Sirf khwabh Nhi

27/07

1] Concept 1

Java Abstract Class and Abstract Methods

Java abstract classes and methods with the help of examples. We will also learn about abstraction in Java.

Java Abstract Class

The abstract class in Java cannot be instantiated (we cannot create objects of abstract classes). We use the `abstract` keyword to declare an abstract class. For example,

```
// create an abstract class
abstract class Language {
  // fields and methods
}
...

// try to create an object Language
// throws an error
Language obj = new Language();
```

An abstract class can have both the regular methods and abstract methods. For example,

```
abstract class Language {

  // abstract method
  abstract void method1();
```

```
  // regular method
  void method2() {
    System.out.println("This is regular method");
  }
}
```

Java Abstract Method

A method that doesn't have its body is known as an abstract method. We use the same `abstract` keyword to create abstract methods. For example,

```
abstract void display();
```

Here, `display()` is an abstract method. The body of `display()` is replaced by `;`.

If a class contains an abstract method, then the class should be declared abstract. Otherwise, it will generate an error. For example,

```
// error
// class should be abstract
class Language {

  // abstract method
  abstract void method1();
}
```

Example: Java Abstract Class and Method

Though abstract classes cannot be instantiated, we can create subclasses from it. We can then access members of the abstract class using the object of the subclass. For example,

```
abstract class Language {


  // method of abstract class
  public void display() {
    System.out.println("This is Java Programming");
  }
}


class Main extends Language {


  public static void main(String[] args) {


    // create an object of Main
    Main obj = new Main();


    // access method of abstract class
    // using object of Main class
    obj.display();
  }
}
```

Output

```
This is Java programming
```

In the above example, we have created an abstract class named `Language`. The class contains a regular method `display()`.

We have created the Main class that inherits the abstract class. Notice the statement,

```
obj.display();
```

Here, `obj` is the object of the child class `Main`. We are calling the method of the abstract class using the object `obj`.

Implementing Abstract Methods

If the abstract class includes any abstract method, then all the child classes inherited from the abstract superclass must provide the implementation of the abstract method. For example,

```
abstract class Animal {

  abstract void makeSound();


  public void eat() {
    System.out.println("I can eat.");
  }
}


class Dog extends Animal {


  // provide implementation of abstract method
  public void makeSound() {
    System.out.println("Bark bark");
  }
```

```
}


class Main {
  public static void main(String[] args) {


    // create an object of Dog class
    Dog d1 = new Dog();


    d1.makeSound();
    d1.eat();
  }
}
```

Output

```
Bark bark
I can eat.
```

In the above example, we have created an abstract class `Animal`. The class contains an abstract method `makeSound()` and a non-abstract method `eat()`.

We have inherited a subclass `Dog` from the superclass `Animal`. Here, the subclass `Dog` provides the implementation for the abstract method `makeSound()`.

We then used the object `d1` of the `Dog` class to call methods `makeSound()` and `eat()`.

> Note: If the `Dog` class doesn't provide the implementation of the abstract method `makeSound()`, `Dog` should also be declared as abstract. This is because the subclass `Dog` inherits `makeSound()` from `Animal`.

Accesses Constructor of Abstract Classes

An abstract class can have constructors like the regular class. And, we can access the constructor of an abstract class from the subclass using the `super` keyword. For example,

```
abstract class Animal {

    Animal() {

        ….

    }

}



class Dog extends Animal {

    Dog() {

        super();

        ...

    }

}
```

Here, we have used the `super()` inside the constructor of `Dog` to access the constructor of the `Animal`.

Note that the `super` should always be the first statement of the subclass constructor.

Java Abstraction

The major use of abstract classes and methods is to achieve abstraction in Java.

Abstraction is an important concept of object-oriented programming that allows us to hide unnecessary details and only show the needed information.

This allows us to manage complexity by omitting or hiding details with a simpler, higher-level idea.

A practical example of abstraction can be motorbike brakes. We know what brake does. When we apply the brake, the motorbike will stop. However, the working of the brake is kept hidden from us.

The major advantage of hiding the working of the brake is that now the manufacturer can implement brake differently for different motorbikes, however, what brake does will be the same.

Let's take an example that helps us to better understand Java abstraction.

Example 3: Java Abstraction

```
abstract class Animal {

  abstract void makeSound();



}



class Dog extends Animal {



  // implementation of abstract method

  public void makeSound() {

    System.out.println("Bark bark.");

  }

}



class Cat extends Animal {



  // implementation of abstract method

  public void makeSound() {

    System.out.println("Meows ");
```

```
  }
}


class Main {
  public static void main(String[] args) {
    Dog d1 = new Dog();
    d1.makeSound();


    Cat c1 = new Cat();
    c1.makeSound();
  }
}
```

Output:

```
Bark bark
Meows
```

In the above example, we have created a superclass `Animal`. The superclass `Animal` has an abstract method `makeSound()`.

The `makeSound()` method cannot be implemented inside `Animal`. It is because every animal makes different sounds. So, all the subclasses of `Animal` would have different implementation of `makeSound()`.

So, the implementation of `makeSound()` in `Animal` is kept hidden.

Here, `Dog` makes its own implementation of `makeSound()` and `Cat` makes its own implementation of `makeSound()`.

> Note: We can also use interfaces to achieve abstraction in Java.

Key Points to Remember

- We use the `abstract` keyword to create abstract classes and methods.
- An abstract method doesn't have any implementation (method body).
- A class containing abstract methods should also be abstract.
- We cannot create objects of an abstract class.
- To implement features of an abstract class, we inherit subclasses from it and create objects of the subclass.
- A subclass must override all abstract methods of an abstract class. However, if the subclass is declared abstract, it's not mandatory to override abstract methods.
- We can access the static attributes and methods of an abstract class using the reference of the abstract class. For example,
  `Animal.staticMethod();`

1] Concept 2

Java Interface

how to implement interfaces and when to use them

An interface is a fully abstract class. It includes a group of abstract methods (methods without a body).

We use the `interface` keyword to create an interface in Java. For example,

```
interface Language {

  public void getType();



  public void getVersion();

}
```

Here,

- `Language` is an interface.
- It includes abstract methods: `getType()` and `getVersion()`.

Implementing an Interface

Like abstract classes, we cannot create objects of interfaces.

To use an interface, other classes must implement it. We use the `implements` keyword to implement an interface.

Example 1: Java Interface

```
interface Polygon {

  void getArea(int length, int breadth);

}


// implement the Polygon interface

class Rectangle implements Polygon {


  // implementation of abstract method

  public void getArea(int length, int breadth) {

    System.out.println("The area of the rectangle is " + (length * breadth));

  }

}


class Main {
```

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

    Rectangle r1 = new Rectangle();

    r1.getArea(5, 6);

  }

}
```

Output

```
The area of the rectangle is 30
```

In the above example, we have created an interface named `Polygon`. The interface contains an abstract method `getArea()`.

Here, the `Rectangle` class implements `Polygon`. And, provides the implementation of the `getArea()` method.

Example 2: Java Interface

```
// create an interface

interface Language {

  void getName(String name);

}


// class implements interface

class ProgrammingLanguage implements Language {


  // implementation of abstract method

  public void getName(String name) {

    System.out.println("Programming Language: " + name);
```

```
  }
}


class Main {
  public static void main(String[] args) {
    ProgrammingLanguage language = new ProgrammingLanguage();
    language.getName("Java");
  }
}
```

Output

```
Programming Language: Java
```

In the above example, we have created an interface named `Language`. The interface includes an abstract method `getName()`.

Here, the `ProgrammingLanguage` class implements the interface and provides the implementation for the method.

Implementing Multiple Interfaces

In Java, a class can also implement multiple interfaces. For example,

```
interface A {
  // members of A
}


interface B {
  // members of B
```

```
}


class C implements A, B {
  // abstract members of A
  // abstract members of B
}
```

Extending an Interface

Similar to classes, interfaces can extend other interfaces. The `extends` keyword is used for extending interfaces. For example,

```
interface Line {
  // members of Line interface
}


// extending interface
interface Polygon extends Line {
  // members of Polygon interface
  // members of Line interface
}
```

Here, the `Polygon` interface extends the `Line` interface. Now, if any class implements `Polygon`, it should provide implementations for all the abstract methods of both `Line` and `Polygon`

.

Extending Multiple Interfaces

An interface can extend multiple interfaces. For example,

```
interface A {
   ...
}
interface B {
   ...
}

interface C extends A, B {
   ...
}
```

Advantages of Interface in Java

Now that we know what interfaces are, let's learn about why interfaces are used in Java.

- Similar to abstract classes, interfaces help us to achieve abstraction in Java.

  Here, we know `getArea()` calculates the area of polygons but the way area is calculated is different for different polygons. Hence, the implementation of `getArea()` is independent of one another.
- Interfaces provide specifications that a class (which implements it) must follow.

  In our previous example, we have used `getArea()` as a specification inside the interface `Polygon`. This is like setting a rule that we should be able to get the area of every polygon.

  Now any class that implements the `Polygon` interface must provide an implementation for the `getArea()` method.

Interfaces are also used to achieve multiple inheritance in Java. For example,

```
interface Line {
…
}

interface Polygon {
…
}

class Rectangle implements Line, Polygon {
…
```

- }

  Here, the class `Rectangle` is implementing two different interfaces. This is how we achieve multiple inheritance in Java.

Note: All the methods inside an interface are implicitly `public` and all fields are implicitly `public static final`. For example,

```
interface Language {

  // by default public static final
  String type = "programming language";
```

```
  // by default public

  void getName();

}
```

---

default methods in Java Interfaces

With the release of Java 8, we can now add methods with implementation inside an interface. These methods are called default methods.

To declare default methods inside interfaces, we use the `default` keyword. For example,

```
public default void getSides() {

   // body of getSides()

}
```

Why default methods?

Let's take a scenario to understand why default methods are introduced in Java.

Suppose, we need to add a new method in an interface.

We can add the method in our interface easily without implementation. However, that's not the end of the story. All our classes that implement that interface must provide an implementation for the method.

If a large number of classes were implementing this interface, we need to track all these classes and make changes to them. This is not only tedious but error-prone as well.

To resolve this, Java introduced default methods. Default methods are inherited like ordinary methods.

Let's take an example to have a better understanding of default methods.

Example: Default Method in Java Interface

```
interface Polygon {

  void getArea();
```

```
  // default method
  default void getSides() {
    System.out.println("I can get sides of a polygon.");
  }
}

// implements the interface
class Rectangle implements Polygon {
  public void getArea() {
    int length = 6;
    int breadth = 5;
    int area = length * breadth;
    System.out.println("The area of the rectangle is " + area);
  }

  // overrides the getSides()
  public void getSides() {
    System.out.println("I have 4 sides.");
  }
}

// implements the interface
```

```
class Square implements Polygon {

  public void getArea() {

    int length = 5;

    int area = length * length;

    System.out.println("The area of the square is " + area);

  }

}



class Main {

  public static void main(String[] args) {



    // create an object of Rectangle

    Rectangle r1 = new Rectangle();

    r1.getArea();

    r1.getSides();



    // create an object of Square

    Square s1 = new Square();

    s1.getArea();

    s1.getSides();

  }

}
```

Output

```
The area of the rectangle is 30

I have 4 sides.

The area of the square is 25

I can get sides of a polygon.
```

In the above example, we have created an interface named `Polygon`. It has a default method `getSides()` and an abstract method `getArea()`.

Here, we have created two classes `Rectangle` and `Square` that implement `Polygon`.

The `Rectangle` class provides the implementation of the `getArea()` method and overrides the `getSides()` method. However, the `Square` class only provides the implementation of the `getArea()` method.

Now, while calling the `getSides()` method using the `Rectangle` object, the overridden method is called. However, in the case of the `Square` object, the default method is called.

---

private and static Methods in Interface

The Java 8 also added another feature to include static methods inside an interface.

Similar to a class, we can access static methods of an interface using its references. For example,

```
// create an interface

interface Polygon {

  staticMethod(){..}

}



// access static method

Polygon.staticMethod();
```

Note: With the release of Java 9, private methods are also supported in interfaces.

We cannot create objects of an interface. Hence, private methods are used as helper methods that provide support to other methods in interfaces.

Practical Example of Interface

Let's see a more practical example of Java Interface.

```
// To use the sqrt function

import java.lang.Math;



interface  Polygon {

   void getArea();



 // calculate the perimeter of a Polygon

   default void getPerimeter(int... sides) {

      int perimeter = 0;

      for (int side: sides) {

         perimeter += side;

      }



   System.out.println("Perimeter: " + perimeter);

   }

}



class Triangle implements Polygon {
```

```
    private int a, b, c;
    private double s, area;

// initializing sides of a triangle
    Triangle(int a, int b, int c) {
        this.a = a;
        this.b = b;
        this.c = c;
        s = 0;
    }

// calculate the area of a triangle
    public void getArea() {
        s = (double) (a + b + c)/2;
        area = Math.sqrt(s*(s-a)*(s-b)*(s-c));
        System.out.println("Area: " + area);
    }
}

class Main {
    public static void main(String[] args) {
        Triangle t1 = new Triangle(2, 3, 4);
```

```
// calls the method of the Triangle class
      t1.getArea();


// calls the method of Polygon
      t1.getPerimeter(2, 3, 4);
   }
}
```

Output

```
Area: 2.9047375096555625
Perimeter: 9
```

In the above program, we have created an interface named `Polygon`. It includes a default method `getPerimeter()` and an abstract method `getArea()`.

We can calculate the perimeter of all polygons in the same manner so we implemented the body of `getPerimeter()` in `Polygon`.

Now, all polygons that implement `Polygon` can use `getPerimeter()` to calculate perimeter.

However, the rule for calculating the area is different for different polygons. Hence, `getArea()` is included without implementation.

Any class that implements `Polygon` must provide an implementation of `getArea()`

3] Concept 3 :Hashing

In hashing there is a hash function that maps keys to some values. But these hashing function may lead to collision that is two or more keys are mapped to same value. **Chain hashing** avoids collision. The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

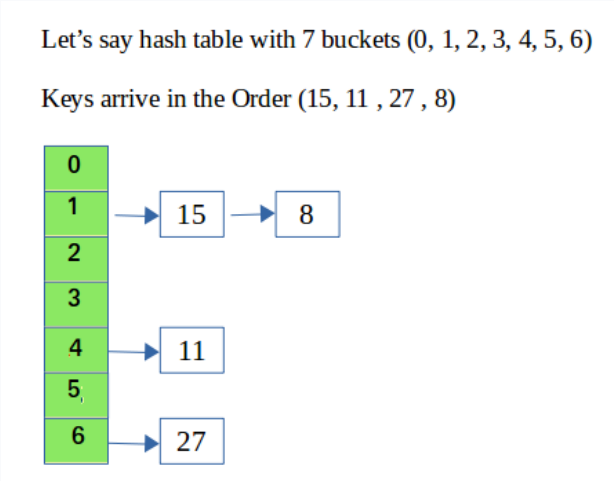
Let's create a hash function, such that our hash table has 'N' number of buckets.

To insert a node into the hash table, we need to find the hash index for the given key. And it could be calculated using the hash function.

Example: hashIndex = key % noOfBuckets

Insert: Move to the bucket corresponds to the above calculated hash index and insert the new node at the end of the list.

Delete: To delete a node from hash table, calculate the hash index for the key, move to the bucket corresponds to the calculated hash index, search the list in the current bucket to find and remove the node with the given key (if found).



Methods to implement Hashing in Java

Hyat khup sarre methods ahet ata sadhya Main doon methods baghto apan , hashmap ani hashtable , in detail udya karnar ahot

1. **With help of Hashtable (A synchronized implementation of hashing)**

```
// Java program to demonstrate working of HashTable

import java.util.*;



class GFG {

    public static void main(String args[])

    {



        // Create a HashTable to store

        // String values corresponding to integer keys

        Hashtable<Integer, String>

            hm = new Hashtable<Integer, String>();



        // Input the values

        hm.put(1, "Geeks");

        hm.put(12, "forGeeks");

        hm.put(15, "A computer");

        hm.put(3, "Portal");



        // Printing the Hashtable

        System.out.println(hm);

    }
```

```
}
```

Output:

```
{15=A computer, 3=Portal, 12=forGeeks, 1=Geeks}
```

2. **With the help of** Hashmap**(A non-synchronized faster implementation of hashing)**

```
// Java program to create HashMap from an array
// by taking the elements as Keys and
// the frequencies as the Values

import java.util.*;

class GFG {

    // Function to create HashMap from array
    static void createHashMap(int arr[])
    {
        // Creates an empty HashMap
        HashMap<Integer, Integer> hmap = new HashMap<Integer, Integer>();

        // Traverse through the given array
        for (int i = 0; i < arr.length; i++) {

            // Get if the element is present
            Integer c = hmap.get(arr[i]);

            // If this is first occurrence of element
```

```
            // Insert the element
            if (hmap.get(arr[i]) == null) {
                hmap.put(arr[i], 1);
            }

            // If elements already exists in hash map
            // Increment the count of element by 1
            else {
                hmap.put(arr[i], ++c);
            }
        }

        // Print HashMap
        System.out.println(hmap);
    }

    // Driver method to test above method
    public static void main(String[] args)
    {
        int arr[] = { 10, 34, 5, 10, 3, 5, 10 };
        createHashMap(arr);
```

```
    }

}
```

Output:

```
{34=1, 3=1, 5=2, 10=3}
```

1] Concept 4 :Hashing madhla collusion ani problems

Hyat pann doon methods ahet collusion handle karayla , Separate Chaining ani Open Addressing methods ahet.

What is Collision?

Since a hash function gets us a small number for a key which is a big integer or string, there is a possibility that two keys result in the same value. The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique.

What are the chances of collisions with large table?

Collisions are very likely even if we have big table to store keys. An important observation is Birthday Paradox With only 23 persons, the probability that two people have the same birthday is 50%.

How to handle Collisions?

There are mainly two methods to handle collision:

1) Separate Chaining

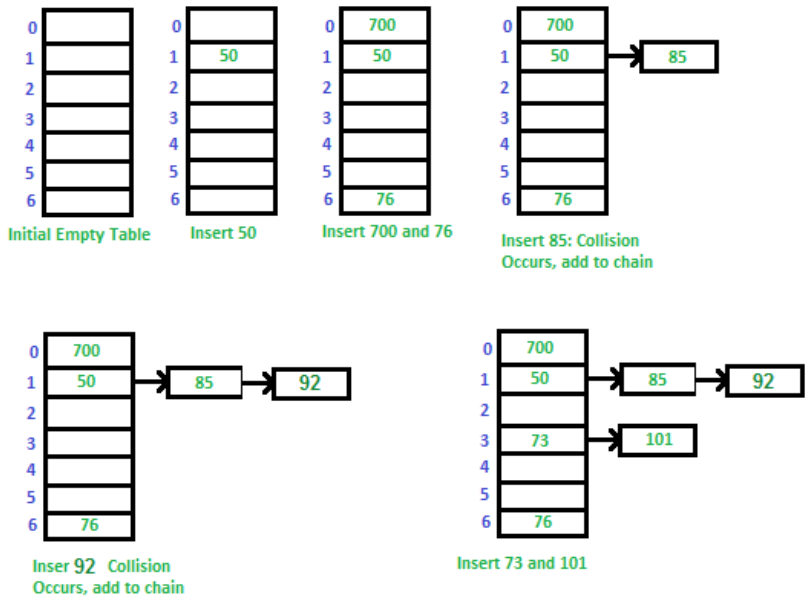
2) Open Addressing

In this article, only separate chaining is discussed. We will be discussing Open addressing in the next post.

Separate Chaining:

The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



Advantages:

1) Simple to implement.

2) Hash table never fills up, we can always add more elements to the chain.

3) Less sensitive to the hash function or load factors.

4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

1) Cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.

2) Wastage of Space (Some Parts of hash table are never used)

3) If the chain becomes long, then search time can become O(n) in the worst case.

4) Uses extra space for links.

Performance of Chaining:

Performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of table (simple uniform hashing).

```
m = Number of slots in hash table

 n = Number of keys to be inserted in hash table



 Load factor α = n/m



 Expected time to search = O(1 + α)



 Expected time to delete = O(1 + α)



Time to insert = O(1)



 Time complexity of search insert and delete is

 O(1) if  α is O(1)
```

Data Structures For Storing Chains:

- Linked lists
    - Search: O(l) where l = length of linked list
    - Delete: O(l)
    - Insert: O(l)

    - Not cache friendly
- Dynamic Sized Arrays (Vectors in C++, ArrayList in Java, list in Python)
    - Search: O(l) where l = length of array
    - Delete: O(l)
    - Insert: O(l)
    - Cache friendly
- Self Balancing BST (AVL Trees, Red Black Trees)
    - Search: O(log(l))
    - Delete: O(log(l))
    - Insert: O(l)
    - Not cache friendly
    - Java 8 onwards use this for HashMap

Open Addressing

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed).

Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.

Search(k): Keep probing until slot's key doesn't become equal to k or an empty slot is reached.

Delete(k): *Delete operation is interesting*. If we simply delete a key, then the search may fail. So slots of deleted keys are marked specially as "deleted".

The insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

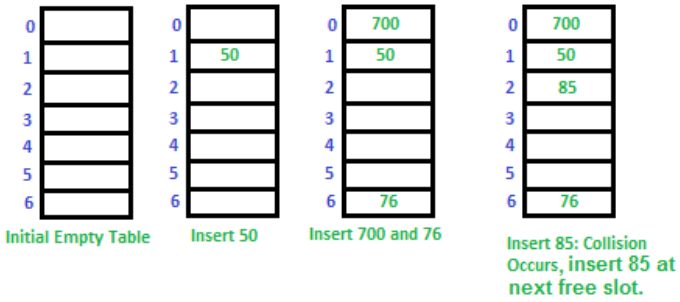
Open Addressing is done in the following ways:

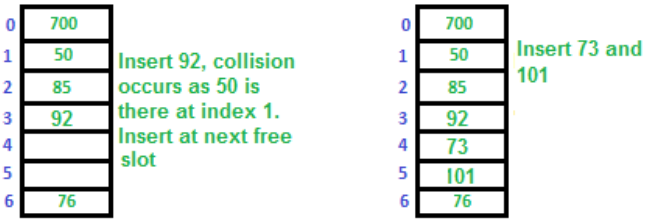
a) Linear Probing: In linear probing, we linearly probe for next slot. For example, the typical gap between two probes is 1 as seen in the example below.

Let hash(x) be the slot index computed using a hash function and S be the table size

If slot hash(x) % S is full, then we try (hash(x) + 1) % S
If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S
If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S
.....
.....

Let us consider a simple hash function as “key mod 7” and a sequence of keys as 50, 700, 76, 85, 92, 73, 101.





Challenges in Linear Probing :

1. Primary Clustering: One of the problems with linear probing is Primary clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search for an element.
2. Secondary Clustering*:* Secondary clustering is less severe, two records only have the same collision chain (Probe Sequence) if their initial position is the same.

b) Quadratic Probing We look for i2‘th slot in i’th iteration.

let hash(x) be the slot index computed using hash function.
If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S
If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S
If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S
.....
.....

c) Double Hashing We use another hash function hash2(x) and look for i*hash2(x) slot in i’th rotation.

let hash(x) be the slot index computed using hash function.
If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S
If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S
If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S
.....
.....

Comparison of above three:

Linear probing has the best cache performance but suffers from clustering. One more advantage of Linear probing is easy to compute.

Quadratic probing lies between the two in terms of cache performance and clustering.

Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

S.No.	Separate Chaining	Open Addressing
1.	Chaining is Simpler to implement.	Open Addressing requires more computation.
2.	In chaining, Hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full.
3.	Chaining is Less sensitive to the hash function or load factors.	Open addressing requires extra care to avoid clustering and load factor.
4.	Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Open addressing is used when the frequency and number of keys is known.
5.	Cache performance of chaining is not good as keys are stored using linked list.	Open addressing provides better cache performance as everything is stored in the same table.
6.	Wastage of Space (Some Parts of hash table in chaining are never used).	In Open addressing, a slot can be used even if an input doesn't map to it.
7.	Chaining uses extra space for links.	No links in Open addressing

Performance of Open Addressing:

Like Chaining, the performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of the table (simple uniform hashing)

m = Number of slots in the hash table
n = Number of keys to be inserted in the hash table

Load factor α = n/m (< 1)

Expected time to search/insert/delete < 1/(1 - α)

So Search, Insert and Delete take (1/(1 - α)) time

2] Coding Question from *sirf kwaab nhi ///code* QUESTION 1 TO QUESTION 6

3] 20 pages of Book Alchemist

4]LT

5] Way to Instagram