ADVANCED DATA STRUCTURES THROUGH JAVA

JAVA 2.0 — BOOK

Herbert Schildt — Author

UNIT — I : Generics } Herbert Schildt

UNIT — II : Collection Framework

UNIT — III : Dictionaries implementation

Sorted chain

Hash tables

Binary Search Tree (BST)

UNIT — IV : Dictionary implementation

AVL Trees

Red Black Trees

B-Trees

UNIT — V : Pattern matching & Trees

Priority Queues

$\boxed{\text{UNIT — 3,4,5}}$: Data structures using JAVA

L Sartaj Sahni

R18 CSE Team Saviors

⇒ WAP to Demonstrate the concept of Assignment of objects in Java.

Ans:-

Reference is located at Stack memory
→ objects located at heap memory

```
class Rectangle {
  int l,b;
  Rectangle(int x, int y){
    l=x;
    b=y;
  }
  void display(){
    S.o.p("length = "+l," breadth = "+b);
  }
}
```

Address

Address.l

Address.b

```
class AssignmentDemo {
  PSVM(){
    Rectangle r1 = new Rectangle(5,6);
    Rectangle r2 = new Rectangle(1,2);
    r1.display();
    r2.display();
```

r1 [2000] points to → 2000 [5 | 6]

r2 [3000] → 3000 [1 | 2]

this // current Address of r1 implicity

∴ If r1 = r2 [r1 replaced as r2]

R18 CSE Team Saviors

$r_1 = r_2$ [// Address of $r_2 = r_1$]

r_1.display();

Rectangle r_3 = new Rectangle (1,2);

S.O.P ($r_1 == r_3$); // false

S.O.P ($r_1 == r_2$); // True

→ [==] compares only Addresses of the objects and not values.

S.O.P (r_1.equals(r_3)); // false

↓

compares Addresses

→ Equals method in object class.

note

→ without overriding Equals method the statement (r_1.equals(r_3)) returns false.

→ override the equal method as following so that the statement (r_1.equals(r_3)) returns True

R18 CSE Team Saviors

```
class Rectangle {
    !
    public boolean Equals(Rectangle r) {
        return (this.l = r.l && this.b = r.b);
    }
}
```

→ Above one is overriding case of Equals methods.

→ overriden of Equals method check the values of the objects Not Addresses.

⇒ WAP to demonstrate various access Priviledges in inheritance.

Ans:-

```
class A {
    Private int i;
    A(int x) {
        i = x;
    }
    void display() {
        S.o.p("i=" + i);
    }
}
void set(int z) {
    i = 3;
}
```

output will be used

R18 CSE Team Saviors

```
class B Extends A {
    int j;
    B(int x, int y) {
        super(x);
        j = y;
    }
    void displayB() {
        Set(10);
        display();
        S.o.p("j = " + j);
    }
}

class InheritanceDemo {
    PSVM(---) {
        A o1 = new A(5);
        o1.display();  // i = 5
        B o2 = new B(2,3);
        o2.display();
    }
}
```

R18 CSE Team Saviors

NOTE :-

→ o_2 object has i but as it is a private member it cannot be accessed directly. But the private members can be accessed by using methods of the class. So we use the methods setters and getters.

⇒ STACK USING LINKED LIST :-

```
class Node {
    int data;
    Node link;
    Node (int d, int l) {
        data = d;
        link = l;
    }
}

class Stack {
    Node top;
    Stack() {
        top = null;
    }
```

R18 CSE Team Saviors

```
void push(int x) {
  Node temp = new Node (x, top);
  top = temp;
}

int pop() {
  if( isstackEmpty())
    return -1;
  else
    int y;
    y = top.data;
    top = top.Link;
    return y
}

boolean isstackEmpty() {
  return (top==NULL);
}
```

R18 CSE Team Saviors

UNIT-I

Generics

Wrapper classes :-

Wrapper class Hierarchy :-

~~⇒ Wrap to do~~

```
Integer job = new Integer(10); // Boxing
int i = job.intValue();// Unboxing
Integer job2 = 20; // Auto boxing
```

[done by compiler]

3000

```
int k = job2; // Autounboxing
```

R18 CSE Team Saviors

~~Z) WAp to demonstrate~~

⇒ WAp to demonstrate operations on wrapper classes.

Ans:

```
class WD {
    PSVM(————) {
        Integer Job = new Integer(10);
        int i = Job.intvalue();

        Integer Job2 = 20;

        int K = job2;
        S.O.pln("i=" + i);
        S.O.pln("j=" + K);
        int l = Integer.parseInt("2020");
        S.O.pln("l=" + l);                    //passing
        Integer job3 = Integer.valueof("2020
        S.O.pln(job3);
    }
}
```

→ If we pass reference variable in print statement by default to string method is called. So classname followed by hashcode is printed.

R18 CSE Team Saviors

Exceptions:-

→ In wrapper class, String class the toString method is overridden and hence the value in the reference variable will be printed if passed to print statement.

⇒ Pregeneric code

```
class NonGen {
    object ob;
    NonGen(object o){
        ob=o;
    }
    object getOb(){
        return ob;
    }
    void showType(){
        S.o.pln("Type of ob=" + ob.getclass().getNam
                                                ())
    }
```

NOTE:-

ob.getclass().getName() gives full class name

Eg:- java.util.Scanner

ob.getclass().getSimpleName() gives class name

Eg:- Scanner

R18 CSE Team Saviors

```
class NonGenDemo {
    PSVM(____) {
        NonGen iob = new NonGen(10)
```

New Integer(10)

Auto boxing

```
int i = ((Integer) iob.getOb()).intValue();
                                  // UnBoxing

int i = ((Integer) iob.getOb()) // AutoUnBoxing
iob.showtype(); // java.lang.Integer
```

prints Type of ob =

```
NonGen sob = new NonGen("xyz");
```

Auto boxing

```
String str = (String) sob.getOb();

System.out.println("str = " + str);

sob.showType(); // java.lang.String
```

R13 CSE Team Saviors

```
iob = sob ; // No compilation errors
int i = (Integer) iob.getob();
}  // NumberFormatException
}
```

Problems:-

1. Explicit conversion required while input
2. This is not type safe.

→ Generic class with one type parameter :-

Generic class :-

→ Type parameter [contains any object]

```
class Gen <T> {
  T ob;
  Gen (T o) {
    ob = o;
  }
  T getob() {
    return ob;
  }
  void showtype () {
    S.o.pln("Type of ob = " + ob.getclass().getName());
  }
}
```

P18 CSE Team Saviors

```
class GenDemo{
    PSVM(){
Gen<Integer> igo=new Gen<Integer>(10);
```

```
int i = igo.getob();
S.o.pln("i="+i); // java.lang.Integer
igo.showtype(); // java.lang.Integer

Gen<String> sgo = new Gen<String>("xyz");
String str = sgo.getob();
S.o.pln("str="+str);
str.showtype(); // java.lang.String
igo = sgo; // compilation Errors.
}
}
```

R18 CSE Team Saviors

⇒ WAP to create a class with generic arrays.

Ans :-

```
class GenArray<T> {
    T[] a;
    GenArray(T[] o) {
        a = o;
    }
    void display() {
        for (int i = 0; i < a.length; i++) {
            S.o.pln(a[i]);
        }
    }
}

class GenArrayDemo {
    PSVM(——) {
        Integer inums[] = {1, 2, 3, 4, 5};
        GenArray<Integer> iga = new GenArray<Integer>(inums);
        iga.display();
        String strs[] = {"abc", "xy", "mn"};
        GenArray<String> sga = new GenArray<String>(strs);
        sga.display();
    }
}
```

R18 CSE Team Saviors

→ In pre Generic code Java has given us the ability to create generalized classes, Interfaces & methods by operating through references of type object.

→ Since object class is the super class of other classes, an object reference can refer to any type of object, But the problem with this approach is that it lacks type safety. i.e., A reference of one specific version of a generic type (iob in NonGen Example) is treated as the same type as that of another version (sob in NonGen Example).

→ This leads to runtime Exceptions while Executing the code.

→ To avoid such Exceptions Generics introduced by JDK file.

→ Generics Enables us to create parameterized type i.e., it enables us to create class, Interfaces & Methods in which the type of data can be specified as a parameter. Hence Generics gives us the capability to create single class that works with different types of data with type safety.

R18 CSE Team Saviors

→ A key point to understand about generic types is that a reference of one specific version of a generic type is not type compatible with another version of the same generic type.

→ Thus generic avoids undesirable exceptions at runtime.

→ Generics also streamlines the process because it is no longer necessary to explicitely type cast the objects returned to the type of data i.e. actual operated upon.

→ with Generics all casts are automatic and implicites. Thus generics our ability to reuse code in a safe manner.

<u>Generic class with two type parameters</u>:-

→ A generic class can take 2 or more type parameters.

→ To specify 2 or more type parameters use a comma separated list.

R18 CSE Team Saviors

⇒ WAP to demonstrate use of a generic class with two type parameters.

Ans :-

```
class GenTwoparams<T, V> {
    T ob1;
    V ob2;
    GenTwoparams(T o1, V o2) {
        ob1 = o1;
        ob2 = o2;
    }
    void display() {
        S.o.pln("ob1 = " + ob1);
        S.o.pln("ob2 = " + ob2);
    }
}

class GenTwoparamsDemo {
    PSVM(——) {
GenTwoparams<Integer, String> gio =
    new GenTwoparams<Integer, String>
                    (10, "xyz");
```

R18 CSE Team Saviors

```
gio.display();  ob1 = 10
                ob2 = xyz
GenTwoparams<double, short> gdo =
    new GenTwoparams<double, short>(8.96, (short)10);
gdo.display();  ob1 = 8.96;
                ob2 = 10
    }
}
```

Generic class Hierarchies :-

```
class Gen1<T> {
    T ob1;
    Gen1(T o) {
        ob1 = o;
    }
}

class GenSub<T,V> Extends Gen1<T> {
    V ob2;
    Gen(T o1, V o2) {
        super(o1);
        ob2 = o2;
    }
    void display() {
        S.O.pln("ob1 = " + ob1);
        S.O.pln("ob2 = " + ob2);
    }
}
```

R18 CSE Team Saviors

```
class Main {
    PSVM(--){
GenSub<Integer, String> igo = new
        GenSub<Integer, String> (10, "xyz");
    igo.display();
  }
}
```

⇒ Generic classes can be part of class Hierarchie in just the same way as Non-generic class, Thus a generic class can act as a super class or a subclass.

→ Any type parameter needed by a generic super class must be passed up the hierarchy by all subclasses.

→ Thus even if a subclass of a generic superclass does not use generic types in its methods, it still must specify the type parameter in its header.

→ The subclass is free to add its own type parameters.

→ A method in generic class hierarchy can be overridden just like the methods in Non generic class hierarchy.

R18 CSE Team Saviors

* Generic Subclasses :-

→ It is perfectly acceptable to the super class of a generic subclass.

⇒ WAP to demonstrate Generic subclass :-

```
class NonGenSuper {
    int i;
    NonGenSuper(int k) {
        i = k;
    }
}

class Gen<T> extends NonGenSuper {
    T ob1;
    GenSub(int i, T o1) {
        super(i);
        ob1 = o1;
    }
    void display() {
        S.o.pln("i = " + i);
        S.o.pln("ob1 = " + ob1);
    }
}

class main {
    PSVM(—) {
        Gen<Integer> G = new Gen<Integer>(100,5);
        G.display();
    }
}
```

R18 CSE Team Saviors

Bounded types :-

⇒ WAp to demonstrate Bounded types ?

Ans :-

```
class GenBounded<T extends Number> {
                           ⇓
                        wrapper class
    T[] arr;
    GenBound (T[] o) {
        arr = o;
    }
    void display () {
        for (int i=0; i<arr.length; i++)
            S.O.pln (arr[i]);
    }

    double avg () {
        double sum = 0.0;
        for (int i=0; i<arr.length; i++) {
            sum = sum + arr[i].doubleValue();
        }
        return sum/arr.length;
    }
}
```

R18 CSE Team Saviors

```
class GenBoundDemo {
    PSVM(——) {
        Integer inums[] = {8, 11, 15, 3, 2};
        GenBounded<Integer> gio = new GB<Integers>(inums);
        gio.display();
        S.O.pln("Avg of Integer values = "+gio.avg());
    }
    Double inum[] = {8.0, 12.0, 15.0, 6.96, 7.6}
    GenBounded<Double> gia = new GB<Double>(inum);
        gia.display();
        S.O.pln(gia.avg());
    }
}
```

compilation Error:

```
String strs[] = {"ab", "xy", "mn"}
GB<String> gia = new GB<String>(strs);
```

R18 CSE Team Saviors

⇒ Some times it is useful to create a generic class that limits the types that can be passed through its type parameters.

→ To Handle such classes java provides Bounded types.

→ Bounded types are specified using Extends keyword while declaring type parameters.

→ Bounded Type (T) means T can be replaced Either the Super class T (or) any of its subclasses.

→ In addition to using a class type as a Bound we can also use an Interface type in this case classes that implements the interface only are legal.

→ A type parameter may Extend a class and implement one or more interfaces we use '&' symbol to connect them.

Syntax: <T Extends superclass & Interface1 & Interface2>

R18 CSE Team Saviors

Wild card arguments :-

Program :-

same as above class → In Main

```
if (gio.sameAverage(gdo))
        S.O.Pln("Avg is same");
else
        S.O.Pln("Avg is not same");
```

class

GenBounded

```
Boolean sameAverage(GenBounded<?> o)
{
    if (this.average() == o.average())
        return true;
    else
        return false;
}
```

It can take whatever T takes

↳ wild card argument

→ A wild card argument is specified by '?' and it represents any class that matches T as defined in the class Header.

R18 CSE Team Saviors

⇒ WAP to implement Bubble sort on user defined arrays and Built in arrays:

Ans:

```
class Bubblesort<T Extends comparable<T>>
{
    T[] a;
    Bubblesort(T[] o) {
        a = o;
    }

    void display() {
        for (int i=0; i<a.length; i++)
            S.o.pln(a[i]);
    }

    void Bubblesort() {
        int n = a.length ; T temp;
        for (int i=0; i<n-1; i++)
        for (int j=0; j<n-i-1; j++) {
            if (a[j].compareto(a[j+1])>0){
                temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
}
```

R18 CSE Team Saviors

```
class student implements comparable<student>
{
  int roll no;
  double marks;
  student(int r; double m) {
    roll no = r;
    marks = m;
  }
  public int compareTo(Student s) {
    if (This.marks > s.marks)
        return 1
    else if (this.marks < s.marks)
        return -1;
    else
        return 0;
  }
  public String toString() {
    return "rollno =" + rollno + " " "marks" +
        marks;
  }
}
```

R18 CSE Team Saviors

```
class BSDemo{
    PSVM(——){
    Integer inums[] = {8,1,3,15,10};
    Bubblesort<Integer> gbio = new Bubblesort<Integer>(Inums);
        gbio.display();
    S.o.pln("After sorting ...");
        gbio.Bubblesort();
        gbio.display();
    String strs[] = {"sai", "KL"};
    Bubblesort<String> sio = new BS<String>(strs);
        sio.display();
        sio.Bubblesort();
        sio.display();
    Student starr[] = {new student(1,90.8),
                       new student(2,76.3),
                       new student(3,50.9)};
    Bubblesort<Student> gbsto = new Bubblesort<Student>(starr);
```

R18 CSE Team Saviors

```
gbsto.display();
gbsto.bubblesort();
gbsto.display();
}
}
```

⇒ WAP to Demonstrate Generic Methods?

Ans:-

```
class GenMethodDemo{
    PSVM(){
    static <T> boolean isIn(T[] a, T x){
        for(int i=0; i<a.length; i++){
            if(a[i].equals(x))
                return True;
        }
        return false;
    }
    public static void main(String[] args){
    Integer inums[] = {8, 10, 3, 5, 21}
    if(isIn(inums, 10))
        S.O.pln("10 is present in inums");
    else
        S.O.pln("10 is not present in inums");

    String strs = {"ab", "xy", "cd"}
    if(isIn(strs, "mn"))
        S.O.pln("mn is present in strs");
```

R18 CSE Team Savors

```
else
    S.o.pln("mn is not present in strs");
  }
}
```

→ It is possible to declare Generic method inside a Nongeneric class.

→ A Generic method may use one or more Type parameters.

→ while calling generic methods types of parameters need not be given Explicitly, they are automatically inferred from the types of actual parameters.

→ The Type parameters list should precede the ~~result~~ ~~the~~ return type in method declaration.

Type Erasure:-

⇒ WAP to demonstrate type Erasure.

Ans:

```
class Gen<T>{
  T ob;
  Gen(T o){
    ob = o;
  }
  T getob(){
    return ob;
  }
}
```

R18 CSE Team Savors

```
class ErasureDemo{
    PSVM(--){
Gen<Integer> gio = new <Integer>(10);
    S.O.pln(gio.getob());
Gen<String> gso = new Gen<String>("xyz");
    S.O.pln(gso.getob());
    S.O.pln("Type of gio = "+gio.getclass().
                                getName());
    S.O.pln("Type of gso = "+gso.getclass().
                                getName());
  }
}
```

→ when a generic class is compile all generic type Information is Erased.

→ This means type parameters are replaced by:

→ object class if no Explicit bound is Specified for type.

→ appropriate bound if Explicit bound is specified.

→ Then, appropriate casts are applied to maintain type compatability.

→ compiler rewrites the above code as follows after type Erasure.

R18 CSE Team Saviors

```
class Gen {
    object ob;
    Gen( object o) {
        ob = o;
    }
    object getob() {
        return ob;
    }
}
class GrasureDemo {
    psvm(--) {
        Gen gio = new Gen(10);
        s.o.pln ((java.lang.Integer)gio.getob());
        Gen gso = new Gen("xyz");
        s.o.pln((java.lang.String)gso.getob());
s.o.pln("Type of gio = " + gio.getclass().getName());
s.o.pln("Type of gso = " + gso.getclass().getName())
    }
}
```

R18 CSE Team Saviors

Ambiguity Error:-

⇒ Demonstrate Ambiguity Errors due to type Erasure.

Ans:-
```
class Ambiguity<T,V>{
    T ob1;
    V ob2;
    void set(T o1){
        ob1=o1;
    }
    void set(V o2){
        ob2=o2;
    }
}
class AmbiguityErrorDemo{
    psvm(--){
        Ambiguity<Integer, Double> gido
        =new Ambiguity<Integer, Double>()
        gido.set(10); // results in compilation Errors
    }
}
```

→ Ambiguity Errors occur when Erasure causes to seemingly distinct Generic declaration resolve to the same Erased type causing a conflict.

R18 CSE Team Saviors

→ Method overloading is the root cause of such Ambiguity Errors. Hence when a Generic class uses two or more Type parameters do not overload methods.

⇒ WAP to implement Generic stack implementations.

Ans: stack.java

```
interface stack<T> {
    void push (T x);
    T pop();
    T topElement();
    boolean isEmpty();
    boolean isFull();
}
```

stackLL.java

```
class node<T>{
    T data;
    node<T> link;
    node (T d, node<T> l){
        data = d;
        link = l;
    }
}
```

R18 CSE Team Saviors

```
class stackLL<T> implements stack<T> {
    Node<T> top;
    stackLL() {
        top = null;
    }
    public void push(T, x) {
        Node<T> temp = new Node<T>(x, top);
        top = temp;
    }
}
```

R18 CSE Team Saviors

Type Inference :-

→ Type inference is the java compiler's ability to look at each method's invocation & corresponding declaration to determine the type of the argument that makes the invocation applicable.

→ Generic methods introduce us to type inference.

→ Type inference enables us to invoke a generic method as we would invoke an ordinary method without specifying a type B/w angle Brackets (< >).

→ Hence isIn(inums, 10) is used to invoke the method instead of <Integer> isIn(inums, 10).

→ Instantiation of generic class is another Example of type inference.

```
Gen<Integer> gio = new Gen<Integer>(10);
```

Can be replaced by

```
Gen<Integer> gio = new Gen<>(10);
```

P18 CSE Team Saviors

Restrictions on Generic types :-

1) Type parameters cannot be instantiated in Generic class.

Ex:-
```
class Gen<T> {
    T ob;
    Gen() {
        ob = new T(); // illegal
    }
}
```

2) static members cannot be used in generic context.

Ex:-
```
class Gen<T> {
    Static T ob;
    Static T method1() {
        // illegal
    }
    static void method2() {
        T ob;
    }
    Static void method3(T ob) // illegal
}
```

R18 CSE Team Saviors

3) Generic arrays can not be created.

Ex:- T[] ob = new T[];

4) A generic class cannot extend throwable i.e., we cannot create generic Exceptions.

R18 CSE Team Saviors

UNIT-II

Collections Framework

→ A collection is a container that represent a group of objects, often referred to as elements.

→ collections were added in 1.2 but went through modification in 1.5 version (due to generics) and is present in java.util. package.

→ Prior to the collection Framework, JAVA provided adhoc classes such as Dictionary, Vector, stack, HashTable and properties to store and manipulate group of objects.

→ The Java collections Framework supports four types of collections sets, lists, queues and maps.

→ The collections Framework was designed to meet several goals :-

1) High-performance:-

→ The implementations for the fundamental collections (dynamic arrays, linked lists, trees and hash tables) must be highly Efficient.

R18 CSE Team Saviors

2) Inter operability :-

→ The framework had to allow different types of collection to work in a similar manner and with a high degree of inter operability.

3) Extending and/or adapting :-

→ A collection had to be easy, Hence, the Entire collections framework is built upon a set of standard interface.

R18 CSE Team Saviors

JAVA'S ID collection Framework hierarchy

R18 CSE Team Saviors

Important methods of collection Interface

1) boolean add (E obj)
2) boolean addAll(collection<? Extends E>c)
3) void clear()
4) boolean contains(object obj)
5) boolean containsAll(collection<?>c)
6) boolean equals(object obj)
7) int hashcode()
8) boolean isEmpty()
9) Iterator<E> iterator()
10) boolean remove(object obj)
11) boolean removeAll(collection<?>c)
12) boolean retainAll(collection<?>c)
13) int size()

R18 CSE Team Saviors

List Interface :-

→ The list interface extends collection and stores a sequence of elements.

→ Elements can be inserted or accessed by their position in the list, using a zero-biased index.

→ A list may contain duplicate elements

→ list is a generic interface —

interface List<T>

Important methods of List interface :-

1) void add (int index, E obj)

2) boolean addAll(int index, collection<? Extends E> c)

3) E get(int index)

4) int indexof (Object obj)

5) int lastindexof(Object obj)

6) listiterator<E> listiterator()

7) listiterator<E> listiterator(int index)

8) E remove (int index)

9) E Set (int index, E obj)

10) List<E> sublist (int start, int end)

R18 CSE Team Saviors

Array List

→ The Array list class extends AbstractList and implements List interface.

→ Arraylist is a generic class -

class Arraylist<T>

→ Arraylist supports dynamic arrays that can grow as needed.

→ Arraylist has following constructors :

1. Arraylist() → default size is 16 byte
2. Arraylist (Collection<? Extends T> c)
3. Arraylist (int capacity)

→ One can increase the capacity of an Arraylist object manually by calling ensure capacity().

→ By increasing its capacity once, at the start, you can prevent several reallocations later.

→ Because reallocations are costly in terms of time, preventing unnecessary one improves performance.

void Ensurecapacity (int cap)

R18 CSE Team Saviors

→ To reduce the size of the array that underlies an Arraylist object so that it is precisely as large as the number of items that it is currently holding, use trimToSize()

void trimToSize()

//Demonstrate Arraylist.

⇒ WAP to Demonstrate methods of Array list.

Ans

```
import java.util.*;
class ArrayListDemo{
    PSVM(------){
ArrayList<String> al=new ArrayList<String>();
S.O.P("Initial size of al: "+al.size());
al.add("C"); al.add("A"); al.add("F");
al.add(1,"A2");
S.O.P("size of al after additions:"+al.size());
S.O.P("Contents of arraylist:"+al);
al.remove("C");
al.remove(2);
S.O.P("size of al after deletions:"+al.size());
S.O.P("contents of al:"+al);
}
}
```

P18 CSE Team Saviors

⇒ WAP to read list of n integer values; a arraylist replace all occurrance of the given element -1.

Ans:-

```
import java.util.*;
class ArraylistDemo{
    psvm(——){
Arraylist<Integer> a = new ArrayList<Integer>();
    a.add(1);
    a.add(2);
    a.add(
Scanner s=new Scanner(System.in);
int n = s.nextInt();
S.O.P("Enter Elements"+n);
for(int i=0; i<n; i++){
    a.add(s.nextInt());
}
S.O.P("arraylist = "+a);
S.O.P("Enter the key Elements");
int k = s.nextInt();
```

R18 CSE Team Saviors

```
for (int i=0; i<5; i++) {
if (a.get(i) == key) {
        a.set(i, (-1));
    }
}
S.O.P("After replacing Array list=" +a);
}
}
```

⇒ WAP to check whether list L2 is a sublist of L1 and print the starting index if it is present.

⇒ WAP to implement stack interface using ArrayList collection.

Ans:

```
class StackAL<T> implements Stack<T> {
Arraylist<T> als;
  StackAL() {
  als = new Arraylist<T>();
}
```

R18 CSE Team Saviors

```
public void push(T x) {
    als.add(x);
}
public T pop(){
    T x = null;
    if(als.isEmpty())
        return x;
    else {
        x = als.get(als.(size()-1));
        als.remove(als.(size()-1));
        return x;
    }
}
public T top(){
    T x = als.get(als.(size()-1));
    return x;
}
```

R18 CSE Team Saviors

```
public boolean isFull() { }

public boolean isStackEmpty(){
    return(als.isEmpty());
}

void display(){
 if(als.isEmpty())
    S.o.p("Stack is empty");
 else{
    for(int i=als.size()-1; i>0; i--){
    S.o.p(als.get(i));
    }
 }
}
class StackALDemo {
  psvm(--){
StackAL<Integer> sti = new StackAL<>();
  sti.push(5);
  sti.push(10);
Integer x=sti.pop();
  S.o.p(x);
}
}
```

R18 CSE Team Saviors

METHODS Defined by Interface Queue

→ boolean add(E obj)

→ boolean offer

Returns null	Returns Exception
→ boolean add(E obj)	→ boolean add(E obj)
→ E poll() [Return head, removing head]	→ E remove() [Remove ~~Return~~ head & return Ele]
→ E peek() (~~Element~~ Return head & Element will not removed)	→ E element() (Return head & Ele is not remove)

Methods of DEQueue Interface:-

Dequeue
- Double Ended Queue
- can add first or last
- remove from first or last
- double linked list are used to implement dequeue.

i) addFirst() or offerFirst()

ii) addLast() or offerLast()

iii) getFirst() or peekFirst()

iv) getLast() or peekLast()

v) removeFirst() or pollFirst()

vi) removeLast() or pollLast()

R18 CSE Team Saviors

LINKED LIST

→ The linked list class extends Abstract-sequential list and implements the List, Deque, and Queue interfaces.

→ It provides a linked list data structure.

→ LinkedList is a generic class

class LinkedList<T>

→ LinkedList has 2 constructors

i) LinkedList( )

ii) LinkedList(collections<? extends T> c)

⇒ WAP to demonstrate methods of Linked list.

Ans:-

```
import java.util.*;
class LinkedListDemo{
    psvm(——){
LinkedList<String> ll= new LinkedList
                        <String>();

ll.add("F");
ll.add("B");
ll.addLast("Z");
ll.addFirst("A");
```

R18 CSE Team Saviors

```
ll.add(1, "F");
S.O.P(" ll);
ll.remove("F");
ll.remove(2);
ll.removeFirst();
ll.removeLast();
String val = ll.get(2);
ll.set(2, val + "changed");
S.O.P(ll);
}
}
```

R18 CSE Team Saviors

⇒ WAP to implement stack interface using LinkedList collection class.

Ans:

```
class stackLL<T> implements stack<T>{
    LinkedList<T> sll;
    stackLL(){
    sll = new LinkedList<T>;
    }
public void push(T x)
{
    sll.addFirst(x);
}
public T pop(){
    if(sll.isEmpty())
        return null;
    else
    {
        T x;
        x = sll.removeFirst();
        return x;
    }
}
```

R18 CSE Team Saviors

```
public ~~void~~ T Top(){
    T x;
    x = sll.getFirst();(or) sll.getFirst();
    return x;
}
public boolean isStackFull{
    return false;
}
public boolean isStackEmpty(){
    return sll.isEmpty();
}
void display(){
    if(sll.isEmpty()){
        S.o.p("stack is empty");
    }
    else
    {
        S.o.p(sll);
    }
}
```

R18 CSE Team Saviors

```
class StackLLDemo {
    PSVM (——) {
StackLL<String> sstr = new
                    StackLL<>();
    sstr.push("ab");
    sstr.push("xyz");
    sstr.display();
S.O.println("Deleted : " + sstr.POP());
    sstr.display();
}
}
```

R18 CSE Team Saviors

⇒ WAP to implement Queue Interface using LinkedList collections class.

⇒ WAP to sort a given array using Built-in methods.

Ans:

```
import java.util.Arrays;
class sortcities{
    PSVM(---){
String[] cities = new String[]
                    {"Hyd", "Delhi",
                    "Mumbai"}
Arrays.sort(cities);
int i=0;
for(String temp : cities){
    S.o.pln("cities"+ ++i+" : "+temp)
  }
 }
}
```

Output:

cities 1 : Delhi
Cities 2 : hyd
Cities 3 : Mumbai

R18 CSE Team Saviors

⇒ WAP to sort a Given collection.

Ans:

```
import java.util.ArrayList;
import java.util.Collections;
class sortcitiesAL {
    PSVM(——){
ArrayList<String> cities = new
                ArrayList<String>();
cities.add("HYD");
cities.add("Delhi");
cities.add("Mumbai");
Collections.sort(cities);
int i=0;
for(String temp : cities) {
S.O.println("cities"+ ++i + ":" + temp);
}
}
}
```

~~Output~~

NOTE :-

→ collection is an Interface.

→ collections is a class.

R18 CSE Team Saviors

⇒ WAP to ~~sort~~ sort arrays of object.

Ans:

```
import Java.util.Arrays;
import java.util.Comparator;
class city implements Comparable<City>
{
  Private String cityName;
  Private int population;
  Private int Area;
  Public city(String cn, int p, int A){
      cityName = cn;
      Population = p;
      Area = A;
  }
  public String getcityName(){
      return cityName;
  }
  Public ~~String~~ int getpopulation(){
      return population;
  }
  public int getArea(){
      return Area;
  }
```

R18 CSE Team Saviors

```
public int compareTo(city cityob) {
int comparepopulation = cityob.getpopulation();
//ascending order
return this.population - comparepopulation();
}
}
class NameCompare implements Comparator<city>{
public int compare(city c1, city c2) {
return c1.getcityName().compareTo(c2.getcityName());
}
}
class AreaCompare implements Comparator<city>{
public int compare(city c1, city c2){
return (int)(c1.getArea() - c2.getArea());
}
}
```

R18 CSE Team Saviors

```
public class objectsort {
    .PSVM(—){
        city[] cities = {new city("Hydrabad",
                                 7000000),
                         new city("Delhi",1000000),
                         new city("Mumbai",8000000)};

S.O.pln("cities in their order of insertion");
int i=0;
for(City temp : Citis) {
 S.O.pln("city"+ ++i + ":"+ temp.getcity
                                Name()
        ,population:" + temp.getpopulation()
}
S.O.pln("sorting citis by their Name using
                    comparator");
NameCompare nc = new NameCompare();
Arrays.sort(citis, nc);
int i=0;
for(city temp:citis){
        S.O.pln("city"+ ++i +":"+ temp.get
                                cityName()
        +",population:"+ temp.get
                                Population()
    }
}
```

R18 CSE Team Saviors

The Set Interface

→ The Set Interface Extends collection interface and does not allow duplicate Elements.

→ the add() method returns false if an attempt is made to add duplicate Elements to a set.

→ It does not define any additional methods of its own.

→ set is a generic interface that has its declaration:

```
interface Set<E>
```

The HashSet class

→ Hash Set Extends AbstractSet and implements the set interface.

→ It creates a collection that uses a hash table for storage.

→ Hash set is a generic class

```
class HashSet<E>
```

→ The following Constructors

- HashSet()
- HashSet(collection<? Extends E>c)
- HashSet(int capacity)
- HashSet(int capacity, float fillRatio) → 0.75

R18 CSE Team Saviors

→ HashSet does not add any additional methods.

→ HashSet does not guarantee the order of its elements because hashing does not usually lend itself to the creation of sorted sets.

⇒ WAP to demonstrate HashSet.

Ans:- import java.util.*;

```
class HashSetDemo {
        PSVM(--) {
    HashSet<Integer> hs = new HashSet<>();
      hs.add("A");
      hs.add("B");
      S.O.P.ln(hs);
      hs.remove("A");
      S.O.P.ln(hs);
    }
}
```

retainAll
removeAll
addAll

R18 CSE Team Saviors

The Linked Hashset :-

→ The linkedhashset class Extends hashset and adds no members of its own.

→ It is a generic class

class LinkedHashset <E>

→ Linked Hasset maintains a linked list of the entries in the set, in the order in which they were inserted.

→ To see the Effect of Linked Hashset, substituate Linked Hashset for Hashset.

→ It is in the order of insertion.

The sorted set Interface :-

→ The sorted set Interface Extends set and declares the behaviour of a setsorted in asending order.

→ sorted set is generic interface

interface sortedset <E>

- tailset ( ) → ele
- headset ( ) → ele

R18 CSE Team Saviors

Methods of SortedSet:-

→ comparator <? super E> comparator();

→ E first()

→ SortedSet<E> headSet (E End)

→ E last()

→ SortedSet<E> subSet (E Start, E End)

→ SortedSet<E> tailSet (E Start)

Ex: S={3 8 9 15 25}

S_1 = S.tailSet(9);

S_1 = {9, 15, 25}

S_2 = S.headSet(15);

S_2 = {3, 8, 9}

S_2.remove(8);

S.o.p(S);

↓

S = {3 9 15 25}

R18 CSE Team Saviors

Navigablset

S = {3 8 9 15 25}

s.ceiling(7) → 8

s.ceiling(9) → 9

s.higher(7) → 8

s.higher(9) → 15

s.floor(7) → 3

s.floor(8) → 8

s.lower(7) → 3

s.lower(8) → 3

ceiling

higher

floor

⇒ WAP to Demonstrate treeset class

Ans:-
```
import java.util.*;
class TreeSetDemo{
    PSVM(--){
Treeset<String> hs = new Treeset<String>();
hs.add("B"); hs.add("D"); hs.add("E");
hs.add("C"); hs.add("H"); hs.add("A");
S.O.pln(hs);
S.O.pln("First element = " + hs.first());
S.O.pln(hs.ceiling("F"));
S.O.pln(hs.lower("F"));
Treeset<String> SS = (Treeset<String> hs.subset("A",
```

R18 CSE Team Saviors

```
S.o.pln("ss = " + ss);
ss.remove("B");
S.o.pln("hs = "+hs);
}
}
```

⇒ WAP to read to sets of integer value into TS1 and TS2. Compute & print the following sets.

Ans:-

```
import java.util.*;
class Treesetdemo{
    public s VM(—){
Treeset<Integer> TS1 = new Treeset<>();
Treeset<Integer> TS2 = new Treeset<>();
Treeset<Integer> TS3 = new Treeset<>(TS);
Treeset<Integer> TS4 = new Treeset<>(TS);
Treeset<Integer> TS5 = new Treeset<>(TS);
Scanner s = new Scanner(—);
S.o.pln("Enter size of TS1");
int n = s.nextInt();
S.o.pln("Enter the values");
for (int i=0; i<n; i++){
TS1.add(s.nextInt());
}
```

R18 CSE Team Saviors

```
int m = s.nextInt();
for (int i=0; i<m; i++) {
    TS2.add(s.nextInt());
}
s.o.pln (TS3.addall(ts2));
s.o.pln (TS4.retainall(ts2));
s.o.pln (TS5.removeall(ts2));
s.o.pln(TS1 \n TS2 \n TS3 \n TS4 \n
                    TS5);
}
}
```

⇒ WAP to create a class called person with income, age & name as its members. Read set A of persons from a user and compute & print the following sets.

i) set B of persons whose age > 60

ii) set C of persons whose income < 10000

iii) B ∩ C

R18 CSE Team Saviors

Ans:-

```
import java.util.*;
class person implements Comparable<person> {
  String name;
  int age;
  double income;
  Person(String n, int a, Double I){
    name = n;
    age = a;
    income = I;
  }
  public int compareTo(Person pob) {
  return this.name.compareTO(pob.name);
  }
  public String toString(){
  return ("Name = " + name + "age: " + age +
          " income: " + income);
  }
}
```

R18 CSE Team Saviors

```
class Main(){
    psvm(——){
Treeset<person> P1 = new Treeset<>();
                (or)
Treeset<person> A = {new person("xyz",65,
                                5000),
                     new person("abc",36,
                                7000),
                     new person("mnc",70,
                                11000),
                     new person("def",45,8000)};
Treeset<person> B = new Treeset<>(A);
for(person x : A){
  if(x.age >60)
      B.add(x);
  }
Treeset<person> C = new Treeset<>();
for(person x : A){
  if(x.age <10000)
      C.add(x);
  }
```

R18 CSE Team Saviors

~~Treeset<person> B=~~ new Treeset<>(B);

s.o.pln(B.~~add~~retainAll(C));

s.o.pln("persons from set A are ....");

printperson(A);

s.o.pln("person from setB are ....");

printperson(B);

s.o.pln("person from set C are ....");

printperson(C);

}

static void printperson(Treeset<person> x){

for (person x : x){

s.o.pln(x);

}

}

}

R18 CSE Team Saviors

Iterator :-

→ Iterator is a generic interface which offers general-purpose, standardized way of accessing the elements within a collection, one at a time.

interface Iterator <E>

→ Each collection implements iterator hence the elements of any collection class can be accessed to the methods of iterator.

→ Methods of iterator interface :-

- boolean hasNext()
- E next()
- void remove()

List Iterator :-

→ ListIterator extends Iterator, to allow bidirectional traversal of a list, and the modification of elements.

→ ListIterator is a generic interface

interface ~~Iterator~~ ListIterator <E>

R18 CSE Team Saviors

Methods of ListIterator :-

- void add (E obj)
- boolean hasNext ()
- boolean hasPrevious ()
- E next ()
- int nextIndex ()
- E previous()
- int previousIndex ()
- void remove ()
- void set (E obj)

⇒ WAP to demonstrate the methods of Iterator & ListIterator interfaces.

Ans:

```
import java.util.*;
class IteratorDemo{
    PSVM(——){
ArrayList<String> al = new
                ArrayList <String>();
al.add("C");
al.add ("A");
al.add ("E");
al.add ("B");
```

R18 CSE Team Saviors

```
al.add("D"); al.add("F");
s.o.pln("Original contents of al using
                Iterator:");
Iterator<String> itr = al.iterator();
while (itr.hasNext()){
  String element = itr.next();
  s.o.pln(element + " ");
}
s.o.pln();
//List Iterator
ListIterator<String> litr = al.listIterator();
while (litr.hasNext()){
  String element = litr.next();
  litr.set(element + "+");
}
s.o.pln("Modified contents of al:" +al);
s.o.pln("Modified ~~contents of~~ List
                Backwards:");
while (litr.hasprevious()){
  String element = litr.previous();
  s.o.pln(element + " ");
}  s.o.pln();
```

RIB CSE Team Saviors

NAVIgable set Interface

→ E ceiling(E obj)

→ Iterator<E> descendingIterator()

→ NavigableSet<E> descendingSet()

→ E floor(E obj)

→ Navigable set<E> headset(E upperBound
boolean incl)

→ E higher(E obj)

→ E lower(E obj)

→ E pollfirst()

→ E pollLast()

R18 CSE Team Saviors

UNIT-3

Dictionary :-

→ A Dictionary is a collection of pairs of the form (K, V) where K is a key and V is the value associated with the key no two pairs in the dictionary have the same key. (In normal dictionary)

operations performed on dictionary :-

1) Insert a pair into the dictionary.
2) delete the pair with the given key.
3) find the pair with the given key. (Search)
4) determine the size of the dictionary.
5) determine whether or not the dictionary is empty.

R18 CSE Team Saviors

Eg-1 :-

A Natural language dictionary like oxford dictionary.

$$(\text{key}, \text{value}) = (\underbrace{\text{word}}_{\text{key}}, \underbrace{(\text{meaning}, \text{synonyms}, \ldots)}_{\text{value}})$$

Eg-2:-

Student database

(Key, value) = (Rollno, (name, age, class ...))

Rollno → Key

(name, age, class ...) → value

dictionary with duplicates:-

→ It is similar to a dictionary but it permits two or more pairs to have the same key.

Eg1:- Telephone directory

(Key, value) = (name, (phnono1, phno2, Email, address, ...))

name → Key

(phnono1, phno2, Email, address, ...) → value

Ex-2:- Symbol Table

(Key, value) = (variablename, (Type, linenoaccessed, scope, value))

variablename → Key

(Type, linenoaccessed, scope, value) → value

R18 CSE Team Saviors

possible implementations of a dictionary:-

Sorted chain:-

→ It is a linked list of values in which all the values are maintained in the sorted order.

R18 CSE Team Saviors

⇒ WAP to implement dictionary using sorted chain.

Ans:-

pair.Java

```
class Pair <K ~~Extends~~ Extends Comparable <K>, V>
{
    K Key;
    V value;
    Pair (K k, V v)
    {
        Key = k;
        Value = v;
    }
    public string tostring () {
    return "(" + Key + "," + value + ")";
    }
}
```

R18 CSE Team Saviors

Dictionary.java

```
interface Dictionary<K extends Comparable<K>,
                                        V>
{
    void insert(Pair<K,V> p);
    Pair<K,V> remove(K key);
    Pair<K,V> find(K key);
    boolean isEmpty();
    int size();
}

class PairNode<K extends Comparable<K>, V>
{
    Pair<K,V> data;
    PairNode<K,V> link;
    PairNode(){ }
    PairNode(Pair<K,V> d, PairNode<K,V> l){
        data = d;
        link = l;
    }
}
```

P18CSE Team Saviors

```
class SortedChain<K extends Comparable<K>, V>
                implements Dictionary<K, V>
{
    PairNode<K,V> head;
        int dsize;
    SortedChain() {
        head = null;
        dsize = 0;
    }
```

```
    public boolean isEmpty() {
        return dsize == 0;
    }
    public int size() {
        return dsize;
    }
```

```
    public void insert(Pair<K,V> P) {
        PairNode<K,V> npnode = new PairNode<>(P, null);
        if(isEmpty()) // inserting into an empty dictionary
        {
            head = npnode;
            dsize ++;
        }
```

R18CSE Team Saviors

```
else
{
pairNode<K,V> temp=head ; prev = null;
//Traverse the sc to find out position for
                                  insertion
while (temp! = null && p.Key.compareTo
                        (temp.data.Key)>0)
{
    prev = temp;
    temp = temp.link;
}
if (temp! = null & p.Key.compareTo (temp.data.Key))
                                        ==0)
    S.O.pln ("Duplicate are not allowed");
else
{
    npnode.link = temp;
    if (prev = null) //insertion at the front
                                    of sc
    head = npnode;
else
    prev.link = npnode;
    dsize++;
}
}
```

R18 CSE Team Saviors

```
public pair<K,V> bind(K x) {
  if(isEmpty()) {
    S.O.pln("Dictionary is Empty");
    return null;
  }
  else
  {
    pairNode<K,V> temp=head;
    while(temp!=null & x.compareTo(temp.data.key)>0)
    {
      temp=temp.link;
    }
    if(temp==null) {
      S.O.pln("pair is not found");
      return null;
    }
    else if(x.compareTo(temp.data.key)<0)
    {
      S.O.pln("pair is not found");
      return null;
    }
```

P18 CSE Team Saviors

```
else {
    return temp.data;
  }
}
}
public pair<K,V> remove(K x) {
  if (isEmpty()) {
    S.O.Pln("Dictionary is empty");
    return null;
  }
else
  {
  PairNode<K,V> temp=head, prev=null;
while (temp!=null && x.CompareTo(temp.data.key)>0)
  {
    prev=temp;
    temp=temp.link;
  }
if (temp==null)
  S.O.Pln("pair is not found");
  return null;
if (temp!=null && x.compareTo(temp.data.key)==0)
{
    if(prev!=null) {
      prev.link=temp.link;
      Temp=temp.link;
      return temp.data.key;
  } dsize--;
```

R18 CSE Team Saviors

```
else {
    head = temp.link;
    dsize --;
    return temp.data;
}
}

else if (x.compareTo(temp.data.key) < 0)
{
    S.o.pln("pair is notfound");
    return null;
}
}
```

R18 CSE Team Saviors

```
class sortchaindemo{
  psvm(——) {
  sortedchain<Integer, String> sc = new
                    sortedchain<>();

  pair<Integer, String> p1 = new pair<>(2, "xyz");
  pair<Integer, String> p2 = new pair<>(8, "mn");
  pair<Integer, String> p3 = new pair<>(2, "abc");
  sc.display();
  sc.insert(p1); sc.insert(p2); sc.insert(p3);
  sc.display();
  pair<Integer, String> p4 = sc.find(8);
  if (p4 != null)
    s.o.pln("pair with key 8 =" + p4);
  s.o.pln("No of pairs = " + sc.size());
  pair<Integer, String> p5 = sc.remove(8);
  if (p5 != null)
    s.o.pln("pair removed 8 = " + p5);
  s.o.pln("No of pairs = " + sc.size());
  }
}
```

R18 CSE Team Saviors

MAPS (Unit 2 IInd Half)

→ A map is an object that stores key/value pairs.

→ A map is a collection of key value pair.

→ Keys must be unique, but values may be duplicated.

→ we cannot cycle through a map using a for-each style for loop.

The MAP Interface

→ Map is a generic interface

```
interface Map<K, V>
```

→ Although part of collections framework, maps are not themselves collections because they do not implement the collection interface.

R18 CSE Team Saviors

→ we can obtain a collection-view of a map by using the Entryset() method.

→ Entryset() returns a set containing the map entries.

→ Each of these set elements is a Map.Entry object.

The Map.Entry Interface

→ The Map.Entry interface enables us to work with a map entry.

→ Map.Entry is a generic inner interface of Map:

interface Map.Entry<K,V>

The methods defined by Map:

→ void clear()

→ boolean containsKey(object k)

→ set<Map.Entry<K,V>> entryset()

→ boolean equals(object obj)

→ V get(object k)

→ int hashCode()

→ boolean isEmpty()

→ set<K> keyset()

R18 CSE Team Saviors

→ V put (K k, V v)

→ void putALL (Map <? Extends K, ? Extends V> m)

→ V remove (object k)

→ int size()

→ Collection<V> values()

The methods Defined by Map.Entry :-

→ boolean equals (object obj)

→ K getKey()

→ V getValue()

→ int hashcode()

→ V setvalue(V v)

The HashMap class :-

→ The HashMap is a class that Extends Abstract map and implement the Map interface.

→ HashMap is a generic class

class HashMap <K, V>

R13 CSE Team Saviors

Constructors:

→ HashMap()

→ HashMap(Map<? Extends K, ? Extends V> m)

→ HashMap(int capacity) 16 default

→ HashMap(int capacity, float fillRatio)
⇓
0.75 default

→ HashMap does not guarantee any order of its elements.

⇒ WAP to demonstrate methods of HashMap.

Ans
```
import java.util.*;
class HashMapDemo{
    psvm(--){
HashMap<String, Double> hm = new HashMap<>();
hm.put("John", new Double(3434.34));
hm.put("Rajesh", new Double(3555.35));
// get a set of entries
Set<Map.Entry<String, Double>> myset
                    = hm.entrySet();
// Display the set
for(Map.Entry<String, Double> me : myset){
    S.o.p(me.getKey() + ":");
    S.o.println(me.getValue());
}
```

R18 CSE Team Saviors

```
s.o.pln();
s.o.pln(hm);
// Deposit 1000 into John's account
double balance = hm.get("John");
hm.put("John", balance + 1000);
s.o.pln("John's new balance :" + hm.get("John"));
}
}
```

LinkedHashMap :—

→ LinkedHashMap extends HashMap.

→ It maintains a linked list of the entries in the map, in the order in which they were inserted.

→ LinkedHashMap is a generic class

class LinkedHashMap<K, V>

→ LinkedHashMap defines the constructors

- LinkedHashMap()
- LinkedHashMap(Map<? extends K, ? extends V> m)
- LinkedHashMap(int capacity)

R18 CSE Team Saviors

• LinkedHashMap(int capacity, float fillRatio)

• LinkedHashMap(int capacity, float fillRatio, boolean order)

→ LHM adds only one method to those defined by HashMap.

→ This method is removeEldestEntry() and it is shown here:

protected boolean removeEldest Entry(Map.Entry <K,V> e)

SortedMap Interface:-

→ The SortedMap interface extends Map.

→ SortedMap is generic and is declared as shown here:

interface SortedMap<K,V>

→ To obtain a submap, use headMap(), tailMap(), or subMap().

→ To get the first key in the set, call firstKey().

→ To get the lastKey, use lastKey().

R18 CSE Team Saviors

Methods of SortedMap:

- Comparator<? super K> comparator()
- K firstKey()
- SortedMap<K,V> headMap(K end)
- K lastKey()
- SortedMap<K,V> subMap(K start, K end)
- SortedMap<K,V> tailMap(K start)

NAVIGABLEMap Interface

→ The NavigableMap interface was added by Java SE 6.

→ It extends SortedMap and declares the behavior of a Map that supports the retrieval of entries based on the closest match to a given key or keys.

→ NavigableMap is a generic interface.

```
interface NavigableMap<K,V>
```

R18 CSE Team Savors

Methods of NavigableMap

→ Map.Entry<K,V> ceilingEntry(K obj)

→ K ceilingKey(K obj)

→ NavigableSet<K> descendingKeySet()

→ NavigableMap<K,V> descendingMap()

→ Map.Entry<K,V> firstEntry()

→ Map.Entry<K,V> floorEntry(K obj)

→ K floorKey(K obj)

→ NavigableMap<K,V>
headMap(K upperBound,
boolean incl)

→ Map.Entry<K,V> higherEntry(K obj)

→ K higherKey(K obj)

→ Map.Entry<K,V> lastEntry()

→ Map.Entry<K,V> lowerEntry(K obj)

→ K lowerKey(K obj)

→ NavigableSet<K> navigableKeySet()

→ Map.Entry<K,V> pollFirstEntry()

→ Map.Entry<K,V> pollLastEntry()

R18 CSE Team Saviors

→ NavigableMap<K,V>

SubMap(K lowerBound, boolean lowInc, K upperBound, boolean highInc)

→ NavigableMap<K,V>

tailMap(K lowerBound, boolean Inc)

TreeMap class

→ The TreeMap class Extends Abstract Map and implements the Navigable Map interface.

→ It creating maps stored in a tree structure.

→ A TreeMap provides an Efficient means of storing key/value pairs in sorted order and allow rapid retrieval.

→ A treeMap guarantees that its Elements will be sorted in ascending key order.

→ Treemap is a generic class.

class TreeMap<K,V>

R18CSE Team Saviors

→ TreeMap constructors are defined.

- TreeMap()
- TreeMap(Comparator<? super K> comp)
- TreeMap(Map<? extends K, ? extends V> m)
- TreeMap(SortedMap<K, ? extends V> sm)

//WAP to demonstrate TreeMap class.

Ans:
```
import java.util.*;
class TreeMapDemo{
    psvm(---){
TreeMap<String, Double> tm =
  new TreeMap<>();
tm.put("Rajesh", new Double(343434));
tm.put("Ramesh", new Double(123.22));
Set<Map.Entry<String, Double>> set
        = tm.entrySet();
for (Map.Entry<String, Double> me:
                set){
    s.o.p(me.getKey()+":");
    s.o.pln(me.getValue());
}
```

R18 CSE Team Saviors

```
s.o.println();
double balance = tm.get("Rajesh");
tm.put("Rajesh", balance + 1000);
s.o.println("Rajesh new balance:" +
                    tm.get("Rajesh"));
}
}
```

<u>output</u> :-

Rajesh : 3434.34

Ramesh : 123.22

Rajesh new balance : 4434.34

R18 CSE Team Saviors

HASHING

→ Another possibility for representation of a dictionary is hashing.

→ Hashing uses a hash function to map dictionary pairs into positions in a table called Hashtable.

→ If a pair p has the key k and if H is the hash function then the pair p is stored in the position H(k) of the table.

→ Each position of the table is called as a BUCKET and H(k) is the HomeBucket for pairs whose key is k.

→ Hashing performs insertion, deletions and searches in constant time O(1). However worst case time is O(n).

→ If no two pairs hash into the same Buckets we call it has ideal hashing.

→ Ideal hashing is possible if the dictionary is too small.

→ But in Many application the Range in key values is so large that ideal hashing is impractical.

R18 CSE Team Saviors

In such cases hashing results in collisions.

→ Efficiency of hashing depends upon the hash function used.

→ The hash function should be selected in such a way that it take less time to compute and results in less number of collisions.

→ Hash functions that give good distribution of keys are called as uniform hash function.

→ While inserting an element if it hashes into the same bucket as a previously inserted element then we say that there is a collision.

→ There are 2 methods for dealing with collisions, which we called as collision resolution strategies.

Collision resolution strategies

- Separate chaining (closed hashing)
- Open Addressing
  - Linear probing
  - Quadratic probing
  - Double hashing

R18 CSE Team Saviors

Separate chaining :-

→ Insert the following keys into a hash table of size 11 | 8, 19, 76, 50, 66, 10, 89,

Ans:

→ In this strategy we keep a sorted linked list of all the elements that hash into a same bucket to perform an insertion we first determine the Home Bucket for the pair using the hash function and then traverse through the chain associated with home bucket to find an appropriate position for the pair and insert it at that position.

R18 CSE Team Saviors

→ Similarly we do for find & delete.

(or)

→ Any datastructure like a BST or another hash table may be used insted of sorted chain to resolve collision but if the dictionary is not too large and the hash function is not very good then it is not worth to try a complicated datastructure.

→ Separate chaining has the major dis-advantage of using linked list.

→ All the disadvantages associated with linked list are also associated with Separate chaining.

1. Search algorithm tends to slow down as it has to traverse through the sorted chains.

2. Insertion Algorithm takes longer time because it has to create a new node for every pair.

3. It has 2 datastructures i.e., hash table and sorted chain

R18 CSE Team Saicics

→ WAP to implement collision resolution using separate chaining.

Ans:

Open-Addressing schemes :-

→ All open-Addressing schemes probe in buckets $h_0(x)$, $h_1(x)$, $h_2(x)$ ....

$$h_i(x) = (h(x) + f(i)) \% \text{ Table size}$$

$$\{h(x) = x \% \text{ Table size}\}$$

i = no of the attempt to resolve collision

for Linear probing

$$f(i) = i$$

Quadratic probing

$$f(i) = i^2$$

Double Hashing.

$f(i)$ = Another function of x & i

R18 CSE Team Saviors

⇒ Hash the following keys into table of size 11 using linear probing for collision.

55, 79, 80, 4, 26, 36, 11, 98, 3

Ans:

Hash table

0	55
1	11
2	79
3	80
4	4
5	26
6	36
7	3
8	
9	
10	98
11	

$h_0(55) = (0)$

$h_0(79) = 2$

$h_0(80) = 3$

$h_0(4) = 4$

$h_0(26) = 4$

$h_1(26) = (4+1) \% 11 = 5$

$h_0(36) = 3$ ← C

$h_1(36) = (4) \% 11 = 4$ ← C

$h_2(36) = 5 \% 11 = 5$ ← C

$h_3(36) = (5+1) \% 11 = 6$

$h_0(11) = 11 \% 11 = 0$ ← C

$h_1(11) = 1$

$h_0(98) = 10$

$h_0(3) = 3$

→ Group of elements are clusters.

R18 CSE Team Saviors

primary clusters	secondary clusters
- large	- small
- fewer in number	- more in number compared to primary
- search time more	- search time less
- will fill in empty bucket	- will fill only in secondary clusters
- less disastrous	- More disastrous.

→ In an open-Addressing scheme a second datastructure is not used for collision resolution instead the same hashtable is used and alternate cells are tried until an empty cell is found.

→ In Linear Probing, the collision resolution function is $f(i) = i$.

→ This amounts to trying cells sequencially with wrap around in search of an empty cell.

R18 CSE Team Saviors

→ Load Factor λ = $\frac{\text{No of pairs in the dictionary}}{\text{Total size of hash table}}$

$\lambda = 9/11 \simeq 1$

→ Always Load factor ≤ 0.5 else rehash so that search time decreases.

→ For linear probing $\lambda <= 0.5$ ~~that~~ i.e., the hash table should never be more than half full.

→ As long as the table is big enough a free cell can always be found. But sometimes groups of occupied cells start forming which are called as clusters.

→ clusters formed by using linear probing are called primary clusters.

→ Any key that hashes into the clusters will require several attempts to resolve the collision and then it will add to cluster

R18 CSE Team Saviors

Algorithms:

⇒ Insertion using linear probing.

Ans → To insert a pair, ~~into the~~ first search for an empty bucket using the formula

$$h_i(x) = (h(x) + i) \% \text{ tablesize}$$

$$i = \text{no of attempts}.$$

→ insert the pair at the first empty bucket encountered.

⇒ Searching using linear probing:

Ans

→ The search for a key k begins at the home bucket h(k) and continues by examining successive buckets until one of the following happens.

1. A bucket containing pair with k is found.
2. An empty bucket is reached.
3. We return back to home bucket.

→ In these 2 cases table doesn't contain the pair.

R18 CSE Team Saviors

⇒ deletion using linear probing :-

→ deletion method should be written in such a way that search algorithm works correctly.

→ If we simply delete a pair from cluster searches for pairs down the cluster will fail, hence initial solution is to move several pairs in the cluster up in the table but while moving pairs we must take care not to move a pair to a position beyond its home bucket.

Another alternative :-

→ Is to use never used bit for each bucket, when the table is initialised this field is set to true for all buckets, when a pair is inserted into a bucket its never used bit is set to false. Now condition 2 for search algorithm is rewritten as

- A bucket with its never used bit is true is reached.

R18 CSE Team Saviors

Quadratic probing:-

⇒ Hash the following keys into hashtable of size 8 and use quadratic probing for collision resolution.

8, 21, 56, 77, 68, 5, 13.

②

Ans:

0	8
1	56
2	
3	
4	68
5	21
6	77
7	

$h_0(5) \rightarrow$ will not

R18 CSE Team Saviors

Separate chaining

```
class Separate chaining {
    psvm( ——— ) {
ArrayList <sortedchain<Integer, String>>
    ht = new ArrayList<>(11);  size
for(int i=0; i<11; i++){
    ht.add(i, new sortedchain<Integer,
                String>);
}
Scanner s = new Scanner( ——— );
int ch;
while(true){
s.o.pln("Enter 1 for insert \n 2 for search \n
        3 for delete\n 4 for display\n5
        for Exit");

  ch=s.nextInt();
Switch(ch){
  case 1:{s.o.p("enter key & value");
        int k=s.nextInt();
        String val = s.next();
        Pair<Integer,String> p = new
                Pair<>(k,val);
```

R18 CSE Team Saviors

```
ht.get(k%11).insert(p);
} break;

case 2: { S.o.pln("Enter key to search");
        int k = S.nextInt();

        Pair<Integer,String> P1 = ht.get(k%11).
                                  find(k);

        if(P1 == null)
            S.o.pln("pair not found");

        else
            S.o.pln("found the pair:" + P1);
        } break;

case 3: { S.o.pln("delete");
        int k = S.nextInt();
        Pair<Integer,String> P2 = ht.get(k%11).
                                  remove(k);

        if(P2 == null)
            S.o.pln("Not found");

        else
            S.o.pln("found" + P2);

        } break;
```

R18 CSE Team Saviors

```
case 4: { S.o.pln("display"); ~~Pair<Integer, String> temp;~~
        for(int i=0; i<11; i++){
            ht.get(i).display();
        } break;
case 5: S.o.p("wrong choice");
        System.exit(0);
    }
  }
}
}
```

R18 CSE Team Saviors

Linear Probing:

```
class LinearProbing<K extends Integer, V>
                                implements Dictionary
{                                          <K,V>
ArrayList<pair<K,V>> ht;
  int dsize;
LinearProbing () {
    ht = new ArrayList<pair<K,V>> (11);
    dsize = 0;
  }
```

R18 CSE Team Saviors

Quadratic Probing :-

→ Quadratic probing, it eliminates primary clusters

→ Even quadratic probing results in formation of clusters called as secondary clusters.

→ If the choice of hash function is poor there would be no gaurantee of finding an empty cell even through some empty cells exist.

→ This is more common when the table gets more than half filled and if the table size is not prime. Hence the following 2 things must be kept in mind while using open addressing schemes.

a) Load factor should be below 0.5.

This is because all the data goes inside the table, a bigger table is needed for finding empty cells quickly.

b) The table size should be prime.

R18 CSE Team Saviors

Double Hashing :-

→ In double hashing we use a second hash function for resolving collision.

$$h_i(x) = (h(x) + i \cdot g(x)) \bmod \text{Table size}$$

is used for collision resolution where $g(x)$ is a function such as

$g(x) = R - x \bmod R$ where R is a prime number smaller than the table size will work well. It is also important to make sure that $g(x)$ does not evaluate to 0 and table size is prime.

⇒ Hash the following keys into table of size 13 using double hashing the keys are

8, 21, 56, 77, 68, 5, 13

Ans :-

0	13
1	
2	21
3	68
4	56
5	5
6	
7	
8	8
9	
10	
11	
12	77

$h_0(8) = 8$

$h_0(21) = 8$

$h_1(21) = (8 + 1 \cdot g(21)) \% 13$

$g(21) = R - 21 \% R$

$R = 7$

$= 7 - 0 = 7$

$h_1(21) = (8 + 7) \% 13$

$= 2$

R18 CSE Team Saviors

Rehashing:-

⇒ Rehash the given table using linear probing

0	6
1	15
2	23
3	24
4	
5	
6	13

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

$h_0(6) = 6 \% 17 = 6$

$h_0(15) = 15 \% 17 = 2$

$h_0(23) = 6$

$h_1(23) = 7$

$h_0(24) = 7$

$h_1(24) = 8$

$h_0(13) = 4$

NOTE:-

→ Take table size as prime number near double the size of original table size (greater).

→ If the table gets too full the running time for the operations may take too long and insertion might fail for open addressing schemes.

→ A solution then is to build another table that is about twice as big as the original hash table.

R18 CSE Team Saviors

→ To Rehash the Entire hash table compute new hash value for each element and insert in the new value.

→ obviously, rehashing is very Expensive operation.

→ The running time is O(n) where n is the size of the dictionary.

→ Rehashing is actually not that bad because it frees the programmer from worrying about the table size.

→ Rehashing cannot be avoided because hash tables cannot be made arbitrarily large in Real time applications.

→ Rehashing should be done when

1) The table is half full or the table reaches certain load factor.

2) The insertions failed.

R18 CSE Team Saviors

Binary search tree :-

Graph :-

→ A Graph G is non-linear data structure which is represented as $G = \langle V, E \rangle$ where V is set of nodes or vertices and E is set of edges connecting the nodes.

→ The Tree is a Acyclic Graph and hence it is also a non-linear data structure.

Tree :-

→ A Tree is a finite non-empty set of nodes such that there is a specifically designated node called as the root.

→ The remaining nodes are partitioned into n disjoint sets $T_1, T_2, \ldots, T_n$ where each of these sets is a Tree and is called as a subtree of the root.

Indegree of a node :-

→ The no of edges directed towards the node.

Indegree(A) = 0

Indegree(C) = 1

Indegree(E) = 1

R18 CSE Team saviors

outdegree :-

→ The no of Edges directed away from the node.

out-degree (A) = 3

out-degree (C) = 1

out-degree (E) = 1

degree :-

→ In case of trees outdegree of a node is called as degree of a Node.

→ Root Node is a node whose indegree is 0 and is the topmost node of the tree.

→ All non-root nodes have indegree 1 Exactly.

→ A node in a tree can have 0, 1 or more as out degree.

Leaf Node :-

→ The Node whose outdegree is 0 and is called as terminal node or outer node.

Internal Node :-

→ The Node that is neither a root nor leaf. It is also called as inner node or a branch node.

→ the

R18 CSE Team Saviors

Child nodes and parent nodes :-

→ The nodes which are connected by edges that are directed away from a node N are called as child nodes of N.

→ N is called as the parent node for all its child node.

→ A root node does not have a parent node.

→ A leaf node does not have a child node.

→ Two or more nodes with the same parent are called as siblings.

Adjacent Nodes :-

→ 2 Nodes are set to be Adjacent if they are connected by a common edge.

→ A path is the sequence of nodes in which each node is adjacent to next one.

→ Every node in a tree can be reached by following a unique path of starting from the root.

R18 CSE Team Saviors

Levels :-

→ The level of a Node is defined as follows.

i) The root is at level - 1

ii) The level of a node is 1 greater the level of his parent.

→ siblings are always at same level. But all nodes in the same level are not necessarily siblings.

Binary Tree :-

→ A Binary Tree (T) is a finite collection of nodes when it is not Empty it has a root node and the remaining nodes are partitioned into 2 binary trees which are called as left and right subtrees of T.

→ The height of the BT is the level of the leaf in the longest path from the root node.

height of Tree = 4

R18 CSE Team Saviors

Properties of a BT :-

→ For a BT with n nodes and height h, the following properties holds good.

1. $h \le n \le 2^n - 1$, Given h
2. $\log_2(n+1) \le h \le n$, Given n

Strictly BT :-

→ A BT is called as strictly BT if Every node in tree has 0 or 2 child nodes exactly.

Strictly BT
||
NOT FULL BT

Full Binary Tree :-

→ A FBT is a strictly BT & all leaf nodes are ~~not~~ same level.

FBT

R18 CSE Team Saviors

Complete Binary Tree :-

→ A CBT at height h comprises of full BT of height h−1 (+). the nodes at the level h such that all the nodes in the last level h are as left as possible.

Representation of Binary Trees :-

1) Arrays
2) Double Linked list.

→ Given a BT of height h and

$m = 2^h - 1$

where m = max no of nodes

→ The Representation of the tree using an Array can be done as follows.

R18 CSE Team Saviors

1) Take an Array of size m+1.

2) put the root node at index 1

3) for any root with index i :

i) Its left child if present should be put at index 2i.

ii) Its Right child if present should be put at index 2i+1.

iii) It's parent is found at $[i/2]$

Ex :- Represent the following tree using an array.

h = 3

Array Size = 2^h = 8

X	A		B		C		C
0	1	2	3	4	5	6	7

R18 CSE Team Saviors

→ Given the following array representation draw the BT.

	X	Y	Z	L		M	N						P		Q
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Ans: $2^h = 16$

$\boxed{h = 4}$

Binary Tree

R18 CSE Team Saviors

REPRESENTING BST using Double linked List :-

→ For Representing BST using linked list we should use a node with the following structure.

⇒ WAP to implement dictionary using Binary Search Tree.

Ans:-

```
class TreeNode<K extends comparable<K>,V>
{
    TreeNode<K,V> lptr, rptr ;

    Pair<K,V> data;

    TreeNode (Pair<K,V> d) {

    data = d ;

    lptr = rptr = null;

    }
}
```

R18CSE Team Saviors

```
class BST<K extends Comparable<K>, V>
        implements Dictionary<K,V>{

  TreeNode <K,V> head;
    int dsize;
    BST(){
      head = null;
      dsize = 0;
    }

public boolean isEmpty(){
    return (head == null);
}

public int size(){
    return dsize;
}

public Pair<K,V> find(K key){
  if (isEmpty()){
    S.o.pln("BST is Empty"); return null;
  }

TreeNode <K,V> temp = head;
  while(Temp != null){
    if (key.compareTo(temp.data.key) < 0)
        temp = temp.lptr;
```

R18 CSE Team Saviors

```
else if (key.compareTo(temp.data.key) > 0)
        temp = temp.rptr;
else
        return temp.data;
}
System.out.println("pair Not found");
return null;
}
public void insert(pair<K,V> p) {
TreeNode<K,V> ntnode = new TreeNode<K,V>(p);
if(isEmpty()) {
        head = ntnode;
        dsize++;
        return;
}
TreeNode<K,V> temp=head, ptemp=null;
while (temp! = null) {
        ptemp = temp;
if(p.key.compareTo(temp.data.key)<0)
        temp = temp.lptr;
else if (p.key.compareTo(temp.data.key)>0)
        temp = temp.rptr;
```

R18 CSE Team Saviors

```
else {
    system.out.println("Duplicates not allowed");
    return ;
  }
}
if (p.key.compareTo(ptemp.data.key)<0)
        ptemp.lptr = ntnode;

else
        ptemp.rptr = ntnode;

    dsize++;

}
public void inorder ( ){
    if (isEmpty()){
        S.O.pln("BST is Empty");
        return ;
    }
    inorder(head);
}
void inorder (TreeNode <K,V> temp){
    if (temp! =null){
    inorder (temp.lptr);
    S.O.pln(temp.data);
    inorder (temp.rptr);
  }
}
```

P18 CSE Team Saviors

Delete from BST

3 cases of deletion:-

1) Deleting a leaf node.

Ex:- 37.

2) Deleting a node with one child node

Ex:- 32

3) Deleting a node with 2 child nodes

Ex:- 25

R18 CSE Team Saviors

program

```
public void remove(K key){
    if(isEmpty()){
        S.O.println("BST is Empty");
        return;
    }

TreeNode<K,V> temp=head, ptemp, S,
                    PS, C=null;
//temp is the node to be deleted.
//ptemp is parent of temp
//S is inorder predecessor of temp
//PS is parent of S
//C is child of temp
while (temp!=null & key.compareTo
                    (temp.data.key)!=0){
    ptemp=temp;
    if(key.compareTo(temp.data.key)<0)
        temp=temp.lptr;
    else
        temp=temp.rptr
}
```

R18 CSE Team Saviors

```
if(temp==null){ //Key is not found.
    S.o.pln("Key is not found");
    return;
}
//Key is found.
//Two child nodes case
if(temp.lptr!=null && temp.rptr!=null)
{
    S=temp.lptr;
    PS=temp;
//Locating inorder predecessor
while(S.rptr!=null){
    PS=S;
    S=S.rptr;
}
//replace temp.data with S.data
temp.data=S.data;
```

R18 CSE Team Saviors

```
//convert s to temp
    temp=s;
    ptemp=ps;
}
//one child node case
//mark child of temp
  if(temp.lptr!=null)
        c=temp.lptr;
  else if(temp.rptr!=null)
        c=temp.rptr;
// first node is to be deleted
  if (head==temp){
        head=c;
  }
  else{
      if (ptemp.lptr==temp)
            ptemp.lptr=c;
```

R18 CSE Team Saviors

```
else
    ptemp.xptr = c;
}
dsize --;
}
```

R18 CSE Team Saviors

AVL Search Trees

→ The complexity of search algorithm in BST is $O(\log(n))$ But worst case complexity is $O(n)$ where n is number of nodes in the true. Hence BST cannot gaurantee $O(\log n)$ for search, insert and delete operation.

→ Trees with wort case height of $O(\log n)$ are called balanced trees.

AVL — Adelson velskil & Landis (1962)

→ one of the popular tree is AVL search tree.

→ An Empty binary tree is an AVL trees if T is a non-Empty binary tree with T_L and T_R as its left and right subtrees then T is an AVL tree if and only if

$|h_L - h_R| \leq 1$

R18 CSE Team Saviors

→ h_L - height of T_L

h_R - height of T_R

→ T_L and T_R are AVL trees themselves

AVL Search Tree:-

→ It is a BST that is also an AVL tree.

Representation of AVL trees:-

→ They are represented using doubly linked list scheme that is used for Binary trees.

Balance Factor:-

$$\boxed{b(x) = h_L - h_R}$$

→ To facilitate insertion and deletion a balance factor is associated with each node. The balance factor $b(x)$ of a node x is defined as $h_L - h_R$

R18 CSE Team Saviors

→ According to definition of AVL Search trees the permissable balance factors are -1, 0, 1.

Searching an AVL Search Tree :-

→ The search algorithm used for BST may be used for searching AVL search tree also.

Insertion into AVL Search Tree :-

→ The insertion algorithm of BST cannot be used as it is for insertion into AVL Search Tree since resulting tree may no longer be an AVL Search Tree.

Insertion Algorithm :-

→ use the following steps for inserting a value into AVL Search Tree.

1) Insert the node into proper position following the properties of BST.

2) Compute the Balance factor on the Path starting from root node to the newly inserted node computi

R18 CSE Team Saviors

the Balance factor of each node. It can be seen that change in balance factor will occur only in this path.

3) Decide the pivot node :-

→ On the path traced in step 2 determine whether the balance factor of any node changes to 2 (or) −2.

The node whose |balance factor| changes from 1 to 2 is called pivot node.

There may be more than one node whose |balance factor| switches from 1 to 2. Take the node nearest to the newly inserted node should be consider as pivot node.

4) Balance the unbalanced tree. It is important to manipulate the pointers centered at the pivot node, so as to bring back the tree into height balance.

→ This pointer manipulation is known as rotation.

R18 CSE Team Saviors

→ There are 4 cases of rotation

1) LL
2) LR
3) RR
4) RL

→ Among this LL & RR are single rotation. where as LR & RL are called as Double rotation.

○ → Single Node

△ → Subtree

LL Rotation:-

LL → Left child | Left SubTree

LL imbalance.

R18 CSE Team Saviors

→ Imbalance is due to Insertion in Left Subtree of the Left child of the pivot node.

→ To balance the tree perform the following pointer management.

→ 1) The Right SubTree A_R of A Becomes Left SubTree of P.

→ 2) P Becomes the Right child of A

3) A Becomes the new root of the subtree earlier rooted as P

Ex:-

LL imbalance at 6

R18 CSE Team Saviors

→ Imbalance is due to in Right subtree of right child in pivot Node.

RR Rotation :-

RR imbalance

→ To balance the tree perform the following pointer management

R18 CSE Team Saviors

1) Left subtree B_L of B becomes the RST of p

2) P becomes left child of B

3) B becomes the new root of the sub-tree earlier rooted at p.

R18 CSE Team Saviors

LR Rotation :-

→ Imbalance is due to insertion in Right sub tree of Left child of pivot node to balance this imbalance the following manipulations in pointers.

→ Rotation 1 B_L Becomes the RST of A Becomes left child of B.

→ Rotation 2 B_R Becomes LST of P and P becomes Right child of B.

→ B Becomes the new root of the sub-tree earlier rooted at P.

R18 CSE Team Saviors

Eg:-

insert 7 →

LR →

R18 CSE Team Saviors

RL Rotation:-

→ Imbalance is due to Insertion in left sub tree of right child of pivot node.

→ To restore balance perform the following manipulation in pointers.

Rotation 1 :- B_R becomes LST of A, A becomes right child of B.

→ Rotation 2 :- B_L becomes RST of P, and P becomes left child of B.

→ B Becomes the new root of the sub-tree earlier rooted at P.

R18 CSE Team Saviors

Eg:-

R18 CSE Team Saviors

⇒ construct AVL Trees using

8, 9, 10, 2, 1, 5, 3, 6, 4, 7, 11, 12

Ans :-

Insert 8 :-

Insert 9 :-

Insert 10 :-

Insert 2 :-

R18 CSE Team Saviors

Insert 1:-

Insert 5:-

Insert -3:-

R18 CSE Team Saviors

Insert 6 :-

Insert 4 :-

Insert 7 :-

R13 CSE Team Saviors

Insert -11 :-

Insert 12 :-

R18 CSE Team Saviors

Deletion from AVL Trees:-

→ Let P be the node which has become imbalance after deleting a node.

→ If the deletion takes place from RST of P then B(P) changes from 1 to 2.

→ Similarly, if the deletion took place from LST of P then B(P) changes from -1 to -2.

Procedure for deletion:-

Step-1 :- Use BST deletion procedure to delete a given node from the AVL node.

Step-2 :- Calculate BF of all nodes that fall on the path from the deleted node to the root node.

→ If BF of all the nodes are 0, 1 or -1 then the tree is Balanced.

Otherwise,

the Tree is imbalance and the following steps have to be used,

R18 CSE Team Saviors

Step-3 :- Let P be the node whose Balance factor is changes to 2 or -2 after deletion.

→ In case of multiple search node select the one which is closest to the deleted node.

Step-4 :- To restore balance at P, the type of imbalance is classified either R-type or L-type imbalance.

→ If the deletion took place from P's RST then it is called as type-R imbalance.

Otherwise,

it is type-L imbalance.

→ Type-R imbalance is further classified into R_0, R_1, R_{-1} imbalance.

R18 CSE Team Saviors

R_0 Imbalance:-

deleting a node from P_R

Before deletion, height = h+2

After deletion, height = h+2

Right Rotation

After Rotation, height = h+2

→ The type R_0 refers to the case when the deletion took place from Right Sub Tree of P and B(A) = 0. R_0 imbalance is balanced by performing R_0 rotation which is Right Rotation.

R18 CSE Team Saviors

→ The height of the subtree rooted at p before and after rotation is h+2. hence the BF of the remaining node on the path from p to the root are unchanged as a result the entire tree is balanced after rotation.

R1 Imbalance

Before deleting, height = h+2

After deletion, height = h+2

Right Rotation

After rotation, height = h+1

R18 CSE Team Saviors

→ The type R_1 refers to the case where in deletion, took place from RST of P and $B(A) = 1$.

→ R_1 imbalance is balanced by R_L rotation which is right rotation.

→ But the new height of the subtree after rotation is 1 less than the height of the subtree before rotation, hence 1 rotation will not suffice to balance the entire tree rotations might be performed on the path from the deleted node to the root node wherever BF becomes 2 or −2.

→ The no of rotations needed is $O(\log n)$

R−1 Imbalance:

Before deletion, height = h+2

After deletion, height = h+2

After rotation, height = h+1

R18 CSE Team Saviors

→ The type R_{-1} imbalance refers to the case where deletion took place from RST of P and B(A) = −1.

→ R_{-1} imbalance is balanced by performing R_{-1} rotation which is a double rotation - Left, right rotation. Even in this case height of the subtree reduces by 1 after rotation. Thus, if p has ancestors, their BF's would change. Hence more no of Rotation might have to be performed to balanced the entire tree.

Type-L Imbalance:-

→ It occurs when deletion took place from LST of node p.

→ It is further classified into L_0, L_{-1}, L_1 Imbalances.

R18 CSE Team Saviors

Deletion Rotations:-

R0 $\xrightarrow{\text{similar to}}$ LL → one rotation is enough to balance the entire tree.

R1 → LL → one or more rotations

R-1 → LR → one or more rotations

L0 → RR → one rotation

L-1 → RR → one or more rotations

L1 → RL → one or more rotation

R18 CSE Team Saviors

Priority Queue :-

→ The priority Queue is a collection of zero or more elements in which each element has a priority associated with it.

→ The operations that are performed on priority Queues are find, insert and delete, named as top, push and pop respectively.

→ Priority queues are implemented using Max Heap (or) Min Heap.

Max Tree :-

→ It is a Binary Tree in which value in each node is greater than or equal to those in its child nodes.

~~Max~~

Ex :-

R18 CSE Team Saviors

Max Heap :-

→ The Max Heap is the Max tree that is also a complete Binary Tree.

Ex :-

Min Tree :-

→ It is a Binary Tree in which value in Each node is less than or equal to those in its child nodes.

Ex :-

Min Heap :-

→ A Min Heap is a min tree that is also complete Binary Tree.

R18 CSE Team Saviors

Ex)

Operations on Priority Queue:-

1) Insertion:

swap 95 ↔ 76

swap: 95 ↔ 85

Max Heap

R18 CSE Team Saviors

→ when a value has to be inserted into a max heap containing n elements. It is first added at n+1 position and then resulting tree is simplified.

→ The process of swapping parent node values with its child node values to obtain a max heap is called as ~~heapification~~ heapification.

2) deletion:-

R18 CSE Team Saviors

→ when an Element is to be removed from Max Heap it is to be deleted from root of the Tree.

→ The Element from n^{th} node is copied into the root node and n^{th} node is deleted then the resulting tree is heapified.

HeapSort :-

56, 8, 75, 95, 32, 10

Heapsort

step 1 | step 2

Heap sort the values

initialisation

P18 CSE Team Saviors

Array :-

0	1	2	3	4	5	6
0	56	8	75	95	32	10

1. Heap Initialization :-

Heapify subtree rooted at $\frac{n}{2} = \frac{6}{2} = 3$

swap 8 & 95

Heapify subtree rooted at 2

Heapify subtree rooted at 1

R18 CSE Team Saviors

2. Sort the values :-

R18 CSE Team Saviors

Sort. values = | 8 | 10 | 32 | 56 | 75 | 95 |

R18CSE Team Saviors

No imbalance.

4/03/17

B-Trees :-

→ AVL search trees and Red-Black trees give good performance when dictionary is small enough to reside in main memory.

→ For larger dictionaries that must reside on a disk, B-trees give good performance.

→ Dictionary elements are packed into disk blocks in ascending order.

→ If each disk block can accommodate 'm' elements, all the elements can be retrieved in one disk read.

→ The data structure that gives good performance for storing dictionaries in disk blocks is B-trees (and a variety of it called B+ tree).

R18 CSE Team Saviors

Scanned with CamScanner

→ B-tree is a balanced m-way search tree.

M-way search tree (Multi-way search tree) :-

→ An m-way search tree may be empty. If not, it is a tree that satisfies the following properties

(i) Every node may contain upto m child node pointers and m-1 elements.

(ii) Consider any node with 'p' elements. Let $K_1, K_2, \ldots K_p$ be the keys of these elements. The elements are ordered so that $K_1 < K_2 < \ldots < K_p$. Let $C_0, C_1, C_2, \ldots C_p$ be the p+1 child pointers of the node. The elements in the sub-tree with root C_0 have keys smaller than K_1, those in the subtree with root C_p have keys larger than K_p and those in the subtree with root C_i have keys larger than K_i but smaller than K_{i+1} where $1 \le i < p$

Properties of m-way search trees:-

1) Minimum no. of elements in an m-way search tree of height h is h.

2) Maximum no. of elements in an m-way search tree of height h is $m^h - 1$

Proof: Consider 3-way (m) search tree of height 1

R18 CSE Team Saviors

Scanned with CamScanner

$\therefore$ Total no. of elements $= m-1 + m(m-1) + m^2(m-1) + \dots + m^{h-1}(m-1)$

$= (m-1)(1 + m + m^2 + \dots + m^{h-1})$

$= \cancel{(m-1)} \dfrac{(m^h - 1)}{\cancel{(m-1)}}$

$= m^h - 1$

→ Eg:- 3-way search tree

3) Height of an m-way search tree of n elements ranges from $\log_m(n+1)$ to n

→ In actual applications, the height can be as large as 200. Hence, to achieve best performance, m-way search trees must be balanced.

27/03/19

B-Trees :-

→ B-Tree stands for balanced m-way search tree.

→ A B-tree of order 'm' is an m-way search tree which satisfies the following properties.

(i) A root node has atleast 2 child nodes.

(ii) All internal nodes other than the root have atleast $\lceil m/2 \rceil$ child nodes
→ ceiling value

(iii) All external nodes are at the same level. (A node which has ~~no~~ elements and no child node pointers is called as an external node).

R18 CSE Team Saviors

Scanned with CamScanner

Eg:

B-tree of order 3

→ B-trees of order 2 are full binary trees.

→ B-trees of order 3 are called as "2-3 trees" because internal nodes have either two or three child node pointers.

Eg:

→ Similarly, B-trees of order 4 are known as "2-3-4 trees".

*→ Start with empty ~~binary~~ 2-3 tree and insert the keys 20, 40, 30, 10, 25, 28, 27, 32, 36, 34, 35 in the given order. Draw 2-3 tree following each insert.

Sol: Step 1:- insert 20

0 reads + 1 write
= 1 disk access

Step 2:- insert 40

1 read + 1 write
= 2 disk accesses.

R18 CSE Team Saviors

Scanned with CamScanner

Step 3:- insert 30

1 read + 3 writes
= 4 disk accesses

Step 4:- insert 10

2 reads + 1 write
= 3 disk accesses

Step 5:- insert 25

2 reads + 3 writes
= 5 disk accesses

Step 6:- insert 28

2 reads + 1 write
= 3 disk accesses

R18 CSE Team Saviors

Scanned with CamScanner

Step 7:- Insert 27

2 reads + 5 writes
= 7 disk accesses

28/03/19

Step 8:- insert 32

3 reads + 1 write
= 4 disk accesses

Step 9:- insert 36

R18 CSE Team Saviors

Scanned with CamScanner

3 reads + 3 writes
= 6 disk accesses

Step 10:- insert 34

3 reads + 1 write
= 4 disk accesses.

Step 11:- insert 35

3 reads + 5 writes
= 8 disk accesses.

R19 CSE Team Saviors

Scanned with CamScanner

Procedure for insertion into B-trees:-

→ To insert an element into a B-tree, we first search for the presence of an element with the same key. If such an element is found, insert fails.

→ Else we attempt to insert the new element into the last internal node encountered.

→ If the new element goes into a full node, the full node becomes overfull and is split.

→ Let P be the full node. Insert the new element e together with a null pointer into P to get an overfull node with m elements and m+1 child pointers. Denote this overfull node as $m, c_0, (e_1, c_1) \ldots (e_m, c_m)$ → old p.

→ where e_is are the elements and c_is are the child node pointers. The node is split around the element e_d where $d = \lceil m/2 \rceil$. Elements to the left remain in P and those to the right move into a new node Q. The pair (e_d, Q) is inserted into parent of P. The formats of the new P and Q are as follows:

new P : $d-1, c_0, (e_1, c_1) \ldots (e_{d-1}, c_{d-1})$

Q : $m-d, c_d, (e_{d+1}, c_{d+1}) \ldots (e_m, c_m)$

R18 CSE Team Saviors

Scanned with CamScanner

Disk Accesses

→ In general, if an insertion causes 's' nodes to split, the number of disk accesses = $h + 2s + 1$ where h reads, 2s writes to write back 's' nodes that are split, 1 (write) modified root or the node into which the value is finally inserted and that node wouldn't become over-full.

→ No. of disk accesses needed in the worst case scenario is $3h + 1$ (where the splitting propagates to the root of the tree).

06/04/17

Deletion from B-Trees:-

→ The deletion of an element with key k from a B-tree of order m may trigger one of the following 4 cases.

Case 1: Key k belongs to a leaf node and its deletion doesnot result in the node having less than the minimum no. of elements.

Eg: B-tree of order 5

fig. 1

Eg: delete 80 from fig. 1

$d = \lceil \frac{m}{2} \rceil = 3$

Min. no. of elements $= d - 1 = 2$

fig. 2

R18 CSE Team Saviors

Scanned with CamScanner

Case 2: Key k belongs to a non-leaf node. In this case, replace k with the largest key (k') in its left subtree or the smallest key (k'') in its right subtree. k' or k'' is bound to occur in a leaf node and hence the deletion of k' or k'' may trigger one of the remaining cases.

Eg: delete 77 from fig. 2

fig. 3

Case 3: Deletion of key k from a ~~leaf~~ left node 'x' leaves it with less than minimum no. of elements. Elements are borrowed from either left or right sibling nodes. Hence move either k' or k'' depending on which node has elements to spare to the parent node. The intervening element p in the parent node is moved down to set right the vaccancy created by the deletion of key k in the node 'x'.

Eg: delete 70 from fig. 3.

fig. 4

R18 CSE Team Saviors

Scanned with CamScanner

Case 4: Deletion of key K from node X leaves it with less than minimum no. of elements and both its siblings don't have enough elements to spare. Node X is merged with one of the sibling nodes along with intervening element P in the parent node to create a new node. This in turn ~~falls~~ calls for the deletion of element P which may trigger one or more of the deletion cases.

Eg: delete 46 from fig. 4

fig. 5

→ Delete keys in the given order from the following B-tree of order 7.

delete: 10, 1, 70, 25

Sol: $m = \text{order} = 7$

$d = \left\lceil \frac{m}{2} \right\rceil = 4$

Min. no. of elements $= d - 1 = 3$

Step 1: delete 10 (case 2)

R18 CSE Team Saviors

Scanned with CamScanner

Step 2:- delete 1 (case 1)

Step 3:- delete 70 (case 3)

→ 3 disk reads + 3 disk writes

Step 4:- delete 25 (case 4)

(or)

R18 CSE Team Saviors

Scanned with CamScanner

→ Prove that new node created ~~has~~ in case 4 has enough space to hold all the elements after merging.

Proof:

Node 'x' has $d-2$ elements after deletion.

Both its siblings have $d-1$ elements.

No. of elements in new node $= d-2+d-1+1$

$= 2d-2$

$$d = \left\lceil \frac{m}{2} \right\rceil$$

$\Rightarrow d = \frac{m+1}{2}$ if m is odd $\Rightarrow m = 2d-1$

$d = \frac{m}{2}$ if m is even $\Rightarrow m = 2d$.

$\therefore$ No. of elements in new node $= 2d-2$

$= m-2$ if m is even

$= m-1$ if m is odd.

→ Delete 44 from the given 2-3 tree

Sol:-

2-3 tree $\Rightarrow m = 3$

$d = 2$ (min no

i.e., min. no. of elements $= 1$

R18 CSE Team Saviors

Scanned with CamScanner

07/04/17

Step 1:- deleting 44

Merging the node X with 35 & 40

after deleting 44:

Step 2:-

Merging the node Y with 20 & 30

Step 3:- Merging the node Z with 50 & 80

R18 CSE Team Savlors

Scanned with CamScanner

Discarding empty root node

R18 CSE Team Saviors

Scanned with CamScanner

① Insert the following keys into a B tree of order 5 : 32, 56, 22, 68, 12, 5, 8, 16, 27, 34, 47, 37, 6, 21, 57, 23, 44, 62, 55, 11, 2, 53

A :- Given $m = 5$, $d = \lceil \frac{5}{2} \rceil = 3$

Max no of child node pointers = 5

Max no of keys in any node = 4.

Step 1:- insert 32

Step 2:- insert 56

Step 3:- insert 22

Step 4:- insert 68

Step 5:- insert 12

overfull node, split at element 32

R18 CSE Team Saviors

Scanned with CamScanner

~~Step 7~~
insert 5 and 8

Step 8 :- insert 16

overfull node, split at element 12

~~Step~~

Step 9 :- insert 27 and 34
&
Step 10

R18 CSE Team Saviors

Scanned with CamScanner

Steps 11 & 12 insert 47 and 37

overfull node, split at 47.

Step

Steps 13 & 14 :- insert 6 and 21

Steps 15 & 16 :- insert 57 and 23

overfull node, split at 22

R18 CSE Team Saviors

Scanned with CamScanner

step 17 & 18 :- insert 44 & 62

step 19 & 20 :- insert 55 and 11

overfull node, split at 57

R18 CSE Team Saviors

Scanned with CamScanner

Welcome to Online Classes
on
Advanced Data Structures Through Java

Dr A Vani Vathsala

R18 CSE Team Savors

M Way Search Trees and B Trees

R18 CSE Team Saviors

Binary Trees and Binary Search Trees

- A binary tree is a tree in which every node can have a maximum of 2 child nodes

- A binary search tree is a binary tree in which all the **values in the left sub tree** of a node are **lesser** than the value in the node and **all the values in the right sub tree** are **greater** than the value in the node.

R18 CSE Team Saviors

Height Balanced Binary Search Trees

If the height of a binary tree can be maintained at **log n, where n is the number of values in the tree, then we call it as a height balanced tree.**

- Ex: AVL Search Trees and Red Black Trees

- While implementing dictionaries, maintaining **height of a tree at log n** is very important .
- **Searching time in a tree is directly proportional to the height of the tree**. Hence if height of a tree is maintained at around log n, searching time also can be maintained proportional to log n.

R18 CSE Team Saviors

M-ary trees

A rooted tree is called as **m-ary tree** if every node **has no more** than **m child node pointers.**

4 way tree

R18 CSE Team Saviors

m-ary trees are also called as m-way trees.

M way Search Trees

An m-way search tree is very similar to a binary search tree with reference to storing mechanism in the tree, but here the arity of the tree is m.

Ex :3 way search tree

Here **Maximum Number of key elements** in every node = **2**

And

Maximum Number of child node pointers in every node **= 3**

R18 CSE Team Saviors

M way Search Trees

Ex: 5 way search tree

R18 CSE Team Saviors

Here **Maximum Number of key elements** in every node =4

And

Maximum Number of child node pointers in every node **= 5**

M way Search Tree :Definition

An m-way search tree may be empty. If not, it is a tree that satisfies the following properties:

1. Every node may contain **upto m child node pointers and m-1 elements.**
2. Consider any node with p elements. Let K_1, K_2,…K_p be **the keys** of these elements. The elements are ordered so that $\mathbf{K_1 < K_2 < \ldots < K_p}$. Let $\mathbf{C_0, C_1, C_2, C_p}$ **be the p+1 child pointers** of the node. The elements in the sub tree with root C_0 have keys smaller than K_1, those in the subtree with root C_1 have keys larger than K_1, those in the subtree with root C_p have keys larger than Kp and those in the subtree **with root C_i have keys larger than K_i but smaller than K_{i+1}** where 1<=i<p

R18 CSE Team Saviors

M way Search Trees

Again, we face the same problem with m-way search tress, as with binary search trees:

If the keys are already sorted before insertion and if we construct an m-way search tree from them, we get either left skewed tree or right skewed tree.

Hence even m-way search trees need to be balanced to have better performance while searching, inserting and deleting operations**.**

B-tree is a balanced m-way search tree.

R18 CSE Team Saviors

B Tree: Definiton

B- Trees :-

→ B- Tree stands for balanced m-way search tree.

→ A B-tree of order 'm' is an m-way search tree which satisfies the following properties.

(i) A root node has atleast 2 child nodes.

(ii) All internal nodes other than the root have atleast $\lceil m/2 \rceil$ child nodes

→ ceiling value

(iii) All external nodes are at the same level. (A node which has ~~no~~ elements and no child node pointers is ~~called as~~ attached with an external node).

R18 CSE Team Saviors

B Tree: Ex

Insert the following keys into a B tree of order 5:

32,56,22,68,12,5,8,16,27,34,47,37,6,21,57,23,44,62,55,11,2,53

R18 CSE Team Saviors

AVL Trees :-

→ The complexity of search algorithm of BST, in the best case is $O(\log n)$ but worst case complexity is $O(n)$ where n is no. of nodes in the tree.

Hence BSTs cannot guarantee $O(\log n)$ performance for search, delete and insert operations, since they cannot ensure the search tree height to be always $O(\log n)$

→ Trees with worst case height of $O(\log n)$ are called Balanced trees.

→ One of the popular balanced trees is AVL tree which was introduced in 1962 by Adelson, Velskil, Landis

Eg:

AVL tree

not an AVL tree

$h_L = 3 \quad h_R = 1$

$h_L - h_R = 2$

→ AVL tree :- An empty binary tree is an AVL tree. If T is a non-empty binary tree with T_L, T_R as its left and right subtrees then T is an AVL tree iff (i) $|h_L - h_R| \leq 1$ where h_L & h_R are the heights of T_L and T_R respectively.

(ii) T_L & T_R are AVL trees themselves.

R18 CSE Team Saviors

Scanned with CamScanner

→ **AVL search trees:-** An AVL search tree is a binary search tree that is also an AVL tree.

Representation of an AVL tree:-

→ AVL trees are represented by using with doubly linked list scheme that is used for binary trees.

Balance factors:-

→ To facilitate insertion & deletion, a balance factor is associated with each node.

→ The balance factor b.f(x) of a node x is defined as: height of left subtree of x − height of right subtree of x.

→ According to definition of AVL search trees, the permissible balance factors are −1, 0 and 1.

Searching an AVL search tree:-

→ The search algorithm used for BST may be used for searching AVL search trees also.

Inserting into an AVL search tree:-

→ The insertion algorithm of BST cannot be used as it is for insertion into AVL search trees since resulting tree may no longer be an AVL search tree.

R18 CSE Team Saviors

Scanned with CamScanner

Insertion Algorithm:

→ To insert a node into an AVL search tree, the following steps are to be used.

Step 1: Insert the node into proper position following the properties of BST.

Step 2: Compute the balance factors:

On the path, starting from root node to the newly inserted node, compute the balance factors of each node. It can be seen that change in balance factors will occur only in this path.

Step 3: Decide the pivot node:

On the path traced in step 2, determine whether the balance factor of any node changes to 2 or -2. The node whose |balance factor| changes from 1 to 2 is called as pivot node.

There may be more than one node whose |balance factor| switches from 1 to 2. But the node nearest to the newly inserted node should be considered as the pivot node.

Step 4: Balance the unbalanced tree:

It is important to manipulate the pointer centered at the pivot node so as to bring back the tree into height balance. This pointer manipulation is known as rotation. There are 4 cases of rotation. They are LL, LR, RR, RL. Among these, LL & RR are called as single rotations & LR and RL are called as double rotations

R18 CSE Team Saviors

Scanned with CamScanner

Case (i)

LL Rotation :- Unbalance is due to insertion in the left subtree of the left child of the pivot node.

left child left subtree

→ To balance the tree, we perform LL rotation.

→ The following manipulations in pointers take place.

(i) The right subtree (A_R) of A becomes the left subtree of P

(ii) 'P' becomes the right child of A.

(iii) A_L remains the same.

Eg:

P18 CSE Team Saviors

Scanned with CamScanner

Case (2):

RR Rotation :- Unbalance is due to insertion in right subtree of right child of pivot node.

right child right subtree

→ To balance this tree, RR rotation is performed.

→ The following manipulations in pointers take place.

(i) The left subtree (B_L) of B becomes the right subtree of P.

(ii) P becomes left child of B

(iii) B_R remains the same.

Eg:

R18 CSE Team Saviors

Scanned with CamScanner

Case (B) :-

LR Rotation :- Unbalance is due to insertion in right subtree of left child of pivot node.

left child right subtree

→ To balance this tree, LR rotation is performed.

→ The following manipulations in pointers take place.

Rotation 1:

(i) B_L becomes the right subtree of A

(ii) A becomes the left child of B.

(iii) A_L remains the same

Rotation 2:

(i) B_R becomes the left subtree of P

(ii) P becomes the right child of B

(iii) P_R remains the same.

Eg:-

R18 CSE Team Saviors

Scanned with CamScanner

Eg:

R18 CSE Team Saviors

Scanned with CamScanner

Case (ii) :-

RL Rotation :- Unbalance is due to insertion in left subtree of right child of pivot node.

right child, left subtree

→ To balance this tree, RL rotation is performed.

→ The following manipulations in pointers take place.

Rotation 1 :

(i) B_R becomes ~~right~~ left subtree of A

(ii) A becomes right child of B

(iii) A_R remains the same.

Rotation 2 :

(i) B_L becomes right subtree of P

(ii) P becomes left child of B

(iii) P_L remains the same.

Eg

insert 10

R18 CSE Team Saviors

Scanned with CamScanner

RL rotation

→ Continued...

R18 CSE Team Saviors

Scanned with CamScanner

20/03/17

Deletion from AVL Trees :-

→ Let P be the ~~parent~~ node ~~of the physically~~ which has become imbalanced after deleting a node. If the deletion takes place from right subtree of P then bf(P) ~~increases by 1~~ changes from 1 to 2. Similarly if the deletion took place from left subtree of P then bf(P) ~~decreases by 1~~. changes from -1 to -2

→ Procedure for deletion from AVL search trees:

Step 1:- Use BST deletion ~~to~~ procedure to delete a given node from the AVL tree.

Step 2:- Calculate the balance factors of all nodes that fall on the path from the ~~dele~~ deleted node to the root node. If balance factors of all the nodes are either -1, 0 or 1, then the tree is balanced. Otherwise, the tree is imbalanced, and the following steps have to be used.

Step 3:- Let P be the node whose balance factor changes to 2 or -2 after deletion. In case of multiple such nodes, select the one which is closest to the deleted node.

R18 CSE Team Saviors

Scanned with CamScanner

Step 4:- To restore balance at P, the type of imbalance is classified into either type R imbalance or type L imbalance.

If the deletion took place from P's right subtree, then it is called as type R imbalance. Otherwise it is of type L imbalance.

Type R imbalance:- R_0, R_1, R_{-1}

R_0 imbalance:

Before deletion (h+2)

After deletion (h+2)

After rotation (h+2)

→ The type R_0 ~~case~~ refers to the case when the deletion took place from right subtree of P and bf(A) = 0. R_0 imbalance is balanced by performing R_0 rotation. R_0 rotation is similar to LL Rotation. The height of the tree before & after deletion is h+2. Hence the balance factors of the remaining nodes on the path from P to the root are unchanged.

R18 CSE Team Saviors

Scanned with CamScanner

As a result, the entire tree is rebalanced.

23/03/17

R_1 imbalance:

Before deletion
(h+2)

After deletion
(h+2)

R_1 rotation

After rotation
(h+1)

→ The type R_1 refers to the case wherein deletion took place from right subtree of P and bf(A) = 1. R_1 imbalance is balanced by performing R_1 rotation. R_1 rotation is similar to LL Rotation. But the new height of the subtree after rotation is h+1 which is 1 less than the height of the subtree before deletion. Hence 1 rotation will not suffice to restore the balance following a deletion. Rotations might be performed on the path from the deleted node to the root node wherever the balance factor becomes 2 or -2. The no. of rotations needed is O(log n).

P18CSE Team Saviors

Scanned with CamScanner

R_{-1} imbalance:

→ The type R_{-1} imbalance refers to the case where deletion is from right subtree of P and bf(A) = R_{-1} imbalance is balanced by R_{-1} rotation. R_{-1} rotation is similar to LR rotation. Even in this case height of the tree rooted at P reduces by 1 after rotation. Thus, if P has ancestors, their balance factors would ~~have~~ changed. Hence, more no. of rotations might have to be performed to balance the entire tree.

R18 CSE Team Saviors

Scanned with CamScanner

Type 1 imbalance:

→ It occurs when the deletion takes place from the left subtree of the pivot node. This imbalance is further sub classified into L_0, L_{-1} & L_1 imbalances. They are balanced by performing L_0, L_{-1}, L_1 rotations respectively.

Deletion Rotations:

R_0 $\xrightarrow{\text{similar to}}$ LL : one rotation is enough to balance the tree

R_1 ⟶ LL
R_{-1} ⟶ LR
} one or more rotations

L_0 ⟶ RR : one rotation

L_{-1} ⟶ RR
L_1 ⟶ RL
} one or more rotations.

R18 CSE Team Saviors

Scanned with CamScanner

Red Black Trees

Dr A Vani Vathsala

R18 CSE Team Saviors

Why Red-Black Trees?

Most of the **BST operations** (e.g., search, insert, delete.. etc) take **O(h)** time where h is the height of the BST.

The cost of these operations may become **O(n) for a skewed Binary tree**, where n is the number of nodes.

If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can **guarantee time complexity of O(Logn)** for all these operations.

The **height of a Red-Black tree is always O(Logn)** where n is the number of nodes in the tree.

Comparison with AVL Tree

AVL trees are more balanced compared to Red-Black Trees, but they **may cause more rotations** during insertion and deletion.

So if our application involves **many frequent insertions and deletions**, then **Red Black** trees should be preferred.

And if the insertions and deletions are less frequent and **search is a more frequent operation**, then **AVL tree should be preferred over Red-Black Tree.**

R18 CSE Team Saviors

Red Black Trees

A Red-Black tree is a **self-balancing binary search tree** in which each node contains an extra bit for denoting the color of the node, either red or black. (0 for black, 1 for Red)

A red-black tree satisfies the following properties:

Red/Black Property: Every node is colored, either red or black.

Root Property: The root is black.

Leaf Property: Every leaf (NIL Node) is black.

Red Property or Color Property: If a red node has children then, the children are always black nodes. (There cannot be two consecutive Red nodes on a path)

Depth Property: For each node, any simple path from this node to any of its descendant leaves has the same number of black nodes (also known as black-depth).

R18 CSE Team Saviors

EXAMPLE OF A RED-BLACK TREE

R18 CSE Team Saviors

Relation between RED-BLACK Trees and AVL Trees

All AVL Trees are Red Black Trees. But not all Red Black Trees are AVL Trees.

An example of a Red Black Tree that is not an AVL Tree

R18 CSE Team Saviors

Number of nodes from a node to its farthest descendant leaf is no more than twice as the number of nodes to the nearest descendant leaf.

Possible Red Black Trees

(1)

(2)

R18 CSE Team Savors

Not Red Black Trees

A Skewed tree containing chain of 3 nodes cannot be a Red Black Tree

R18 CSE Team Saviors

Insertion into RED BLACK Tree

The insertion operation in Red Black Tree is **similar to insertion operation in Binary Search Tree**.

But every node is **inserted with a color attached to it**.

After every insertion operation, we need to check all the properties of Red-Black Tree. If all the properties are satisfied then we go to next operation otherwise we perform any of the following operations to make it Red Black Tree.

1. **Recolor**
2. **Rotation**
3. **Rotation followed by Recolor**

The **insertion operation** in Red Black tree is performed using the following steps...

Step 1 - Check whether tree is Empty.

Step 2 - If tree is Empty then insert the **newNode** as Root node with color **Black** and exit from the operation.

Step 3 - If tree is not Empty then insert the newNode as leaf node with color **Red.**

Step 4 - If the parent of newNode is **Black** then exit from the operation.

Step 5 - If the parent of newNode is **Red** then check the color of parent node's sibling of newNode.

a) If it is colored **Red** then Recolor. Repeat the same until tree becomes Red Black Tree.

b) If it is colored **Black** or NULL then make suitable Rotation and Recolor it.

R18 CSE Team Saviors

Balancing RED-Black Trees during Insertions

- **The following notation is used to name** the keys while defining cases of balancing
    - **u** is the **new**ly inserted node.
    - **p** is the **parent** node of **u**.
    - **g** is the **grandparent** node of **u**.
    - **Un** is the **uncle** node of **u**. (Child of g, sibling of p)

- Insertion cases can be broadly classified in to two cases:
    - **Case a):** Colour of **Un** node is **Red**. This case has 4 subcasses depending on positions of u,p, and g. No Rotations are required in this case, Only Recoloring has to be done.
        - LLr, LRr, RRr, RLr

    - **Case b):** Colour of **Un** node is **Black or there is no Un node**. This case has 4 subcasses depending on positions of u,p, and g. Rotations are first done and then, recoloring has to be done if necessary.
        - LL, LR, RR, RL

R18 CSE Team Saviors

1) **LRr imbalance :** In this Red Black Tree violates its property in such a manner that **p** and **u** have **red** color at left and right positions with respect to grandparent **g** respectively. Therefore it is termed as Left Right imbalance**.** **r** in the case name represents **red** color for node Un.

R18 CSE Team Saviors

Removal of LRr imbalance can be done by:
Change color of p from red to black.
Change color of Un from red to black.
Change color of g from black to red, provided g is not a root node. But then, recheck for continuous red colored nodes up the tree till its root node.
Note: If given g is root node then there will be no changes in color of g.

2) LLr imbalance : In this Red Black Tree violates its property in such a manner that **p** and **u** have **red** color at left and left positions with respect to grandparent **g** respectively. Therefore it is termed as Left Left imbalance**.**

R18 CSE Team Saviors

Removal of LLr imbalance can be done by:

i.Change color of p from red to black.

ii.Change color of Un from red to black.

iii.Change color of g from black to red, provided g is not a root node. But then, recheck for continuous red colored nodes up the tree till its root node.

Note: If given g is root node then there will be no changes in color of g.

3) RRr imbalance : In this Red Black Tree violates its property in such a manner that **p** and **u** have **red** color at right and right positions with respect to grandparent **g** respectively. Therefore it is termed as Right Right imbalance**.**

R18 CSE Team Saviors

Removal of RRr imbalance can be done by:
Change color of p from red to black.
Change color of Un from red to black.
Change color of g from black to red, provided g is not a root node. But then, recheck for continuous red colored nodes up the tree till its root node.
Note: If given g is root node then there will be no changes in color of g.

4) RLr imbalance : In this Red Black Tree violates its property in such a manner that **p** and **u** have **red** color at right and left positions with respect to grandparent **g** respectively. Therefore it is termed as Right Left imbalance**.**

R18 CSE Team Saviors

Removal of RLr imbalance can be done by:
Change color of p from red to black.
Change color of Un from red to black.
Change color of g from black to red, provided g is not a root node. But then, recheck for continuous red colored nodes up the tree till its root node.
Note: If given g is root node then there will be no changes in color of g.

In the remaining four cases, color of node **Un is black or Un is not present.**

5) LL imbalance: It can be balanced by following two steps:

- Apply single rotation of p around g. (LL Rotation of AVL Tree)
- Recolor **p to black** and **g to red**.

R18 CSE Team Saviors

6) RR imbalance: It can be balanced by following two steps:

- Apply single rotation of p around g. (RR Rotation of AVL Tree)
- Recolor **p to black** and **g to red**.

R18 CSE Team Saviors

7) LR imbalance can be balanced by following steps:

- Apply double rotation of **u** around **p** followed by **u** around **g**. (LR Rotation of AVL Tree)
- Recolor **u to black** and **g to red.**

R18 CSE Team Saviors

8) RL imbalance can be balanced by following steps:

- Apply double rotation of **u** around **p** followed by **u** around **g**. (RL Rotation of AVL Tree)
- Recolor **u to black** and **g to red.**

R18 CSE Team Saviors

Red Black Tree Insertion Ex

- Create a Red Black Tree With the following nos: **8,18,5,15,17,25,40,80**

Step 1:- Insert 8

Root should be black

Step 2:- Insert 18

Any node, other than Root should be given Red color at the time of insertion

R18 CSE Team Saviors

Red Black Tree Insertion Ex

- Create a Red Black Tree With the following nos: **8,18,5,15,17,25,40,80**

R18CSE Team Saviors

Red Black Tree Insertion Ex

- Create a Red Black Tree With the following nos: **8,18,5,15,17,25,40,80**

R18 CSE Team Saviors

Red Black Tree Insertion Ex

- Create a Red Black Tree With the following nos: **8,18,5,15,17,25,40,80**

R18 CSE Team Saviors

Red Black Tree Insertion Ex

- Create a Red Black Tree With the following nos: **8,18,5,15,17,25,40,80**

In RR and LL Rotations, remember , **p is made black** And **g is made red**.

R18 CSE Team Saviors

Red Black Tree Insertion Ex

- Create a Red Black Tree With the following nos: **8,18,5,15,17,25,40,80**

R18CSE Team Saviors

Red Black Tree Insertion Ex

- Create a Red Black Tree With the following nos: **8,18,5,15,17,25,40,80**

In RR and LL Rotations, remember , **p becomes the new root of the subtree and is made black, a**nd **g is made red**.

In RL and LR Rotations, remember , **u becomes the new root of the subtree and is made black, a**nd **g is made red**.

Exercise: Create a Red Black Tree With the following nos: **10,3,5,2,6,1,16,14,25,13,11**

R18 CSE Team Saviors

Red Black Tree Deletion

- Like Insertion, even in deletion recoloring and rotations are used to maintain the Red-Black properties.
- The main property **that violates after insertion is two consecutive reds**.
- In imbalances during insert operation, **we check color of uncle** to decide the appropriate case.
- In **delete**, the main violated property is, **change of black height in subtrees as deletion of a black node may cause reduced black height in one root to leaf path.**
- In imbalances during delete operation, ***we check color of sibling*** to decide the appropriate case.

R18 CSE Team Saviors

Red Black Tree Deletion

Following are detailed steps for deletion.

Step 1) Perform standard BST delete.

When we perform standard delete operation in BST, we always end up **deleting a node which is either leaf or has only one child** (For an internal node, we copy the successor and then recursively call delete for successor, successor is always a leaf node or a node with one child).

So we only need to handle cases of deleting a node that is a leaf or has one child.

Let **v be the node to be deleted** and **u be the child that replaces v** (Note that u is NULL when v is a leaf and color of NULL is considered as Black).

Step 2) Recolor or Rotate/Recolor

Case 1: Simple Case: If either v or u is red, we mark the replaced child as black (In this case, there will be no change in black height). Note that both u and v cannot be red as v is parent of u and two consecutive reds are not allowed in red-black tree.

R18 CSE Team Saviors

v is leaf

v has one child node

Red Black Tree Deletion

Ex: Simple case

Simple case where either u or v is red

Change color of u from red to black.

Case 2) If Both u and v are Black.

When a black node is deleted and replaced by a black child, the child is marked as ***double black***. The main task now is to **convert this double black to single black**. **Color u as double black.**

Note : If v is leaf, and u is NULL , color of NULL is considered as black. So the **deletion of a black leaf also causes a double black.**

When 20 is deleted, it is replaced by a NULL, so the NULL becomes double black.
Note that deletion is not done yet, this double black must become single black

R18 CSE Team Saviors

Red Black Tree Deletion

Case 2 is further divided into following sub cases: **(In all these case Let sibling of node u be s.)**

2.1) If root is double black, just remove double black and keep it as black node.

In all the remaining cases, the current node **u is double black** and it is not root. The remaining cases are broadly classified into the following two cases:

2.2) Color of sibling s **is black:**

- **a)** both children of s **are also black: Only Recolor**
- **b)** at least one of sibling's children **is red. Has 2 subcases: Perform Rotation and Recolor in both the cases**
  - **i)** parent p of u **is black.**
  - **ii)** parent p of **u is red.**

2.3) Color of sibling s is red: Perform Single Rotation and Recolor

R18 CSE Team Saviors

Red Black Tree Deletion

2.2) Color of s is black:

a) both children of s are also black: Pass one black color from u to p, and make s to red. So if p was red, it becomes black and we can exit. But if **p was black**, it becomes double black. So we need to apply other cases to balance the tree.

In this case**, if parent was red**, then we can simply make it black.

R18 CSE Team Saviors

Red Black Tree Deletion : Example (delete 1,8,25,13,6)

R18 CSE Team Saviors

Red Black Tree Deletion : Example (delete 1,8,25,13,6)

Case : 2.2 a) : Sibling is black and both its children are also black

R18 CSE Team Saviors

Red Black Tree Deletion : Example (delete 1,8,25,13,6)

Case : 2.2 a) : Sibling is black and both its children are also black : Only Recolor

R18 CSE Team Saviors

Red Black Tree Deletion :
Example (delete 1,8,25,13,6)

delete 25 :- Replace with inorder predecessor

Replace 25 with 22

Same

When deleting a Red node, simply delete it

Case 1:-

Same

R18 CSE Team Saviors

Red Black Tree Deletion :
Example (delete 1,8,25,13,6)

R18 CSE Team Saviors

Red Black Tree Deletion

2.2 b) Current node u is double black and it is not root. **Sibling s is black and at least one of sibling's children is red.**

This case is further divided into two sub cases based on whether **p is black or p is red.**

i) p is black: Perform rotation and recolor.

Let the **red** child of **s** be **r**. This case can be divided in four sub cases depending upon positions of **s and r**

(i) LLbbr Case (s is left child of its parent and r is left child of s).

(ii)LRbbr Case(s is left child of its parent and r is right child).

(iii)RRbbr Case (s is right child of its parent and r is right child of)

(iv)RLbbr Case (s is right child of its parent and r is left child of s)

Ex for RRbbr Case

Change color of r from red to black.

R18 CSE Team Saviors

Red Black Tree Deletion

Ex for RLbbr Case : **Perform rotation.**

R18 CSE Tech Saviors

Red Black Tree Deletion

2.2 b) ii) Current node u is double black and it is not root. **Sibling s is black and at least one of sibling's children is red. and p is red: Perform rotation and recolor.**

Let the **red** child of **s** be **r**. This case can be divided in four sub cases depending upon positions of **s and r**

(i) LLrbr Case (s is left child of its parent and r is left child of s).
(ii)LRrbr Case(s is left child of its parent and r is right child).
(iii)RRrbr Case (s is right child of its parent and r is right child of)
(iv)RLrbr Case (s is right child of its parent and r is left child of s)

Ex for RRrbr Case

R18 CSE Team Saviors

Remember we are deleting from a valid RBT, so parent of **p** should have been black in color, we will not be having two consecutive red nodes even before rotation and after rotation.

Change color of p from red to black.
Change color of s from black to red
Change color of r from red to black

Red Black Tree Deletion

Ex for RLrbr Case : **Perform rotation and recolor.**

Hence change color of p from red to black.

R18 CSE Team Saviors

Red Black Tree Deletion

Case 2.3) **Sibling s is red:** **perform a rotation**, recolor

This case can be divided in two subcases.

(i) Left Case (s is left child of its parent). This is mirror of right right case shown in below diagram. We right rotate the parent p.

(ii) Right Case (s is right child of its parent). We left rotate the parent p.

Recolor **s to black and p's** **new child to red**

R18 CSE Team Saviors

Red Black Tree Deletion : Example (delete 1,8,25,13,6)

RBS CSE Team Saviors

Exercise: Red Black Tree Deletion : (delete 35, 40, 11, 20, 5, 12)

R18 CSE Team Saviors

14/08/20

5. Priority Queues

→ A priority queue is a collection of 0 or more elements wherein each element has a priority associated with it.

→ The operations that are performed on priority queues are find, insert and delete which are named as top, push and pop respectively.

→ Priority queues are implemented using maxheap or minheap.

Maxtree:

→ A maxtree is a binary tree in which value in each node is greater than or equal to those in its child nodes.

Eg:

→ maxtree but not a maxheap

Mintree:

→ A mintree is a binary tree in which value in each node is less than or equal to those in its child nodes.

Eg:

→ mintree but not a minheap.

R28 CSE Team Saviors

Scanned with CamScanner

Maxheap:

→ A maxheap is a maxtree that is also a complete binary tree.

Eg:

Minheap:

→ A minheap is a mintree that is also a complete binary tree.

Eg:

Operations on priority queues:

1) Insertion:

→ When a value has to be inserted into a maxheap containing n elements, it is first added at n+1 position and then the resulting tree is heapified.

or swapping
exchanging of nodes
to obtain the maxheap.

R18 CSE Team Saviors

Scanned with CamScanner

Eg. insert 56 in the following maxheap.

Step 1:- Add 56 at position 6

Step 2:- Heapify the resulting tree.

2) Deletion:

→ When an element is to be removed from maxheap, it is taken from root of the heap. The element from n^{th} node is copied into root node and the nth node is deleted. The resulting tree is then heapified.

R18 CSE Team Saviors

Scanned with CamScanner

Eg:

Step 1: Remove 80 from root node

Step 2: Put the value 35 from last node into root node, delete the last node.

Step 3: Heapify the resulting tree

(i) Compare and swap 35 with max (65, 56) if required

(ii) Compare and swap 35 with max (13, 26) if required

R18 CSE Team Saviors

Scanned with CamScanner

16/03/17

Heap Sort:

Heap Initialisation:

Eg:-1)

0	1	2	3	4	5	6
0	20	18	5	7	27	6

root

Step 1:

Heapify the subtree rooted at position 3 = $\frac{heapsize}{2}$

Step 2:

Heapify the subtree rooted at position $\left(\frac{heapsize}{2} - 1\right) = 2$

Step 3:

Heapify the subtree rooted at position 1

⇒

0	1	2	3	4	5	6
0	27	20	6	7	18	5

R18 CSE Team Saviors

Scanned with CamScanner

Eg:-2)

0	1	2	3	4	5	6	7	8	9	10
0	30	26	57	3	11	19	21	42	9	33

Step 1:- heapify the subtree rooted at position $\frac{10}{2} = 5$

Step 2 :- heapify the subtree rooted at position 4

Step 3:- heapify the subtree rooted at position 3

R18 CSE Team Saviors

Scanned with CamScanner

Step 4: heapify the subtree rooted at position 2

Step 5: heapify the subtree rooted at position 1

0	1	2	3	4	5	6	7	8	9	10
0	57	42	30	26	33	19	21	3	9	11

R18 CSE Team Saviors

Scanned with CamScanner

→ Heap Sort:

Eg:-1)

	1	2	3	4	5	6
a	0	27	20	6	7	18	5

Step 1:- delete 27 and copy it in a[6]

0	5	20	6	7	18	27

0	20	5	6	7	18	27

0	20	18	6	7	5	27

Step 2:- delete 20 and copy it in a[5]

0	5	18	6	7	20	27

0	18	5	6	7	20	27

0	18	7	6	5	20	27

Step 3:- delete 18 and copy it in a[4]

0	5	7	6	18	20	27

0	7	5	6	18	20	27

Step 4:- delete 7 and copy it in a[3]

0	6	5	7	18	20	27

R18 CSE Team Saviors

Scanned with CamScanner

Step 5:- delete 6 and copy it in a[2]

Step 6:- delete 5 and copy it in a[1]

→ This step is not required.

* Program to implement heapsort.

↳ Refer lab-12, program-1.

17/03/19

Heap initialization:-

→ As heap is a complete binary tree, a heap can be efficiently represented using an array.

→ To represent a heap of size n, an array of size n+1 is used.

→ Heap initialization is a process of constructing a heap from given list of elements.

→ In order to construct a heap, we begin with heapifying the subtree rooted at position heapsize/2.

→ We proceed by heapifying the subtrees rooted at position i-1, i-2, ... upto 1 where $i = \frac{heapsize}{2}$

R18 CSE Team Saviors

Scanned with CamScanner

Heap Sort:-

→ To apply heap sort to a set of n values stored in array a, following steps are to be used.

(1) Construct maxheap from the given values using the above described heap initialization process.

(2) For each position i starting from n down to 2, apply the following steps.

(i) swap i^{th} element with root element.

(ii) heapify the binary tree represented by a[1] to a[i-1]

→ Continued..

R18 CSE Team Saviors

Scanned with CamScanner

Pattern Matching and Tries

R18 CSE Team Saviors

Pattern Matching

An **alphabet ∑** is the set of possible characters for a family of strings. Example of alphabets:

- ASCII (used by C and C++)
- Unicode (used by Java)
- {0, 1}
- {A, C, G, T}

Let **P** be a string of size **m:**

- A **substring P**[**i .. j**] of **P** is the subsequence of **P** consisting of the characters with indexes between **i** and **j.**
- A **prefix** of **P** is a substring of the type **P**[0 **.. i**].
- A **suffix** of **P** is a substring of the type **P**[**i ..m** - 1].

- Given strings **T** (text) and **P** (pattern), the pattern matching problem consists of finding a substring of **T** equal to **P.**

Applications: Text editors, Search engines , Biological research

R18 CSE Team Saviors

Pattern Matching Algorithms

1. Brute Force algorithm
2. Boyer-Moore algorithm1
3. Knuth-Morris Pratt algorithm

R18 CSE Team Saviors

Notations

- T refers to the main string or Text
- P refers to pattern
- n refers to length of main string
- m refers to length of pattern

R18 CSE Team Saviors

Brute –force pattern matching

- The brute-force pattern matching algorithm compares the pattern ***P*** with the text ***T*** for each possible shift of ***P*** relative to ***T***, until either a match is found, or all placements of the pattern have been tried
- Brute-force pattern matching runs in time ***O***(***nm***)
- Example of worst case:
- ***T*** = ***aaa … ah***
- ***P*** = ***aaah***

R18 CSE Team Saviors

Brute-Force Pattern Matching

| | | |

Algorithm for BruteForce Pattern Matching

Algorithm ***BruteForceMatch***(***T, P***)

Input text ***T*** of size ***n*** and pattern ***P*** of size ***m***

Output starting index of a substring of ***T*** equal to ***P*** or −1if no such substring exists

```
for i ← 0 to n − m
        j ← 0
        while j < m ∧ T[i + j] = P[j]
                j ← j + 1

        if j = m
        return i {match at i}
        else
        break while loop {mismatch}
return -1 {no match anywhere}
```

R18 CSE Team Saviors

Boyer-Moore Heuristics

- The Boyer-Moore's pattern matching algorithm is based on two heuristics

  Looking-glass heuristic: Compare P with a subsequence of T moving backwards

  Character-jump heuristic: When a mismatch occurs at $T[i] = c$
  - If P contains c, shift P to align the last occurrence of c in P with $T[i]$
  - Else, shift P to align $P[0]$ with $T[i+1]$
- Example

R18 CSE Team Saviors

Example

R18 CSE Team Saviors

In Boyer-Moore , if a mismatch is found, and the mismatched character of the text does not occur in the pattern, we shift the entire pattern beyond that location, as originally illustrated in the following Figure

R18 CSE Team Saviors

→Let i be the index of Text, k be the index of pattern

→i(index of Text) is to be incremented by m and then k(index of pattern) is to be set to m-1.

In Boyer-Moore , If the mismatched character occurs elsewhere in the pattern, we must consider two possible subcases depending on whether its last occurrence is before or after the character of the pattern that was mismatched. Those two cases are illustrated in Figure.

case: $j < k$

Figure A.

case: $j > k$

Figure B.

R18 CSE Team Saviors

Figure . Additional rules for the character-jump heuristic of the Boyer-Moorealgorithm. We let *i* represent the index of the mismatched character in the text, *k* represent the corresponding index in the pattern, and *j* represent the index of the last occurrence of text[*i*] within the pattern. We distinguish two cases: (a) $j < k$,in which case we shift the pattern by $k - j$ units, and thus, index *i* advances by $m-(j+1)$ units; (b) $j > k$, in which case we shift the pattern by one unit, and index *i* advances by $m-k$ units.

The efficiency of the Boyer-Moore algorithm relies on quickly determining where a mismatched character occurs elsewhere in the pattern. Last-Occurrence function is used for that.

Last-Occurrence Function:

- Boyer-Moore's algorithm preprocesses the pattern **P** and the alphabet ∑ to build the last-occurrence function **L** mapping ∑ to integers.
- **L(c)** is defined as the largest index **i** such that **P[i]** = **c** or -1 if no such index exists.

  Example: ∑ = {**a, b, c, d**} and **P** = **abacab**

0	1	2	3	4	5
a	b	a	c	a	b

c	a	b	c	d
last(c)	4	5	3	−1

R18 CSE Team Saviors

Algorithm BoyerMoore(T, P, ∑):

```
L = lastOccurenceFunction(P)
i = m – 1, k= m - 1
do
{     if (T[i] == P[k])
        {    if (k== 0)
                 return i;                    // match at i
             else
             {       i = i – 1;
                     k = k – 1;
             }
        }
        else
        {          // character-jump
                 j = L[T[i]];
                 i = i + m – min(k, j + l);
                 k = m – 1;
        }
   }while( i > n – 1);
return –1;                                    // no match
```

R18 CSE Team Saviors

Performance of Boyer Moore

If using a traditional lookup table, the worst-case running time of the Boyer-Moore algorithm is $O(nm+|S|)$. The computation of the last function takes $O(m+|S|)$ time. The actual search for the pattern takes $O(nm)$ time in the worst case—the same as the bruteforce algorithm. An example that achieves the worst case for Boyer-Moore is

text $\{aaaaaa\cdots a\}$

pattern = $\{baa\cdots a\}$

The worst-case performance, however, is unlikely to be achieved for English text;in that case, the Boyer-Moore algorithm is often able to skip large portions of text. Experimental evidence on English text shows that the average number of comparisons done per character is 0.24 for a five-character pattern string.

R18 CSE Team Saviors

```
// Java code Boyer Moore
import java.util.HashMap;
import java.util.Scanner;
class BMP{
int m, n;
HashMap<Character , Integer> Last;
BMP(){
  Last=new HashMap<Character, Integer>();
}
void LastOccurence(char[] T, char[] P)
{ for(int i=0;i<n;i++)
        Last.put(T[i], -1);
   for(int k=0;k<m;k++)
        Last.put(P[k], k);

}
```

R18 CSE Team Saviors

```
int bm(char[] T , char[] P)
{   int j,i,k;
    n=T.length; m=P.length; i=m-1; k=m-1;
    LastOccurence(T, P);
    while(i<n)
   {
        if(T[i]==P[k])      {
              if(k==0)return i;
               k--;       i--;     }
       else
         { j=Last.get(T[i]);
           if(j==-1)         i=i+m;
           else if(j<k)     i=i+m-(j+1);
           else       i=i+m-k;
           k=m-1;
          }  // else close
}//while close       return -1;
 }// function close
```

R18 CSE Team Saviors

```
public static void main(String[] args){
   Scanner s1=new Scanner(System.in);
   String T,P;

   System.out.println("Enter main and substring");
   T=s1.nextLine();
   P=s1.next();
   BMP Bm=new BMP();
   int pos= Bm.bm(T.toCharArray(),P.toCharArray());

  if(pos==-1)
    System.out.println("Pattern not found");
  else
    System.out.println("Pattern found at "+ pos +“  position");
}//min close
}//class close
```

R18 CSE Team Saviors

Knuth Morris Pratt (KMP) Algorithm:

- The main idea of the KMP algorithm is to precompute self-overlaps between portions of the pattern so that when a mismatch occurs at one location, we immediately know the maximum amount to shift the pattern before continuing the search.
- Knuth-Morris-Pratt's algorithm compares the pattern to the text in **left-to-right**, but shifts the pattern more intelligently than the brute-force algorithm.
- When a mismatch occurs, what is the most we can shift the pattern so as to avoid redundant comparisons?

  The largest prefix of **P**[0..**j**] that is a suffix of **P**[1..**j**].

R18 CSE Team Saviors

For example, consider the Pattern: **ababaca**

- The prefix which is also the suffix is called **border.**
- Various sub patterns and their possible suffixes, prefixes and borders and their lengths are given here.

Sub pattern	Possible Preffixes	Possible Suffixes	borders	Border length
a	NIL	NIL	NIL	0
ab	a	b	NIL	0
aba	a , ab	a , ba	a	1
abab	a , ab , aba	b , ab , bab	ab	2
ababa	a , ab , aba , abab	a , ba , aba , baba	aba	3
ababac	a , ab , aba , abab, ababa	c , ac , bac , abac ,babac,	NIL	0
ababaca	a , ab , aba , abab, ababa, ababac,	a, ca, aca, baca , abaca ,babaca,	a	1

R18 CSE Team Saviors

Border length defines number of characters of the pattern to be skipped. Failure function is used to calculate the border length.

KMP Failure Function:

- Knuth-Morris-Pratt's algorithm preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself.

- The **failure function F(j)** is defined as the size of the largest prefix of **P**[0..**j**] that is also a suffix of **P**[1..**j**].

k	0	1	2	3	4	5	6	7	8	9	10	11
$P[k]$	a	m	a	l	g	a	m	a	t	i	o	n
$f(k)$	0	0	1	0	0	1	2	3	0	0	0	0

R18 CSE Team Saviors

Knuth-Morris-Pratt's algorithm modifies the brute force algorithm so that if a mismatch occurs at **P**[**j**] = **T**[**i**] we set **j** = **F**(**j** - 1).

Failure function:

k	0	1	2	3	4	5
pattern[k]	a	b	a	c	a	b
$f(k)$	0	0	1	0	1	2

R18 CSE Team Saviors

Algorithm KMPMatch(T, P)

```
F = failureFunction(P)
i = 0, j = 0
while (i < n)
{           if (T[i] == P[j])
            {    if (j = = m – 1)
                         return i – j;                    // match
                 else
                 {          i = i + 1;
                            j = j + 1;
                 }
            }
          else
         { if (j > 0)
                         j = F[j - 1];
                       else
                         i = i + 1;
          }
}
return -1;                                  //no match
```

R18 CSE Team Saviors

Computing the Failure Function:

- The failure function can be represented by an array and can be computed in **O**(**m**) time. The construction is similar to the KMP algorithm itself.
- At each iteration of the while loop, either **i** increases by one, or the shift amount **i** – **j** increases by at least one (observe that **F**(**j** - 1) < **j**).
- Hence, there are no more than 2**m** iterations of the while-loop.

Algorithm failureFunction(**P**):

```
F[0] = 0, i = 1, j = 0;
while (i < m)
{           if (P[i] == P[j])
        {                   F[i] = j + 1; //we have matched j + 1 chars
                             i = i + 1;
                             j = j + 1;
        }
        else if (j > 0)  //use failure function to shift P
              j = F[j - 1];
         else
         { F[i] = 0;                    // no match
                         i = i + 1;
          }
}
```

R18 CSE Team Saviors

- At each iteration of the whileloop, either ***i increases by one, or*** the shift amount ***i – j*** increases by at least one
- Hence, there are no more than 2***n iterations of the*** while-loop. Thus, KMP's algorithm runs in optimal time ***O(m + n).***

R18 CSE Team Saviors

Example: compute F for the pattern ‘p’ below:

p							
	a	b	a	b	a	c	a

Initially: m = length[p] = 7
F[0] = 0
j= 0

Step 1: i = 1, j=0, since P[i]!=P[j]
F[1] = 0

I	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0					

Step 2: i = 2, j= 0, since P[i]=P[j]
F[2] = 1

i	0	1	2	3	4	5	6
p	a	b	a	b	a	c	a
f	0	0	1				

Step 3: i = 3, j = 1, since P[i]=P[j]
F[3] = 2

i	0	1	2	3	4	5	6
p	a	b	a	b	a	c	a
F	0	0	1	2			

P18 CSE Team Saviors

Step 4: i = 4, j =2, since P[i]=P[j]

F[4] = 3

i	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2	3		

Step 5: i = 5, j = 3, since P[i]!=P[j]

j=F[j-1]=F[2] =1

i=5 , j=1 , since P[i]!=P[j]

j=F[j-1]=F[0]=0

Therefor F[5]=0

i	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2	3	0	

Step 6: i = 6, j = 0

F[6] = 1

i	0	1	2	3	4	5	6
p	a	b	a	b	a	c	a
F	0	0	1	2	3	0	1

After iterating 6 times, the Failur function computation is complete:
→

i	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2	3	0	1

R18 CSE Team Saviors

Illustration: given a String ‘S’ and pattern ‘p’ as follows:

S

b	a	c	b	a	b	a	b	a	b	a	c	a	c	a

p

a	b	a	b	a	c	a

Let us execute the KMP algorithm to find whether ‘p’ occurs in ‘S’.

For ‘p’ the prefix function, F was computed previously and is as follows:

i	0	1	2	3	4	5	6
P	a	b	A	b	a	c	a
F	0	0	1	2	3	0	1

R18 CSE Team Savors

Initially: n = size of S = 15;
m = size of p = 7

Step 1: i = 0, j = 0
comparing p[0] with S[0]

P[0] does not match with S[0]. 'p' will be shifted one positioto the right.

Step 2: i = 1, j = 0
comparing p[0] with S[1]

P[0] matches S[1]. Since there is a match, p is not shifted.

R13 CSE Team Saviors

Step 3: i = 2, j = 1

Comparing p[1] with S[2] p[1] does not match with S[2]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

Backtracking on p set j=F[J-1]=F[0]=0, comparing p[0] and S[2] since does not macthc and j=0 set i=i + 1

Step 4: i = 3, j = 0 comparing p[0] with S[3] p[0] does not match with S[3] since j=0 set i=i+1

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

Step 5: i = 4, j= 0 comparing p[0] with S[4] p[0] matches with S[0]so set i=i+1 ,j=j+1

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

RIT CSE Team Saviors

Step 6: i = 5, j= 1

Comparing p[1] with S[5] p[1] matches with S[5], set i=i+1 j=j+1

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

Step 7: i = 6 j = 2

Comparing p[2] with S[6] p[2] matches with S[6] ,set i=i+1 , j=j+1

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

Step 8: i = 7 j = 3

Comparing p[3] with S[7] p[3] matches with S[7], set i=i+1 ,j=j+1

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

R18 CSE Team Saviors

Step 9: i = 8, j = 4

Comparing p[4] with S[8] p[4] matches with S[8], set i=i+1, j=j+1

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

Step 10: i = 9 , j= 5

Comparing p[5] with S[9] p[9] doesn't match with S[9] set j=F[j-1]=F[4]=3,

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

Backtracking on P set j=3, comparing p[3] with S[9] because after mismatch j = F[4] = 3
P[3] matches to S[9] so set i=i=1, j=j+1

Step 11: i = 10, j = 4

Comparing p[4] with S[10] p[4] matches with S[10] so set i=i+1 ,j=j+1

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

P18 CSE Team Saviors

Step 12: i = 11, j = 5

Comparing p[5] with S[11] p[5] matches with S[11] set i=i+1 ,j=j+1

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p							a	b	a	b	a	c	a		

Step 13: i = 12, qj= 6

Comparing p[6] with S[12] p[6] matches with S[12] pattern is matched

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p							a	b	a	b	a	c	a		

R18 CSE Team Saviors

Pattern ‘p’ has been found to completely occur in string ‘S’. Pattern position is i.j = 12-6 =6

Tries

R18 CSE Team Saviors

A ***trie*** (pronounced “try”) is a tree-based data structure for storing strings in order to support fast pattern matching.

The main application for tries is in information retrieval.

This approach is suitable for applications in which many queries are performed on a fixed text, so that the initial cost of preprocessing the text is compensated by a speedup in each subsequent query

we can use a trie to perform a special type of pattern matching, called ***word matching***, where we want to determine whether a given pattern matches one of the words of the text exactly.

Word matching differs from standard pattern matching because the pattern cannot match an arbitrary substring of the text—only one of its words.

R18 CSE Team Saviors

Standard Trie

Let *S* be a set of *s* strings from alphabet S such that no string in *S* is a prefix of another string. A ***standard trie*** for *S* is an ordered tree *T* with the following properties :

- Each node of *T*, except the root, is labeled with a character of S.
- The children of an internal node of *T* have distinct labels.
- *T* has *s* leaves, each associated with a string of *S*, such that the concatenation of the labels of the nodes on the path from the root to a leaf *v* of *T* yields the string of *S* associated with *v*.

In addition to the bove ,a standard trie storing a collection *S* of *s* strings of total length *n* from an alphabet S has the following properties:

- The height of *T* is equal to the length of the longest string in *S*.
- Every internal node of *T* has at most |S| children.
- *T* has *s* leaves.
- The number of nodes of *T* is at most *n*+1.

The worst case for the number of nodes of a trie occurs when no two strings share a common nonempty prefix; that is, except for the root, all internal nodes have one child.

R18 CSE Team Saviors

Example :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
s	e	e		a		b	e	a	r	?		s	e	l	l		s	t	o	c	k	!

23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45
	s	e	e		a		b	u	l	l	?		b	u	y		s	t	o	c	k	!

46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68
	b	i	d		s	t	o	c	k	!		b	i	d		s	t	o	c	k	!	

69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88
h	e	a	r		t	h	e		b	e	l	l	?		s	t	o	p	!

(a)

(b)

Figure Word matching with a standard trie:
(a) text (articles and prepositions, which are also known as ***stop words***, excluded)
(b) standard trie for the words in the text, with leaves augmented with indications of the index at which the given word begins in the text. For example, the leaf for the word "stock" notes that the word begins at indices 17, 40, 51, and 62 of the text.

R18 CSE Team Saviors

Compressed Tries

A ***compressed trie*** is similar to a standard trie but it ensures that each internal node in the trie has at least two children.

It enforces this rule by compressing chains of single-child nodes into individual edges. Let *T* be a standard trie.

We say that an internal node *v* of *T* is ***redundant*** if *v* has one child and is not the root. For example, the trie of Figure 13.7 has eight redundant nodes.

Compressed trie, is also known (for historical reasons) as the ***Patricia trie***

A compressed trie storing a collection *S* of *s* strings from an alphabet of size *d* has the following properties:

- Every internal node of *T* has at least two children and at most *d* children.
- *T* has *s* leaves nodes.
- The number of nodes of *T* is $O(s)$.

R18 CSE Team Saviors

Compressed Trie Example

Figure Compressed trie for the strings {bear, bell, bid, bull, buy, sell, stock, stop}.

R18 CSE Team Saviors

One of the primary applications for tries is for the case when the strings in the collection *S* are all the suffixes of a string *X*. Such a trie is called the ***suffix trie*** (also known as a ***suffix tree*** or ***position tree***) of string *X*. For example, Following Figure shows the suffix trie for the eight suffixes of string "minimize."

R18 CSE Team Saviors

→Using a suffix trie allows us to save space over a standard trie by using several space compression techniques, including those used for the compressed trie.

→The compact representation of a suffix trie *T* for a string *X* of length *n* uses *O*(*n*) space.

Using a Suffix Trie:

The suffix trie *T* for a string *X* can be used to efficiently perform pattern-matching queries on text *X*. Namely, we can determine whether a pattern is a substring of *X* by trying to trace a path associated with *P* in *T*. *P* is a substring of *X* if and only if such a path can be traced. The search down the trie *T* assumes that nodes in *T* store some additional information, with respect to the compact representation of the suffix trie:

If node *v* has label j..k and *Y* is the string of length *y* associated with the path from the root to *v* (included), then $X[k-y+1..k] = Y$.
This property ensures that we can compute the start index of the pattern in the text when a match occurs in $O(m)$ time.

R18 CSE Team Saviors