**UNIT –II**

**Linked Lists**

**Learning Material**

**2.1 LINKED LISTS-BASIC CONCEPTS**

* In arrays once memory is allocated, it can’t extended any more. So array is known as *static Data Structure*.
* Linked List is *dynamic* Data Structure, where amount of memory required can be vary during it use.

**Definition:** A Linked List is an ordered collection of homogeneous data elements called nodes. Where the linear order is maintained by means of *links* (or) *pointers*.

* The representation of node is as follows:

Data Link

***Node: an element in Linked List***

* A node consists of two parts. i.e. **Data part** and **Link part**.
* The data part contains actual data to be represented.
* The link part is also referred as address field, which contains address of the next node.

**2.1.1 Representation of Linked List in memory**

There are two ways to represent a Linked List in memory.

1. Static memory allocation using arrays.

2. Dynamic allocations using pool of storage.

**2.1.1.1 Static memory allocation using arrays**

* This representation maintains two arrays.
* One array used for Data and another array is used for link.

**X** 45

102 43

105 46

103 44

header

N1

N2

N4

N3

***SLL with 4 nodes***

101 **X**

* The static representation for a linked list shown in the above figure is given in the following figure.

|  |  |  |  |
| --- | --- | --- | --- |
| **Address** | **Array 1 (Data)** |  | **Array 2 (Link)** |
| 41 |  |  |  |
| 42 |  |  |  |
| 43 | 105 |  | 46 |
| 44 | 101 |  | **X** |
| 45 | 102 |  | 43 |
| 46 | 103 |  | 44 |
| 47 |  |  |  |

**Static Representation of SLL using arrays**

**2.1.1.2 Dynamic allocations using pool of storage**

* The efficient way of representation of linked list is using pool of storage.
* In this method memory bank, memory manager and garbage collector is available.

**Memory bank:** is a collection of free memory spaces.

**Memory manager:** is a program.

* Whenever a linked list requires a node, then request is placed to memory manager.
* If the required node is available in the memory bank, then that node is send to caller.
* If the required node is not available in the memory bank, then memory manager send NULL to caller.

**Garbage collector**: collect the unused nodes in the linked list and send back to memory bank.

**Types of linked lists**

1. Single Linked List (SLL)

2. Double Linked List (DLL)

3. Circular Linked List (CLL)

**2.2 SINGLE LINKED LIST**

* In SLL, each node contains only one link, which points to the next node in the list.
* The pictorial representation of SLL is as follows.

**X** N1 **NNNNNNNNNN1**

102 N2

105 N3

103 N4

101 **X**

header

N1

N2

N4

N3

SLL with 4 nodes

* Here header is an empty node, i.e. data part is NULL, represented by X mark.
* The link part of the header node contains address of the first node in the list.
* In SLL, the last node link part contains NULL.
* In SLL we can move from left to right only. So SLL is called as one way list.
* If a SLL is empty, then the link part of the header node is NULL.

**X X**

header

**Empty Single Linked List**

**Operations On Single Linked List**

1.Traversing a list

2. Insertion of a node in to SLL

3. Deletion of a node from SLL

**2.2.1 Traversing a list**

* Traversing a SLL means, visit every node in the list starting from first node to the last node.

**Algorithm SLL\_Traverse(header)**

**Input:** header is the pointer to header node.

**Output:** Visiting of every node in SLL.

1. ptr = header
2. if(ptr->link==NULL)
3. print “SLL is empty”

3. else

a) while(ptr🡪link != NULL)

i) ptr = ptr🡪link

ii) print "ptr🡪data"

1. end while

3. end if

**End SLL\_Traverse**

**2.2.2 Insertion of a node into SLL**

The Insertion of a node in to SLL can be done in various positions.

i) Insertion of a node into SLL at beginning.

ii) Insertion of a node into SLL at ending.

iii) Insertion of a node into SLL at any position.

* For insertion of a node into SLL, we must get node from memory bank.
* The procedure for getting node from memory bank is as follows:

**Procedure for getnewnode( )**

1. Check for availability of node in memory bank

2. if ( AVAIL = = NULL)

a) print "Required node is not available in memory"

b) return NULL

3. else

a) return address of node to the caller

4. end if

**End Procedure for getnewnode**

**2.2.2.1 Insertion of a node into SLL at beginning**

* Insertion of a node into SLL at beginning means insert new node after header node.

**Algorithm SLL\_Insert\_Begin(header,x)**

**Input:** *header* is a pointer to the header node, *x* is data part of new node to be insert.

**Output:** SLL with new node inserted at beginning.

1. new = getnewnode( )
2. if(new==NULL)
3. print “node not creates”

3. else

a) new🡪link = header🡪link /\* 1 \*/

b) header🡪link = new /\* 2 \*/

c) new🡪data = x

4. end if

**End SLL\_Insert\_Begin**

**X**

**X**

header

N1

N2

N3

new

**Before Insertion**

**X**

**X**

header

new

N1

N2

N3

**After Insertion**

1. Link part of new node is replaced with address of first node in list, i.e. link part of header node.
2. Link part of header node is replaced with new node address.

**2.2.2.2 Insertion of a node into SLL at ending**

* To insert a node into SLL at ending first we need to traverse to last node, then insert as new node as last node.

**Algorithm SLL\_Insert\_Ending(header,x)**

**Input:** *header* is a pointer to the header node, *x* is data part of new node to be insert.

**Output:** SLL with new node at ending.

1. new = getnewnode()
2. if(new==NULL)
3. print “node not creates”

3. else

a) ptr = header

b) while(ptr🡪link != NULL)

i) ptr = ptr🡪link

c) end loop

d) ptr🡪link = new /\* 1 \*/

e) new🡪link = NULL /\* 2 \*/

f) new🡪data = x

3. end if

**End SLL\_Insert\_Ending**

**X**

**X**

header

N1

N2

N3

new

**Before Insertion**

**X**

**X**

header

N1

N2

N3

new

**After Insertion**

ptr

ptr

NULL

1. Previous last node link part is replaced with address of new node.
2. Link part of new node is replaced with NULL, because new node becomes the last node.

**2.2.2.3 Insertion of a node into SLL at any position.**

* For insertion of a node at any position in SLL, a key value is specified. Where key being the data part of a node, after this node new node has to be inserting.

**Algorithm SLL\_Insert\_ANY(header,x,key)**

**Input:** *header* is a pointer to the header node, *x* is data part of new node to be inserting, *key* is the data part of a node, after this node we want to insert new node.

**Output:** SLL with new node inserted after the node with data part as specified key value.

1. new = getnewnode( )
2. if(new==NULL)
3. print “node not creates”

3. else

a) ptr = header

b) while(ptr🡪link != NULL && ptr🡪data != key)

i) ptr = ptr🡪link

c) end loop

d) if(ptr🡪link = = NULL && ptr🡪data != key)

i) print"Required node with data part as key value is not available, so unable to process"

e) else

i) new🡪link = ptr🡪link /\* 1 \*/

ii) ptr🡪link = new /\* 2 \*/

iii) new🡪data = x

f) end if

4. end if

**End SLL\_insert\_ANY**

**X**

100

20

30 **X**

header

N1

N2

N3

new

**Before Insertion**

**X**

100

20

30 **X**

header

N1

N2

N3

new

**After Insertion**

ptr

Consider key is 20

1. Link part of new node is replaced by the address of next node. i.e. in the above example N3 becomes next node for newly inserting node.
2. Link part of previous node is replaced by the address of new node. i.e. in the above example N2 becomes previous node for newly inserting node.

**2.2.3 Deletion of a node from SLL**

* The deletion of a node in from SLL can be done in various positions.

i) Deletion of a node from SLL at beginning.

ii) Deletion of a node from SLL at ending.

iii) Deletion of a node from SLL at any position.

**2.2.3.1 Deletion of a node from SLL at beginning**

* Deletion of a node from SLL at beginning means, delete the node which is after the header node.

**Algorithm SLL\_Delete\_Begin(header)**

**Input:** *header* is a pointer to the header node.

**Output:** SLL with node deleted at Beginning.

1. if(header🡪link = = NULL)

a) print "SLL is empty, so unable to delete node from list"

2. else /\*SLL is not empty\*/

a) ptr = header🡪link /\* ptr points to first node into list\*/

b) header🡪link=ptr🡪link /\* 1 \*/

c) print "Deleted node is" ptr🡪data

d) free(ptr) /\*send back deleted node to memory bank\*/

3. end if

**End SLL\_Delete\_Begin**

**X**

header

N1

N2

N3

**Before Deletion**

**X**

**X**

header

N2

N3

N4

**After Deletion**

**X**

N4

ptr

1. Link part of the header node is replaced with address of second node. i.e. address of second node is available in link part of first node.

**2.2.3.2 Deletion of a node from SLL at ending**

* To delete a node from SLL at ending, first we need to traverse to last node in the list. After reach the last node in the list, last but one node link part is replaced with NULL.

**Algorithm SLL\_Delete\_End (header)**

**Input:** *header* is a pointer to the header node.

**Output:** SLL with node deleted at ending.

1. if(header🡪link == NULL)

a) print"SLL is empty, so unable to delete the node from list"

2. else /\*SLL is not empty\*/

a) ptr = header /\*ptr initially points to header node\*/

b) while(ptr🡪link!=NULL)

i) ptr1 = ptr

ii) ptr = ptr🡪link

c) end loop

d) ptr1🡪link = NULL /\* 1 \*/

e) print "Deleted node is" ptr🡪data

f) free(ptr) /\*send back deleted node to memory bank\*/

3. end if

**End SLL\_Delete\_End**

**X**

**X**

header

N1

N2

N3

N4

**Before Deletion**

**X**

**X**

header

N1

N2

N3

**After Deletion**

ptr1

ptr1

NULL

ptr

1. Link part of last but one node is replaced with NULL. Because after deletion of last node in the list, last but one node become the last node.

**2.2.3.3 Deletion of a node from SLL at any position**

* For deletion of a node from SLL at any position, a key value is specified. Where key being the data part of a node to be deleting.

**Algorithm SLL\_Delete\_ANY (header,key)**

**Input:** *header* is a pointer to the header node, key is the data part of the node to be delete.

**Output:** SLL with node deleted at Any position. i.e. Required element.

1. if(header🡪link = = NULL)

a) print "SLL is empty, so unable to delete the node from list"

2. else /\*SLL is not empty\*/

a) ptr =header /\*ptr initially points to header node\*/

b) while(ptr🡪link != NULL && ptr🡪data != key)

i) ptr1 = ptr

ii) ptr=ptr🡪link

c) end loop

d) if(ptr🡪link = = NULL && ptr🡪data != key)

i) print "Required node with data part as key value is not available"

e) else /\* node with data part as key value available \*/

i) ptr1🡪link = ptr🡪link /\* 1 \*/

ii) print "Deleted node is" ptr🡪data

iii) free(ptr)

f) end if

3. end if

**End SLL\_Delete\_ANY**

**X**

100

20

30

header

N1

N2

N3

**Before Deletion**

**X**

100

20

18 **X**

header

N1

N2

N4

**After Deletion**

ptr1

18 **X**

N4

ptr

ptr1

Consider key value is 30

1. Previous node link part is replaced with address of next node in the list. i.e. in the above example N2 becomes the previous node and N4 becomes the next node for the node to be delete.

**2.3 CIRCULAR LINKED LIST**

* A linked list where last node link part points to *header* node is called as Circular Linked List.

**X**

header

N1

N2

N3

N4

**Circular Linked List**

* If a CLL is empty, then the link part of the header node points to itself.

**X**

header

**Empty Circular Linked List**

**Operations on Circular Linked List**

1. Traversing list

2. Insertion of a node in to CLL

3. Deletion of a node from CLL

**2.3.1 Insertion of a node in to CLL**

* The Insertion of a node in to CLL can be done in various positions.

i) Insertion of a node into CLL at beginning.

ii) Insertion of a node into CLL at ending.

iii) Insertion of a node into CLL at any position.

**2.3.1.1 Insertion of a node into CLL at beginning**

* Insertion of a node into CLL at beginning means insert new node after header node.

**Algorithm CLL\_Insert\_Begin(header,x)**

**Input:** *header* is pointer to header node, x is data part of new node to be inserting.

**Output:** CLL with new node inserted at beginning.

1. new = getnewnode( )

2. new🡪link = header🡪link /\* 1 \*/

3. header🡪link = new /\* 2 \*/

4. new🡪data = x

**End CLL\_Insert\_Begin**

header

**X**

N1

N2

N3

new

**Before Insertion**

**X**

header

new

N1

N2

N3

**After Insertion**

1. Link part of new node is replaced with address of first node in list, i.e. link part of header node.
2. Link part of header node is replaced with new node address.

**2.3.1.2 Insertion of a node into CLL at ending**

* To insert a node into CLL at ending first we need to traverse to last node, then insert as new node as last node.

**Algorithm CLL\_ Insert\_End(header,x)**

**Input:** *header* is pointer to header node, *x* is data part of new node to be inserting.

**Output:** CLL with new node inserted at ending.

1. new = getnewnode()

2. ptr = header

3. while(ptr🡪link != header)

a) ptr = ptr🡪link

4. end loop

5. ptr🡪link = new

6. new🡪link = header

7. new🡪data = x

**End CLL\_Insert\_End**

header

**X**

N1

N2

N3

new

**Before Insertion**

**X**

header

N1

N2

N3

new

**After Insertion**

header node address

1. Previous last node link part is replaced with address of new node.
2. Link part of new node is replaced with address of header node, because new node becomes the last node.

**2.3.1.3 Insertion of a node into CLL at any position**

* For insertion of a node at any position in CLL, a key value is specified. Where key being the data part of a node, after this node new node has to be inserting.

**Algorithm CLL\_Insert\_ANY(header,x,key)**

**Input:** header is pointer to header node, x is data pat of new node to be insert, key is the data part of a node, after this node we want to insert new node.

**Output:** CLL with new node inserted after the node with data part as specified key value.

1. new = getnewnode()

2. ptr=header

3. while(ptr🡪link != header && ptr🡪data != key)

a) ptr = ptr🡪link

4. end loop

5. if(ptr🡪link = = header && ptr🡪data != key)

a) print "required node with data part as key value is not available, so unable to process"

6. else

a) new🡪link = ptr🡪link /\* 1\*/

b) ptr🡪link = new /\* 2 \*/

c) new🡪data = x

7. end if

**End CLL\_Insert\_ANY**

**X**

100

20

30

header

N1

N2

N3

new

**Before Insertion**

**X**

100

20

30

header

N1

N2

N3

new

**After Insertion**

ptr

1. Link part of new node is replaced by the address of next node. i.e. in the above example N3 becomes next node for newly inserting node.
2. Link part of previous node is replaced by the address of new node. i.e. in the above example N2 becomes previous node for newly inserting node.

**2.3.2 Deletion of a node from CLL**

* The Deletion of a node from CLL can be done in various positions.

i) Deletion of a node from CLL at beginning

ii) Deletion of a node from CLL at ending

iii) Deletion of a node from CLL at any position

**2.3.2.1 Deletion of a node from CLL at beginning**

* Deletion of a node from CLL at beginning means, delete the node which is after the header node.

**Algorithm CLL\_Delete\_Begin(header)**

**Input:** *header* is pointer to header node.

**Output:** CLL with node deleted at Beginning.

1. if(header🡪link = = header)

a) print "CLL is empty, so unable to delete node from list"

2. else /\*DLL is not empty\*/

a) ptr = header🡪link /\* ptr points to first node into list\*/

b) header🡪link = ptr🡪link /\* 1 \*/

c) print "deleted node is " ptr🡪data

d) free(ptr) /\*send back deleted node to memory bank\*/

3. end if

**End CLL\_Delete\_Begin**

**X**

header

N1

N2

N3

**Before Deletion**

**X**

header

N2

N3

N4

**After Deletion**

N4

ptr

1. Link part of the header node is replaced with address of second node. i.e. address of second node is available in link part of first node.

**2.3.2.2 Deletion of a node from CLL at ending**

* To delete a node from CLL at ending, first we need to traverse to last node in the list. After reach the last node in the list, last but one node link part is replaced with header node address.

**Algorithm CLL\_Delete\_End (header)**

**Input:** *header* is pointer to header node.

**Output: C**LL with node deleted at ending.

1. if(header🡪link = = header)

a) print "CLL is empty, so unable to delete the node from list"

2. else /\* CLL is not empty\*/

a) ptr=header /\*ptr initially points to header node\*/

b) while(ptr🡪link!=header)

i) ptr1=ptr

ii) ptr=ptr🡪link

c) end loop

d) ptr1🡪link=header /\* 1 \*/

e) print "deleted node is" ptr🡪data

f) free(ptr) /\*send back deleted node to memory bank\*/

3. end if

**End CLL\_Delete\_End**

**X**

N1

N2

N3

N4

**Before Deletion**

**X**

header

N1

N2

N3

**After Deletion**

ptr1

ptr1

ptr

header

header node address

1. Link part of last but one node is replaced with address of header node. Because after deletion of last node in the list, last but one node become the last node.

**2.3.2.3 Deletion of a node from CLL at any position**

* For deletion of a node from CLL at any position, a key value is specified. Where key being the data part of a node to be deleting.

**Algorithm CLL\_Delete\_ANY(header,key)**

**Input:** *header* is pointer to header node, key is the data part of the node to be deleting.

**Output:** CLL with node deleted at Any position. i.e. Required element.

1. if(header🡪link = = header)

a) print "CLL is empty, so unable to delete the node from list"

2. else /\*CLL is not empty\*/

a) ptr=header /\*ptr initially points to header node\*/

b) while(ptr🡪link != header && ptr🡪data != key)

i) ptr1 = ptr

ii) ptr = ptr🡪link

c) end loop

d) if(ptr🡪link = = header && ptr🡪data!=key)

i) print "Required node with data part as key value is not available"

e) else

i) ptr1🡪link = ptr🡪link /\* 1 \*/

ii) print "deleted node is " ptr🡪data

iii) free(ptr)

f) end if

3. end if

**End CLL\_Delete\_ANY**

**X**

100

20

30

header

N1

N2

N3

**Before Deletion**

**X**

100

20

18

header

N1

N2

N4

**After Deletion**

ptr1

18

N4

ptr

ptr1

Consider key is 30

* + - 1. Previous node link part is replaced with address of next node in the list. i.e. in the above example N2 becomes the previous node and *N4* becomes the next node for the node to be deleting.

**2.4 DOUBLE LINKED LIST**

* In a SLL one can move from the header node t o any node in one direction only. i.e. from left to right.
* A DLL is a two way list. Because one can move either from left to right or right to left.
* In DLL, each node maintains two links.

Data

RLink

LLink

***Structure of a node in DLL***

* Here LLink refers *Left Link* and RLink refers *Right Link*.
* The LLink part of a node in DLL always points to the previous node. i.e. LLink part of a node Consists address of previous node.
* The RLink part of a node in DLL always points to the next node. i.e. RLink part of a node Consists address of next node.
* If a DLL is empty, then the llink part and rlink of the header node is NULL.

X X X

**Empty Double Linked List**

header

**Operations on Double Linked List**

1. Traversing list

2. Insertion of a node in to DLL

3. Deletion of a node from DLL

**2.4.1 Traversing a list**

* Traversing a DLL means, visit every node in the list starting from first node to the last node.

**Algorithm DLL\_Traverse(header)**

**Input:** header is the pointer to header node.

**Output:** Visiting of every node in DLL.

1. ptr = header

2. while(ptr🡪rlink != NULL)

a) ptr = ptr🡪rlink

b) print "ptr🡪data"

3. end loop

**End DLL\_Traverse**

**2.4.2 Insertion of a node in to DLL**

* The Insertion of a node in to DLL can be done in various positions.

i) Insertion of a node into DLL at beginning

ii) Insertion of a node into DLL at ending

iii) Insertion of a node into DLL at any position

* For insertion of a node into DLL, we must get node from memory bank. The procedure for getting node from memory bank is same as getting node for SLL from memory bank.

**2.4.2.1 Insertion of a node into DLL at beginning**

* Insertion of a node into DLL at beginning means insert new node after header node.

**Algorithm DLL\_Insertion\_Begin(header,x)**

**Input:** *header* is a pointer to the header node, *x* is data part of new node to be inserting.

**Output:** DLL with new node at begin.

1. new = getnewnode()

2. if(header🡪rlink == NULL)

a) header🡪rlink = new

b) new🡪llink = header

c) new🡪rlink = NULL

d) new🡪data = x

3. else

a) ptr = header🡪rlink

b) new🡪rlink = ptr /\* 1 \*/

c) new🡪llink = header /\* 2 \*/

d) header🡪rlink = new /\* 3 \*/

e) ptr🡪llink = new /\* 4 \*/

f) new🡪data = x

4. end if

**End DLL\_Insertion\_Begin**

X X

X

header

N1

N2

N3

X X

X

header

new

N1

N2

N3

**Before Insertion**

After Insertion

new

ptr

ptr

1. *Rlink* part of new node is replaced with the address of first node in the DLL. i.e. address of first node is available in *Rlink* part of header node.
2. *Llink* part of new node is replaced with the address of header.
3. *Rlink* part of header node is replaced with the address of new node.
4. *Llink* part of previous first node is replaced with the address of new node.

**2.4.2.2 Insertion of a node into DLL at ending**

* To insert a node into DLL at ending, first we need to traverse to last node, then insert as new node as last node.

**Algorithm DLL\_ Insertion\_Ending(header,x)**

**Input:** *header* is a pointer to the header node,*x* is data part of new node to be inserting.

**Output:** DLL with new node inserted at the ending.

1. new = getnewnode()

2. ptr = header

3. while(ptr🡪rlink != NULL)

a) ptr = ptr🡪rlink

4. end loop

5. ptr🡪rlink = new /\* 1 \*/

6. new🡪llink = ptr /\* 2 \*/

7. new🡪rlink = NULL /\* 3 \*/

8. new🡪data = x

**End DLL\_Insertion\_Ending**

X X

X

header

N1

N2

N3

new

NULL

X X

X

header

N1

N2

N3

new

**Before Insertion**

**After Insertion**

ptr

ptr

1. *RLink* part of last node in the DLL is replaced with address of new node.
2. *LLink* part of new node is replaced with address of previous last node.
3. *RLink* part of new node is replaced with NULL. Because newly inserted node becomes the last node in the list.

**2.4.2.3 Insertion of a node into DLL at any position**

* For insertion of a node at any position in DLL, a key value is specified. Where key being the data part of a node, after this node new node has to be inserting.

**Algorithm DLL\_Insertion\_ANY(header,x,key)**

**Input:** *header* is a pointer to the header node, *x* is data pat of new node to be inserting, key is the data part of a node, after that node new node is inserted.

**Output:** DLL with new node inserted after the node with data part as specified key value.

1. new = getnewnode()

2. ptr = header

3. while(ptr🡪rlink != NULL && ptr🡪data != key)

a) ptr = ptr🡪rlink

4. end loop

5. if(ptr🡪rlink = = NULL && ptr🡪data != key)

a) print "required node with key value was not available"

6. else if(ptr🡪rlink = = NULL && ptr🡪data == key)

a) ptr🡪rlink = new

b) new🡪llink = ptr

c) new🡪rlink = NULL

d) new🡪data = x

7. else

a) ptr1 = ptr🡪rlink

b) ptr🡪rlink = new /\* 1 \*/

c) new🡪llink = ptr /\* 2 \*/

d) new🡪rlink = ptr1 /\* 3 \*/

e) ptr1🡪llink = new /\* 4 \*/

f) new🡪data = x

8. end if

**End DLL\_Insertion\_ANY**

N2

N3

**20**

**30** X

**10**

N1

X X

**10**

**20**

**30** X

header

N1

N2

new

N3

**Before Insertion**

**After Insertion**

new

ptr1

ptr

X X

header

ptr

ptr1

Consider key value is 20

1. *RLink* part of previous node is replaced with address of new node.
2. *LLink* part of new node is replaced with the address of previous node. i.e. in the above example N2 becomes the previous node for newly inserting node.
3. *RLink* part of new node is replaced with the address of next node. i.e. in the above example N3 becomes the next node for newly inserting node.
4. *LLink* part of next node is replaced with address of new node.

**2.4.3 Deletion of a node from DLL**

* The deletion of a node in from DLL can be done in various positions.

i) Deletion of a node from DLL at beginning.

ii) Deletion of a node from DLL at ending.

iii) Deletion of a node from DLL at any position.

**2.4.3.1 Deletion of a node from DLL at beginning**

* Deletion of a node from DLL at beginning means, delete the node which is after the header node.

**Algorithm DLL\_Deletion\_Begin(header)**

**Input:** header is pointer to header node

**Output:** DLL with node deleted at begin

1. if(header🡪rlink = = NULL)

a) print "DLL is empty, not possible to perform deletion operation"

2. else

a) ptr = header🡪rlink

b) ptr1= ptr🡪rlink

c) header🡪rlink = ptr1 /\* 1 \*/

d) ptr1🡪llink = header /\* 2 \*/

e) print "Deleted node is" ptr🡪data

f) free(ptr)

3. end if

**End DLL\_Deletion\_Begin**

ptr

X X

header

N1

N2

N3

X X

X

header

new

N1

N2

N3

**Before Deletion**

**After Deletion**

ptr

X

N4

ptr1

* 1. RLink part of header node is replaced with the address of second node. i.e. address of second node is available RLink part of first node.
  2. LLink part of second node is replaced with the address of header node.

**2.4.3.2 Deletion of a node from DLL at ending**

* To delete a node from DLL at ending, first we need to traverse to last node in the list. After reach the last node in the list, last but one node rlink part is replaced with NULL.

**Algorithm DLL\_Deletion\_End(header)**

**Input:** header is pointer to header node.

**Output:** DLL with deleted node at ending.

1. if(header🡪rlink = = NULL)

a) Print"DLL is empty, not possible to perform deletion operation"

2. else

a) ptr = header

b) while(ptr🡪rlink != NULL)

i) ptr = ptr🡪rlink

c) end loop

d) ptr1 = ptr🡪llink

e) ptr1🡪rlink = NULL /\* 1 \*/

f) print "Deleted node is" ptr🡪data

g) free(ptr)

3. end if

**End DLL\_Deletion\_Ending**

X X

X

header

N1

N2

N3

N4

ptr

X X

X

header

N1

N2

N3

**Before Deletion**

**After Deletion**

ptr1

ptr1

NULL

1. RLink part of last but one node in DLL is replaced with NULL. Because last but one node becomes last node.

**2.4.3.3 Deletion of a node from DLL at any position**

* For deletion of a node from DLL at any position, a key value is specified. Where key being the data part of a node to be deleting.

**Algorithm DLL\_Deletion\_Any(header,key)**

**Input:** *header* is pointer to header node, key is the data part of a node to be delete.

**Output:** DLL without node as data part is key value.

1. if(header🡪rlink = = NULL)

a) print "DLL is empty, not possible for deletion operation"

2. else

a) ptr=header

b) while(ptr🡪rlink != NULL && ptr🡪data != key)

i) ptr = ptr🡪rlink

c) end loop

d) if(ptr🡪rlink = =NULL && ptr🡪data != key)

i) print "required node was not available in list"

e) else if(ptr🡪rlink = = NULL && ptr🡪data == key)

i) ptr1 = ptr🡪llink

ii) ptr1🡪rlink = NULL

iii) print "Deleted node is" ptr🡪data

iv) free(ptr)

e) else

i) ptr1 = ptr🡪llink

ii) ptr2 = ptr🡪rlink

iii) ptr1🡪rlink = ptr2 /\* 1 \*/

iv) ptr2🡪llink = ptr1 /\* 2 \*/

v) print "Deleted node is" ptr🡪data

vi) free(ptr)

f) end if

3. end if

**End DLL\_Deletion\_Any**

ptr

**20**

**30** X

**10**

N1

N2

X X

**10**

**20**

**40** X

header

N1

N2

N4

**Before Deletion**

**After Deletion**

**40** X

ptr1

X X

header

N3

ptr1

ptr2

ptr2

N4

1. *RLink* part of previous node is replaced with the address of next node. i.e. in the above example N2 become the previous node to node to be delete, N4 becomes the next node to node to be delete.
2. *LLink* part of next node is replaced with the address of previous node. i.e. in the above example N2 become the previous node to node to be delete, N4 becomes the next node to node to be delete.

**Comparison between SLL, CLL and DLL**

|  |  |  |
| --- | --- | --- |
| **Single Linked List** | **Circular Linked List** | **Double Linked List** |
| 1. in SLL, each node has one data part and one link part. | 1. in CLL, each node has one data part and one link part. | 1. in DLL, each node has one data part and two link parts. |
| 2. Traverse from Left to Right. | 2. Traverse from Left to Right in circular fashion. | 2. Traverse from Left to Right and Right to Left. |
| 3. Last node link part is NULL. | 3. Last node link part points to address of header node.. | 3. Last node rlink part is NULL. |
| 4. it require less memory when compared with DLL, because SLL has only one link part. | 4. it require less memory when compared with DLL, because CLL has only one link part. | 4. it require more memory when compared with SLL and CLL, because DLL has two link parts. |