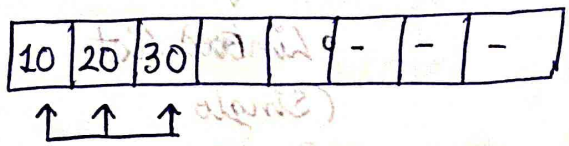
DATA STRUCTURES

- Data structure is a way of organising data in memory so that it can be used efficiently.
It consists of two parts:

1. How data is organised in memory.
2. What all operations can be performed on that data structure.

1.

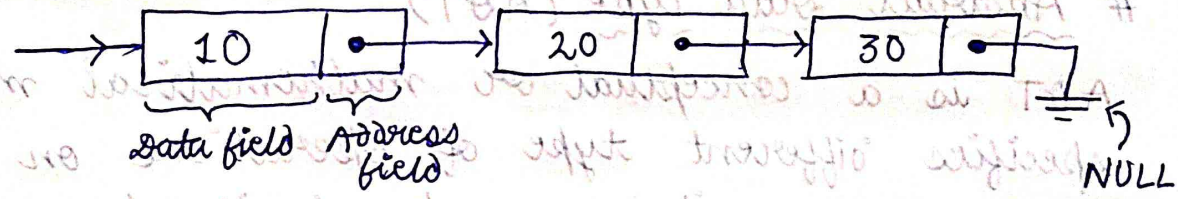


Array (contiguous block of memory)

Elements placed one after other

Advantage: Direct access is available

Linked List (no contiguous block of m/m)

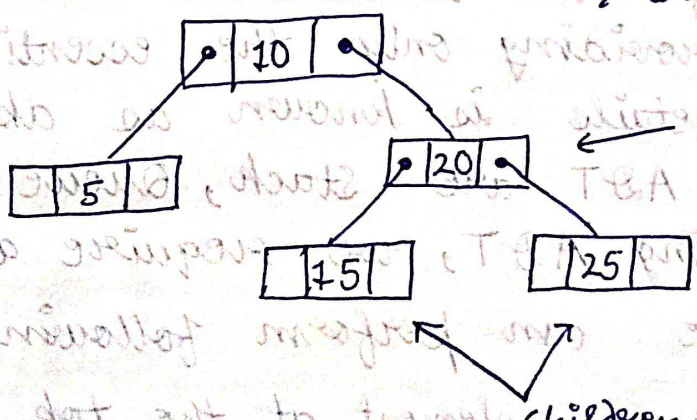


Data field | Address field

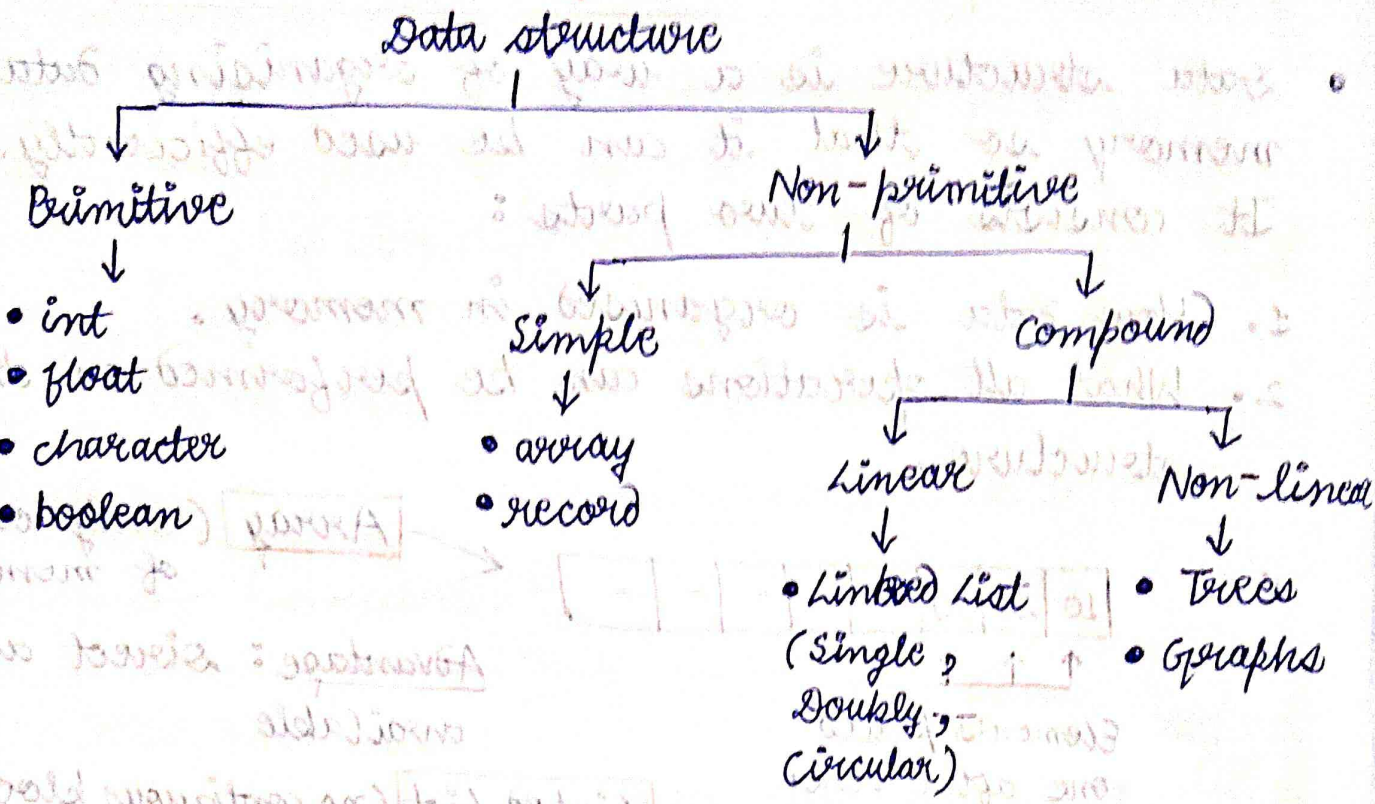
Node

Binary Search Tree (Non-linear D.S.)

↳ Used for fast indexing



Classification of data structure



Abstract Data Type (ADT)

ADT is a conceptual or mathematical model which specifies different type of operations on that data type. However, it does not specify how these operations will be implemented. It is called abstract because it only tells the operations which ADT can perform without telling how these will be implemented. This process of providing only the essentials while hiding the details is known as abstraction.
Examples of ADT are Stack, Queue and List.
(For implementing ADT, we require a suitable DS)
In stack, we can perform following operations:

1. Push: To add an element at the top of stack.
2. Pop: To remove an element from top of stack
3. Display: To print the content of stack from top to bottom.

4. Peek : To print the top most element.

These operations are specified in stack ADT which works on LIFO basis. To implement stack, the programmer will be using a suitable data struct -ure such as an array or linked list.

---

* Implementing Stack using Array

Stack is an ADT which works on LIFO basis. Elements placed at the last is always at the top and the element placed first is at the bottom. Stack is used in no. of applications such as:

1. Converting infix expression to postfix
2. Evaluating the postfix expression
3. Reversing a string
4. Checking the parenthesis
5. Back tracking
6. Storing the return address in function call
7. Memory allocation
8. Compiler design

In stack, we can perform operations like Push, Pop, Display and Peek.

Algorithm to push an element

Initially, top = -1 (stack is empty)

Step 1 : Start

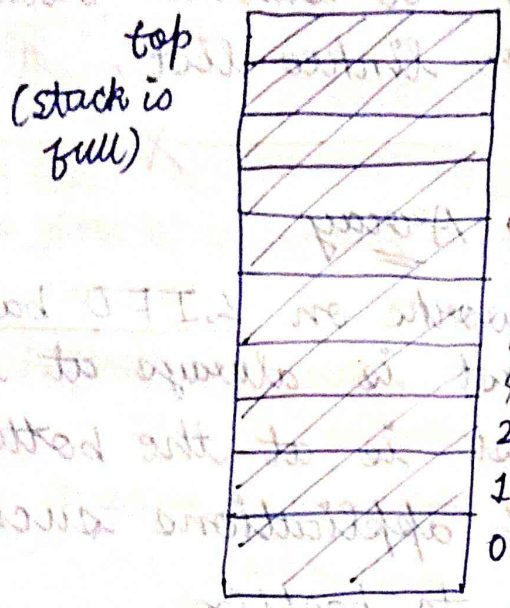
Step 2 : If top = max-1, then display "Stack is full", goto step 6

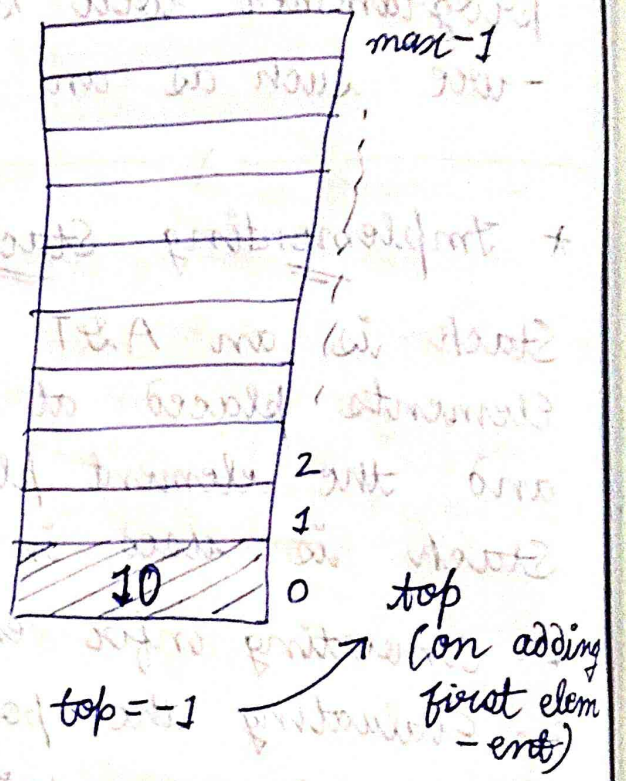
Step 3 : Input the element to be pushed

Step 4 : Increment top by 1.

Step 5 : Assign element to A[top]

Step 6 : Stop





Algorithm to Pop an element

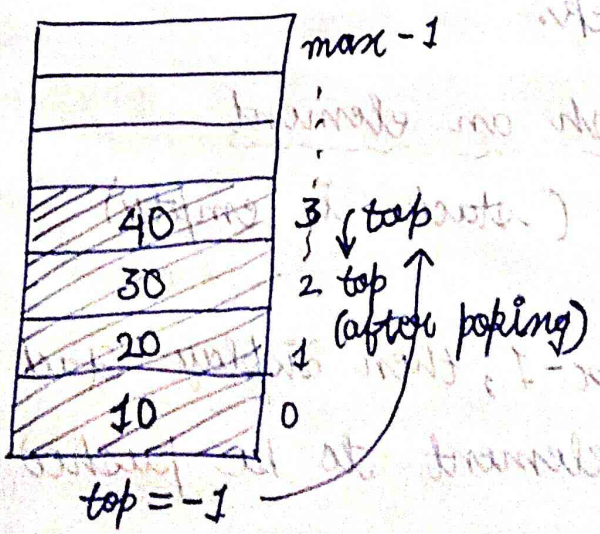
Step 1 : Start

Step 2 : If top = -1, then display "Stack is empty", goto step 5

Step 3 : Display the contents at A[top]

Step 4 : Decrement top by 1

Step 5 : Stop



{ After popping, 40 is still the part of array but not the part of stack. }

↓

0 to top

Algorithm to display elements of stack from top to bottom

Step 1: Start

Step 2: If top = -1, then display "Stack is empty", goto step 8.

Step 3: Take an integer variable temp.

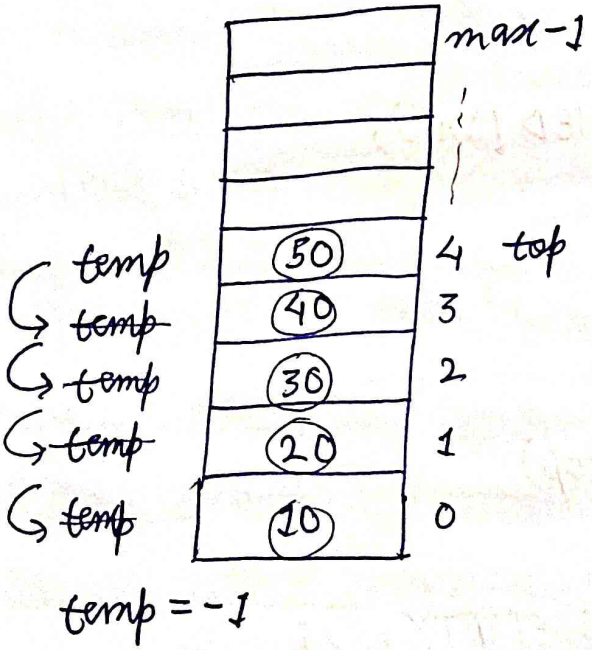
Step 4: Assign top to temp

Step 5: Repeat step 6 and 7 until temp = -1

Step 6: Display contents of A[temp]

Step 7: Decrement temp by 1

Step 8: Stop



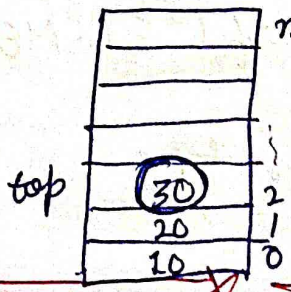
Algorithm to peek

Step 1: Start

Step 2: If top = -1, display "Stack is empty", goto step 4

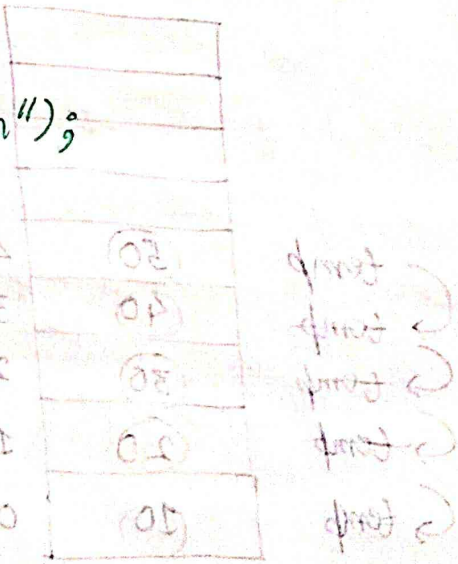
Step 3: Display A[top]

Step 4: Stop



Menu driven program to implement stack using array

```
# include <stdio.h>
# define MAX 10    // Maximum size of Stack assigned as 10
// Push function
int Push (int *stack, int top)
{   if (top == MAX-1)
    {  printf ("\n STACK IS FULL ! ");
    }
    else
    {  int elem;
       printf ("\n Enter the element to be pushed : ");
       scanf ("%d", &elem);
       top += 1;
       stack[top] = elem;
       printf ("ELEMENT PUSHED ! \n");
    }
    return top;
}
// Pop function
int Pop (int *stack, int top)
{   if (top == -1)
    {  printf ("\n STACK IS EMPTY ! ");
    }
    else
    {  printf ("\n The element at top : %d \n", stack[top]);
       top -= 1;
       printf ("ELEMENT POPPED ! \n");
    }
    return top;
}
```



```
// Display function
void Display (int *stack, int top)
{   if ( top ==-1)
    { printf ("\n STACK IS EMPTY!");
    }
    else
    { int temp = top;
      printf ("\n Elements in stack from top to bottom : ");
      while ( temp != -1)
      { printf ("%d ", stack[temp]);
        temp --=1;
      }
      printf ("\n");
    }
}
// Peek function
void Peek (int *stack, int top)
{   if (top == -1)
    { printf ("\n STACK IS EMPTY!");
    }
    else
    { printf ("\n Element at the top in the stack : %d\n",
          stack[top]);
    }
}
int main()
{ int stack[MAX], top =-1, ch;
  do
  { printf ("\n MENU \n 1. Push \n 2. Pop \n 3. Display \n
      4. Peek ");
    printf ("\n Enter your choice : ");
    scanf ("%d", &ch);
```

```
switch (ch)
{   case 1: top = Push(stack, top);
            break;
    case 2: top = Pop(stack, top);
            break;
    case 3: Display(stack, top);
            break;
    case 4: Peek(stack, top);
            break;
    default: ch = 0;        // Exit condition
             break;
}
} while(ch);
return 0;
}
```

21/8/2020

* Implementing Queue using Array

Queue is an ADT which works on FIFO (First In First Out) basis. It can be implemented using an array or a linked list data structure. Here, we are implementing queue ADT using an array.

Applications of Queue

1. Processor scheduling (Single processor, more than one process)
2. Device scheduling (Single O/P device, more than one user)
3. In implementing BFS (Breadth First Search) for graph transversal algorithm
4. In service provider systems where there are multiple inquiries and less number of handlers.

5. In flight landing and take off systems, as more than one aeroplane compete for single runway at a given time for landing and take off.

Queue operations

1. Enqueue : To add an element at the rear of the queue.
2. Dequeue : To remove an element from front of the queue.
3. Display : To print all elements in queue from front to rear.
4. Peek : To print front element of the queue.

Implementing Queue using an Array.

- Assumptions → Static memory allocation

We have an integer array 'a' with size 'max', where 'max' is a constant. Take two integer variables 'front' & 'rear' and initialise them to -1 as initially queue is empty.

- Algorithm to implement Enqueue

Step 1 : Start

Step 2 : If ((rear + 1) % max) = front, then display a message, "Queue is full", goto step 7.

Step 3 : Input the element to be inserted.

Step 4 : rear = (rear + 1) % max

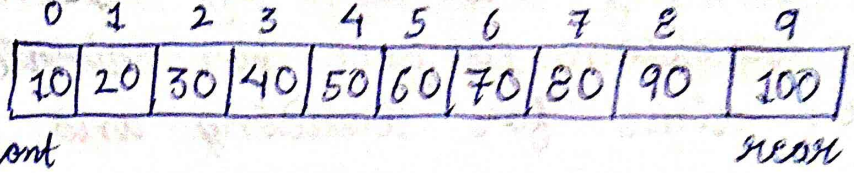
Step 5 : Assign the element to a[rear]

Step 6 : If front = -1, it means element inserted is first element, then increment front to 0.

Step 7 : Stop

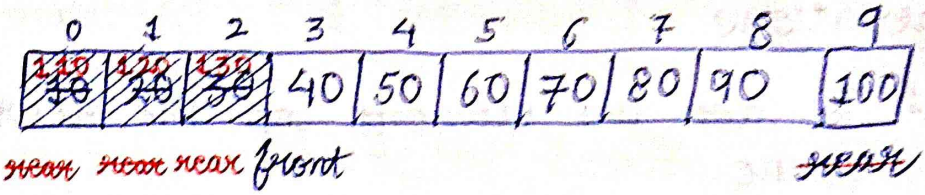
max = 10

(i) a



(Queue is full)

(ii) a



(Queue is full)

If (rear + 1) % max = front,

- For (i), (9 + 1) % 10 = 0
  10 % 10 = 0 = RHS
- For (ii), (2 + 1) % 10 = 3
  3 % 10 = 3 = RHS

Here, we have used '% max' in formula so that value of 'rear + 1' doesn't exceed 'max'.

e.g. If rear = 9,
so, rear + 1 = 10 & 10 % 10 (max) = 0
So, now, the element will be inserted at index '0'.

- **Algorithm to implement Dequeue**

Step 1: Start

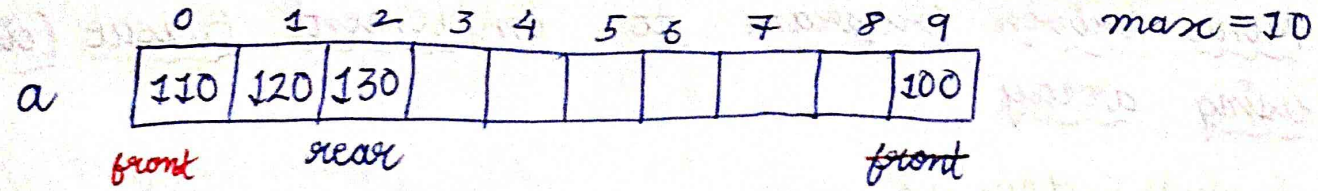
Step 2: If front = -1, then display a message, 'Queue is empty', goto step 6.

Step 3: If front = rear (only 1 elem. present), then display the value of a[front] & reset front & rear to -1, goto step 6.

Step 4: Display a[front]

Step 5: front = (front + 1) % max

Step 6: Stop



{Also called circular queue}

- **Algorithm to implement display**

Step 1 : Start

Step 2 : If front = -1, then display a message that queue is empty, goto step 8.

Step 3 : Take an integer 'temp' & assign value of 'front' to it.

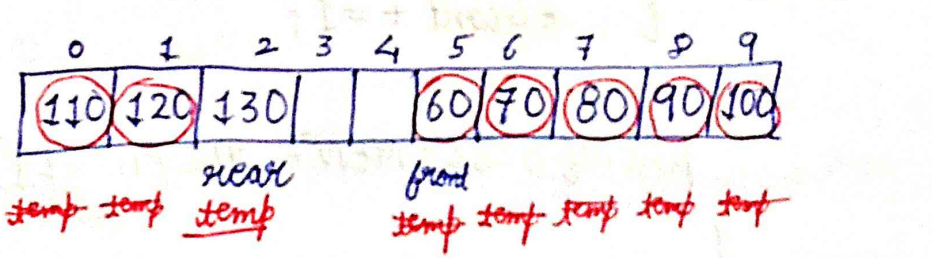
Step 4 : while (temp != rear), repeat step 5 & 6.

Step 5 : Display a[temp]

Step 6 : temp = (temp + 1) % max

Step 7 : Display a[temp]

Step 8 : Stop



- **Algorithm to implement peek**

Step 1 : Start

Step 2 : If front = -1, then display a message that queue is empty, goto step 4.

Step 3 : Display the value of a[front]

Step 4 : Stop

Menu driven program to implement Queue (circular) using array

```
#include <stdio.h>
#define MAX 10   // Maximum size of Queue assigned as 10
// Insert function
void Insert (int queue [], int *front, int *rear)
{   if ((*rear + 1) % MAX == *front)
    {   printf ("\n QUEUE IS FULL ! \n");
    }
    else
    {  int elem;
       printf ("\n Enter the element to be inserted : ");
       scanf ("%d", &elem);
       *rear = (*rear + 1) % MAX;
       queue [*rear] = elem;
       if (*front == -1)
       {   *front += 1;
       }
       printf ("ELEMENT INSERTED\n");
    }
}
// Serve function
void Serve (int queue[], int *front, int *rear)
{   if (*front == -1)
    {   printf ("\n QUEUE IS EMPTY ! \n");
    }
    else if (*front == *rear)
    {   printf ("\n Element at front : %d \n", queue [*front]
        *front = -1 ;
        *rear = -1 ;
```

```
        printf ("ELEMENT SERVED ! \n");
    }
    else
    { printf ("\n Element at front : %d \n", queue[*front]);
      *front = (*front + 1) % MAX;
      printf ("ELEMENT SERVED ! \n");
    }
}
// display function
void Display ( int queue [], int front, int rear).
{   if (front == -1)
    {   printf ("\n QUEUE IS EMPTY ! \n");
    }
    else
    { int temp = front;
      printf ("\n Elements in queue from front to
        rear : ");
      while (temp != rear)
      {   printf ("%d ", queue [temp]);
          temp = (temp + 1) % MAX;
      }
      printf ("%d \n", queue [temp]);
    }
}
int main()
{   int queue [MAX], front = -1, rear = -1, ch;
    do
    {   printf ("\n MENU \n 1. Insert \n 2. Serve \n
            3. Display \n 4. Exit ");
        printf ("\n Enter your choice : ");
```

```
        scanf("%d", &ch);
        switch(ch)
        {   case 1: Insert(queue, &front, &rear);
                    break;
            case 2: Serve(queue, &front, &rear);
                    break;
            case 3: Display(queue, front, rear);
                    break;
            case 4: ch=0;
                    break;
            default: ch=0;          // Exit condition
                    break;
        }
    } while(ch);
    return 0;
}
```

2/9/2020

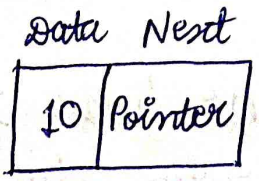
* Linked List

It is a linear data structure. In linked list, nodes are not stored at a contiguous location; nodes are linked using pointers.

Types of linked list

1. Singly linked list
2. Doubly linked list
3. Circular linked list

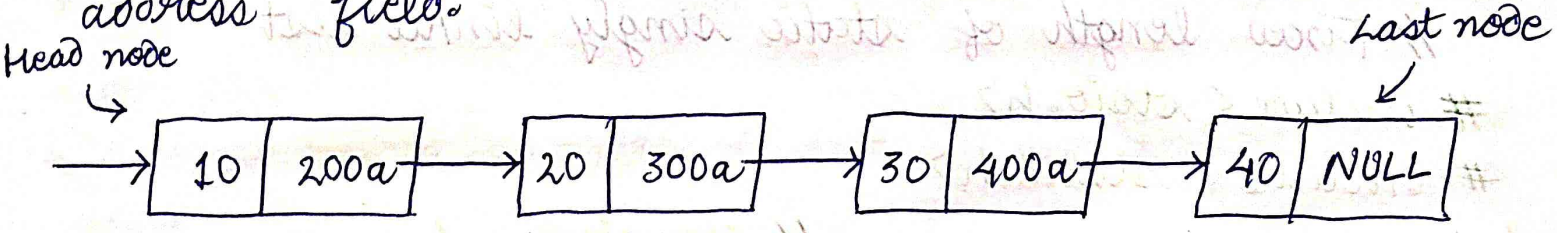
1. **Singly linked list**: A node in singly linked list has two fields, i.e., **Data field** & **address field** (Next) The address field contains the address of next node (using a pointer).



Node

```
struct node
{ int data;
  struct node *Next;
};  // Self referential structure
```

A data part stores the data & 'Next' part stores the address of the next node. The last node in the linked list is represented by a NULL in the address field.



It is **unidirectional**, i.e. we cannot backtrack. We can access nodes sequentially.

Why Linked List?

Arrays can be used to store linear data of similar types, but arrays have the following limitations:

(a) The size of the arrays is fixed, so we must know the upper limit of the no. of elements in advance.

(b) Inserting a new element in an array of elements is expensive in terms of shifting.

(c) For better memory utilisation (in terms of frag - mented memory), linked list is required.

⇒ Advantages over arrays

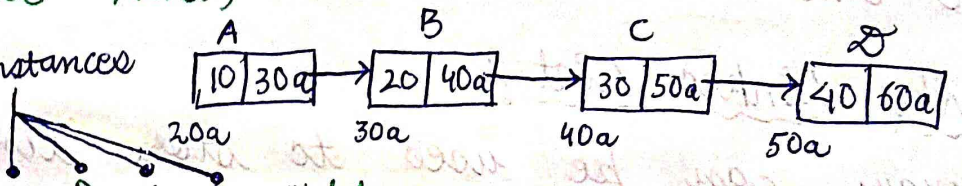
- Dynamic size
- Ease of insertion/deletion
- Heterogeneous data types can be handled.

⇒ Drawbacks

- Random access is not allowed. We have to access elements sequentially starting from the first node.
- Extra memory space for a pointer is required with each element of the list.

Example

```
// Fixed length of static singly linked list
#include<stdio.h>
#include<stdlib.h>
typedef struct node      // structure of node
{  int data;
   struct node *Next;
} nodetype;
int main()
{  nodetype A, B, C, D, *ptr = NULL;
   A.data = 10, B.data = 20, C.data = 30, D.data = 40;
                  // values are stored in data field of
                     each node
   A.Next = &B;   // connecting nodes A & B
   B.Next = &C;   // connecting nodes B & C
   C.Next = &D;   // connecting nodes C & D
   D.Next = NULL; // Assigning NULL in Next field of
                     last node (D)
   ptr = &A;  // Assigning address of first node
                 to pointer 'ptr'   // ptr = 20a
```



```
while (ptr != NULL)
{   printf ("%d -> ", ptr -> data);
    ptr = ptr -> Next;
}
return 0;
}
```

✓ 'typedef' has been used to provide alias name for the structure. So, we can write 'nodetype' instead of 'struct node' now.

⇒ Applications of linked list in Computer Science

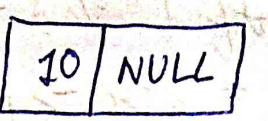
- Implementation of stack & queue
- Implementation of graphs : Adjacency list representation of graphs.
- Dynamic memory allocation : We use linked list of free blocks
- Maintaining directory of names
- Performing arithmetic operations on long integers
- Manipulation of polynomials by storing constants in the node of linked list representing sparse matrices.

3/9/2020

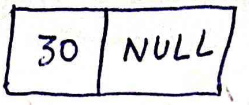
Singly linked list : Inserting node at right hand side

```
typedef struct node
{   int data;
    struct node *next;
} nodetype;
nodetype *left = NULL, *right = NULL, *p = NULL;
p = (nodetype *) malloc (sizeof (nodetype));
```

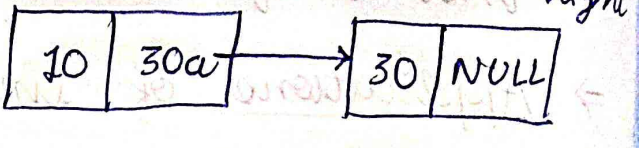
```
if (p != NULL)
{   p -> data = n;
For first node { if (left == NULL)
                 { left = right = p;
                 }
                 else
                 { right -> next = p;
                   right = p;
                 }
                 right -> next = NULL;
}
```



p = 20a
left = right = p = 20a



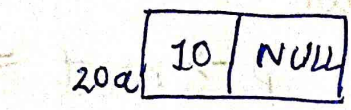
p = 30a
right = p

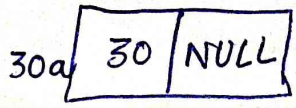


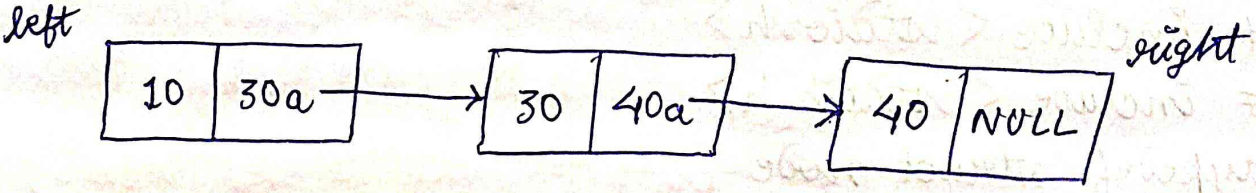
// C program to create a singly linked list by inserting node at right hand side

```
# include<stdio.h>
# include<stdlib.h>
typedef struct node
{   int data;
    struct node *next;
} nodetype;
int main()
{ nodetype *left = NULL, *right = NULL, *p = NULL;
  int n, ch;
  do
  { printf("\n Enter value to insert in node : ");
    scanf("%d", &n);
    p = (nodetype *)malloc(sizeof(nodetype));
    if (p != NULL)
    {   p -> data = n;
        if (left == NULL)
        { left = right = p;
        }
```

```
        else
        {   right -> next = p;
            right = p;
        }
        right -> next = NULL;
    }
    printf("\n Do you want to add more node ? \n
        Press 1 to continue otherwise 0");
    scanf("%d", &ch);
} while (ch==1);
printf("\n Display the list :");
while (left != NULL)
{   printf("%d", left -> data);
    left = left -> next;
}
return 0;
}
```



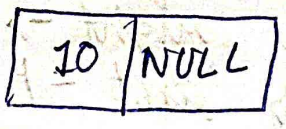




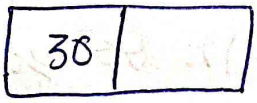
Singly Linked List : Inserting node at left hand side

```
typedef struct node
{   int data;
    struct node *next;
} nodetype
nodetype *left = NULL, *p = NULL;
p = (nodetype *) malloc( sizeof(nodetype));
if (p != NULL)
{   p -> data = n;
    if (left == NULL)
    {   left = p;
        left -> next = NULL;
    }
```

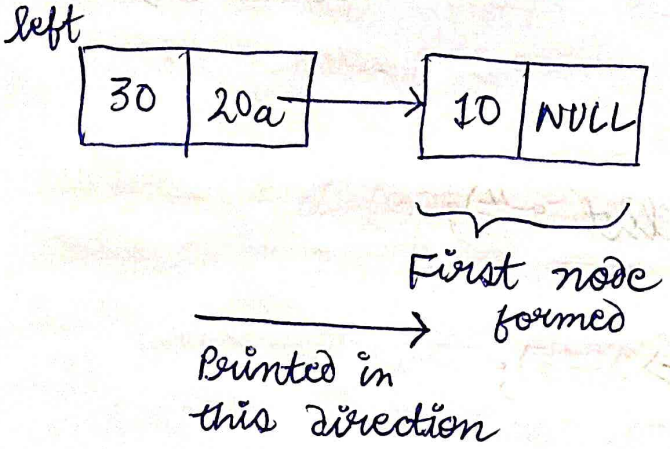
```
else
{   p -> next = left;
    left = p;
}
}
```



p = 20a
left = p



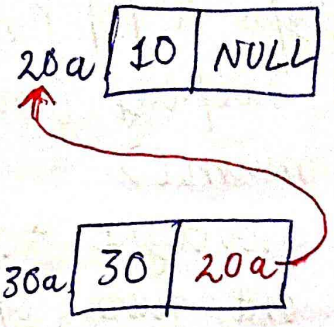
p = 30a
left = p

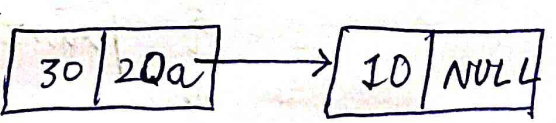


// C program to create a singly linked list by inserting node at left hand side

```
# include <stdio.h>
# include <stdlib.h>
typedef struct node
{   int data;
    struct node *next;
} nodetype;
int main()
{   nodetype *left = NULL, *p = NULL;
    int n, ch;
    do
    {   printf("\n Enter value to insert in node : ");
        scanf("%d", &n);
        p = (nodetype *) malloc (sizeof(nodetype));
        if (p != NULL)
        {   p -> data = n;
```

```
        if (left == NULL)
        {   left = p;
            left -> next = NULL;
        }
        else
        {   p -> next = left;
            left = p;
        }
    }
    printf("\n Do you want to add more node ?
            \n Press 1 to continue otherwise 0");
    scanf("%d", &ch);
} while (ch == 1);
printf("\n Display the list :");
while (left != NULL)
{   printf("%d ", left -> data);
    left = left -> next;
}
return 0;
}
```





4/9/2020

* Implementing Stack ADT using Linked List & Single pointer

```
# include <stdio.h>
# include <stdlib.h>
typedef struct node
{   int info;
    struct node * next;
} nodetype;
```

```
nodetype * push(nodetype *, int);
nodetype * pop (nodetype *);
void display (nodetype *);
int main()
{   nodetype *top = NULL;
    int num, opt;
    char choice;
    do
    {   printf ("Enter opt \n");
        scanf ("%d", &opt);
        if (opt == 1)
        {   scanf ("%d", &num);
            top = push (top, num);
        }
        else if (opt == 2)
        {   if (top != NULL)
            {   top = pop (top);
            }
            else
            {   printf ("STACK IS EMPTY \n");
            }
        }
        else if (opt == 3)
        {   if (top != NULL)
            {   Display (top);
            }
            else
            {   printf ("STACK IS EMPTY \n");
            }
        }
        else
        {   printf ("INVALID OPTION ! \n");
        }
```

```
        printf ("Do you want to continue ? (Y/N) \n");
        scanf ("%c", &choice);
    } while (toupper (choice) == 'Y');
    return 0;
}

nodetype * push (nodetype * tp , int nm)
{       nodetype *p;
        p = (nodetype *) malloc (sizeof (nodetype *));
        if (p == NULL)
        {   printf ("NOT ENOUGH MEMORY ! \n");
        }
        else
        {   p -> info = nm;
            p -> next = tp;
            tp = p;
        }
        return tp;
}
nodetype * pop ( nodetype *tp)
{   nodetype * temp;
    temp = tp;
    printf ("element popped is %d ", tp -> info);
    tp = tp -> next;
    free (temp);
    return tp;
}
void Display (nodetype *tp)
{   while (tp != NULL)
    {   printf ("%d ", tp -> info);
        tp = tp -> next;
    }
}
```

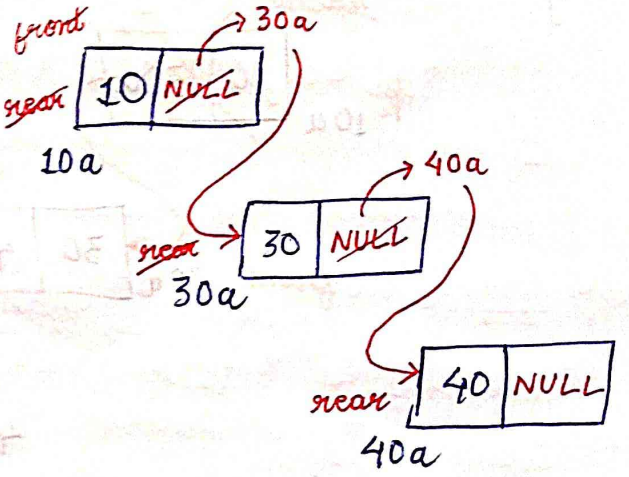
5/9/2020

* Implementing Queue ADT using Linked List & Single pointer

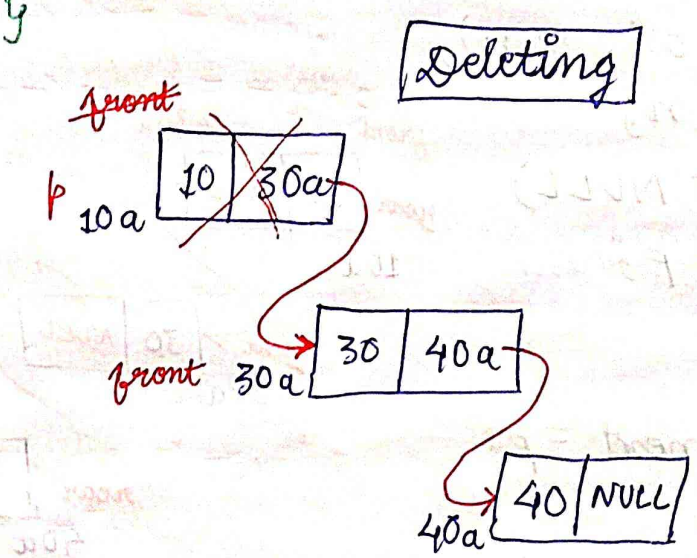
```
# include<stdio.h>
# include<stdlib.h>
typedef struct node
{  int data;
   struct node *next;
} nodetype;
nodetype * insert (nodetype *);
nodetype * delete (nodetype *);
void display (nodetype *);
int main()
{  int ch;
   nodetype *front = NULL, *rear = NULL;
   do
   {  printf ("\n 1. Insert \n 2. Delete \n 3. Display \n
        4. Exit \n Enter your choice : ");
      scanf ("%d", &ch);
      switch(ch)
      {  case 1: rear = insert (rear);
                 if ( front == NULL)
                 {  front = rear; }
                 break;

         case 2: front = delete (front);
                 if (front == NULL)
                 {  rear = NULL; }
                 break;
         case 3: display (front);
                 break;
```

front → NULL rear → NULL

```
} while (ch <= 3);
return 0;
}
nodetype *insert (nodetype *rear)
{ int n;
  nodetype *p = NULL;
  p = (nodetype *) malloc(sizeof(nodetype));
  if (p != NULL)
  { printf("\n Enter value to insert : ");
    scanf("%d", &n);
    p -> data = n;
    if (rear == NULL)
    { rear = p;
    }
    else
    { rear -> next = p;
      rear = p;
    }
    rear -> next = NULL;
  }
  return rear;
}
nodetype *delete (nodetype *front)
{ nodetype *p = NULL;
  if (front == NULL)
  { printf("Nothing to delete !");
  }
  else
  { printf("\n Deleted node's data is -> %d", front
                                          -> data);
    p = front;
    front = front -> next;
  } free(p);
```



```
    return front;
}
void display (nodetype *front)
{   if (front == NULL)
    {   printf ("\nNothing to display!");
    }
    else
    {   while (front != NULL)
        {   printf ("%d ", front->data);
            front = front->next;
        }
    }
}
```



5/9/2020

* Implementing Stack ADT using Linked List & Double pointer

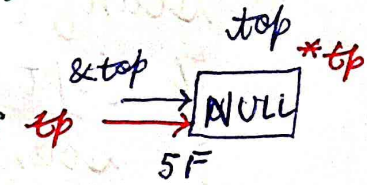
```
#include <stdio.h>
#include <stdlib.h>
typedef struct node
{   int data;
    struct node *next;
} nodetype;
void push (nodetype **, int);
void pop (nodetype **);
```

→ Now, instead of passing value of 'top', we're passing address of 'top'. ('top' itself is an address of type 'nodetype')

```
int main()
{   nodetype *top = NULL;
    int num, opt;
    char choice;
    do
    {  printf("Enter option: \n");
       scanf("%d", &opt);
       if (opt == 1)
       {  scanf("%d", &num);
          push (&top, num);        // call by reference
       }
       else if (opt == 2)
       {  if (top != NULL)
          {   pop (&top);
          }
          else
          {  printf("STACK IS EMPTY \n");
          }
       }
       else if (opt == 3)
       {  if (top != NULL)
          {   display (top);
          }
          else
          {  printf("STACK IS EMPTY \n");
          }
       }
       else
       {  printf("INVALID OPTION! \n");
       }
       printf("Do you want to continue? (Y/N) \n");
       scanf("%c", &choice);
```

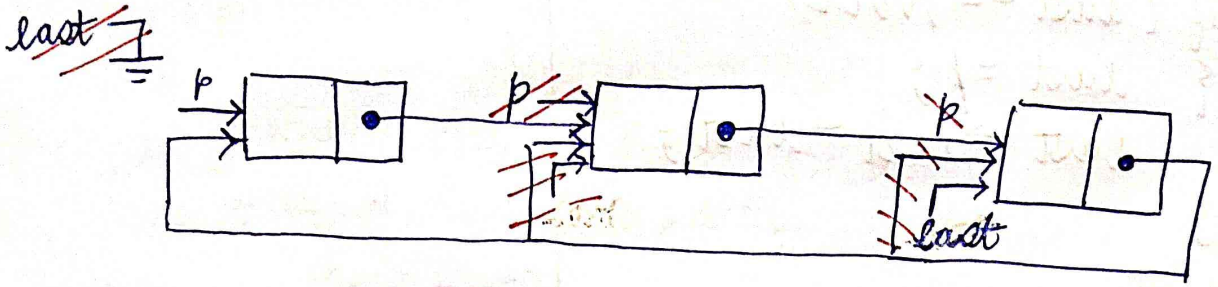
```
} while(toupper(choice)=='Y');
return 0;
}
void push(nodetype **tp, int nm)
{ nodetype *p;
  p=(nodetype *)malloc(sizeof(nodetype *));
  if (p==NULL)
  { printf("NOT ENOUGH MEMORY \n");
  }
  else
  { p->data=nm;
    p->next=*tp;
    *tp=p;
  }
}
void pop(nodetype **tp)
{ nodetype *temp;
  temp=*tp;
  printf("Element popped is %d", *tp->data);
  *tp=*tp->next;
  free(temp);
}
void display(nodetype *tp)
{ while(tp!=NULL)
  { printf("%d ", tp->data);
    tp=tp->next;
  }
}
```

tp == &top
*tp == top



6/9/2020

* Circular Linked List



p -> next = last -> next
last -> next = p

Assumptions

Using structure, define a datatype 'nodetype' having first field 'info' of type 'int' and second field 'next' which is an address of type 'nodetype'.
Take a 'nodetype' pointer 'last' & initialise it to NULL as initially, linked list is empty.

Algorithm to add an element towards start of circular linked list

Step 1 : Start

Step 2 : Using malloc() function, allocate memory for a 'nodetype' & assign its address to a 'nodetype' pointer 'p'.

Step 3 : if (p == NULL) then print that not enough memory & goto step 7.

Step 4 : Input 'num' & assign it to 'info' field of 'p'.

Step 5 : if (last == NULL) then last = p ; last -> next = p ; goto step 7.

Step 6 : p -> next = last -> next ;
last -> next = p ;

Step 7 : Stop

7/9/2020

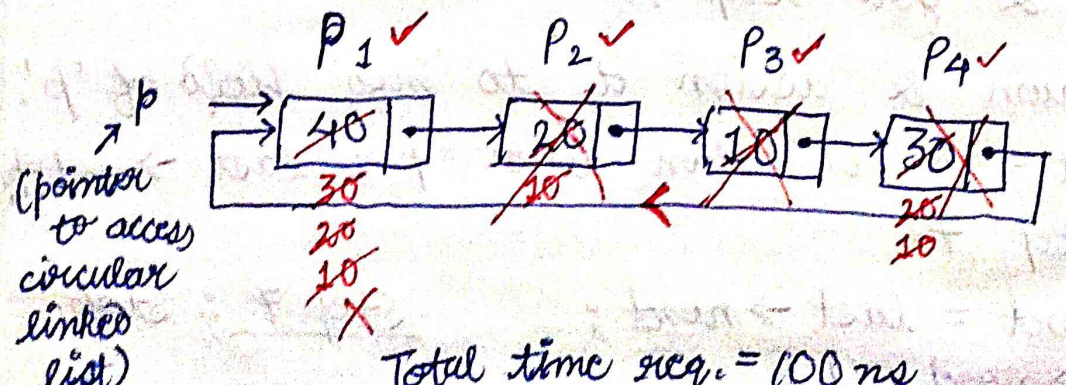
* Circular Linked List - Implementation & Applications

```
if (last == NULL)
{   last = p;
    last -> next = last;
}
else
{   p -> next = last -> next
    last -> next = p;
    last = p;
}
```

Multi-programming (Application)

In multi-programming, there are more than one processes at the state of execution while only one process is getting executed by CPU because there's a single CPU working on more than one processes. So, only one process is getting executed at any given time but there are more than one processes at the state of execution. So, CPU gives fixed time slots to every process.

Assume, there are 4 processes (P_1, P_2, P_3, P_4) & every process is given a fixed time slot of 10 ns by CPU.



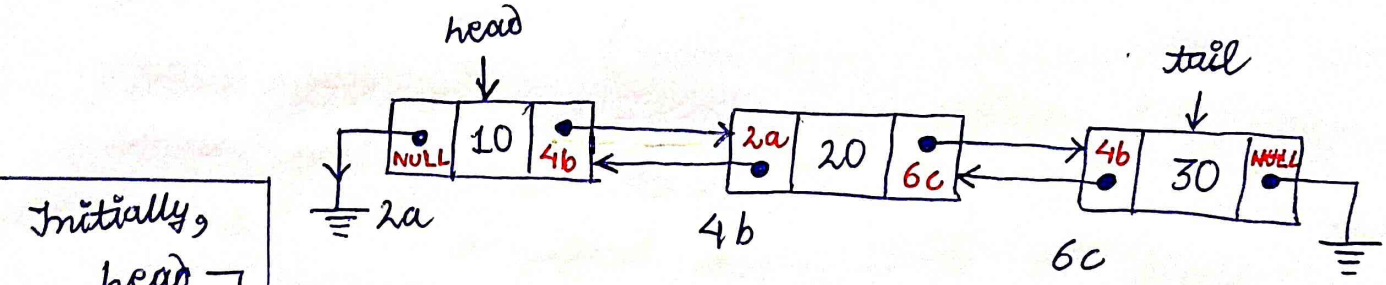
- After 30 ns, P_3 is over.
- After 60 ns, P_2 is over.
- After 90 ns, P_4 is over.
- After 100 ns, P_1 is over.

Total time req. = 100 ns

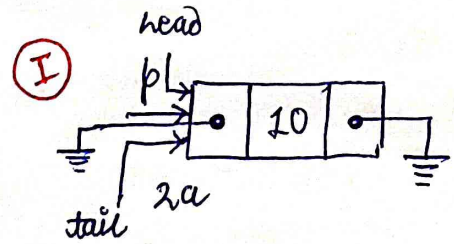
The shortest process got completed first (P_3), i.e. waiting time for shortest process was minimum, then P_2, P_4 and P_1 get completed sequentially. Thus, process requiring maximum time was completed at last (P_1). Hence, the waiting time has been distributed.

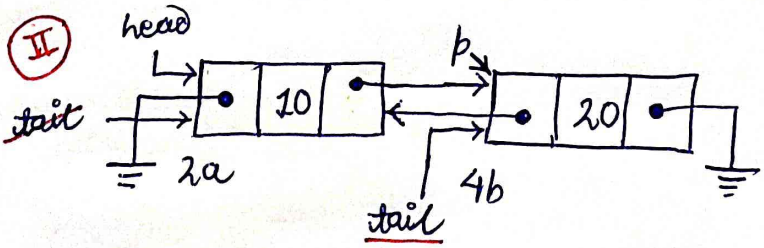
8/9/2020

* Doubly Linked List



Initially,
head
tail






```
if (tail == NULL)
{ head = tail = p;
  p -> prev = NULL;
}
else
{ tail -> next = p;
  p -> prev = tail;
  tail = p;
}
p -> next = NULL;
```

(New node is added after the last node)
