Unit – 2: Stack, Queue and Linked lists : Stacks, Stacks using Dynamic Arrays, Queues, Circular Queues Using Dynamic Arrays, Evaluation of Expressions, Multiple Stacks and Queues. Linked lists: Singly Linked Lists and Chains, Representing Chains in C, Linked Stacks and Queues, Additional List Operations, Doubly Linked Lists.

**2.1 Stacks**

**STACK:** It is a linear data structure and has collection of ordered data. It means elements are inserted and deleted at only one end, called top of the stack. This type of technique is called “LIFO”(Last In First Out). Stack can be represented in static & dynamic data structure. It means, stack can be developed by using arrays and linked list. It is used to solve the arithmetic expression & memory locations. The following rules are used to develop the stack.

*In this most recently entered element treated as a “top” element.*

*Suppose to insert an element in to the stack called “PUSH” operation.*

*Suppose to delete an element from the stack called “POP” operation.*

*Suppose to insert more information in to the stack called “STACK IS OVER FLOW”. It means stack is full ( top=size-1 ).*

*Suppose to delete more information from the stack called “STACK IS UNDER FLOW”. It means stack is empty ( top=-1 ).*

***Stack functions*** : There are two basic functions in the stack. Such as ***push & pop***

***1. Push operation:*** *Push is the term to insert or add an element into stack.* In this we first check the condition that the stack is full or not. If it is full then display the message “STACK IS OVER FLOW” other wise to increase the top position and also to assign the value at top position.

Algorithm: *Algorithm push( num )*

*if ( top == max )*

*printf(”STACK IS OVER FLOW”);*

*else*

*top = top + 1;*

*a[top] = num;*

***2. Pop operation:*** *Pop is the term to delete or remove an element from stack.*

In this we first check the condition stack is empty or not. If it is empty then display the message “*STACK IS UNDER FLOW*” other wise delete the top position element and also decrease the top position.

Algorithm: *Algorithm pop( )*

*if ( top == -1 )*

*printf(”STACK IS UNDER FLOW”);*

*else*

*printf(”Deleted Element = %d“,a[top]);*

*top = top -1;*

***Representation of Stack using Arrays:***Stack can be represented mainly in two methods. Such as *1. Using one dimensional Array (i.e. Linear Stack)*

*2. Using Singly Linked list (i.e. Linked Stack)*

***Representation of stack using Arrays*** *:* A *stack is a data structure* that will be represented by using array is called “Linear Stack”. The insertion and deletion operations are done in stack at one end, called “top” of the stack. TOP is a variable that represent the top element of the stack.

20

30

40

POP

PUSH

top

🡨 STACK

10

In the above representation stack contains four elements, on that “40” is a top element because it is most recently elements. It provides mainly three operations like

1. PUSH 2. POP 3. TRAVERSE

***PUSH Operation****:* Suppose to insert an element into the stack called “PUSH” operation. In this we first check the condition stack is full or not. If it is full then display the message “STACK IS OVER FLOW” other wise to increase the top position and also to assign the value at top position.

Algorithm: Algorithm push( num )

if ( top == max )

printf(”STACK IS OVER FLOW”);

else

top = top + 1;

a[top] = num;

***POP Operation****:* Suppose to delete an element from the stack called “POP” operation. In this we first check the condition stack is empty or not. If it is empty then display the message “STACK IS UNDER FLOW” other wise delete the top position element and also decrease the top position.

Algorithm: Algorithm pop( )

if ( top == -1 )

printf(”STACK IS UNDER FLOW”);

else

printf(”Deleted Element = %d“,a[top]);

top = top -1;

***TRAVERSE:*** Traverse means to ***display*** the list of elements. In this we display from top position to “0” position elements (first to last).

Algorithm: Algorithm traverse( )

{ int i = top;

while ( i >= 0 )

{ printf(“%d”,a[i];

i = i – 1;

}

}

***Representing the Stack using Singly Linked List :***  To develop the stack by using Linked List is called “Linked Stack”. It is dynamic data structure because to insert and delete the elements at run time(size is not fixed). The representation of linked stack is:

10

20

30

40

103

102

101

NULL

PUSH

POP

top

104

103

102

101

STACK

In the above representation stack contains four nodes, on that “40” is the top node because it is most recently entered element. It provides mainly three types of methods

1. PUSH 2. POP 3. TRAVERSE

**2.2 Stack Using Dynamic arrays:**

Stack Creates( ) = typedef struct {

int key;

/\* other fields \*/

}element;

element \*stack;

MALLOC(stack, sizeof(\*stack));

int top =-1;

**2.3 QUEUE:**

Queue is a linear data structure. It is a collection of ordered data. It means elements are inserted at one end and deleted at another end. Totally it follows “FIFO”(First In First Out) order. In queue most recently entered element treated as a last element. It is a static and dynamic data structure because to develop the queue by using arrays and linked list. It is used in reservation counters, cell phones, window applications etc.,

The following are some rules for developing the queue:

In queue elements are inserted at one end called “rear” end and also most recently entered element treated as a “rear” element (last).

In queue elements are deleted at another end called “front” end and also first entered element treated as a “front” element (first).

In queue, to insert more information into the queue called “QUEUE IS OVER FLOW”. It means queue is full (rear=size-1).

In queue, to delete more information from the queue called “QUEUE IS UNDER FLOW”. It means queue is empty (front=rear).

Queue represents mainly two types: 1. Linear Queue 2. Linked Queue

Linear Queue: Suppose we develop the queue by using arrays called “Linear Queue”. The representation of linear queue is:

10

20

30

40

50

front end

rear end

0 1 2 3 4

front QUEUE rear

In the above representation queue contains five elements, on that “10” is a front element and “50” is a rear element. It provides mainly three types of operations like

1. Insertion 2. Deletion 3. Traverse

**Insertion:** Suppose we insert an element into the queue called “Insertion”. In this we first check the condition queue is full or not. If it is full then display the message “QUEUE IS OVER FLOW” other wise increase the rear position and also store the value into the rear position.

**Algorithm**: insert( num )

{ if ( rear == size-1 )

printf(” QUEUE IS OVER FLOW”);

else

{ rear = rear + 1;

a[rear] = num;

} }

**Deletion:** Suppose we delete an element from the queue called “Deletion”. In this we first check the condition queue is empty or not. If it is empty then display the message “QUEUE IS UNDER FLOW” other wise increase the front position and delete the front position element.

**Algorithm**: delete( )

{ if ( rear = = front )

printf(” QUEUE IS UNDER FLOW”);

else

{ front = front + 1;

printf(“Deleted Element = %d“,a[front];

} }

**Traverse**: Traverse means to display the list of the elements. In this we display from front element to rear element.

**Algorithm**: traverse( )

{ i = front + 1;

while ( i <= rear )

{

printf(“%d”,a[i]);

i = i + 1;

} }

**Linked Queue:** Suppose we develop the queue by using linked list called Linked Queue. The representation of linked queue is:

front end

rear end

101 102 103 104 105

10 20 30 40 50

102 103 104 105 NULL

front QUEUE rear

In the above representation queue contains 5 nodes, on that “10” is a front node and “50” is a rear node. It provides mainly three types of operations like 1. Insertion 2. Deletion 3. Traverse

**2.4Circular Queues using dynamic arrays**

Circular Queue is a linear data structure. It is a collection of ordered data. It means elements are inserted at one end and deleted at another end. Totally it follows “FIFO”(First In First Out) order. In this elements are inserted at rear end and deleted at front end. The main purpose of circular queue is front element is always follows the rear element. The representation of circular queue is:

front

30

In the above representation queue contains 4 elements, on that “10” is a front element and “40” is a rear element. The Circular Queue provides mainly two representations like

1. Linear Circular Queue 2. Linked Circular Queue

Linear Circular Queue: Suppose we develop the circular queue by using arrays called “linear circular queue”. It provides mainly three types of operations like

1. Insertion 2. Deletion 3. Traverse

Linked Circular Queue: Suppose we develop the circular queue by using linked list called “linked circular queue”. It provides mainly three types of operations like

1. Insertion 2. Deletion 3. Traverse

**Insertion**: The insertion in Circular queue will be same with the linear queue. It keep track of front and rear with some extra logic. This is shown in algorithm.

Algorithm: INSERT\_CIRCQ(CIRCQ, front, rear, n, item)

rear = (rear + 1) mod n

if( front == rear ) then “Circular Queue is Full”

else

CIRCQ[rear] = item

**Deletion:** Whenever an element is deleted from the queue, the value of front is increased by 1. i.e Algorithm: DELETE\_CIRCQ(CIRCQ, front, rear, n, item)

if( front == rear ) then “Circular Queue is Empty”

else

front = (front + 1) mod n

item = CIRCQ[rear]

**Application of Queue:**

Queue is used to represent the records in a database in memory.

Queue is used to store the interrupt in operating system.

Queue is used to simulation traffic control system

Queue is used for CPU scheduling in multiprogramming and time sharing system.

Queue is used to multilevel queue scheduling.

Queue is used to Multilevel feedback queue scheduling

Queue is used to Round Robin scheduling.

**Simulation of Traffic Control System:**  It is nothing but an automated control system. Suppose in a junction four roads meet as 4 roads. There four signals named A, B, C, D for the traffic on the four roads. At a time only

one signal can be turned on to allow the traffic to pass. In this case, if AQ will be allowed to pass through other 3 roads when signal A is on. At the time the traffic in the Queues BQ, CQ, and DQ will wait till their signals on.

**CPU Scheduling in multiprogramming :** In multiprogramming several processes are kept in memory at one time and executing one by one. The OS commands the CPU and executing the processes in time shared basis.

**Multilevel queue scheduling:** In this scheduling the jobs are executed by the CPU based on priority basis.

**Multilevel feedback queue scheduling:** In this scheduling overcomes the both the difficulties. In this, allocating a time of 8 milliseconds for each processes(job) which is in a queue.

**Round Robin Scheduling:** The round-robin (RR) scheduling algorithm is designed especially for time sharing system. A small unit of time is called a **time quantum**. A time quantum is generally from 10 to 100 milliseconds. This type of algorithm can be implemented using circular queues.

**2.5 Evaluation of Expressions**:

Polish implemented rules to perform the arithmetic notation using reverse notation, called postfix notation.

The rules are inner most parenthesis will be evaluated at first

Exponentiation will be performed in second step

Multiplication and division will be performed in step 3

Additions and Subtraction will be performed in step 4.

Arithmetic Expression: An expression is a combination of operators & operands. It contains arithmetic operators called Arithmetic Expression. Eg: a+b, a+(b\*c)/d etc.,

Arithmetic Expression provides mainly three types of notations like

1. INFIX : In this, operator is placed between the operands.

2. PREFIX : In this, operator is placed before the operands.

3. POSTFIX : In this, operator is placed after the operands.

Generally, we write the expression in infix expression, those expression are automatically convert into prefix or postfix expressions because to solve the expression very fast. Stacks are very useful to convert the infix expression into prefix or postfix expressions.

Algorithm:

The infix expression is scanned from left to right.

When an operand is scanned then it is placed in postfix or prefix expression.

When an operator or left brace is scanned then it is placed in stack.

When right brace is scanned then pop the stack and append that operator in postfix or prefix expression.

Repeat the above steps until you reach the end of the expression.

Finally pop the stack until stack is empty.

To Convert the INFIX expression into POSTFIX expression

Ex : INFIX = A + ( B\*C )/D Scan Stack Postfix

A ------- A

+ + A

( +( A

B +( AB

\* +(\* AB

C +(\* ABC

) +(\*) ABC\*

/ +/ ABC\*

D +/ ABC\*D

--- + ABC\*D/

--- --- ABC\*D/+ => postfix

Evaluate the postfix expression:

INFIX = A + ( B\*C )/D POSTFIX = ABC\*D/+

= 2 + ( 2\*2) / 2 = 2 2 2 \* 2 / +

= 2 + 4/2 = 2 4 2 / +

= 2 + 2 = 2 2 +

= 4 = 4

To Convert the INFIX expression into PREFIX expression

Eg : INFIX = A + ( B\*C )/D

Scan Stack Infix

A ------- A

+ + A

( +( A

B +( BA

\* +(\* BA

C +(\* CBA

) +(\*) \*CBA

/ +/ \*CBA

D +/ D\*CBA

--- + /D\*CBA

--- --- +/D\*CBA => prefix

Evaluate the prefix expression:

INFIX = A + ( B\*C )/D PREFIX = +/D\*CBA

= 2 + ( 2\*2 )/2 = + / 2 \* 2 2 2

= 2 + 4/2 = + / 2 4 2

= 2 + 2 = + 2 2

= 4 = 4

**Transform Infix Expression into Postfix Expression:** Let Q be an arithmetic expression written in Infix notation. Both side of operands and operators contain left and right parentheses. We assume that Q contain exponentiations ( ^ ), multiplications ( \* ), divisions ( / ), additions ( + ), subtractions ( - ). All the operations are performed based on polish rules.

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

**Algorithm : Infix\_to\_Postfix( Q, P)**

Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

Step 1: Push “(“ onto STACK, and add “)” to the end of Q.

Step 2: Scan Q from left to right and repeat steps 3 to 6 for each element of Q until the STACK

is empty.

Step 3: If an operand is encountered, add it to P.

Step 4: If left parenthesis encountered, Push it onto STACK.

Step 5: If an operator is encountered, then Repeatedly pop from STACK and add to P each

operator which has same.

[ End of If structure ]

Step 6: If a right parenthesis is encountered then a)Repeatedly pop from STACK and add to P

each operator until a left parenthesis is encountered. which has same.

b) Remove the left parenthesis.

[ End of If structure ]

[ End of step 2 loop ]

**2.6 Multiple Stacks and Queues**

Assume that I refers to the stack number of one of the n stacks.To establish this stack, we must create indices for both the bottom and top positions of this stack. The relevant declarations are

#define MEMORY\_SIZE 100 /\* size of the memory \*/

#define MAX\_STACKS 10 /\* Max number of stacks plus 1 \*/

/\* global memory declaration \*/

element 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 we use the following code

top[0] = boundary[0] = -1;

for(j=1;j<n;j++)

top[j] = boundary[j] = (MEMORY\_SIZE/N)\*j;

boundary[n] = MEMORY\_SIZE-1;

The below programs are functions for multiple stacks appear to be as simple as those we used for the representation of a single stack.

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

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

**LINKED LISTS**

Drawbacks of sequential data structures: Arrays are fundamental sequential data structures. Even Stacks and Queues are also represented using an array. However arrays (sequential data structures) are having following drawbacks.

inefficient implementation of insertion and deletion operations and

Inefficient use of storage memory.

Suppose, if an array contains 20 locations i.e. a[20] then it allocates 20 contiguous memory locations to store 20 data elements. In this, if we store the elements or not then that an array occupies the memory.

Merits of linked data structure: A linked representation serves the drawbacks of sequential representation by the following merits.

i) Efficient implementation of insertion and deletion operations. Unlike sequential data structures, there is complete absence of data movement of neighboring elements during the execution of this operation.

ii) Efficient use of storage memory. The operation and management of linked data structures are less time to create memory.

**2.7 Single linked list and Chains**

Linked List: A Linked List is a non-linear data structure that allocates non-sequential memory locations in main memory of computer. These locations can’t easy to identify the particular element into the list. It is a dynamic data structure because we can insert or delete the elements at runtime (size is not fixed). To locate the elements from a linked list, we refer address of each memory location. Linked List are mainly divided into three types like:

Singly Linked List, 2. Doubly Linked List, 3. Circular Linked List

Singly Linked List : A Singly linked list is a linear data structure. In this element refers “node”, it contains two fields such as data & link.

*link*

data

Data : It contains actual value

Link : It contains next node address

address

The representation of Singly Linked List in memory: Every node is basically a chunk of memory that carries an address. A set of data elements are represented using a linked list, each data element is represented by a node.

NULL

30

103

102

10

101

103

20

102

head

(101)

In the above diagram we use a special pointer is “head”, it contains always first node address and also last node link contains “NULL” pointer. Singly Linked List provides mainly 4 types of operations like 1. Creation 2. Traversing 3. Insertion 4. Deletion

**2.8 Representing chains in C**

Creation: To create a singly linked list, first we check the condition that the head is NULL or not. If it is NULL, then assign created a new node address to the head other wise assign created new node address to the previous node.

Algorithm: struct node

{ int data;

struct node \*next;

};

struct node \*p,\*temp,\*head = NULL;

create (num)

{ temp = (struct node \*) malloc (sizeof(struct node ))

if(head == NULL)

{ temp->data = num;

temp->next = NULL;

head = temp;

}

else

{

temp->data = num;

temp->next = NULL;

p->next = temp;

p = temp;

}

}

Traversing: Traverse means to display list of the elements. In Singly Linked List traverse can possible only one direction because node contains next node address. It means form first node to last node.

Algorithm: traverse ( )

{ node \*a = head;

while( a!=NULL )

{ printf(“%d”, a->data);

a = a->next;

} pirntf(“NULL”); }

Insertion:Suppose if we want to insert an element in to the list by using three facts like

To insert an element at beginning of the list

To insert an element at end of the list

To insert an element after the specified node

To insert an element at beginning of the list: Suppose if we want to insert an element at beginning of the list, first create new node and next assign that address to the head because it is a first node.

Algorithm: insert-begin(num)

{ temp = (struct node \*) malloc (sizeof(struct node ))

temp->data = num;

temp->next= head;

head = temp;

}

Before Insertion:

head

(101)

NULL

30

103

102

10

101

103

20

102

temp

NULL

80

108

After Insertion:

NULL

30

103

102

10

101

103

20

102

head

(108)

101

80

108

temp

To insert an element at end of the list: Suppose if we want to insert an element at end of the list, first crate new node and next assign that address to the previous node.

Algorithm: insert-end(num)

{

temp = (struct node \*) malloc (sizeof(struct node ))

temp->data = num;

temp->next = NULL;

p->next = temp;

p = temp;

}

Before Insertion:

head

(101)

temp

NULL

30

103

NULL

80

108

102

10

101

103

20

102

After Insertion:

head

(101)

temp

102

10

101

103

20

102

108

30

103

NULL

80

108

To insert an element after specified node: Suppose if we want to insert an element after specified node, first we find out specified node is there or not. If it is found then insert new node other wise display element not found.

Algorithm: insert-after(ele, num)

{

struct \*a = head;

while( a!=NULL )

{

if( a->data == ele )

{

temp = (struct node \*) malloc (sizeof(struct node ));

temp->data = num;

temp->next = a->next;

a->next = temp;

}

else

a = a->next;

}

}

Before Insertion:

NULL

30

103

NULL

80

108

102

10

101

103

20

102

head

(101)

temp

After Insertion:

102

10

101

108

20

102

NULL

30

103

head

(101)

103

80

108

temp

Deletion: Suppose if we want to delete an element form the list, first we checks deleted element is there or not. If it is found then delete that element other wise display element not found.

Algorithm:

del(ele)

{

struct \*b,\*a = head;

if( a->data == ele )

{

head = a->next;

delete a; }

else

{

b = a;

a = a->next;

while( a!=NULL )

{

if( a->data == ele )

{

b->next = a->next;

delete a;

}

else

{

b = a;

a= a->next;

}

}

}

}

Before Deletion:

102

10

101

103

20

102

head

(101)

104

30

103

NULL

40

104

New node

deleted node

After Deletion:

102

10

101

104

20

102

head

(101)

104

30

103

NULL

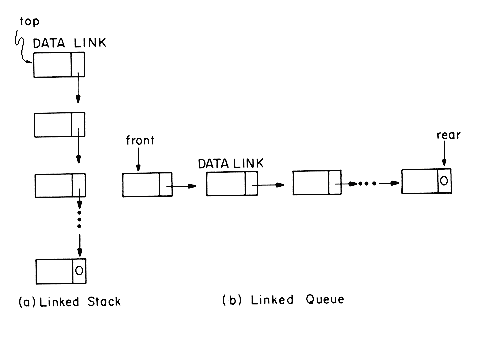
40

104

New node

deleted node

**2.9 Linked stacks and Queues(stacks and queues using Linked list Programs)**

****

**Linked Stack:**

void push(int value)

{

struct Node \*newNode;

newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

if(top == NULL)

newNode->next = NULL;

else

newNode->next = top;

top = newNode;

printf("\nInsertion is Success!!!\n");

}

void pop()

{

if(top == NULL)

printf("\nStack is Empty!!!\n");

else{

struct Node \*temp = top;

printf("\nDeleted element: %d", temp->data);

top = temp->next;

free(temp);

}

}

void display()

{

if(top == NULL)

printf("\nStack is Empty!!!\n");

else{

struct Node \*temp = top;

while(temp->next != NULL){

printf("%d--->",temp->data);

temp = temp -> next;

}

printf("%d--->NULL",temp->data);

}

}

**Linked Queue**

void insert(int value)

{

struct Node \*newNode;

newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

newNode -> next = NULL;

if(front == NULL)

front = rear = newNode;

else{

rear -> next = newNode;

rear = newNode;

}

printf("\nInsertion is Success!!!\n");

}

void delete()

{

if(front == NULL)

printf("\nQueue is Empty!!!\n");

else{

struct Node \*temp = front;

front = front -> next;

printf("\nDeleted element: %d\n", temp->data);

free(temp);

}

}

void display()

{

if(front == NULL)

printf("\nQueue is Empty!!!\n");

else{

struct Node \*temp = front;

while(temp->next != NULL){

printf("%d--->",temp->data);

temp = temp -> next;

}

printf("%d--->NULL\n",temp->data);

}

}

**2.10 Additional List Operations**

**operations for chains:**  One useful function is one that concatenates tow chains ptr1 and ptr2.

**Program:Concatenating single linked list**

listpointer concatenate(listpointer ptr1, listpointer ptr2)

{

/\* Produce a new list that contains the list ptr1 followed by the list ptr2. The list pointed to by ptr1 is changed permanently \*/

listpointer temp;

/\* check for empty lists \*/

if(!ptr1) return ptr2;

if(!ptr2) return ptr1;

/\* neither list is empty, find end of first list \*/

for(temp = ptr1;temp-> link; temp = temp -> link)

/\* link end of first to start of second \*/

temp->link = ptr2;

}

**Operations for circular linked list**

Program: Finding the length of a circular list

int length(listPointer last)

{

/\* find the length of the circular list last \*/

listpointer temp;

int count = 0;

if(last)

{

temp = last;

do

{

count++;

temp = temp->link;

} while(temp!= last);

}

return count;

}

2.11 **Doubly Linked List**

Doubly linked list is a collection of linear data items. It is dynamic data structure because elements are inserted and deleted at runtime. In doubly linked list element refers node. Node contains three fields like left link, data and right link.

1. Left link : It contains previous node address.

2. Data : It contains actual value.

3. Right link : It contains next node address.

The representation of doubly linked list is:

head

(101)

NULL

10

102

101

101

20

103

102

102

30

NULL

103

tail

(103)

In the above representation we use two special pointers like head & tail. Head is always pointing first node address and tail is always pointing last node address.

Difference between Singly linked list & Doubly linked list:

In singly linked list node contains two fields but in doubly linked list node contains three fields. In singly linked list traverse can possible only one direction because node contains next node address but in doubly linked list traverse can possible both direction because node contains previous node and next node address. The structure of node is

Ex: struct node

{

int data;

node \*llink, \*rlink;

};

Doubly linked list provides mainly four types of operations like:

1. Creation 2. Traverse 3. Insertion 4. Deletion

Creation: In this we first check the condition head is NULL or not. If it is NULL then create a new node and assign that address to the head & tail. Other wise create a new node and assign that address to the tail.

Algorithm:

Algorithm create ( num )

{ temp = (struct node \*) malloc (sizeof(struct node ));

if ( head == NULL )

{

temp->data =num;

temp->lnext = NULL;

temp->rnext = NULL;

head = tail = temp;

}

else

{

temp = (struct node \*) malloc (sizeof(struct node ));

temp->lnext = tail;

temp->data = num;

temp->rnext = NULL;

tail->rnext = temp;

tail = temp;

}

}

Traverse: Traverse means to display the list of elements. In this traverse can possible both direction, from first node to last node called “forward traverse” and from last node to first node called “backward traverse”.

Forward Traverse Backward Traverse

Algorithm ftraverse( ) Algorithm btraverse( )

{ {

temp = head; temp = tail;

while( temp!=NULL ) while( temp!=NULL )

{ {

printf(“%d”,temp->data; printf(“%d”,temp->data;

temp = temp->rnext; temp = temp->lnext;

} }

} }

Insertion:Suppose if we want to insert an element into the list by using three methods like

To insert an element at beginning of the list

To insert an element at end of the list

To insert an element after the specified node

To insert and element at beginning of the list: In this we first create a new node and assign that address to the head pointer because it is a first node.

Algorithm: insert-begin( num )

{

temp = (struct node \*) malloc (sizeof(struct node ));

temp->lnext = NULL;

temp->data = num;

temp->rnext = head;

head->lnext = temp;

head = temp;

}

To insert an element at end of the list: In this we first create a new node and assign that address to the tail pointer because it is a last node.

Algorithm: insert-end( num )

{

temp = (struct node \*) malloc (sizeof(struct node ));

temp->lnext = tail;

temp->data = num;

temp->rnext = NULL;

tail->lnext = temp;

tail = temp;

}

To insert an element after the specified node:

In this we first check the condition specified node is there or not. If it is found then create a new node and assign that address to the previous and next node otherwise display element not found.

Algorithm: insert-after( num, ele )

{

a = head;

while( a!=NULL )

{

if( a->data == ele )

{

b = a->rnext;

temp = (struct node \*) malloc (sizeof(struct node ));

temp->lnext = a;

temp->data = num;

temp->rnext = b;

a->rnext = b->lnext = temp;

}

}

}

Deletion: Suppose if we want to delete an element from the list, first to check the condition deleted element is there or not. If it is found then delete that node otherwise display element not found.

Algorithm: delete( num )

{

a = head;

if( head->data == num )

{

head = head->rnext;

head->lnext = NULL;

delete a;

}

else

{

while( a!=NULL )

{

if( a->data == num )

{

c = a->lnext;

b = a->rnext;

c->rnext = b;

b->lnext = c;

delete a;

}

}

}

}

Advantages and disadvantages of a doubly linked list:

Advantages:

The availability of two links Llink and Rlink permit forward and backward movement during the processing of the list.

The deletion of node X from the list calls only for the value of X to be known.

Disadvantages:

1.The disadvantage of doubly linked list is its memory require two links.

Circular Linked List Circular Linked List is a collection of linear data items. It is dynamic data structure because elements are inserted and deleted at runtime. It provides mainly two types like 1. Circular Singly Linked List 2. Circular Doubly Linked List

Circular Singly Linked List: In this element refers node. Node contains two fields like data and link.

1. Data : It contains actual value

2. Link : It contains next node address

The representation of Circular Singly Linked List is:

101

30

103

102

10

101

103

20

102

head

(101)

In the above representation we use a special pointer is head, it always pointing first node address. And also last node link contains first node address because last node is always follows first node.

Circular Doubly Linked List: In this element refers node. Node contains three fields like left link, data and right link.

1. Left Link : It contains previous node address

2. Data : It contains actual value

3. Right Link : It contains next node address

The representation of Circular Doubly Linked List is:

103

10

102

101

101

20

103

102

102

30

101

103

tail

(103)

head

(101)

In the above representation we use types of special pointers like head and tail. Head is always pointing first node address and tail is always pointing last node address. And also last node right link contains first node address and first node left link contains last node address.