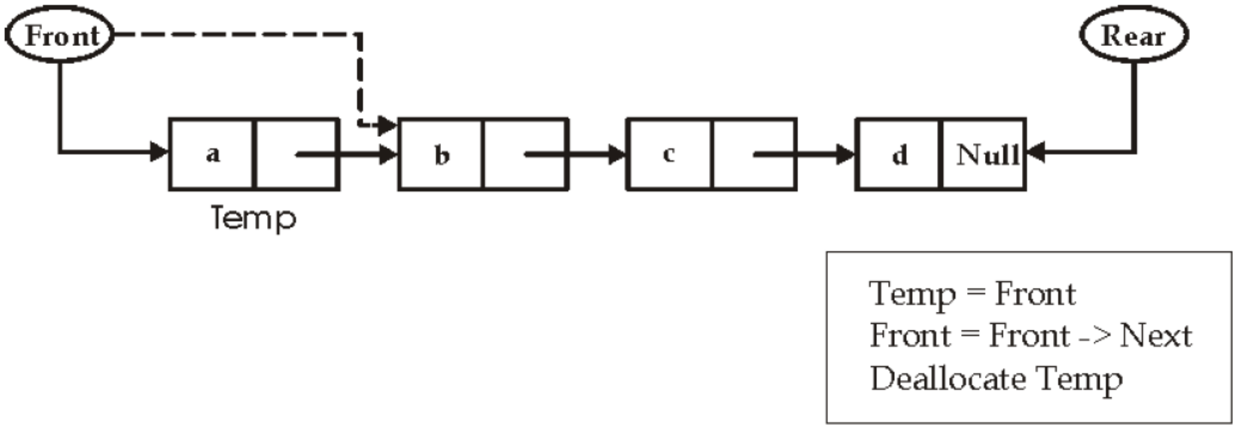
```
// Define a structure for the node
struct Node
{
        int info;               // Data of the node
        struct Node* next;      // Pointer to the next node
};

   struct Node* front = NULL;
   struct Node* rear = NULL;

// Function to insert an element into the queue
void enqueue( int data)
{
        struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

        newNode->info = data;
        newNode->next = NULL;

        if (rear == NULL)
        {
                front = rear = newNode;
        }
        else
        {
                rear->next = newNode;
                rear = newNode;
        }

}
```

Deletion operation

Deletion of an element from the will be performed at the beginning of the linked list. While deleting an element from the queue, underflow condition must be checked. Here, the underflow condition occurs when the queue is empty and we try to delete an element from the queue. Here is the algorithm depicting the deletion operation on queue. The element at the front end of the queue is deleted.



Deletion of an element from the Queue having only one node



Deletion of an element from the Queue

```
// Function to remove an element from the queue
int dequeue()
{
        if (front == NULL)
        {
                printf("Queue is Empty\n");
        }

        int data = front->info;

        struct Node* temp = *front;

        if (front == rear)
        {
                front = rear = NULL;
        }
        else
        {
                front = front->next;
        }

        free(temp);

        return data;
}
```

Application of Linked Lists:

Polynomials:

Polynomial Representation:

In general, we want to represent the Polynomial:

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

where the a_i are nonzero coefficients and the e_i are nonnegative integer exponents such that $e_{m-1} > e_{m-2} > \dots > e_1 > e_0 \geq 0$. 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:

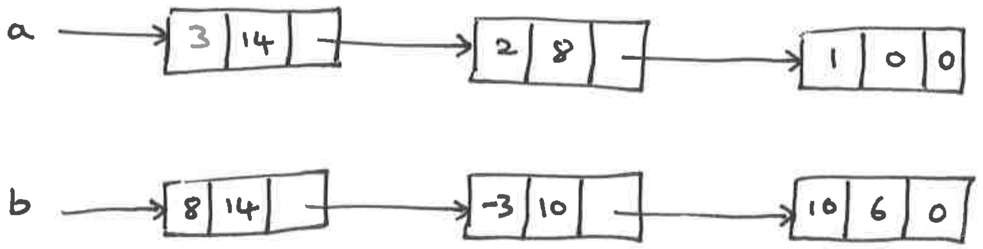
coef	expon	link

The following figure shows how we would store the polynomials

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

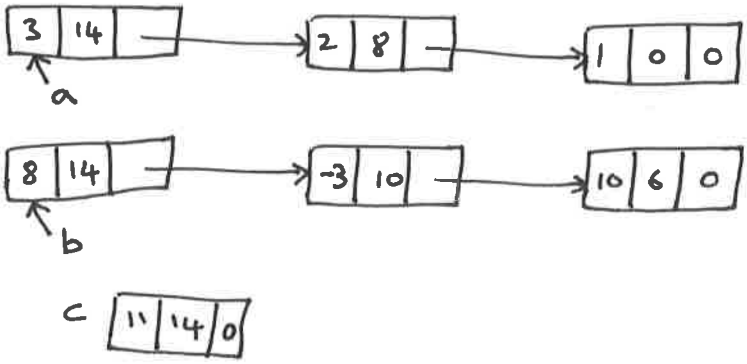
and

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

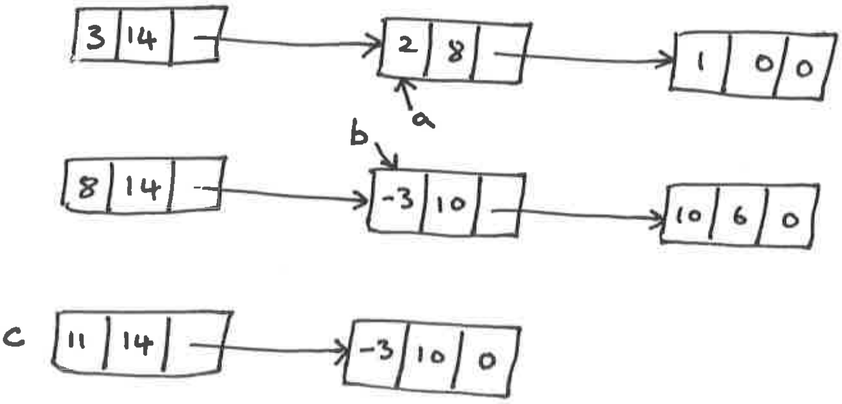


Adding Polynomials:

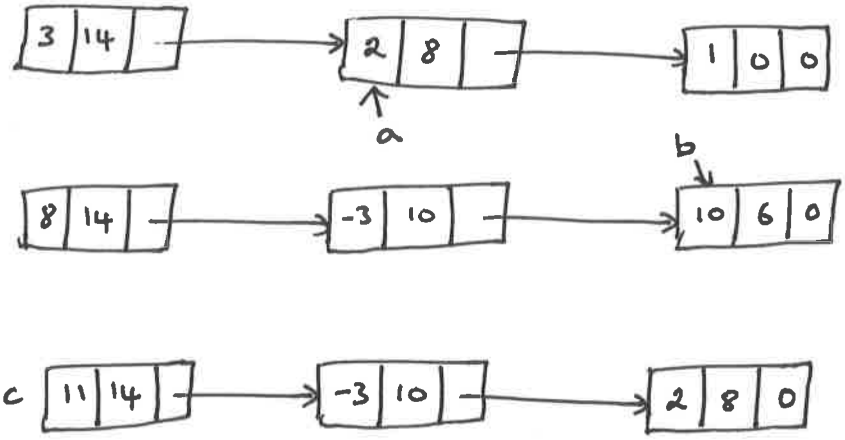
To add two polynomials, we examine their terms starting at the nodes pointed to by a and b. If the exponents of the two terms are equal, we add the two coefficients and create a new term for the result. we also move the pointers to the next nodes in a and b.



If the exponent of the current term in a is less than the exponent of the current term in b, then we create a duplicate term of b, attach this term to the result, called c, and advance the pointer to the next term in b.



we take a similar action on a if a->expon > b->expon.



```
poly Pointer   padd (poly Pointer a, poly Pointer b)
{  /* return a polynomial which is the sum of a and b */
    poly Pointer 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;
}
```

Circular list representation of Polynomials:

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.

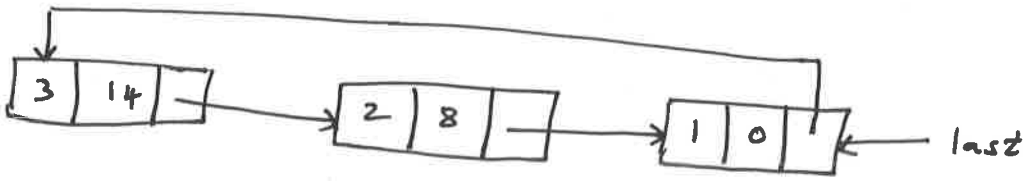


Fig. Circular representation of $3x^{14}+2x^{8}+1$