Data Structures

Module - Agenda

I. Overview of Data Structures
II. Array
III. Linked List
IV. Stack
V. Queue
VI. Tree
VII. Graph Terminology
VIII. Templates

Session I

Overview of Data Structures

I. Overview of Data Structures

Agenda

1. Data
2. Data Structure
3. Abstract Data Type

I. Overview of Data Structures

1. Data

- Data is information that has been translated into a form that is more convenient to move or process.

I. Overview of Data Structures

2. Data Structure

- Data structure provides a way to organize related pieces of information.

I. Overview of Data Structures

3. Abstract Data Type

- A set of data values and associated operations that are precisely specified independent of any particular implementation.

I. Overview of Data Structures

What we have Covered

1. Data
2. Data Structure
3. Abstract Data Type

Session II

Array

II. Array

Agenda

1. Definition
2. Example
3. Alternative Array

II. Array

1. Definition

- An array is a collection of elements.

```
AbstractDataType array
{
    instances
      set of(index,value) pairs, no two pairs have same index
    operations
      get(index) :
        return the value of pair with this index
      set(index,value) :
        add this pair, overwrite existing pair, if
        any, with the same index.
}
```

II. Array

2. Example

- The no of days for each month may be represented by the following array.

  Noofdays = {(Jan,31),(Feb,28),(Mar,31),(Apr,30),(May,31),(Jun,30),(Jul,31),(Aug,31),(Sep,30,(Oct,31),(Nov,30),(Dec,31)}

  Change the no of days for Feb, if leap year, by performing following operation –

  set(Feb,29)

  We can get the no of days for May by performing following operation –

  get(May)

II. Array

3. Alternative Array

- Noofdays= {(0,31),(1,28),(2,31),(3,30),(4,31),(5,30),(6,31),(7,31),(8,30),(9,31),(10,30),(11,31))}

II. Array

What we have Covered

1. Definition
2. Example
3. Alternative Array

Session III

Linked List

III. Linked List

Agenda

1. Definition
2. Singly Linked List
3. Doubly Linked List
4. Circular Linked List

III. Linked List

1. Definition

- A linked list is an ordered collection of elements.
- Each element is represented in a node.
- Each node keeps explicit information about the location of other relevant nodes.
- This explicit information is called a link or pointer.

III. Linked List

2. Singly Linked List

- Each node has exactly one link.
- The nodes are ordered from left to right with each node linking to the next.
- The last node has a NULL link.

III. Linked List

2. Singly Linked List

```
class node
{
public:
        int data;
        node *link;
};

class linklist
{
        node *start;//Points to the first node
public :
        linklist();
        //insert the node at the begining of list
        void insertbeg(int x);
        //insert the node at the end of list
        void insertend(int x);
        void display();
        ~linklist();
};
```

III. Linked List

2. Singly Linked List

```
#include<iostream.h>
#include "linklist.h"
```

```
linklist :: linklist()
{
        start=NULL; //Node initialization
}
```

III. Linked List

2. Singly Linked List

```
void linklist::insertbeg(int x)
{
        node *temp;
        temp=new node;
        if(temp==NULL) //Node creation failed
        {
                cout<<"\nSorry, Overflow";
                return;
        }
        else
        {
                temp->data =x;
                if(start==NULL)
                {
                        start=temp;
                        start->link=NULL;
                        return;
                }
                else
                {
                        temp->link=start;
                        start=temp;
                }
        }
}
```

III. Linked List

2. Singly Linked List

```
void linklist::insertend(int x)
{
	node *temp,*p;
	temp=new node;
	if(temp==NULL)
	{
		cout<<"\nSorry, Overflow";
		return;
	}
	else
	{
		temp->data = x;
		temp->link=NULL;
		if(start==NULL)
		{
				start=temp;
				return;
		}
		else
		{
				p=start;
				while(p->link)
				{
						p=p->link;
				}
				p->link=temp;
		}
	}
}
```

III. Linked List

2. Singly Linked List

```
void doublelinklist::display ()
{
        if(start==NULL)
        {
                cout<<"\nSorry ,no nodes to display";
        }
        else
        {
                node *temp=start;
                while(temp)
                {
                                cout<<"\n\t"<<temp->data ;
                                temp=temp->next;
                }
        }
}
```

III. Linked List

2. Singly Linked List

```
linklist::~linklist()
{
    node *temp;
    while(start)
    {
        temp=start;
        start=start->link;
        delete temp;
    }
    cout<<"\n\nDeleted";
}
```

III. Linked List

3. Doubly Linked List

- Each node has two link previous and next.
- The nodes are ordered from left to right with each node linking to the next.
- There is also a pointer pointing to the previous node.
- The last node has a NULL in its next field.
- The first node has NULL in its previous field.
- This can be used to traverse in any direction.

III. Linked List

3. Doubly Linked List

```
class node
{
public :
        int data;
        node *prev;
        node *next;
};
class doublelinklist
{
        node *start;
public :
        doublelinklist();
        void insertbeg(int x);
        void insertend(int x);
        void display();
        ~doublelinklist();
};
```

III. Linked List

3. Doubly Linked List

```
#include<iostream.h>
#include "linklist.h"


doublelinklist::doublelinklist()
{
        start=NULL;
}
```

III. Linked List

3. Doubly Linked List

```
void doublelinklist::insertbeg(int x)
{
        node *temp;
        temp=new node;
        if(temp==NULL)
        {
                cout<<"\nSorry, Overflow";
                return;
        }
        else
        {
                temp->data =x;
                if(start==NULL)
                {
                        start=temp;
                        start->prev=NULL;
                        start->next=NULL;
                        return;
                }
                else
                {
                        temp->next=start;
                        start->prev=temp;
                        temp->prev =NULL;
                        start=temp;
                }
        }
}
```

III. Linked List

3. Doubly Linked List

```
void doublelinklist::insertend(int x)
{
        node *temp,*p;
        temp=new node;
        if(temp==NULL)
        {
                cout<<"\nSorry, Overflow";
                return;
        }
        else
        {
                temp->data = x;
                temp->next=NULL;
                if(start==NULL)
                {
                        start=temp;
                        temp->prev=NULL;
                        return;
                }
                else
                {
                        p=start;
                        while(p->next)
                        {
                                p=p->next;
                        }
                        p->next=temp;
                        temp->prev =p;
                }
        }
}
```

III. Linked List

3. Doubly Linked List

```
void linklist::display ()
{
        if(start==NULL)
        {
                cout<<"\nSorry ,no nodes to display";
        }
        else
        {
                node *tmp=start;
                while(tmp)
                {
                        cout<<"\n\t"<<tmp->data ;
                        tmp=tmp->link;
                }
        }
}
```

III. Linked List

3. Doubly Linked List

```
doublelinklist::~doublelinklist()
{
        node *temp;
        while(start)
        {
                temp=start;
                start=start->next;
                delete temp;
        }
        cout<<"\n\nDeleted";
}
```

III. Linked List

4. Circular Linked List

- Singly linked list has NULL stored in its last node.
- This NULL pointer is replaced with the address of first node in Circular linked list.
- Any time after reaching the last node we can move to first node.

III. Linked List

4. Circular Linked List

```
class node
{
public :
        int data;
        node *next;
};
class circularlinklist
{
        node *last;
public :
        circularlinklist();
        void insertbeg(int x);
        void insertend(int x);
        void display();
        ~circularlinklist();
};
```

III. Linked List

4. Circular Linked List

```
#include<iostream.h>
#include "linklist.h"
///////////////////////////////////////////////////////////

circularlinklist::circularlinklist()
{
		last=NULL;
}
```

III. Linked List

4. Circular Linked List

```
void circularlinklist::insertbeg(int x)
{
        node *temp;
        temp=new node;
        if(temp==NULL)
        {
                cout<<"\nSorry, Overflow";
                return;
        }
        else
        {
                temp->data =x;
                if(last==NULL)
                {
                        last=temp;
                        last->next=last;
                        return;
                }
                else
                {
                        temp->next=last->next;
                        last->next=temp;

                }
        }
}
```

III. Linked List

4. Circular Linked List

```
void circularlinklist::insertend(int x)
{
        node *temp;
        temp=new node;
        if(temp==NULL)
        {
                cout<<"\nSorry, Overflow";
                return;
        }
        else
        {
                temp->data = x;
                if(last==NULL)
                {
                        last=temp;
                        last->next=last;
                        return;
                }
                else
                {
                        temp->next=last->next;
                        last->next=temp;
                        last=temp;
                }
        }
}
```

III. Linked List

4. Circular Linked List

```
void circularlinklist::display ()
{
        if(last==NULL)
        {
                cout<<"\nSorry ,no nodes to display";
        }
        else
        {
                node *temp=last->next;
                while(temp!=last)
                {
                        cout<<"\n\t"<<temp->data ;
                        temp=temp->next;
                }
                cout<<"\n\t"<<temp->data ;
        }
}
```

III. Linked List

4. Circular Linked List

```
circularlinklist::~circularlinklist()
{
        node *temp;
        while(last->next!=last)
        {
                temp=last->next;
                last->next=temp->next;
                delete temp;
        }
        delete last;
        last=NULL;
        cout<<"\n\nDeleted";
}
```

III. Linked List

What we have covered

1. Definition
2. Singly Linked List
3. Doubly Linked List
4. Circular Linked List

Session IV

Stack

IV. Stack

Agenda

1. Definition
2. Stack Using Linked List

IV. Stack

1. Definition

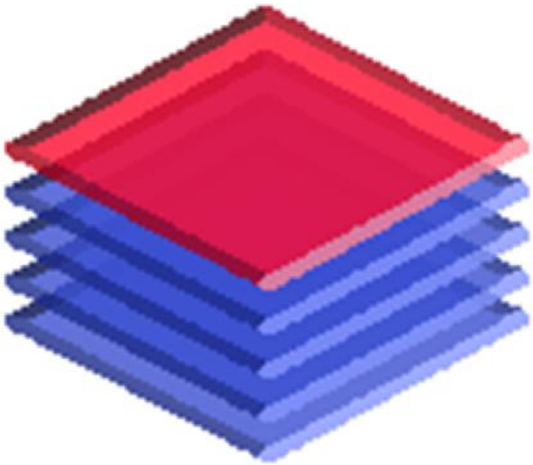
- A stack is a list of elements in which insertions and deletions take place at the same end.
- Also known as "last-in, first-out" or LIFO.
- AbstractDataType stack

  {

  instances

  set of elements; one end is called the *bottom*, the other is the *top*

  }



IV. Stack

2. Stack Using Linked List

```
Stack Using Linked List
class STNODE
{
public :
            int data;
             STNODE * next;
};                                          // a Stack node in the linked list
class Stack
{
            private:
                        STNODE * top;
            public:
                        Stack( int n );
                        // check if stack is empty
                        int IsEmpty( );
                         // try to store ele and return SUCCESS or FAILURE
                        int Push( int ele );
                         // try to remove ele and return SUCCESS or FAILURE
                        int Pop( int * ele );

                        ~Stack( );
};
```

IV. Stack

2. Stack Using Linked List

```
Stack::Stack( )
{
            top = NULL;
}
int Stack::IsEmpty( )
{
            if ( top = = NULL )
                        return 1;
            return 0;
}
Stack::~Stack( )
{
            STNODE * node;
            // cleaning up the dynamic memory when the stack is deleted
            while ( top != NULL )    // traversing the linked list till end
            {
                        node = top;             // store address in the temporary variable
                        top = top->next;        // go ahead to next node in the linked list
                        delete node;
            }
}
```

IV. Stack

2. Stack Using Linked List

```
int Stack::Push( int ele )
{
        STNODE * newNode;
        newNode  = new STNODE;
        if ( newNode = = NULL )
                return -1;                  // memory allocation error
        newNode->data = ele;
        newNode->next = top;    // connect this node to the current top
        top = newNode;                      // make top point to the latest node
        return 1;
}
```

IV. Stack

What we have covered

1. Definition
2. Stack Using Linked List

Session V

Queue

V. Queue

Agenda

1. Definition
2. Queue Implemented Using Array

V. Queue

1. Definition

- A queue is a list of elements in which insertions and deletions take place at different ends.
- Also known as “first-in, first-out" or FIFO.
- AbstractDataType queue

  {

  instances

  set of elements; one end is called the *front*, the other is the *rear*.

  }

V. Queue

2. Queue Implemented Using Array

```
Queue Implemented using Array
class Queue
{
        private:
                int sizeOfQueue;
                int front; // front will refer to one location before the 1st element in the queue
                int rear; // rear will refer to the last location
                int* data;
        public:
                Queue( int n );
                int IsFull( );
                int IsEmpty( );
                // adding element to the end of the queue
                int AddElement(int ele);
                // deleting element from the beginning of the queue
                int DeleteElement(int* ele);
                ~Queue( );
};
```

V. Queue

2. Queue Implemented Using Array

```
Queue::Queue   (int  n)
{
        data = new int[n];          // create a queue of size n
        sizeOfQueue = n;

        front = rear =    -1;        // queue empty condition
}

int Queue::IsFull(   )
{
 if ( rear == (sizeOfQueue      - 1) )  // rear touching th  e last location of array
        return 1;                        // full

        return 0;                        // not full
}

int Queue::IsEmpty(   )
{
        // front as well as rear will advance upon addition and deletion
        // they meet (initial condition or after additions/deletio    ns) indicates Q Empty

        if ( front == rear )
         return 1;                         // empty

         return 0;                        // not empty
}
```

V. Queue

2. Queue Implemented Using Array

```
int Queue::AddElement( int ele )
{
        if ( IsFull(  ) )
                return -1;   // error : can't add any further
        ++rear;
        data[rear] = ele;
        return 0;
}
int Queue::DeleteElement(  int * ele  )
{
        if ( IsEmpty(  ) )
                return -1;   // error : do not have any items for removal
        *ele = queue[front];
++front;

        return 0;
}
Queue::~Queue( )
{
        delete [ ] data;
}
```

V. Queue

2. Queue Implemented Using Array

```
Queue::Queue( )
{
            front = rear = NULL;
}
int Queue::IsEmpty(  )
{
            if ( front == NULL )
                        return 1;                 // empty
            return 0;                             // not empty
}
Queue::~Queue( )
{
            QNODE * node;

            // cleaning up the dynamic memory when the queue is deleted
            while ( front != NULL )               // traversing the linked list till end
            {
                        node = front;                        // store address in the temporary variable
                        front = front->next;      // go ahead
                        delete node;
            }
}
```

V. Queue

2. Queue Implemented Using Array

```
int Queue::AddElement( int ele )
{
        QNODE * newNode;
        newNode = new QNODE;
         if ( newNode == NULL )
                return -1;              // error: memory allocation
        newNode->data = ele;
        newNode->next = NULL;           // new element is added at the end
        if ( IsEmpty( ) )
        {
          front = rear = newNode;   // this is the first element in the queue
                return 1;
        }
        else
        {
        // connect this node to the last node
        rear->next = newNode;
        rear = newNode;                         // the new last node
        }
        return 0;
}
```

V. Queue

2. Queue Implemented Using Array

```
Circular Queue Implemented using Array
class CQueue
{
private:
        int sizeOfQueue;
        int front; // front will refer to one location before the 1st element of the queue
        int rear;  // rear will refer to the last location of the queue
        int * data;

public:
                // functionality of the methods similar to the earlier queue implementation
                Queue(int n);
                int IsFull( );
                int IsEmpty( );
                int AddElement( int ele );
                int DeleteElement( int * ele );
                ~Queue( );
};
```

V. Queue

2. Queue Implemented Using Array

```
Queue::Queue(int n)
{
            data = new int[n];                    // create a queue of size n
            sizeOfQueue = n;
            front = rear = -1;                    // queue empty condition
}
int Queue::IsFull(  )
{
            if ( ((rear == ( sizeOfQueue-1)  )
                        return 1;                           // full
            return 0;                                       // not full
}
int Queue::IsEmpty(  )
{
             if ( front == rear )
                        return 1;                           // empty
            return 0;                                       // not empty
}
```

V. Queue

2. Queue Implemented Using Array

```
int Queue::AddElement(int ele )
{
        if ( IsQFull(  ) )
                return -1;                              // error : can't add any further
        rear = ( rear + 1 ) % sizeOfQueue;   //advance the last location index
        data[rear] = ele;
        return 0;
}
int Queue::DeleteElement( int * ele )
{
        if ( IsEmpty( ) )
                return -1;              // error : do not have any items for removal
        front = (front + 1 ) % sizeOfQueue; //advance the first location index
        * ele = data[front];
        return 0;
}
```

V. Queue

2. Queue Implemented Using Array

```
int Queue::AddElement( int ele )
{
            QNODE * newNode;
            newNode = new QNODE;
             if ( newNode == NULL )
                        return -1;                          // error: memory allocation
            newNode->data = ele;
            newNode->next = NULL;               // new element is added at the end
            if ( IsEmpty( ) )
            {
                front = rear = newNode;          //this is the first element in the queue
                        return 1;
            }
            else
            {
            // connect this node to the last node
            rear->next = newNode;
            rear = newNode;                                 // the new last node
            }
            return 0;
}
```

V. Queue

What we have covered

1. Definition
2. Queue Implemented Using Array

Session VI

Tree

VI. Tree

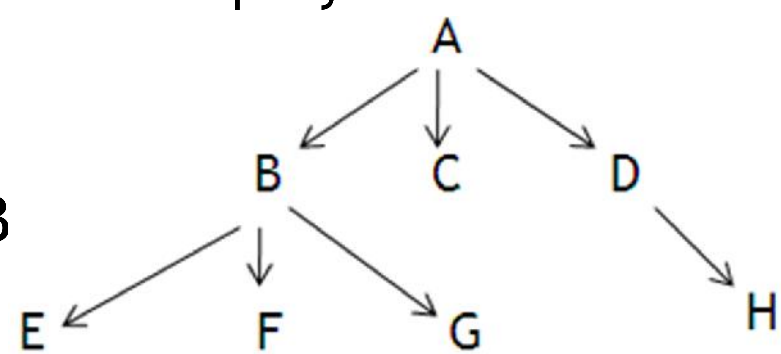
Agenda

1. Definition
2. Binary Tree
3. Tree Traversal

VI. Tree

1. Definition

- Trees are used to store hierarchical data.
- For example - Employee data of a company.
- The above may represent
- Employees
  - E, F, G work under B
  - H work under D
  - B, C, D work under A
- Nobody reports to Employee C and A is the top boss.



VI. Tree

1. Definition

- A tree t is a finite nonempty set of elements.
- One of these elements is called the root, and the remaining elements are partitioned into trees, which are called subtrees of t.
- Terminology :
  - Node - Each element in tree is called a node.
  - Edge - The line connecting an element node and its children.
  - root - A node that does not have parent.
  - Leaf - A node that does not have any child.
- Degree of a node
  - It is the number of children it has.
- Degree of tree
  - Degree of tree is the maximum degree of any of the nodes of the tree.
- Height of a node
  - Height of root is taken as 1. The height of a child node is 1 + the
  - Height of its parent node.
- Height of tree
  - Height of tree is the maximum height of any of the nodes of the tree.

VI. Tree

2. Binary Tree

- A binary tree t is a finite collection of elements.
- When the binary tree is not empty, it has a root element and the remaining elements are partitioned into two binary trees, which are called the left and right subtrees of it.

- Properties of a binary tree :
  - A binary tree with n elements, $n > 0$, has exactly $n-1$ edges.
  - A binary tree of height h, $h > 0$, has at least h and at most $2^h - 1$ elements.
- A binary tree of height h that contains exactly $2^h - 1$ elements is called a full binary tree.

VI. Tree

2. Binary Tree

- Tree Traversal
  - Preorder
    - DLR-First process the current node; traverse left subtree and then the right subtree
    - Output is – A B C
  - Postorder
    - LRD – Fitsrt traverse the left subtree, the right subtree and then process the current node
    - Output is – B C A
  - Inorder
    - LDR – First traverse the left subtree, process the current node and then traverse the right subtree
    - Output is – B A C

VI. Tree

2. Binary Tree

- Tree Traversal

Binary Tree Implementation

```
class node
{
public :
            node *left;
            int data;
            node *right;
};
class tree
{
            node *root;
            //these are called by public functions
            void preorder(node *);
            void inorder(node *);
            void postorder(node *);
            void nonrecinorder(node *);
public :
            tree();
            //function to build a tree
            void insert(int);
            void Preorder();
            void Postorder();
            void Inorder();
            void remove(int);
            ~tree();
};
```

VI. Tree

2. Binary Tree

- Tree Traversal

```
#include<stdio.h>
#include<iostream.h>
#include "tree.h"

tree::tree()
{
        root=NULL;
}


void tree::preorder(node *p)
{
        if(p!=NULL)
        {
                cout<<"\n"<<p->data; //print the current node
                preorder(p->left);  //traverse the left subtree
                preorder(p->right); //traverse the right subtree
        }
}
```

VI. Tree

2. Binary Tree

- Tree Traversal

```
void tree::preorder(node *p)
{
        if(p!=NULL)
        {
                cout<<"\n"<<p->data; //print the current node
                preorder(p->left);  //traverse the left subtree
                preorder(p->right); //traverse the right subtree
        }
}
```

VI. Tree

2. Binary Tree

- Tree Traversal

```
void tree::inorder(node *p)
{
        if(p!=NULL)
        {
                inorder(p->left); //traverse the left subtree
                cout<<"\n"<<p->data;//print the current node (root)
                inorder(p->right);//traverse the right subtree
        }
}
```

VI. Tree

2. Binary Tree

- Tree Traversal

```
void tree::postorder(node *p)
{
        if(p!=NULL)
        {
                postorder(p->left);//traverse the left subtree
                postorder(p->right);//traverse the right subtree
                cout<<"\n"<<p->data;//print the current node (root)
        }
}
```

VI. Tree

2. Binary Tree

- Tree Traversal

```
void tree::Preorder()
{
        preorder(root);
}

void tree::Postorder()
{
        postorder(root);
}
```

VI. Tree

2. Binary Tree

- Tree Traversal

```
void tree::Inorder()
{
        cout<<"\nRecursive Inorder";
        inorder(root);
        cout<<"\nRecursive Inorder";
        nonrecinorder(root);
}

tree::~tree()
{
        delete root;
}
```

VI. Tree

2. Binary Tree

- Tree Traversal

Non recursive traversal in inorder

```
void tree::nonrecinorder(node *p)
{
        int top=-1;
        node *stack[10];//stack to store return addresses
        p=root;
        if(p==NULL)
        {
                cout<<"\nNo nodes to print";
                return;
        }
        do
        {
//goto left till you find leaf node and store
//the return addresses

                while(p!=NULL)
                {
                        stack[++top]=p;
                        p=p->left;
                }
        //reached the end of the leftmost node
                //pop to get its parent
                p=stack[top--];
                cout<<"\n"<<p->data;
        //traverse the right subtree
                p=p->right;
        }while(p!=NULL || top==-1);
}
```

VI. Tree

What we have covered

1. Definition
2. Binary Tree
3. Tree Traversal

Session VII

Graph Terminology

VII. Graph Terminology

Agenda

1. Definition
2. Path
3. Types of Graph
4. Graph Representation
5. Searching Methods

VII. Graph Terminology

1. Definition

- A graph is a collection of vertices.
- Two vertices can be joined by edges.
- An edge is denoted by the tuple (i,j) where i and j are vertices.
- An edge, with an arrow representing the direction, is called a directed edge
- Two vertices i and j are adjacent vertices if and only if (i,j) is an edge in the graph.

VII. Graph Terminology

2. Path

- A path in a graph is a sequence of vertices $i_1, i_2 .. i_k$ such that (i_j, i_{j+1}) is an edge for every j, $1 <= j < k$
- A simple path is a path in which all vertices (except first and last) are different.
- A cycle is a simple path with the same start and end vertex.

VII. Graph Terminology

3. Types of Graph

- A graph H is a subgraph of G if and only if its vertex and edge sets are subsets of those of G.
- A connected undirected graph with n vertices must have at least n-1 edges.
- A connected undirected graph that does not contain cycle is called a tree.
- A subgraph of G that contains all vertices of G and is a tree is called a spanning tree.

VII. Graph Terminology

3. Types of Graph

- A degree of a vertex in an undirected graph is the number of edges incident on it.
- An undirected graph with n vertices and n(n-1)/2 edges is called a complete graph.
- A diagraph with n vertices and n(n-1) directed edges is called a complete diagraph.
- In a diagraph, indegree is no of edges incident to a vertex and outdegree is no of edges incident from a vertex.

VII. Graph Terminology

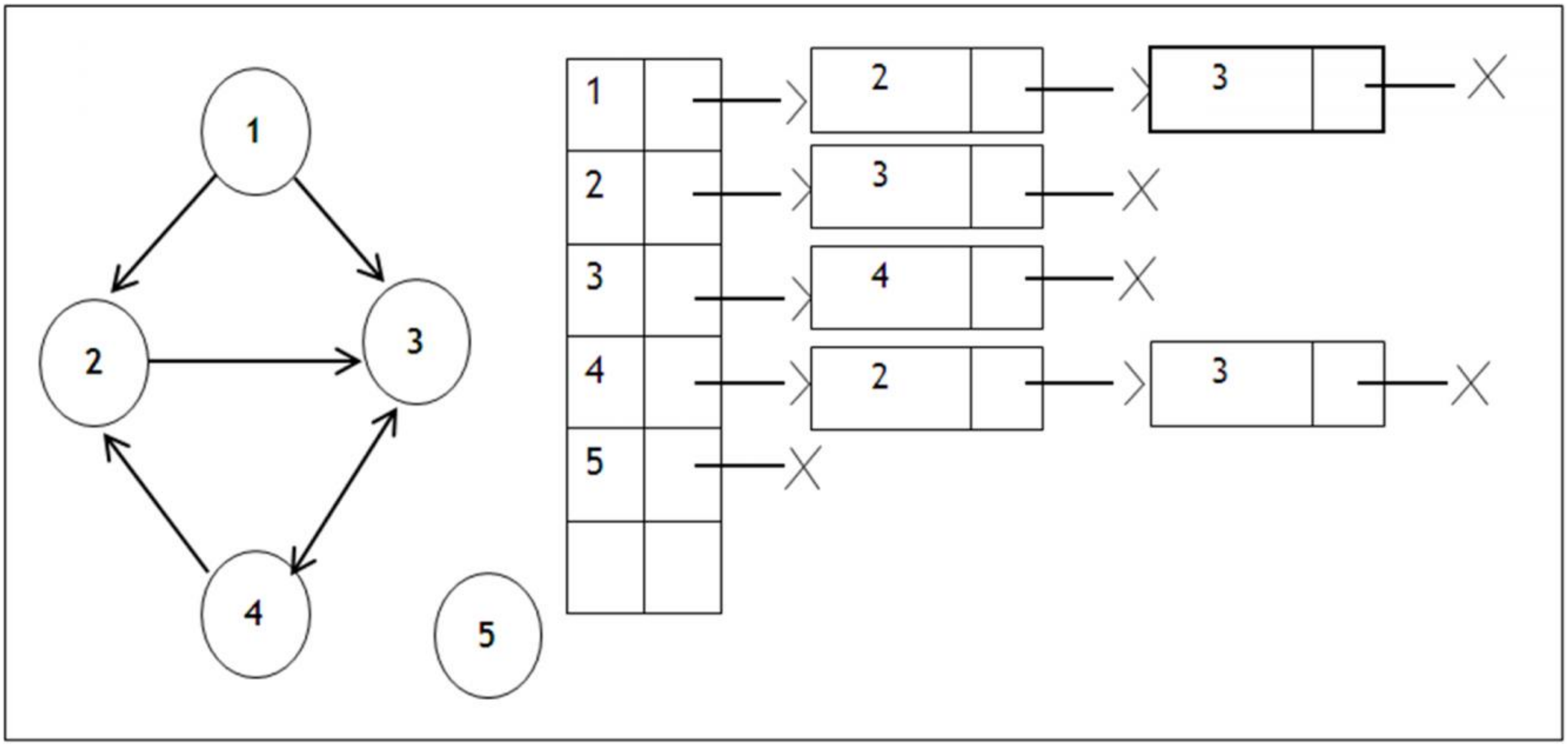
4. Graph Representation

- Graphs can be represented by ,
  - Adjacency matrix
  - Linked adjacency list

VII. Graph Terminology

4. Graph Representation

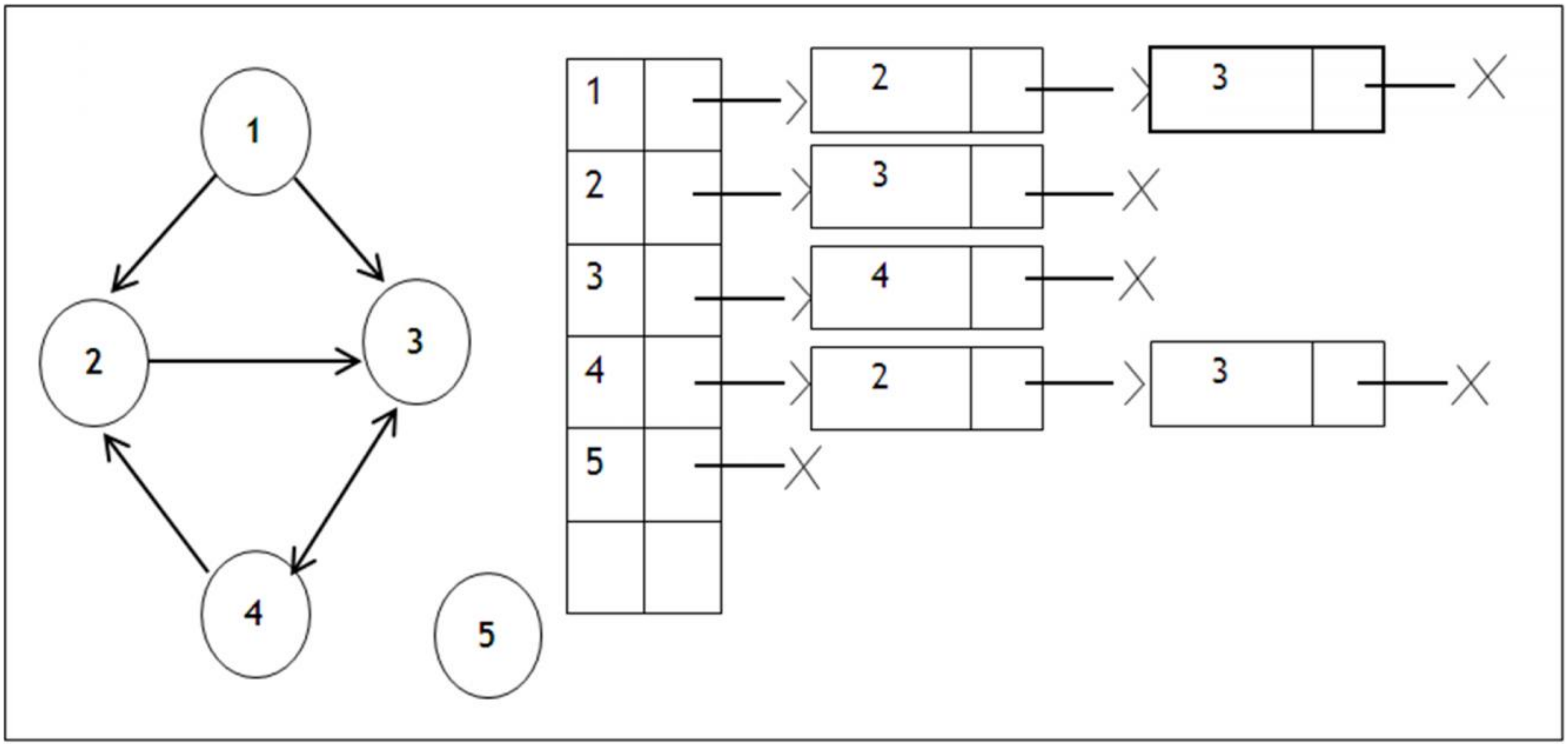
- Linked Adjacency List Representation



VII. Graph Terminology

4. Graph Representation

- Linked Adjacency List Representation



VII. Graph Terminology

5. Searching Methods

- Graph Search Methods :
  - Methods to find out all the vertices which are reachable from a given start vertex.
  - A vertex u is reachable from vertex v if and only if there is a path from v to u.
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)

VII. Graph Terminology

What we have covered

1. Definition
2. Path
3. Types of Graph
4. Graph Representation
5. Searching Methods

Session VIII

Templates

VIII. Templates

Agenda

1. Overview
2. Example
3. Function Templates
4. Class Templates
5. STL

VIII. Templates

1. Overview

- Overview of Templates
- A template is used to define generic classes and functions.
- Generic types are passed as parameters.
- Also called as parameterized classes or functions.

VIII. Templates

2. Example

- Overview of Templates
- A template is used to define generic classes and functions.
- Generic types are passed as parameters.
- Also called as parameterized classes or functions.

VIII. Templates

3. Function Templates

- A more flexible mechanism that provides function definition in terms of some data type as parameter.
- Function template is not an ordinary function.
- Instantiate new functions that apply the same task to arguments of different types.
- A template function depends on underlying data type(s) -- the template parameter(s).
- Each argument to a template function must be an exact match to the data type of the formal parameter.
- In template instantiation, no type conversions take place.

VIII. Templates

3. Function Templates

A function template is used to create a family of functions with different argument types.

The general format of function template is -

```
template <class T>
returntype functionname (arguments of type T)
{
    // define function
    // with anonymous type T used in code,
    // wherever needed.
}
```

More than one data types can be passed as parameters.

VIII. Templates

3. Function Templates

```
template <class GenericType >
GenericType FindBigger( GenericType a, GenericType b )
{
    if ( a > b )
        return a;
    else
        return b;
}
int iMax = FindBigger( 10, 20 );
double dMax = FindBigger( 1.34, 1.346 );
```

```
Room r1, r2;
// ... r1 and r2 filled with some data
// operator '>' has to be overloaded in class Room, for the following to
    work
Room bigRoom = FindBigger( r1, r2 );
```

VIII. Templates

4. Class Templates

- The general format of class template is -

```
template <class T>
class classname
{
    //define class memebers
    //with anonymous type T
    //wherever needed.
};
```

VIII. Templates

4. Class Templates

```
template <class T>
class vector
{
        private:    T * v;      // vector of type T
                                int size;
        public:     Vector( int n )
                                {
                                        v = new T[ m ];
                                        size = m;
                                }
                                T operator * ( vector  & p )
                                {
                                        T sum = 0;
                                        for ( int i = 0; i < size; ++i )
                                                sum += this->v[ i ] * p.v[ i ];
                                  return sum;
// note that sum may be of any type T
                                }
                                // other member functions
};
vector <int> v1( 10 );          // a vector of integers with size 10
vector <float> v2( 25 );        // a vector of floats with size 25
```

VIII. Templates

5. STL

- A collection of generic classes and functions.
- STL classes are used for storing and processing data.
- The STL contains,
  - Containers
  - Algorithms
  - Iterators
- Container :
  - Actually stores data.
  - It can be used to hold different types of data.

VIII. Templates

5. STL

Use of Container classes in the library

```
// multiset is a collection that behaves like a 'bag' data structure, where elements can be
//
//   added, removed from it and no order of elements as such, maintained in the
container.
// declare a container - students that are going to contain items of type string
multiset<string> students;
students.insert("sona");
students.insert("rick");
students.insert("john");
students.insert("riya");
// indexStu is like a pointer, to iterate through the collection
multiset<string>::iterator  indexStu;
for ( indexStu = students.begin(); indexStu != students.end(); ++indexStu)
{
// print the data
cout << * indexStu << "\n";  // standard operators (like '*') are overloaded on iterators
}
```

Similarly, library provides implementations for Stack, Queue, String, and Vector etc, with their template classes (with appropriate functions) and iterators.

VIII. Templates

What we have covered

1. Overview
2. Example
3. Function Templates
4. Class Templates
5. STL

In this Module – What we have covered

I. Overview of Data Structures
II. Array
III. Linked List
IV. Stack
V. Queue
VI. Tree
VII. Graph Terminology
VIII. Templates