Unit-III

Dictionary:

Dictionary class is an abstract class that represents a collection of key-value pairs, where each key is associated with a unique value. It is a subclass of the Map interface and provides a skeletal implementation of the Map interface. It provides methods to manipulate the data, such as adding, removing, and retrieving elements based on their keys.

Since Dictionary is an abstract class, you cannot create an instance of it directly. Instead, you need to use one of its concrete subclasses such as Hashtable or Properties to work with it.

It was part of the Java Collections Framework introduced in Java 1.0 but has been largely replaced by the java.util.Map interface since Java 1.2.

Constructor: The class has only a constructor called a sole constructor.

Syntax: public Dictionary()

The Dictionary class provides methods for adding, removing, and retrieving key-value pairs from the collection. Some of the methods in the Dictionary class include:

1. **put (Object key, Object value):** This method adds a key-value pair to the dictionary.

```
Dictionary<String, Integer> dict = new Hashtable<>();
dict.put("one", 1);
dict.put("two", 2);
dict.put("three", 3);
```

Note: If the key already exists, its corresponding value is replaced with the new value, and the old value is returned. If the key is new, null is returned.

2. get(Object key): This method returns the value associated with the specified key, or null if the key is not found.

```
Integer value = dict.get("two"); // returns 2
```

3. remove(Object key): This method removes the key-value pair associated with the specified key.

```
dict.remove("two");
```

4. size(): This method returns the number of key-value pairs in the dictionary.

```
int size = dict.size(); // returns 2
```

5. keys(): This method returns an enumeration of all the keys in the dictionary.

```
Enumeration<String> keys = dict.keys();
while (keys.hasMoreElements()) {
   String key = keys.nextElement();
   System.out.println(key);
}
```

6. elements(): Returns an enumeration of the values stored in the dictionary.

```
Enumeration<String> values = dict.elements();
while (keys.hasMoreElements()) {
   String value = values.nextElement();
   System.out.println(value);
}
```

7. isEmpty(): Returns true if the dictionary is empty, and false otherwise.

The Dictionary class is useful when you need to maintain a collection of key-value pairs and perform operations on them, such as looking up a value by its key. However, it is important to note that the Dictionary class is considered to be a legacy class and is not recommended for new code. Instead, you should use the Map interface and its concrete implementations, such as

HashMap or LinkedHashMap, which provide more efficient and flexible implementations of the key-value pair collection.

Linear List Representation:

A linear list is a data structure that stores a collection of elements in a linear sequence, where each element has a specific position or index within the list. The elements in a linear list are typically stored contiguously in memory, allowing for efficient traversal and accessing of elements based on their position.

There are several techniques or variations of linear lists that can be used to organize and manipulate the elements. Some of the common techniques include:

Array-based List: In this technique, the elements of the linear list are stored in an array. The array provides direct access to any element based on its index. Array-based lists have a fixed size, and if the list needs to grow beyond the capacity of the underlying array, a new larger array needs to be allocated and the elements copied over.

Linked List: In a linked list, each element is stored in a node object, and the nodes are connected through references or pointers. Each node contains the element data and a reference to the next node in the list. Linked lists can be singly linked, where each node has a reference to the next node, or doubly linked, where each node has references to both the next and previous nodes. Linked lists allow for dynamic resizing since nodes can be added or removed by adjusting the references.

Doubly Ended Linked List: This is a variation of a linked list where each node has references to both the next and previous nodes. It allows for efficient insertion and deletion at both ends of the list. It is particularly useful in scenarios where frequent insertion or removal operations are required at the beginning or end of the list.

Circular Linked List: In a circular linked list, the last node's next reference points back to the head of the list, forming a circular structure. This allows for easy traversal from any node to any other node in the list. Circular linked lists are often used in scenarios where cyclic behavior is required, such as scheduling algorithms or round-robin systems.

Skip List: A skip list is a probabilistic data structure that allows for efficient searching and insertion operations. It consists of multiple layers of linked lists, with each higher-level list containing a subset of elements from the lower-level list. Skip lists use a probabilistic technique to determine the number of levels for each element, providing an efficient balance between search complexity and space usage.

These techniques offer different trade-offs in terms of memory usage, efficiency of insertion, deletion, and search operations, and flexibility in resizing or reordering elements. The choice of technique depends on the specific requirements and constraints of the application at hand.

An array-based linear list, also known as an array list, is a linear list implementation that stores elements in a contiguous block of memory using an array. The array provides direct access to any element based on its index.

Here's an example to illustrate an array-based linear list for integers:

Suppose we want to create an array-based linear list to store the following integers: 10, 5, 3, 7, and 2.

- Initially, the array-based linear list is empty, and we have an array of size 5 (to accommodate the 5 elements we want to store):
  Array-based Linear List: [_, _, _, _, _]
- We add the elements to the array-based linear list:
  Array-based Linear List: [10, 5, 3, 7, 2]
- To access an element, we use its index. For example, to access the element at index 2:
  Element at index 2: 3
- We can easily modify an element at a specific index. For example, let's change the element at index 1 to 8
  Array-based Linear List: [10, 8, 3, 7, 2]
- We can also remove an element from the array-based linear list. Let's remove the element at index 3:
  Array-based Linear List: [10, 8, 3, _, 2], The remaining elements shift to fill the gap left by the removed element.
- If we want to add more elements and the array is full, the array-based linear list needs to expand. For example, let's add the element 15 to the list:
  Array-based Linear List: [10, 8, 3, 15, 2, _, _, _, _, _]
- The array-based linear list automatically increases its capacity by creating a new, larger array and copying the existing elements to the new array.

Array-based linear lists provide efficient constant-time access to elements by index, making them suitable for random access scenarios. However, they may require resizing and copying the entire array when the capacity is exceeded, which can be inefficient in terms of time and memory. Additionally, adding or removing elements from the middle of the list can be expensive as it requires shifting the subsequent elements. The choice of data structure depends on the specific requirements of the application.

implementation of sorted list using user defined generic classes

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

class SortedList<K, V extends Comparable<V>> {
   private List<Pair<K, V>> items;
   public SortedList() {
      items = new ArrayList<>();
   }
   private static class Pair<K, V> {
      private K key;
      private V value;
```

```
        public Pair(K key, V value) {
            this.key = key;
            this.value = value;
        }
        public K getKey() {
            return key;
        }
        public V getValue() {
            return value;
        }
        public String toString() {
            return key + "=" + value;
        }
    }
    public void add(K key, V value) {
        Pair<K, V> pair = new Pair<>(key, value);
        int index = findInsertionIndex(value);
        items.add(index, pair);
    }
    public String toString() {
        return items.toString();
    }
    private int findInsertionIndex(V value) {
        int index = 0;
        while (index < items.size() && items.get(index).getValue().compareTo(value) < 0)
{
            index++;
        }
        return index;
    }
}
public class  sortedListGenClass{
    public static void main(String[] args) {
        SortedList<String, Integer> sortedList = new SortedList<>();
        Scanner scanner = new Scanner(System.in);
        String input;
        do {
            System.out.print("Enter key-value pair (or 'q' to quit): ");
            input = scanner.nextLine();
            if (!input.equalsIgnoreCase("q")) {
                String[] pair = input.split(" ");
                if (pair.length == 2) {
                    String key = pair[0];
                    int value = Integer.parseInt(pair[1]);
                    sortedList.add(key, value);
                    System.out.println("Item added to the sorted list.");
```

```
                } else {
                    System.out.println("Invalid input. Please provide a key-value pair.");
                }
            }
        } while (!input.equalsIgnoreCase("q"));
        System.out.println("Final sorted list: " + sortedList);
    }
}
```

implementation of sorted list using LinkedList Collections class.

```
import java.util.LinkedList;
import java.util.Scanner;
class SortedList<K extends Comparable<K>, V> {
    private LinkedList<Pair<K, V>> items;
    public SortedList() {
        items = new LinkedList<>();
    }
    private static class Pair<K, V> {
        private K key;
        private V value;

        public Pair(K key, V value) {
            this.key = key;
            this.value = value;
        }
        public K getKey() {
            return key;
        }
        public V getValue() {
            return value;
        }
        public String toString() {
            return key + "=" + value;
        }
    }
    public void add(K key, V value) {
        Pair<K, V> pair = new Pair<>(key, value);
        int index = 0;
        while (index < items.size() && key.compareTo(items.get(index).getKey()) > 0) {
            index++;
        }
        items.add(index, pair);
    }
    public String toString() {
        return items.toString();
    }
}
```

```
public class  sortedListLinkedList{
   public static void main(String[] args) {
      SortedList<String, Integer> sortedList = new SortedList<>();
      Scanner scanner = new Scanner(System.in);
      String input;
      do {
         System.out.print("Enter key-value pair (or 'q' to quit): ");
         input = scanner.nextLine();
         if (!input.equalsIgnoreCase("q")) {
            String[] pair = input.split(" ");
            if (pair.length == 2) {
               String key = pair[0];
               int value = Integer.parseInt(pair[1]);
               sortedList.add(key, value);
               System.out.println("Item added to the sorted list.");
            } else {
               System.out.println("Invalid input. Please provide a key-value pair.");
            }
         }
      } while (!input.equalsIgnoreCase("q"));

      System.out.println("Final sorted list: " + sortedList);
   }
}
```

Sorted Chain:

Sorted Chain is a data structure that maintains a sorted collection of elements using a linked list. It guarantees that the elements in the chain are always sorted in ascending order.

In a sorted chain, each element is stored in a node, and the nodes are connected through references to create a linked list. The order of the nodes in the list reflects the sorted order of the elements.

Here's an example to illustrate how a Sorted Chain works:

Let's say we have an empty Sorted Chain. We start by inserting the elements 5, 2, 3, 1, and 9 in that order.

- Initially, the Sorted Chain is empty:
  Sorted Chain:
- We insert the element 5. Since the chain is empty, the new node becomes the head of the chain:
  Sorted Chain: 5
- We insert the element 2. Since 2 is smaller than 5, it becomes the new head of the chain, and the previous head (5) is connected to it:
  Sorted Chain: 2 -> 5
- We insert the element 3. We start from the head (2) and compare the values. Since 3 is greater than 2 and smaller than 5, it becomes the new node between 2 and 5:
  Sorted Chain: 2 -> 3 -> 5

- We insert the element 1. It is smaller than the current head (2), so it becomes the new head and is connected to the previous head:
  Sorted Chain: 1 -> 2 -> 3 -> 5
- Finally, we insert the element 9. Starting from the head, we compare the values until we find the appropriate position for 9. It is greater than all previous elements, so it becomes the last node in the chain:
  Sorted Chain: 1 -> 2 -> 3 -> 5 -> 9

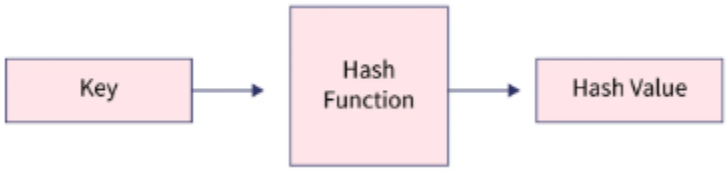
As you can see, the elements are always maintained in sorted order within the Sorted Chain. This allows for efficient insertion of new elements while ensuring that the chain remains sorted at all times.

Hashing

Hashing 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.

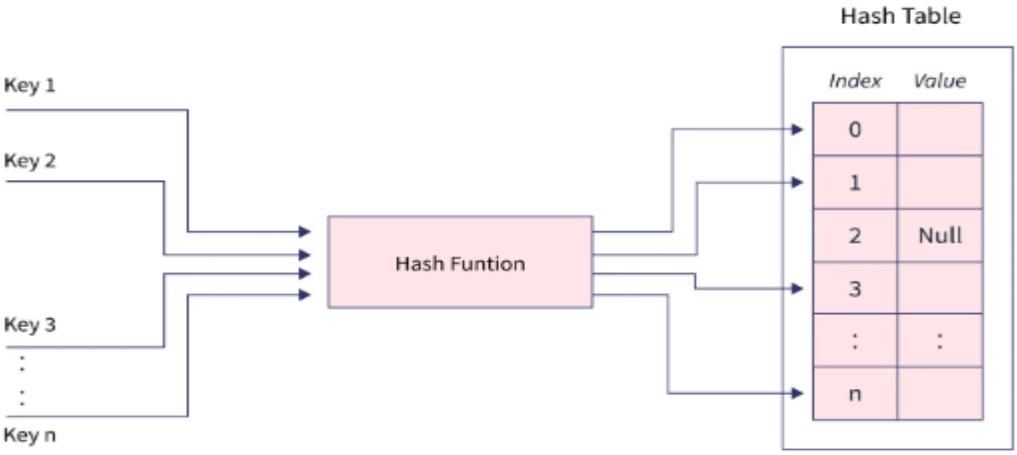
Hash Function

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.



Hash table

A hash table is an array that holds pointers to data that corresponds to a hashed key. Hash table uses hash values as the location index to store the associated data in the array.



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

A good hashing algorithm must have following characteristics:

- There is no limit to the length of a message
- Message digests are generated with a fixed length
- A message digest can be computed quickly (and easily)
- Message digests cannot be generated from the hash - they are irreversible
- Message values change dramatically when small changes are made
- The hash is collision-free because two different messages can't result in the same hash value

Types of Hash functions

Many hash functions use alphanumeric or numeric keys. The main hash functions cover -

1) Division Method.
2) Mid Square Method.
3) Folding Method.

1. Division Method

The division method is the simplest and easiest method used to generate a hash value. In this hash function, the value of k is divided by M and uses the remainder as obtained.

Formula -:h(K) = k mod M (k%M)

(where k = key value and M = the size of the hash table)

Advantages -

- This method works well for any value of M
- The division approach is extremely quick because it only calls for one operation.

Disadvantages -

- This may lead to poor performance as consecutive keys are mapped to consecutive hash values in the hash table
- There are situations when choosing the value of M requires particular caution.

Example -

k = 1320
M = 11
h (1320) = 1320 mod 11
= 0

2.Mid Square Method

The steps involved in computing this hash method include the following -

1. Squaring the value of k (like k*k)
2. Extract the hash value from the middle r digits.

Formula : h(K) = h(k x k)

(where k = key value)

Advantages -

- Since most or all of the key value's digits contribute to the outcome, this strategy performs well. The middle digits of the squared result are produced by a process in which all of the essential digits participate.
- The top or bottom digits of the original key value do not predominate in the outcome.

Disadvantages -

- One of this method's constraints is the size of the key; if the key is large, its square will have twice as many digits.
- Chance of repeated collisions.

Example -

Let's take the hash table with 200 memory locations and r = 2, as decided on the size of the mapping in the table.

- k = 50
- Therefore,
- k = k x k
- = 50 x 50
- = 2500
- Thus,
- h(50) = 50

3.Folding Method

There are two steps in this method -

1. The key-value k should be divided into a specific number of parts, such as k1, k2, k3,..., kn, each having the very same number of digits aside from the final component, which may have fewer digits than the remaining parts.
2. Add each component separately. The last carry, if any, is disregarded to determine the hash value.

Formula: k = k1, k2, k3, k4, ….., kn

s = k1+ k2 + k3 + k4 +….+ kn

h(K)= s

(Where, s = addition of the parts of key k)

Advantages -

- Breaks up the key value into precise equal-sized segments for an easy hash value
- Independent of distribution in a hash table

Disadvantages -

- Sometimes inefficient if there are too many collisions

Example -

- k = 54321
- k1 = 54 ; k2 = 32 ; k3 = 1
- Therefore,
- s = k1 + k2 + k3
- = 54 + 32+ 1
- = 87
- Thus,
- h (k) = 87

Choosing a good hash function

- Creating an effective hash function that distributes the added item's index value evenly across the database is important.
- Quick and easier to compute according to the requirements.
- An approach to successfully resolve collisions in hash tables is essential for generating an index for a key whose hash index corresponds to an existing spot.

Hash Collision?

A hash collision happens when the same hash value is produced for two different input values by a hash algorithm. But it's important to point out that collisions aren't a problem; they're a fundamental aspect of hashing algorithms.

Collisions occur because different hashing techniques in data structure convert every input into a fixed-length code, regardless of its length. Since there are an endless number of inputs and a limited number of outputs, the hashing algorithms will eventually produce repeating hashes.

Types of Hashing in Data Structure

There are two primary hashing techniques in a data structure.

- Open hashing / separate chaining / closed addressing
- Closed hashing / open addressing

Closed hashing / open addressing

The main concept of Open Addressing hashing is to keep all the data in the same hash table and hence a bigger Hash Table is needed. When using open addressing, a collision is resolved by probing (searching) alternative cells in the hash table until our target cell (empty cell while insertion, and cell with value x while searching x) is found.

There are three main Method Resolution Strategies --

a. Linear Probing
b. Quadratic Probing
c. Double Hashing

a)Linear Probing:

- The simplest approach to resolve a collision is linear probing. In this technique, if a value is already stored at a location generated by h(k), it means collision occurred then we do a sequential search to find the empty location.
- Here the idea is to place a value in the next available position. Because in this approach searches are performed sequentially so it’s known as linear probing.
- Here array or hash table is considered circular because when the last slot reached an empty location not found then the search proceeds to the first location of the array.

Clustering is a major drawback of linear probing.

Below is a hash function that calculates the next location. If the location is empty then store value otherwise find the next location.

Following hash function is used to resolve the collision in:

h(k, i) = [h(k) + i] mod m

Where

m = size of the hash table,

h(k) = (k mod m),

i = the probe number that varies from **0 to m–1**.

Therefore, for a given key k, the first location is generated by **[h(k) + 0] mod m**, the first time i=0.

- If the location is free, the value is stored at this location. If value successfully **stores** then probe count is 1 means location is founded on the first go.
- If location is not free then second probe generates the address of the location given by **[h(k) + 1]mod m**.

Similarly, if the generated location is occupied, then subsequent probes generate the address as **[h(k) + 2]mod m**, **[h(k) + 3]mod m**, **[h(k) + 4]mod m**, **[h(k) + 5]mod m**, and so on, until a free location is found.

Note:Probes is a count to find the free location for each value to store in the hash table.

Example:

Insert keys 12,23,30,46 using linear probing with size 7

1.Insert 12 using hash function h(k) = k % 7. Since h(12) = 12%7=5, we insert 12 at index 5

Data						12	
index	0	1	2	3	4	5	6

2.Insert 23. Since h(23) = 23%7=2, we insert 23 at index 2

Data			23			12	
index	0	1	2	3	4	5	6

3.Insert 30. Since h(30) = 30%7=2, there is a collision with index 2. We search for the next available slot and find that index by using h(k)=(h(k)+1)%7.

i.e (h(30)+1)%7= 3, 3 is available, so we insert 30 at index 3:

Data			23	30		12	
index	0	1	2	3	4	5	6

4.Insert 46. Since h(46) = 46%7=4, we insert 45 at index 4

Data			23	30	46	12	
index	0	1	2	3	4	5	6

b) Quadratic Probing

In this technique, if a value is already stored at a location generated by h(k), then the following hash function is used to resolve the collision:

h(k, i) = (h(k) + i^2) mod m

where **m** is the size of the hash table,

h(k) = (k mod m), i is the probe number that varies from **0 to m–1**,

- Quadratic probing solves the clustering problem which is in linear probing because instead of doing a linear search, it does a quadratic search.
- For a given key k, first, the location generated by [h(k) + 0] mod m, where i is 0. If the location is free, the value is stored at this generated location, else new locations will be generated using hash function [h(k) + 1^2] mod m.
- Value of i will change until free space is founded and probe count is increased until free space is founded.

Quadratic probing performs better than linear probing, in order to maximize the utilization of the hash table.

- The disadvantage of quadratic probing is it does not search all locations of the list.

Example:

Insert keys 12,23,30,44 using Quadratic probing with size 7

1.Insert 12 using hash function h(k) = k % 7. Since h(12) = 12%7=5, we insert 12 at index 5

Data						12	
index	0	1	2	3	4	5	6

2.Insert 23. Since h(23) = 23%7=2, we insert 23 at index 2

Data			23			12	
index	0	1	2	3	4	5	6

3.Insert 30. Since h(30) = 30%7=2, there is a collision with index 2. We then use quadratic probing to find the next available slot. We start by checking the next index by using h(k,1)=(h(k)+1^2)%7, which is (2 + 1^2)%7 = 3. insert 12 at index 3

Data			23	30		12	
index	0	1	2	3	4	5	6

4.Insert 44. h(44) = 44%7=2, there is a collision with index 2. We then use quadratic probing to find the next available slot. We start by checking the next index by using h(k,1)=(h(k)+1^2)%7, which is (2 + 1^2)%7 = 3. Index 3 also occupied by 30 so find next available slot, which is h(k,2)=(h(k)+2^2)%7=(2+4)%7=6%7=6. Index 6 is available so insert 44 at index 6

Data			23	30		12	44
index	0	1	2	3	4	5	6

c) Double Hashing

- Double hashing is a collision resolution technique used in conjunction with open-addressing in hash tables.
- In this technique, we use a two hash function to calculate empty slot to store value.
- In the case of collision we take the second hash function h2(k) and look for i * h2(k) free slot in an ith iteration.
- Double hashing requires more computational time becuase two hash functions need to be computed.

To start with, double hashing uses two hash function to calculate an empty location.
In double hashing, we use two hash functions rather than a single
function. The hash function in the case of double hashing can be given as:
h(k, i) = [h1(k) + i*h2(k)] mod m
where m is the size of the hash table,
h1(k) and **h2(k)** are two hash functions
h1(k) = k mod m,
h2(k)= 1 + (k % p) is the probe number that varies from **0 to p–1(**p is nearest prime number which is less than the size)
When we have to insert a key k in the hash table, we first probe the location given by applying [h1(k) mod m] because during the first probe, i = 0. If the location is vacant, the key is inserted into it.

Insert keys 12,23,30,44 using double hashing with size 7

1.Insert 12 using hash function h(k) = k % 7. Since h(12) = 12%7=5, we insert 12 at index 5

Data						12	
index	0	1	2	3	4	5	6

2.Insert 23. Since h(23) = 23%7=2, we insert 23 at index 2

Data			23			12	
index	0	1	2	3	4	5	6

3.Insert 30. Since h(30) = 30%7=2, there is a collision with index 2. We then use Double hasing to find the next available slot. We start by checking the next index by using h(k,1)= **[h1(k) + 1*h2(k)] mod m**

h1(30)=2

h2(30)=1+(30%5)=1+0=1

next available slot is

h(k,1)=(2+1)%7=3, insert 30 at index 3.

Data			23	30		12	
index	0	1	2	3	4	5	6

4. Insert 43. Since h(43) = 43%7=1, we insert 43 at index 1

Data		43	23	30		12	
index	0	1	2	3	4	5	6

Comparison of Quadratic Probing & Double Hashing

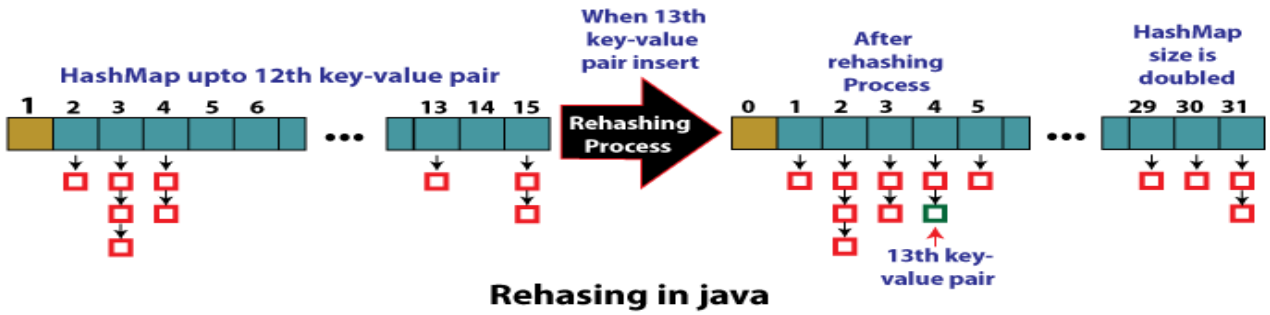
The double hashing requires another hash function whose probing efficiency is same as some another hash function required when handling random collision. The double hashing is more complex to implement than quadratic probing. The quadratic probing is fast technique than double hashing.

REHASHING

Rehashing is a technique in which the table is resized, i.e., the size of table is doubled by creating a new table. It is preferable is the total size of table is a prime number. There are situations in which the rehashing is required.

- When table is completely full
- With quadratic probing when the table is filled half.
- When insertions fail due to overflow.

In such situations, we have to transfer entries from old table to the new table by re computing their positions using hash functions.



Rehasing in java

Consider we have to insert the elements 37, 90, 55, 22, 17, 49, and 87. the table size is 10 and will use hash function.,

H(key) = key mod tablesize
37 % 10 = 7
90 % 10= 0
55 % 10 = 5
22 % 10 = 2
17 % 10 = 7 Collision solved by linear probing
49 % 10 = 9
Now this table is almost full and if we try to insert more elements collisions will occur and eventually further insertions will fail. Hence we will rehash by doubling the table size. The old table size is 10 then we should double this size for new table, that becomes 20. But 20 is not a prime number, we will prefer to make the table size as 23. And new hash function will be
H(key) key mod 23

37%23=14
90%23=21
55%23=9
22%23=22
17%23=17
49%23=3
87%23=18

index	Key	Index	key
0		12	
1		13	
2		14	37
3	49	15	
4		16	
5		17	17
6		18	87
7		19	
8		20	
9	55	21	90
10		22	22
11		23	

Now the hash table is sufficiently large to accommodate new insertions.
Advantages:
1. This technique provides the programmer a flexibility to enlarge the table size if required.
2. Only the space gets doubled with simple hash function which avoids occurrence of collisions.

2). Closed Addressing
In closed addressing, all the keys are stored inside and outside the hash table. Each slot of the hashtable is linked with linked list So if a collision occurs key stores in the linked list.
Separate Chaining

- separate Chaining is a Collision Resolution technique in hash tables.
- In hash tables collision occurs when two keys are hashed to the same index in a hash table.
- It means the calculated hash value for the two keys is the same. Collisions are a problem because every slot in a hash table is supposed to store a single element.

- In the chaining approach, the hash table is an array of linked lists. This means each index of the hash table has its own linked list.
- And if Collision arises then a new value will be store in the linked list of that index. And at index, this linked list appears like a chain that is why this technique is known as the chaining technique.

Time complexity of Chaining

For Searching

In worst case all the value is present in the linked list of the same index. So in this case sequential search is performed to search the value.
So in the worst case, time complexity will be O(n) for searching.

Example: Suppose we have a hash table of size 5 and we want to insert the following keys: 7, 8, 23, 35, 14, 6, 21.
We will use the hash function h(x) = x % 5.

1)Insert 7:
h(7) = 7%5=2, so we insert 7 in the linked list at index 2.

Index	data
0	
1	
2	7
3	
4	

2. Insert 8:
h(8) = 8%5=3, so we insert 8 in the linked list at index 3.

Index	data
0	
1	
2	7
3	8
4	

3. Insert 23:
h(23) = 23%5=3, but index 3 is already occupied. We insert 23 in the same linked list at index 3.

Index	data		
0			
1			
2	7	→	23
3	8		
4			

4.Insert 35:
h(35) = 35%5=0, so we insert 35 in the linked list at index 0.

Index	data
0	35
1	
2	7 → 23
3	8
4	

5.Insert 14:h(14) =14%5= 4, so we insert 14 in the linked list at index 4.

Index	data
0	35
1	
2	7 → 23
3	8
4	14

6.Insert 6: h(6) = 6%5=1, so we insert 6 in the linked list at index 1.

Index	data
0	35
1	6
2	7 → 23
3	8
4	14

7.Insert 21:
h(21) = 21%5=1, but index 1 is already occupied. We insert 21 in the same linked list at index 1.

Index	data
0	35
1	6 → 21
2	7 → 23
3	8
4	14

Now, suppose we want to search for key 21. We first compute its hash value h(21) = 1 and look in the linked list at index 1. We find the key 21 in the linked list, so the search is successful.

Similarly, suppose we want to delete key 8. We compute its hash value h(8) = 3 and look in the linked list at index 3. We find the key 8 in the linked list, so we delete it by updating the pointers in the linked list.

Separate chaining provides a flexible way to handle collisions and can accommodate a large number of keys even if the hash table size is relatively small. However, it can result in increased memory usage due to the overhead of maintaining linked lists.

Binary Search Trees

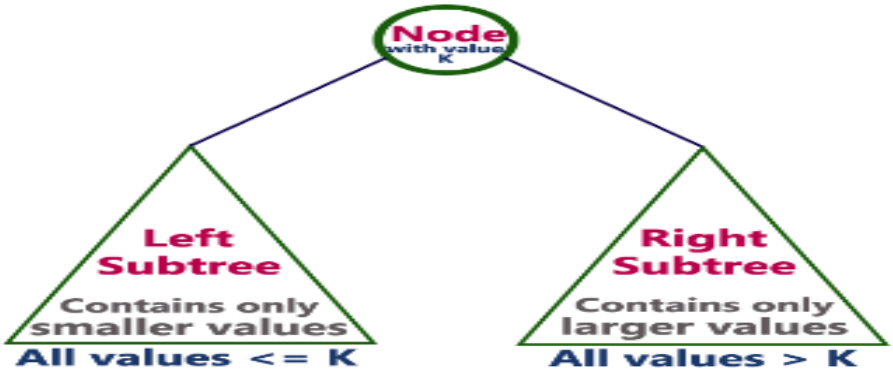
In a binary tree, every node can have a maximum of two children but there is no need to maintain the order of nodes basing on their values. In a binary tree, the elements are arranged in the order they arrive at the tree from top to bottom and left to right.

A binary tree has the following time complexities...

- Search Operation - O(n)
- Insertion Operation - O(1)
- Deletion Operation - O(n)

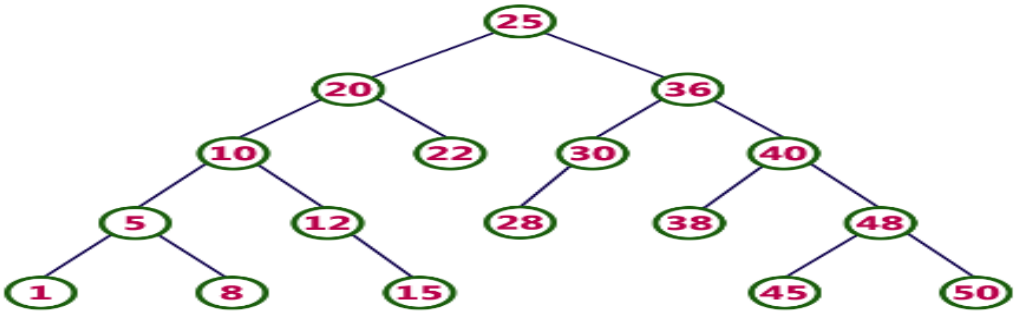
To enhance the performance of binary tree, we use a special type of binary tree known as Binary Search Tree. Binary search tree mainly focuses on the search operation in a binary tree. Binary search tree can be defined as follows.

Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.



Example

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.



Note: Every binary search tree is a binary tree but every binary tree need not to be binary search tree.

Operations on a Binary Search Tree

The following operations are performed on a binary search tree...

- Search
- Insertion
- Deletion
- Traversals

Search Operation in BST

In a binary search tree, the search operation is performed with O(log n) time complexity. The search operation is performed as follows...

Step 1 - Read the search element from the user.
Step 2 - Compare the search element with the value of root node in the tree.
Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function
Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.
Step 5 - If search element is smaller, then continue the search process in left subtree.
Step 6- If search element is larger, then continue the search process in right subtree.
Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node
Step 8 - If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.
Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function

Insertion Operation in BST

In a binary search tree, the insertion operation is performed with O(log n) time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

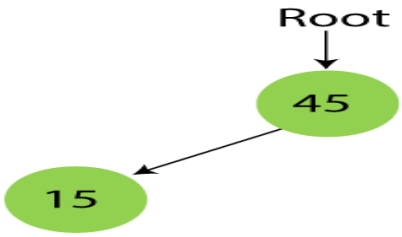
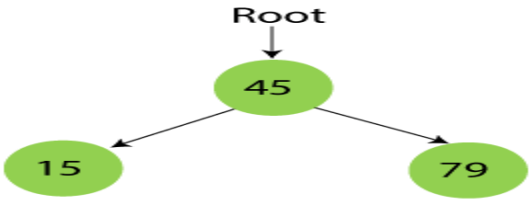
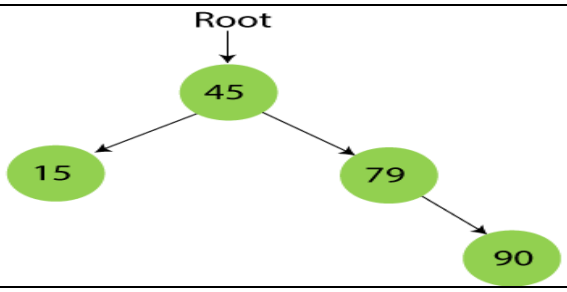
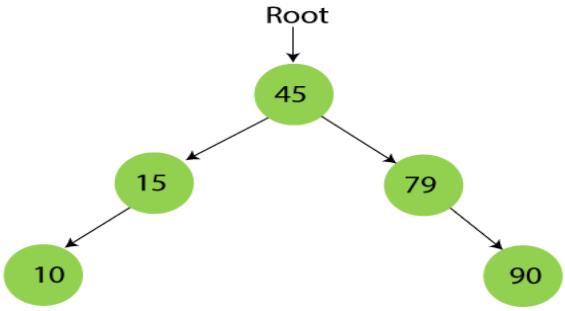
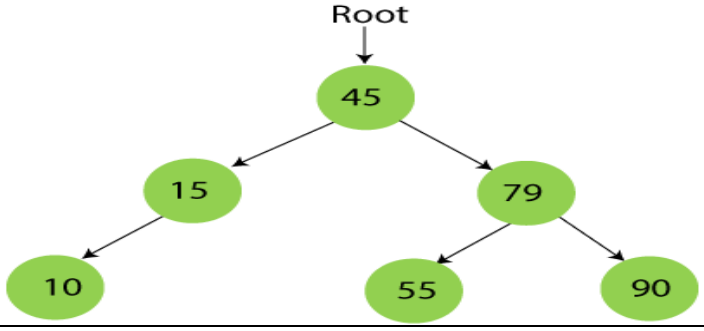
Step 1 - Create a newNode with given value and set its left and right to NULL.
Step 2 - Check whether tree is Empty.
Step 3 - If the tree is Empty, then set root to newNode.
Step 4 - If the tree is Not Empty, then check whether the value of newNode is smaller or larger than the node (here it is root node).
Step 5 - If newNode is smaller than or equal to the node then move to its left child. If newNode is larger than the node then move to its right child.
Step 6- Repeat the above steps until we reach to the leaf node (i.e., reaches to NULL).
Step 7 - After reaching the leaf node, insert the newNode as left child if the newNode is smaller or equal to that leaf node or else insert it as right child.

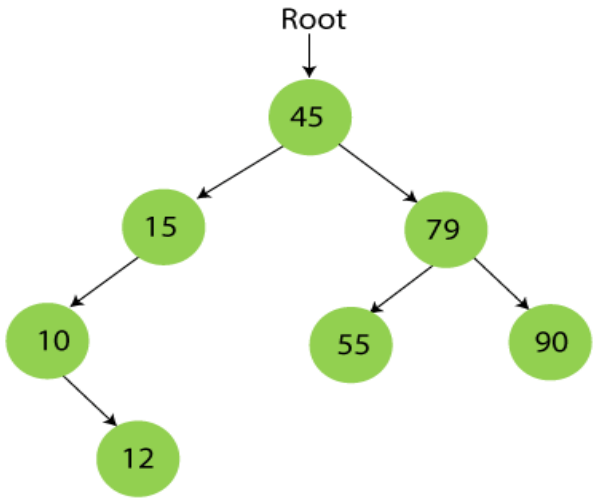
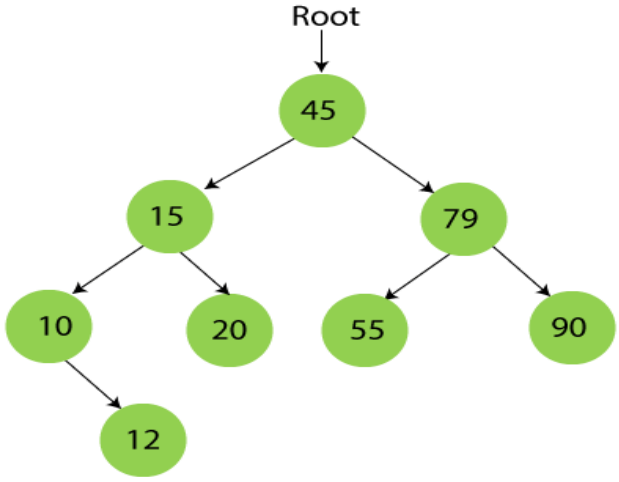
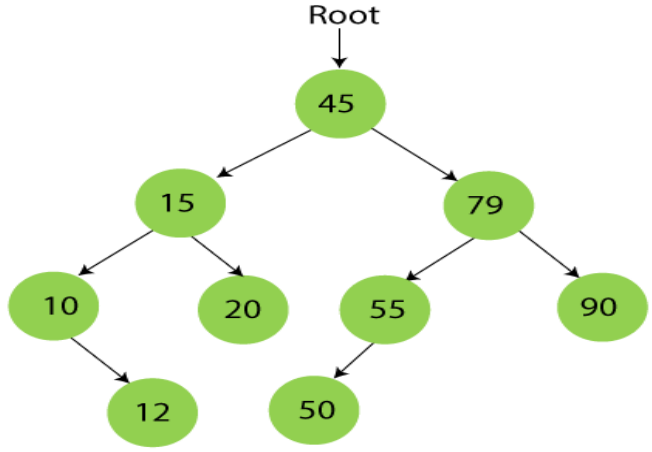
Example of creating a binary search tree

Suppose the data elements are - 45, 15, 79, 90, 10, 55, 12, 20, 50

- First, we have to insert 45 into the tree as the root of the tree.
- Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
- Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

Now, let's see the process of creating the Binary search tree using the given data element. The process of creating the BST is shown below –

Step 1 - Insert 45.	
Step 2 - Insert 15. As 15 is smaller than 45, so insert it as the root node of the left subtree.	
Step 3 - Insert 79. As 79 is greater than 45, so insert it as the root node of the right subtree.	
Step 4 - Insert 90. 90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.	
Step 5 - Insert 10. 10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.	
Step 6 - Insert 55. 55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.	

Step 7 - Insert 12. 12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.	
Step 8 - Insert 20. 20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.	
Step 9 - Insert 50. 50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.	

Deletion in Binary Search tree

In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated. To delete a node from BST, there are three possible situations occur -

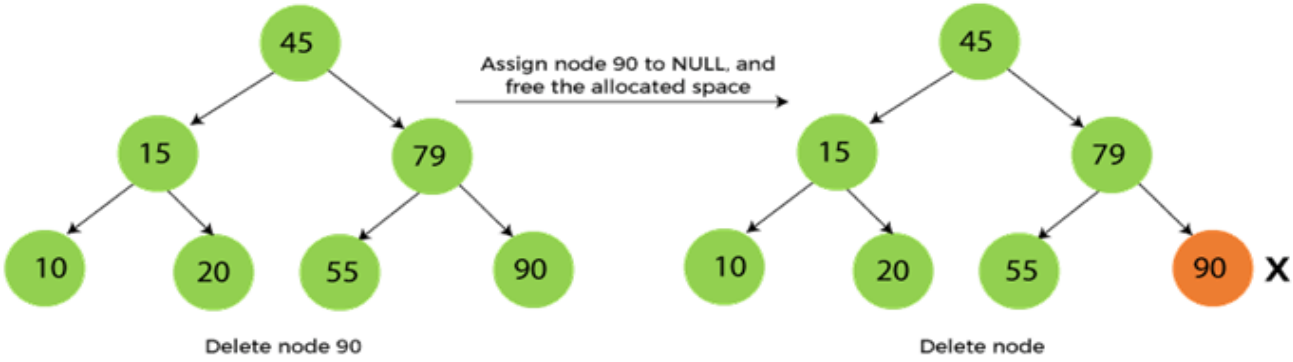
- The node to be deleted is the leaf node, or,
- The node to be deleted has only one child, and,
- The node to be deleted has two children

We will understand the situations listed above in detail.

When the node to be deleted is the leaf node

It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.

We can see the process to delete a leaf node from BST in the below image. In below image, suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.

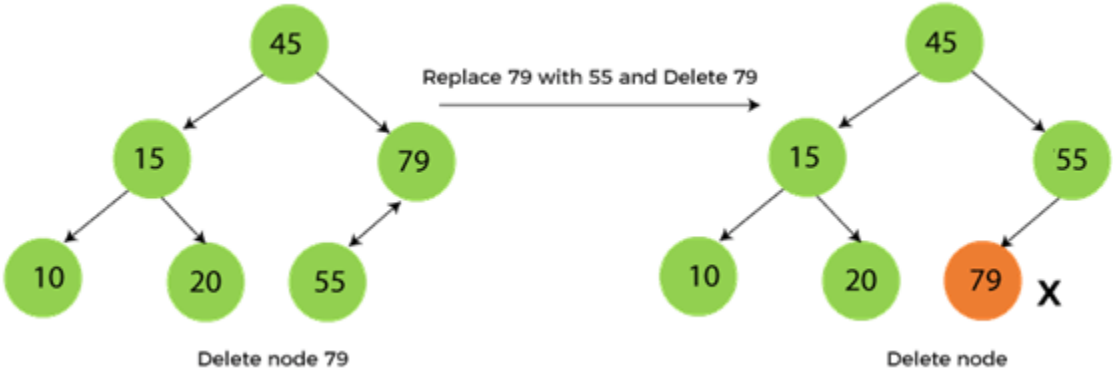


When the node to be deleted has only one child

In this case, we have to replace the target node with its child, and then delete the child node. It means that after replacing the target node with its child node, the child node will now contain the value to be deleted. So, we simply have to replace the child node with NULL and free up the allocated space.

We can see the process of deleting a node with one child from BST in the below image. In the below image, suppose we have to delete the node 79, as the node to be deleted has only one child, so it will be replaced with its child 55.

So, the replaced node 79 will now be a leaf node that can be easily deleted.



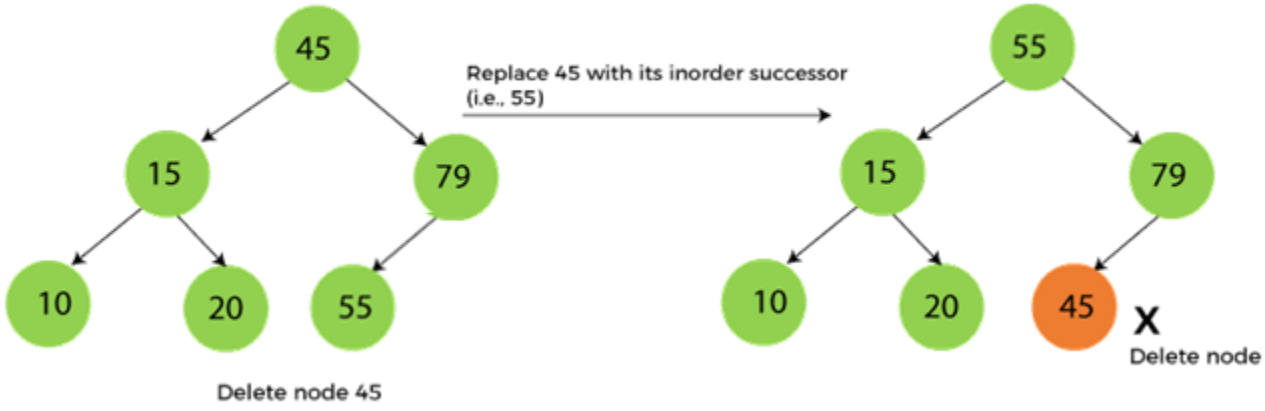
When the node to be deleted has two children

This case of deleting a node in BST is a bit complex among other two cases. In such a case, the steps to be followed are listed as follows -

- First, find the inorder successor of the node to be deleted.
- After that, replace that node with the inorder successor until the target node is placed at the leaf of tree.
- And at last, replace the node with NULL and free up the allocated space.

The inorder successor is required when the right child of the node is not empty. We can obtain the inorder successor by finding the minimum element in the right child of the node.

We can see the process of deleting a node with two children from BST in the below image. In the below image, suppose we have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder successor. Now, node 45 will be at the leaf of the tree so that it can be deleted easily.



Traversals

The term 'tree traversal' means traversing or visiting each node of a tree. There is a single way to traverse the linear data structure such as linked list, queue, and stack. Whereas, there are multiple ways to traverse a tree that are listed as follows -

- Preorder traversal
- Inorder traversal
- Postorder traversal

Recursive Preorder traversal

- This technique follows the **'root- left- right'** policy. It means that, first root node is visited after that the left subtree is traversed recursively, and finally, right subtree is recursively traversed. As the root node is traversed before (or pre) the left and right subtree, it is called preorder traversal.
- So, in a preorder traversal, each node is visited before both of its subtrees.

Algorithm

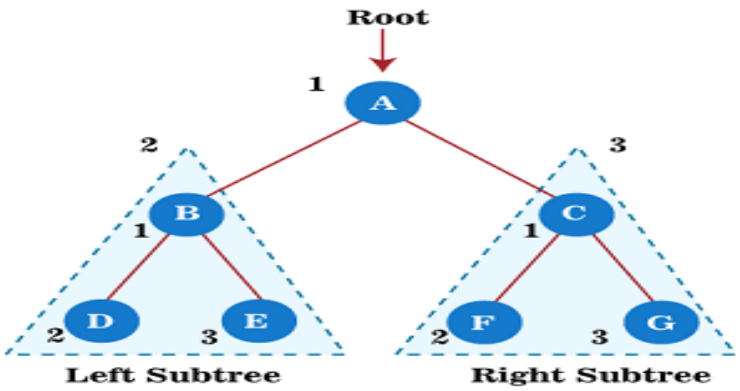
Until all nodes of the tree are not visited

Step 1 - Visit the root node

Step 2 - Traverse the left subtree recursively.

Step 3 - Traverse the right subtree recursively.

Now, let's see the example of the preorder traversal technique.



Now, start applying the preorder traversal on the above tree. First, we traverse the root node **A;** after that, move to its left subtree **B**, which will also be traversed in preorder.

So, for left subtree B, first, the root node **B** is traversed itself; after that, its left subtree **D** is traversed. Since node **D** does not have any children, move to right subtree **E**. As node E also does not have any children, the traversal of the left subtree of root node A is completed.

Now, move towards the right subtree of root node A that is C. So, for right subtree C, first the root node **C** has traversed itself; after that, its left subtree **F** is traversed. Since node **F** does not have any children, move to the right subtree **G**. As node G also does not have any children, traversal of the right subtree of root node A is completed.

Therefore, all the nodes of the tree are traversed. So, the output of the preorder traversal of the above tree is - **A → B → D → E → C → F → G**

Recursive Postorder traversal

This technique follows the 'left-right root' policy. It means that the first left subtree of the root node is traversed, after that recursively traverses the right subtree, and finally, the root node is traversed. As the root node is traversed after (or post) the left and right subtree, it is called postorder traversal.

So, in a postorder traversal, each node is visited after both of its subtrees.

Algorithm

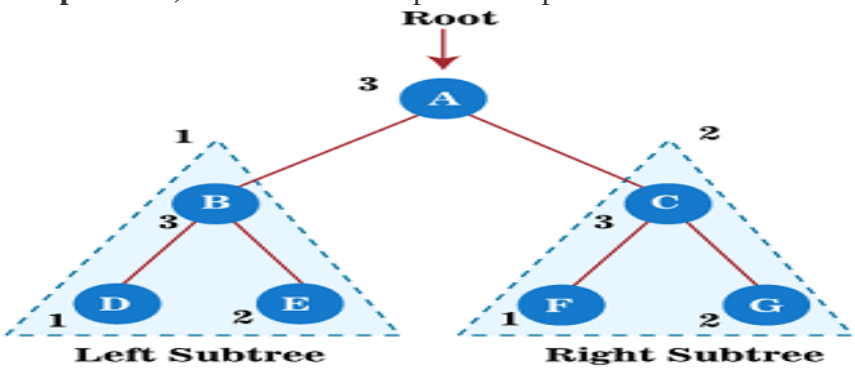
Until all nodes of the tree are not visited

Step 1 - Traverse the left subtree recursively.

Step 2 - Traverse the right subtree recursively.

Step 3 - Visit the root node.

Example: Now, let's see the example of the postorder traversal technique.



- Now, start applying the postorder traversal on the above tree. First, we traverse the left subtree B that will be traversed in postorder. After that, we will traverse the right subtree **C** in postorder. And finally, the root node of the above tree, i.e., **A**, is traversed.
- So, for left subtree B, first, its left subtree **D** is traversed. Since node **D** does not have any children, traverse the right subtree **E**. As node E also does not have any children, move to the root node **B.** After traversing node **B,** the traversal of the left subtree of root node A is completed.
- Now, move towards the right subtree of root node A that is C. So, for right subtree C, first its left subtree **F** is traversed. Since node **F** does not have any children, traverse the right subtree **G**. As node G also does not have any children, therefore, finally, the root node of the right subtree, i.e., **C,** is traversed. The traversal of the right subtree of root node A is completed.
- At last, traverse the root node of a given tree, i.e., **A**. After traversing the root node, the postorder traversal of the given tree is completed.
- Therefore, all the nodes of the tree are traversed. So, the output of the postorder traversal of the above tree is **-D → E → B → F → G → C → A**

Recursive Inorder Traversal

This technique follows the 'left root right' policy. It means that first left subtree is visited after that root node is traversed, and finally, the right subtree is traversed. As the root node is traversed between the left and right subtree, it is named inorder traversal.

So, in the inorder traversal, each node is visited in between of its subtrees.

Algorithm

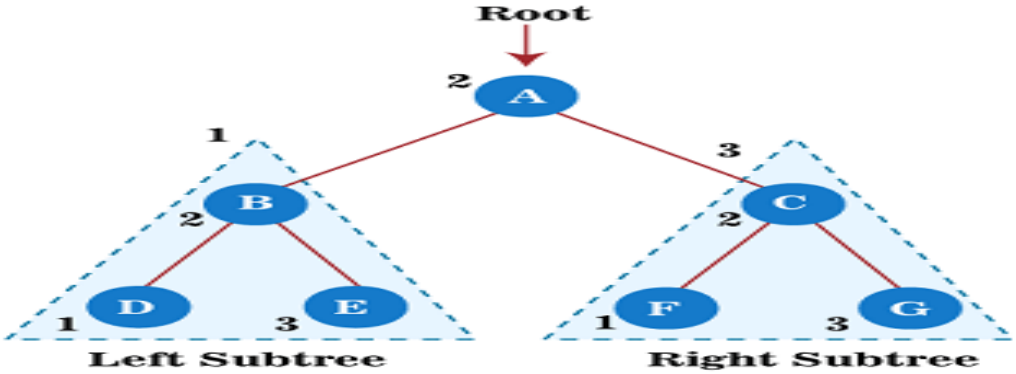
Until all nodes of the tree are not visited

Step 1 - Traverse the left subtree recursively.

Step 2 - Visit the root node.

Step 3 - Traverse the right subtree recursively.

Example: Now, let's see the example of the Inorder traversal technique.



- Now, start applying the inorder traversal on the above tree. First, we traverse the left subtree **B** that will be traversed in inorder. After that, we will traverse the root node **A**. And finally, the right subtree **C** is traversed in inorder.
- So, for left subtree **B**, first, its left subtree **D** is traversed. Since node **D** does not have any children, so after traversing it, node **B** will be traversed, and at last, right subtree of node B, that is **E**, is traversed. Node E also does not have any children; therefore, the traversal of the left subtree of root node A is completed.
- After that, traverse the root node of a given tree, i.e., **A**.
- At last, move towards the right subtree of root node A that is C. So, for right subtree C; first, its left subtree **F** is traversed. Since node **F** does not have any children, node **C** will be traversed, and at last, a right subtree of node C, that is, **G**, is traversed. Node G also does not have any children; therefore, the traversal of the right subtree of root node A is completed.
- As all the nodes of the tree are traversed, the inorder traversal of the given tree is completed. The output of the inorder traversal of the above tree is –

  D → B → E → A → F → C → G

Non-recursive Inorder Traversal of a Binary Search Tree

The non-recursive inorder traversal of a binary search tree can be performed using a stack. Here are the steps:

1. Create an empty stack to store the tree nodes.
2. Initialize a pointer variable to the root of the binary search tree.
3. Repeat the following steps until the stack is empty or the pointer is null:
    - While the pointer is not null, push the pointer onto the stack and move the pointer to its left child.
    - If the pointer is null and the stack is not empty, pop a node from the stack, process its value (print it, store it in an array, or perform any desired operation), and set the pointer to the popped node's right child.
    - Repeat the above steps until all nodes have been processed.
4. Finish when the stack is empty and all nodes have been visited.

In summary, the non-recursive inorder traversal uses a stack to simulate the recursive function call stack. It iteratively traverses the left subtree, processes the current node, and then moves to the right subtree.

This approach guarantees that the nodes will be processed in ascending order because it visits the left subtree before the current node and the right subtree after the current node.

Logic:

```
class InorderTraversal {
    public static void inorderTraversal(TreeNode root) {
        if (root == null) {
            return;
        }
        Stack<TreeNode> stack = new Stack<>();
        TreeNode current = root;
        while (!stack.isEmpty() || current != null) {
            while (current != null) {
                stack.push(current);
                current = current.left;
            }
            current = stack.pop();
            System.out.print(current.val + " ");  // Process the node
            current = current.right;
        }
    }
```

Non-recursive Postorder Traversal of a Binary Search Tree

To perform a non-recursive postorder traversal of a binary search tree, you can use a stack. Here are the steps:

1. Create an empty stack.
2. Initialize a current node pointer variable to the root of the binary search tree.
3. Repeat the following steps until the stack is empty or the current node is null:
    - While the current node is not null, push it onto the stack and set the current node to its left child.
    - If the current node is null, peek the top element from the stack (without removing it) and store it in a variable (let's call it "top").
        - If the right child of the top node exists and is not equal to the previously visited node (let's call it "prev"), set the current node to the right child of the top node.
        - If the right child of the top node is null or equal to the previously visited node, it means we have visited the left and right subtrees of the top node, so process the top node's value (print it, store it in an array, or perform any desired operation) and set "prev" to the top node.
        - Pop the top node from the stack.
4. Finish when the stack is empty and all nodes have been visited.

The idea behind this approach is to simulate the recursive postorder traversal by iteratively visiting the left subtree, then the right subtree, and finally the current node. We use the stack to keep track of the nodes and ensure the correct order of traversal.

Logic

```
class PostorderTraversal {
    public static void postorderTraversal(TreeNode root) {
        if (root == null) {
            return;
        }
        Stack<TreeNode> stack = new Stack<>();
        TreeNode current = root;
        TreeNode prev = null;
        while (current != null || !stack.isEmpty()) {
            while (current != null) {
                stack.push(current);
                current = current.left;
            }
            TreeNode top = stack.peek();
            if (top.right != null && top.right != prev) {
                current = top.right;
            } else {
                System.out.print(top.val + " ");  // Process the node
                prev = top;
                stack.pop();
            }
        }}
```

Non-recursive Preorder Traversal of a Binary Search Tree

To perform a non-recursive preorder traversal of a binary search tree, you can use a stack. Here are the steps:

1. Create an empty stack.
2. Push the root node onto the stack.
3. Repeat the following steps until the stack is empty:
    - Pop a node from the stack and process its value (print it, store it in an array, or perform any desired operation).
    - Push the right child of the popped node onto the stack (if it exists).
    - Push the left child of the popped node onto the stack (if it exists).
4. Finish when the stack is empty and all nodes have been visited.

The idea behind this approach is to visit the current node, then push its right child onto the stack (to be visited later), and finally push its left child onto the stack (to be visited next). This ensures that the left subtree is visited before the right subtree, as per the preorder traversal order.

Logic:

```
class PreorderTraversal {
    public static void preorderTraversal(TreeNode root) {
        if (root == null) {
            return;
        }
        Stack<TreeNode> stack = new Stack<>();
        stack.push(root);
        while (!stack.isEmpty()) {
            TreeNode current = stack.pop();
            System.out.print(current.val + " ");  // Process the node
            if (current.right != null) {
                stack.push(current.right);
            }

            if (current.left != null) {
                stack.push(current.left);
            }
        }
    }
```