

**A ■ P ■ I ■ I ■ T**

---

ASIA PACIFIC INSTITUTE OF  
INFORMATION TECHNOLOGY

# Linked List

# LINKED LIST

## Problems in an Array and Queue

- \* Fixed no of memory allocated to the structure when the structure actually using small amount of memory
- \* If the structure requires more memory space, then introducing the possibilities of overflow

## LINKED LIST

In arrays, many of the cells are not utilized. Suppose we have two arrays of size 100 each storing Qs ( or Stack). Even if one stack is empty, the other cannot grow beyond 100.

We can think if a single array of 200 elements so that one stack may start from the top and the other may start from the bottom.

E.g..        S1 : item[0],..item[top1]

              S3 : item[199],.....,item[top2]

We need two sets of push and pop operations since S1 grows by incrementing top1, whereas s2 grows by decrementing top2.

This method fails, when # stack is more than 2.

# LINKED LIST

Suppose the items of a stack or a queue were explicitly ordered, that is each item contained within itself the address of the next item. Such structure implemented using

**LINKED list**

## Components In Linked List

Each item in the linked list called *node*

Each node contains two part - an **information** & next  
address

The **information part** holds the actual element **on the list**

The next address **field** contains the address of the next node  
in the list - is known as a **POINTER**

# LINKED LIST

The next address field of the last node in the list contains a special value known as **NULL**. This **null pointer** is used to signal the end of the list.

The list with no nodes on it is called **EMPTY LIST** or **NULL LIST**

# LINKED LIST

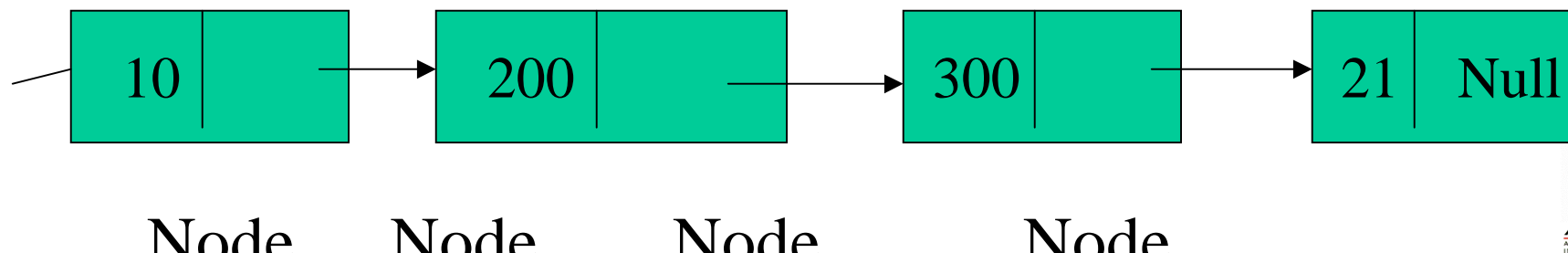
Such situation can be handled by using a data structure called **LIST** that allocate memory space only when an **item(element)** is added.

## Types of LISTS

- \* Linear Linked List [ Singly Linked List]
- \* Circular Linked List
- \* Doubly Linked List

## Linear Linked List

Data    Add

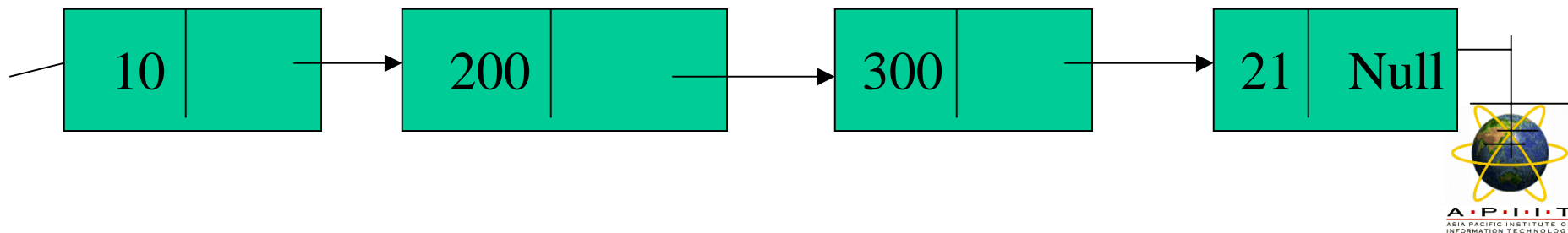


# LINKED LIST

Node : An element in the data structure (ie in list). A node structure consist of two members :

- \* data
- \* address of the next node in the list [ a pointer ]

An external pointer list stores the address of the first node that is used to start traversing the list. The last node in the list will have a NULL in its address field ( ie the last node is grounded)





# LINKED LIST

Thus for an empty list, we have

list = NULL:

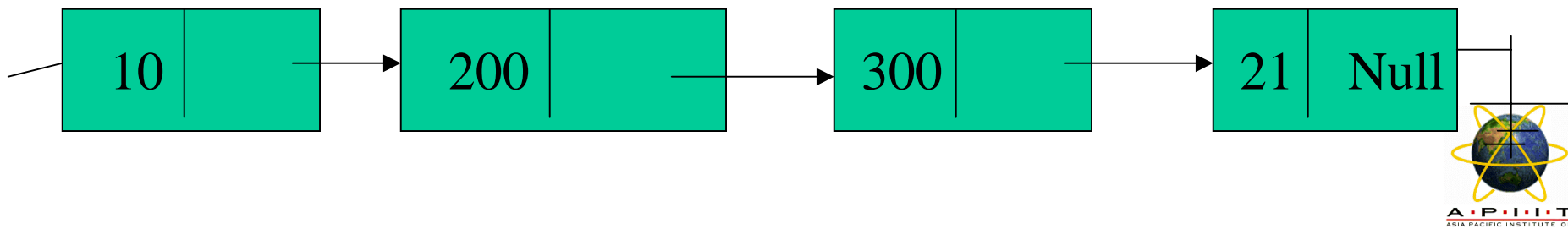
list is initialized to NULL whenever a list is declared  
(Created)

## Notation :

node(p)      :- node pointed to by p

info(p)        :- data stored in the node pointed to by p

next(p) : address of the node next to the node pointed to by p



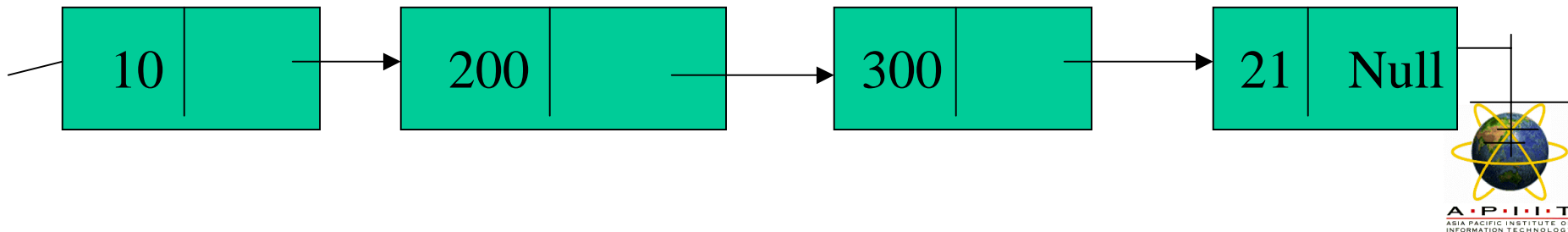
# LINKED LIST

Example :

`info(next(p))` returns the data stored in the node pointed to by `next(p)` if `next(p) != NULL`

Note :-

Contrary to an array, a list is a dynamic memory data structure. The amount of memory used by a list depends on the number of nodes in the list at any time.



# OPERATIONS ON LINKED LIST

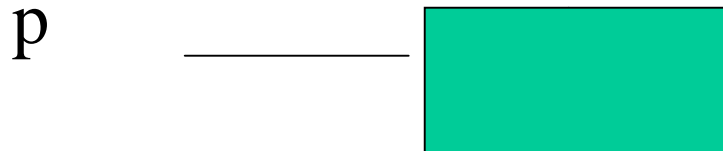
getnode :- for inserting a node

freenode :- or deleting/ removing a node

Insert Operation :

The operation  $p = \text{getnode}();$

assigns the address of an empty node to p



The value say x is stored in this node by the statement

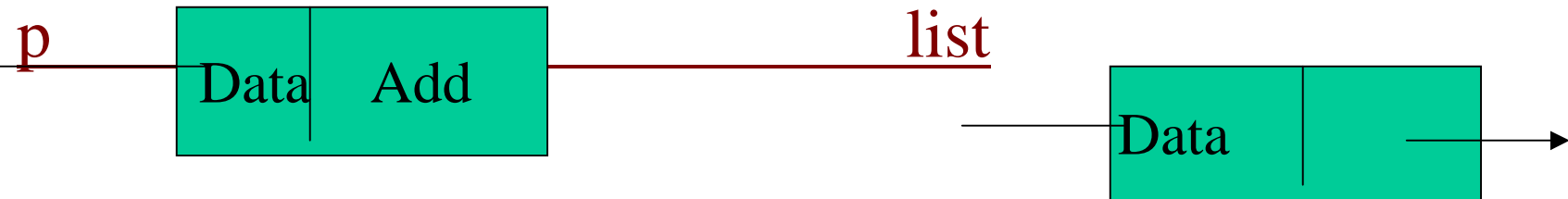
$\text{info}(p) = x;$

# OPERATIONS ON LINKED LIST

This new node is attached to an already existing list by executing the statements

$\text{next}(p) = \text{list};$

$\text{list} = p;$



# OPERATIONS ON LINKED LIST

See page non172 in your text book

```
p= getnode();  
info(p)=6;  
next(p)=list;  
list = p;
```

# Delete Operation

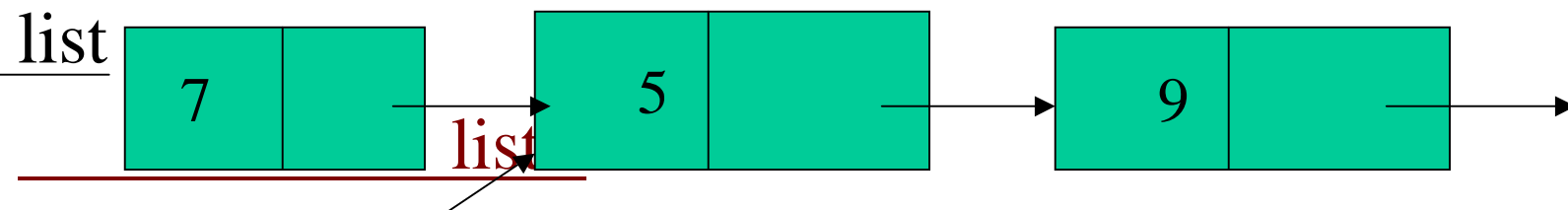
```
p = list;  
list = next(p);  
x = info(p);
```

The data stored in the node pointed to by **list** (now p as well) is copied to x. Now the first node in the initial linked list (ie the node pointed to by p now) cannot be reached through list, since it is list, since it is not a part of our new linked list.

This node need to be destroyed and the memory space should be released. This is done by the statement.

```
freenode(p);
```

x=7



# Delete Operation

After this operation `freedom(p)`, it is illegal to reference `node(p)` since node pointed to by `p` is freed from the linked list

Exercise :

- 1) What are the advantage of arrays over linked lists?
- 2) Will the `getnode()` operation get back a node that was freed by `freenode()` operation?
- 3) Write a procedure to insert a node after 'N' nodes in a linked list.
- 4) Write a procedure to remove the N<sup>th</sup> node of a linked list
- 5) Write a procedure to interchange the N<sup>th</sup> and M<sup>th</sup> nodes in a linked list( Do not interchange the content);

## The Getnode and Freenode Operations

Empty pool - cannot accessed by the programmer except through getnode() and freenode() function is in the form of Linked List

The list of available node is called the available list

How to check the available list is empty or not ?

```
p=getnode();  
if (avail == null){  
    print("Overflow");  
    exit(1);}  
  
p = avail;  
avail = next(avail);
```



## *Implementation of Stacks using Linked List*

A stack can be implemented by means of a linked list...

The operation Push(s,x) can be implemented as:

```
p = getnode();
```

```
info(p) = x;
```

```
next(p) = s;
```

```
s = p;
```

empty(s) can be written as

```
s == NULL;
```

The operation `empty(s)` is merely a test of whether a `s` equals `null`.

The operation `x = pop(s)` removes the first node from a nonempty list and signals `underflow` if the list is empty:

```
if(empty(s){    printf("Stack Underflow");
                exit(1);
            }
else           {    p =s;
                  s = next(p);
                  x = info(p);
                  freenode(p)
            }
```

## Implementation of Queues using Linked List

Items are deleted from the front of the queue and items are added at the rear of the queue.

A queue can be implemented as a list with two pointers corresponding to `q.front` and `q.rear`.

The algorithm for `x = remove(q)` is therefor

```
if (empty(q)){    printf("Q Underflow"); exit(1);}
p = q.front;
x = info(p);
q.front = next(p);
if(q.front == null)    q.rear = NULL;
freenode(p);
return(x);
```

# Implementation of Queues using Linked List

The operation **insert(q,x)** is implemented by :

**Insert(q,x)**

```
{  p=getnode();  
    info(p) = x;  
    next(p) = NULL;  
    if(q.rear == NULL)  
        q.front = p;  
    else  
        next(q.rear) = p;  
    q.rear = p;  
}
```

```

remove(q)
{
    if (empty(q))    {
        printf(“%s \n “, Q Underflow”); exit(1);
    }

    p=q.front;
    x=info(p);
    q.front = next(p);
    if(q.front == NULL)
        q.rear == NULL;
    freenode(p);
    return(x);
}

```

Note :

- (a) Initialize  $q.front = q.rear = NULL$ ;
- (b) When the last element is removed  $q.rear$  must be set to  $NULL$
- c)  $empty(q) \rightarrow q.rear == NULL$

Exercise :

Compare the advantage and disadvantages of representing a stack or queue ?

### Disadvantage

- \* Extra Memory
- \* Additional process needed to maintain avail

### Advantage

- \* stack and queues using the same memory space

## Insert a Node after a node pointed to by p :

To insert an element 'x' into a list after a node pointed to by p:

```
q=getnode();
```

```
info(q) = x;
```

```
next(q) = next(p);
```

```
next(p) = q;
```

Note :

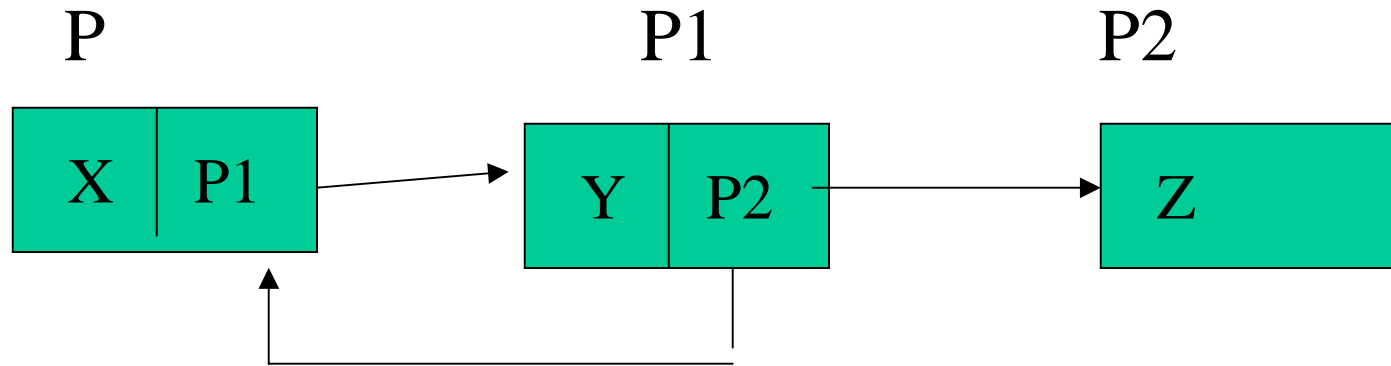
In a singly linked list, we cannot insert an item before a node pointed to by p - Why ?

This effect can be achieved by inserting a node after the node pointed to by p and interchanging info(p) and x (info field of the newly added node)

Similarly a node pointed to by p cannot be deleted Why ?



If the node pointed to by **p** is the last node, then save the contents of the node pointed to by p1 by executing the following operation.





## Delete After

delafter(p,x)

int x;

```
{ struct node *q;  
  q=next(p);  
  x = info(q);  
  next(p) = next(q);  
  freenode(q);  
  return(1);  
}
```

and save the contents of the node pointed to by q in the info filed of the node pointed to by p. **Info(p) = x;**

## An Example of operation on a list :

Remove all the occurrences of a particular value 'k' from a list say "list"

Since we need to know the predecessor of a node in order to delete that node, we use two pointers p and q to traverse the list. We use pop operation to remove a starting node and del after to remove an intermediate node.

```

q = null;    p = list;    // q always points to the predecessor
while(p!=NULL)
{
    if(info(p) == k)
        if(q == NULL)
        {
            x = pop(list);  p = list; // remove 1st node
        }
        else                // delete the node after q and
        {
            p= next(p); delafter(q,x); // move up p
        }
    else                    // traversing the list
    {
        q = p;              p = next(p);
    }
}

```

## Ordered List :

A list in which the contents of the info. **Field of the nodes are ordered in some manner ( eg. Asscending or descending)**

An example **operation on an Ordered List**

**place** :- This is an operation to insert an element into an ordered list in its proper place. We use ‘push’ operation to add a node to the front of the list and “insafter” operation to add in an intermediate position

```
place(list,x);
```

```
q = NULL;
```

```
for(p=list; p!=NULL && info(p) < x; p = next(p))
```

```
    q = p;
```

```
if (q == NULL)    push(list,x)
```

```
else              insafter(q,x)
```

## Efficiency of Place Operation :

What is the average no. of nodes accessed while inserting a new node (elements) into the list ?

The new element say 'X' can be placed in one of  $n+1$  positions ie before 1,2,...n nodes or after n th node.

If  $x < \text{info}(\text{list})$ , then one node need to be accessed.

If  $x$  is between  $k$  th and  $(k+1)$  th nodes, then  $(k+1)$  nodes need to be accessed.

The average number of nodes accessed say 'm' is given by **adding the products of probability of inserting at a particular position** and the **number of access required to insert an element** at that position.

n

n

$$A = E(m) = \sum_{I=1}^n m * p(m)$$

$$= \sum I * \frac{1}{(n+1)} + \frac{1}{n+1}$$

$$= \frac{n(n+1)}{2} * \frac{1}{n+1} + \frac{n}{(n+1)} = \frac{n}{2} + \frac{n}{n+1}$$

$$A = \frac{n}{2} + \frac{1}{(1/n + 1)}$$

For large value of n,  $A \sim \frac{n}{2} + 1$   
 $\sim \frac{n}{2}$

Therefore the operation of randomly inserting an element into an ordered list requires about  $n/2$  node accesses for large value of n, on an average.

# Allocating / Freeing Variables Dynamically

Allocating :

```
int *p, *q, x;
```

```
p=(int *)malloc(sizeof(int));
```

```
*p=3;
```

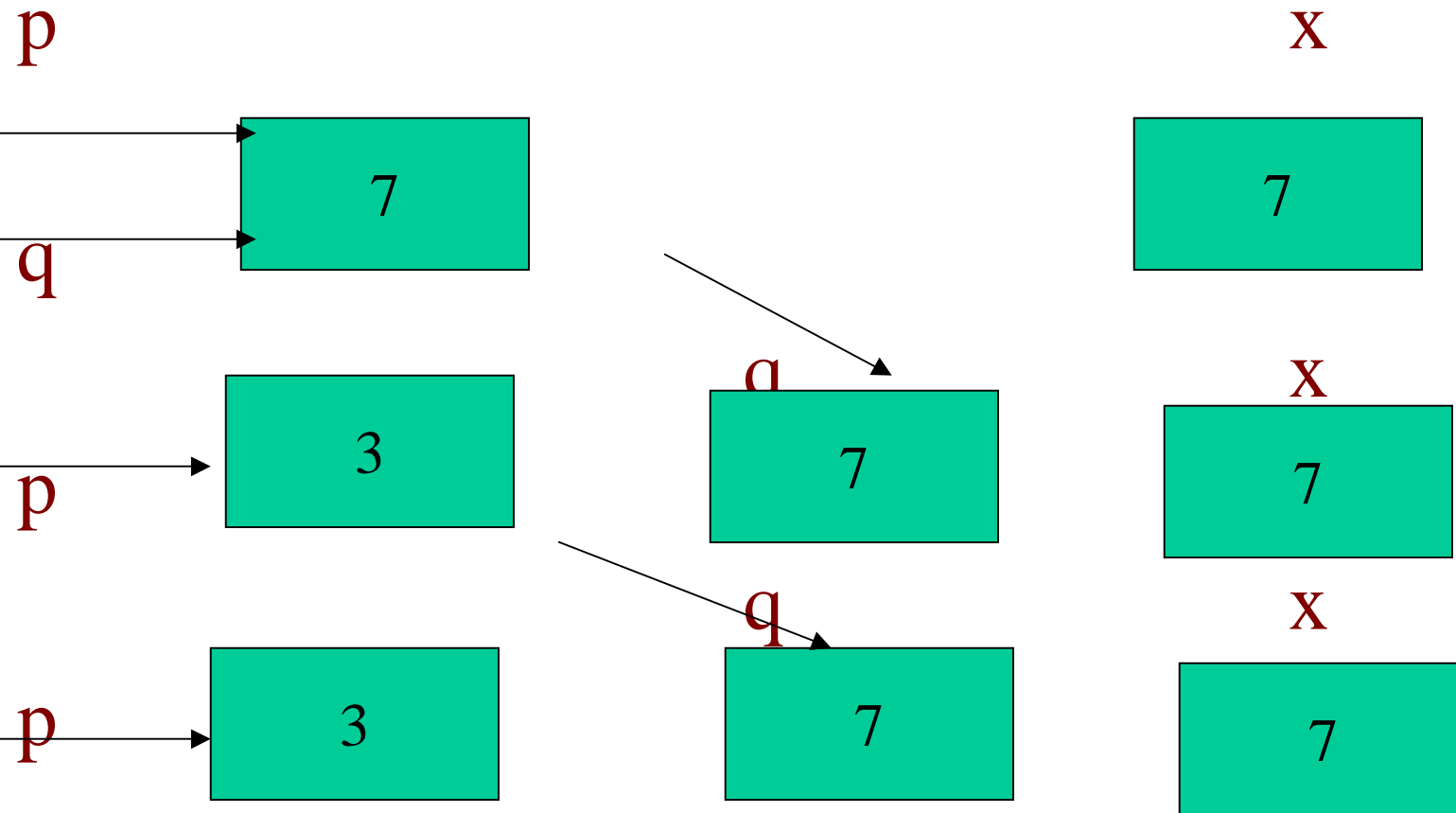
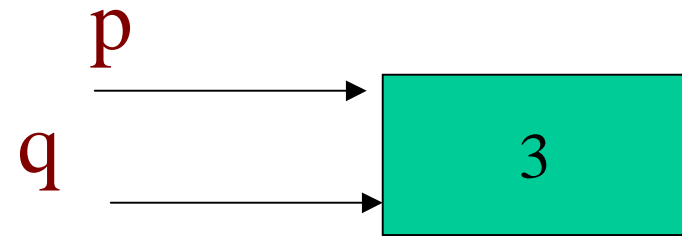
```
q=p;
```

```
printf(“%d %d\n”, *p,*q);
```

```
x = 7;    *q = x;
```

```
printf(“%d %d \n”,*p,*q);
```

# Allocating / Freeing Variables Dynamically





# Allocating / Freeing Variables Dynamically

What are the values of \*p and \*q printed at various storage ?

Note :

malloc returns a character pointer. So type casting is necessary.

# Freeing a Dynamic Variable in C

```
p=(int *)malloc(sizeof(int));  
*p = 5;  
q = (int*) malloc(sizeof(int));  
*q = 8;  
free(p);  
p=q;  
*q = 6;  
printf(“%d %d\n”, *p,*q);
```

What are the values of \*p and \*q during various stages?

# Implementing Linked List in C

Pointer variables are used to implement Linked lists.

```
Struct node {  
    int info;  
    struct node *next;  
};
```

```
typedef struct node *NODEPTR;
```

# Implementing Linked List in C

With this structure declaration, the function getnode() can be defined as

```
NODEPTR getnode()
```

```
{ NODEPTR p;
```

```
  p =(NODEPTR)malloc(sizeof(struct node));
```

```
  return(p);
```

```
}
```

Usage :    p = getnode()

where p is declared as type NODEPTR

# Implementing Linked List in C

The function **freenode()** can be defined as follows :

```
freenode(p)
NODEPTR p;
{
    free(p);
}
```

So **getnode()** can be replaced by

```
p = (NODEPTR)malloc(sizeof(struct node));
```

similarly **freenode()** can be replaced by **free(p);**

# Procedure insafter can be implemented as follows :

```
insafter(p,x);  
NODEPTR p;  
int x;  
{  
    NODEPTR q;  
    if ( p == NULL)  
        { printf("Underflow"); exit(1); }  
    q = getnode();           q --> info =x;  
    q -->next = p ---> next;  p ---> next = q;  
}
```

## Procedure delafter can be implemented as follows :

```
delafter(p,px);  
NODEPTR p;  
int *px;  
{  
    NODEPTR q;  
    if (( p == NULL) || (p --> next == NULL))  
        { printf("Underflow");  exit(1); }  
    q = p --> next;          *px = q --> info;  
    p -->next = q ---> next; freenode(q);  
}
```

# Implementing Qs as Lists

```
struct queue
```

```
{ NODEPTR front, rear; };
```

```
struct queue q;
```

For an empty Q, front = rear = NULL;

```
empty(pq)
```

```
struct queue *pq;
```

```
{ return(pq --> front == NULL ? TRUE : FALSE)  
}
```



# Implementing Qs as Lists

```
insert(pq,x)
struct queue *pq;
int x;
{ NODEPTR p;
  p = getnode();
  p --> info = x;   p --> next = NULL;
  if(pq --> rear == NULL)
    pq --> front = p;
  else
    (pq --> rear) --> next = p;
  pq --> rear = p;
}
```

# Implementing Qs as Lists

```
remove(pq)
struct queue *pq;
{ NODEPTR p;
  int x;
  if (empty(pq))
    { printf("\n Underflow"); exit(1);
    }

  p = pq --> front;          x = p --> info;
  pq --> front = p --> next;
  if(pq --> front == NULL)   pq --> rear = NULL;
  freenode(p); return(x);
}
```

## Place Operation ( To Insert an element in an Ordered List

place (plist, x)

NODEPTR \*plist;

int x;

{

    NODEPTR p,q;

    q = NULL;

    for (p =\*plist; p != NULL && x > info(p); p = p -->next)

        q = p;

    if(q == NULL) push(plist,x)

    else insafter(q,x);

Usage :- place(&plist,x);

## Procedure to insert an element x at the end of a List

```
insend(plist,x)
```

```
NODEPTR *plist;
```

```
int x;
```

```
{
```

```
    NODEPTR p,q;
```

```
    p = getnode();
```

```
    p-->info = x;
```

```
    p-->next = NULL;
```

```
    if(*plist == NULL)
```

```
        *plist = p;
```

```
    else {
```

```
        for(q = *plist;q-->next != NULL;
```

```
            q = q-->next)
```

```
        ;
```

```
        q --> next = p;
```

```
    }
```

```
}
```

## Procedure to search the first occurrence of x in a List

NODEPTR search(list,x)

NODEPTR list;

int x;

{

    NODEPTR p;

    for(p = list; p!= NULL; p= p --> next)

        if (p --> info == x)

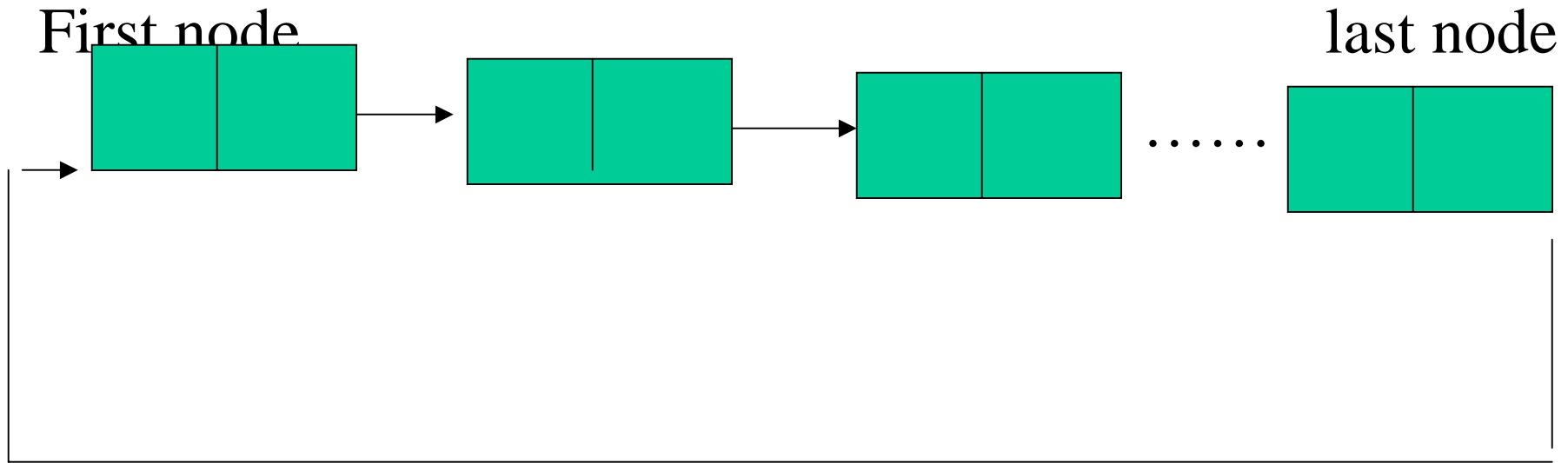
            return(p);

        return(NULL);

}

# CIRCULAR LINKED LIST

A singly linked list with last node pointing to the first node

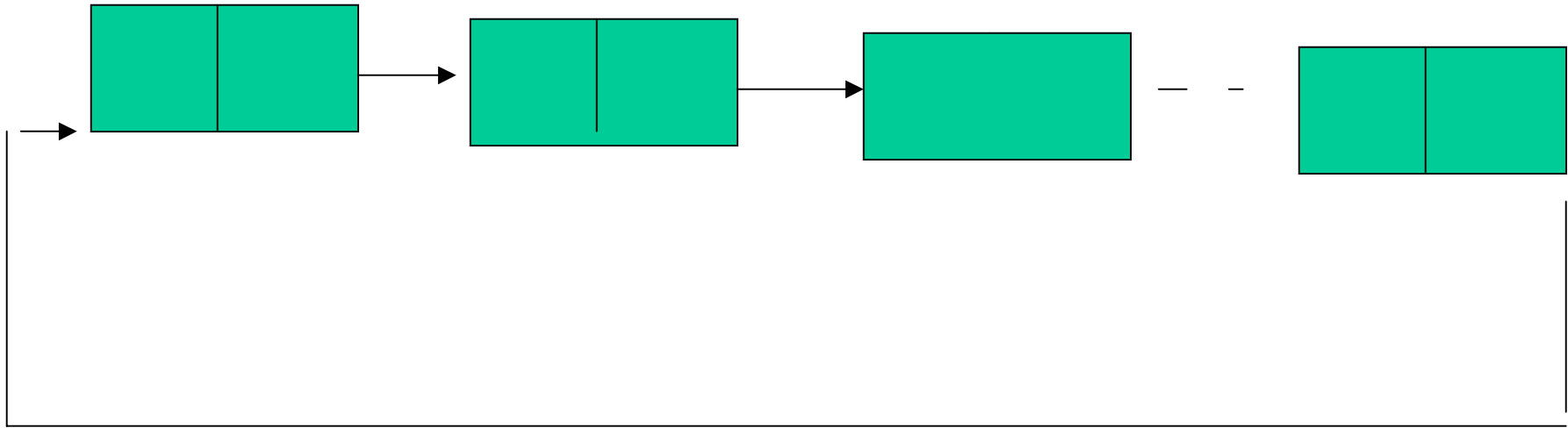


A circular list LL does not have a natural first and last node.

We need to establish first and last nodes by convention. Let the external pointer point to the last node and the following node be the first node

# CIRCULAR LINKED LIST

If  $p$  is an external pointer, then



$\text{node}(p)$  - last node

$\text{node}(\text{next}(p))$  - first node

## Stack as a CIRCULAR LIST

Let Stack be a pointer to the node and let the node following this node (first node) be the top of the stack.

\_\_\_An empty stack is represented by a NULL list.

\_\_\_Empty(pstack)

```
NODEPTR *pstack;
```

```
{ return(*pstack == NULL) ? TRUE:FALSE);  
}
```



## Stack as a CIRCULAR LIST

```
push(pstack,x)
    NODEPTR *pstack;
    int x;
{
    NODEPTR p;
    p = getnode();
    p --> info = x;
    if(empty(pstack) == TRUE)          *pstack = p;
    else
        p --> next = (*pstack) --> next;
    (*pstack) --> next =p;
} /* End push */
```

## POP Operation

```
pop(pstack)
NODEPTR *pstack;
{
    int x;
    NODEPTR p;
    if (empty(pstack) == TRUE)
    {
        printf(“%s \n”, “Stack Underflow”);
        exit(1);
    }
}
```

## POP Operation

p=(\*pstack) --> next;

x = p --> info;

if( p == \*pstack)

    \*pstack = NULL;

else

    (\*pstack) --> NEXT = p --> next;

freenode(p);

return(x);

} /\* End pop \*/

## QUEUE as a Circular List

We use **two pointers** while implementing a queue by means of a linear linked list.

A Q can be implemented by a **Circular LL** by using only one pointer.

If **node(q)** is the rear of the Q then the **node next** to **node(q)** is the **front** of the Q.

## QUEUE as a Circular List

```
insert(pq,x);
```

```
NODEPTR *pq;
```

```
{
```

```
    NODEPTR P;
```

```
    p = getnode();
```

```
    p --> info = x;           // Empty function is the same as
```

```
    if (empty(pq) == TRUE) // as that in stack
```

```
        *pq = p;
```

```
    else    p --> next = (*pq) --> next;
```

```
    (*pq) --> next = p;
```

```
    *pq = p;
```

```
    return; }
```

## QUEUE as a Circular List

```
deafter(p,px);  
NODEPTR p;  
int *px;  
{  
    NODEPTR q;  
    if(( p == NULL) || ( p == p --> next))  
        /* circular list is either empty */  
        /* or it contains only one node */  
        {  
            printf("\n Underflow");  
            exit(1);  
        }  
}
```

## QUEUE as a Circular List

q = p --> next;

\*px = q --> info;

p --> next = q --> next;

freenode(q);

return;

} /\* End delafter\*/

# DOUBLY LINKED LIST

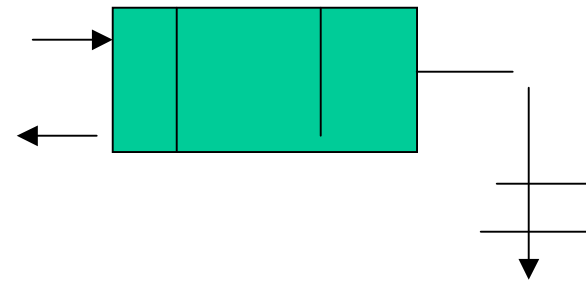
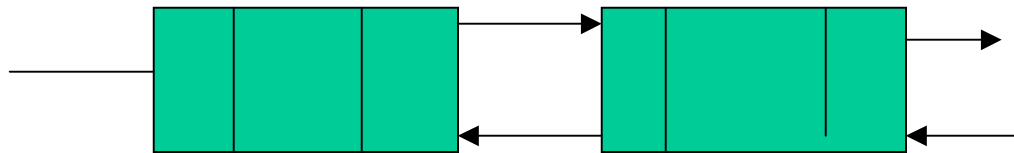
## Disadvantage of a Linear Circular List/ non-circular LL:

- backward traversal not possible

## Doubly LL

- Linear Doubly LL
- Circular Doubly LL

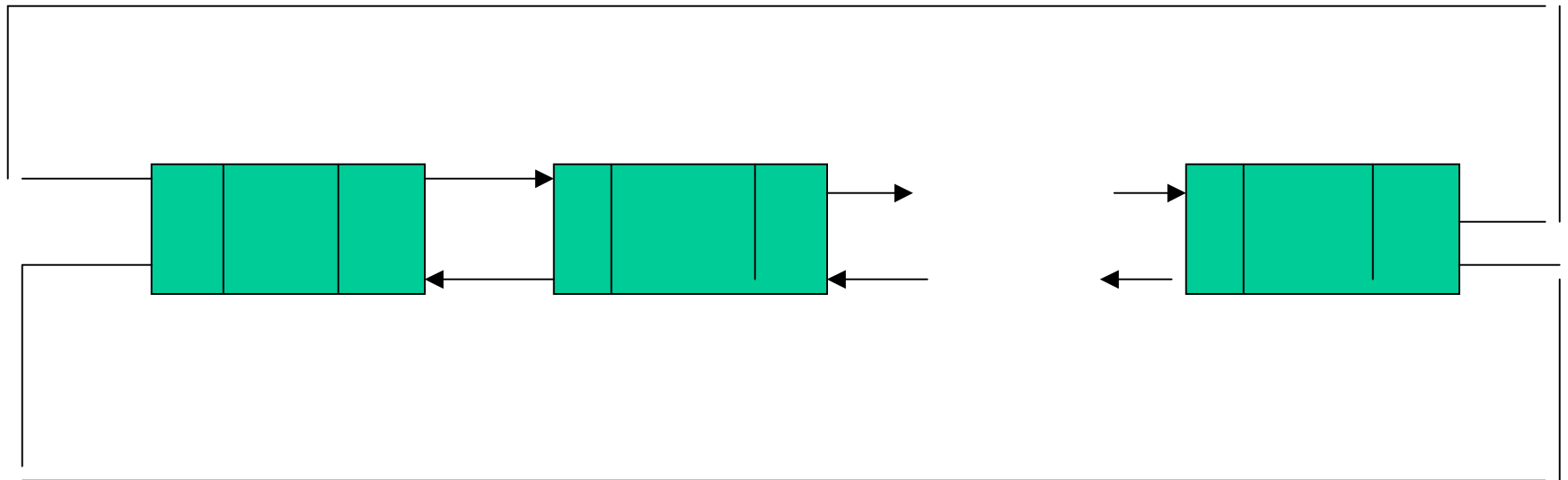
## Linear Doubly LL;





# DOUBLY LINKED LIST

## Circular Doubly LL



# DOUBLY LINKED A

A node in a doubly LL consist of three fields :

- info - contains info(data) stored in that node
- left - contains the address of the left node ( a pointer)
- right - contains the address of the right node

Abbreviations:

left(p) stands for  $p \rightarrow \text{Left}$

right(p) stands for  $p \rightarrow \text{right}$

$\text{left}(\text{right}(p)) = p = \text{right}(\text{left}(p))$

## DOUBLY LINKED A

A node in a doubly LL can be declared as

```
struct node
{
    int info;
    struct node *left, *right;
};
typedef struct node *NODEPTR;
```

## DOUBLY LINKED A

```
delete(p,px)
NODEPTR p;
int px;
{
    NODEPTR q,r;
    if( p == NULL)
    {printf(“%s\n”, “Underflow”);
        exit(1);
    }
```

```
*px = p --> info;
q = p --> left;
r = p --> right
q-->right = r;
r -->left = q;
freenode(p);
}
```

## Inserting a node to the right of the node pointed to by a DLL

```
inserttright(p,x)
```

```
NODEPTR p;
```

```
int x;
```

```
{
```

```
    NODEPTR q,r;
```

```
    if( p == NULL)
```

```
    {printf(“%s\n”, “Underflow”);
```

```
        exit(1);
```

```
    }
```

```
q = getnode();
```

```
q -->info = x;
```

```
    r = p --> right
```

```
r --> left = q;
```

```
q --> right = r;
```

```
q --> left = p;
```

```
p --> right = q;
```

```
return;
```

```
}
```

# THE END OF LESSON IV