Data Structures Using C

Linked Lists

Lesson Objectives

- **To understand the concept of Linked List, and its operations like:**
  - Insert a node
  - Delete a node
  - Modify a node

- **To analyze applications of Linked Lists**

- **To understand the concept of trees, binary trees**

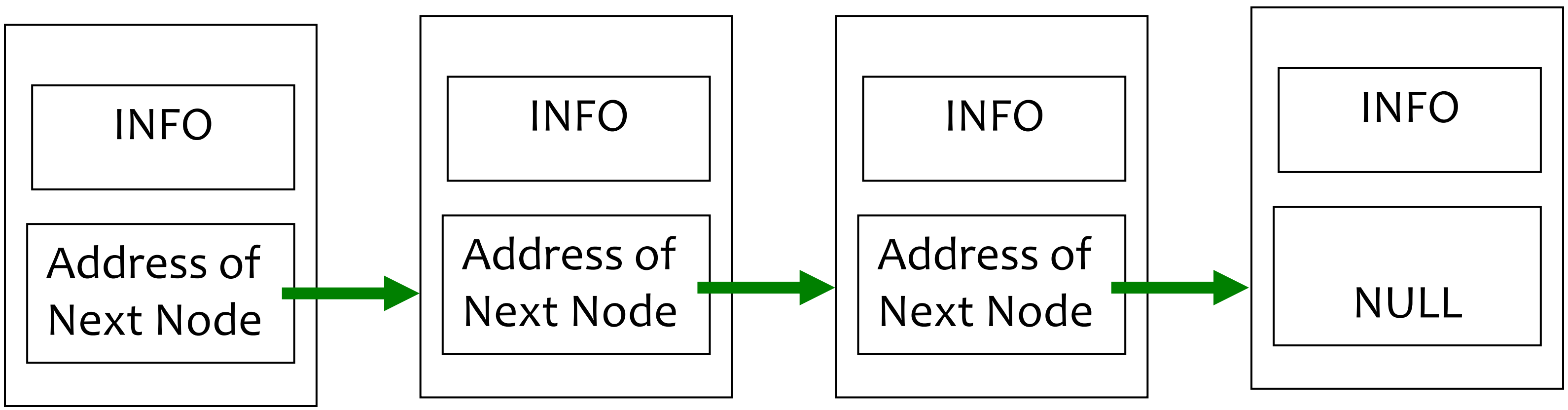
Where to Use Linked List?

- **Consider a program to be processed**
  - We have files in a folder or in a directory tree
  - We have a mailing list of users for sending a New Year mail

- **Can we use “Arrays” to process such data?**
  - Arrays are good to use when the number of elements is fixed, and known.
  - If this number varies at runtime, and the variation is quite high, then Arrays may not be the best data structure.

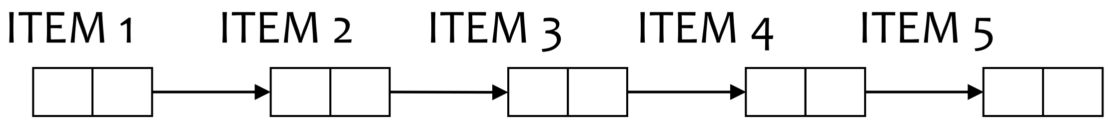
What Is Linked List?

- **A Linked List has two components for each item in the list:**
  - Data Value
  - Pointer to the next element



What Is Linked List?

DATA VALUES …	LINK OR POINTER TO NEXT ELEMENT



We also need a pointer to the start of the list.

DATA VALUES …	POINTER TO NEXT ELEMENT

LIST_HEAD = 4 /* INDEX OF FIRST ITEM IN THE LIST */

ELEMENT # 1	SHEELA	3
ELEMENT # 2	PALAK	0
ELEMENT # 3	ROHAN	2
ELEMENT # 4	RAMESH	1

IGATE
Speed.Agility.Imagination

Operations

- **Operations performed on a list are:**
  - Insert an element
  - Search an element
  - Modify an element
  - Delete an element

Elements (Contd…)

- **A Linked List is made up of multiple elements.**
  - Each element has two parts:
    - a Data part, and
    - an Address or Pointer to the next element in the list

- **To create a Dynamic List, we need a mechanism to dynamically "allocate" and "de-allocate" the "new elements" at runtime**
  - A function like "malloc" to allocate memory, and "free" to de-allocate memory needs to be used.

Elements (Contd…)

- **Assuming the memory is correctly allocated, we can access it as follows:**
  - List_ptr → data
  - List_ptr → next
  - The basic steps are:
    - Declare the appropriate Structure
    - Declare a Pointer to the defined Structure
    - Allocate "dynamic memory" and initialize the Pointer
    - Access the allocated memory by using the Pointer and Structure
    - De-allocate the memory when it is no longer required

Structure of the Node - Declaration

- **The Linked List can be declared as follows:**

```
struct node
    {
        data_type info;
        struct node * next ;
    };
```

where,

- Info – It is the information of the node. It can be a record or any member. There is no limitation for members.
- Next – It is the pointer to next node in the list. It is NULL for last node.

Creating Linked List - Process Steps

- **Steps for creating a Linked List are as follows:**
  1. Define the Structure for the Linked List:
     - Let us assume that the data we intend to store is the empid of the emp (which is unique), his name, and salary.
     - Then the structure we require will be defined as follows:

```
struct node
{
  int empid;
  char name[20];
  float salary;
  struct node *next;
};
```

Creating Linked List - Process Steps (Contd…)

2. Create a node, i.e., define the getnode() function.

```
struct node *getnode()  /* creates a node and accepts data */
{
        struct node *temp;
        temp=(struct node *)malloc(sizeof(struct node));
        printf("enter the empid:");
        scanf("%d",&temp->empid);
        fflush(stdin);
        printf("enter the name:");
        scanf("%s",temp->name);
        fflush(stdin);
```

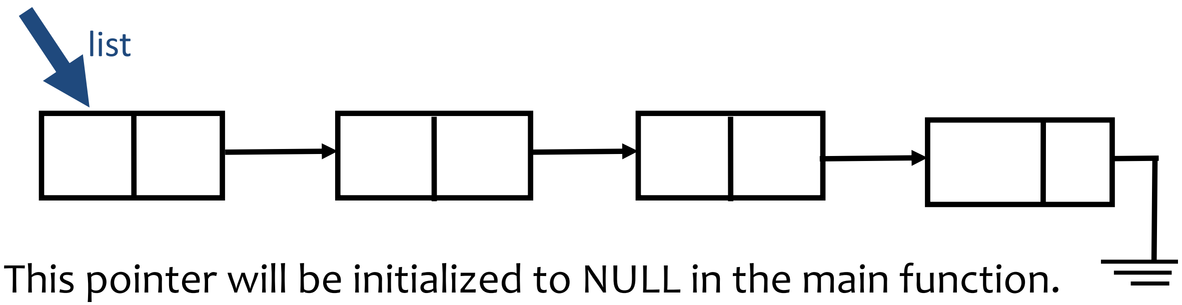

IGATE
Speed.Agility.Imagination

Creating Linked List - Process Steps (Contd…)

```
printf("enter salary:");
    scanf("%f",&temp->salary);
    fflush(stdin);
    temp->next=NULL;
    return temp;
}
```

Creating Linked List - Process Steps (Contd…)

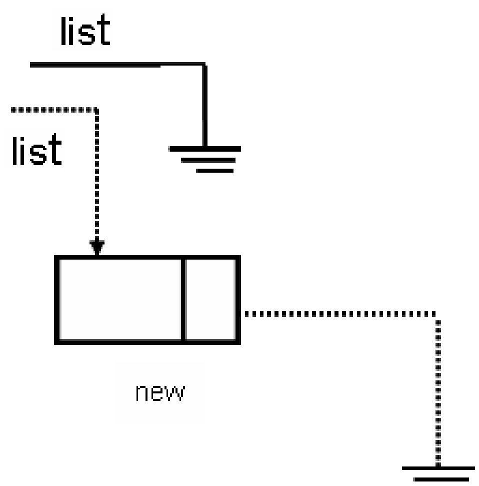
2. Check the start of list pointer's declaration.
   - The list needs a pointer that will point to the beginning of the list.



   - This pointer will be initialized to NULL in the main function.

Insert First Node

- **Defining the Insert function:**
  - This function will take the "list pointer" (indicate Starting of the list) and "address of the node" to be inserted as a parameter. If the list is empty, then the new node becomes the first node:

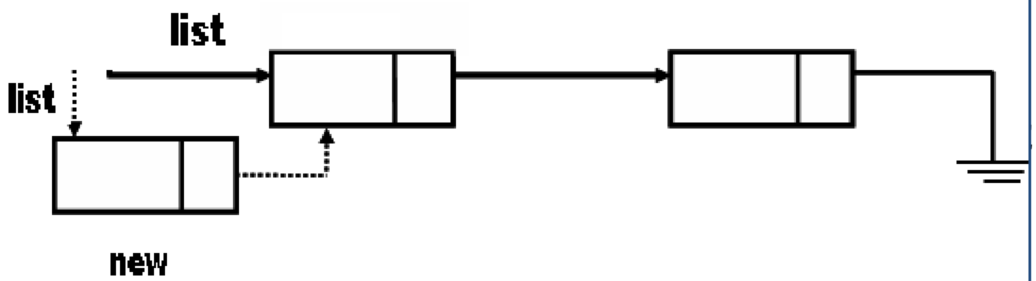


Different Scenarios

- **Insert function should take care of the following:**
  1. Inserting a Node at the Beginning of the list
  2. Inserting a Node in the Middle of the list
  3. Inserting a Node at the End of the list

Different Scenarios (Contd…)

1.



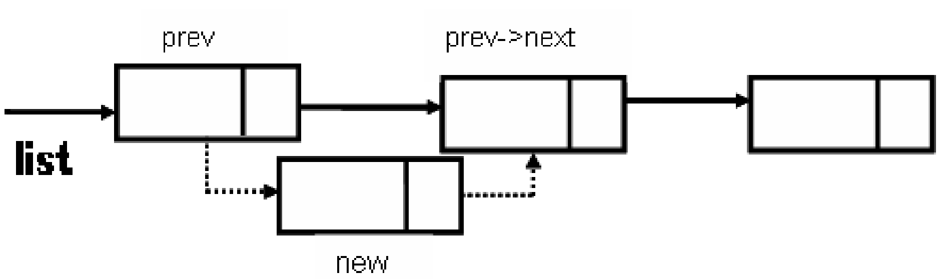
The new node has to point to the existing first element of the list. This is done by the following statement:

new -> next=list;

The list pointer must point to the new node:

list=new;

2. **Inserting a Node in the Middle of the list**



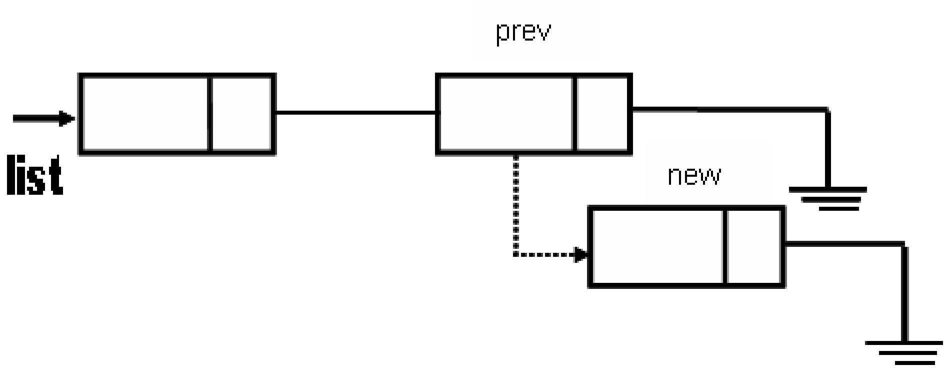
The new node has to be inserted after the node pointed by prev node, then previous node should point to new, and new will point to next of prev.

new ->next = prev-> next;

prev -> next = new;

Different Scenarios (Contd…)

3. Inserting a Node at the End of the list



The last node should point to new node, and new node should point null to become last node.

new -> next = null

which is equivalent to

new-> next = prev-> next;

then

prev-> next = new;

Illustration

- **Example 1:**

Insert Function: Inserting a node

```
int insert(Struct node *list,struct node *new)
{
        struct node *prev;
    int flag;
    if (list==NULL)  /* list empty */
    {
        list=new;
         return 0;
    }
    prev=search(new->empid,&flag);
```

Illustration (Contd…)

```
if(flag==1)  /* duplicate empid */
        return -1;
    if(prev==NULL)  /* insert at beginning */
    {
        new->next=list;
            list=new;
    }
    else  /* insert at middle or end */
    {
        new->next=prev->next;
         prev->next=new;
    }
    return 0;
}
```

Illustration (Contd…)

- **Example 2:**

Search Function: Defining the search(struct node *list) function

```
struct node * search(struct node *list,int cd,int *flag)
{
struct node *prev,*cur;
    *flag=0;
    if (list==NULL)  /* list empty */
       return NULL;
    for(prev=NULL,cur=list;(cur) && ((cur->empid) < cd);
                           prev=cur,cur=cur->next);
```

Illustration (Contd…)

```
if( (cur) && ( cur->empid==cd))
                        /* node with given empid exists */
    *flag=1;
return prev;


}
```

Display Details

- **displayall Function:**

Define the displayall(Struct node *list) function as shown below:

```
/*traverse the list sequentially and  print the details*/
void displayall(struct node *list)
{
    struct node *cur;
    if(list==NULL)
    {
        printf("list is empty\n");
        return;
    }
```

Display Details (Contd…)

```
printf("empid, name, salary\n");
for(cur=list;cur;cur=cur->next)
 {
      printf("%4d%-22s%8.2f\n",cur->empid,cur->name,
                          cur->salary);
}
}
```

Modify a Node

- **modify Function:**

For modifying a node, search for the node and accept the details again.

```
int modifyt(Struct node*list,int emp_id)
{
    struct node *cur;
    int flag;
    char ans='Y';
    if (list==NULL)  /* list empty */
    {
       printf("The list is empty")
        return 0;
    }
    cur=search(new->empid,&flag);
```

Modify a Node (Contd…)

```
if(flag==1)  /*record found for modification
{
    printf(“The current record is :”);
            printf("%4d%-22s%8.2f\n",cur->empid,cur->name,
                        cur->salary);
            printf(“Do you want to modify the record t(y/n)”);
            scanf(“\n%c”,&ans);
            if(ans==‘y’)||(ans=‘y’)
        {
                printf(“Enter new record”);
             printf("enter the empid:");
```

Modify a Node (Contd…)

```
scanf("%d",&cur->empid);
            fflush(stdin);
            printf("enter the name:");
      scanf("%s",cur->name);
      fflush(stdin);
      printf("enter salary:");
      scanf("%f",&cur->salary);
       }
    }
   else
            printf("Record not found");
  return 0;
}
```

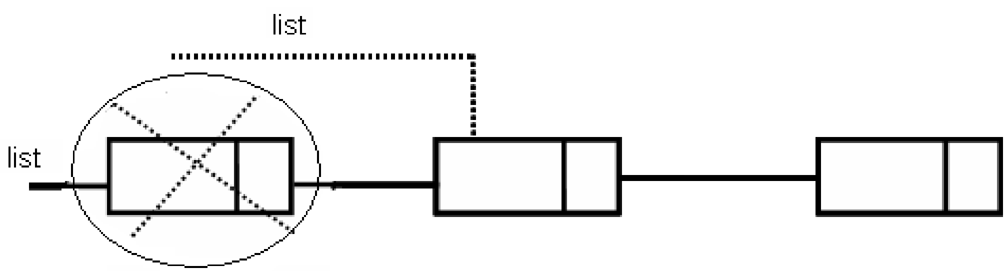
Delete a Node

- **To delete a node, the links have to be reformulated to exclude the deleted node. The memory allocated for the deleted node must also be freed.**

- **The node to be freed can exhibit the following traits:**
  - It may not be existing in the list.
  - It may be the first node (in which case the list pointer must be reinitialised).
  - It may be any other node or the list may be empty.

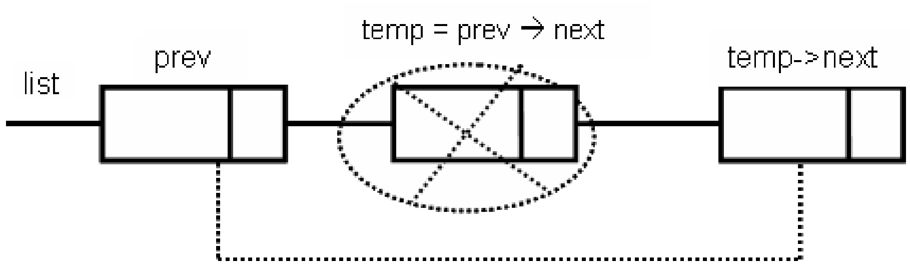
Delete a Node (Contd…)

- **To delete the first node:**




```
temp=list;

/* where temp is
  defined as struct node*/

  list = list->next;

  free(temp);
```

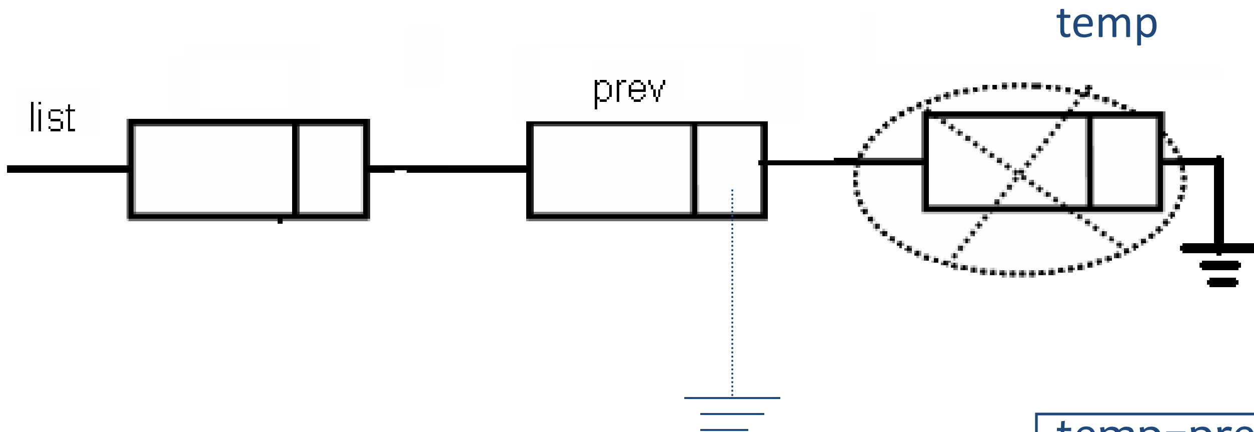
-




```
temp=prev->next;

/* prev is the node
  returned by search */

prev->next=temp->next;

free(temp);
```

Delete a Node (Contd…)

- **To delete the node from the end:**




```
temp=prev->next;
/* prev is the node
  returned by search */
prev->next=temp->next;
free(temp);
```

delet Function

- **The delet() function can be coded as:**

```
int delet(struct node *list,int cd)
{
struct node *prev,*temp;
    int flag;
    if (list==NULL)  /* list empty */
                    return -1;
    prev=search(cd,&flag);
    if(flag==0)  /* empid not found */
                    return -1;
  if(prev==NULL)
```

delet Function (Contd…)

```
/* node to delete is first node (as flag is 1) */
   {
                    temp=list;
      list=list->next;
      free(temp);
   }
   else  /*delete node from middle or from the end*/
   {
                    temp=prev->next;
                    prev->next=temp->next;
                    free(temp);
   }    return 0;
}
```

Circular Linked List

- **What change is required to the Structure Definition?**
  - A “Linked List” is a data structure, where we can add a “new element”:
    - at the “start” of the List,
    - at the “end” of the List, or
    - in a “sorted manner”
  - We can also remove any element from the “Linked List”.
  - If the pointer of the last node, points to the first node in the list, then such a List is called a “Circular Linked List”.

Use of Linked List

- **Consider the files in a folder or directory.**

- **If we want to write a program to manipulate all files in a folder, "Arrays" are not the best data structures to be used!!**
  - An "Array" is a "static" data structure, whose size cannot be changed dynamically at runtime.
  - If the Array Size is too small, we may run out of space!
  - If Array Size is too large, we end up wasting space.

- **If elements are deleted, we may leave holes in the array.**
  - We then need to keep track of the holes, and that is complicated!

Problems

- **Problems (issues) faced in using Linked Lists are:**
  - The number of files in a folder is variable.
    - A folder may be empty, may have 5-10 files, or may even have 500 files.
  - It may be necessary to sort the files
    - Sort can be sometimes on filename, sometimes on file size, or some times on date / time of creation

- **A better option is to use a "Dynamic Linked List".**

Demo

- **Linked List demo**

Lab

- **Lab 2 (Lab on Linked List)**

Data Structures

- **There are two more data structures, which are frequently used:**
  - Queues
  - Stacks

Queues

- **A "Queue" is a data structure, which can be easily implemented as a Dynamic Linked List.**
- **Special characteristics of the Queue are:**
  - a "new item" is always added at the "end" of the list (called as the rear-end of the Queue),
  - an item to be removed is always removed from the "start" or "head" of the list (called as the front-end of the List)
- **This means that data can be added at rear-end, and can be removed from front-end.**

Queues (Contd…)

- **A Queue uses the FIFO concept.**
- **Queues are useful in many applications**

  For example:
  - Sharing CPU time between users / processes in a Queue
  - Queuing up requests on a shared printer
- **Like the Linked List, the size of a Queue can be unpredictably short or long, as well.**
- **Hence, Dynamic structures like Linked Lists are better than Arrays.**

Structure

- **Use the following structure:**

```
struct queue
{
    int data;
    struct queue *next;
}*front,*rear;
```

Different Functions

- **Example:**

```
Typedef struct queue QUEUE;
//To initialize queue
void initqueue()
{
    front=rear=NULL; }
//to check for empty queue
int emptyqueue()
{
    return(front==NULL);}
```

Different Functions (Contd…)

- **Example (Contd…):**

```
//to add element in queue
void insert(int num)
{
    QUEUE *temp;
    temp=(QUEUE *) malloc(sizeof(QUEUE));
    temp->data=num;
    temp->next=NULL;
    if(front==NULL)
        rear=front=temp;
     else
      {
            rear->next=temp;
            rear=temp;
}
}
```

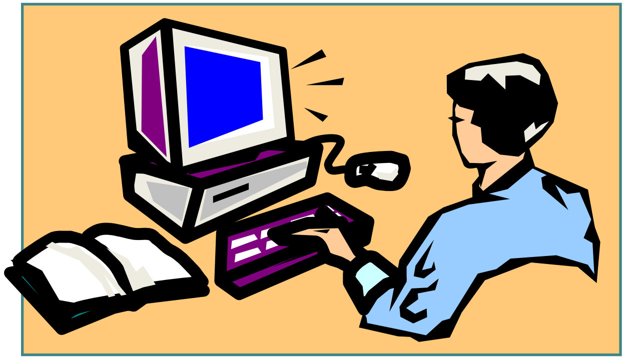
Different Functions (Contd…)

- **Example (Contd...):**

```
//To remove element from queue
int remove()
{
    int num;
    QUEUE *temp=front;
    num=front->data;
    front=front->next ;
    free(temp);
    if(front==NULL)
      rear=NULL;
    return(num);
}
```

Demo

- **Queue using linked list**



Stacks

- **A "Stack" is a Data structure, which can be easily implemented as a Dynamic Linked List.**

- **Special characteristic of a Stack is that:**
  - a "new item" is always added to the "start" or "head" of the list (called as the top),
  - an item to be removed, is removed from the "start" or "head" of the List

- **This means that data can be added or removed only from top.**

Stacks (Contd…)

- **A Stack uses the LIFO concept.**
- **“Stacks” are useful in many applications.**
  - For example: Evaluating postfix expressions

Stacks (Contd…)

- **Operations performed on a Stack are:**
  - Push
    - checks whether stack is full
    - increases top by one
    - adds an element from the top
  - Pop
    - checks whether stack is empty or not
    - Deletes an element from top
    - reduces top by one

Structure

- **Use following structure:**

```
struct stack
{
    int data;
    struct stack *next;
}*top;
```

Use of Different Functions

- **Example:**

```
typedef struct stack * stack_ptr;

#define NODE_ALLOC  (struct stack *) malloc (sizeof(struct stack))

//Initializing stack - Initializes stack top
 void initstack()
{   top = NULL;  }


//Isempty Function returns true if stack is empty false otherwise
 int isempty()
{ return (top == NULL);}
```

Use of Different Functions (Contd…)

- **Example (Contd…):**

```
//push function
void push (int num)
{
  stack_ptr newnode;     //Push a integer in the stack
  newnode=NODE_ALLOC;
  newnode->next=NULL;
  newnode->data=num;
  if(top==NULL)
      top=newnode;
  else
   {
      newnode->next=top;
      top=newnode;
    }
 }
```

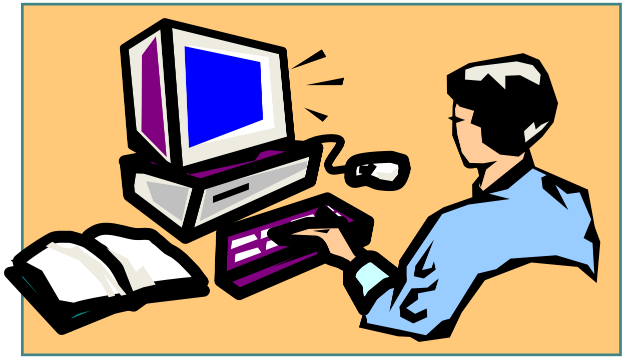
Use of Different Functions (Contd…)

- **Example (Contd…):**

```
//pop function
int  pop()
{//Pops one integer form the top position of the stack
     int num
     stack_ptr temp=top;
     num= top->data;
     top=top->next;
     free(temp);
      return(num);
}
```

Demo

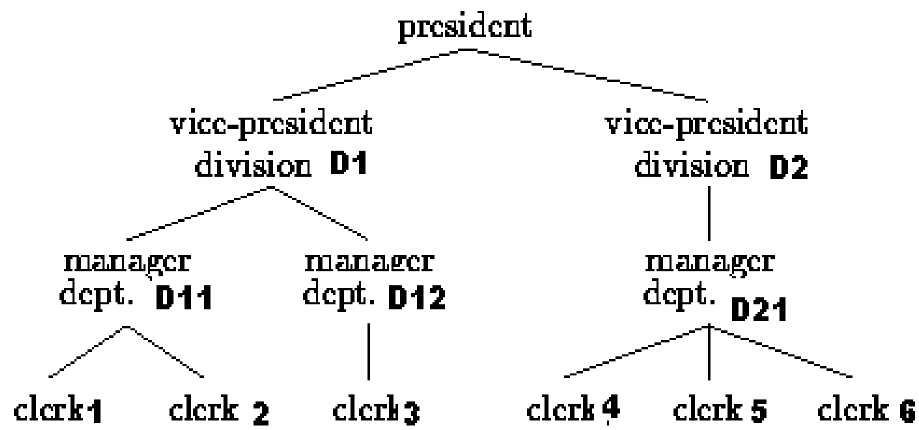
- **Stack using linked list**



Lab

- **Lab 10**
- **Lab 11**

Terminologies



- Non linear data structure
- Edge-connects an element node and its children node
- Siblings(Children of the same parent)-VP div D1 and VP div D2
- Leaves(Elements with no children)-Clerk1,clerk2,… ..
- Degree of tree-Max of its element degree=3;degree(leaf node)=0
- Depth of the tree – Maximum length of leaf node from the root =4

Description

- **A binary tree is a finite (possibly empty) collection of elements**
- **When the binary tree is not empty, it has a root element and the remaining elements are partitioned into 2 binary trees which are called left and right sub_trees of tree t.**

Trees	Binary Trees
Non Empty Collection	Can be empty
Can contain any no:of:subtrees	Can have max 2 subtrees
Contains unordered elements	Contains ordered elements

Types of Binary Tree Traversal

- **There are 2 different types of Tree traversals**
  1. Depth First Traversal
     - Preorder ->Root,Left,Right
     - Inorder ->Left,Root,Right
     - Postorder ->Left,Right,Root
  2. Breadth first Traversal

- **Preorder - j,f,a,d,h,k,z**

- **Inorder - a,d,f,h,j,k,z**

- **Postorder - d,a,h,f,z,k,j**

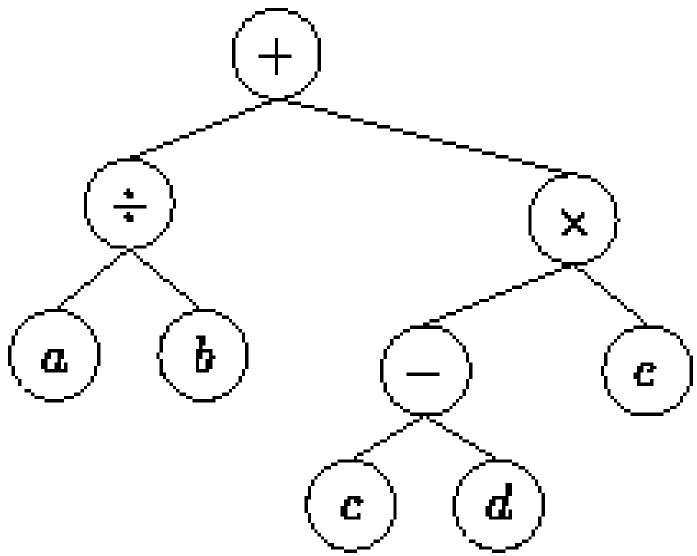
- **Breadth first - j,f,k,a,h,z,d**

```
      tree
      ----
       j     <-- root
     /   \
    f      k
  /   \      \
 a     h      z
  \
   d
```


IGATE Speed.Agility.Imagination

Expression Trees

- **Expression Tree for a/b+(c-d)*e**

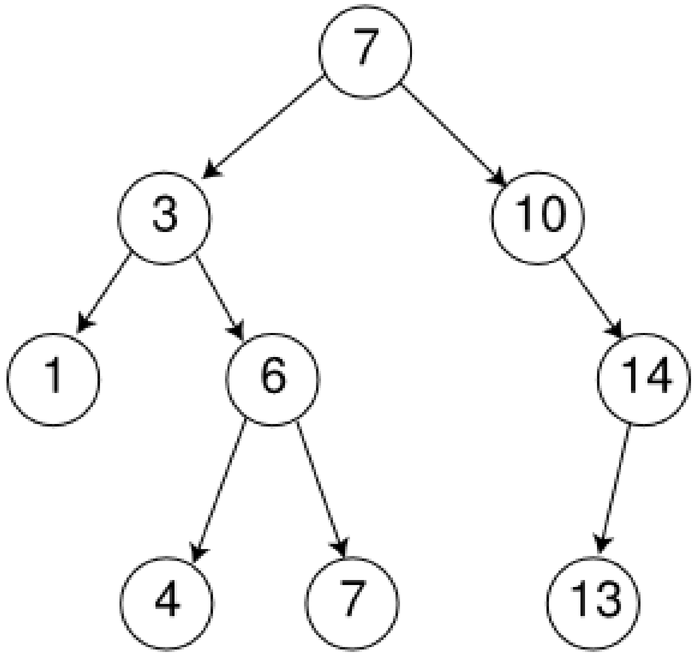


Expression Trees (Contd…)

- **Infix Notation**
  - Operator appears in between its operands
  - Inorder Traversal produces infix notation
- **Prefix Notation**
  - The operator is written before its operands
  - Preorder Traversal produces Prefix Notation
- **Postfix Notation**
  - An operator always follows its operands
  - Postorder Traversal produces Postfix Notation

Binary Search Tree

- **A binary search tree (BST) is a binary tree which has the following properties:**
  - Each node has a value
  - The left subtree < node value
  - The right subtree > node value

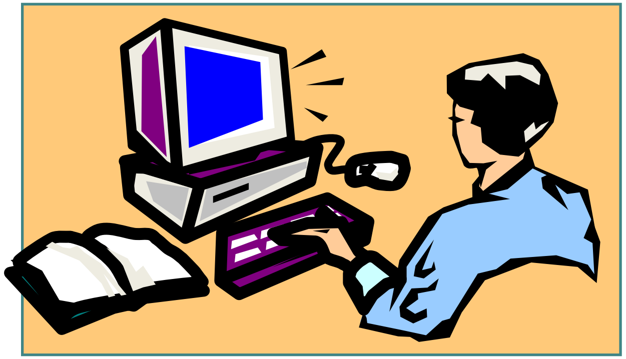


Binary Tree Operations

- **Searching of a node**
- **Insertion of a node**
- **Deletion from a binary tree**

Demo

- **Demo on BinaryTree.c**
- **Code**
  - BinaryTree_search.doc
  - BinaryTree_delete.doc

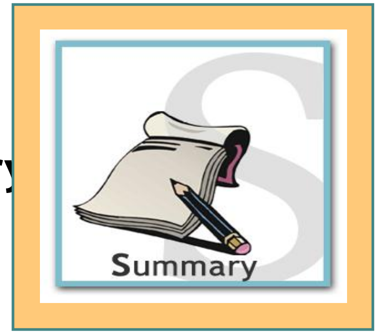


Lab

- **Lab 12**

Summary

- **When the size of the list is not known, it is better to use Linked List.**
- **"Malloc" and "Calloc" functions are useful for dynamic memory allocation.**
- **Linked Lists can be handled in FIFO manner, namely a "Queue".**
- **Linked Lists can be handled in LIFO manner, namely a "Stack".**



Summary

- **Tree is a non linear data structure**
- **Binary tree has maximum 2 child nodes**
- **Types of tree traversal**
  - Depth first search
    - Preorder
    - Inorder
    - Post order
  - Breadth first search

Review Question

- **Question 1: ------- function is use to allocate memory in C**
  - malloc
  - calloc
  - alloc

- **Question 2: -------- are components of each node in a singly linked list**
  - Data
  - Pointer to the next node
  - Pointer to first node
  - Pointer to last node

Review Question

- **Question 3: Queue uses --------- concept**
  - FIFO
  - LIFO
  - Random access
  - Add and remove from any ends

- **Question 4: Inorder travesal produces -----**
  - Infix expression
  - Postfix expression
  - Prefix expression