

# Unit III - Dictionaries:

Introduction: Dictionary definition, Dictionary ADT.

Dictionaries implementation-I :

**Linear List Representation:** Basics of linear list, implementation of sorted list using user defined generic classes and, LinkedList Collections class.

**Hashing:** basics, closed hashing – linear probing, quadratic probing, double hashing, rehashing, extendible hashing and their implementation, open hashing-separate chaining and its implementation using user defined generic classes.

**Binary Search Trees:** definition and basics, implementation of operations-searching, non-recursive traversals, insertion and deletion using user defined generic classes.

## Hashing

### Introduction

- We've all used hashing in our daily lives; remember how in university, we were all identified by a unique number called our Roll number? In this case, the Roll number serves as the key, and the student details (name, address, course, age, etc.) serve as the values, and we can easily store this data in an array where the Roll number serves as the index to store the associated details.
- However, Hashing can be used when the keys are large or non-integer and cannot be used directly as an index. Hashing in java is primarily the process of converting a given key (roll no) into another value (array indexable integer in this case).

### **What is Hashing in Java?**

**Hashing** in Java is the technique that enables us to store the data in the form of **key-value pairs**, by modifying the original key using the hash function so that we can use these modified keys as the

index of an array and store the associated data at that **index location** in the Hash table for each key.

## Hash Function

While implementing hashing in java it uses a function called hash function, it is the most important part of hashing; it transforms supplied keys into another **fixed-size value (hash-value)**. The value returned by a hash function is called hash value, hash code, or simply hashes.

Basically, the given keys are converted into hash values using a hash function and these hash values are used as the index of the hash table to store the associated data.

Hashing in java can be termed as the entire process of storing data in a hash table in the form of key-value pairs, with the key computed using a hash function.

## Characteristic of a Hashing Algorithm

The hashing algorithm is used for generating fixed-length numeric hash-value from the input keys. We anticipate our hash function to have the following features because we created hashing for **quick and efficient core** operations.

- It should be very fast on its computation and should convert the given key into a hash value quickly.
- The algorithm should avoid generating a hash value from a message's (input) generated hash value (one way).
  - 
  - Hashing algorithm **must avoid the collision**. Actually, a collision occurs when two differed inputs to the hash function are converted to the same hash value.
- When the message's hash value changes even slightly, the message's hash value must change as well. The avalanche effect is what it's called.

## How does Hashing Work?

Hashing in java is a **two-step** process:

- firstly, A hash function is used to turn an **input key into hash values**.
- This hash-value is used as an index in the hash table and corresponding data is stored at that location in the table. The element is stored in a hash table and can be retrieved quickly using the hashed key. In order to retrieve data from the hash table, we first calculate the hash value for a given key and as we know this key was used as an index while storing data in the hash table so we extract the data from that index.

To better understand how hashing works, consider the following scenario: we are given the token number and the name of the people who will be vaccinated today. The person is uniquely identified by token no.

The table below shows contains the name and token numbers as stated above.

<b>Token No.</b>	<b>Name</b>
16	Virat
1	Alex
40	Ishika
5	Sonu
3	Mrinalini
38	John

Our goal is to create a quick and space-efficient hash table for data storage and retrieval.

To solve this problem, one naive solution we can think of is to use an array of **size 41** so that we can be able to use each key(token no) as the index to the array and store the data at those index locations.

This works but it is inefficient and we will be wasting the majority of the space we have used because we will be having data stored at only **six (1,5,3,16,38 and 40)** locations out of **41**. we should think of a method to narrow down the search space for us.

In such cases, we can use hashing, We can use think of a function that is able to convert the given keys (token no) into less spread

hashed keys and follows all the characteristics of the hash function discussed in the section above.

### Choosing hash function

If we look closely at the keys, we can see that they can easily be converted to numbers from 0 to 10 if we use

$$\text{Hash}(\text{key}) = \text{key} \% 10.$$

Using this hash function we can observe that

- **Hash(16) = 16%10 = 6**, indicating that the value corresponding to **key 16** i.e. (virat) will be stored in the array at **index 6**.
- Similarly other keys can be hashed in the same way to find a suitable location in the array.
- Our hash table should be of **size 10** because the hash function can be able to give hash values from **0 to 9**.

Token No.	Name	Hash(key) = key%10
16	Virat	16%10 = 6
1	Alex	1%10 = 1
40	Ishika	40%10 = 0
5	Sonu	5%10 = 5
3	Mrinalini	3%10 = 3
38	John	38%10 = 8

So our hash-table will look like:

Index	0	1	2	3	4	5	6	7	8	9
Value	Ishika	Alex		Mrinalini		Sonu	Virat		John	

Now if we want to be able to get the name of the person with token no 38, then we can generate an index using the hash function we used above.

$$\text{index} = \text{hash}(38) = 38 \% 10 = 8.$$

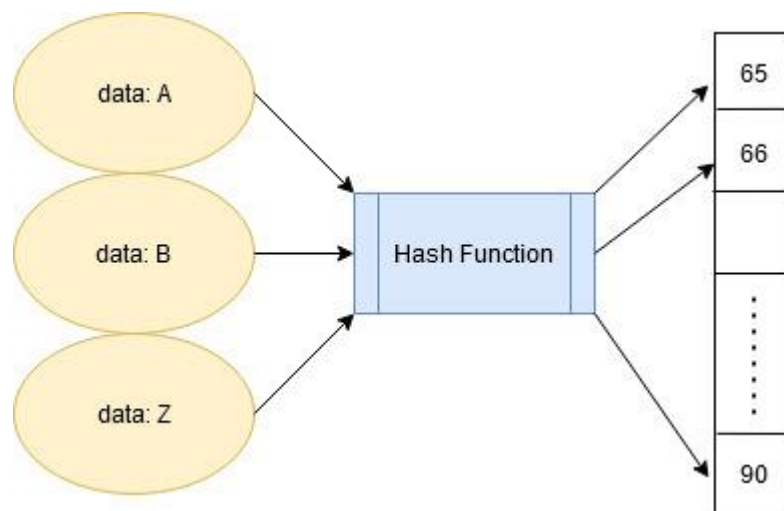
and we can get the value from the array now, table[8] will return John as the name of that person.

Now suppose we have one more person's data with token no 23, in that case, the hash function we used above will generate the same hash values for both **23 and 3**. This is what we call collision when two different keys are converted to the same hash value by the hash function. We have no. of techniques to resolve **collision**.

**Hashing** is the process of transforming data and mapping it to a range of values which can be efficiently looked up.

## **Collision resolution techniques such as:**

- Open Hashing (Separate chaining)
- Closed Hashing (Open Addressing)
  - Liner Probing
  - Quadratic probing
  - Double hashing



- *Hash table*: a data structure where the data is stored based upon its hashed key which is obtained using a hashing function.
- *Hash function*: a function which for a given data, outputs a value mapped to a fixed range. A hash table leverages the hash function to efficiently map data such that it can be retrieved and updated quickly. Simply put, assume  $S = \{s_1, s_2, s_3, \dots, s_n\}$  to be a set of objects that we wish to store into a map of size  $N$ , so we use a hash function  $H$ , such that for

all  $s$  belonging to  $S$ ;  $H(s) \rightarrow x$ , where  $x$  is guaranteed to lie in the range  $[1, N]$

- *Perfect Hash function*: a hash function that maps each item into a unique slot (no collisions).

**Hash Collisions:** As per the Pigeonhole principle if the set of objects we intend to store within our hash table is larger than the size of our hash table we are bound to have two or more different objects having the same hash value; a hash collision. Even if the size of the hash table is large enough to accommodate all the objects finding a hash function which generates a unique hash for each object in the hash table is a difficult task. Collisions are bound to occur (unless we find a perfect hash function, which in most of the cases is hard to find) but can be significantly reduced with the help of various collision resolution techniques.

Following are the collision resolution techniques used:

- Open Hashing (Separate chaining)
- Closed Hashing (Open Addressing)
  - Liner Probing
  - Quadratic probing
  - Double hashing

### 1. Open Hashing (Separate chaining)

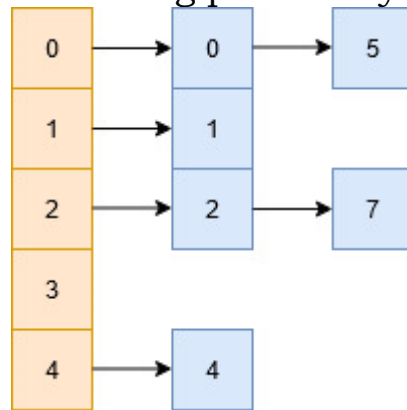
Collisions are resolved using a list of elements to store objects with the same key together.

- Suppose you wish to store a set of numbers =  $\{0, 1, 2, 4, 5, 7\}$  into a hash table of size 5.
- Now, assume that we have a hash function  $H$ , such that  $H(x) = x \% 5$
- So, if we were to map the given data with the given hash function we'll get the corresponding values

- $H(0) \rightarrow 0 \% 5 = 0$
- $H(1) \rightarrow 1 \% 5 = 1$
- $H(2) \rightarrow 2 \% 5 = 2$
- $H(4) \rightarrow 4 \% 5 = 4$
- $H(5) \rightarrow 5 \% 5 = 0$

- $H(7) \rightarrow 7\%5 = 2$

- Clearly 0 and 5, as well as 2 and 7 will have the same hash value, and in this case we'll simply append the colliding values to a list being pointed by their hash keys.



Obviously in practice the table size can be significantly large and the hash function can be even more complex, also the data being hashed would be more complex and non-primitive, but the idea remains the same. This is an easy way to implement hashing but it has its own demerits.

- The lookups/inserts/updates can become linear  $[O(N)]$  instead of constant time  $[O(1)]$  if the hash function has too many collisions.
- It doesn't account for any empty slots which can be leveraged for more efficient storage and lookups.
- Ideally we require a good hash function to guarantee even distribution of the values.
- Say, for a **load factor**  $\lambda = \text{number of objects stored in table} / \text{size of the table}$  (can be  $>1$ ) a good hash function would guarantee that the maximum length of list associated with each key is close to the load factor.

*Note that the order in which the data is stored in the lists (or any other data structures) is based upon the implementation*

requirements. Some general ways include insertion order, frequency of access etc.

## 2. Closed Hashing (Open Addressing)

This collision resolution technique requires a hash table with fixed and known size. During insertion, if a collision is encountered, alternative cells are tried until an empty bucket is found. These techniques require the size of the hash table to be supposedly larger than the number of objects to be stored (something with a load factor  $< 1$  is ideal). There are various methods to find these empty buckets:

- a. Linear Probing
- b. Quadratic probing
- c. Double hashing

### a. Linear Probing

The idea of linear probing is simple, we take a fixed sized hash table and every time we face a hash collision we linearly traverse the table in a cyclic manner to find the next empty slot.

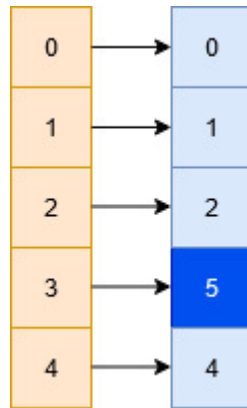
- Assume a scenario where we intend to store the following set of numbers =  $\{0,1,2,4,5,7\}$  into a hash table of size 5 with the help of the following hash function  $H$ , such that  $H(x) = x \% 5$ .
- So, if we were to map the given data with the given hash function we'll get the corresponding values

- $H(0) \rightarrow 0 \% 5 = 0$
- $H(1) \rightarrow 1 \% 5 = 1$
- $H(2) \rightarrow 2 \% 5 = 2$
- $H(4) \rightarrow 4 \% 5 = 4$
- $H(5) \rightarrow 5 \% 5 = 0$

- in this case we see a collision of two terms (0 & 5). In this situation we move linearly down the table to find the first empty slot. Note that this linear traversal is cyclic in nature, i.e. in the event we exhaust the last element during the search



we start again from the beginning until the initial key is reached.



- In this case our hash function can be considered as this:  

$$H(x, i) = (H(x) + i) \% N$$
 where  $N$  is the size of the table and  $i$  represents the linearly increasing variable which starts from 1 (until empty bucket is found).

Despite being easy to compute, implement and deliver best cache performance, this suffers from the problem of clustering (many consecutive elements get grouped together, which eventually reduces the efficiency of finding elements or empty buckets).

## b. Quadratic Probing

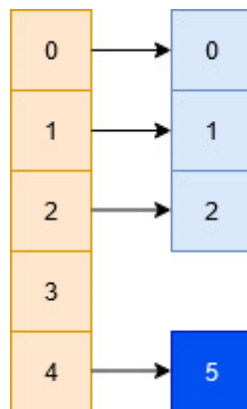
This method lies in the middle of great cache performance and the problem of clustering. The general idea remains the same, the only difference is that we look at the  $Q(i)$  increment at each iteration when looking for an empty bucket, where  $Q(i)$  is some quadratic expression of  $i$ . A simple expression of  $Q$  would be  $Q(i) = i^2$ , in which case the hash function looks something like this:  

$$H(x, i) = (H(x) + i^2) \% N$$

- In general,  $H(x, i) = (H(x) + ((c1 \cdot i^2 + c2 \cdot i + c3))) \% N$ , for some choice of constants  $c1$ ,  $c2$ , and  $c3$
- Despite resolving the problem of clustering significantly it may be the case that in some situations this technique does not find any available bucket, unlike linear probing which always finds an empty bucket.
- Luckily, we can get good results from quadratic probing with the right combination of probing function and hash table size which will guarantee that we will visit as many slots in the

table as possible. In particular, if the hash table's size is a prime number and the probing function is  $H(x, i) = i^2$ , then at least 50% of the slots in the table will be visited. Thus, if the table is less than half full, we can be certain that a free slot will eventually be found.

- Alternatively, if the hash table size is a power of two and the probing function is  $H(x, i) = (i^2 + i)/2$ , then every slot in the table will be visited by the probing function.
- Assume a scenario where we intend to store the following set of numbers = {0,1,2,5} into a hash table of size 5 with the help of the following hash function H, such that  $H(x, i) = (x \% 5 + i^2) \% 5$ .



Clearly 5 and 0 will face a collision, in which case we'll do the following:

- we look at  $5 \% 5 = 0$  (collision)
- we look at  $(5 \% 5 + 1^2) \% 5 = 1$  (collision)
- we look at  $(5 \% 5 + 2^2) \% 5 = 4$  (empty -> place element here)

### c. Double Hashing

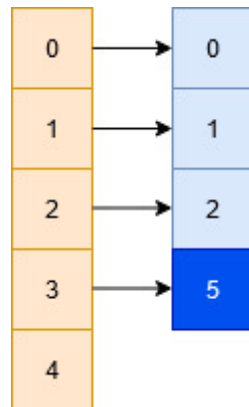
This method is based upon the idea that in the event of a collision we use an another hashing function with the key value as an input to find where in the open addressing scheme the data should actually be placed at.

- In this case we use two hashing functions, such that the final hashing function looks like:  

$$H(x, i) = (H1(x) + i * H2(x)) \% N$$
- Typically for  $H1(x) = x \% N$  a good  $H2$  is  $H2(x) = P - (x \% P)$ , where  $P$  is a prime number smaller than  $N$ .
- A good  $H2$  is a function which never evaluates to zero and ensures that all the cells of a table are effectively traversed.
- Assume a scenario where we intend to store the following set of numbers = {0,1,2,5} into a hash table of size 5 with the help of the following hash function  $H$ , such that  

$$H(x, i) = (H1(x) + i * H2(x)) \% 5$$

$$H1(x) = x \% 5 \text{ and } H2(x) = P - (x \% P), \text{ where } P = 3$$
(3 is a prime smaller than 5)



Clearly 5 and 0 will face a collision, in which case we'll do the following:

- we look at  $5 \% 5 = 0$  (collision)
- we look at  $(5 \% 5 + 1 * (3 - (5 \% 3))) \% 5 = 1$  (collision)
- we look at  $(5 \% 5 + 2 * (3 - (5 \% 3))) \% 5 = 2$  (collision)
- we look at  $(5 \% 5 + 3 * (3 - (5 \% 3))) \% 5 = 3$  (empty -> place element here)

LinearProbing:

KeyValue.java

```
public class KeyValue<T extends Comparable<T>>{
    T key;
    String name;
    int age;
    public KeyValue(){
    public KeyValue( T key,  String name,  int age) {
        this.key = key;
        this.name = name;
        this.age = age;
    }
}
```

LinerProbing.java

```
import java.util.Scanner;
public class LinearProbing<T extends Comparable<T>> {
    private int currentSize;
    int maxSize;
    KeyValue<T>[] keyvalue;

    public LinearProbing( int capacity) {
        currentSize = 0;
        maxSize = capacity;
        keyvalue = new KeyValue[maxSize];
        for(int i=0;i<capacity;i++) {
            keyvalue[i]=null;
        }
    }

    private int hash( T key) {
        return key.hashCode() % maxSize;
    }

    int probe( int index, T key)
    {
        //Fill your code here
        return 0;
    }
}
```

```

public boolean insert( T key,  String name,  int age) {
    //Fill your code here
    int i=hash(key);
    KeyValue<T> k=new KeyValue<T>(key,name,age);
    if(currentSize==maxSize)
        return false;
    while(keyvalue[i]!=null)
    {
        i=(i+1)%maxSize;
    }
    keyvalue[i]=k;
    currentSize++;
    return true;
}

public KeyValue get( T key) {
    int i=hash(key);
    int j=i;
    while(keyvalue[i]!=null) {
        if(keyvalue[i].key.compareTo(key)==0) {
            return keyvalue[i];
        }
        i=(i+1)%maxSize;
        if(j==i)
            break;
    }
    return null;
}

public boolean delete( T key) {
    //Fill your code here
    if(get(key)==null)
        return false;
    else {
        int i,j;
        i=hash(key);
        j=i;
        while(keyvalue[j]!=null) {
            if(keyvalue[j].key.compareTo(key)==0) {
                keyvalue[j]=null;
                currentSize--;
                break;
            }
            j=(j+1)%maxSize;
        }
    }
}

```

```

    }
    return true;
}
public KeyValueType[] getList()
{
    return keyvalue;
}
}

```

## Main.java

```

import java.util.*;
public class Main {
    public static void main( String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the Table Size:");
        /** maxSizeake object of LinearProbingHashTable */
        int n1=scan.nextInt();
        System.out.println("Enter the data type of the key to be added in Hash Table");
        System.out.println("1.Integer");
        System.out.println("2.String");
        int nch=scan.nextInt();
        if(nch==1) {
            LinearProbing lpht = new LinearProbing<Integer>(n1);
            while(true)
            {
                System.out.println("1. Insert");
                System.out.println("2. Delete");
                System.out.println("3. Search");
                System.out.println("4. Display");
                System.out.println("5. Exit\nEnter your choice:");
                int choice = scan.nextInt();
                switch (choice)
                {
                    case 1 :
                        scan.nextLine();
                        System.out.println("Enter the key value:");
                        int key = Integer.parseInt(scan.nextLine());
                        System.out.println("Enter the Name:");
                        String name = scan.nextLine();
                        System.out.println("Enter the Age:");
                        int age = Integer.parseInt(scan.nextLine());
                        //Fill your code here
                        boolean j =lpht.insert(key, name, age);

```

```

        if(j==false)
            System.out.println("Can't Insert. Hash Table is full!");
        break;
    case 2 :
        scan.nextLine();
        System.out.println("Enter the key value:");
        int k1 = scan.nextInt();
        //Fill your code here
        boolean l=lpht.delete(k1);
        if(l==false)
            System.out.println(k1+" is not available in Hash Table");
        else
            System.out.println(k1+" deleted from Hash Table");
        break;
    case 3 :
        scan.nextLine();
        System.out.println("Enter the key value:");
        int k = scan.nextInt();
        KeyValue n11 = lpht.get(k);
        if(n11!=null) {
            System.out.println("Voter ID : "+n11.key);
            System.out.println("Name      : "+n11.name);
            System.out.println("Age       : "+n11.age);
        }
        else {
            System.out.println(k+" is not present in hash table");
        }
        //Fill your code here
        break;
    case 4 :
        System.out.printf("%-11s%-14s%-11s%-14s\n", "ID", "Name", "Age", "Index");
        System.out.println("-----");
        KeyValue<Integer>[] x=lpht.getList();
        for(int i=0;i<x.length;i++) {
            if(x[i]!=null)
                System.out.printf("%-11d%-15s%-14d\n", x[i].key, x[i].name, x[i].age, i);
        }
        System.out.println();
        break;
    case 5:
        System.exit(0);
        break;

```

```

        default :
            System.out.println("Wrong Entry \n ");
            break;
        }
    }
}
else if(nch==2) {
    LinearProbing lpht = new LinearProbing<String>(n1);
    while(true)
    {
        System.out.println("1. Insert");
        System.out.println("2. Delete");
        System.out.println("3. Search");
        System.out.println("4. Display");
        System.out.println("5. exit\nEnter your choice:");

        int choice = scan.nextInt();
        switch (choice)
        {
            case 1 :
                scan.nextLine();
                System.out.println("Enter the key value:");
                String key = scan.nextLine();
                System.out.println("Enter the Name:");
                String name = scan.nextLine();
                System.out.println("Enter the Age:");
                int age = Integer.parseInt(scan.nextLine());
                //Fill your code here

                boolean j =lpht.insert(key, name, age);
                if(j==false)
                    System.out.println("Can't Insert. Hash Table is full!");
                break;
            case 2 :
                scan.nextLine();
                System.out.println("Enter the key value:");
                String k1 = scan.nextLine();
                //Fill your code here
                boolean l=lpht.delete(k1);
                if(l==false)
                    System.out.println(k1+" is not available in Hash Table");
                else
                    System.out.println(k1+" deleted from Hash Table");
                break;
            case 3 :

```



```

        scan.nextLine();
        System.out.println("Enter the key value:");
        String k3 = scan.nextLine();
        KeyValue n11 = lpht.get(k3);
        if(n11!=null) {
            System.out.println("Voter ID : "+n11.key);
            System.out.println("Name      : "+n11.name);
            System.out.println("Age       : "+n11.age);
        }
        else {
            System.out.println(k3+" is not present in hash table");
        }
        //Fill your code here
        break;
    case 4 :

        System.out.printf("%-11s%-14s%-11s%\n", "ID", "Name", "Age", "Index");
        System.out.println("-----");

        KeyValue<Integer>[] x=lpht.getList();
        for(int i=0;i<x.length;i++) {
            if(x[i]!=null)
                System.out.printf("%-11s%-15s%-14d%\n",x[i].key,x[i].name,x[i].age,i);
        }
        System.out.println();
        //Fill your code here

        break;
    case 5:
        System.exit(0);
    default :
        System.out.println("Wrong Entry \n ");
        break;
    }

}

}

}

}

```

Sample	Input	and	Output	1:
Enter the	the	Table		Size:
7				
Enter the data type of the key to be added in Hash Table				

1.Integer

2.String

1

1.

2.

3.

4.

5.

Enter

your

Insert

Delete

Search

Display

Exit

choice:

1

Enter

the

key

value:

10

Enter

the

Name:

Arun

Enter

the

Age:

21

1.

2.

3.

4.

5.

Enter

your

Insert

Delete

Search

Display

Exit

choice:

1

Enter

the

key

value:

14

Enter

the

Name:

Sam

Enter

the

Age:

25

1.

2.

3.

4.

5.

Enter

your

Insert

Delete

Search

Display

Exit

choice:

1

Enter

the

key

value:

21

Enter

the

Name:

Dane

Enter

the

Age:

50

1.

2.

3.

4.

5.

Enter

your

Insert

Delete

Search

Display

Exit

choice:

1

Enter

the

key

value:

36

Enter

the

Name:

Mike

Enter

the

Age:

12

1.

Insert

2.

Delete

3.

Search

4.

Display

5.

Exit

Enter

your

choice:

4

ID

Name

Age

Index

14

Sam

25

0

21

Dane

50

1

36

Mike

12

2

10

Arun

21

3

1.

Insert

2.

Delete

3.

Search

4.

Display

5.

Exit

Enter

your

choice:

1

Enter

the

key

value:

13

Enter

the

Name:

Don

Enter

the

Age:

34

1.

Insert

2.

Delete

3.

Search

4.

Display

5.

Exit

Enter

your

choice:

1

Enter

the

key

value:

19

Enter

the

Name:

Krish

Enter

the

Age:

13

1.

Insert

2.

Delete

3.

Search

4.

Display

5.

Exit

Enter

your

choice:

4

ID

Name

Age

Index

14			Sam	25	0
21			Dane	50	1
36			Mike	12	2
10			Arun	21	3
19			Krish	13	5
13	Don	34	6		

## SeparateChaining

### SeparateChaining.java

```
import java.util.*;
public class SeparateChaining<K extends Comparable<K>, V> {

    private int n;
    private int m;
    private Node<K, V>[] table;

    private class Node<K extends Comparable<K>, V> {
        private K key;
        private V value;
        private Node<K,V> next;

        public Node() {

        }

        public Node(K key, V value, Node next) {
            this.key = key;
            this.value = value;
            this.next = next;
        }
    }

    public SeparateChaining(int capacity) {
        //Fill your code here
        table=new Node[capacity];
        m=capacity;
        for(int i=0;i<m;i++) {
            table[i]=null;
        }
    }

    public int size() {
```

```

        return n;
    }

    public boolean isEmpty() {
        //Fill your code here
        return (n==0);
    }

    public V get(K key) {
        //Fill your code here
        return null;
    }

    public boolean contains(K key) {
        //Fill your code here
        return true;
    }

    public void insert(K key, V value) {
        //Fill your code here
        int k=hash(key);
        Node<K,V> temp=new Node<K,V>(key,value,null);
        Node<K,V> node=table[k];
        if(table[k]==null)
            table[k]=temp;
        else{
            node.next=temp;
            temp=node;
        }
        n++;
    }

    public V delete(K key) {
        //Fill your code here
        int k=hash(key);
        Node<K,V> temp=table[k];
        Node<K,V> prev=null;
        while(temp!=null && key.compareTo(temp.key)!=0){
            prev=temp;
            temp=temp.next;
        }
        if(temp!=null && key.compareTo(temp.key)==0){
            if(prev==null)
                table[k]=null;

            else
                prev.next=temp.next;
        }
    }

```

```

        System.out.println("Data deleted successfully from Hash Table"
);
        n--;
        return temp.value;

    }
    else
        System.out.println("Given data is not present in Hash Table");
        return null;

}

public void search(K key) {
    //Fill your code here
    int k=hash(key);
    Node<K,V> temp=table[k];
    while(temp!=null && key.compareTo(temp.key)!=0){
        temp=temp.next;
    }
    if(temp!=null && key.compareTo(temp.key)==0){
        System.out.println("ID : "+temp.key);
        System.out.println("Value : "+temp.value);
        return;
    }
    else{
        System.out.println("Search element unavailable in hash table");
    }

}

public void display() {
    //Fill your code here
    if(n==0)
        System.out.println("Hash Table is empty");
    else{
        for(int i=0;i<m;i++){
            System.out.println();
            System.out.println("Data at index "+i+" in Hash Table:");
            System.out.println("ID      Value");
            System.out.println("-----");
            Node<K,V> t=table[i];

            while(t!=null){
                System.out.println(t.key+"      "+t.value);
                t=t.next;
            }
        }
    }
}

```

```

    }

    private int hash(K key) {
        //Fill your code here
        if(key.getClass().getSimpleName().equals("Integer"))
            return (Integer)key%m;
        return ((String)key).codePointAt(0)%m;
    }
}

```

## Main.java

```

import java.util.*;

public class Main{

    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the size of the Hash Table:");
        int n=scan.nextInt();
        int choice=0;
        System.out.println("Enter the combination of data type you want to be
added in hash table");
        System.out.println("1. Integer and String\n2. String and Double");
        int n1 = scan.nextInt();
        if(n1==1)
        {
            SeparateChaining<Integer,String> sc=new SeparateChaining<Integer,S
tring>(n);
            int val;
            String name;
            //Fill your code here
        }
        do
        {
            System.out.println("1. Insertion");
            System.out.println("2. Deletion");
            System.out.println("3. Searching");
            System.out.println("4. Display");
            System.out.println("5. Exit");
            System.out.println("Enter your choice:");
            choice = scan.nextInt();
            switch (choice)
            {
                case 1 :

```

```

        System.out.println("Enter the key:");
        val=scan.nextInt();
        scan.nextLine();
        System.out.println("Enter the value:");
        name=scan.nextLine();
        //Fill your code here
        sc.insert(val,name);
        break;
    case 2 :
        System.out.println("Enter the key to perform deletion:");
        val=scan.nextInt();
        //Fill your code here
        String res=sc.delete(val);
        break;
    case 3 :
        System.out.println("Enter the key to search:");
        val=scan.nextInt();
        //Fill your code here
        sc.search(val);
        break;
    case 4 :
        //Fill your code here
        sc.display();
        break;
    case 5 : System.exit(0);
            break;
    default:
        System.out.println("Invalid option");
        break;
    }

    } while (choice<5);
}
if(n1==2)
{
    SeparateChaining<String,Double> sc=new SeparateChaining<String,Double>
(n);
    String val;
    double d;
    //Fill your code here
    do
    {
        System.out.println("1. Insertion");
        System.out.println("2. Deletion");
        System.out.println("3. Searching");
        System.out.println("4. Display");
        System.out.println("5. Exit");
        System.out.println("Enter your choice:");

```



```

        choice = scan.nextInt();
        switch (choice)
        {
        case 1 :
            System.out.println("Enter the key:");
            scan.nextLine();
            val=scan.nextLine();
            System.out.println("Enter the value");
            d=scan.nextDouble();
            //Fill your code here
            sc.insert(val,d);
            break;
        case 2 :
            System.out.println("Enter the key to perform deletion:");
            scan.nextLine();
            val=scan.nextLine();
            //Fill your code here
            Double rd=sc.delete(val);
            break;
        case 3 :
            System.out.println("Enter the key to search:");
            scan.nextLine();
            val=scan.nextLine();
            //Fill your code here
            sc.search(val);
            break;
        case 4 :
            //Fill your code
            sc.display();
            break;
        case 5 : System.exit(0);
                break;
        default:
            System.out.println("Invalid option");
            break;
        }

    } while (choice<5);
}
}
}

```

## Dictionary ADT

The most common objective of computer programs is to store and retrieve data.

The dictionary ADT provides operations for storing records, finding records, and removing records from the collection.

This ADT gives us a standard basis for comparing various data structures. Loosly speaking, we can say that any data structure that supports insert, search, and deletion is a “dictionary”.

Dictionaries depend on the concepts of a **search key** and **comparable** objects. To implement the dictionary’s search function, we will require that keys be **totally ordered**. Ordering fields that are naturally multi-dimensional, such as a point in two or three dimensions, present special opportunities if we wish to take advantage of their multidimensional nature. This problem is addressed by **spatial data structures**.

Here is code to define a simple abstract dictionary class.

```
public interface Dictionary<K, E> {  
  
    /** Reinitialize dictionary */  
    public void clear();  
  
    /** Insert a record  
        @param k The key for the record being inserted.  
        @param e The record being inserted. */  
    public void insert(K key, E elem);  
  
    /** Remove and return a record.  
        @param k The key of the record to be removed.  
        @return A matching record. If multiple records match  
        "k", remove an arbitrary one. Return null if no record  
        with key "k" exists. */
```

```

public E remove(K key);

    /** Remove and return an arbitrary record from dictionary.
        @return the record removed, or null if none exists. */

public E removeAny();

    /** @return A record matching "k" (null if none exists).
        If multiple records match, return an arbitrary one.
        @param k The key of the record to find */

public E find(K key);

    /** @return The number of records in the dictionary. */

public int size();
}

```

The methods insert and find are the heart of the class. Method insert takes a record and inserts it into the dictionary. Method find takes a key value and returns some record from the dictionary whose key matches the one provided. If there are multiple records in the dictionary with that key value, there is no requirement as to which one is returned.

Method clear simply re-initializes the dictionary. The remove method is similar to find, except that it also deletes the record returned from the dictionary. Once again, if there are multiple records in the dictionary that match the desired key, there is no requirement as to which one actually is removed and returned. Method size returns the number of elements in the dictionary.

The remaining Method is removeAny. This is similar to remove, except that it does not take a key value. Instead, it removes an arbitrary record from the dictionary, if one exists. The purpose of this method is to allow a user the ability to iterate over all elements in the dictionary (of course, the dictionary will become empty in

the process). Without the `removeAny` method, dictionary users could not get at a record of the dictionary that they didn't already know the key value for. With the `removeAny` method, the user can process all records in the dictionary as shown in the following code fragment.

```
while (dict.size() > 0) {  
    Object it = dict.removeAny();  
    doSomething(it);  
}
```

There are other approaches that might seem more natural for iterating through a dictionary, such as using a “first” and a “next” function. But not all data structures that we want to use to implement a dictionary are able to do “first” efficiently. For example, a hash table implementation cannot efficiently locate the record in the table with the smallest key value. By using `RemoveAny`, we have a mechanism that provides generic access.

Given a database storing records of a particular type, we might want to search for records in multiple ways. For example, we might want to store payroll records in one dictionary that allows us to search by ID, and also store those same records in a second dictionary that allows us to search by name.

Here is an implementation for a payroll record.

```
/** A simple payroll entry with ID, name, address fields */  
public class Payroll {  
  
    private Integer ID;  
    private String name;  
    private String address;
```

```

/** Constructor */
Payroll(int inID, String inname, String inaddr) {
    ID = inID;
    name = inname;
    address = inaddr;
}

/** Data member access functions */
public Integer getID() { return ID; }
public String getname() { return name; }
public String getaddr() { return address; }
}

```

Class Payroll has multiple fields, each of which might be used as a search key. Simply by varying the type for the key, and using the appropriate field in each record as the key value, we can define a dictionary whose search key is the ID field, another whose search key is the name field, and a third whose search key is the address field. Here is an example where Payroll objects are stored in two separate dictionaries, one using the ID field as the key and the other using the name field as the key.

```

// IDdict organizes Payroll records by ID
Dictionary IDdict = new UALDictionary();

// namedict organizes Payroll records by name
Dictionary namedict = new UALDictionary();

Payroll foo1 = new Payroll(5, "Joe", "Anytown");
Payroll foo2 = new Payroll(10, "John", "Mytown");

```

```
IDdict.insert(foo1.getID(), foo1);
IDdict.insert(foo2.getID(), foo2);
namedict.insert(foo1.getname(), foo1);
namedict.insert(foo2.getname(), foo2);

Payroll findfoo1 = (Payroll)IDdict.find(5);
Payroll findfoo2 = (Payroll)namedict.find("John");
```

The fundamental operation for a dictionary is finding a record that matches a given key. This raises the issue of how to **extract the key** from a record. We will usually assume that dictionary implementations store a **key-value pair** so as to be able to extract the key associated with a record for this particular dictionary.

The insert method of the dictionary class supports the key-value pair implementation because it takes two parameters, a record and its associated key for that dictionary.

Now that we have defined the dictionary ADT and settled on the design approach of storing key-value pairs for our dictionary entries, we are ready to consider ways to implement it. Two possibilities would be to use an array-based or linked list. Here is an implementation for the dictionary using an (unsorted) array-based list.

```
// Dictionary implemented by unsorted array-based list.
public class UALDictionary implements Dictionary {
    private static final int defaultSize = 10; // Default size
    private AList list; // To store dictionary

    // Constructors
    UALDictionary() { this(defaultSize); }
    UALDictionary(int sz) { list = new AList(sz); }
```

*// Reinitialize*

```
public void clear() { list.clear(); }
```

*// Insert an element: append to list*

```
public void insert(Comparable k, Object e) {  
    KVPair temp = new KVPair(k, e);  
    list.append(temp);  
}
```

*// Use sequential search to find the element to remove*

```
public Object remove(Comparable k) {  
    Object temp = find(k);  
    if (temp != null) { list.remove(); }  
    return temp;  
}
```

*// Remove the last element*

```
public Object removeAny() {  
    if (size() != 0) {  
        list.moveToEnd();  
        list.prev();  
        KVPair e = (KVPair)list.remove();  
        return e.value();  
    }  
    else { return null; }  
}
```

```

// Find k using sequential search
// Return the record with key value k
public Object find(Comparable k) {
    for(list.moveToStart(); list.currPos() < list.length();
        list.next()) {
        KeyValuePair temp = (KeyValuePair)list.getValue();
        if (k.compareTo(temp.key()) == 0) {
            return temp.value();
        }
    }
    return null; // "k" does not appear in dictionary
}

// Return list size
public int size() { return list.length(); }
}

```

## Linear List Representation

A dictionary can be maintained as order of Key-Value Pairs( where Key-Value Pairs can be in ascending or descending order based on Keys.

To facilitate this representation, we may implement dictionary as

→ Sorted Array

→ Sorted Chain



## **Basics of linear list**

A linear list, also known as an array or a one-dimensional array, is a fundamental data structure in computer science. It is a collection of elements of the same data type arranged in a sequential manner. Each element in the list is assigned a unique index, starting from 0 for the first element and incrementing by 1 for each subsequent element.

Here are some basic concepts related to linear lists:

**Elements:** A linear list consists of a fixed number of elements, each of which can be of any data type, such as integers, floating-point numbers, characters, or even objects. The elements are stored in contiguous memory locations.

**Indexing:** Each element in the linear list is uniquely identified by its index. The index represents the position of an element within the list. In most programming languages, the indices start from 0 and increment by 1 for each subsequent element.

**Length:** The length of a linear list refers to the total number of elements it contains. It indicates the size or capacity of the list. The length is often fixed when the list is created, but some programming languages allow dynamic resizing.

**Accessing Elements:** Elements in a linear list can be accessed directly by their index. For example, to access the element at index 2, you would use the syntax `list[2]`. This allows for efficient retrieval of individual elements.

**Insertion and Deletion:** Elements can be inserted or deleted from a linear list. When an element is inserted, the other elements may need to be shifted to accommodate the new element. Similarly, when an element is deleted, the subsequent elements may need to be shifted to fill the gap.

Searching: Linear lists can be searched for a specific element. This can be done by iterating over the elements and comparing them with the target value. The search can be performed sequentially from the beginning of the list or using more efficient algorithms like binary search on a sorted list.

Operations: Apart from basic access, insertion, deletion, and searching, linear lists support various other operations, such as sorting the elements, merging two lists, splitting a list, reversing the order of elements, and more.

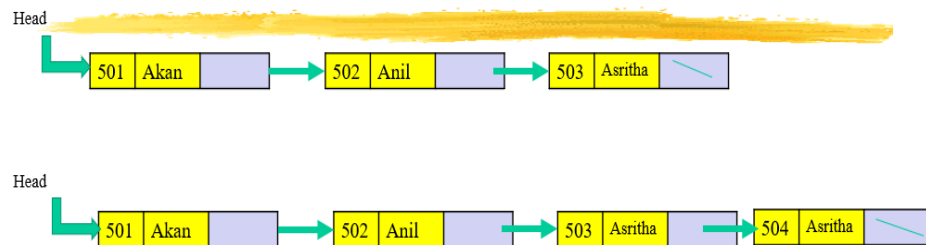
It's worth noting that in some programming languages, the term "list" may refer to a different data structure, such as a linked list, which has different characteristics and behaviors compared to a linear list (array).

## **Sorted Chain:**

- It is a sequence of Nodes that are arranged w.r.t Key( either in ascending or descending)
- Each Node holds on Key-Value Pair.

**Example:**

### Student Dictionary with Sorted Chain



### Code:

```
Interface DDictionary<K extends Comparable<K>, V >{  
    void insert( Pair<K,V> P);  
    void delete(K Key);  
    Pair<K,V> find(K Key);  
    int size();  
    boolean isEmpty();  
    void display();  
}
```

### **Pair.java**

```
class Pair<K extends Comparable<K>,V>{  
  
    K Key;  
  
    V Value;  
  
    Pair(){}  
  
    Pair(K Key, V value){  
  
        this.Key=Key;  
  
        this.Value=Value;  
  
    }  
  
    @Override  
  
    public String toString(){  
  
        return “ Key :” + Key+ “Valeue “+Value  ;  
  
    }  
  
}
```

### **PairNode.java**

```
class PairNode<K extends Comparable<K>, V> {  
  
    Pair<K, V> data;  
  
    PairNode<K, V> next;  
  
    PairNode( Pair<K,V> P, PairNode<K,V> next){  
  
        data =P;  
  
        this.next=next;  
  
}
```

```
}  
}
```

### Sorted\_Chain.java

```
class Sorted_Chain<K extends Comparable<K> ,V> implements  
DDictionary<K , V >{
```

```
    PairNode<K,V> Head;
```

```
    int dsize;
```

```
    public boolean isEmpty() {
```

```
        return Head==null;
```

```
    }
```

```
    public int size(){
```

```
        return dsize;
```

```
    }
```

```
    public void insert(Pair<K,V> P){
```

```
        PairNode<K,V> npNode =new PairNode<K,V>(P, null);
```

```
        PairNode<K,V> temp;
```

```
        PairNode<K,V> prev;
```

```
        //insertion
```

```
        if(isEmpty())// check for emptyr
```

```
        {Head=npNode;}
```

```
        else
```

```
        if( P.Key.compareTo(Head.data.Key)<0) //Insert at the  
beginning
```

```
        { npNode.next=Head;
```

```

        Head=npNode;
    }

    else{ temp=Head;

        prev=null;

        while(temp !=null &&
P.Key.compareTo(temp.data.Key)>0){

            prev=temp;

            temp=temp.next;

        }

        npNode.next=prev.next;

        prev.next=npNode;

    }

    dsize++;

}

public void display(){

    PairNode<K,V> temp;

    if(isEmpty()){

        System.out.println(" DICATIONARY IS EMPTY");

        return;

    }

```

```

        else{
            temp=Head;
            while(temp!=null){
                System.out.println(temp.data);
                temp=temp.next;
            }
        }
    }

    public Pair<K,V> find(K Key){
        Pair<K,V> T=null;
        PairNode<K,V> temp=Head;
        while(temp !=null && Key.compareTo(temp.data.Key)>0){
            temp=temp.next;
        }
        if(temp!=null)
            if(temp.data.Key.equals(Key))
                T=temp.data;
        return T;
    }

    public void delete(K Key){
        PairNode<K,V> temp=Head;
        PairNode<K,V> prev=null;
    }

```

```

        while(temp!=null
Key.compareTo(temp.data.Key)>0){
            prev=temp;
            temp=temp.next;
        }
        if(prev==null && temp==null){
            System.out.println("DICTIONARY    MIGHT    is
EMPTY");
            return;
        }

        if(temp.data.Key.equals(Key)){
            System.out.println(" Deleted Entry is"+ temp.data);
            if(temp==Head){
                Head=Head.next;
                dsize--;
            }
            else{
                prev.next=temp.next;
                dsize--;
            }
        }
        else
            System.out.println("Element not found");
    }

```



```

temp=null;

prev=null;

}

```

### **SortedChainDriver.java**

```

class SortedChainDriver{

    public static void main(String[] args) throws exception{

        int option;

        Integer rno;

        String name;

        BufferedReader br=new        BufferedRead(new
InputStreamReader(System.in))

        SortedChain<Integer, String> Sc=new
SortedChain<Integer, String>();

        Pair<Integer, String> P1;

        do{

            System.out.println("    SORTED    CHAIN    DICTIONARY
OPERATIONS");

            System.out.println("1. INSERT");

            System.out.println("2.DELETE")

            System.out.println("3.FIND");

            System.out.println("4.SIZE");

            System.out.println("5.DISPLAY");

```

```

System.out.println("ENTER OPERATION NUMBER");

option=Integer.parseInt(br.readLine());

switch(option){

    case 1:  // insert

        System.out.println("Enter an entry details ,Key type is
integer and value type is String");

        rno= Integer.parseInt(br.readLine());

        name=br.readLine();

        P1=new Pair<Integer,String>( rno,name);

        Sc.inser(P1); break;

case 2:

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

    rno=Integer.parseInt(br.readLine());

    Sc.delete(rno); break;

Case 3:

Pair<Integer,String> e;

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

rno=Integer.parseInt(br.readLine());

e=Sc.find(rno);

if(e==null){

    System.out.println("NO ELEMENT");

}

else {

```

```

        System.out.println(e);
    }
    Case 4: sop(Sc.size()); break;
    Case 5:
        Sc.display();break;
    Case 6: System.exit(0);
}}while(true);
}}
```

## **Dictionary using sorted array**

To implement a dictionary using a sorted array in Java, you can use binary search for efficient lookup operations. Here's an example implementation:

```

import java.util.Arrays;

public class SortedArrayDictionary<K extends Comparable<K>, V>
{
    private static final int INITIAL_CAPACITY = 10;
    private Entry<K, V>[] entries;
    private int size;

    public SortedArrayDictionary() {
        this.entries = new Entry[INITIAL_CAPACITY];
    }
}
```

```

        this.size = 0;
    }

    public void put(K key, V value) {
        Entry<K, V> newEntry = new Entry<>(key, value);
        if (size == entries.length) {
            expandCapacity();
        }

        int index = Arrays.binarySearch(entries, 0, size, newEntry);
        if (index >= 0) {
            // Key already exists, update the value
            entries[index].setValue(value);
        } else {
            // Key doesn't exist, insert the new entry at the
            appropriate position

            int insertionPoint = -index - 1;

            System.arraycopy(entries, insertionPoint, entries,
            insertionPoint + 1, size - insertionPoint);

            entries[insertionPoint] = newEntry;

            size++;
        }
    }
}

```

```

public V get(K key) {
    Entry<K, V> searchEntry = new Entry<>(key, null);
    int index = Arrays.binarySearch(entries, 0, size,
searchEntry);
    if (index >= 0) {
        return entries[index].getValue();
    }
    return null;
}

```

```

public void remove(K key) {
    Entry<K, V> searchEntry = new Entry<>(key, null);
    int index = Arrays.binarySearch(entries, 0, size,
searchEntry);
    if (index >= 0) {
        System.arraycopy(entries, index + 1, entries, index, size -
index - 1);
        entries[size - 1] = null;
        size--;
    }
}

```

```

public int size() {
    return size;
}

```

```
}
```

```
public boolean isEmpty() {
```

```
    return size == 0;
```

```
}
```

```
private void expandCapacity() {
```

```
    int newCapacity = entries.length * 2;
```

```
    entries = Arrays.copyOf(entries, newCapacity);
```

```
}
```

```
private static class Entry<K extends Comparable<K>, V>  
implements Comparable<Entry<K, V>> {
```

```
    private K key;
```

```
    private V value;
```

```
public Entry(K key, V value) {
```

```
    this.key = key;
```

```
    this.value = value;
```

```
}
```

```
public K getKey() {
```

```
    return key;
```

```
}
```

```
public V getValue() {  
    return value;  
}
```

```
public void setValue(V value) {  
    this.value = value;  
}
```

```
@Override
```

```
public int compareTo(Entry<K, V> other) {  
    return key.compareTo(other.getKey());  
}
```

```
}
```

```
public static void main(String args[])  
{
```

```
    SortedArrayDictionary<String, Integer> dictionary = new  
SortedArrayDictionary<>();
```

```
    dictionary.put("apple", 10);
```

```
    dictionary.put("banana", 5);
```

```
    dictionary.put("orange", 15);
```

```
    System.out.println(dictionary.get("apple")); // Output: 10
```

```

        System.out.println(dictionary.get("banana")); // Output: 5

        System.out.println(dictionary.get("orange")); // Output: 15

        dictionary.remove("banana");

        System.out.println(dictionary.get("banana")); // Output:
null

        System.out.println(dictionary.size()); // Output: 2

        System.out.println(dictionary.isEmpty()); // Output:
false

    }

}

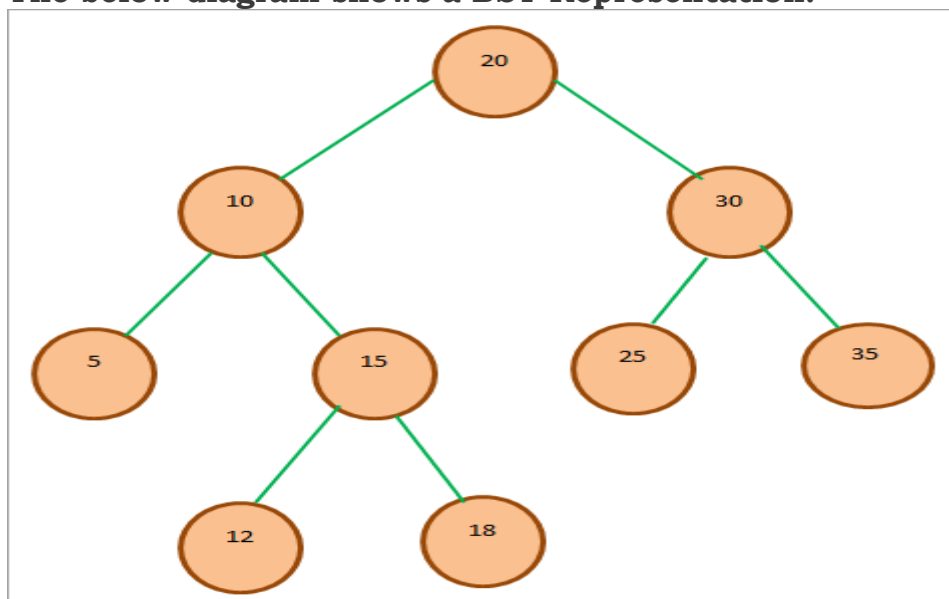
```

**Note :** This implementation assumes that the keys in the dictionary are unique.

## Binary Search Tree In Java

A BST does not allow duplicate nodes.

**The below diagram shows a BST Representation:**





Above shown is a sample BST. We see that 20 is the root node of this tree. The left subtree has all the node values that are less than 20. The right subtree has all the nodes that are greater than 20. We can say that the above tree fulfills the BST properties.

The BST data structure is considered to be very efficient when compared to Arrays and Linked list when it comes to insertion/deletion and searching of items.

BST takes  $O(\log n)$  time to search for an element. As elements are ordered, half the subtree is discarded at every step while searching for an element. This becomes possible because we can easily determine the rough location of the element to be searched.

Similarly, insertion and deletion operations are more efficient in BST. When we want to insert a new element, we roughly know in which subtree (left or right) we will insert the element.

### **Creating A Binary Search Tree (BST)**

Given an array of elements, we need to construct a BST.

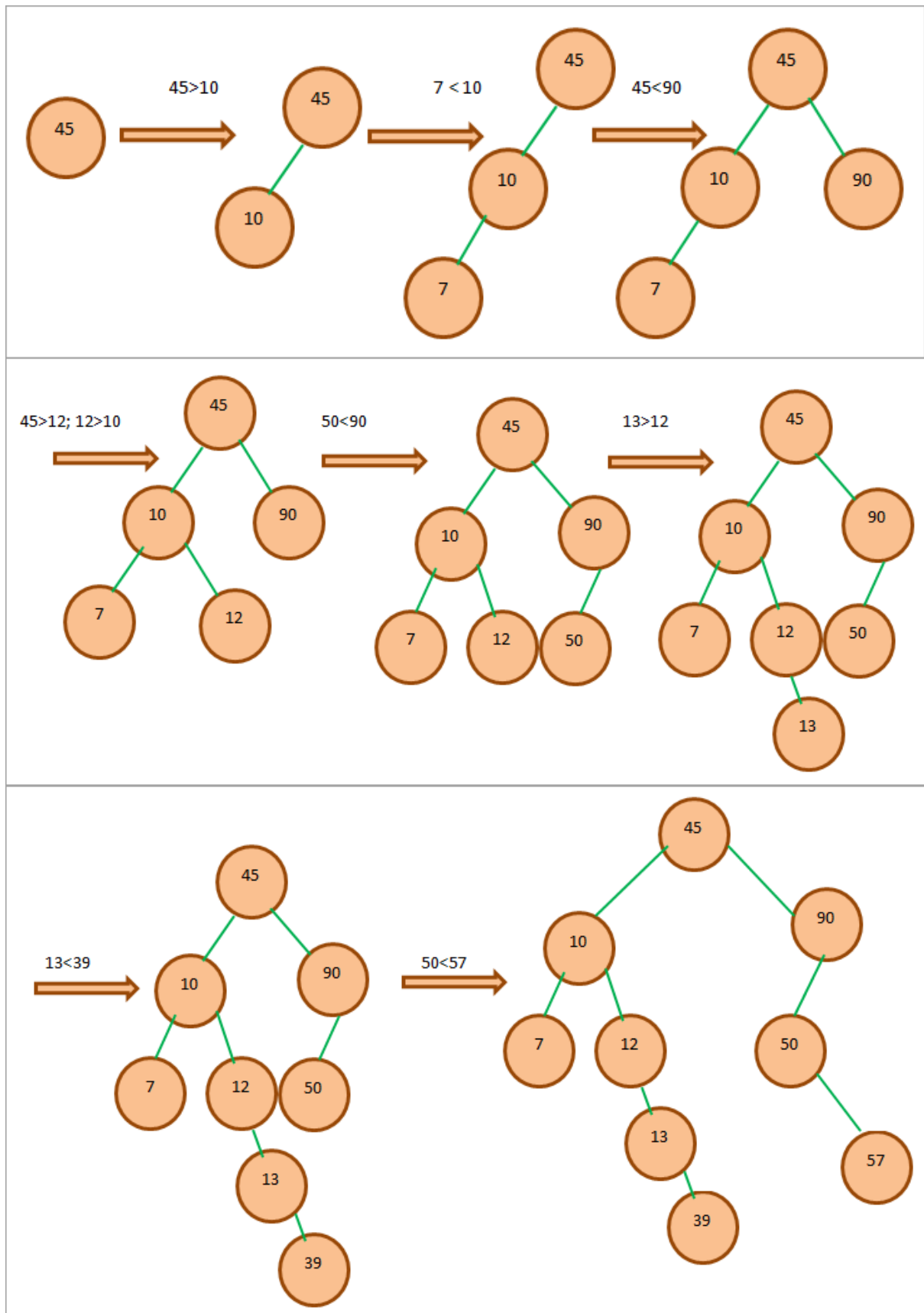
**Let's do this as shown below:**

**Given array:** 45, 10, 7, 90, 12, 50, 13, 39, 57

Let's first consider the top element i.e. 45 as the root node. From here we will go on creating the BST by considering the properties already discussed.

To create a tree, we will compare each element in the array with the root. Then we will place the element at an appropriate position in the tree.

**The entire creation process for BST is shown below.**



Completed BST

## Binary Search Tree Operations

BST supports various operations. The following table shows the methods supported by BST in Java. We will discuss each of these methods separately.

Method/operation	Description
Insert	Add an element to the BST by not violating the BST property.
Delete	Remove a given node from the BST. The node can be the root, internal node, leaf, or leaf node.
Search	Search the location of the given element in the BST. The method checks if the tree contains the specified key.

### ***Insert An Element In BST***

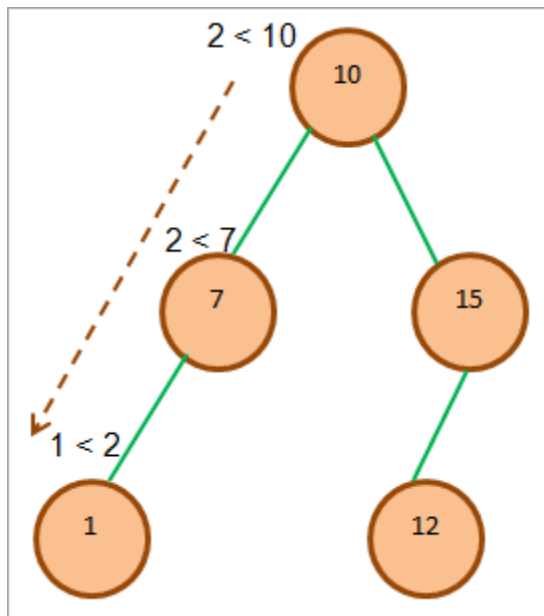
An element is always inserted as a leaf node in BST.

**Given below are the steps for inserting an element.**

1. Start from the root.
2. Compare the element to be inserted with the root node.  
If it is less than root, then traverse the left subtree or traverse the right subtree.
3. Traverse the subtree till the end of the desired subtree.  
Insert the node in the appropriate subtree as a leaf node.

**Let's see an illustration of the insert operation of BST.**

Consider the following BST and let us insert element 2 in the tree.



Figure

1

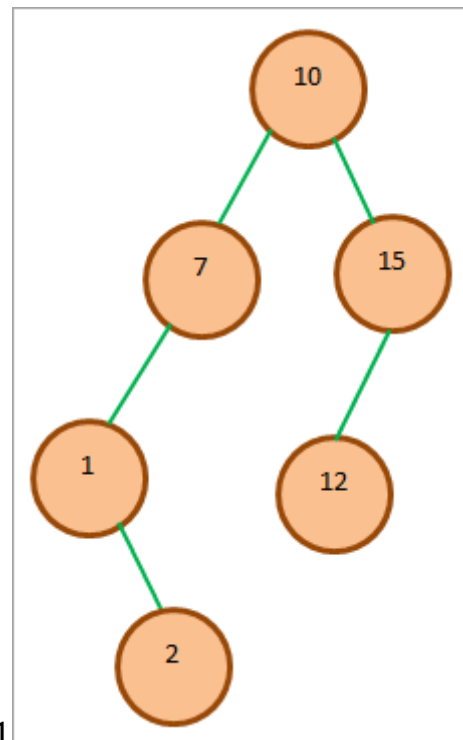


Figure 2

The insert operation for BST is shown above. In fig (1), we show the path that we traverse to insert element 2 in the BST. We have also shown the conditions that are checked at each node. As a result of the recursive comparison, element 2 is inserted as the right child of 1 as shown in fig (2).

### **Search Operation In BST**

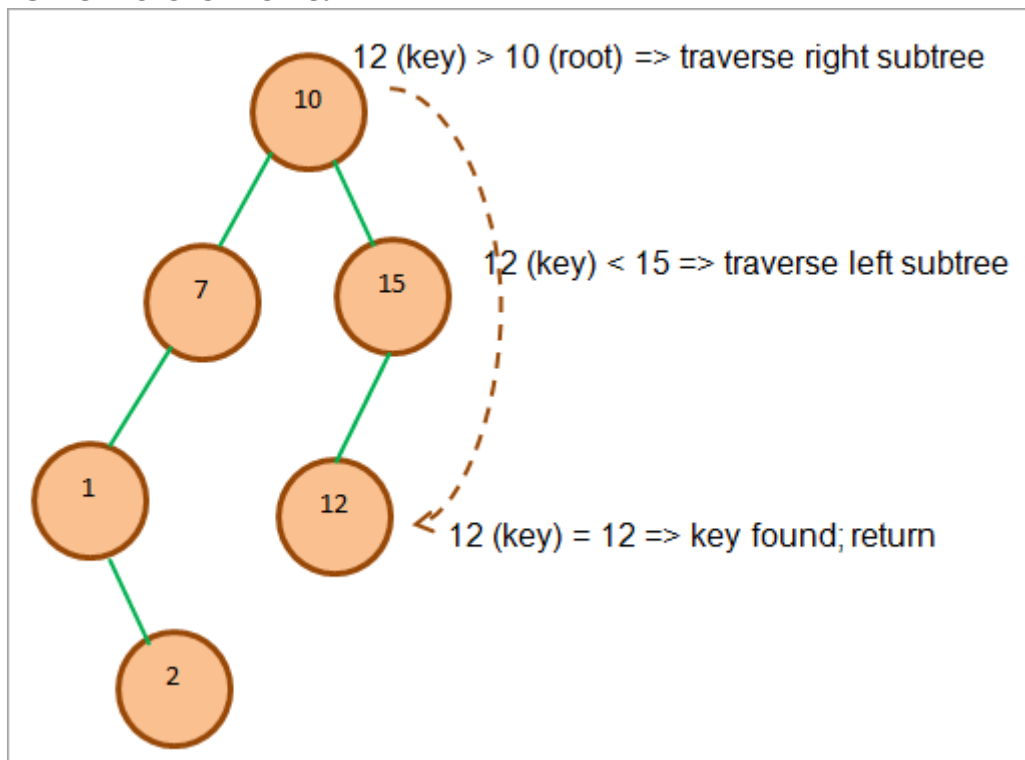
To search if an element is present in the BST, we again start from the root and then traverse the left or right subtree depending on whether the element to be searched is less than or greater than the root.

**Enlisted below are the steps that we have to follow.**

1. Compare the element to be searched with the root node.
2. If the key (element to be searched) = root, return root node.
3. Else if key < root, traverse the left subtree.
4. Else traverse right subtree.
5. Repetitively compare subtree elements until the key is found or the end of the tree is reached.

Let's illustrate the search operation with an example. Consider that we have to search the key = 12.

**In the below figure, we will trace the path we follow to search for this element.**



As shown in the above figure, we first compare the key with root. Since the key is greater, we traverse the right subtree. In the right subtree, we again compare the key with the first node in the right subtree.

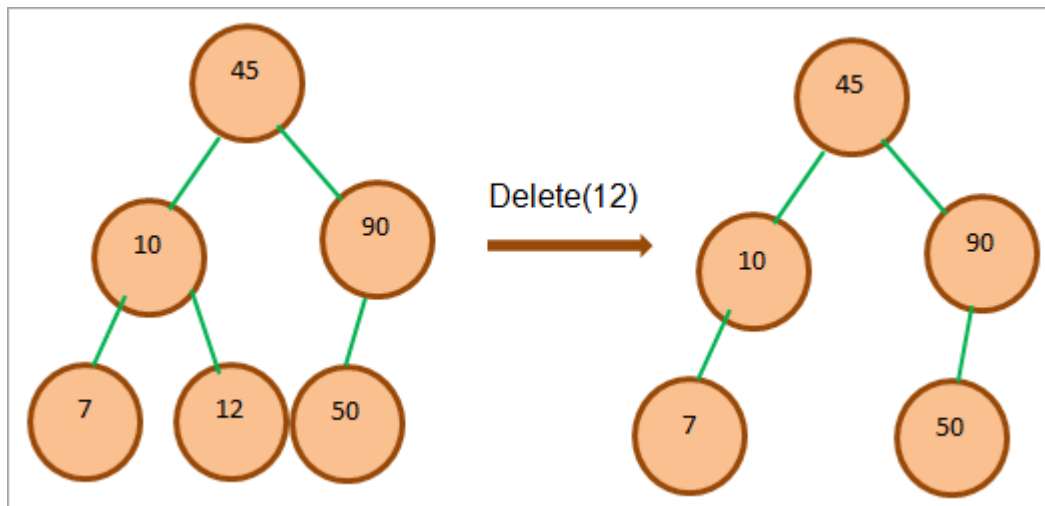
We find that the key is less than 15. So we move to the left subtree of node 15. The immediate left node of 15 is 12 that matches the key. At this point, we stop the search and return the result.

### ***Remove Element From The BST***

When we delete a node from the BST, then there are three possibilities as discussed below:

#### **Case 1: Node Is A Leaf Node**

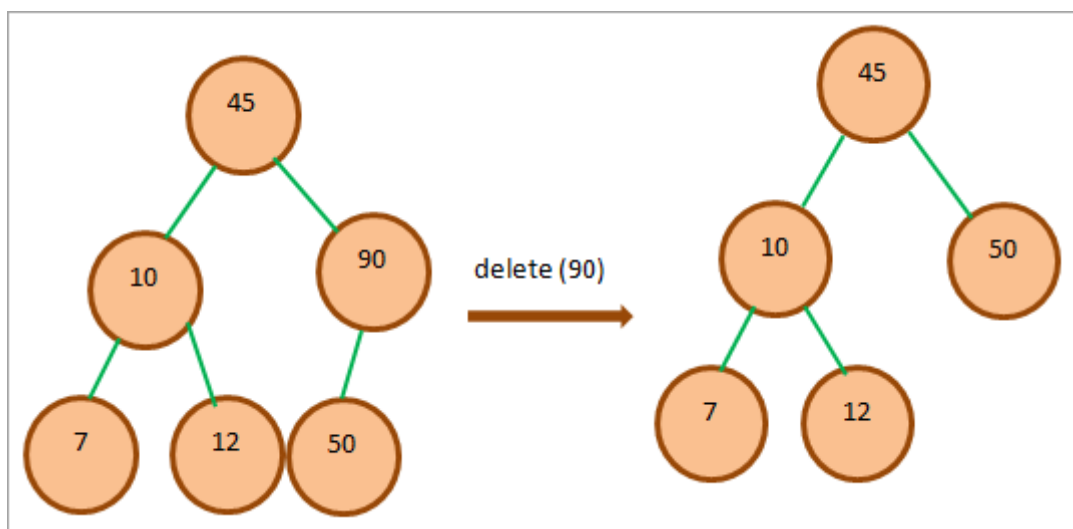
If a node to be deleted is a leaf node, then we can directly delete this node as it has no child nodes. This is shown in the below image.



As shown above, the node 12 is a leaf node and can be deleted straight away.

### **Case 2: Node Has Only One Child**

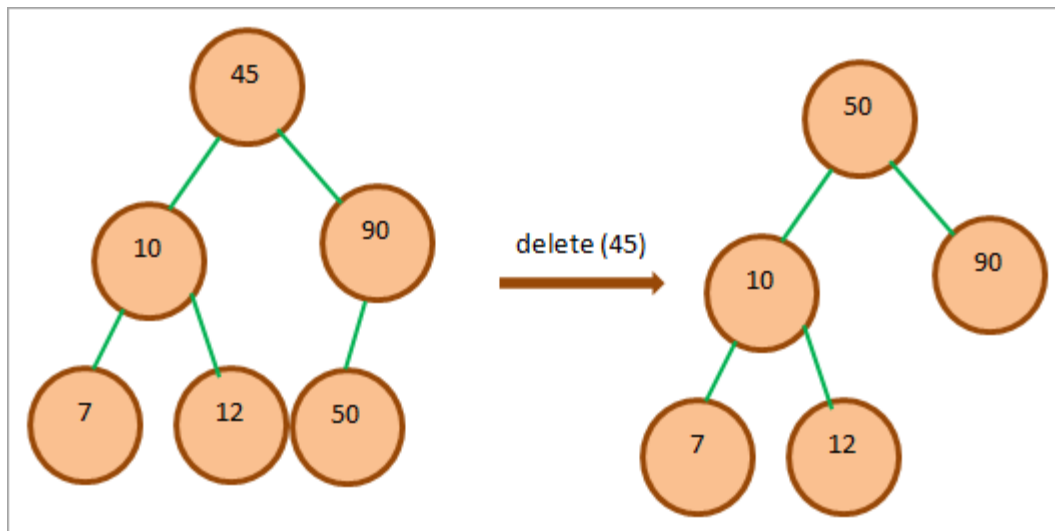
When we need to delete the node that has one child, then we copy the value of the child in the node and then delete the child.



In the above diagram, we want to delete node 90 which has one child 50. So we swap the value 50 with 90 and then delete node 90 which is a child node now.

### **Case 3: Node Has Two Children**

When a node to be deleted has two children, then we replace the node with the inorder (left-root-right) successor of the node or simply said the minimum node in the right subtree if the right subtree of the node is not empty. We replace the node with this minimum node and delete the node.



In the above diagram, we want to delete node 45 which is the root node of BST. We find that the right subtree of this node is not empty. Then we traverse the right subtree and find that node 50 is the minimum node here. So we replace this value in place of 45 and then delete 45.

If we check the tree, we see that it fulfills the properties of a BST. Thus the node replacement was correct.

### Binary Search Tree (BST) Implementation In Java

The following program in Java provides a demonstration of all the above BST operation using the same tree used in illustration as an example.

```
class BST_class {
    //node class that defines BST node
    class Node {
        int key;
        Node left, right;

        public Node(int data){
            key = data;
            left = right = null;
        }
    }
}
```

```

}
// BST root node
Node root;

// Constructor for BST =>initial empty tree
BST_class(){
    root = null;
}
//delete a node from BST
void deleteKey(int key) {
    root = delete_Recursive(root, key);
}

//recursive delete function
Node delete_Recursive(Node root, int key) {
    //tree is empty
    if (root == null) return root;

    //traverse the tree
    if (key < root.key)    //traverse left subtree
        root.left = delete_Recursive(root.left, key);
    else if (key > root.key) //traverse right subtree
        root.right = delete_Recursive(root.right, key);
    else {
        // node contains only one child
        if (root.left == null)
            return root.right;
        else if (root.right == null)

```



```

        return root.left;

    // node has two children;
    //get inorder successor (min value in the right subtree)
    root.key = minValue(root.right);

    // Delete the inorder successor
    root.right = delete_Recursive(root.right, root.key);
}
return root;
}

int minValue(Node root) {
    //initially minval = root
    int minval = root.key;
    //find minval
    while (root.left != null) {
        minval = root.left.key;
        root = root.left;
    }
    return minval;
}

// insert a node in BST
void insert(int key) {
    root = insert_Recursive(root, key);
}

```

//recursive insert function

```
Node insert_Recursive(Node root, int key) {
```

```
    //tree is empty
```

```
    if (root == null) {
```

```
        root = new Node(key);
```

```
        return root;
```

```
    }
```

```
    //traverse the tree
```

```
    if (key < root.key)    //insert in the left subtree
```

```
        root.left = insert_Recursive(root.left, key);
```

```
    else if (key > root.key)    //insert in the right subtree
```

```
        root.right = insert_Recursive(root.right, key);
```

```
    // return pointer
```

```
    return root;
```

```
}
```

// method for inorder traversal of BST

```
void inorder() {
```

```
    inorder_Recursive(root);
```

```
}
```

// recursively traverse the BST

```
void inorder_Recursive(Node root) {
```

```
    if (root != null) {
```

```
        inorder_Recursive(root.left);
```

```
        System.out.print(root.key + " ");
```

```
        inorder_Recursive(root.right);
```

```
    }
```

```
}
```

```
boolean search(int key) {  
    root = search_Recursive(root, key);  
    if (root!= null)  
        return true;  
    else  
        return false;  
}
```

```
//recursive insert function
```

```
Node search_Recursive(Node root, int key) {  
    // Base Cases: root is null or key is present at root  
    if (root==null || root.key==key)  
        return root;  
    // val is greater than root's key  
    if (root.key > key)  
        return search_Recursive(root.left, key);  
    // val is less than root's key  
    return search_Recursive(root.right, key);  
}  
}
```

```
class Main{  
    public static void main(String[] args) {  
        //create a BST object  
        BST_class bst = new BST_class();  
        /* BST tree example  
        45
```

```

    /   \
   10    90
  /  \  /
 7  12 50 */

```

```
//insert data into BST
```

```
bst.insert(45);
```

```
bst.insert(10);
```

```
bst.insert(7);
```

```
bst.insert(12);
```

```
bst.insert(90);
```

```
bst.insert(50);
```

```
//print the BST
```

```
System.out.println("The BST Created with input data(Left-root-right):");
```

```
bst.inorder();
```

```
//delete leaf node
```

```
System.out.println("\nThe BST after Delete 12(leaf node):");
```

```
bst.deleteKey(12);
```

```
bst.inorder();
```

```
//delete the node with one child
```

```
System.out.println("\nThe BST after Delete 90 (node with 1 child):");
```

```
bst.deleteKey(90);
```

```
bst.inorder();
```

```
//delete node with two children
```

```
System.out.println("\nThe BST after Delete 45 (Node with two children):"
```

```
bst.deleteKey(45);
```

```
bst.inorder();
```

```

//search a key in the BST
boolean ret_val = bst.search (50);
System.out.println("\nKey 50 found in BST:" + ret_val );
ret_val = bst.search (12);
System.out.println("\nKey 12 found in BST:" + ret_val );
}
}

```

### Output:

```

The BST Created with input data(Left-root-
right):
7 10 12 45 50 90

The BST after Delete 12(leaf node):
7 10 45 50 90

The BST after Delete 90 (node with 1 child):
7 10 45 50

The BST after Delete 45 (Node with two children):
7 10 50

Key 50 found in BST:true

```

## Binary Search Tree (BST) Traversal In Java

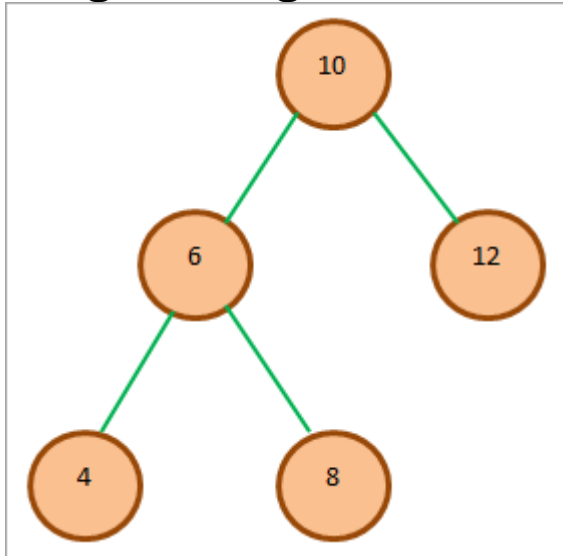
A tree is a hierarchical structure, thus we cannot traverse it linearly like other data structures such as arrays. Any type of tree needs to be traversed in a special way so that all its subtrees and nodes are visited at least once.

Depending on the order in which the root node, left subtree and right subtree are traversed in a tree, there are certain traversals as shown below:

- Inorder Traversal
- Preorder Traversal
- PostOrder Traversal

All the above traversals use depth-first technique i.e. the tree is traversed depthwise.

The trees also use the breadth-first technique for traversal. The approach using this technique is called **“Level Order”** traversal. **In this section, we will demonstrate each of the traversals using following BST as an example.**



**With the BST as shown in the above diagram, the level order traversal for the above tree is :**

Level Order Traversal: 10 6 12 4 8

### ***Inorder Traversal***

The inorder traversal approach traversed the BST in the order, **Left subtree=>RootNode=>Right subtree**. The inorder traversal provides a decreasing sequence of nodes of a BST.

**The algorithm InOrder (bstTree) for InOrder Traversal is given below.**

1. Traverse the left subtree using InOrder (left\_subtree)
2. Visit the root node.
3. Traverse the right subtree using InOrder (right\_subtree)

**The inorder traversal of the above tree is:**

4      6      8      10      12

As seen, the sequence of the nodes as a result of the inorder traversal is in decreasing order.

### ***Preorder Traversal***

In preorder traversal, the root is visited first followed by the left subtree and right subtree. Preorder traversal creates a copy of the tree. It can also be used in expression trees to obtain prefix expression.

**The algorithm for PreOrder (bst\_tree) traversal is given below:**

1. Visit the root node
2. Traverse the left subtree with PreOrder (left\_subtree).
3. Traverse the right subtree with PreOrder (right\_subtree).

**The preorder traversal for the BST given above is:**

10    6    4    8    12

### ***PostOrder Traversal***

The postOrder traversal traverses the BST in the order: **Left subtree->Right subtree->Root node**. PostOrder traversal is used to delete the tree or obtain the postfix expression in case of expression trees.

**The algorithm for postOrder (bst\_tree) traversal is as follows:**

1. Traverse the left subtree with postOrder (left\_subtree).
2. Traverse the right subtree with postOrder (right\_subtree).
3. Visit the root node

**The postOrder traversal for the above example BST is:**

4    8    6    12    10

Next, we will implement these traversals using the depth-first technique in a Java implementation.

```
//define node of the BST
```

```
class Node {
```

```
    int key;
```

```
    Node left, right;
```

```
    public Node(int data){
```

```
        key = data;
```

```
        left = right = null;
```

```
    }
```

```
}
```

```
//BST class
```

```
class BST_class {
```

```

// BST root node
Node root;

BST_class(){
    root = null;
}

//PostOrder Traversal - Left:Right:rootNode (LRn)
void postOrder(Node node) {
    if (node == null)
        return;

    // first traverse left subtree recursively
    postOrder(node.left);

    // then traverse right subtree recursively
    postOrder(node.right);

    // now process root node
    System.out.print(node.key + " ");
}

// InOrder Traversal - Left:rootNode:Right (LnR)
void inOrder(Node node) {
    if (node == null)
        return;

    //first traverse left subtree recursively
    inOrder(node.left);

```



```

        //then go for root node
        System.out.print(node.key + " ");

        //next traverse right subtree recursively
        inOrder(node.right);
    }

//PreOrder Traversal - rootNode:Left:Right (nLR)
    void preOrder(Node node) {
        if (node == null)
            return;

        //first print root node first
        System.out.print(node.key + " ");
        // then traverse left subtree recursively
        preOrder(node.left);
        // next traverse right subtree recursively
        preOrder(node.right);
    }
// Wrappers for recursive functions
    void postOrder_traversal() {
        postOrder(root); }
    void inOrder_traversal() {
        inOrder(root); }
    void preOrder_traversal() {
        preOrder(root); }
}

class Main{

```

```

public static void main(String[] args) {
    //construct a BST
    BST_class tree = new BST_class();
    /*      45
           //  \
          10  90
         //  \
        7   12    */
    tree.root = new Node(45);
    tree.root.left = new Node(10);
    tree.root.right = new Node(90);
    tree.root.left.left = new Node(7);
    tree.root.left.right = new Node(12);
    //PreOrder Traversal
    System.out.println("BST => PreOrder Traversal:");
    tree.preOrder_traversal();
    //InOrder Traversal
    System.out.println("\nBST => InOrder Traversal:");
    tree.inOrder_traversal();
    //PostOrder Traversal
    System.out.println("\nBST => PostOrder Traversal:");
    tree.postOrder_traversal();
}
}

```

**Output:**

```
BST => PreOrder Traversal:
45 10 7 12 90
BST => InOrder Traversal:
7 10 12 45 90
BST => PostOrder Traversal:
7 12 10 90 45
```

## **//Java program to implement Dictionary Operations using BST**

```
import java.util.*;
```

```
class pair<K extends Comparable<K>,V>{
    K key;
    V value;
    pair(K a, V b){
        key=a;
        value=b;
    }
    public String toString(){
        return key+":"+value;
    }
}
```

```
class node<K extends Comparable<K>,V>{
```

```

pair<K,V> data;
node<K,V> l,r;
node(pair<K,V> d, node<K,V> n,node<K,V> m){
    data=d;
    l=n;
    r=m;
}
}

```

```

class BST<K extends Comparable<K>,V> {
    node<K,V> prev,temp,nn,root;
    int cnt;

    BST(){
        cnt=0;
        root=null;
    }

```

```

    void insert(pair<K,V> p){
        nn=new node<>(p,null,null);
        if(root==null)
            root=nn;
        else{

```

```

    prev=null;
    temp=root;
    while(temp!=null){
        if(p.key.compareTo(temp.data.key)<0){
            prev=temp;
            temp=temp.l;
        }
        else{
            prev=temp;
            temp=temp.r;
        }
    }
    if(prev.data.key.compareTo(p.key)>0)
        prev.l=nn;
    else
        prev.r=nn;

    cnt++;
}
}

```

```

pair<K,V> find(K key){
    temp=root;

```

```

        while(temp!=null && !key.equals(temp.data.key)){
            if(key.compareTo(temp.data.key)<0)
                temp=temp.l;
            else
                temp=temp.r;
        }
        if(temp==null)
            return null;
        else
            return temp.data;
    }

    void delete(K key)
    {
        if(root==null)
        {
            System.out.println("BST is empty");
            return;
        }

        node<K,V>
temp=root,ptemp=null,predecessor,ppredecessor,c=null;

        while(temp!=null                                &&
key.compareTo(temp.data.key)!=0)
        {
            ptemp=temp;

```

```

        if(key.compareTo(temp.data.key)<0)
            temp=temp.l;
        else
            temp=temp.r;
    }
    //Key is not found
    if(temp==null)
    {
        System.out.println("Key is not found");
        return;
    }
    //Key is found
    //Case3:Node with two children
    if(temp.l!=null && temp.r!=null)
    {
        //Replacing with inorder successor
        predeccessor=temp.r;
        ppredeccessor=temp;
        //locating inorder predeccessor
        while(predeccessor.l!=null)
        {
            ppredeccessor=predeccessor;
            predeccessor=predeccessor.l;
        }
    }

```

```

    }

    //replace temp.data with predecessor.data
    temp.data=predecessor.data;

    //convert predecessor to temp
    temp=predecessor;
    ptemp=ppredecessor;
}

//Single child node
//mark child of temp
if(temp.l!=null)
    c=temp.l;
else if(temp.r!=null)
    c=temp.r;

//front node to be deleted
if(root==temp)
{
    root=c;
}
else
{
    if(ptemp.l==temp)
        ptemp.l=c;
    else if(ptemp.r==temp)

```



```
        ptemp.r=c;
    }
```

```
    cnt--;
}
```

```
void inorder(){
    inorder_Recursive(root);
}
```

```
void inorder_Recursive(node<K,V> n){
    if(n!=null){
        inorder_Recursive(n.l);
        System.out.print(n.data+" ");
        inorder_Recursive(n.r);
    }
}
```

```
void preorder(){
    preorder_Recursive(root);
}
```

```
void preorder_Recursive(node<K,V> n){
    if(n!=null){
        System.out.print(n.data+" ");
    }
}
```

```

        preorder_Recursive(n.l);
        preorder_Recursive(n.r);
    }
}

void postorder(){
    postorder_Recursive(root);
}

void postorder_Recursive(node<K,V> n){
    if(n!=null){
        postorder_Recursive(n.l);
        postorder_Recursive(n.r);
        System.out.print(n.data+" ");
    }
}

}

class BSTDemo{
public static void main(String args[]){
    Scanner sc=new Scanner(System.in);
    BST<Integer,String> bst=new BST<Integer,String>();
    pair<Integer,String> p;

```

```

do{
    System.out.println();
    System.out.println("1.Insert()");
    System.out.println("2.Search()");
    System.out.println("3.Delete()");
    System.out.println("4.InOrder()");
    System.out.println("5.PreOrder()");
    System.out.println("6.PostOrder()");
    System.out.println("7.Exit");
    System.out.println("Enter choice: ");
    int ch=sc.nextInt();

    switch(ch){
        case 1: p=new pair<>(sc.nextInt(),sc.next());
                bst.insert(p);
                break;
        case 2: System.out.println("Enter the element to
Search:");
                int search=sc.nextInt();
                p=bst.find(search);
                if(p==null)
                    System.out.println("not found");
                else

```

```

        System.out.println(p);

        break;

        case 3:

            System.out.println("Enter the element to be deleted:");

            int del=sc.nextInt();

            bst.delete(del);

        case 4:bst.inorder();

            break;

        case 5: bst.preorder();

            break;

        case 6: bst.postorder();

            break;

        case 7: System.exit(0);

    }

    }while(true);

}

}

```

/\*An implementation of a Binary Search Tree (BST) using the Collection API in Java. The BST class includes recursive procedures to perform in-order, pre-order, and post-order traversals.\*/

```
import java.util.Collection;
```

```
import java.util.LinkedList;
```

```
class BST<T extends Comparable<T>> {
```

```
    private Node<T> root;
```

```
    static class Node<T> {
```

```
        T data;
```

```
        Node<T> left;
```

```
        Node<T> right;
```

```
        Node(T data) {
```

```
            this.data = data;
```

```
            left = null;
```

```
            right = null;
```

```
        }
```

```
    }
```

```
    // Insert a value into the BST
```

```
    public void insert(T value) {
```

```
        root = insertRecursive(root, value);
```

```
    }
```

```
    // Recursive helper method to insert a value
```

```
private Node<T> insertRecursive(Node<T> current, T value) {  
    if (current == null) {  
        return new Node<>(value);  
    }
```

```
    if (value.compareTo(current.data) < 0) {  
        current.left = insertRecursive(current.left, value);  
    } else if (value.compareTo(current.data) > 0) {  
        current.right = insertRecursive(current.right, value);  
    }
```

```
    return current;  
}
```

```
// Perform in-order traversal
```

```
public Collection<T> inOrderTraversal() {  
    Collection<T> result = new LinkedList<>();  
    inOrderRecursive(root, result);  
    return result;  
}
```

```
// Recursive helper method for in-order traversal
```

```
private void inOrderRecursive(Node<T> current, Collection<T>
result) {
```

```
    if (current != null) {
```

```
        inOrderRecursive(current.left, result);
```

```
        result.add(current.data);
```

```
        inOrderRecursive(current.right, result);
```

```
    }
```

```
}
```

```
// Perform pre-order traversal
```

```
public Collection<T> preOrderTraversal() {
```

```
    Collection<T> result = new LinkedList<>();
```

```
    preOrderRecursive(root, result);
```

```
    return result;
```

```
}
```

```
// Recursive helper method for pre-order traversal
```

```
private void preOrderRecursive(Node<T> current, Collection<T>
result) {
```

```
    if (current != null) {
```

```
        result.add(current.data);
```

```
        preOrderRecursive(current.left, result);
```

```
        preOrderRecursive(current.right, result);
```

```
    }
```

```
}
```

```
// Perform post-order traversal
```

```
public Collection<T> postOrderTraversal() {
```

```
    Collection<T> result = new LinkedList<>();
```

```
    postOrderRecursive(root, result);
```

```
    return result;
```

```
}
```

```
// Recursive helper method for post-order traversal
```

```
private void postOrderRecursive(Node<T> current,  
Collection<T> result) {
```

```
    if (current != null) {
```

```
        postOrderRecursive(current.left, result);
```

```
        postOrderRecursive(current.right, result);
```

```
        result.add(current.data);
```

```
    }
```

```
}
```

```
}
```

```
public class Main {
```

```
    public static void main(String[] args) {
```

```
        BST<Integer> bst = new BST<>();
```



```
bst.insert(5);
```

```
bst.insert(3);
```

```
bst.insert(7);
```

```
bst.insert(2);
```

```
bst.insert(4);
```

```
bst.insert(6);
```

```
bst.insert(8);
```

```
System.out.println("In-order traversal: " +  
bst.inOrderTraversal());
```

```
System.out.println("Pre-order traversal: " +  
bst.preOrderTraversal());
```

```
System.out.println("Post-order traversal: " +  
bst.postOrderTraversal());
```

```
}
```

```
}
```

### **Lab Session Program 7:**

Write a java program to create a class called Person with income, age and name as its members. Read set A of persons from a user and compute the following sets: i) Set B of persons whose age > 60  
ii) Set C of persons whose income < 10000 and iii)  $B \cap C$

### **Program:**

```
import java.util.HashSet;
```

```
import java.util.Scanner;
```

```
import java.util.Set;
```

```
class Person {
```

```
    private String name;
```

```
    private int age;
```

```
    private double income;
```

```
    public Person(String name, int age, double income) {
```

```
        this.name = name;
```

```
        this.age = age;
```

```
        this.income = income;
```

```
    }
```

```
    public int getAge() {
```

```
        return age;
```

```
    }
```

```
    public double getIncome() {
```

```
        return income;
```

```
    }
```

```
@Override
```

```
public String toString() {
```

```
    return "Person [name=" + name + ", age=" + age + ", income=" + income + "];"
```

```
}
```

```
}
```

```
public class PersonSet {
```

```
    public static void main(String[] args) {
```

```
        Set<Person> setA = new HashSet<>();
```

```
        Set<Person> setB = new HashSet<>();
```

```
        Set<Person> setC = new HashSet<>();
```

```
        Scanner scanner = new Scanner(System.in);
```

```
        System.out.print("Enter the number of persons: ");
```

```
        int numPersons = scanner.nextInt();
```

```
        scanner.nextLine();
```

```
        for (int i = 0; i < numPersons; i++) {
```

```
            System.out.println("Enter details for Person " + (i + 1) + ":");
```

```
            System.out.print("Name: ");
```

```
            String name = scanner.nextLine();
```

```
System.out.print("Age: ");
```

```
int age = scanner.nextInt();
```

```
System.out.print("Income: ");
```

```
double income = scanner.nextDouble();
```

```
scanner.nextLine(); // Consume the newline character
```

```
Person person = new Person(name, age, income);
```

```
setA.add(person);
```

```
}
```

```
for (Person person : setA) {
```

```
    if (person.getAge() > 60) {
```

```
        setB.add(person);
```

```
    }
```

```
    if (person.getIncome() < 10000) {
```

```
        setC.add(person);
```

```
    }
```

```
}
```

```
System.out.println("Set B (persons whose age > 60):");
```

```
printPersonSet(setB);
```

```
System.out.println("Set C (persons whose income < 10000):");  
printPersonSet(setC);
```

```
Set<Person> setBIntersectionC = new HashSet<>(setB);  
setBIntersectionC.retainAll(setC);
```

```
System.out.println("Set B Intersection of Set C :");  
printPersonSet(setBIntersectionC);  
}
```

```
public static void printPersonSet(Set<Person> set) {  
    for (Person person : set) {  
        System.out.println(person);  
    }  
    System.out.println();  
}
```

}This program creates a **Person** class with the necessary fields and constructors. It then reads the number of persons from the user and prompts for their details (name, age, and income). The program populates the **setA** with the entered persons and computes the sets B and C based on the given conditions. Finally, it prints the sets B, C, and their Intersection (set B Intersection C).

# **Non-recursive traversals of BST**

An example of a Java program that implements binary search tree (BST) traversals without recursion using generics:

```
import java.util.Stack;

class Node<T extends Comparable<T>> {

    T data;

    Node<T> left, right;

    public Node(T item) {

        data = item;

        left = right = null;

    }

}

class BST<T extends Comparable<T>> {

    Node<T> root;

    public BST() {

        root = null;

    }

    // Insert a node in the BST

    public void insert(T item) {

        root = insertRec(root, item);

    }

}
```

```
}
```

```
private Node<T> insertRec(Node<T> root, T item) {
```

```
    if (root == null) {
```

```
        root = new Node<>(item);
```

```
        return root;
```

```
    }
```

```
    if (item.compareTo(root.data) < 0)
```

```
        root.left = insertRec(root.left, item);
```

```
    else if (item.compareTo(root.data) > 0)
```

```
        root.right = insertRec(root.right, item);
```

```
    return root;
```

```
}
```

```
// Inorder traversal without recursion
```

```
public void inorder() {
```

```
    if (root == null)
```

```
        return;
```

```
    Stack<Node<T>> stack = new Stack<>();
```

```
    Node<T> current = root;
```

```
while (current != null || !stack.isEmpty()) {
```

```
    while (current != null) {
```

```
        stack.push(current);
```

```
        current = current.left;
```

```
    }
```

```
    current = stack.pop();
```

```
    System.out.print(current.data + " ");
```

```
    current = current.right;
```

```
}
```

```
}
```

```
// Preorder traversal without recursion
```

```
public void preorder() {
```

```
    if (root == null)
```

```
        return;
```

```
    Stack<Node<T>> stack = new Stack<>();
```

```
    stack.push(root);
```

```
    while (!stack.isEmpty()) {
```



```
Node<T> current = stack.pop();
```

```
System.out.print(current.data + " ");
```

```
if (current.right != null)
```

```
    stack.push(current.right);
```

```
if (current.left != null)
```

```
    stack.push(current.left);
```

```
}
```

```
}
```

```
// Postorder traversal without recursion
```

```
public void postorder() {
```

```
    if (root == null)
```

```
        return;
```

```
Stack<Node<T>> stack1 = new Stack<>();
```

```
Stack<Node<T>> stack2 = new Stack<>();
```

```
stack1.push(root);
```

```
while (!stack1.isEmpty()) {
```

```
    Node<T> current = stack1.pop();
```

```
    stack2.push(current);
```

```
        if (current.left != null)
            stack1.push(current.left);

        if (current.right != null)
            stack1.push(current.right);
    }

    while (!stack2.isEmpty()) {
        Node<T> current = stack2.pop();
        System.out.print(current.data + " ");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        BST<Integer> bst = new BST<>();

        bst.insert(45);
        bst.insert(10);
        bst.insert(7);
        bst.insert(90);
    }
}
```

```

        bst.insert(12);

        bst.insert(50);

        bst.insert(13);

        System.out.print("Inorder traversal: ");

        bst.inorder();

        System.out.println();

        System.out.print("Preorder traversal: ");

        bst.preorder();

        System.out.println();

        System.out.print("Postorder traversal: ");

        bst.postorder();

        System.out.println();

    }

}

```

This program defines a **Node** class that represents a node in the binary search tree and a **BST** class that implements the binary search tree operations. The **BST** class uses a generic type **T** that extends the **Comparable** interface to allow comparisons between elements.

The **inorder()**, **preorder()**, and **postorder()** methods perform the respective traversals on the binary search tree without using recursion. They use a stack to keep track of nodes to be visited.

In the **main()** method, a binary search tree is created, and some nodes are inserted into it. Then, the inorder, preorder, and postorder traversals are performed and printed to the console.

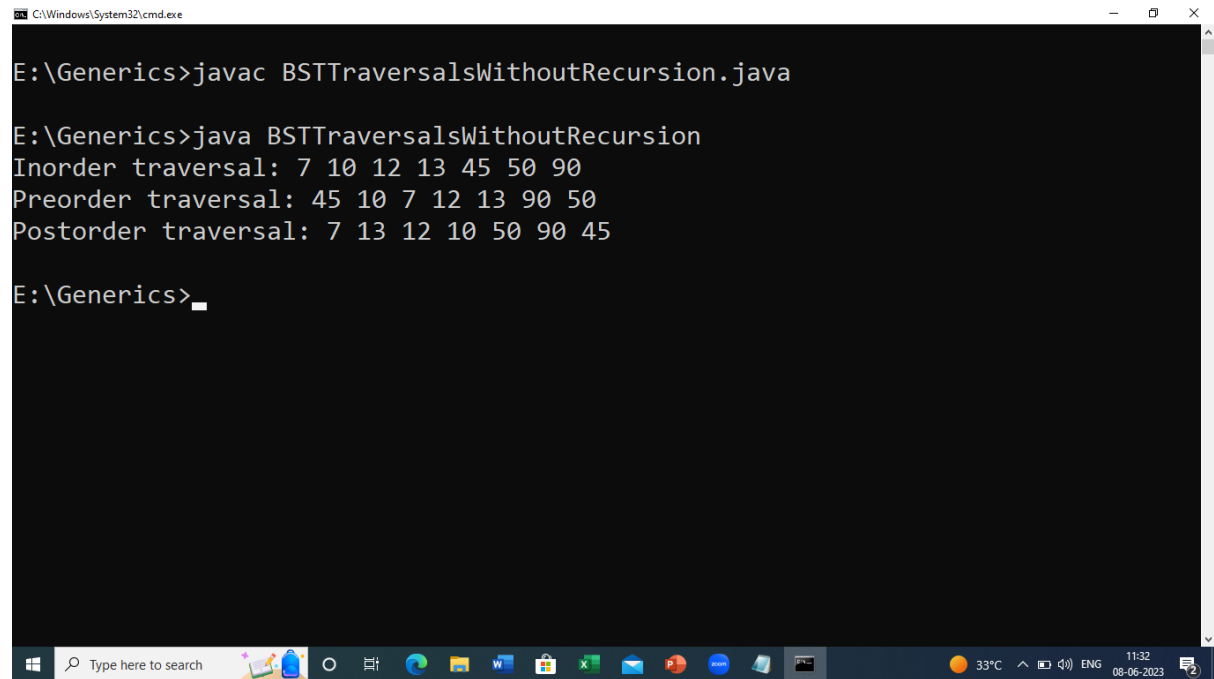
### **Output:**

```
C:\Windows\System32\cmd.exe

E:\Generics>javac BSTTraversalsWithoutRecursion.java

E:\Generics>java BSTTraversalsWithoutRecursion
Inorder traversal: 7 10 12 13 45 50 90
Preorder traversal: 45 10 7 12 13 90 50
Postorder traversal: 7 13 12 10 50 90 45

E:\Generics>
```

A screenshot of a Windows command prompt window. The title bar at the top reads "C:\Windows\System32\cmd.exe". The command prompt shows the user's current directory as "E:\Generics". The user has executed the command "javac BSTTraversalsWithoutRecursion.java", which compiled the Java file. Then, the user executed "java BSTTraversalsWithoutRecursion", which ran the program. The program's output is displayed on the next three lines: "Inorder traversal: 7 10 12 13 45 50 90", "Preorder traversal: 45 10 7 12 13 90 50", and "Postorder traversal: 7 13 12 10 50 90 45". The prompt "E:\Generics>" is shown again on the next line, indicating the program has finished. At the bottom of the window, the Windows taskbar is visible, showing the Start button, a search bar with the text "Type here to search", and several application icons including Edge, File Explorer, Word, and others. The system tray on the right shows the temperature as 33°C, the time as 11:32, and the date as 08-06-2023.