DATA STRUCTURES

The importance of data structures defines

- how to store a collection of objects in memory,
- what operations we can perform on that data,
- the algorithms for those operations, and
- how time and space efficient those algorithms are.
- Much of programming involves deciding how to arrange information in memory.
- Choice of data structures can make a big speed difference.
- Abstract Data Types are a way to encapsulate and hide the implementation of a data structure, while presenting a clean interface to other programmers.

Certain examples and applications of data structures are:

- How does Google quickly find web pages that contain a search term?
- What's the fastest way to broadcast a message to a network of computers?
- How can a subsequence of DNA be quickly found within the genome?
- How does your operating system track which memory (disk or RAM) is free?
- In the game Half-Life, how can the computer determine which parts of the scene are visible?

Operations to support the following scenarios...

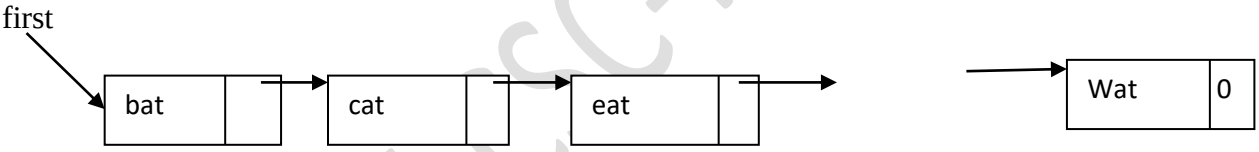
- Suppose You searching for a address in Google Maps...
- You want to store data about cities (location, elevation, population)...
- Finding addresses on map? - Lookup city by name...
- Mobile iPhone user? - Find nearest point to me...
- Car GPS system? - Calculate shortest-path between cities...
- Show cities within a given window...
- Political revolution? - Insert, delete, rename cities

ABSTRACT DATA TYPES

An important element to good data structure design is to distinguish between the functional definition of a data structure and its implementation. By an abstract data structure (ADT) we mean a set of objects and a set of operations defined on these objects.

LINKED LISTS

Linked lists are useful to study for two reasons. Most obviously, linked lists are a data structure which you may want to use in real programs. Seeing the strengths and weaknesses of linked lists will give you an appreciation of the some of the time, space, and code issues which are useful to thinking about any data structures in general. Linked lists and arrays are similar since they both store collections of data. The terminology is that arrays and linked lists store “elements”. A linked list allocates space for each element separately in its own block of memory called a "linked list element" or "node". The list gets is overall structure by using pointers to connect all its nodes together like the links in a chain. Each node contains two fields: a "data" field to store whatever element type the list holds for its client, and a "next" field which is a pointer used to link one node to the next node. Each node is allocated in the heap with a call to malloc(), so the node memory continues to exist until it is explicitly deallocated with a call to free(). The front of the list is a pointer to the next node.



Defining a node list

```
class nodea {
private:
 int data1;
int data2;
float data3;
nodea *linka;
nodea *linkb;
};
Class nodeb{
Private:
int data;
nodeb *link;
};
```

The above code defines nodea to consist of three data fields and two link fields, whereas nodes of type nodeb will consist of one data field and one link data field. Data member of nodes of type nodea must point to nodes of type nodeb.

Linked list operations

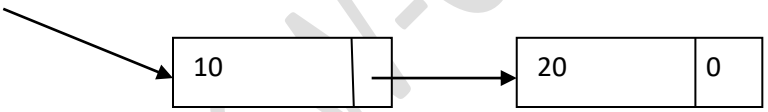
Pointer manipulation are used to implement list operations in data structure. Major operations performed by linked list are 1. Creation 2.Insertion 3.Deletion

Creation:

```
void List:: create2(){
First=new ListNode (10);// create and initialize first node
// create and initialize second node and place its address in first->link
First->link=new ListNode(20);
}
ListNode::ListNode(int element=0)// 0 is the default argument in constructor for ListNode
{
data=element; link=0;// null pointer constant
}
```

The above code creates a two node list

first



Insertion:

```
void List::Insert50(ListNode *x){
ListNode *t=new LirstistNode(50);//create and initialize new node/
If(!first)// insert into empty list
{
 first =t ;
return;//exit List::Insert 50
}// insert after x
t->link=x->link;
x->link=t;
```

```
}
```

The above code is used to insert a node in a linked list.

Deletion:

```
void List::Delete(ListNode *x, ListNode *y)
{
If(!y)first=first->link;
Else y->link=x->link;
Delete x;//return the node
}
```

The above code is used to delete a node from a linked list.

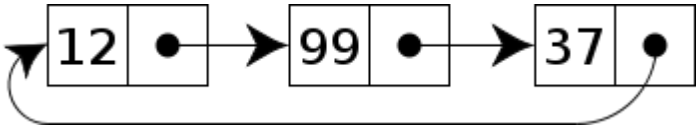
CIRCULAR LIST

In a circularly linked list, all nodes are linked in a continuous circle, without using null. For lists with a front and a back (such as a queue), one stores a reference to the last node in the list. The next node after the last node is the first node. Elements can be added to the back of the list and removed from the front in constant time.

Circularly-linked lists can be either singly or doubly linked.

Both types of circularly-linked lists benefit from the ability to traverse the full list beginning at any given node. This often allows us to avoid storing firstNode and lastNode, although if the list may be empty we need a special representation for the empty list, such as a lastNode variable which points to some node in the list or is null if it's empty; we use such a lastNode here. This representation significantly simplifies adding and removing nodes with a non-empty list, but empty lists are then a special case.

A circular list is obtained by modifying singly linked list so that the link field of the last node points to the first node in the list. Let us look at operations on circular linked list.



Circular linked list

In circular linked list nodes can be inserted at the front or at the rear. Code for inserting the node at the front of the circular list is given below.

```
 void  circlist:: Insertfront(ListNode<type>*x)
//  insert the node pointed at by x at the front of the circular list this where last points to the last
//node in the list
{
If(!last){// empty list
Last=x; x->link=x;
}
else{
x->link=last->link; last->link=x;
}
}
```

Circularly-linked vs. linearly-linked

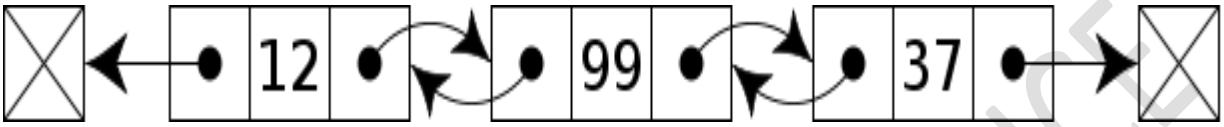
A circularly linked list may be a natural option to represent arrays that are naturally circular, e.g. the corners of a polygon, a pool of buffers that are used and released in FIFO order, or a set of processes that should be time-shared in round-robin order. In these applications, a pointer to any node serves as a handle to the whole list.

With a circular list, a pointer to the last node gives easy access also to the first node, by following one link. Thus, in applications that require access to both ends of the list (e.g., in the implementation of a queue), a circular structure allows one to handle the structure by a single pointer, instead of two.

A circular list can be split into two circular lists, in constant time, by giving the addresses of the last node of each piece. The operation consists in swapping the contents of the link fields of those two nodes. Applying the same operation to any two nodes in two distinct lists joins the two list into one. This property greatly simplifies some algorithms and data structures, such as the quad-edge and face-edge.

DOUBLY LINKED LIST

A node in doubly linked list has atleast three fields 1. data 2.llink 3.rlink. A doubly linked list may or may not be circular. A sample doubly linked list is given below.



Besides these three nodes a special node called head node has been added. The data field of the head node usually contains no information. Suppose p is a node that points to any node in a doubly linked list it is represented by p==p->llink->rlink==p->rlink-llink. Class definition of a doubly linked list is given above.

```
class dbllist;
class dbllistnode{
friend class dbllist;
private:
int data;
dbllistnode *llink, *rlink;
};
class  dbllist{
public:
//list manipulation operations
……
private:
dbllistnode *first;// points to head node
};
```

To delete a node from doubly linked list following code has been implemented.

```
void dbllist::Delete(dbllistnode *x)
{
if(x==first) cerr<<"Deletion of head node not permitted"<<endl;
else {
x->llink->rlink=x->rlink;
x->rlink->llnik=x->llink;
delete x;
}
}
```

To insert a node in an doubly linked list following code has been implemented.

```
void dbllist::insert(dbllistnode *p, dbllistnode *x)// insert node p to the right of node x
{
p->llink=x; p->rlink=x->rlink;
x->rlink->llink=p;x->rlink=p;
}
```

Doubly-linked vs. singly-linked

Double-linked lists require more space per node and their elementary operations are more expensive; but they are often easier to manipulate because they allow sequential access to the list in both directions. In particular, one can insert or delete a node in a constant number of operations given only that node's address. To do the same in a singly-linked list, one must have the previous node's address. Some algorithms require access in both directions. On the other hand, doubly-linked lists do not allow tail-sharing and cannot be used as persistent data structures.

TREES

Tree is a widely-used data structure that emulates a hierarchical tree structure with a set of linked nodes. An acyclic connected graph where each node has zero or more children nodes and at most one parent node. Furthermore, the children of each node have a specific order.

A node is a structure which may contain a value, a condition, or represent a separate data structure (which could be a tree of its own). Each node in a tree has zero or more child nodes, which are below it in the tree (by convention, trees are drawn growing downwards). A node that has a child is called the child's parent node (or ancestor node, or superior). A node has at most one parent.

Nodes that do not have any children are called leaf nodes. They are also referred to as terminal nodes. A free tree is a tree that is not rooted.

The height of a node is the length of the longest downward path to a leaf from that node. The height of the root is the height of the tree. The depth of a node is the length of the path to its root (i.e., its root path).

The topmost node in a tree is called the root node. Being the topmost node, the root node will not have parents. It is the node at which operations on the tree commonly begin. All other nodes can

be reached from it by following edges or links. (In the formal definition, each such path is also unique). In diagrams, it is typically drawn at the top. In some trees, such as heaps, the root node has special properties. Every node in a tree can be seen as the root node of the subtree rooted at that node.

An internal node or inner node is any node of a tree that has child nodes and is thus not a leaf node. Similarly, an external node or outer node is any node that does not have child nodes and is thus a leaf.

A subtree of a tree T is a tree consisting of a node in T and all of its descendants in T. The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree.

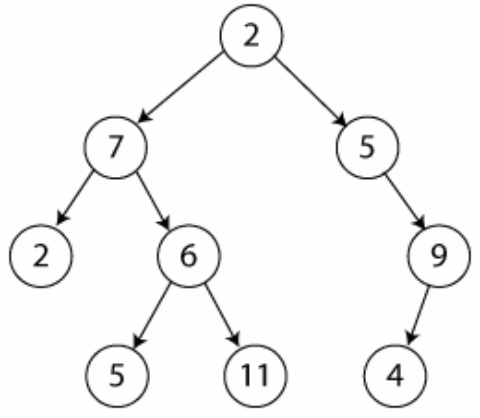
Tree Representations

There are many different ways to represent trees; common representations represent the nodes as records allocated on the heap with pointers to their children, their parents, or both, or as items in an array, with relationships between them determined by their positions in the array (e.g., binary heap).

Binary Tree

A binary tree is a finite set of nodes that either is empty or consists of a root and two disjoint binary trees called the left subtree and the right sub tree.

```
template <class keytype>
class Binarytree{
public:
BinaryTree();
Boolean IsEmpty();
BinaryTree(BinaryTree bt 1, Element<keytype>item, BinaryTree bt2);
BinaryTree Lchild();
Element<Keytype> data();
BinaryTree Rchild();
};
```



Complete Binary Tree

Binary tree with n nodes and depth k is complete if its nodes correspond to the nodes numbered from 1 to n in the full binary tree of depth k.

A binary search tree is itself a special kind of binary tree. A binary tree is a tree which is either empty or consists of a node called the root, together with two children called the left subtree and the right subtree of the root. Each of these children is itself a binary tree.

Binary Tree Traversals

There are two fundamentally different kinds of binary trees traversals--those that are depth-first and those that are breadth-first. When you watch the animation, notice that the path followed by each of these traversals is along the branches of the tree. Each node of the tree is visited three times during each of the depth-first traversals, once on its way down the tree, a second time coming up from the left child, and a third time coming up from the right child. When you watch the animation of these traversals, notice that a checkmark is placed beneath each node each time the node is visited.

There are three different types of depth-first traversals, preorder, inorder, and postorder. The preorder traversal extracts the value the first time it visits the node--when there is one checkmark beneath the node. During the animation, the value of the node is copied to the list at the bottom of the screen when the value is extracted. During the inorder traversal, the value is extracted during the second visit to the node--when there are two checkmarks beneath the node. During the

postorder traversal, the value is extracted during the third visit--when there are three checkmarks beneath the node.

Inorder Traversal

In order traversal calls for moving down the tree towards the left until we can go farther. Then visit the node move one node to the right and continue. If we can't move to the right, go back one more node. Inorder traversal of a binary tree representation is given below. Elements get output in the order of **a/b*c*d+e**

```
void Tree::inorder()
{
Inorder(root);
}
void  Tree::inorder(TreeNode *CurrentNode)
{
if(CurrentNode){
inorder (CurrentNode→LeftChild);
cout<<CurrentNode->data;
inorder(CurrentNode->RightChild);
}
}
```

Preorder Traversal

The second form of traversal is preorder, in words we would say visit a node, traverse left and continue. When you can't continue move right and begin again or move back until you can move right and resume. Preorder traversal of a binary tree representation is given below. Elements get output in the order of **+**/abcde**

```
void Tree::preorder()
{
preorder(root);
}
```

```
void  Tree::preorder(TreeNode *CurrentNode)
{
if(CurrentNode){
cout<<CurrentNode->data;
preorder (CurrentNode→LeftChild);
preorder(CurrentNode->RightChild);
}
}
```

Post order traversal

Reverse operation of pre order traversal is post order . Code for implementing post order is given below. Output of the post order traversal will come in the order of **ab/c*d*e+**

```
void Tree::postorder()
{
postorder(root);
}
void  Tree::postorder(TreeNode *CurrentNode)
{
if(CurrentNode){
postorder (CurrentNode→LeftChild);
postorder(CurrentNode->RightChild);
cout<<Current Node->data;
}
}
```

BINARY SEARCH TREE

A binary search tree is a binary tree. It may be empty. If it is not empty then satisfies the following properties:

1) Every element has a key and no two elements have the same key.

2) The keys in the left subtree are smaller than the key in the root.

3) The keys in the right subtree are larger than the key in the root.

4) The left and right subtrees are also binary search trees.

Thers is some redundancy in the abouve properties definition. Properties (2) (3) & (4) together imply that the keys must be distinct. So, property (1) can be replaced by the property: the root has a key.

Searching a Binary search Tree

Suppose we wish to search for an element with key x. we begin at the root. If the root is 0, then the search tree contains no elements. Search is unsuccessful in this case. Otherwise compare x with the key in the root. If x equals this key then the search terminates successfully. If x is less than the key in the root , then no elements in the right subtree can have key value x, and the left tree have to be searched. If x is larger than the key in the root, only the right subtrees need to be searched.

```
template <class Type>
BstNode<Type>*BST<Type>::Search(const Element<Type>&x)
{
return  Search(root,x);
}
template <classType>
BstNode<Type>*BST<Type>::Search(BstNode<Type>*b,const Element<Type>&x)
{
if(!b) return 0;
if(x.key==b->data.key) return b;
if(x.key<b->data.key) return Search(b->LeftChild, x);
return search(b->RightChild,x);
}
```

Insertion

To insert a new element x, first verify that its key is different from those of existing element. For this search has to be carried out. If search is unsuccessful, then the element is inserted at the point the search terminated.

```
template<classType>
Boolean BST<Type>::insert(const element <Type>&x)
{
BstNode<Type>*p=root; BstNode<Type>*q=0;
while(p){
q=p;if(x.key==p->data.key)return FALSE;
if(x.key<p->data.key)p=p->LeftChild;
else p=p->RightChild;
}
p=new BstNode<Type>;
p->LeftChild=p->RightChild=0;p->data=x;
if(!root)root=p;
else if(x.key<q->data.key)q->LeftChild=p;
else q->RightChild=p;
return TRUE;
}
```

Deletion

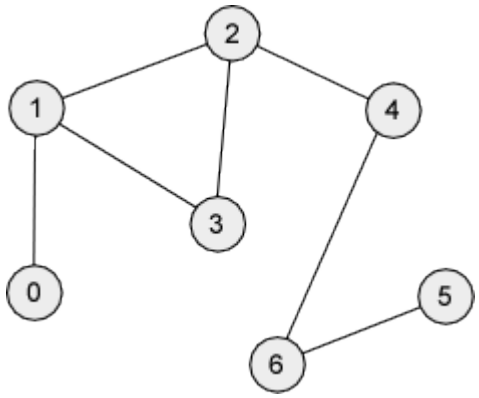
Deletion of a leaf element is quite easy. The deletion of a non leaf element that has only one child is also easy. The node containing the element to be deleted is disposed and the single child takes the place of the disposed node. When the element to be deleted is in a non leaf node that has two children, the element is replaced by either largest element in its left subtree or the smallest one in its right subtree. Then we proceed to delete this replacing element from the subtree from which it was taken.

GRAPHS

All tree structures are hierarchical. This means that each node can only have one parent node. Trees can be used to store data which has a definite hierarchy; for example a family tree or a computer file system.

Some data need to have connections between items which do not fit into a hierarchy

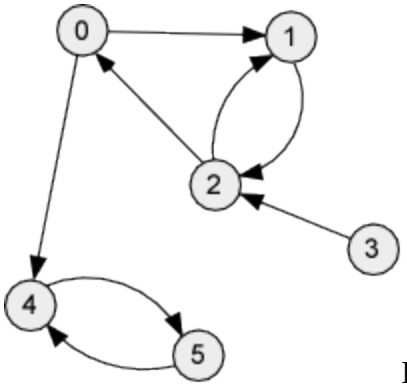
like this. Graph data structures can be useful in these situations. A graph consists of a number of data items, each of which is called a vertex. Any vertex may be connected to any other, and these connections are called edges.



The graph in the figure above is known as an undirected graph.

An undirected graph is complete if it has as many edges as possible – in other words, if every vertex is joined to every other vertex. The graph in the figure is not complete. For a complete graph with n vertices, the number of edges is n(n – 1)/2. Two vertices in a graph are adjacent if the form an edge. A path is a sequence of vertices in which each successive pair is an edge. A cycle is a path in which the first and last vertices are the same and there are no repeated edges. An undirected graph is connected if, for any pair of vertices, there is a path between them. A tree data structure can be described as a connected, acyclic graph with one element designated as the root element. It is acyclic because there are no paths in a tree which start and finish at the same element.

In a directed graph, or digraph, each edge is an ordered pair of vertices – it has a direction defined. A path in a directed graph must follow the direction of the arrows. A directed graph is connected if, for any pair of vertices, there is a path between them.



Directed graph

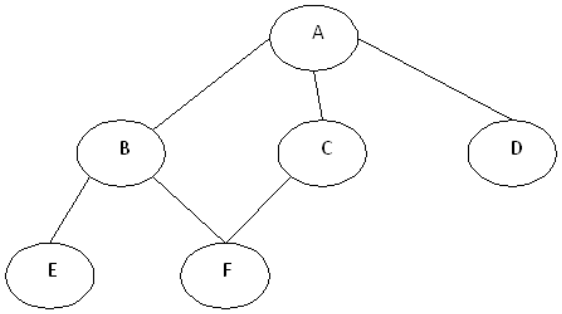
Traversing a graph

Traversal is the facility to move through a structure visiting each of the vertices once. We looked previously at the ways in which a binary tree can be traversed. Two possible traversal methods for a graph are breadth-first and depth-first.

Breadth-First Traversal

This method visits all the vertices, beginning with a specified start vertex. It can be described roughly as “neighbours-first”. No vertex is visited more than once, and vertices are only visited if they can be reached – that is, if there is a path from the start vertex. Breadth-first traversal makes use of a queue data structure. The queue holds a list of vertices which have not been visited yet but which should be visited soon. Since a queue is a first-in first-out structure, vertices are visited in the order in which they are added to the queue.

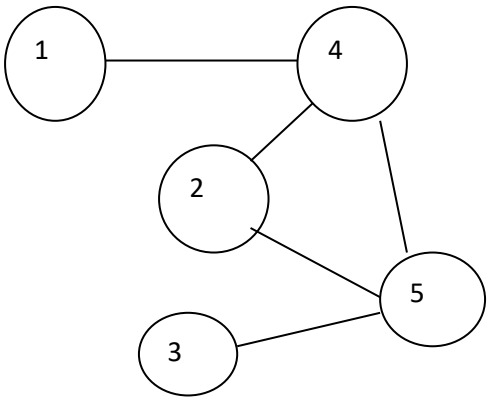
Visiting a vertex involves, for example, outputting the data stored in that vertex, and also adding its neighbours to the queue. Neighbours are not added to the queue if they are already in the queue, or have already been visited.




```
void graph::BFS(int v)
{
visited=new Boolean[n];
for (int i=0;i<n;i++) visited[i]=FALSE;
visited[v]=TRUE;
queue<int>q;
q.Insert(v);
while(!q.IsEmpty()){
v=*q.delete(v);
for(all vertices w adjacent to v)
if(!visited[w]){
q.insert(w);
visited[w]=TRUE;
}
}delete[]visited;
}
```

Depth first Traversals

Depth-first search, or DFS, is a way to traverse the graph. Initially it allows visiting vertices of the graph only, but there are hundreds of algorithms for graphs, which are based on DFS. Therefore, understanding the principles of depth-first search is quite important to move ahead into the graph theory. The principle of the algorithm is quite simple: to go forward (in depth) while there is such possibility, otherwise to backtrack.



Depth first traversal is given by 1->4->2->5->3 . DFS doesn't go through all edges. The vertices and edges, which depth-first search has visited is a tree. This tree contains all vertices of the graph (if it is connected) and is called graph spanning tree. This tree exactly corresponds to the recursive calls of DFS.

```
void graph::DFS()
(
visited=new Boolean[n];
for(int i=0;i<n;i++)
DFS[0];
delete[] visited;
}
void graph::DFS(const int v)
{
visited[v]=TRUE;
for(each vertex w adjacent to v)
if(!visited[w]0DFS(w);
}
```