Pre-requisites — Module - III — 4 Copy S4 D.B Wilson

Pointers

LINKED LIST

* Each memory location is identified by unique numbers, & these nos are called addresses. These addresses are stored in variables known as pointers.

Pointers — Uses

↳ Support dynamic mem. allocation
↳ has efficient tool for manipulating dynamic D.S's such as stacks, queues, linked list & trees.

declaration

datatype * pointername ;

eg: int *j ;

i	j	k
5	8483	3287
8483	3287	5386

int i ;
int *j ; j = &i;
int **k ;

* * ⇒ value at address operator.
  *j ⇒ value ~~at~~ a~~ddress~~ of i

Arrays & Pointers

↳ *(a+i) represents a[i]
↳ 2-D array:
((a+i)+j) ⇒ a[i][j].

Pointers & Structures

```
struct abc
{
 char name[20];
 int age;
 float sal;
} emp[5], *ptr;
```

ptr = emp;

↳ assign the address of 0th element of emp to ptr.

↳ ptr now pt - to emp[0]. Its members can be accessed by:

ptr → name, ptr → age, ptr → sal.

↳ → is arrow operator / member selection operator.

Memory Allocations In C

2 types:

↳ Compile-time or Static allocation (using arrays)

↳ Run-time or Dynamic memory allocation (using ptrs)

Dynamic Memory Allocation Functions

1. Malloc ()
2. Calloc ()
3. Free ()
4. Realloc ()

Malloc

↳ ptr = (data type)* malloc(no. of elements * size of each element)

↳ Allocates a continuous single block of mem. with specified size in parameter.

Calloc

↳ ptr = (datatype)* calloc(no. of elements, size of each element)

↳ Allocates multiple blocks of memory & each block contains equal size.

Self - Referential Structures

↳ is one which contains a pointer of its own type.

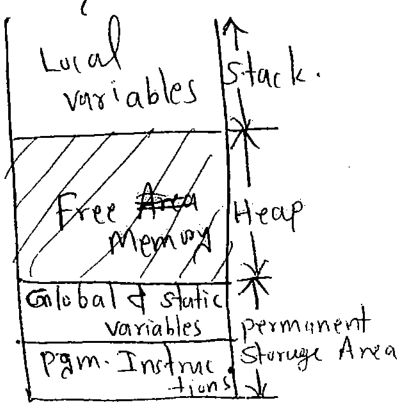
eg:
```
struct student
{
    int rno;
    float marks;
    int subject;
    struct student *ptr;
};
```

↳ In this, the structure student has a pointer ptr, which can point to another student structure.

↳ Self referential structures are used in the area of linear D.S. such as linked lists, stacks & non-linear D.S such as trees, graphs etc.

↳ eg: linked list
```
struct node
{
    int info;
    struct node *link;
}
```

Memory Allocation Process



↳ The program instructions & global & static variables are stored in a region known as permanent storage area.

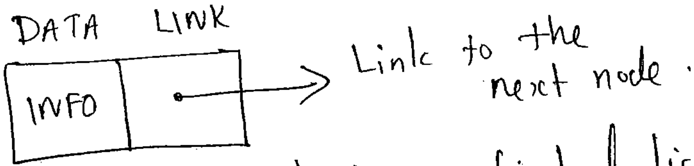
↳ Local variables are stored in another area called stack.

↳ The memory space ie, located b/w these 2 regions is available for dynamic allocation during execution of the program. This free mem. region is called heap.

Introduction

* Array is a very useful D.S. However it has some limitations:
  - ↳ Memory storage space is wasted, as the memory remains allocated to the array throughout the program execution even few nodes are stored.
  - ↳ The size of array can't be changed after its declaration.
  - ↳ The insertion needs some shifting ~~oper~~ procedures.
* These limitations can be overcome by using linked list D.S.
* Linked list is the most commonly used D.S to store similar type of data in memory.
* In array once the memory space is allocated it cannot be extended. That is why this D.S is called Static D.S whereas in linked list, memory space allocated for the elements of the list can be extended at any time. That is why, the linked list is called dynamic D.S.
* In the case of static D.S, each element will have only the data field, whereas in dynamic D.S each element will have 2 types of fields:
  - ↳ Data fields
  - ↳ pointer fields

↳ Data fields → have actual data
↳ pointer (LINK) fields → has the address of next element.



(Node: An element in a linked list)

Definition

* A linked list is an ordered collection of finite homogeneous data elements called nodes where the linear order is maintained by means of links or pointers.

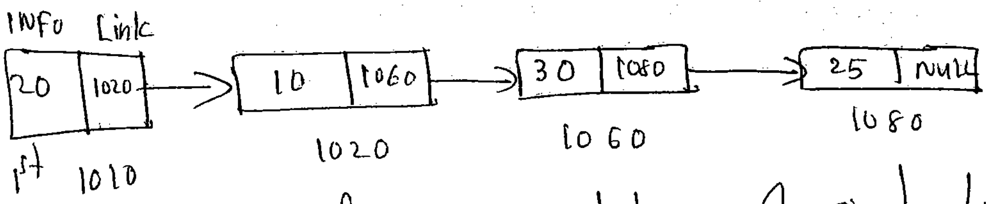
* ie, each node is divided into 2 parts:
    - ↳ 1st part contains the information of the element &
    - ↳ 2nd part, called the link field / next pointer field contains the address of the next node in the list.

* The no. of pointers maintained depending on the requirements & usage. Based on this, linked list are classified into ~~3~~ diff categories:
    1. Linear linked list / singly linked list
    2. Doubly " "
    3. Circular " "
    4. ~~Header " "~~
    4. Circular doubly linked list.

① Linear / Singly Linked List

(*) In linear linked list, each node is divided into 2 parts:
- ↳ 1st part contains the information of the element
- ↳ 2nd part contains the address of the next node in the list.

(*) ie, each node has a single pointer to the next node & in the last node's case a null pointer representing that there are no more nodes in the linked list.



(Logical representation of singly Linked list)

(*) Here one can move from left to right only, ~~this~~ this is why it is also alternatively termed as one way list.

~~② Doubly Linked List~~

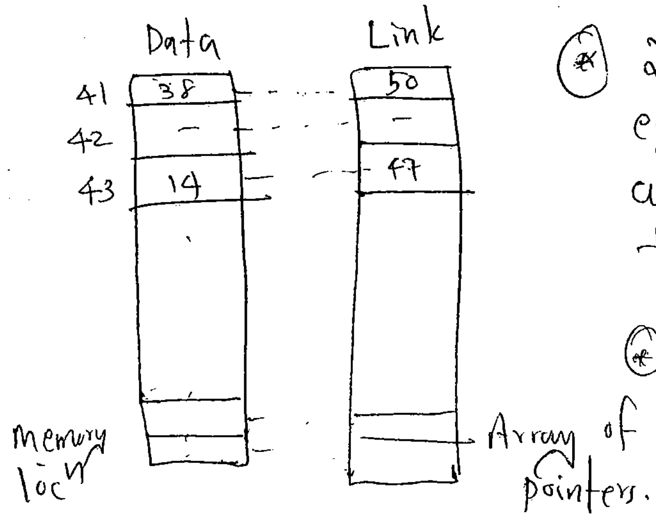
~~(*) Also~~

Representation of a Linked list in Memory

(*) There are 2 ways to represent a linked list in memory:
- ↳ static representation using array.
- ↳ dynamic " using free pool of storage.

Static Representation

(*) It maintains 2 arrays:
- ↳ one array for data & other for links.



* 2 parallel arrays of equal size are allocated which should be sufficient to store the entire linked list.
* So it is not preferable.

2) Dynamic Representation

* The efficient way of representing a linked list is using a free pool of storage.
* In this method, there is a memory bank (collection of free memory spaces) & a memory manager (a pgm.).
* ↳ During the creation of linked list, whenever a node is required the request is placed to the memory manager; memory manager will then search the memory bank for the block requested & if found grants a desired block to the caller.

↳ There is also another program called garbage collector, it plays whenever a node is no more in use, it returns the unused node to the memory bank. ~~Memory~~

↳ Memory bank is also a list of memory space ie, available to a programmer

↳ Such a memory management is known as dynamic memory management

(*) Dynamic representation of linked list uses the dynamic memory management policy.

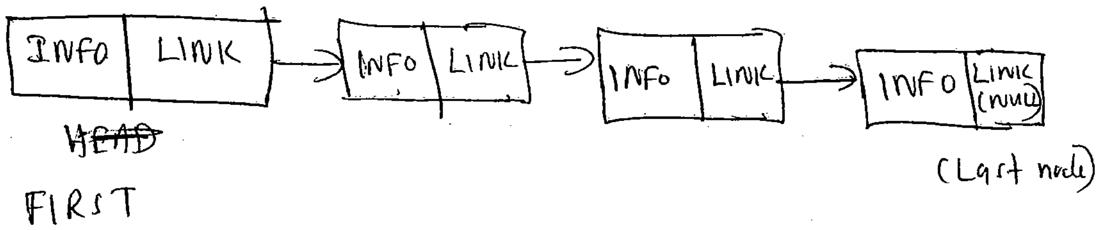
Advantages & Disadvantages of Using Linked List

Advantages

1. Linked lists are dynamic D.S :- They can grow or shrink during the execution of a program.
2. The size is not fixed.
3. Data can store non-continuous memory blocks.
4. Insertion & deletion of nodes are easier & efficient.
5. More complex applications can be easily carried out with linked lists.

Disadvantages

1. More memory :- In the linked list, there is a special field called link field which holds the address of the next node, so linked list requires extra space.



* Linked list is accessed from an external pointer variable say FIRST, which pts to the 1st node in the list.
* The last node of list contains a special value in the LINK field known as NULL & indicates the end of the list.
* The list with no ~~no~~ nodes is called empty list / null list & is denoted by the null pointer in the list pointer variable FIRST.

  ie, FIRST = NULL.

Linked list Creation

* struct node

```
{
  int info;
  struct node *link;
};
struct node *first, *ptr;
```

  ↳ link points to a location in memory which of the struct node.

  ↳ Link contains the address of the next node in the linked list.

* Now apply the dynamic allocation feature to implement linked lists.
* In dynamic memory allocation we use a pointer variable & its size is allocated during the execution of the program.

* malloc () is used here. ~~This~~ It simply finds a free block in memory, based on the memory block size requested & then returns a pointer to it.

struct node * ptr

Ptr = (struct node *) malloc (sizeof (struct node));

* The allocated memory by malloc() is not free automatically. So use free () fn, that frees the allocated space.

free(ptr);

Initializing a node & operating a node

* In order to access the elements of a structure using a pointer to that structure the → operator is used.

ptr → info

ptr → link

ie, scanf ("%d", & ptr → info);

* Initializing link part to null

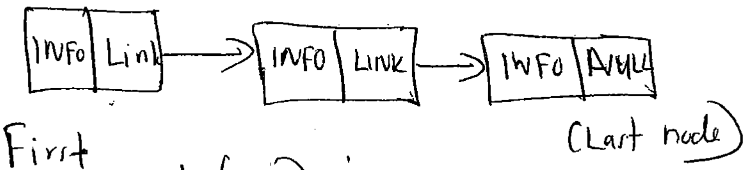
ptr → link = NULL

Operations On Singly Linked List

1. Creation
2. Traversing
3. Insertion
4. Deletion
5. Sorting
6. Searching

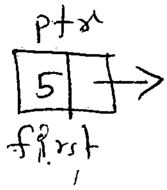
Operations on Singly linked list

① Creation of a Singly Linked List

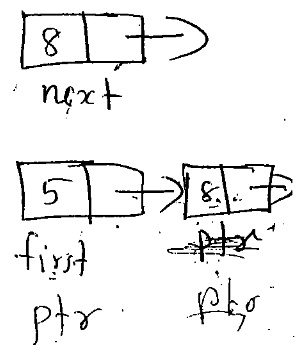



```
void create(int);
struct node
{ int info;
  struct node *link;
} *first;

main()
{  int n;
   printf("Enter how many nodes you want");
   scanf("%d", &n);
   create(n);
   getch();
}
void create(int n)
{ struct node *ptr, *next;
  int i;
  ptr = (struct node *) malloc(sizeof(struct node));
  printf(" Enter first element ");
  scanf("%d", &ptr->info);
  first = ptr;
  for(i=1; i<n; i++)
  {
   next = (struct node *) malloc(sizeof(struct node));
   printf("Enter next elements");
   scanf("%d", &next->info);
```



```
    ptr -> Link = next ;
    ptr = ~~next~~ ptr -> link
  }
  ptr -> Link = NULL
}
```



2) Traversing of a sll

1. PTR = First
2. Repeat 3 to 4 while PTR ≠ NULL
3. print INFO (PTR)
4. PTR = ~~Link (ptr)~~ PTR → LINK
5. Stop.

```
Void traverse ()
{
  struct node * ptr;
    ptr = first;
    while(ptr != NULL)
    {
      printf("%d", ptr -> info);
      ptr = ptr -> link;
    }
}
```

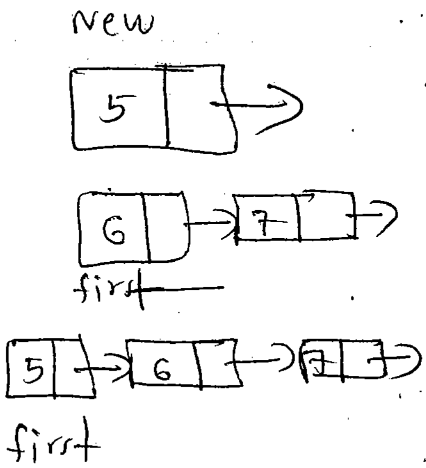
3) Insertion

Insertions can be done in 3 ways:

1. Insertion at the beginning / front
2. " " " end
3. " " " any position.

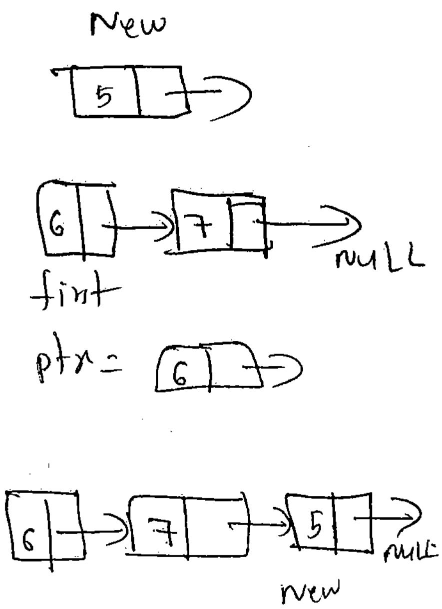
I) Insertion at front

1. Create a new node
2. new → data = data
3. new → link = first.
4. first = new
5. Stop.



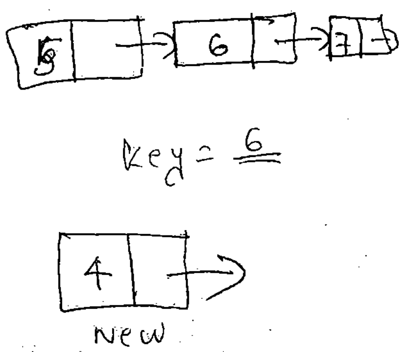
II) Insert at rear

1. Create a new node.
2. new → data = data
3. new → link = null
4. ptr = first
5. while (ptr → link ! = NULL)
    1. ptr = ptr → link
6. End while.
7. ptr → link = new.

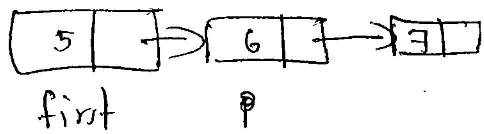


III) Insertion at any Position

1. Read 'Key' as data of node after which data is to be inserted.
2. Create a new node
3. new → data = data
4. ptr = first.

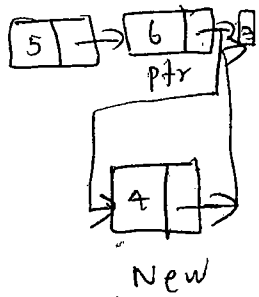


5. while((ptr→data! = key)
   and (ptr→link! = null) do



   1. ptr = ptr→link.

6. End while
7. If (ptr→link = null) then
   1. print " key is not available in the list".
   2. exit.
8. Else
   1. new→link = ptr→link
   2. ptr→link = new



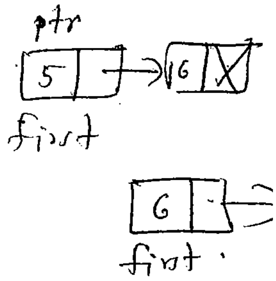
9. End

IV) Deletion

(*) 3 possibilities are there:
1. From the beginning
2. " " end
3. From any position

1) Deletion from the beginning

1. ptr = first
2. If first = NULL then
   1. print " list is empty".
3. Else
   2. first = ptr→link
   3. free(ptr);



4. EndIf.

② Deletion from end

ptr=first

1. ptr = first
2. If first = NULL then
   1. print "list is empty".
3. Else
   1. while (ptr→link != NULL) do
      1. tptr = ptr
      2. ptr = ptr → link
   2. End while
4. tptr→link = NULL
5. ~~stop~~ free(ptr)

③ Delete Any Node

1. ptr = first.
2. If first = NULL then
   1. print "list is empty"
3. Else
   1. Enter ~~data~~ key to be deleted".
   2. while (ptr→link != NULL) do
      1. if (ptr → data != key) then ~~f (ptr→link~~ != ~~NULL~~) ~~do~~
         1. tpt = ptr
         2. ptr = ptr → link
      2. Else ~~End while~~.
         1. tpt → link = ptr → link.
         2. free (ptr);
      3. EndIf
4. ~~EndIf~~ End while
5. If

③ Delete Any node

1. ptr = first
2. If first = NULL then
   1. print "list is empty".
3. Else
   1. Enter key to be deleted"
   2. while (ptr → link ! = NULL) do
      1. If (ptr → data != key) then
         1. tptr = ptr
         2. ptr = ptr → link
      2. Else
         1. tptr → link = ptr → link;
         2. free (ptr)
         3. Exit
      3. EndIf.
   3. End While.
4. EndIf
~~5. If (ptr~~
5. stop.

V Sorting

Arranging the informations in a linked list either in an ascending or in a descending order.

struct node *tptr
int temp;

1. ptr = first
2. While (ptr → link ! = null) ~~then~~ do

1. tptr = ptr → link ;
2. while (tptr != null) do
   1. If (ptr → data > tptr → data) then
      1. temp = ptr → data
      2. ptr → data = tptr → data
      3. tptr → data = temp
   2. EndIf
   3. tptr = tptr → link
3. EndWhile .
4. ptr = ptr → link .
3. EndWhile .
4. stop .

VI Searching

1. Read key as information of node to be searched.

2. ptr = first .
3. flag = 0, location = null
4. while (ptr != null) and (flag = 0) do
   1. If (ptr → data = key) then
      1. flag = 1
      2. Location = ptr
   2. Else
      1. ptr = ptr → link .
      2.
   3. End If

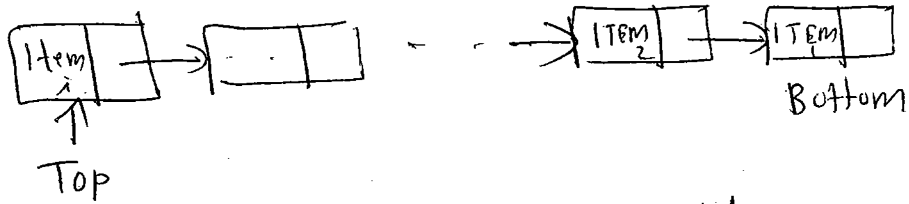
5. End While
6. If ptr = null & flag = 0 then
   1. print ~~unsuccessful~~ unsuccessful search.
7. Else
   1. print element is found ~~at position~~ i
8. EndIf
9. stop.

Linked Stacks & Queues

* Although array representation of stack is very easy & convenient but it allows only to represent a fixed sized stack.
* In several applications, size of the stack may vary during program execution.
* One solution to this problem is to represent a stack using linked list.
* Singly linked list structure is sufficient to represent any stack.
* Here 'Data' field is for the ITEM, & Link field is usual to point to the next item.
* In the linked list representation, 1st node on the list is the current item ie, the item at the top of the stack

& the last node is the node containing bottom-most item.

(*) ~~PUSH~~



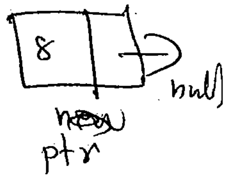
(*) PUSH Operation will add a new node at the TOP & POP will remove a node from the TOP of the list.

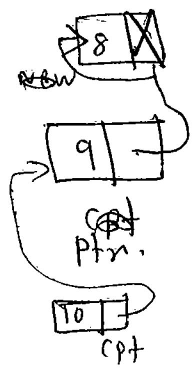
Operations ~~on~~ S/

Creation

```
struct node
{ int data;
  ~~int~~ link;
};
struct node * top, *ptr, * cpt;

ptr = (struct node *)malloc(sizeof(struct node));
printf("Enter first node");
scanf("%d", & ptr->data);
ptr->link = NULL;
do
{
  cpt = (struct node *) malloc (sizeof (struct node));
  printf("Enter next node");
  scanf("%d", & cpt->data);
  cpt->link = ptr
  ptr = cpt
  printf("Continue (Y/N)");
} ch = getchar();
```





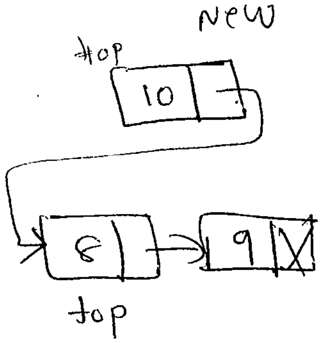
} while (ch = = 'Y');

top = ptr ;

Operations

① PUSH

1. Create a new node.
2. new → data = data
3. new → link = top
4. top = new
5. stop.



② POP

1. If top = NULL then
   1. print "underflow
2. Else
   1. ptr = top;
   2. top = ptr → Link.
   3. free (ptr)
3. EndIf.
4. stop.

③ Traverse

1. ptr = top
2. while (ptr → Link != NULL) do
   1. print "ptr → data"
   2. ptr = ptr → Link
3. Endwhile

Advantages of Linked Stack

(1) Stack does not need to be of fixed size. There can be any no. of elements or nodes in the stack.

(2) Insertion & deletion operation do not involve more data movements.

(3) Memory space is not wasted, because it is allocated only when the user wants to push an element into the stack.

Linked Queue

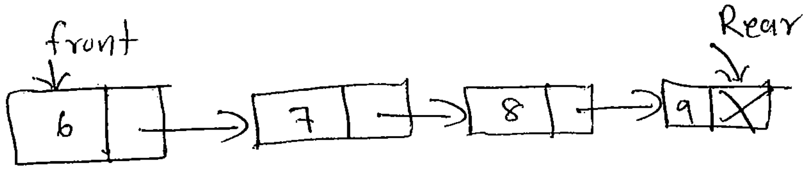
* In several applications, length of the queue may not be predicted ~~before~~ & it varies abruptly.
* To overcome this, another preferable representation of queue is with linked list.
* ~~Queue~~ The structure of a node will be as:

```
struct node
{ int data;
  struct node *link;
} *front, *rear;
```

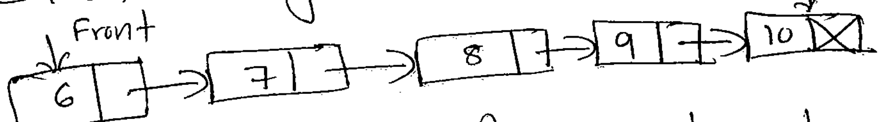
↳ data field of node holds the elements of the queue & link field holds pointer to the neighbouring element in the queue.

(*) Empty queue contains no element & can be indicated by:

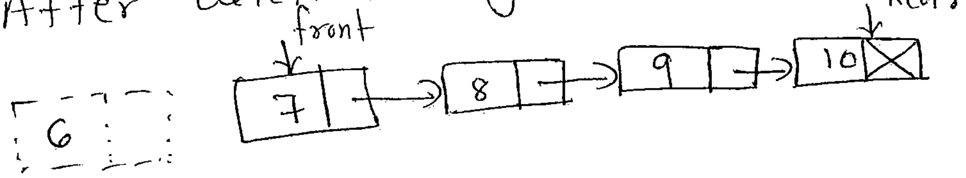
front = rear = NULL.



Insertion of 10

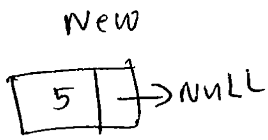


After deletion of an element

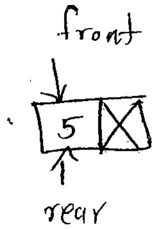


Operations on Queue.

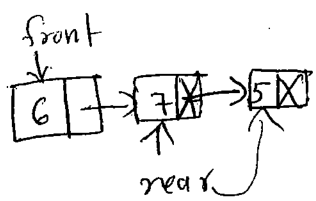
I) Enqueue



1. Create a 'new' node
2. new → data = item
3. new → link = NULL.
4. If (front = NULL) then
   1. front = new
   2. rear = new



5. Else
   1. rear → link = new
   2. rear = new



6. End If
7. Stop

II) Dequeue

1. Create a node ptr
2. If (front = NULL) then
   1. print "Queue is empty".

3. Else if (front = rear) then
    1. free (front);
    2. rear = null;
4. Else
    1. ptr = front
    2. front = ptr ⇒ link
    3. free (ptr);
5. EndIf
6. Stop.

III) Display

1. Create a node ptr
2. ptr = front
3. If (front = NULL) then
    1. print queue is empty.
4. Else
    1. while (ptr != NULL) do
        1. print " ptr ⇒ data "
        2. ptr = ptr ⇒ link
    2. End While
5. EndIf
6. Stop.

Doubly Linked Lists

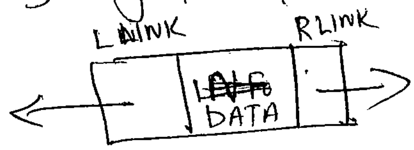
* In one-way or ~~s~~ singly linked list we can traverse only in one direction ie, in forward direction only because in this ~~linked~~ each node containing only one link which stores the address of next successor node.

* The doubly linked list uses double set of pointers, one pointing to the next node & the other pointing to the preceding/ previous node.

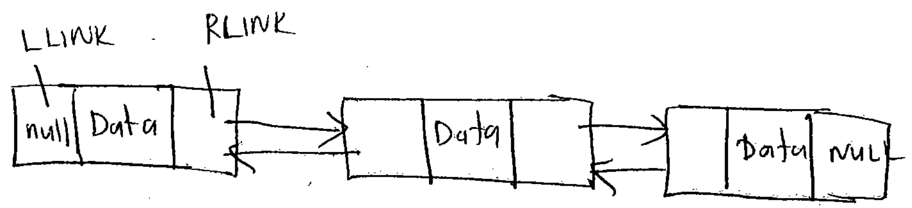
* ie, doubly linked list is one in which all nodes are linked together by multiple links which help in accessing both the successor & predecessor node for any arbitrary node within the list.

* Doubly linked list is a 2-way list because one can move in either direction, ie, either from left to right or from right to left.

* Every nodes in a doubly linked list has 3 fields :



* Here LLINK will point to the node in the left side (or previous node) ie, LLINK will hold the address of the previous node.
* RLINK will point to the node in the right side (or next node) ~~that~~ ie, RLINK will hold the address of next node.
* Data field store the information of the node.



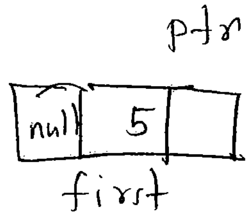
* Structure declaration is:

```
struct node
{
    int data;
    struct node *rlink;
    struct node *llink;
}
```

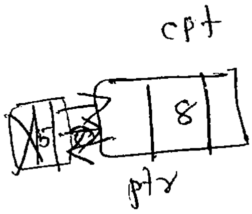
Operations Of Doubly Linked List

1) Creation

1. Create a node ptr.
2. ptr → data = key.
3. ptr → llink = NULL
4. First = ptr.
5. while (ch = 'y') do
    1. create node cpt



2. cpt → data = key
3. ptr → rlink = cpt
4. cpt → llink = ptr
5. ptr = cpt.



6. End While
7. ptr → rlink = null .
8. stop.

2) Traversing of 2 way - linked list

a) Forward Traversing
b) Backward "

a) Forward Traversing

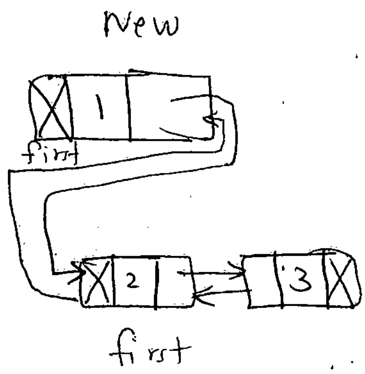
1. ptr = first .
2. While (ptr ! = NULL) do
   1. print " ptr → data"
   2. ptr = ptr → rlink
3. End While
4. stop.

b) Backward Traversing

1. ptr = first
2. while (ptr → rlink ! = null) do
   1. ptr = ptr → rlink
3. End While.
4. While (ptr ! = NULL)
   1. print " ptr → data"
   2. ptr = ptr → llink
5. End While
6. stop

3) Insertion

a) At beginning



1. Create new node.
2. new → data = key.
3. new → rlink = first
4. first → llink = new
5. new → llink = null.
6. first = new
7. Stop.

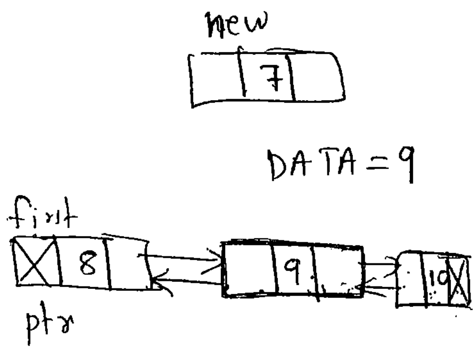
b) At End

1. Create new node
2. new → data = key
3. Ptr = first
4. While (ptr → rlink ! = NULL) do
   1. ptr = ptr → rlink
5. End While.
6. ptr → rlink = ~~ptr~~ new
7. new → llink = ptr
8. new → rlink = NULL
9. Stop.

c) Insertion In between

1. Create new node
2. new → data = ~~key~~ ~~(ro)~~ key
3. Read ~~data~~ DATA as information after which insertion will be made.

4. ptr = first
5. while (ptr → data ≠ DATA)
    1. ptr = ptr → rlink



6. EndWhile.
7. tptr = ptr → rlink
8. ptr → rlink = ~~ptr~~ new
9. new → llink = ptr
10. new → rlink = tptr
11. tptr → llink = new
12. stop.

4) Deletion

a) From beginning

1. If first = NULL then
    1. print "underflow"
    2. Exit.
3. Else
    1. ptr = first
    2. First = ptr → rlink
    3. First → llink = NULL.
    4. free (ptr)
4. EndIf
5. Stop

b) From End

1. If first = NULL then
    1. print "underflow"
    2. Exit.