Introduction to Data Structure

bitCode technologies

learn.innovate.share

Introduction

- Data is a basic unit that any computing system centers around.

- What is data structure?

  Data Structure is an arrangement of data in computer's memory (or sometimes on a disk) so that we can retrieve it & manipulate it correctly and efficiently.

Definitions

- **Data type**

  - Information itself has no meaning because it is just sequence of bytes.

  - It is interpretation of bit pattern that gives it's meaning.

  - For example 00100110 can be interpreted as number 38(binary), number 26(binary coded decimal) or the character '&'.

  - A method of interpreting bit pattern is called as a data type.

  - So data type is a kind of data that variable may hold in a programming language. For example in 'C++' int, float, char & double

Definitions

- **Data object** is a term that refers to set of elements. Such set may be finite or infinite.

- **Data structure** is a set of domains D, a set of functions F and a set of axioms A.

- A triple (D, F,A) denotes the data structure d.

Abstract Data Type

- ADT is a conceptual representation.
- ADT is a mathematical model together with various operations defined in that model.
- ADT is a way of looking at Data Structure focusing on what it does and not how it does.
- Data Structure is implementation of ADT.

Examples of Data Structure

- Arrays
- Stacks
- Queues
- Linked Lists
- Trees
- Graphs

Arrays

bitCode technologies

learn.innovate.share

Introduction to Arrays

- Arrays are defined as a finite ordered set of homogeneous elements.
- Finite means there is a specific number of elements in the array.
- Ordered means the elements of the array are arranged so that so that there is zeroth, first, second and so on elements.
- Homogeneous means all the elements in the array must be of the same type.
- E.g. An array may contain all integers or all characters but not integers and characters.
- Declaration in C++ int a[100];

Operations on Array

- The two basic operations that access an array are Extraction and Storing.
- The Extraction operation is a function that accepts an array a, and an index I, and returns an element of the array.

  In C++ this operation is denoted by the expression a[i];
- The storing operation accepts an array a, an index I and an element x.

  In C++ this operation is denoted by the expression

  a[I] = x;

Arrays Continued…

- The smallest element of an array's index is called it's lower bound. It is 0 in C++.
- The highest element is called it's upper bound.
- If Lower bound is "lower" and Upper bound is "upper" then number of elements in the array, called it's "range" is given by

  upper – lower + 1.

E.g. In array a, the lower bound is 0 and the upper bound is 99, so the range is 100.

Arrays Continued..

- An important feature of array in C++ is that the upper bound nor the lower bound may be changed during the program execution.
- Arrays will always be stored in contiguous memory locations.
- You can have static as well as dynamic arrays, where the range of the array will be decided at run time.

Single Dimension Array Implementation

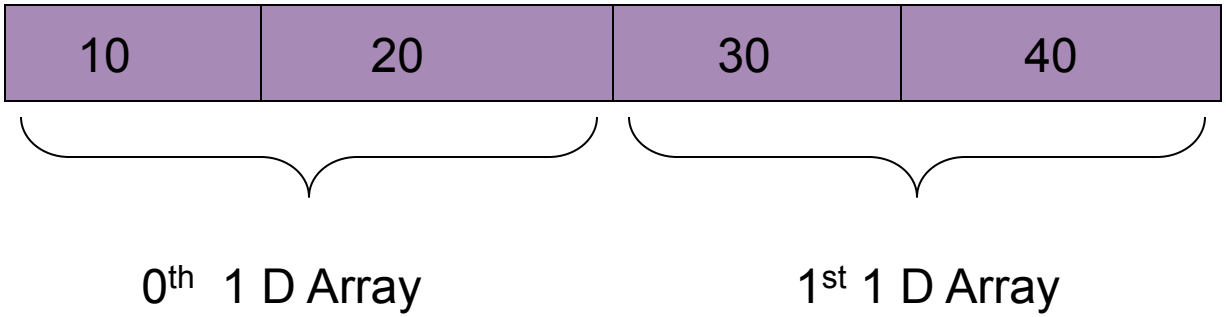
```
Int arr[10], i;

for( i = 0; i < 10; i++ )

   arr[i] = i;


for ( i = 0; i < 10; i++ )

  cout<< i ;
```

Two Dimensional Array

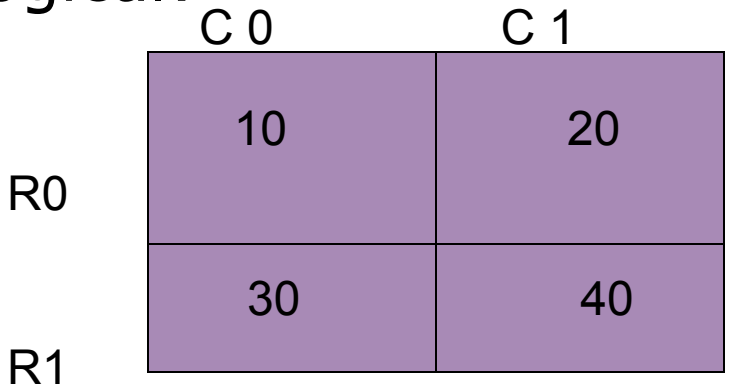
- Two dimensional array can be considered as an array of 1-D array.
- Declaration int arr[2][2];
- This defines a new array containing two elements. Each of these elements is itself an array containing 2 integers.
- Each element of this array is accessed using two indices: row number and the column number.

2-D Array Representation

- Physical:



- Logical:



Using 2D Array

```
Int arr[2][2], I , j;


For( I = 0; I < 2; I ++ )
  for( j = 0; j < 2; j++ )
            arr[ I ][ j ] = I + j;


For( I = 0; I < 2; I ++ )
  for( j = 0; j < 2; j++ )
            cout<< arr[ I ][ j ];
```

Advantages of Arrays

- Simple and easy to understand.
- Contiguous allocation.
- Fast retrieval because of indexed nature.
- No need for the user to be worried about allocation and de allocation of arrays.

Disadvantages of Array

- If you need m elements out of n locations defined. n-m locations are unnecessarily wasted if n>m
- You can not have more than n elements. -
(I.e static allocation of memory.)
- large number of data movements in case of insertion & deletion, which leads to more overheads.

Disadvantages of Array

- If you need m elements out of n locations defined. n-m locations are unnecessarily wasted if n>m
- You can not have more than n elements. -
(I.e static allocation of memory.)
- large number of data movements in case of insertion & deletion, which leads to more overheads.

Linked List

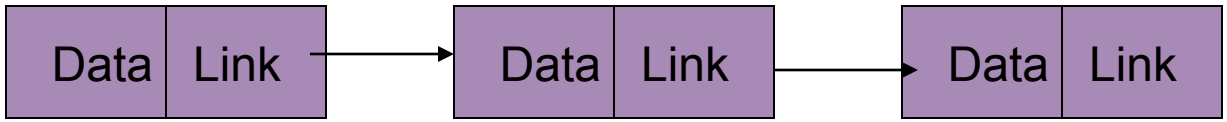
bitCode technologies

Drawbacks of Arrays

- Array does not grow dynamically (i.e length has to be known)
- Inefficient memory management.
- In ordered array insertion is slow.
- In both ordered and unordered array deletion is slow.
- Large number of data movements for insertion & deletion which is very expensive in case of array with large number of elements.
  Solution : Linked list.

Introduction to Linear Linked List

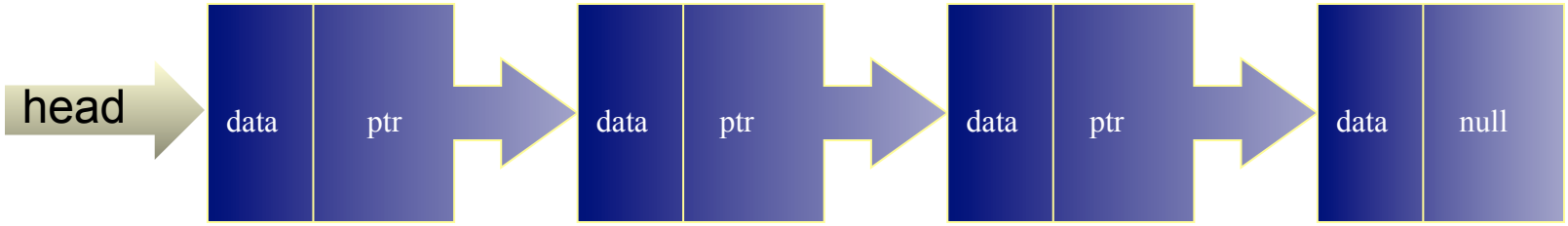
- We can overcome the drawbacks of the sequential storage.
- If the items are explicitly ordered, that is each item contained within itself the address of the next Item.
- Such an explicit ordering gives rise to a data structure called Linear Linked List.
- In Linked Lists elements are logically adjacent, need not be physically adjacent.



Linked List Structure

- Each Item in the list is called a node and contains two fields, an data field and next address field.
- The data field holds the actual element on the list.
- The link or the next address field contains the address of the next node in the list.
- Such an address which is used to access a particular node, is known as a pointer.
- The entire linked list is accessed from an external pointer Head, which points to the first node of the list.
- The next field of the last node in the list contains a special value, known as null, which is not a valid address.
- This null pointer is used to signal the end of the list.

Linked List



- Here the head is a pointer which is pointing to the first node of the list.
- The entire list can be accessed through head.
- So to maintain a list is nothing but maintaining a head pointer for the list.

Operations

- Insert ( )

1. Create a new node.
2. Set the fields of the new node.
3. If the linked list is empty insert the node as the first node.
4. If the node precedes all others in the list then insert the node at the front of the list.
5. Repeat through step 6 while information content of the node in the list is less than the information content of the new node.
6. Obtain the next node in the list.
7. Insert the new node in the list.

Operations

- Delete ( )

1. If the list is empty then write underflow and return.
2. Repeat through step 3 while the end of list has not been reached and the node has not been found.
3. Obtain the next node in the list and record its predecessor node.
4. If the end of the list is reached and node not found then write node not found and return.
5. If found delete the node from the list.
6. Free the node deleted.
7. Set the links of the nodes one which follows the deleted node and one which precedes the deleted node.

Operations

- Search ( )

  Start from the head node. Compare the key with the data item of each node. If match not found and end of list is reached then the element being searched is not present.

  If found return it's position.

- Display ()

  Start with the first node. Print the data of the current node. Get the address of the next node, and make it as current node.

  Continue the process un till the next node is NULL.

Implementation of Linked List - Node

```
class Node {

  int data;
  Node * next;

public:
  Node ( int data ) ;
  int getData ();
  void setData ( int data );
  Node * getNext ();
  void setNext ( Node * next );

};
```

Implementation of Linked List

```
class SinglyLinkedList {

  Node * head;

public:
        SinglyLinkedList();
        void insert ( int data );
        void insertByPos ( int data, int pos );
        void delByVal ( int val );
        void delByPos ( int pos );
        void display ();
        int search ( int val );
        void printReverse();

};
```

Stack

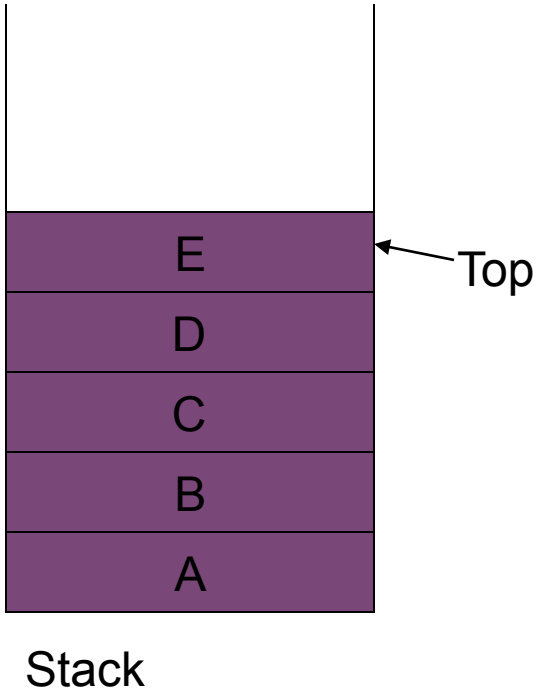
bitCode technologies

learn.innovate.share

Introduction to Stack

- A Stack is an ordered collection of items into which new items may be inserted and from which items maybe deleted at one end called the top of stack.
- The definition of the Stack provides insertion and deletion of items. So a stack is dynamically, constantly changing object.
- The definition specifies that a single end of the stack is designated as the stack top.
- New items may be put on top of the stack or items which are at the top of the stack may be removed.

The Stack Structure



- While adding the elements to stack A is added first then B, C, D, E and along with that the top of the stack keeps growing.
- Initially the Top of the stack is -1.
- While deleting the items from the stack E is deleted first and the D, C, B, A and the stack Top keeps decrementing.

Stack Operations

- You can perform following operations on the Stack.
  - Push
  - Pop
  - StackEmpty
  - StackFull

Stack Operations

Push:

The Items are put on the Stack using Push function.

If S is the Stack the items could be added to it as

```
S.Push( Item );
```

Stack Operations

Pop:

- The items from the stack can be deleted using the Pop operation.

- Using pop only the topmost element can be deleted.

e.g. S.Pop();

- The pop function returns the item that is deleted form the stack.

Stack Operations

- There is one more operation that can be performed on the Stack is to determine what the top item on the stack is without removing it.
- This operation is written as

  S.Peek();

  It returns the top element of the Stack S.
- The operation Peep is not a new operation, since it can be decomposed into Pop & a Push.
- I = S.Peek(); is equivalent to

  I = S.Pop();

  S.Push(I);
- Like the operation Pop, Peep is not defined for an empty stack.

Facts about Stack!

- There is no upper limit on the number of items that may be kept in a Stack, since the definition does not specify how many items are allowed in the Stack collection.
- If the stack does not contain any element then it is called the 'Empty Stack'.
- Although the push operation is applicable to any stack , the Pop operation can not be applied to the Empty stack.
- So it is necessary to check out whether the stack is empty before Popping the elements from the Stack.

Stack Usage

- Stacks are mainly used to support function call mechanism.
- To support or remove recursion.
- Conversion of Infix expressions to post fix expression, pre fix expression, and their evaluation.

Stack Implementation

Stack as an Object

State:

The data, Stack is holding. The stack Top position.

Identity:

Every stack will have a name & location.

Behavior:

Push the elements on Stack.

Pop the elements from Stack.

Responsibility:

Manage the data in last in first out fashion.

Stack Implementation

- Stack can be implemented in two ways
  - Using an Array
  - Using Linked list representation

Class Stack

```
class Stack {

  int * arr;
  int top;

Public:

  Stack();
  Stack( int );
  void push( int data );
  int pop();
  int StackFull();
  int StackEmpty();

};
```

Stack Implementation Using Linked List

- The Stack may be represented by a linear linked list.
- The operation of adding an element to the front of a linked list is quite similar to that of pushing an element onto a stack.
- In both the cases the element is added as the only immediately accessible item in a collection.
- A Stack can be accessed only through it's top element, and a list can be accessed through the pointer to the first element.

List as a Stack

- The operation of removing the first element from a Linked List is analogous to popping a stack.
- In both cases only immediately accessible item of a collection is removed from that collection, and the next item becomes the immediately accessible.
- The first node of the list is the top of the stack.
- The advantage of implementing a stack as a linked list is that stack is able to grow and shrink to any size.
- No space has been pre allocated to any single stack and no stack is using the space that it does not need.

Implementation

```
class Stack {

  Node * top;

Public:

  Stack ();
  void push( int data );
  int pop ();
  int stackEmpty ();

};
```

Linked Lists

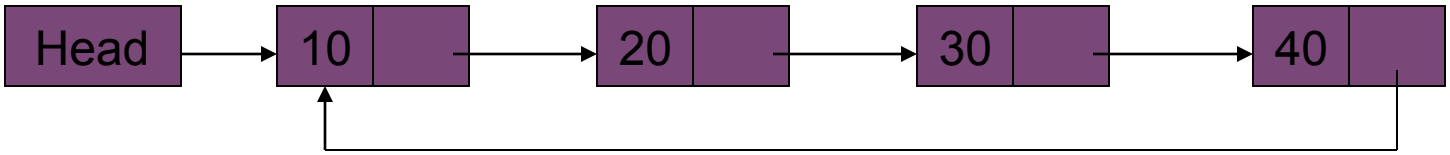
bitCode technologies

learn.innovate.share

Singly Circular Linked List

- A small change is made to the linear list, so that the next field in the last node contains a pointer back to the first node rather than the NULL pointer.
- Such a linked list is called as the circular linked list.
- From any point in such a list it is possible to reach any other point in the list.
- A NULL pointer represents an empty circular linked list.

Structure



• In This case as the list is circular there is no first or last node of the list. However some convention should be used so that a certain node is last node and the node following that is first node.

•The address of the first node is placed in the head.

Singly Circular Linked List

- Advantages

  - Every node is accessible from a given node.

  - In singly linked list to do every operation we have to go sequentially starting from the head. This is not a case with circular linked list.

- Disadvantages :

  - Without some care in programming it is possible to get into an infinite loop (identify the head node).

Class “CLinkedList”

```
class CLinkedList {
  Node * head;

public:
  CLinkedList();
  void insert( int data );
  void insertAtBeg( int data );
  void delByPos( int pos );
  void display();
  ~CLinkedList();
};
```

Doubly Linked List

- Singly and Circular are Linked lists in which we can traverse in only one direction. Sometimes it is required that we should be able to traverse the list in both directions.

For example to delete a node from the list we ought to have a address of previous node in the list.

- For a linked list to have such a property implies that each node must contain two link fields instead of one. The links are used to denote the predecessor and successor of a node.

- A linked list containing this kind of nodes is called Doubly linked list.

Doubly Linked List - Structure

- Here a node is having three parts
  - data part
  - address part to store address of previous node
  - address part to store the address of next node.

Doubly Linked List

- The advantage here is that the list could be traversed in both the directions. Which removes the drawbacks of Singly and Singly circular linked list.
- But still in some situations it may be costly to put two address fields into a Node.

Implementation

```
class DLinkedList {

      Node * head;

public:

  DLinkedList();
  void insert( int data );
  void insert( int data, int pos );
  void deleteByVal( int data );
  void deleteByPos( int pos );
  void display();
  ~DLinkedList();
};
```

Queue

bitCode technologies

learn.innovate.share

Queues

- A Queue is an ordered collection of items from which items may be deleted at one end called front of the queue, and into which items may be inserted at the other end called the rear of the queue.
- The first element inserted in the queue is first element deleted from the queue.

Operations on Queue

- Four primitive operations can be performed on the queue.
- Insert:

  The operation insert inserts the element at the rear of the queue.
- Delete:

  The operation delete deletes the front element from the queue.
- QEmpty:

  Returns true if the queue is empty.
- QFull:

  Returns true if the queue is full.

Queue

- Initially the front and rear is set to -1.
- The queue is empty whenever the rear < front.
- The number of elements in the queue at any time is equal to the value rear – front.
- In simple queues it is possible to reach the absurd situation where the queue is empty, yet no new element could be added.
- One solution is to modify the remove operation so that when an item is deleted the entire queue is shifted to the beginning of the array.

Queue

- The queue will no longer need a front field. Since the element at the position 0 is always the first element.
- The empty queue is represented by the condition if rear equals -1.
- This method is inefficient as every time an element is deleted all the elements need to be shifted.
- Another solution is building a queue as a linked list or as an circular queue.

Queue Implementation using array

```
class Queue {

  int * arr;
  int size, front, rear;

public:

  Queue( int size);
  void insert( int data );
  int deleteData ( );
  int queueEmpty ( );
  int queueFull ( );

};
```

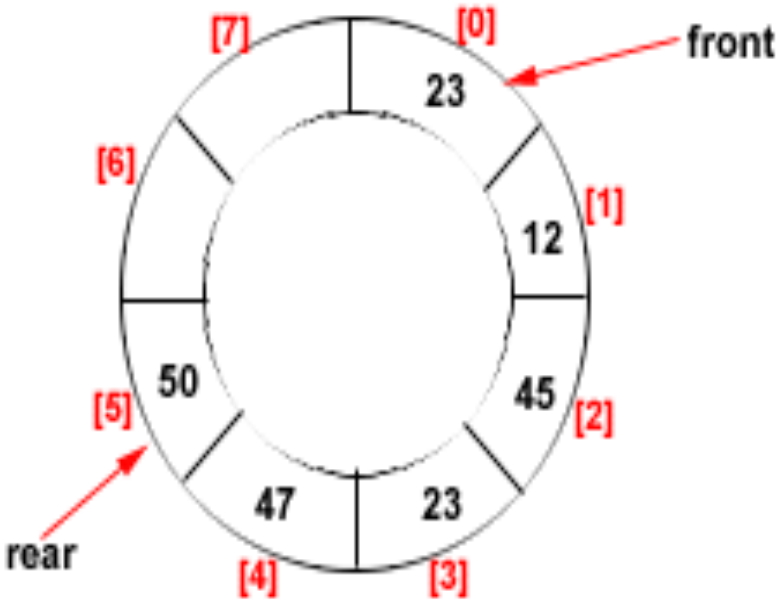
Queue Implementation using Lists

```
class Queue {

  Node * front;
  Node * rear;

public:
  Queue( );
  void insert( int data );
  int deleteData ( );
  int queueEmpty ( );
  int queueFull ( );

};
```

Circular Queue

- Static queues have a very big drawback that once the queue is FULL, even though we delete few elements from the "front" and relieve some occupied space, we are not able to add anymore elements, as the "rear" has already reached the Queue's rear most position.
- The solution lies in a queue in which the moment "rear" reaches the Queue's last position, the "first" element will become the queue's new "rear".
- As the name suggests, this Queue is not straight but circular, and so called as “Circular Queue”.

Struture



- Here in this queue as the rear will reach the last position, then it will be advanced to zero so that memory locations can be reused.

- The operations that can be performed on circular queue are
  - Insert
  - Delete
  - QueueEmpty
  - QueueFull
  - Display

bitCode technologies

Is the queue Empty or Full

- It is difficult under this representation to determine when the queue is empty.
- Both an empty queue and a full queue would be indicated by having the head and tail point to the same element.
- There are two ways around this: either maintain a variable with the number of items in the queue, or create the array with one more element that you will actually need so that the queue is never full.

Other Types Of Queues

- **DQueue:**

  It is a linear list in which elements can be added or deleted at either ends of the queue.

- **Priority Queue:**

  It is a linear list in which elements are stored according to their priority of processing.

- **Bulk Move Queues:**

  Here make the array much bigger than the maximum number of items in the queue. When you run out of room at the end of the array, you move the entire contents of the queue back to the front of the array (this is the "bulk move" from which the structure gets its name).

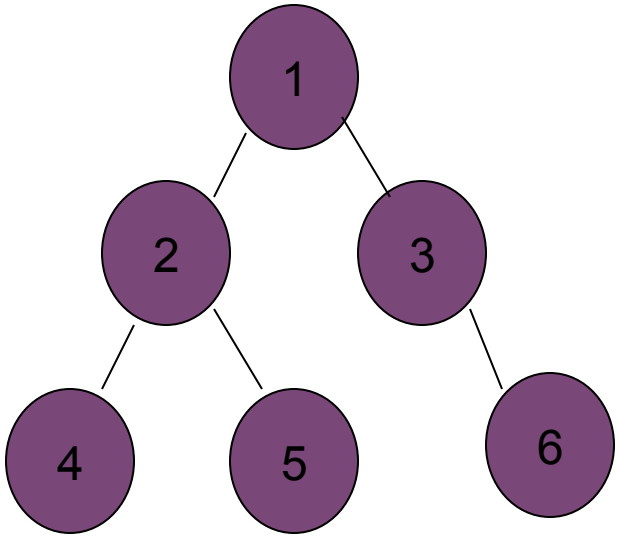
Trees

bitCode technologies

bitCode technologies

General Definition of Tree

- A tree consists of nodes connected by edges, which do not form cycle.
- For collection of nodes & edges to define as tree, there must be one & only one path from the root to any other node.
- A tree is a connected graph of N vertices with N-1 Edges.
- Tree is nonlinear data structure.



Recursive definition of Tree

A tree is finite set of one or more nodes such that:

a) There is specially designated node called root.

b) The remaining nodes are partitioned into n>=0 disjoint sets T1… Tn where each of these sets is a tree. T1….Tn are called subtree of the root.

Tree Terminologies

Node: A node stands for the item of information plus the branches of other items. E.g. 1,2,3,4,5,6

Siblings: Children of the same parent are siblings. E.g. siblings of node 2 are 4 & 5.

Degree: The number of sub trees of a node is called degree. The degree of a tree is the maximum degree of the nodes in the tree. E.g. degree of above tree is 2.

Leaf Nodes: Nodes that have the degree as zero are called leaf nodes or terminal nodes. Other nodes are called non terminal nodes.

Tree Terminologies

Ancestor: The ancestor of a node are all the nodes along the path from the root to that node.

Level: The level of a node is defined by first letting the root of the tree to be level = 1. If a node is at level x then the children are at level x+1.

Height/Depth: The height or depth of the tree is defined as the maximum level of any node in the tree.

Binary Tree

- If the degree of a tree is 2 then it is called binary tree.
- A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint binary trees called left sub tree & right sub tree.
- The max number of nodes on level n of binary tree is 2^(n-1), n>= 1.
- The max number of nodes in binary tree of level k is (2^k)-1.
- If every non leaf node in a binary tree has non empty left and right sub trees, the tree is termed as a strictly binary tree.

Application

Information retrieval is the most important use of Binary tree.

A binary tree is useful when two way decisions must be made at each point in a process.

e.g. Binary search tree.

In decision trees where each node has a condition and based on the answer to the condition the tree traversal takes place.

Tree traversals

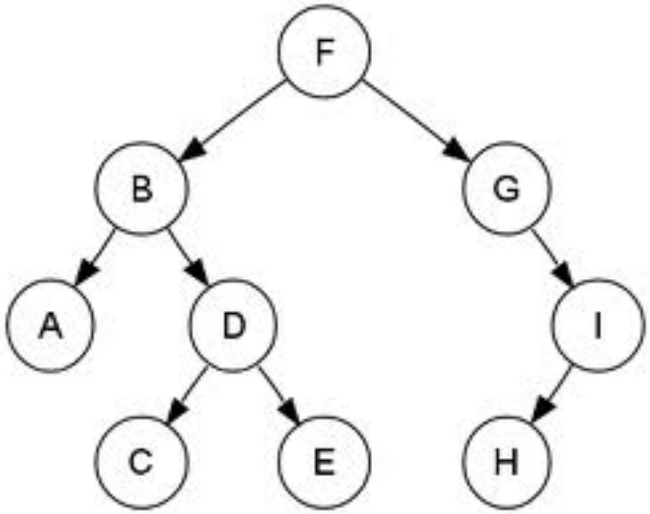
- Preorder
- Inorder
- Postorder

Algorithm for Preorder traversal (DLR)

- Visit the node first
- Traverse left subtree in preorder
- Traverse right subtree in preorder
- Continue this process till all nodes have been visited

Preorder traversal

- Find out Preorder traversal for following tree.



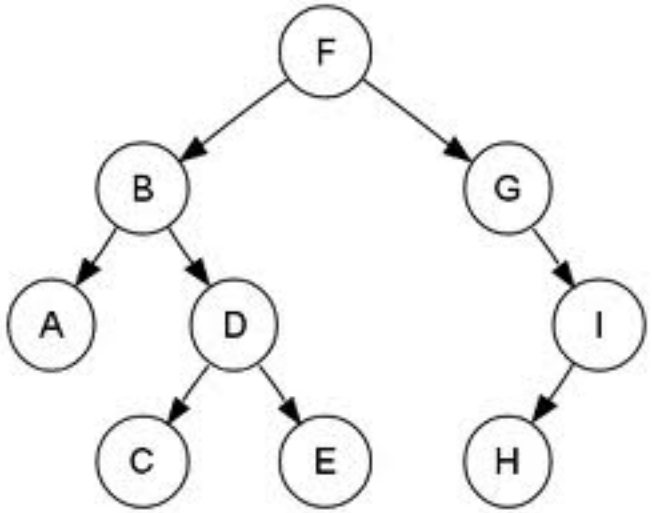
- F B A D C E G I H

Algorithm for Inorder traversal (LDR)

- Traverse left subtree in inorder
- Visit the node
- Traverse right subtree in inorder
- Continue this process till all nodes have been visited

Inorder traversal

- Find out Inorder traversal for following tree.



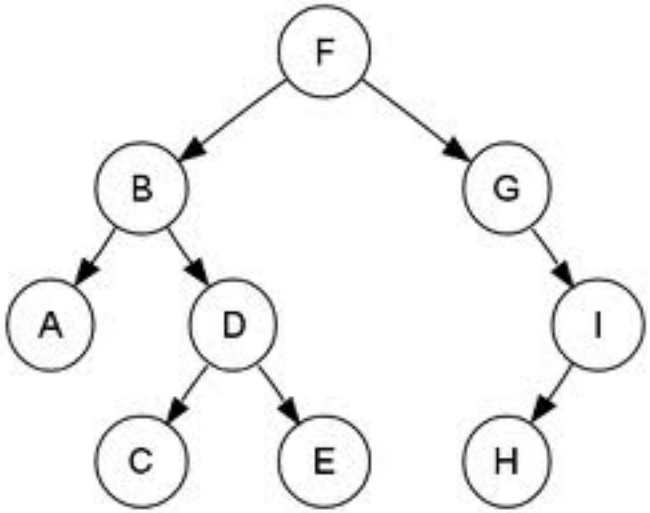
- A B C D E F G H I

Algorithm for Postorder traversal (LRD)

- Traverse left subtree in postorder
- Traverse right subtree in postorder
- Visit the node
- Continue this process till all nodes have been visited

Postorder traversal

- Find out Postorder traversal for following tree.



- A C E D B H I G F

Level wise printing data in a tree (BFS)

- Insert the root node into queue
- While the queue is not empty do the following
- Delete a node from the queue and print it
- Insert a node corresponding to it's left subtree into the queue if it is not NULL
- Insert a node corresponding to it's right subtree into the queue if it is not NULL
- Go to the step 2
- Stop

Classes for Binary Tree

- Class Node will have the following structure

```
class Node {
        int data;
        Node * right;
        Node * left;
  public:
  Node( int data );
  int  getData();
  void setData( int );
  void setRight( Node * );
  Node * getRight();
  void setLeft( Node * );
  Node * getLeft();

};
```

Class BinaryTree will have the following structure

```
class BinaryTree {
        Node * root;
public:
        TBinaryTree();
        void insert( int );
        void inOrder();
        void preOrder();
        void postOrder();
        void deleteData( int );
        ~BinaryTree();
};
```

bitCode technologies

Binary search tree

It is a binary tree that is either empty or in which each node contains a key that satisfies the conditions:

1. All keys (if any) in the left sub tree of the root precede the key in the root.

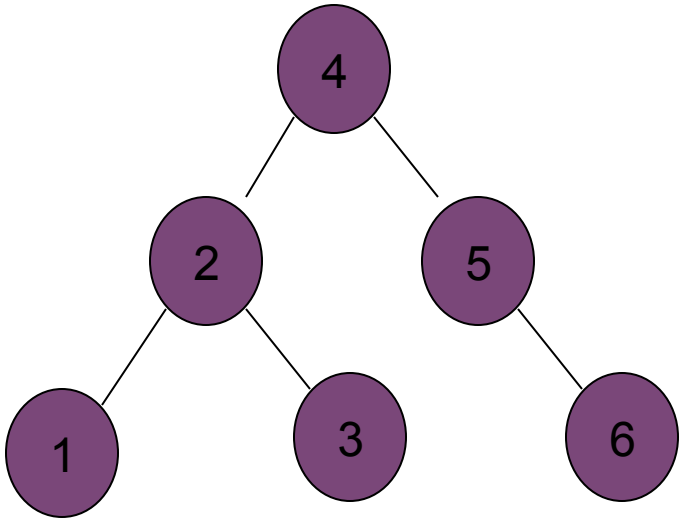
2. The key in the root precedes all keys (if any) in the right sub tree.

3. The left and right sub trees of the root are again search trees.

Example Binary search Tree

Put following numbers in a linked list into a binary search tree

4,2,5,1,3,6



Exercise

Create a binary search tree for the following sequence of numbers

10, 30, 50, 5, 40, 7, 3, 45, 20

Traverse the created in

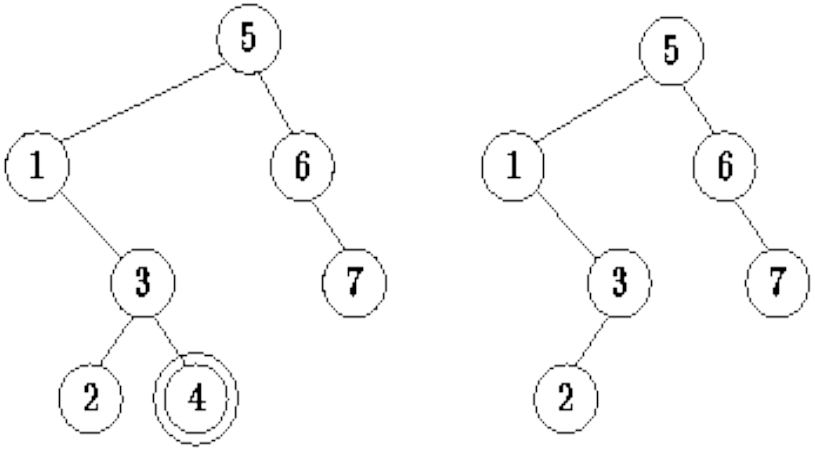
1. Preorder
2. Inorder
3. Postorder

Removing Items from Binary Search Tree

- When removing an item from a search tree, it is imperative that the tree satisfies the data ordering criterion.
- If the item to be removed is in a leaf node, then it is fairly easy to remove that item from the tree since doing so does not disturb the relative order of any of the other items in the tree.

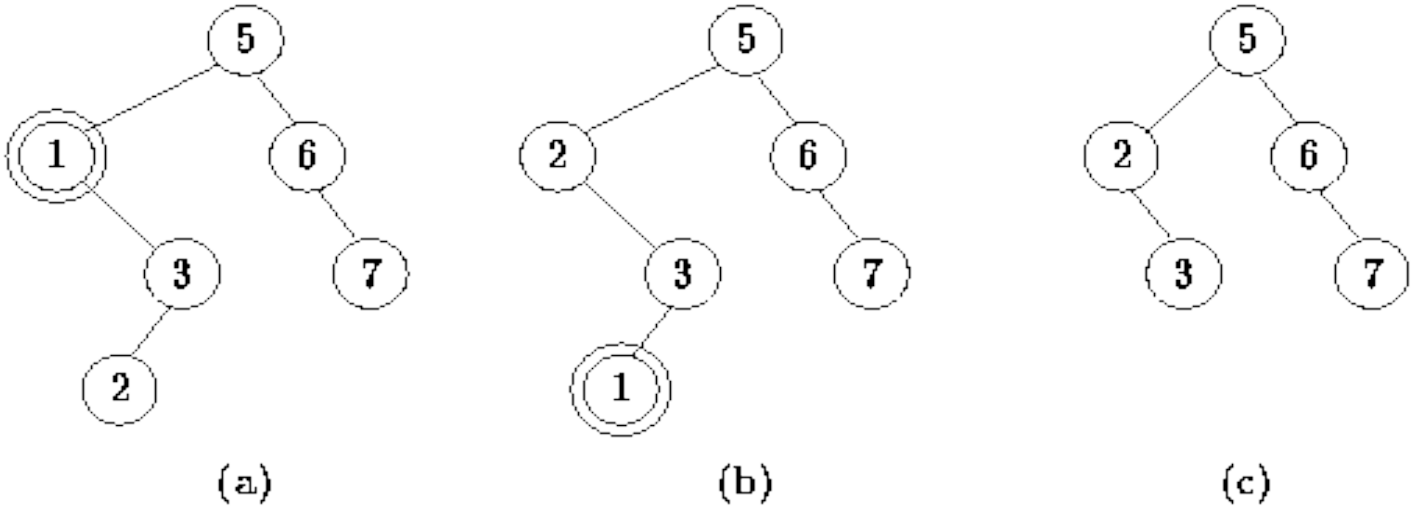
Example

- For example, consider the binary search tree shown in Figure (a). Suppose we wish to remove the node labeled 4. Since node 4 is a leaf, its subtrees are empty. When we remove it from the tree, the tree remains a valid search tree as shown in Figure (b).



Removing Non Leaf Node from a Binary Search Tree

- To remove a non-leaf node, we move it down in the tree until it becomes a leaf node since a leaf node is easily deleted.
- To move a node down we swap it with another node which is further down in the tree.
- For example, consider the search tree shown in Figure (a). Node 1 is not a leaf since it has an empty left sub tree but a non-empty right sub tree.
- To remove node 1, we swap it with the smallest key in its right sub tree, which in this case is node 2, Figure (b).
- Since node 1 is now a leaf, it is easily deleted. Notice that the resulting tree remains a valid search tree, as shown in Figure (c).



- To move a non-leaf node down in the tree, we either swap it with the smallest key in the right subtree or with the largest one in the left subtree.
- At least one such swap is always possible, since the node is a non-leaf and therefore at least one of its subtrees is non-empty.
- If after the swap, the node to be deleted is not a leaf, then we push it further down the tree with yet another swap. Eventually, the node must reach the bottom of the tree where it can be deleted

Creation Binary Tree from Traversal Sequence

- Creation of Binary Tree from Preorder & Inorder Traversals
  - E.g. Inorder E A C K F H D B G
  - Preorder F A E K C D H G B

- Creation of Binary Tree from Postorder & Inorder Traversals.
  - Inorder: B I D A C G E H F
  - Postorder: I D B G C H F E A

Expression Tree

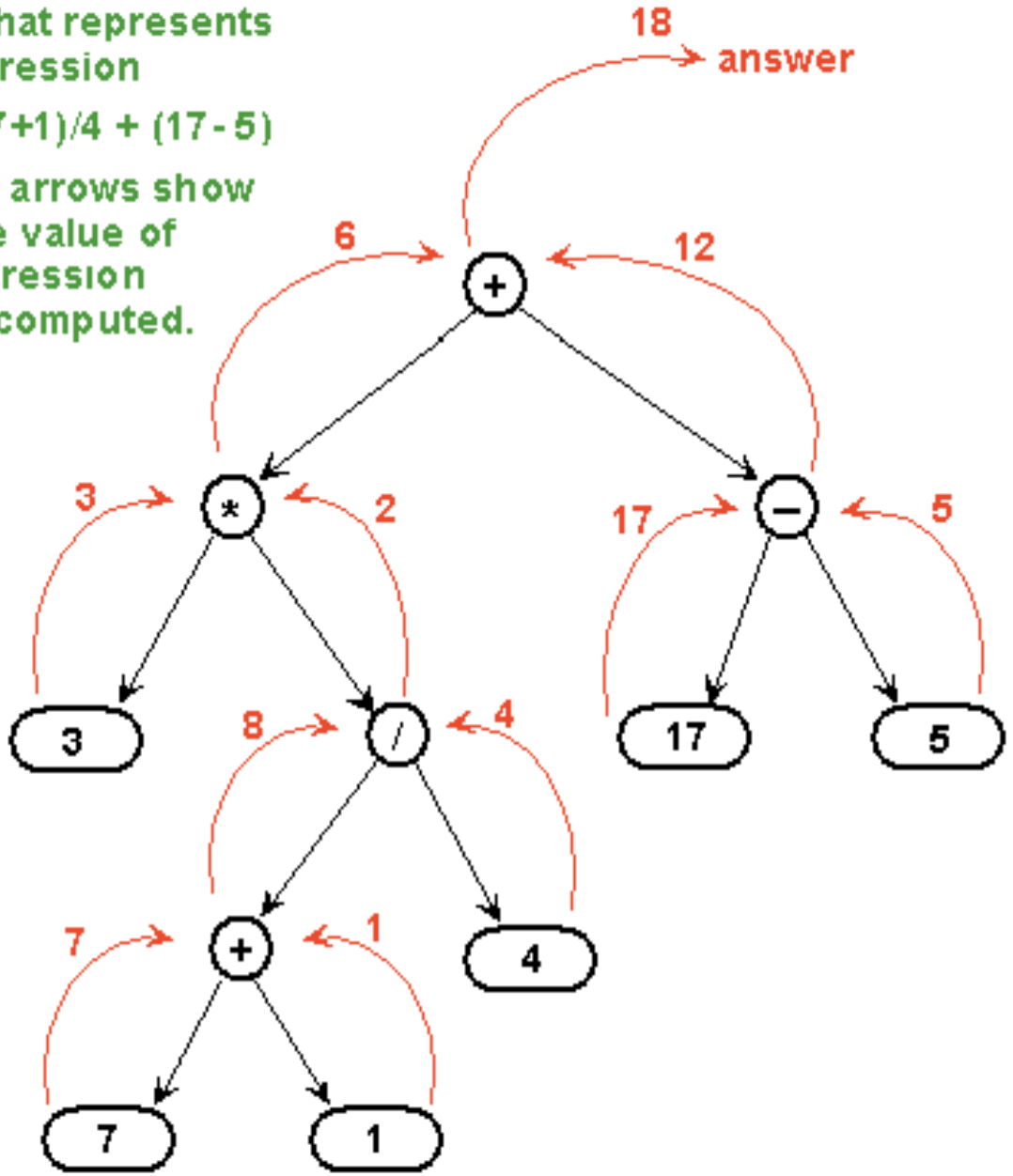
- When an expression is represented through a tree, it is known as expression tree.
- The leaves of an expression tree are operands, such as constant variable names and all internal nodes contain operators.
- Preorder traversal of an expression tree gives prefix equivalent of the expression.
- Postorder traversal of an expression gives the postfix equivalent of the expression.

Expression Tree

A tree that represents the expression

3 * (7+1)/4 + (17 - 5)

The red arrows show how the value of the expression can be computed.



bitCode technologies

Threaded Binary Search Tree

- A **threaded binary search tree** may be defined as follows:

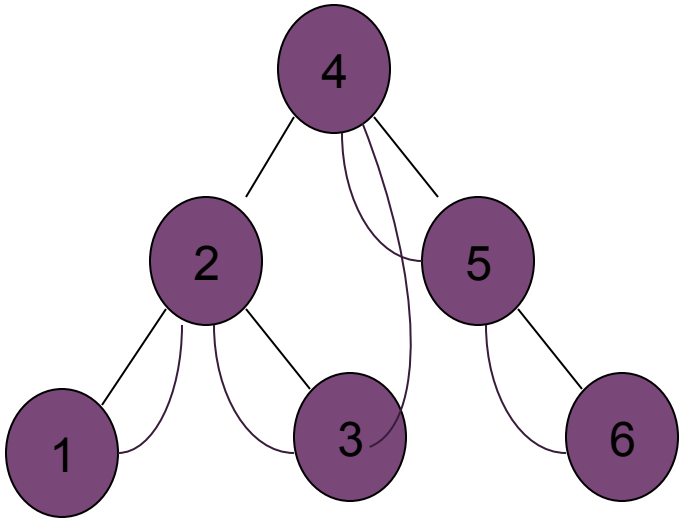
  A binary tree is *threaded* by making all right child pointers that would normally be null point to the inorder successor of the node, and all left child pointers that would normally be null point to the inorder predecessor of the node.“

- A threaded binary tree makes it possible to traverse the values in the binary tree via a linear traversal that is more rapid than a recursive in-order traversal.

- It is also possible to discover the parent of a node from a threaded binary tree, without explicit use of parent pointers or a stack.

- This can be useful however where stack space is limited.

Example



In order Traversal: 1 2 3 4 5 6

Classes for Threaded Binary Tree

```
class Node {

  int data;
  Node * left, * right;
  char lflag,rflag;

public:

  Node( int data );
  int getData();
  void setRight( Node * );
  Node * getRight();
  void setLeft( Node * );
  Node * getLeft();
  void setlflag( char );
  char getlflag();
  void setrflag( char );
  char getrflag();

};
```

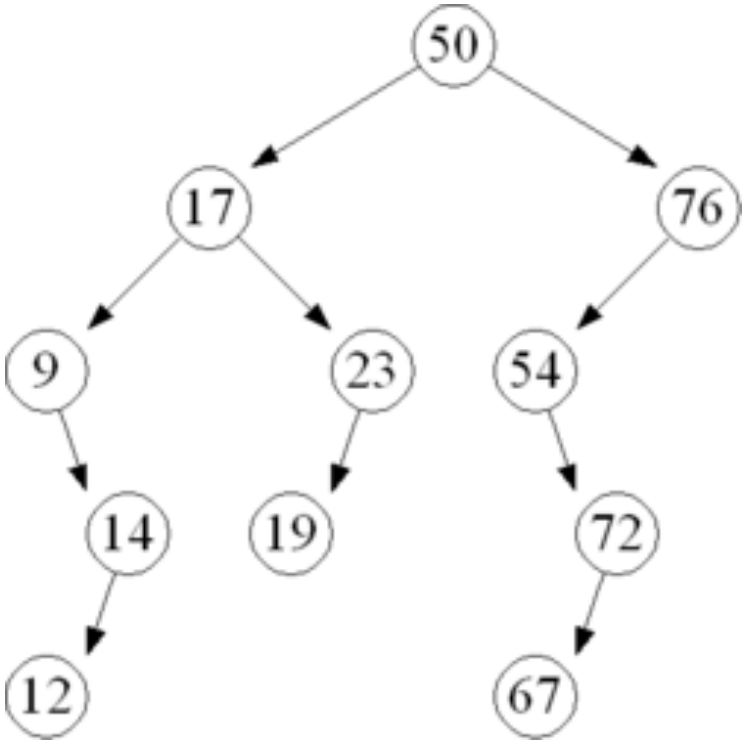
```
class TBinaryTree {
        Node * root;
public:
        TBinaryTree();
        void insert( int );
        void inOrder();
        void postOrder();
        void preOrder();
        void deleteData( int );
        ~TBinaryTree();
};
```

bitCode technologies

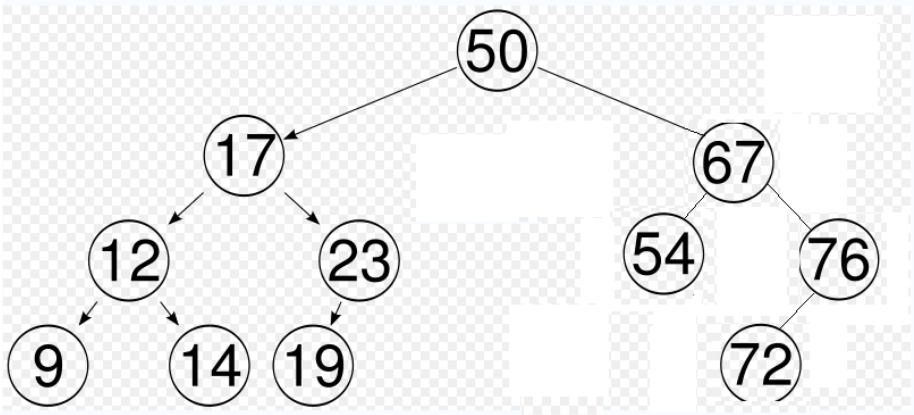
Introduction to AVL Trees

- An **AVL tree** is a self-balancing binary search tree .
- In an AVL tree the heights of the two child sub trees of any node differ by at most one, therefore it is also called height-balanced tree.
- Additions and deletions may require the tree to be rebalanced by one or more tree rotations .
- The **balance factor** of a node is the height of its right sub tree minus the height of its left sub tree.
- A node with balance factor 1, 0, or -1 is considered balanced. A node with any other balance factor is considered unbalanced and requires rebalancing the tree.
- The balance factor is either stored directly at each node or computed from the heights of the subtrees.

Example AVL Tree



e.g: Non AVL Tree



e.g. AVL Tree

bitCode technologies

Insertion into AVL Trees

- Insertion into an AVL tree may be carried out by inserting the given value into the tree as if it were an unbalanced binary search tree, and then retracing one's steps toward the root updating the balance factor of the nodes.

- Retracing is stopped when a node's balance factor becomes 0, 1, or -1.

- If the balance factor becomes 0 then the height of the sub tree hasn't changed because of the insert operation. The insertion is finished.

Deletion from AVL Tree

- If the node is a leaf, remove it. If the node is not a leaf, replace it with either the largest in its left subtree or the smallest in its right subtree, and remove that node.
- After deletion retrace the path back up the tree to the root, adjusting the balance factors as needed.

The AVL Tree Rotations **Left Rotation (LL)**

Imagine we have this situation:

```
    a
     \
      b
       \
        c
```

- To fix this, we must perform a left rotation, rooted at A. This is done in the following steps:
  - b becomes the new root.
  - a takes ownership of b's left child as its right child, or in this case, null.
  - b takes ownership of a as its left child.
- The tree now looks like this:

```
     b
    / \
   a   c
```

bitCode technologies

The AVL Tree Rotations **Right Rotation (RR)**

- A right rotation is a mirror of the left rotation operation described above. Imagine we have this situation:

```
      c
     /
    b
   /
  a
```

- To fix this, we will perform a single right rotation, rooted at C. This is done in the following steps:
  - b becomes the new root.
  - c takes ownership of b's right child, as its left child. In this case, that value is null.
  - b takes ownership of c, as it's right child.

- The resulting tree:

```
    b
   / \
  a   c
```

bitCode technologies

The AVL Tree Rotations **Left-Right Rotation (LR) or "Double left"**

Sometimes a single left rotation is not sufficient to balance an unbalanced tree. Take this situation:

```
  a
   \
    c
```

- It's balanced. Let's insert 'b'.

```
  a
   \
    c
   /
  b
```

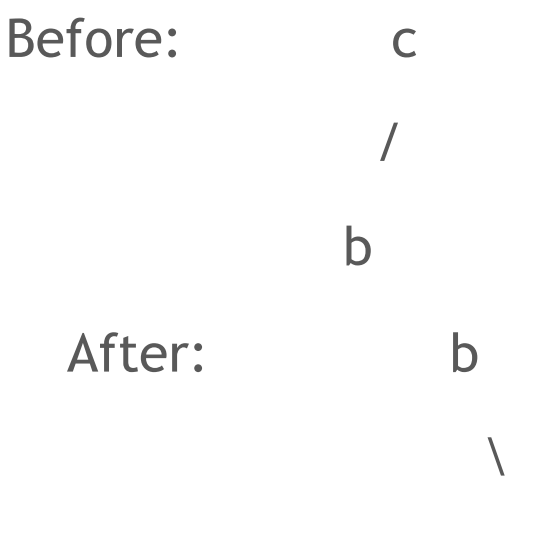
- Our initial reaction here is to do a single left rotation. Let's try that.

```
    c
   /
  a
   \
    b
```

bitCode technologies

The AVL Tree Rotations **Left-Right Rotation (LR) or "Double left“**

- This is a result of the right subtree having a negative balance.
- Because the right subtree was left heavy, our rotation was not sufficient.
- The answer is to perform a right rotation on the right subtree. We are not rotating on our current root. We are rotating on our right child.
- Think of our right subtree, isolated from our main tree, and perform a right rotation on it:



bitCode technologies

The AVL Tree Rotations **Left-Right Rotation (LR) or "Double left“**

- After performing a rotation on our right subtree, we have prepared our root to be rotated left. Here is our tree now:

left rotation.

Right-Left Rotiation (RL) or "Double right"

- A double right rotation, or right-left rotation, is a rotation that must be performed when attempting to balance a tree which has a left subtree, that is right heavy.

- This is a mirror operation of what was illustrated in the section on Left-Right Rotations. Let's look at an example of a situation where we need to perform a Right-Left rotation.

```
    c
   /
  a
   \
    b
```

- The left subtree has a height of 2, and the right subtree has a height of 0. This makes the balance factor of our root node, c, equal to -2. Some kind of right rotation is clearly necessary, but a single right rotation will not solve our problem.

```
  a
   \
    c
   /
  b
```

bitCode technologies

Right-Left Rotiation (RL) or "Double right"

- The reason our right rotation did not work, is because the left subtree, or 'a', has a positive balance factor, and is thus right heavy. Performing a right rotation on a tree that has a left subtree that is right heavy will result in the problem .

- The answer is to make our left subtree left-heavy. We do this by performing a left rotation our left subtree. Doing so leaves us with this situation:

```
      c
     /
    b
   /
  a
```

- This is a tree which can now be balanced using a single right rotation. We can now perform our right rotation rooted at C. The result:

```
    b
   / \
  a   c
```

bitCode technologies

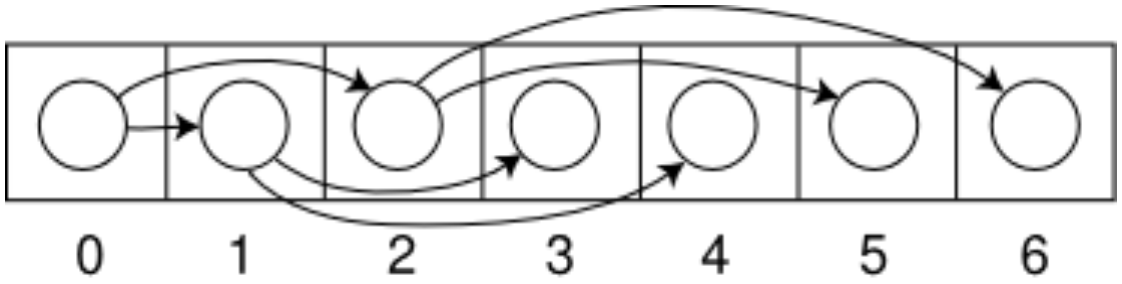
Rotations, When to Use

```
IF tree is right heavy   {

   IF tree's right subtree is left heavy
     Perform Double Left rotation
   }
   else   {
     Perform Single Left rotation
   }

}

Else IF tree is left heavy {
     IF tree's left subtree is right heavy   {
        Perform Double Right rotation
     }
   ELSE   {
     Perform Single Right rotation
   }

}
```

bitCode technologies

Binary Tree Representation Using Array

- Binary trees can also be stored as an implicit data structure in arrays, and if the tree is a complete binary tree, this method wastes no space.
- In this compact arrangement, if a node has an index *i*, its children are found at indices $2i + 1$ and $2i + 2$, while its parent (if any) is found at index (i-1)/2 (assuming the root has index zero).
- This method benefits from more compact storage and better locality of reference, particularly during a preorder traversal.



- However, it is expensive to grow and wastes space.

B Tree & B+ Tree

bitCode technologies

learn.innovate.share

B Tree

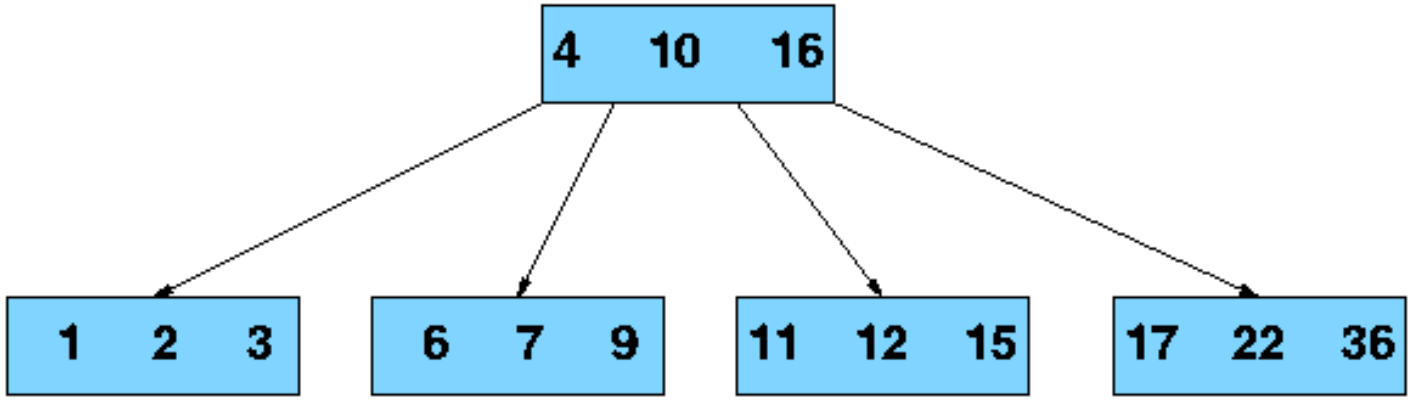
- In computer science, a **B-tree** is a tree data structure that keeps data sorted and allows searches, insertions, and deletions in logarithmic amortized time. It is most commonly used in databases and filesystems.
- Each node of a b-tree may have a variable number of keys and children.
- Each key has an associated child that is the root of a subtree containing all nodes with keys less than or equal to the key but greater than the preceeding key.
- A node also has an additional rightmost child that is the root for a subtree containing all keys greater than any keys in the node.

B Tree

- A b-tree has a minumum number of allowable children for each node known as the *minimization factor*. If *t* is this *minimization factor*, every node must have at least *t - 1* keys.
- Since each node tends to have a large branching factor (a large number of children), it is typically necessary to traverse relatively few nodes before locating the desired key.
- If access to each node requires a disk access, then a b-tree will minimize the number of disk accesses required.
- The minimzation factor is usually chosen so that the total size of each node corresponds to a multiple of the block size of the underlying storage device.

B Tree

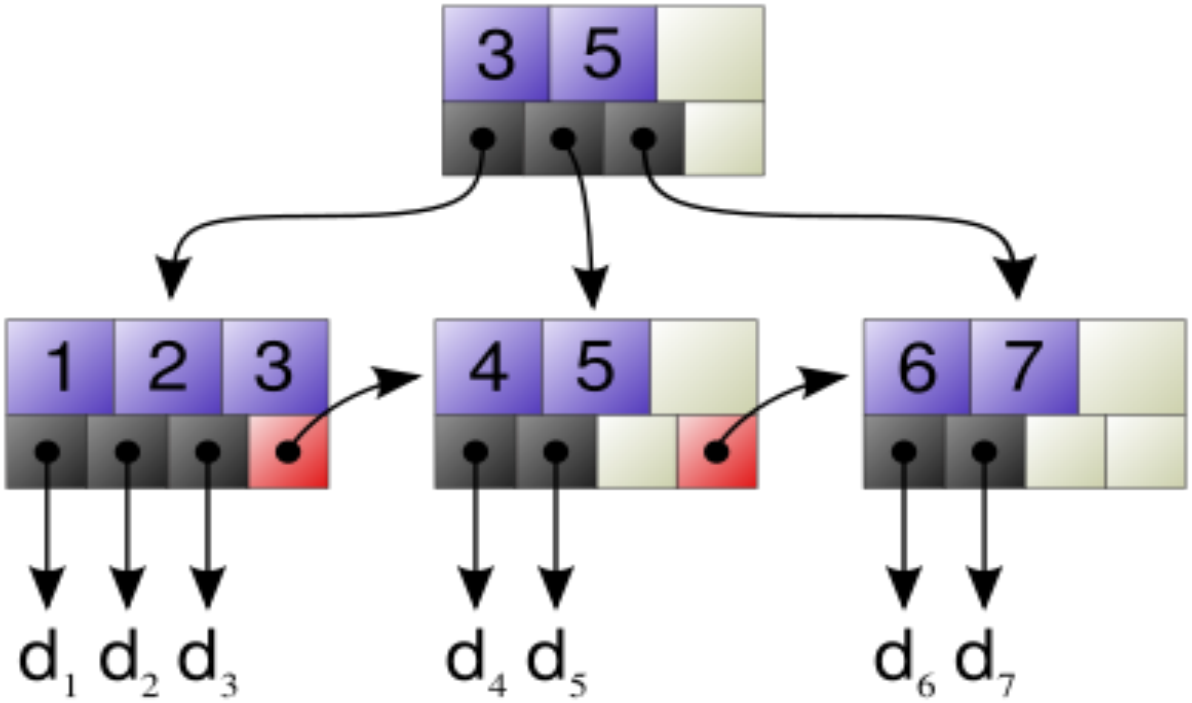
- This choice simplifies and optimizes disk access. Consequently, a b-tree is an ideal data structure for situations where all data cannot reside in primary storage and accesses to secondary storage are comparatively expensive (or time consuming).



B+ Tree

- a **B+ tree** (also known as a **Quaternary Tree**) is a type of tree, which represents sorted data in a way that allows for efficient insertion, retrieval and removal of records, each of which is identified by a *key*.
- In a B+ tree, in contrast to a B-tree, all records are stored at the lowest level of the tree; only keys are stored in interior blocks.
- The ReiserFS filesystem (for Unix and Linux), XFS filesystem (for IRIX and Linux), JFS2 filesystem (for AIX, OS/2 and Linux) and NTFS filesystem (for Microsoft Windows) all use this type of tree for block indexing. Relational databases also often use this type of tree for table indices.

B+ Tree



Graphs

bitCode technologies

learn.innovate.share

Introduction to Graphs

- A Graph is a collection of nodes, which are called vertices ‘V’, connected in pairs by line segments, called Edges E.

- Sets of vertices are represented as V(G) and sets of edges are represented as E(G).

So a graph is represented as G = (V,E).

- There are two types of Graphs

  - Undirected Graph

  - Directed Graph

Undirected Graph

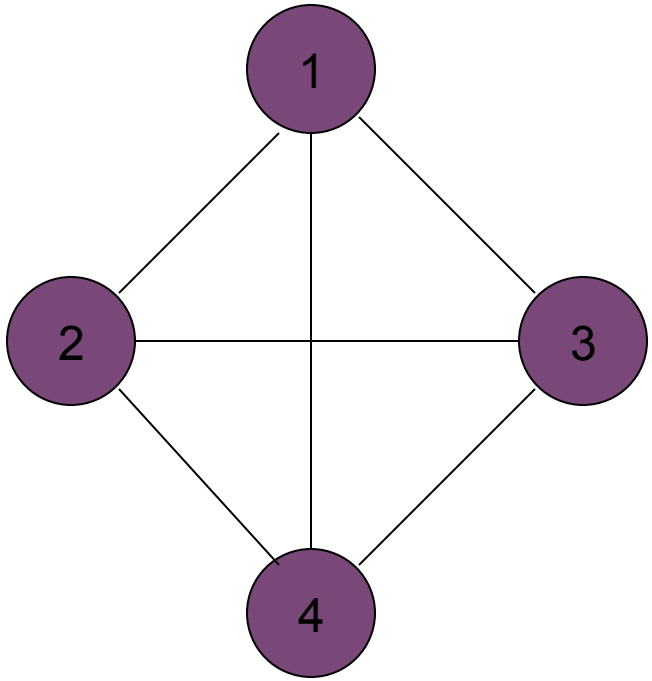
- An undirected Graph is one, where each edge E is an unordered pair of vertices. Thus pairs (v1, v2) & (v2, v1) represents the same edge.

G1

V(G1) = { 1, 2, 3, 4 };

E(G1) = { (1,2), (1,3), (1,4),

(2, 3), (2, 4), (3, 4) };



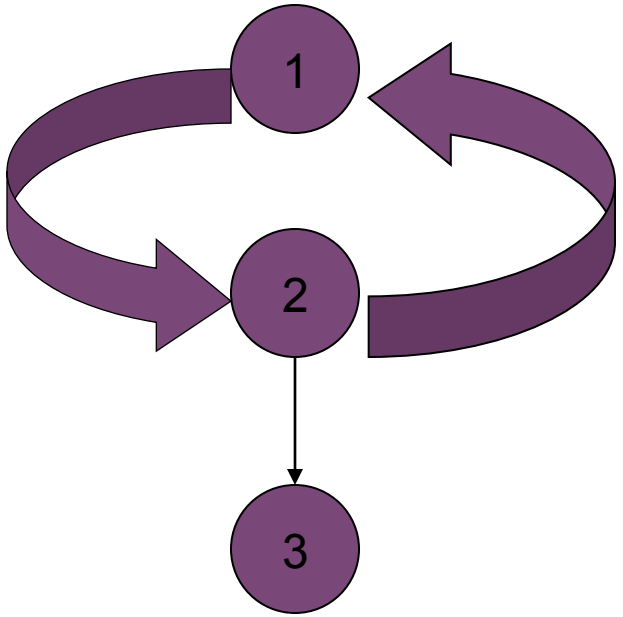
Directed Graph

- Directed Graph are usually referred as Digraph for Simplicity.
- A directed Graph is one, where each edge is represented by a specific direction or by a directed pair <v1, v2>.
- Hence <v1, v2> & <v2, v1> represents two different edges.

G2

V(G2) = { 1, 2, 3 };

E(G2) = { < 1, 2 >, < 2 , 1> , < 2, 3 > };



Definitions Related To Graphs

- Out – degree: The number of Arc exiting from the node is called the out degree of the node.
- In- Degree: The number of Arcs entering the node is the In degree of the node.
- Sink node: A node whose out-degree is zero is called Sink node.
- Path: path is sequence of edges directly or indirectly connected between two nodes.
- Cycle: A directed path of length at least L which originates and terminates at the same node in the graph is a cycle

Definitions Related To Graphs

- Adjacent: Two vertices in an undirected Graph are called adjacent if there is an edge from the first to the second.
- Incident: In an Undirected graph if e=(v, w) is an edge with vertices v and w, then v and W are said to lie on e, and e is said to be incident with v and w.
- Sub-Graph: A sub-graph of G is G1 if

  V(G1) is subset of V(G) .

  and E(G1) is subset of E(G)

Implementing Graph

- The graphs can be implemented in two ways

  1. Array Method

  2. Linked List

Array Method

- In this an array is used to store the values of nodes.

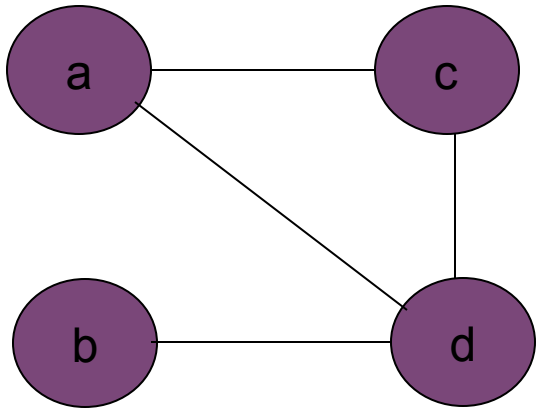
  e.g. V[] = { a, b, c, d };

  It means that node 1 has value a, 2 has value b, 3 has value c and 4 has value d.

- To show the connectivity between nodes a two dimensional array is used.
- This matrix will have an entry ‘1’ if there is an edge between the two nodes, otherwise ‘0’.
- This matrix is known as “adjacency matrix”.
- In undirected graph this matrix will be symmetric.

Example – Array Method

- Char v[] = { a, b, c, d}



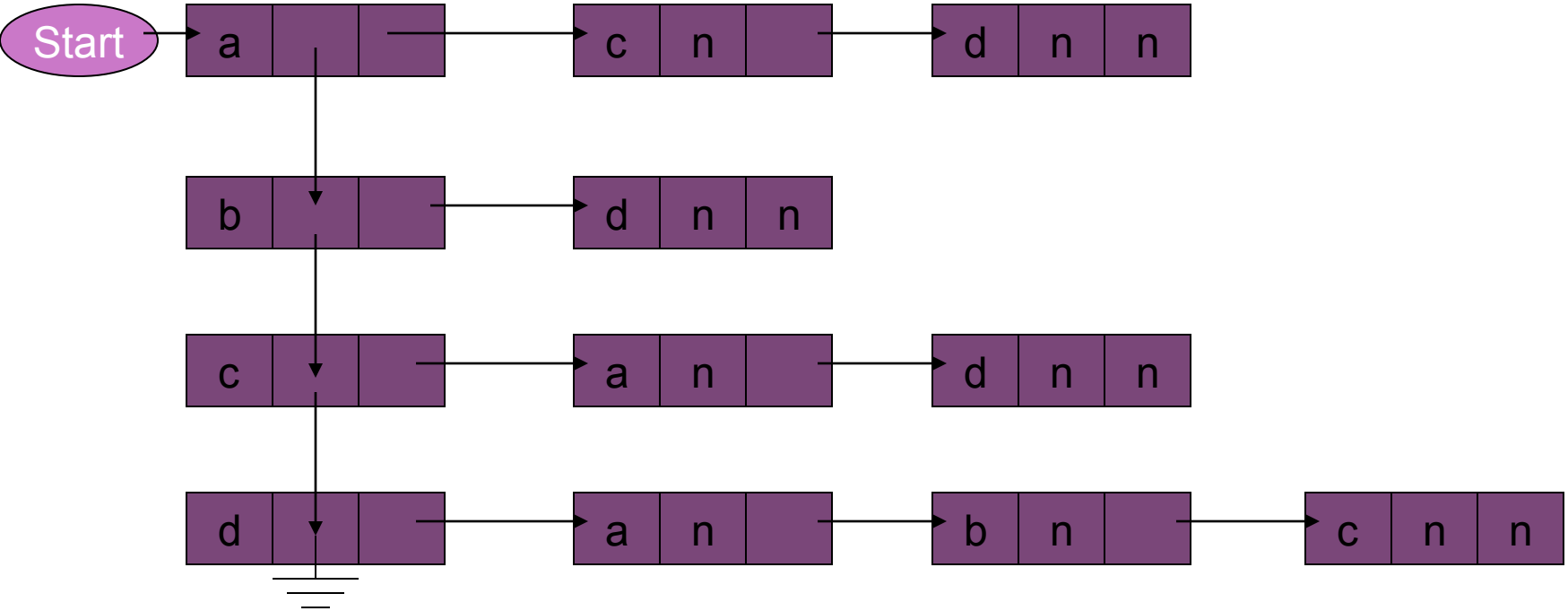
- Adj – Matrix adj[][4] =

	1	2	3	4
1	0	0	1	0
2	0	0	0	1
3	1	0	0	1
4	1	1	1	0

Linked List Method

- The information and edges must be stored in the node structure.
- Consider the same graph in last example.
- The linked list representation is:

Linked List Representation of graphs



Graph traversal

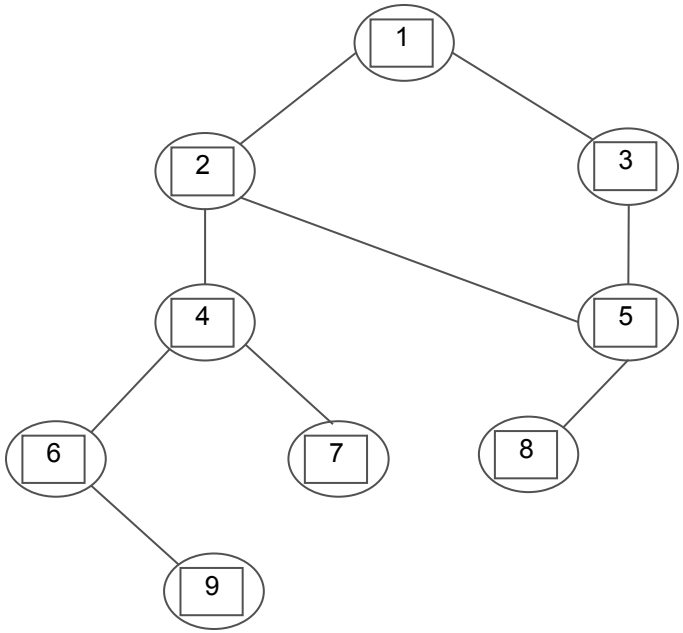
- Depth first search
- Breadth first search

Algorithm for Depth first search

- Step 1 : Select any node in the graph. Visit that node. Mark this node as visited and push this node onto the stack.
- Step 2: Find the adjacent node to the node on top of the stack, and which is not yet visited. Visit this new node. Make this node as visited and push it onto the stack.
- Step 3: Repeat the step 2 until no adjacent node to the top of stack node can be found. When no new adjacent node can be found, pop the top of stack.
- Step 4 : Repeat step 2 and 3 till stack becomes empty.
- Step 5: Repeat above steps if there are any more nodes which are still unvisited.

Depth first search

- Find out DFS for following graph.



- 1 2 4 6 9 7 5 8 3

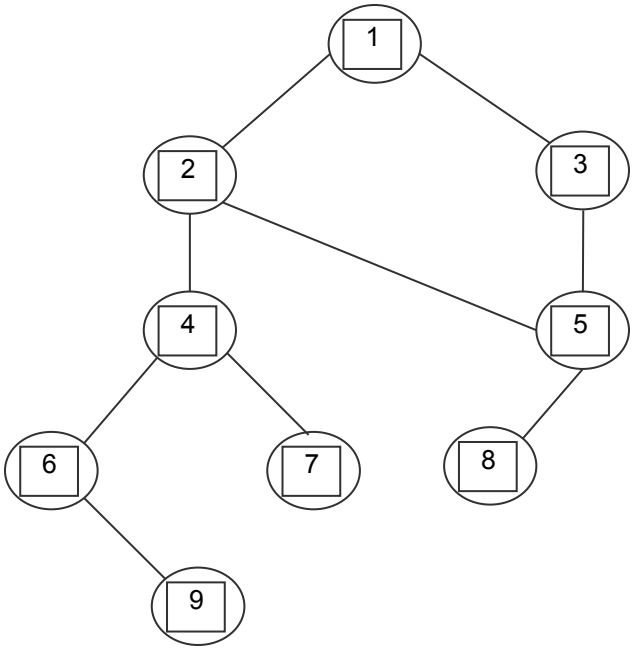
bitCode technologies

Algorithm for Breadth first search

- Step 1 : Start with any node and mark it as visited & place it into the queue.
- Step 2: Find the adjacent nodes to the node marked in step 1 and place it in the queue.
- Step 3: Visit the node at the front of the queue. Delete from the queue, place it's adjacent nodes in the queue.
- Step 4 : Repeat step 3 till the queue is not empty.
- Step 5: Stop.

Breadth first search

- Find out BFS for following graph.



- 1 2 3 4 5 6 7 8 9

bitCode technologies

Applications of graph

- Game theory
- Telephone networking
- Scheduling of interrelated tasks for job
- Routing from one location to another

Spanning Tree

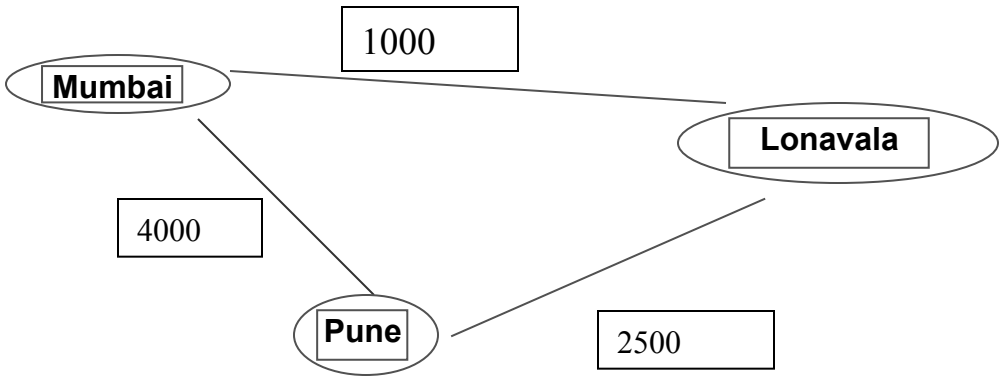
- A Sub-graph of a graph ‘g’ is a tree containing all the nodes of ‘g’ but less number of links with minimum cost.
- Minimal spanning tree is a spanning tree with smallest possible weight values

Prim's algorithm

- This method builds minimum spanning tree edge by edge
- Select an edge from the graph whose cost is minimum among all the edges.
- Add the next edge (i,j) to the tree such that vertex i is already in the tree & vertex j is new one & cost of the edge (i,j) is minimum among all the edges (k.l) such that k is in the tree & l is not in the tree.
- While including any edge ensure that it doesn't form a cycle.
- Time complexity: O(n2)

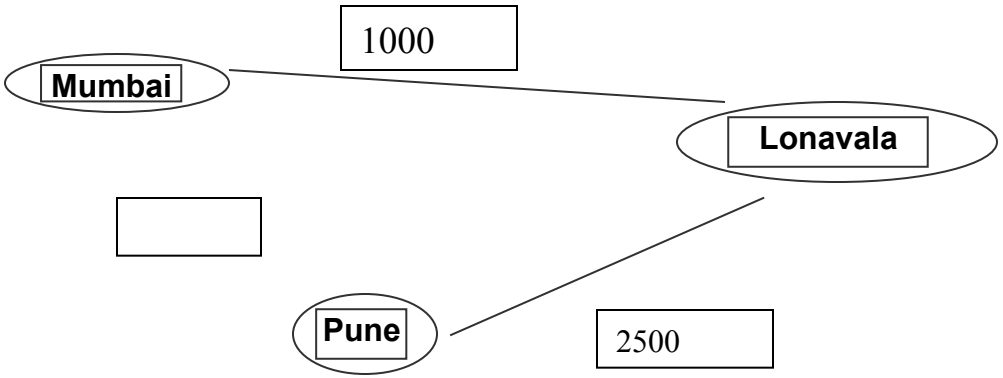
Prim's algorithm

- Find out minimum cost & spanning tree for following graph by applying Prim's algorithm



Prim's algorithm

- Spanning tree using Prim's algorithm is



- Total weight = 3500 This is also minimal spanning tree

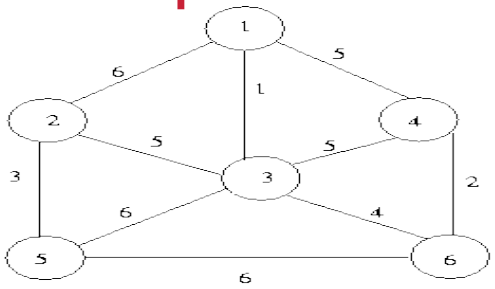
Kruskal's algorithm

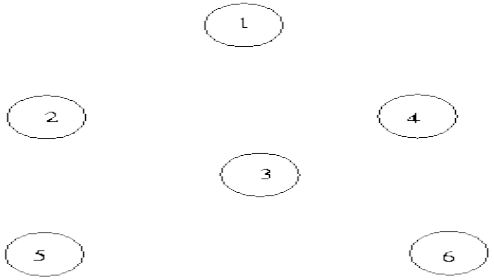
- It is another method for finding minimum cost spanning tree.
- Edges are added to the spanning tree in increasing order of cost.
- If the edge to added forms a cycle then it is discarded.
- Time complexity of algorithm:
  - O(e log e) + O(e log n)

Algorithm

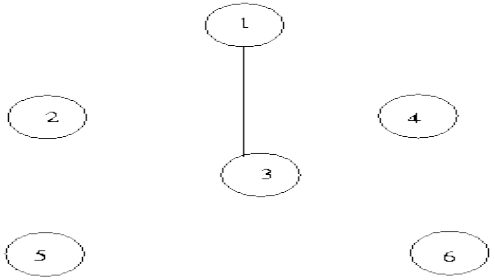
- create a forest *F* (a set of trees), where each vertex in the graph is a separate tree
- create a set *S* containing all the edges in the graph
- while *S* is nonempty
  - remove an edge with minimum weight from *S*
  - if that edge connects two different trees, then add it to the forest, combining two trees into a single tree
  - otherwise discard that edge
- At the termination of the algorithm, the forest has only one component and forms a minimum spanning tree of the graph.

Example

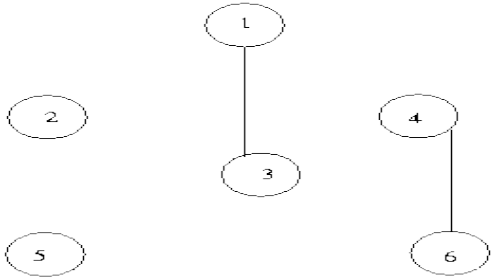




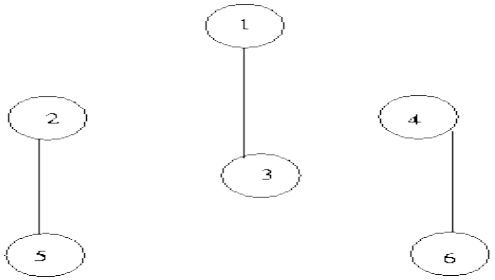
Initial Configuration



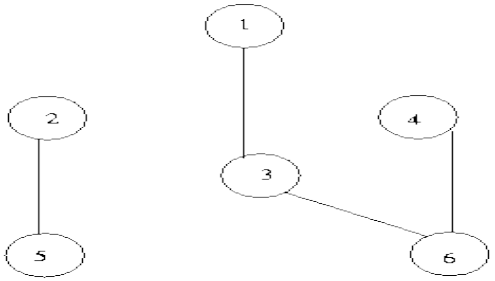
step1. choose (1,3)



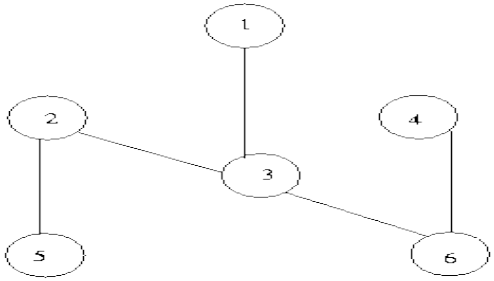
step2. choose (4,6)



step3. choose (2,5)



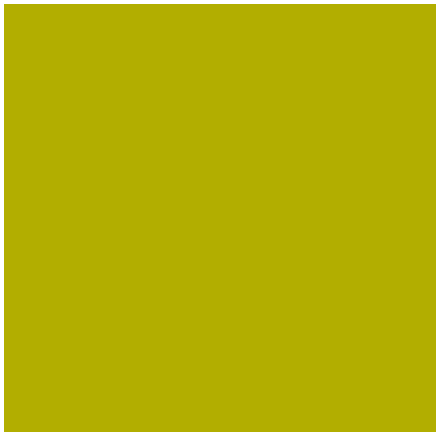
step4. choose (3,6)

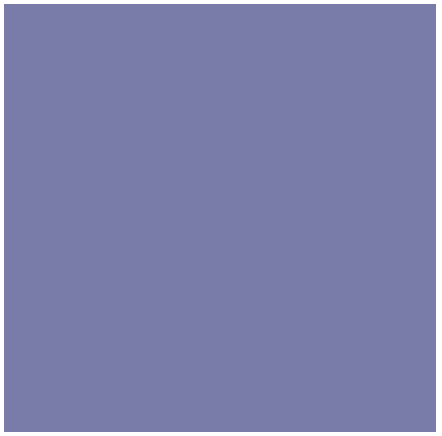


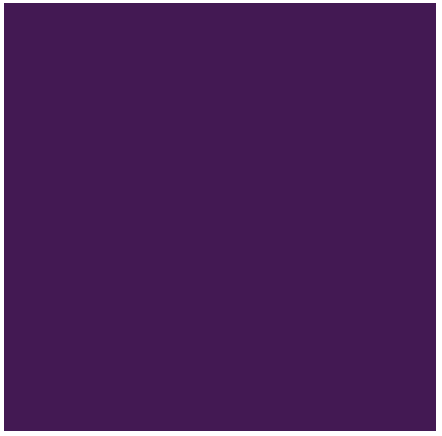
step5. choose (2,3)

bitCode technologies

Tables and Hashing







bitCode technologies

learn.innovate.share

Tables: rows & columns of information

- A *table* has several *fields* (types of information)
  - A telephone book may have fields **name**, **address**, **phone number**
  - A user account table may have fields **user id**, **password**, **home folder**
- To find an *entry* in the table, you only need know the contents of one of the fields (not all of them). This field is the *key*
  - In a telephone book, the key is usually **name**
  - In a user account table, the key is usually **user id**
- Ideally, a key *uniquely identifies* an entry
  - If the key is **name** and no two entries in the telephone book have the same name, the key uniquely identifies the entries

The Table ADT: operations

- **insert**: given a key and an entry, inserts the entry into the table
- **find**: given a key, finds the entry associated with the key
- **remove**: given a key, finds the entry associated with the key, *and* removes it

Also:

- **getIterator**: returns an iterator, which visits each of the entries one by one (the order may or may not be defined)

etc.

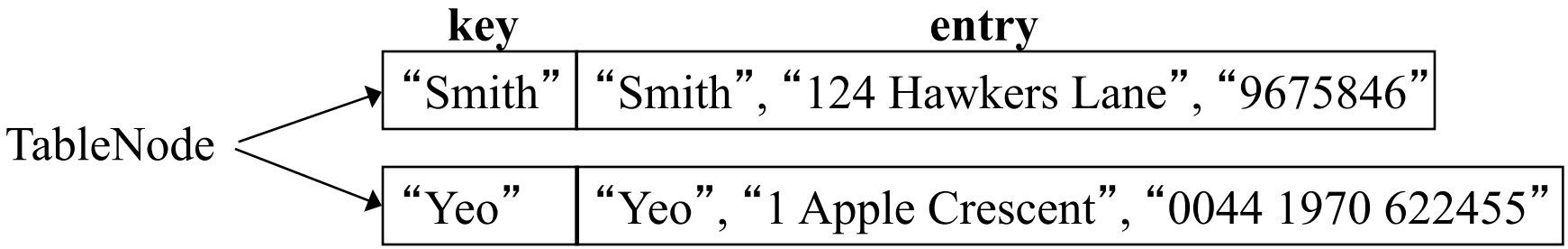
How should we implement a table?

Our choice of representation for the Table ADT depends on the answers to the following

- How often are entries inserted and removed?
- How many of the possible key values are likely to be used?
- Is the table small enough to fit into memory?
- How long will the table exist?

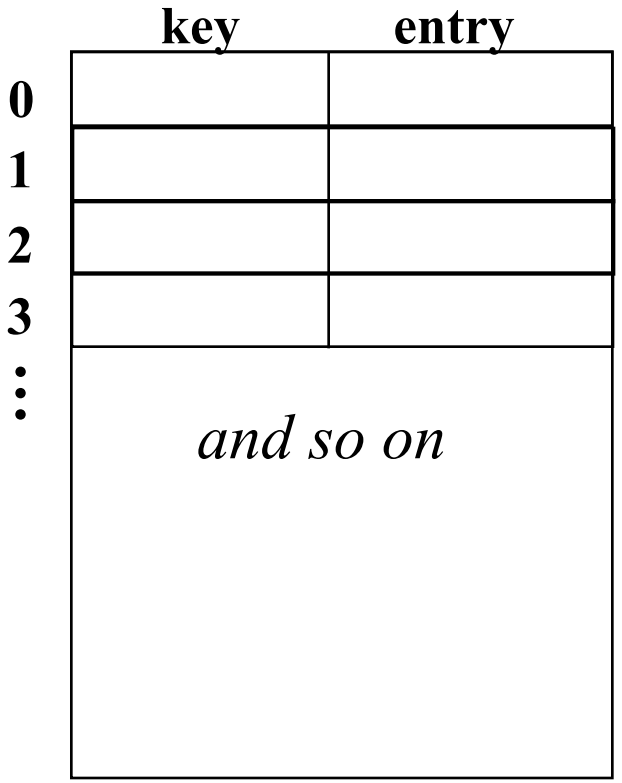
TableNode: a key and its entry

- For searching purposes, it is best to store the key and the entry separately (even though the key's value may be inside the entry)



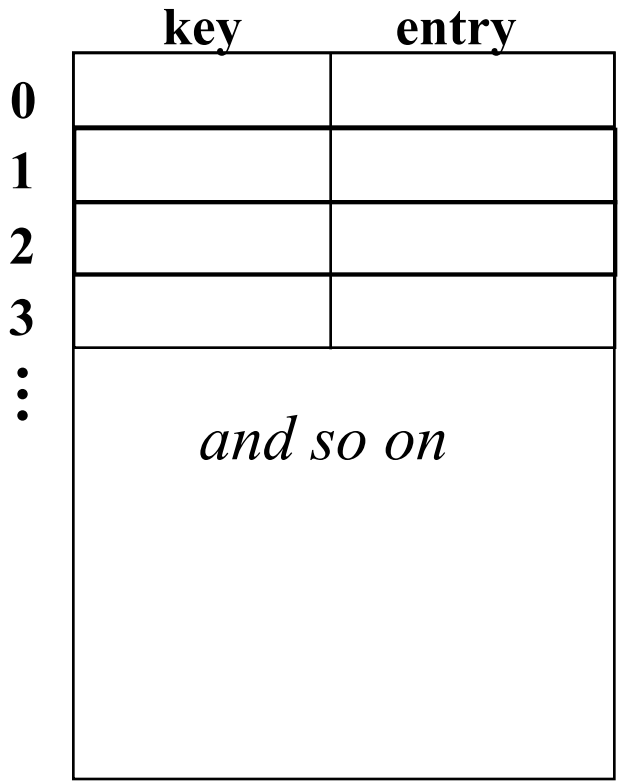
Implementation 1: unsorted sequential array

- An array in which TableNodes are stored consecutively in *any* order
- **insert**: add to back of array; O(1)
- **find**: search through the keys one at a time, potentially all of the keys; O(*n*)
- **remove**: find + replace removed node with last node; O(*n*)



Implementation 2:sorted sequential array

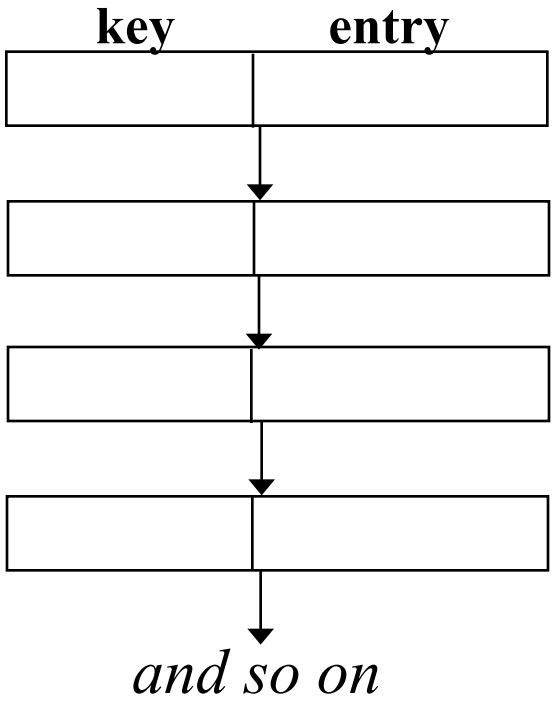
- An array in which TableNodes are stored consecutively, *sorted* by key
- **insert**: add in sorted order; O(*n*)
- **find**: binary chop; O(log *n*)
- **remove**: find, remove node and shuffle down; O(*n*)



We can use binary chop because the array elements are sorted

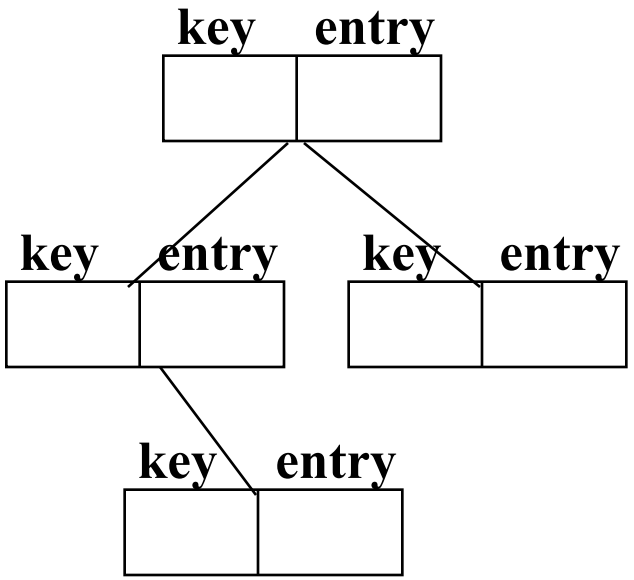
Implementation 3:linked list (unsorted or sorted)

- TableNodes are again stored consecutively
- **insert**: add to front; O(1)
  or O(n) for a sorted list
- **find**: search through potentially all the keys, one at a time; O(*n*)
  still O(n) for a sorted list
- **remove**: find, remove using pointer alterations; O(*n*)



Implementation 4: AVL tree

- An AVL tree, ordered by key
- **insert**: a standard insert; O(log *n*)
- **find**: a standard find (without removing, of course); O(log *n*)
- **remove**: a standard remove; O(log *n*)



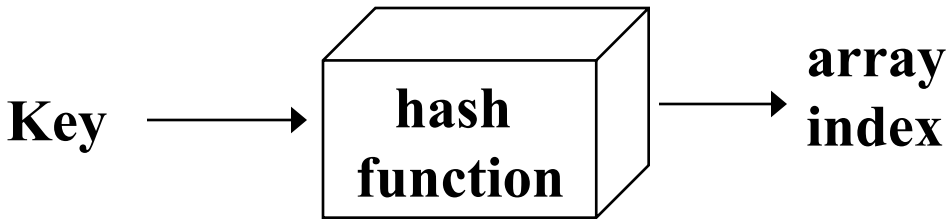
and so on

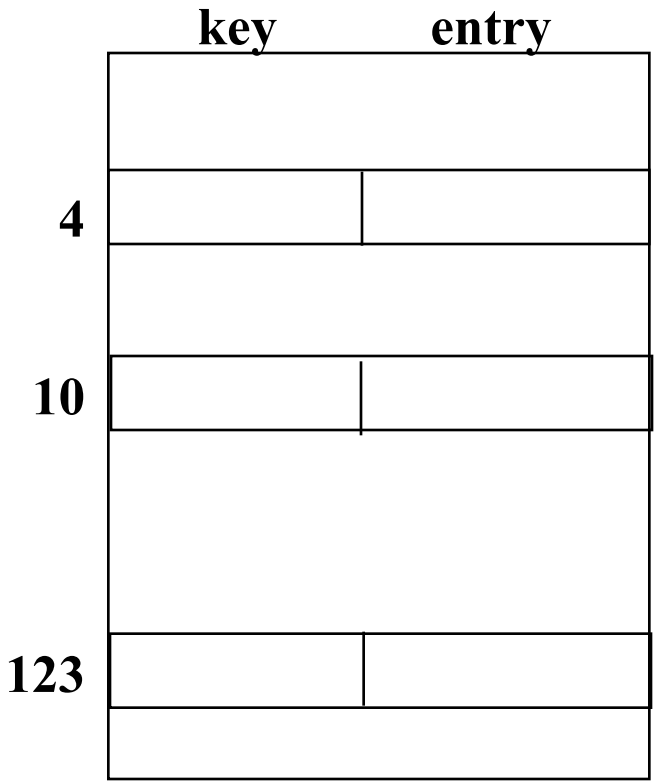
O(log *n*) is very good…

…but O(1) would be even better!

Implementation 5: *hashing*

- An array in which TableNodes are ***not*** stored consecutively - their place of storage is calculated using the key and a *hash function*
- *Hashed key*: the result of applying a hash function to a key
- Keys and entries are scattered throughout the array

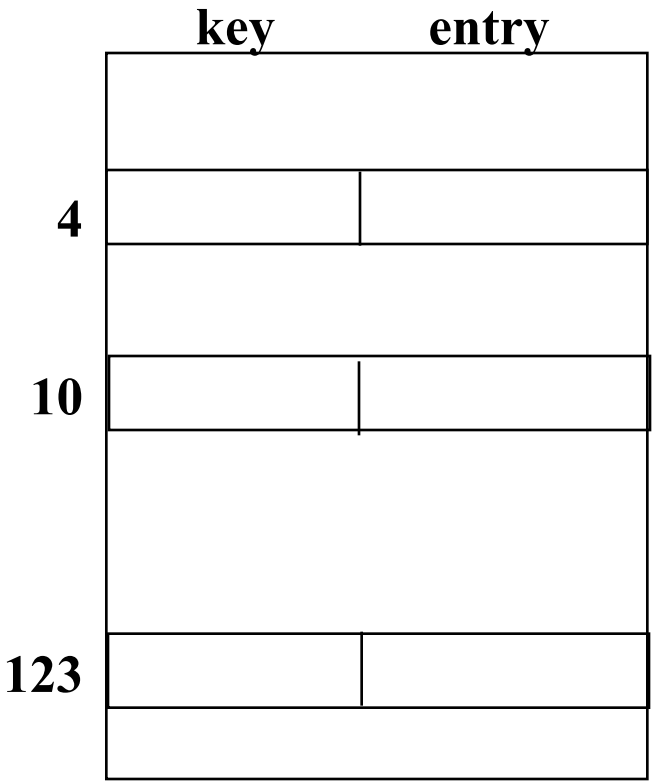




bitCode technologies

Implementation 5: *hashing*

- An array in which TableNodes are ***not*** stored consecutively - their place of storage is calculated using the key and a *hash function*
- **insert**: calculate place of storage, insert TableNode; O(1)
- **find**: calculate place of storage, retrieve entry; O(1)
- **remove**: calculate place of storage, set it to null; O(1)



All are O(1) !

bitCode technologies

Hashing example: a fruit shop

- 10 stock details, 10 table positions

- Stock numbers are between 0 and 1000

- **Use *hash function*: stock no. /100**
- What if we now insert stock no. 350? Position 3 is occupied: there is a *collision*
- *Collision resolution strategy*: insert in the next free position (*linear probing*)
- Given a stock number, we find stock by using the hash function again, and use the collision resolution strategy if necessary

	key	entry
0	85	85, apples
1		
2		
3	323	323, guava
4	462	462, pears
5	350	350, oranges
6		
7		
8		
9	912	912, papaya

bitCode technologies

Three factors affecting performance of hashing

- The hash function
  - Ideally, it should distribute keys and entries evenly throughout the table
  - It should minimise *collisions*, where the position given by the hash function is already occupied
- The collision resolution strategy
  - *Separate chaining*: chain together several keys/entries in each position
  - *Open addressing*: store the key/entry in a different position
- The size of the table
  - Too big will waste memory; too small will increase collisions and may eventually force *rehashing* (copying into a larger table)
  - Should be appropriate for the hash function used

Examples of hash functions (1)

- Truncation: If students have an 9-digit identification number, take the last 3 digits as the table position
  - e.g. 925371622 becomes 622
- Folding: Split a 9-digit number into three 3-digit numbers, and add them
  - e.g. 925371622 becomes 925 + 376 + 622 = 1923
- Modular arithmetic: If the table size is 1000, the first example always keeps within the table range, but the second example does not (it should be mod 1000)
  - e.g. 1923 mod 1000 = 923

Examples of hash functions (2)

- Using a telephone number as a key
  - The area code is not random, so will not spread the keys/entries evenly through the table (many collisions)
  - The last 3-digits are more random
- Using a name as a key
  - Use full name rather than surname (surname not particularly random)
  - Assign numbers to the characters (e.g. a = 1, b = 2; or use Unicode values)
  - Strategy 1: Add the resulting numbers. Bad for large table size.
  - Strategy 2: Call the number of possible characters *c* (e.g. *c* = 54 for alphabet in upper and lower case, plus space and hyphen). Then multiply each character in the name by increasing powers of *c*, and add together.

Choosing the table size to minimise collisions

- As the number of elements in the table increases, the likelihood of a *collision* increases - so make the table **as large as practical**
- If the table size is 100, and all the hashed keys are divisible by 10, there will be many collisions!
  - Particularly bad if table size is a power of a small integer such as 2 or 10
- More generally, collisions may be more frequent if:
  - greatest common divisor (hashed keys, table size) > 1
- Therefore, make the table size **a prime number** (gcd = 1)

Collisions may still happen, so we need a *collision resolution strategy*

Collision resolution: open addressing (1)

Probing: If the table position given by the hashed key is already occupied, increase the position by some amount, until an empty position is found

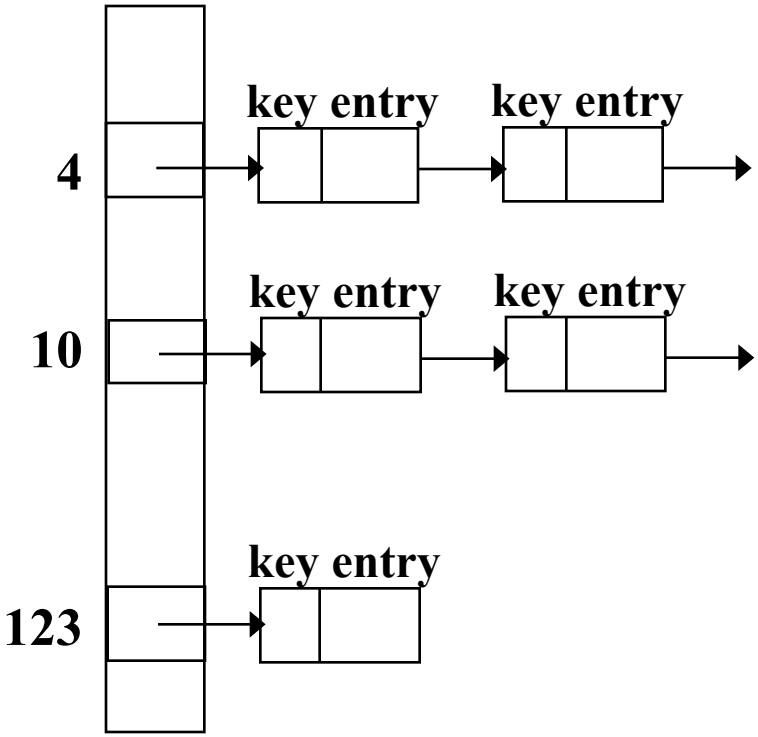
- **Linear probing:** increase by 1 each time
- **Quadratic probing:** to the original position, add 1, 4, 9, 16,…

Use the collision resolution strategy when inserting *and* when finding (ensure that the search key and the found keys match)

Collision resolution: chaining

- Each table position is a linked list
- Add the keys and entries anywhere in the list (front easiest)
- Advantages:
  - Simpler insertion and removal
  - Array size is not a limitation (but should still minimise collisions: make table size roughly equal to expected number of keys and entries)
- Disadvantage
  - Memory overhead is large if entries are small

No need to change position!



bitCode technologies

Applications of Hashing

- Compilers use hash tables to keep track of declared variables
- A hash table can be used for on-line spelling checkers — if misspelling detection (rather than correction) is important, an entire dictionary can be hashed and words checked in constant time
- Game playing programs use hash tables to store seen positions, thereby saving computation time if the position is encountered again
- Hash functions can be used to quickly check for inequality — if two elements hash to different values they must be different

When are other representations more suitable than hashing?

- Hash tables are very good if there is a need for many searches in a reasonably stable table
- Hash tables are not so good if there are many insertions and deletions, or if table traversals are needed — in this case, AVL trees are better
- Also, hashing is very slow for any operations which require the entries to be sorted
  - e.g. Find the minimum key

Searching Techniques

bitCode technologies

learn.innovate.share

Searching Techniques

- Sequential Search
- Indexed Sequential Search
- Binary Search
- Fibonacci Search

Sequential Search

- **sequential search**, also known as linear search is suitable for searching a set of data for a particular value.
- It operates by checking every element of a list one at a time in sequence until a match is found.
- If the data are distributed randomly, on average N/2 comparisons will be needed.
- The best case is that the value is equal to the first element tested, in which case only 1 comparison is needed.
- The worst case is that the value is not in the list (or is the last item in the list), in which case N comparisons are needed.

Sequential Search

- The simplicity of the linear search means that if just a few elements are to be searched it is less trouble than more complex methods that require preparation such as sorting the list to be searched or more complex data structures, especially when entries may be subject to frequent revision.
- Another possibility is when certain values are much more likely to be searched for than others and it can be arranged that such values will be amongst the first considered in the list.

Example

```
for( int I = 0; I < n; i++ )
{
    if( arr[ I ] == key )
        break;
}
If( I == n )
  return not found;
Else
  return found;
```

Indexed Sequential Search

- Indexed sequential search improves search efficiency for a sorted file, but it involves an increase in the amount of space required.
- A table, called an index, is set aside in addition to the sorted file itself.
- Each element in the index consists of a key kindex and a pointer to the record in the file that corresponds to the kindex.
- The elements in the index as well as the elements in the file, must be sorted on the key.
- If the index is one eighth the size of the file, every eighth record of the file is represented in the index.

Binary Search

- A **binary search algorithm** is a technique for finding a particular value in a linear array, by ruling out half of the data at each step.
- A binary search finds the median, makes a comparison to determine whether the desired value comes before or after it, and then searches the remaining half in the same manner.
- A binary search is an example of a divide and conquer algorithm.

Binary Search

- The entries in the table are sorted in increasing order.
- The approximate middle entry of the array is located, and it's key value is examined.
- If it's value is high, then the key value is compared with the middle half of the first half and the procedure is repeated on the first half until the required item is found.
- If the value is too low, then the key value is compared with the middle entry of second half and the procedure is repeated until the required item is found.
- The process continues until the desired key is found or the search interval becomes empty.

Example

```
BinarySearch( A, value ) {

    low = 0 high = N - 1

     p = low+((high-low)/2) //Initial probe position

     while ( low <= high) {

       if ( A[p] > value )
             high = p - 1
        else if ( A[p] < value )
          low = p + 1
             else
             return p

       p = low+((high-low)/2) //Next probe position.

 }

 return not_found
```

bitCode technologies

Fibonacci Search

- The Fibonacci sequence is

  0 1 1 2 3 5 8 13

- Here the index will be taken according to the fibonacci sequence and a range in which the num falls will be selected and searched.

- This searching technique needs sorted array.

Sorting Techniques

bitCode technologies

learn.innovate.share

Sorting

- Sorting is the operation of arranging the records/elements/keys in some sequential order according to an ordering criterion.
- Ordering criterion
- - Ascending order
- - Descending order
- Sorting data is preliminary step to searching.

Types of sorting

- Bubble sort
- Selection sort
- Insertion sort
- Merge sort
- Radix sort
- Quick sort

Algorithm for Bubble sorting

- two records are compared & interchanged immediately upon discovering that they are out of order
- This method will cause records with small keys to move or "bubble up"
- After the first pass, the record with the largest key will be in the nth position

Bubble sorting

Unsorted		Pass number(i)					Sorted
j	K_j	1	2	3	4	5	6
1	42	23	23	11	11	11	11
2	23	42	11	23	23	23	23
3	74	11	42	42	42	36	36
4	11	65	58	58	36	42	42
5	65	58	65	36	58	58	58
6	58	74	36	65	65	65	65
7	94	36	74	74	74	74	74
8	36	94	87	87	87	87	87
9	99	87	94	94	94	94	94
10	87	99	99	99	99	99	99

Contd.

- On each successive pass, the records with the next largest key will be placed in position n - 1, n - 2, . . ., 2, respectively, thereby resulting in a sorted table.

Contd.

```
/* Bubble sort for integers */

#define SWAP(a,b)   { int t; t=a; a=b; b=t; }

 /*State the advantage of using macro instead of function */

 void bubble( int a[], int n )    {

        int i, j,passes, comps;
        passes = n-1;
        comps = n-1;

        for(i=0;i<passes;i++) {
           for(j=0;j<(comps-i);j++) {

                if( a[j]>a[j+1] )
                   SWAP(a[j],a[j+1]);
           }
        }
    }
```

Rules for Selection sorting

- Select the highest one and put it where it belongs.
- Select the second highest and place it in order.
- Do this for all the keys to be sorted.

+ Contd.

Initial Order	Pass I	Pass II	Pass III	IV
50	50	40	20	
70	30	30	30	
20	20	20	40	
40	40	50	50	
30	70	70	70	

Algorithm for Insertion sorting

- This makes use of the fact the keys are partly ordered in the entire set of keys.
- One key from the unordered array is selected and compared with the keys which are ordered.
- The correct location of this key would be when a key greater than this particular key is found

Contd..

- The keys from that key onwards in the ordered array are shifted one position to the right.
- The hole developed is then filled in by the key from the unordered array.
- This is repeated for all keys in the unordered array

Original	34	8	64	51	32	21	Positions Moved
After p = 2	8	34	64	51	32	21	1
After p = 3	8	34	64	51	32	21	0
After p = 4	8	34	51	64	32	21	1
After p = 5	8	32	34	51	64	21	3
After p = 6	8	21	32	34	51	64	4

Merge Sort

- Merging is a process of combining two or more sorted arrays/files into third sorted array/file
- In merge sort the first elements in both tables are compared & the smallest key is then stored in a third table.
- This process is repeated till end of the both arrays
- In this way two sorted tables are merged to form third sorted table

- Table 1 : 11 23 42
- Table 2 : 9 25

- We obtain the following trace:
- Table 1 11 23 42
- Table 2 25
- New Table 9

- Table I 23 42
- Table 2 25
- New Table 9 11

Quick Sort terminology

- Pivot :The key whose exact location is to be found in the sorted array.
- Keys to the left of pivot are smaller than the pivot and keys to the right of pivot are greater than the pivot.
- This is known as partitioning.

Quick Sort(A, lb,up)

If (lb >= ub) then return

If lb < ub then pivot_loc = partition (A, lb,ub)

Quick_sort(A, lb, pivot_loc)

Quick_sort(A, pivot_loc+1, ub)

Partition algorithm

- Let down = lower bound of array and up = upper bound of array, pivot = A[lb]
- Repeatedly increase the pointer down by one position while A[down] < pivot
- Repeatedly decrease pointer up by one position while A[up] >= pivot
- If up > down then interchange A[down] with A[up]
- If down < up goto 2
- Else interchange pivot, A[up]
- Pivot_loc = up
- Return pivot_loc

Quick Sort

Line No.	Key[1]	Key[2]	Key[3]	Key[4]	Key[5]	Key[6]	Key[7]	Key[8]	Key[9]	key[10]
										←
1	15	20	5	8	95	12	80	17	9	55
		→						←		
2	9	20	5	8	95	12	80	17	()	55
3	9	( )	5	8	95	12	80	17	20	55
			→							
4	9	12	5	8	95	( )	80	17	20	55
					←					
5	9	12	5	8	( )	95	80	17	20	55
6	9	12	5	8	15	95	80	17	20	55

bitCode technologies

Quick Sort

- Steps Involved

1. Remove the first data item, 15 as the pivot, mark it's position, scan the array from right to left, comparing the data item value 15. When you find the first smaller value remove it from it's current position and put it in position key[i] (Shown in line 2).

2. Scan line 2 from left to right beginning with position key[2], comparing value 15. when you find the first value greater than 15, store it in position marked by parenthesis (shown in line no 4).

3. Begin the right to left scan of line 3 with position key[8] looking for a value smaller than 15. When found store it in position marked by parenthesis.

4. Begin scanning line 4 from left to right at position key[3]. Find a value greater than 15, remove it, mark it's position, store it inside parenthesis in line 4. (shown in line 5).

5. Now when you attempt to scan line 5 from right to left beginning at position key[5], you are immediately at a parenthesized position determined by the previous left to right scan. This is the position to put the pivot data item 15. At this stage 15 is in correct position relative to the final sorted array.

Radix Sort

- In radix sort method the given set of unsorted numbers are compared on the basis of columns (digit place like units tens hundreds etc).
- Each comparison through the entire set is termed as Pass.
- The number of passes required to sort the given set of numbers depends on the number of digits of the largest number
- After each pass the number are placed in the respective pockets.

- 42, 23, 74, 11, 65, 57, 94, 36, 99, 87, 70, 81, 61

After the first pass on the unit digit position of each number we have:

		61								
		81			94			87		
	70	11	42	23	74	65	36	57		99
poc	0	1	2	3	4	5	6	7	8	9

- combining the contents of the pockets so that the contents of the "0" pocket are on the bottom and the contents of the "9" pocket are on the top, we obtain:

- 70, 11, 81, 61, 42, 23, 74, 94, 65, 36, 57, 87, 99

							61	70	81	94
		11	23	36	42	57	65	74	87	99
Poc	0	1	2	3	4	5	6	7	8	9

Comparison of Sorting methods

Algorithm	Average	Worst Case	
SELECTION	$n^2/4$	$n^2/4$	
BUBBLE	$n^2/4$	$n^2/2$	
MERGE	$0(n\log_2 n)$	$0(n\log_2 n)$	
QUICK	$0(n\log_2 n)$	$n^2/2$	
RADIX	0(m+n)	0(m+n)	

Complexity of Algorithm

- In general the *complexity* of an algorithm is

the amount of time and space (memory use) required to execute it.

- Since the actual time required to execute an algorithm depends on the details of the program implementing the algorithm and the speed and other characteristics of the machine executing it, it is in general impossible to make an estimation in actual physical time,

- however it is possible to measure the length of the computation in other ways, say by the number of operations performed.

- the following loop performs the statement x := x + 1 exactly *n* times,

```
for i := 1 to n do
    x := x + 1
```

- The following double loop performs it *n*2 times:

```
for i := 1 to n do
for j := 1 to n do
x := x + 1
```

- The following one performs it $1 + 2 + 3 + \cdots + n = n(n + 1)/2$ times:

```
for i := 1 to n do
    for j := 1 to i do
        x := x + 1
```

- Since the time that takes to execute an algorithm usually depends on the input, its complexity must be expressed as a function of the input, or more generally as a function of the *size* of the input.
- Since the execution time may be different for inputs of the same size, we define the following kinds of times:
- Best Case Time
- Worst Case Time
- Average Case Time

- 1. *Best-case time*: minimum time needed to execute the algorithm among all inputs of a given size *n*.
- 2. *Worst-case time*: maximum time needed to execute the algorithm among all inputs of a given size *n*.
- 3. *Average-case time*: average time needed to execute the algorithm among all inputs of a given size *n*.

Example

- For instance, assume that we have a list of n objects one of which is

  colored red and the others are colored blue, and we want to find the one that is colored red by examining the objects one by one. We measure time by the number of objects examined. In this problem the minimum time needed to find the red object would be 1 (in the lucky event that the first object examined turned out to be the red one). The maximum time would be n (if the red object turns out to be the last one).

- The average time is the average of all possible times: 1, 2, 3, . . . , n, which is $(1+2+3+\cdots+n)/n = (n+1)/2$.

- So in this example the best-case time is 1, the worst-case time is n and the average-case time is $(n + 1)/2$.

Complexity for Bubble Sort

- Since bubble sort is just a double loop its inner loop is executed

$(n - 1) + (n - 2) + \cdots + 1 = n(n - 1)/2$ times,

so it requires $n\ (n - 1)/2$ comparisons and possible swap operations.

- Hence its execution time is O(n2).