Data Structure:

Array - continuous memory allocation.

Types

1. linear Data Structure 2.Non-linear Data Structure

(sequential) (Non-sequential)

1.Array

2.List - Single linked list 1.Trees

- Double linked list 2.graph

- Circular linked list

3.Stack

4.Queue

Bubble Sort:

->sorting algorithm based on swapping.

->compare adjacent elements.

->arrange elements in ascending order array.

Ex: I/p: 15,10,23,13,25.

O/p: 10,13,15,23,25

A=10, b=20

Third variable/temp

T=a; t=10

A=b; a=20

B=t; b=10

Program:

import java.util.Arrays;

public class BubbleSort {

public static void main(String[] args) {

int a[]= {10,5,84,94,39,6,4,32};

System.out.println(Arrays.toString(a));

for(int i=0;i<a.length-1;i++) {

for(int j=0;j<a.length-1-i;j++) {

if(a[j]>a[j+1]) {

int t=a[j];

a[j]=a[j+1];

a[j+1]=t;

}

}

}

System.out.println(Arrays.toString(a));

}

}

Linear Search:

->Searching algorithm.

->used with array.

->finding location of element.

Order(n)

0 1 2 3 4 5

Eg: a[]={1,2,3,4,5,6}

I/p: key=3, 1==key

2==key

3==key

Break;

Disadvantages:

1. …….to 1000 elements array

Program:

import java.util.Scanner;

public class LinearSearch {

public static void main(String[] args) {

//0 1 2 3 4 5

int a[]= {10,30,22,19,63,73};

Scanner sc=new Scanner(System.in);

System.out.println("Enter Key : ");

int key=sc.nextInt();

int flag=0;

for(int i=0;i<a.length;i++) {

if(a[i]==key) {

System.out.println("Found at "+i);

flag=1;

break;

}

}

if(flag==0) {

System.out.println("Element not found");

}

}

}

Binary Search:

-> finding location of an elements.

->Divide and conquer.

->Array should be sorted order.

Order(log n)

Ex: 0 1 2 3 4 5 ………….. 10000

Int a[]=(10,20,30,40,50,60}

Program:

import java.util.Scanner;

public class BinarySearch {

public static void main(String[] args) {

// array must be sorted order

Scanner sc = new Scanner(System.in);

int a[] = { 10, 20, 30, 40, 50, 60 };

System.out.println("Enter key element: ");

int key = sc.nextInt();

int lb = 0;

int ub = a.length - 1;

int flag = 0;

while (lb < -ub) {

int mid = (lb / ub) / 2;

if (a[mid] == key) {

System.out.println("Found at " + mid);

flag = 1;

break;

} else if (a[mid] < key) {

lb = mid + 1;

} else {

ub = mid - 1;

}

}

if (flag == 0) {

System.out.println("Element not Found");

}

}

}

Linked List:

Single Linked List:

->fixed size (array size should be known in advance).

->Insertion and deletion operation costlier.

Structure:

Data next



10 2000 20 3000 30 null



1000 2000 3000



Head node

Ex: creation and display, Insertion at Beginning ,End ,Specified Position, Deletion at Beginning ,End ,Specified Position.

class Node{

int data;// default 0

Node next;// default null

public Node(int data) {

this.data=data;

}

}

class LinkedList{

Node head,tail;

public void insert(int data) {

Node n=new Node(data);

if(head==null) {

head=n;

tail=n;

}else {

tail.next=n;

tail=n;

}

}

public void display() {

Node temp=head;

while(temp!=null) {

System.out.println(temp.data);

temp=temp.next;

}

}

public void insertBeg(int data) {

Node n=new Node(data);

n.next=head;

head=n;

}

public void insertAt(int pos,int data) {

Node n=new Node(data);

Node temp=head;

for(int i=0;i<pos-1;i++) {

temp=temp.next;

n.next=temp.next;

temp.next=n;

}

}

public void deleteBeg() {

Node temp=head;

head =head.next;

temp.next=null;

//or

// head =head.next;

}

public void deleteEnd() {

Node temp=head;

while(temp.next!=tail) {

temp =temp.next;

}

temp.next=null;

tail=temp;

}

public void deleteAt(int pos) {

Node temp=head;

for(int i=0;i<pos-1;i++) {

temp=temp.next;

}

temp.next=temp.next.next;

}

}

public class SingleLinkedList {

public static void main(String[] args) {

LinkedList l=new LinkedList();

l.insert(10);

l.insert(20);

l.insert(30);

l.insertBeg(50);//beginning

l.insertAt(2, 15);

l.display();

System.out.println("after delete begin");

l.deleteBeg();

l.display();

System.out.println("after delete end");

l.deleteEnd();

l.display();

System.out.println("after delete postion");

l.deleteAt(2);

l.display();

}

}

Stack :

->Stack is a list which follows LIFO (last in first out).

->Insertion and deletion of elements can be done at only top end.

->Major operations:

1. push(Int data)- adding
2. Pop() - it always pop out the top element.(Remove)
3. Peek()-print top
4. Display()-top to zero

Ex:

class Stack{

int a[]=new int[5];

int top=-1;

public void push(int data) {

if(top==a.length-1) {

System.out.println("stack is full");

}else {

top++;

a[top]=data;

}

}

public void pop() {

if(top==-1) {

System.out.println("stack empty");

}else {

System.out.println("popped element = "+a[top]);

top--;

}

}

public void peek() {

if(top==-1) {

System.out.println("stack empty");

}else {

System.out.println("peek element = "+a[top]);

}

}

public void display() {

if(top==-1) {

System.out.println("stack empty");

}else {

for(int i=top;i>=0;i--) {

System.out.println(a[i]);

}

}

}

}

public class StackDemo {

public static void main(String[] args) {

Stack s=new Stack();

s.push(10);

s.push(20);

s.push(30);

s.peek();

s.pop();

s.display();

}

}

Application of Stack:

1. only one stack is used (operand stack)
2. If the character is operand then push it into stack.
3. If the character is operator , then pop top two operands from the stack , perform calculation and push the result back into stack

->first pop - second arugment

->second pop - first arugment

1. After reading all characters from the postfix expression ,stack will be having only one value which is the result.

Infix Expression:

<operand1><operator><Operand2>

Ex: a+b

1. 546+\*/493/+\*- ans= 350



5 4 6 4+6=10 10 9/3=3



1. 4 5 5\*10=50 50 4+3=7



5 50\*7=350

Postfix Expression:

<operand1><operand2><operator>

Ex: ab+

Prefix Expression:

<operator><operand1><operand2>

Ex: +ab

INFIX to POSTFIX conversion:

1. if the character is left paranthesis , push to the stack
2. if the character is an OPERAND,ADD to the POSTFIX EXPRESSION
3. If the character is an operator , check whether stack is empty

->if the stack is empty ,push operator into stack

->if not empty, check the priority of the operator.

1. if the priority of operator > operator present at top of stack, then push operator into stack.
2. If the priority of operator <= operator present at top of the stack, then pop the operator from stack and add to postfix expression and GOTO step (a)
3. if the character is right paranthesis , then pop all the operators from the stack until it reaches left paranthesis and add to postfix expression.
4. After reading all characters, if Stack is not empty, the pop and Add to Postfix expression.

Ex: a-(b\*c-d)/e

character stack postfix

1. a a



1. - a



1. ( a



1. B ab



1. \* ab



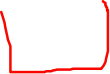
1. C abc



1. - abc\*



1. D abc\*d



1. ) abc\*d-



1. / abc\*d-



1. E abc\*d-e



Ans:abc\*d-e/-

Ex:infix- (a/(b-c+d))\*(e-a)\*c - postfix- abc-d+ea-\*c\*

Balanced Paranthesis/symbols:( (, [, {, < chevrons)

1. read all characters one by one.
2. If it is open symbol (,[,{,< push it into the stack.
3. If it is close symbol

If stack is empty then UNBALANCED

Else

Check it with the top of stack , if it matches,pop it from stack else UNBALANCED

1. If stack is empty -->BALANCED

If not empty -->UNBALANCED

<>,{},[],()

EX:1. [()] - balanced

1. [(]) - unbalanced.

Queue (FIFO) data structure (Array implementation):

-> queue is a list/array which follows FIFO (first in first out)

-> Insertion will be done at rear end

-> Deletion will be done at front end

-> major operations/functions

1. enqueue(int data)
2. Dequeue()
3. Peek() - beginning of element a[front].



1. Display() -

Front rear



a=10,20,30,40,50

->for enqueue/insertion

Rear++;

A[rear]=data;

->queue elements are from front front to rear end.

->for dequeue /deletion -> front++;

Ex:

class Queue {

int a[] = new int[5];

int r = -1;

int f = -1;

public void enqueue(int data) {

if (r == a.length - 1) {

System.out.println("stack is full");

} else if (r == -1 && f == -1) {

r = 0;

f = 0;

a[r] = data;

} else {

r++;

a[r] = data;

}

}

public void display() {

if (r == -1 && f == -1) {

System.out.println("queue empty");

} else {

for (int i = f; i <= r; i++) {

System.out.println(a[i]);

}

}

}

public void dequeue() {

if (r == -1 && f == -1) {

System.out.println("queue empty");

} else if (r == f) {

System.out.println("popped element = " + a[f]);

r = -1;

f = -1;

} else {

System.out.println("popped element = " + a[f]);

f++;

}

}

public void peek() {

System.out.println("peek = " + a[f]);

}

}

public class QueueDemo {

public static void main(String[] args) {

Queue q = new Queue();

q.enqueue(10);

q.enqueue(20);

q.enqueue(30);

q.enqueue(40);

q.enqueue(50);

q.dequeue();// remove

q.display();

q.peek();

q.enqueue(60);

}

}

Tree Data Structure - basic terminology:

Data is organised in a hierarchical fashion without any closed region(cycle).

Eg: A cycle



B C elements ->Node



Edge

1. Node (element)
2. Root Node - Node which has no parent starting node ‘A’
3. Edge - n ->node , n-1 edges
4. Siblings - Nodes which have same parent B,C
5. Leaf Node - Node has no children



A



B c



D E F



G H



1. Internal Node - every node other than leaf node.
2. Degree - No of child nodes represents degree.

Deg(B) - 3 Deg(A) - 2

Deg(leaf node) = 0

Degree(tree) = Maximum degree among all nodes

= deg(B) = 3

Subtree - nodes with children forms subtree.

Height - longest path from leaf node to that node

H(E)=1;

H(leaf node)=0 ; h(a)=3; h(b)=2;

Depth - longest path from root node to that node

D(root node)=0;

D(d)=2; d(e)=2; d(h)=3;

Binary Tree:

Every node in a tree should atmost 2 children (0,1,2)



1. A 2. A 3. A



B C B C

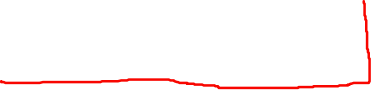


Tree Binary D

Representation -> Linked List (Double).



Node prev data next

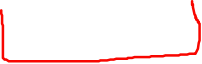


Node reference of data reference of

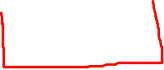
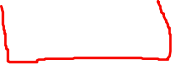
Left child 10 right child



R(20) R(30)



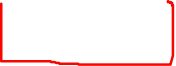
Reference of B A reference of C



Reference B Null Null C Null



Of D



Null D Null

Tree Traversals - inorder, preorder, postorder.(visiting all nodes)



1. Inorder - left root right - BAC A Root
2. Preorder - root left right -ABC



1. Postorder - left right root-BC Left B C Right



Recursive A



B C



D E F G

Inorder - DBEAFCG

Preorder - ABDECFG

Postorder - DEBFGCA

Ex:Binary Tree implementation:

class TNode{

int data;

TNode left; 1



TNode right; 2 3



TNode(int data){ 4 5

this.data=data;

}

}

class BinaryTrees{

TNode root;

BinaryTrees(int data){

root=new TNode(data);

}

public void insertLeft(TNode root,int data) {

root.left=new TNode(data);

}

public void insertRight(TNode root,int data) {

root.right=new TNode(data);

}

public void inorder(TNode root) {

if(root==null) {

return;

}else {

inorder(root.left);

System.out.print(root.data+" ");

inorder(root.right);

}

}

}

public class BinaryTreeImplementation {

public static void main(String[] args) {

BinaryTrees bt=new BinaryTrees(1);

bt.insertLeft(bt.root,2);

bt.insertRight(bt.root , 3);

bt.insertLeft(bt.root.left , 4);

bt.insertRight(bt.root.left , 5);

//TRaversal (inorder)

bt.inorder(bt.root);

}

}



Binary Search Tree:

Condition: elements must be in sorted order.



10 20 30 40 50



left mid right



Left sub tree Root right sub tree

Binary Search + Binary tree :

1. all the elements of left subtree must be less than root.
2. Elements of right subtree must be greater than root.



Operations:

1. Insert
2. Delete
3. Search/Find
4. Findmin()
5. Findmax()



8



1. 10



1. 6 14



1. 7 13

->All the elements of left subtree must be less than root.

->elements of right subtree must be greater than root.

Binary Search Tree - insertion



1. if tree has no element then consider that element as root node.
2. If the element is greater than root,insert into right subtree.
3. If the element is less than root,insert into left sub tree.

Ex: 50,30,10,60,80,20,70,55,35,5



50



1. 60



1. 35 55 80



5 20 70

BinarySearchTree - deleting a node.

Three cases:

1. Node having no children.(leaf)
2. Node having one children.
3. Node having two children.

->No children simply remove leaf node.

->one children - child will replace position of parent.

Child:

->min of right subtree

->max of left subtree

After deletion also tree should follow bst property

Ex:

class TreeNode {

int data;

TreeNode left;

TreeNode right;

public TreeNode(int data) {

this.data = data;

}

}

class BST {

TreeNode root;

public BST(int data) {

root = new TreeNode(data);

}

public void insert(int data) {

insertNode(root, data);

}

public TreeNode insertNode(TreeNode root, int data) {

if (root == null) {

return new TreeNode(data);

}

if (data < root.data) {

root.left = insertNode(root.left, data);

} else {

root.right = insertNode(root.right, data);

}

return root;

}

public void inOrder(TreeNode root) {

if (root == null) {

return;

} else {

inOrder(root.left);

System.out.print(root.data + " ");

inOrder(root.right);

}

}

public void searchNode(TreeNode root, int data) {

if (root == null) {

System.out.println("element not found");

return;

}

if (data == root.data) {

System.out.println("found");

return;

} else if (data < root.data) {

searchNode(root.left, data);

} else {

searchNode(root.right, data);

}

}

public int findMinimum(TreeNode root) {

if (root == null) {

System.out.println("no tree exists");

} else if (root.left == null) {

return root.data;

}

return findMinimum(root.left);

}

public int findMaximum(TreeNode root) {

if (root == null) {

System.out.println("no tree exists");

} else if (root.right == null) {

return root.data;

}

return findMaximum(root.right);

}

public TreeNode deleteNode(TreeNode root,int value) {

if(root==null) {

System.out.println("No tree exists");

return root;

}

if(value<root.data) {

root.left=deleteNode(root.left, value);

}else if(value>root.data){

root.right=deleteNode(root.right, value);

}else {

if(root.right==null) {

return root.left;

}else if(root.left==null) {

return root.right;

}

root.data=findMinimum(root.right);

root.right=deleteNode(root.right, root.data);

}

return root;

}

}

public class BinarySearchTree {

public static void main(String[] args) {

// 50,30,10,60,55

BST bst = new BST(50);

bst.insert(30);

bst.insert(10);

bst.insert(60);

bst.insert(55);

bst.inOrder(bst.root);

// searching

bst.searchNode(bst.root, 60);

// find min

System.out.println(bst.findMinimum(bst.root));

// find max

System.out.println(bst.findMaximum(bst.root));

bst.deleteNode(bst.root, 10);

bst.inOrder(bst.root);

}

}

Graph:

->Graph is a set of (V,E) where,

V->set of vertices/Node A



->represented by circle. B C



1. >set of edges

->a line connecting two vertices. A,B; A,C; B,C

Terminology:

1. Node(Vertices) Two nodes are adjacent if they connected by an edge
2. Edge
3. Adjacent Nodes
4. Degree of a Node - No of edges connected to that node
5. Size of a graph - total no of edges in a graph
6. Path - sequence of vertices from source Node to Destination Node. EX: 0-1-2-4

0-3-4



Adjacent 0 1 2



nodes of



3 4



Deg(3)=3;

Types of Graph:

1. Directed



B



A C



D



1. Undirected



A



B C



1. Weighted - Directed as well as undirected



1. Cyclic

Weighted - every edge will have cost/weight



Ex: 2 B 3



A 9 C B-D => 1.B-D(9)



5 D 4 2.B-C-D = 3+4=7 Path way



Cyclic:



B



A C



D



A-B-C-D-A

1. Adjacency Matrix(2-D Array)

1. 1 2



space - order(n square)



For(I=0){



3 4 for(j=0){}}



Adjacency matrix = 0 1 2 3 4



0 0 1 0 1 0

1 1 0 1 0 1

2 0 1 0 0 1

3 1 1 0 0 1

4 0 0 1 1 0



1. Adjacency List

->Array list - one list another list inside anthor list.



1. 1 2



3 4



0 1 3



1 0 2 3 order(n+2e)



2 1 4 2e mean - 0 to 1



3 0 1 4 1 to 0



4 2 3



Ex: graph implementation

import java.util.ArrayList;

class MyGraph{

ArrayList<ArrayList<Integer>> list=new ArrayList<>();

public MyGraph(int v) {

for(int i=0;i<v;i++) {

list.add(new ArrayList<Integer>());

}

}

public void addEdge(int u,int v) {

list.get(u).add(v);

list.get(v).add(u);

}

public void display() {

for(int i=0;i<list.size();i++) {

System.out.println("Elements in list "+0);

for(int j=0;j<list.get(i).size();j++) {

System.out.println(list.get(i).get(j));

}

}

}

}

public class GraphRepresentations {

public static void main(String[] args) {

MyGraph g=new MyGraph(5);

g.addEdge(0, 1);

g.addEdge(0, 3);

g.addEdge(1, 2);

g.addEdge(1, 3);

g.addEdge(2, 4);

g.addEdge(3, 4);

g.display();

}

}

Graph Traversals:

-> visiting all node of a graph only once

1. BFS(Queue) -Breadth first search
2. DFS(Stack) - Depth First Search

Ex:

import java.util.ArrayList;

import java.util.Queue;

import java.util.Stack;

import java.util.LinkedList;

class MyGraphs {

ArrayList<ArrayList<Integer>> list = new ArrayList<>();

public MyGraphs(int v) {

for (int i = 0; i < v; i++) {

list.add(new ArrayList<Integer>());

}

}

public void addEdge(int u, int v) {

list.get(u).add(v);

list.get(v).add(u);

}

public void display() {

for (int i = 0; i < list.size(); i++) {

System.out.println("Elements in list " + 0);

for (int j = 0; j < list.get(i).size(); j++) {

System.out.println(list.get(i).get(j));

}

}

}

public void bfs(int v) {

Queue<Integer> q = new LinkedList();

boolean visited[] = new boolean[list.size()];

q.add(v);

visited[v] = true;

System.out.println("Breadth first search");

while (q.size() != 0) {

int vertex = q.remove();

System.out.print(vertex + " ");

for (int i = 0; i < list.get(vertex).size(); i++) {

int av = list.get(vertex).get(i);

if (!visited[av]) {

q.add(av);

visited[av] = true;

}

}

}

}

public void dfs(int v) {//recursion

Stack<Integer> s = new Stack<>();

boolean visited[] = new boolean[list.size()];

s.add(v);

visited[v] = true;

System.out.println("Depth First Search");

while (s.size() != 0) {

int vertex = s.pop();

System.out.print(+vertex + " ");

for (int i = 0; i < list.get(vertex).size(); i++) {

int av = list.get(vertex).get(i);

if (!visited[av]) {

s.add(av);

visited[av] = true;

}

}

}

}

}

public class GraphTraversals {

public static void main(String[] args) {

MyGraphs g = new MyGraphs(5);

g.addEdge(0, 1);

g.addEdge(0, 3);

g.addEdge(1, 2);

g.addEdge(1, 3);

g.addEdge(2, 4);

g.addEdge(3, 4);

// g.display();

g.bfs(0);

System.out.println();

g.dfs(0);

}

}

Minimum cost spanning tree:



tree A A tree



Graph graph



B C B C



->tree shouldn’t contain cycles(loop)

Spanning tree:

->subgraph of a graph G(V,E) but without having any cycle.

->Spanning tree should contain all vertices of a graph

->if graph has ‘n’ vertices , spanning tree must have n-1 edge

->no cycle



5 A A A A



2 5 2 5 2



B C B C B C B C



3 3 3

Cost=7 cost=8 cost=5



Minimum cost

Prim’s Algorithm:

->start with any vertex of the graph

->select the minimum cost edge and add that edge to the graph if it is not forming any cycle, if cycle forms then discarded

->repeat step 2 till all vertices are covered.

1. Vertices.
2. N-vertices , (n-1)edges.
3. no cycle form.

7 11



A B E



3 4



9



C D



10

Ex: Vertix A:

1. B(7) -(cycle)



1. c(3) A B E



C-B(4)



C-D(10)(cycle) C D



1. D(9)



B-E(11)



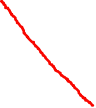
Kruskal’s Algorium:

1. Write all edges in ascending order based upto its cost.
2. Select minimum cost edge from the edge list and add it to the spanning tree if it’s not forming cycle,if forming cycle means discard that edge
3. Repeat step 2 until all vertices are covered.
4. n-vertices(6)
5. N-vertices, n-1 edge(5)
6. No cycle

3



2 B D 3



A 5 4 2 F



3 C E 5



4



B D



A F



C E

Dijkstra’s Algorithm:

->single source shortest path algorithm

1. Assume any vertex as source vertex
2. Find shortest path from source vertex to all remaining vertices

If(d[u]+c(u,v)<d[v])

d[v]=d[u]+c(u,v);

30



1 6 35



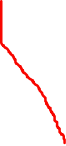
10 20



4 7 1 2 =10



2 12 3 =30



15 7 4 =45



20 3 5 5 =35



5 6 =30



7 =42



Selected vertex visited set d[2] d[3] d[4] d[5] d[6] d[7]

1. {1} 10 - - - 30 -
2. {1,2} 10 30 - - 30 -
3. {1,2,3} 10 30 45 35 30 -

6 {1,2,3,6} 10 30 45 35 30 65

5 {1,2,3,6,5} 10 30 45 35 30 42

4 {1,2,3,6,5,4} 10 30 45 35 30 42