BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
CS F213
LAB 6

AGENDA

TIME: 02 Hrs

- Hash Set
- Array List
- Linked List
- Stack

6.1 HashSet

6.1.1 Introduction

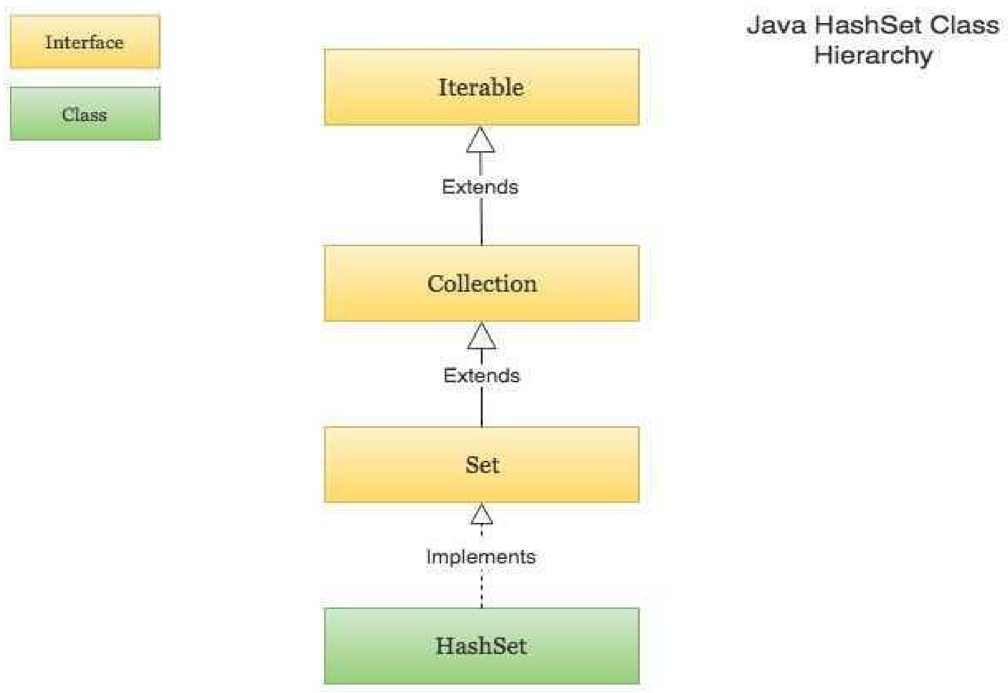
In this section, I'll walk you through the need for Set, as well as why we are adding the hashing functionality over the Set. Consider a situation, where we are trying to create a clone of the "WhatsApp" application. The tasks involved in it are:

- The application needs to import the WhatsApp linked contacts dynamically.
- Ensuring the uniqueness of the contacts in the application.
- Search, Insert the contacts to form a separate WhatsApp chat group rapidly.
- Search and remove the contacts as per User preference.

```
On an abstract level, We could say the importing valid users from mobile
contacts to WhatsApp at a rapid scale needs to have an efficient Searching
operation in order to insert or remove the contacts and also all the
imported contacts need to be unique which needs someone to take care of and
doesn't allow duplicates.
```

Set Interface which implements "**Collection**" Interface ensures the "uniqueness" of the contacts in the WhatsApp application as well as inside the WhatsApp groups. The Hashing functionality acts as an overlay on it which boosts the searching and deleting operations.

Hashing functionality is achieved using "*Hash Functions*", which will uniquely map the input data to a specified location inside the memory. So, if you ask this function where a particular data say "Student's name" is there, it will straight away use the mapping to fetch the required data within no time, making the search operation very fast which eventually improves operations like removal and insertion operations.



6.1.2 HashSet in Java

From the above figure, one could say it implements the “Set” interface. This is available in the “**java.util**” package.

We can create a HashSet for a particular object type, like Integer, String, Student, etc., using the following statement

```
HashSet<T> hs = new HashSet<T>();
//Here 'T' represents an Integer, String, Student, Employee, etc.,
```

We will see a program that provides more insights into the behavior of our beloved HashSet.

Program 6.1.a:

Here, we have a class called “**Student**”. We are exploring the use of the “add” in-built method of HashSet class. We will also see what type of elements can be considered to add inside the hashset and some other aspects.

```
import java.util.HashSet;
class Student{
    int id;
```

```java
    String name;
    Student(String name, int id){
        this.name = name;
        this.id = id;
    }
    void printDetails(){
       System.out.println(id+","+name);
    }
}
public class Main {
    public static void main(String[] args) {
        //Part1
        HashSet<Integer> hs1 = new HashSet<Integer>();
        System.out.println("Integer values: ");
        hs1.add(23);
        hs1.add(32);
        hs1.add(45);
        hs1.add(67);
        hs1.add(32);
        //Output1
        for(Integer i : hs1){
            System.out.println(i);
        }

        //Part2
        HashSet<Student> hs2 = new HashSet<Student>(3); //line1
        //----check size---- using size method
        hs2.add(new Student("Amit",1)); //line2
        Student s = new Student("Geetika",2);
        hs2.add(s);
        s = new Student("Teja",3); //line3
        hs2.add(s);//line4
        //---check size---
        hs2.add(new Student("Amit",1)); //line5
        hs2.add(new Student("Harshith",4));

        //Output2
        System.out.println("Student Details: ");
        for(Student t: hs2){
            t.printDetails();
        }
    }
}
```

```
/*
Hey JavaBeans!

In Part1 section, we are creating a HashSet which holds Integer type
elements:
1) We have added elements 23,32,45,67,32 [Insertion order] - one duplicate
we have right?
2) Try to see the output1, Can you see any duplicates?
3) Are the elements printed in the same order of insertion?
4) What are your inferences about HashSet working?


In Part2 section, we are creating HashSet which holds "Student" type
objects:
1) In Line1, the initial capacity of our hs2 is '3'. What will be the
initial size?
2) We have added (Amit,1)(Geetika,2)(Teja,3)(Amit,1)(Harshith,4) -
Insertion order
3) Will there be any problem at line3?
4) Do (Teja,3) be added inside the HashSet successfully at line4?

5) Is the size of hs2 growing?
6) Will the student object (Amit,1) in line5 be treated as a duplicate and
not added into hs2? If it adds, why? State your reasons.

*/
```

6.1.3 HashSet with Non-primitive types

Program 6.1.b:

This program will allow us to figure out how HashSet handles different kinds of non-primitive types. Here we have a base class “**Student**” and its children “**EnggStudent**” and “**MangStudent**” with their own data members and methods.

```
import java.util.HashSet;
class Student{
    int id;
    String name;
    Student(String name, int id){
        this.name = name;
        this.id = id;
    }
    void printDetails(){
       System.out.print(id+","+name+",");
    }
}
class EnggStudent extends Student{
    String branch;
    int year;
    EnggStudent(String name, int id, String branch, int year){
        super(name,id);
        this.branch=branch;
        this.year = year;
    }
    void printEngDetails(){
        printDetails();
        System.out.println(branch+","+year);
    }
}
class MangStudent extends Student{
    String branch;
    int year;
    MangStudent(String name, int id, String branch, int year){
        super(name,id);
        this.branch=branch;
        this.year = year;
    }
    void printMangDetails(){
        printDetails();
        System.out.println(branch+","+year);
    }
}
public class Main {
    public static void main(String[] args) {
        //part1
        Student s1 = new Student("Amit",1);
        Student s2 = new Student("Geetika",2);
        EnggStudent es1 = new EnggStudent("Teja",3,"cse",2016);
        EnggStudent es2 = new EnggStudent("Harsh",4,"ece",2016);
        MangStudent ms1 = new MangStudent("surya",5,"fin",2017);
        MangStudent ms2 = new MangStudent("ajith",6,"hr",2018);
        //part2
        HashSet<Student> stdhash = new HashSet<Student>();
        HashSet<EnggStudent> enghash = new HashSet<EnggStudent>();
        HashSet<MangStudent> mghash = new HashSet<MangStudent>();

       //part3
        stdhash.add(s1);
        enghash.add(es1);
        mghash.add(ms1);

        //part4
        /*
           Try to print the respective student details from each hashset
           using for-each loop
        */
        //part5
        /*Try to do the following operations:
        op1 => add s2 in "enghash" and "mghash".
        op2 => add es2 and ms2 in "stdhash".
        op3 => add es2 in "mghash".
        op4 => add ms2 in "enghash"

        Observations:
        1) Are all the above operations working fine?
        2) Which operations are causing trouble?
        3) Try to figure out which type-hashset is accepting
           Which type-objects?
           ex: is engg student hashset accepting management
               student object?
        Questions:
        1) Will a HashSet<Student> be able to accept
           Enggstudent objects?
        2) Does it means a HashSet of 'E' type can accept the
           classes instances that extends 'E'?
        3) Does a HashSet<MangStudent> able to accept Student
           type objects?
        4) Does this mean a HashSet of 'E' type can't be populated with
           its parents or superclasses instances?
        */
    }
}
```

6.1.4 Iterating through HashSet

As we know the one of iterating through the elements of HashSet is using our friend “**for-each**” loop right? But, we have another way also using “**Iterators**”. In the introduction, we have seen a figure which has the java hierarchy of HashSet, it starts with an interface called the “**Iterable**” interface.

This interface provides us with a method called “**iterator()**” which creates an iterator of type “**Iterator<T>**” where ‘T’ refers to the same type as of our HashSet which we are thinking of traversing.

I’ll be illustrating the program, which uses some in-built methods of the “**Iterator**” interface which is defined in the HashSet class, allowing us to traverse the elements inside our hashset.

The in-built methods of “Iterator” are:

- **hasNext( )** - helps us to know if there is any next element.
- **next( )** - it gives access to the current pointing element for operating upon, after that it increments the pointer to the next element.
- **remove( )** - it removes the current pointing element, but doesn’t increment the pointer.

Program 6.1.c:

```java
import java.util.HashSet;

class Student{
    int id;
    String name;
    Student(String name, int id){
        this.name = name;
        this.id = id;
    }
    String getName(){
        return name;
    }
    void printDetails(){
       System.out.println(id+","+name);
    }
}

public class Main {
    public static void main(String[] args) {
        //part1
        Student s1 = new Student("Amit",1);
        Student s2 = new Student("Geetika",2);
        Student s3 = new Student("Teja",3);
        Student s4 = new Student("stud4",4);
        //part2
        HashSet<Student> stdhash = new HashSet<Student>();
        //part3
        stdhash.add(s1);
        stdhash.add(s2);
        stdhash.add(s3);
        stdhash.add(s4);
        System.out.println("Using for-each loop");

       for(Student s : stdhash){
            s.printDetails();
        }
        //creating an iterator
        Iterator<Student> it = stdhash.iterator();
        System.out.println("Iterating using hashNext and next methods");
        while(it.hasNext() == true){
        //the above while condition means "is there any next element?".

          // Try to print it.next()
            Student stdit = it.next(); //line1
            if(stdit.getName() == "stud4"){
                it.remove();
          //line2 -> Try to add one more remove() here- see what happens
            }
            else{
               stdit.printDetails();
            }
        }
    }


/*
Questions:
1) Will there be any problem at line1?
2) See the behavior of the program, if we add more than one remove() call
at line2, when our iterator is pointing to the current element.
*/
```

6.1.5 In-built methods of HashSet

In this section, we will be looking into the in-built methods defined inside the HashSet class

Program 6.1.d:

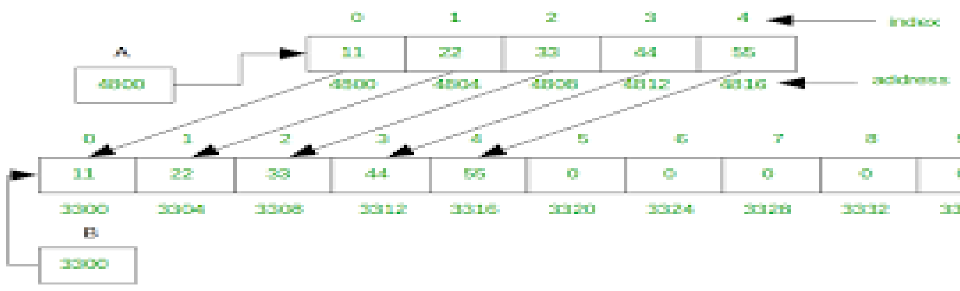
```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        String[] str1 = new String[]{"Hyd","Goa","Mumbai","Patna"};
        //line1
        HashSet<String> hs1 =new HashSet<String>(Arrays.asList(str1));
        System.out.println("Hashset1 elements: ");
        System.out.println(hs1);

        hs1.remove("Goa");   //remove specific elements

        System.out.println("After using remove method on hs1: ");
        System.out.println(hs1);

        HashSet<String> hs2=new HashSet<String>();

        String[] str2 = new String[]{"Kochi","Hyd","Kochi"};

        System.out.println("Empty status of hs2 is "+hs2.isEmpty());

        System.out.println("Adding elements in hs2: ");
        for(int i=0;i<str2.length;i++){
           if(hs2.add(str2[i])==true){  //line2
               System.out.println(i + " index element is added");
           }
            else{
               System.out.println(i + " index element is not added");
            }
        }
        System.out.println("Empty status of hs2 is "+hs2.isEmpty());
        System.out.println("Hashset2 elements are: " +hs2);

        hs1.addAll(hs2);  //Used to add one hashset into other hashset
        System.out.println("Updated Hashset after adding hs2 into hs1 :
\n"+hs1);
```

```
       //Removing all the new elements from HashSet
        hs1.removeAll(hs2);  //line3
        System.out.println("After using removeAll() method: \n"+hs1);

        //Clearing out the hashset
        hs1.clear();
        System.out.println("After using clear() method: "+ hs1);
    }
}

/*
Hey JavaBeans!
1) Observe that we are adding an array of strings to a hashset of type
   string at line1 using Arrays.asList() method, what if we directly
   pass 'str1' to hashset?
2) Could you depict the nature of the 'add' method from line2?
3) At line3, what if hs2 is an empty hashset?
*/
```

6.2 ArrayList

ArrayList class provides functionality of **dynamic arrays** in Java. It is found in the java.util package, and is a part of Java's collection framework. As expected, ArrayList maintains the insertion order, also allowing duplicate elements and random access. ArrayList can not be used with primitive types (int, char, etc.) without the use of wrapper classes.

Although actual library implementation is quite complex, basically **doubling strategy** is used for implementation of dynamic arrays. For details about the basic idea behind implementation of doubling strategy dynamic array, please refer to this article. Following image provides a sneak peak of the doubling strategy, wherein a 10-sized array is created, elements are copied and original memory is deleted.



src:- https://www.geeksforgeeks.org/how-do-dynamic-arrays-work/

6.2.1 Difference from Array

It differs from array in being **dynamic**, basically meaning the size of the array automatically increases when we dynamically add and remove items. However, the demerit is that its implementation is **slower than standard arrays** because a lot of shifting needs to occur if any element is removed from the array list. Hence, it can be useful in scenarios where array manipulation operations are frequent.

6.2.2 ArrayList Class Declarations

This is the declaration of ArrayList class in Java:-

```
public class ArrayList<E> extends AbstractList<E> implements List<E>,
RandomAccess, Cloneable, Serializable
```

As AbstractList class implements List interface, all the methods of List interface can be used in ArrayList.

6.2.3 ArrayList Constructors

ArrayList()	used to build an empty array list
ArrayList(Collection c)	used to build an array list initialized with the elements from the collection c
ArrayList(int capacity)	used to build an array list with initial *capacity* being specified

```
// different ArrayList constructors; old non generic way

ArrayList arr = new ArrayList();
ArrayList arr = new ArrayList(c);
ArrayList arr = new ArrayList(N);

// new generic way
ArrayList<String> list=new ArrayList<String>();
ArrayList<Integer> list = new ArrayList<Integer>();
```

6.2.4 Some Important ArrayList methods

add(int index, Object element)	Used to insert a specific element at a specific position index in a list.
add(Object o)	Used to append a specific element to the end of a list.
set(int index, Object element)	Used to set an element at a specified index.
remove(Object)	Used to remove the first occurrence of Object from ArrayList.
remove(int index)	Used to remove the element present at that specific index in the ArrayList and left-shift further elements.
get(index)	Used to get the element of a specified index.
trimToSize()	Used to trim the capacity of the instance of the ArrayList to the list's current size.
toArray()	Used to return an array containing all of the elements in the list in the correct order.

Note : For an exhaustive list of methods, please refer to this documentation article.

6.2.5 Exercises

Program 6.2.a:

```
 // Importing all utility classes
import java.util.*;

// Main class
public class Demonstration {

    // Main driver method
    public static void main(String args[])
    {
        // Creating an Array of string type
        ArrayList <String> arr = new ArrayList<>();

        // Adding elements to ArrayList

     // 1a Write the 2 lines to sequentially add "Learning", "Knowledge"
string to the ArrayList.

      // 1b. Write the line to add "Tricky" string , such that it is at the
second last position from left after insertion.


      // 1c. Print all the elements of the array and verify your example.

    }
}
```

Program 6.2.b:

```
//Importing all utility classes
import java.util.*;

// Main class
public class Demonstration  {

    // Main driver method
    public static void main(String args[])
    {
        // Creating an object of ArrayList class
        ArrayList<String> arr = new ArrayList<>();


        arr.add("STR1");
        arr.add("STR2");
        arr.add("STR3");
        arr.add(1, "STR1");

// 2a correct following line of code
       arr.set("STR5", 2);
// 2b remove the second element from arr, by adding a line.

 // 2c Try to write one line code code for printing ArrayList elements.
// 2d Removing only the first instance of "STR1"

// 2e print all ArrayList elements by advanced for loop

// 2f traverse arr using Iterator

// 2g check whether ArrayList is empty or not
           ArrayList<String> al2=new ArrayList<String>();
           al2.add("Sonoo");
           al2.add("Hanumat");
// 2h add all the elements of al2 to arr
// 2i finally sort the ArrayList arr
 }
}
```

Program 6.2.c:

3) Write a java program to instantiate an ArrayList from user input. Write a program to search for a specified integer from this given ArrayList, appropriating handling border cases like numbers not found in the list, etc. Use the binary search algorithm for searching the number, after sorting the input array. (please refer to the binary search algorithm, or if not possible then apply linear search only).

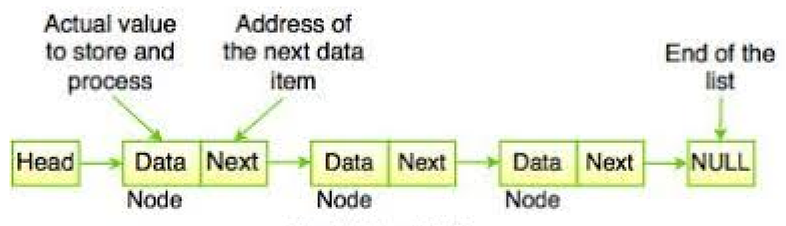
Example
5
4
2
6
8
9
8

This means user input is {4,2,6,8,9} ; 5 no of elements , and 8 is to be searched.

6.3 Linked List

A part of the Collection framework present in java.util. These are dynamically created memory units allocated in a non-contiguous way. Here as shown in figure it contains two fields:

1) Data part -> contains data
2) Next pointer -> contains address of next node.


Actual value to store and process
Address of the next data item
End of the list
Head
Data
Next
Data
Next
Data
Next
NULL
Node
Node
Node
Fig. Linked List

Where to use would be discussed in the latter part.

6.3.1 Implementation Details

The underlying data structure for the implementation of Linked List is Doubly Linked List & Deque DS. So, the operations of both Doubly Linked List and Deque are available.
However, **one drawback** is that :
They are *unsynchronised* which means no internal synchronization would be implemented, you need to impose external synchronization. [Will be taught later in the class].

A little bit about Deque :

Doubly Ended Queue, and is a linear structure that is efficiently modifiable at both ends, i.e. insertion/deletion is O(1) both at the beginning and at the end.

Features of Deque:
We can perform operations on the first and last element in constant time.

A little bit about Doubly linked List :
It has all the features of Linked List with some extra features :In a doubly linked list, a node consists of three parts:

1) *node data*,
2) p*ointer to the next node* in sequence (*next pointer*) ,
3) pointer to the previous node (previous pointer).

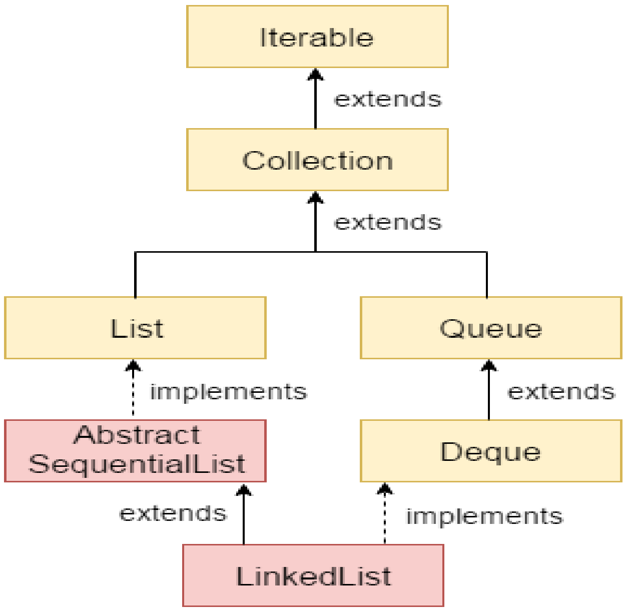
Features of Doubly Linked List:
Unlike Linked Link, we can move back and forth in constant time.

So, Linked List is the combination of features of Doubly Linked List & Deque. This makes Linked List really a great Data structure to use

Object Instantiation :

```
LinkedList<Type> linkedList = new LinkedList<>();
Here, Type indicates the type of a linked list. For example,
```

```
// create Integer type linked list
LinkedList<Integer> linkedList = new LinkedList<>();

// create String type linked list
LinkedList<String> linkedList = new LinkedList<>();
```



6.3.2 Methods of Java LinkedList

A lot of operations are provided by Linked List. But now we will only discuss the most commonly used functions :

- Add elements
- Access elements
- Change elements
- Remove elements

a) Add element : We can use the add() method to add an element (node) at the end of the LinkedList. For example

Program 6.3.a:

```
import java.util.*;
public class LinkedList1{
 public static void main(String args[]){

  LinkedList<String> al=new LinkedList<String>();
  al.add("BITS Pilani");
  al.add("CSIS");
  al.add("2020H1120271P");
  al.add("LastName");

  a1.add(1,"FirstName");    // Observe the statement
     animals.add(1, "Horse");



  Iterator<String> itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

Output :

```
LinkedList: [BITS Pilani, CSIS, 2020H1120271P, LastName]
Updated LinkedList: [BITS Pilani, CSIS, 2020H1120271P, FirstName, LastName]
```

b) get : get() method of the LinkedList class is used to access an element from the LinkedList.

Program 6.3.b:

```
import java.util.LinkedList;

class Main {
  public static void main(String[] args) {
    LinkedList<String> languages = new LinkedList<>();

    // add elements in the linked list
    languages.add("Python");
    languages.add("Java");
    languages.add("JavaScript");
    System.out.println("LinkedList: " + languages);

    // get the element from the linked list
    String str = languages.get(1);
    System.out.print("Element at index 1: " + str);
  }
}
```

c) Change Elements of a LinkedList: The set() method of the LinkedList class is used to change elements of the LinkedList. For example,

Program 6.3.c:

```
import java.util.LinkedList;

class Main {
  public static void main(String[] args) {
    LinkedList<String> languages = new LinkedList<>();

    // add elements in the linked list
    languages.add("Java");
    languages.add("Python");
    languages.add("JavaScript");
    languages.add("Java");
    System.out.println("LinkedList: " + languages);

    // change elements at index 3
    languages.set(3, "Kotlin");
    System.out.println("Updated LinkedList: " + languages);
  }
}

LinkedList: [Java, Python, JavaScript, Java]
Updated LinkedList: [Java, Python, JavaScript, Kotlin]
languages.set(3, "Kotlin");
Here, the set() method changes the element at index 3 to Kotlin.
```

d) Remove element from a LinkedList : The remove() method of the LinkedList class is used to remove an element from the LinkedList. For example,

Program 6.3.d:

```
import java.util.LinkedList;

class Main {
  public static void main(String[] args) {
    LinkedList<String> languages = new LinkedList<>();

    // add elements in LinkedList
    languages.add("Java");
    languages.add("Python");
    languages.add("JavaScript");
    languages.add("Kotlin");
    System.out.println("LinkedList: " + languages);

    // remove elements from index 1
    String str = languages.remove(1);
    System.out.println("Removed Element: " + str);

    System.out.println("Updated LinkedList: " + languages);
  }
}
```

Output :

```
LinkedList: [Java, Python, JavaScript, Kotlin]
Removed Element: Python
New LinkedList: [Java, JavaScript, Kotlin]
```

Note: You can use Linked List as Deque [Doubly ended Queue] using operations like removeFirst(), removeLast, addFirst(), addLast();

6.3.3: Accessing Linked List using Iterators

Program 6.3.e:

```
import java.util.*;
public class JavaExample{
   public static void main(String args[]){

      LinkedList<String> list=new LinkedList<String>();

      //Adding elements to the Linked list
      list.add("Steve");
      list.add("Carl");
      list.add("Raj");
      list.add("Negan");
      list.add("Rick");

      //Removing First element
      //Same as list.remove(0);
      list.removeFirst();

      //Removing Last element
      list.removeLast();

      //Iterating LinkedList
      Iterator<String> iterator=list.iterator();   //line 1
      while(iterator.hasNext()){
         System.out.print(iterator.next()+" ");
      }

      //removing 2nd element, the index starts with 0
      list.remove(1);

      System.out.print("\nAfter removing second element: ");
      //Iterating LinkedList again
      Iterator<String> iterator2=list.iterator();
      while(iterator2.hasNext()){
         System.out.print(iterator2.next()+" ");
      }
   }
}
// Observe Line 1 :
```

Note : You can use Linked List as Deque [Doubly ended Queue] using operations like removeFirst(), removeLast, addFirst(), addLast().

Ans: Iterator is an interface and Each class will have its own implementation.

Program 6.3.f:

```
import java.util.*;
public class LinkedListExample {
     public static void main(String args[]) {

       /* Linked List Declaration */
       LinkedList<String> linkedlist = new LinkedList<String>();

       /*add(String Element) is used for adding
        * the elements to the linked list*/
       linkedlist.add("Item1");
       linkedlist.add("Item5");
       linkedlist.add("Item3");
       linkedlist.add("Item6");
       linkedlist.add("Item2");

       /*Display Linked List Content*/
       System.out.println("Linked List Content: " +linkedlist);

       /*Add First and Last Element*/
       linkedlist.addFirst("First Item");
       linkedlist.addLast("Last Item");
       System.out.println("LinkedList Content after addition: "
+linkedlist);

       /*This is how to get and set Values*/
       Object firstvar = linkedlist.get(0);
       System.out.println("First element: " +firstvar);
       linkedlist.set(0, "Changed first item");
       Object firstvar2 = linkedlist.get(0);
       System.out.println("First element after update by set method: "
+firstvar2);

       /*Remove first and last element*/
       linkedlist.removeFirst();
       linkedlist.removeLast();
       System.out.println("LinkedList after deletion of first and last
element: " +linkedlist);

       /* Add to a Position and remove from a position*/
       linkedlist.add(0, "Newly added item");
       linkedlist.remove(2);
       System.out.println("Final Content: " +linkedlist);
     }
```

The operations for which the Linked List would be preferred over ArrayList :

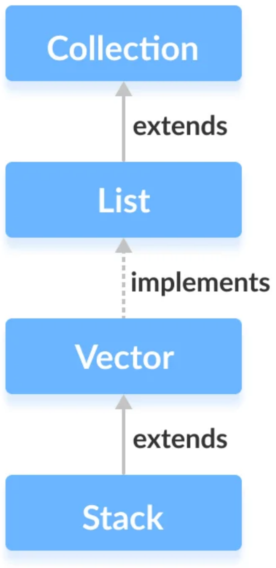
1) get(int index)
2) remove(int index)
3) iterator.remove()
4) ListIterator.add(E element)

6.3.5 : Exercise:

1. Try to imagine the above operations and figure out why [optional] ?
2. Write a Java program to iterate through all elements in a linked list.
3. Write a Java program to iterate through all elements in a linked list starting at the specified position.
4. Write a Java program to clone a linked list to another linked list.

6.4 Java Stack Class

The stack is a linear data structure that is used to store the collection of objects. It is based on Last-In-First-Out (LIFO). The Java collections framework has a class named Stack that provides the functionality of the stack data structure. The Stack class extends the Vector class.



6.4.1 Creating a Stack

In order to create a stack, we must import the java.util.Stack package first. Once we import the package, here is how we can create a stack in Java.

```
Stack<Type> stacks = new Stack<>();
```

Here, Type indicates the stack's type. For example,

```
// Create Integer type stack
Stack<Integer> stacks = new Stack<>();
// Create String type stack
Stack<String> stacks = new Stack<>();
```

6.4.2 Methods of the Stack Class

We can perform push, pop, peek and search operations on the stack. The Java Stack class provides mainly five methods to perform these operations. Along with this, it also provides all the methods of the Java Vector Class.

push() Method

To add an element to the top of the stack, we use the push() method. For example

Program 6.4.a

```
import java.util.Stack;

class Main {
    public static void main(String[] args) {
        Stack<String> animals= new Stack<>();

        // Add elements to Stack
        animals.push("Dog");
        animals.push("Horse");
        animals.push("Cat");

        System.out.println("Stack: " + animals);
    }
}
```

Output

```
 Stack: [Dog, Horse, Cat]
```

pop() Method

To remove an element from the top of the stack, we use the pop() method. For example,

Program 6.4.b

```
import java.util.Stack;

class Main {
    public static void main(String[] args) {
        Stack<String> animals= new Stack<>();

        // Add elements to Stack
        animals.push("Dog");
        animals.push("Horse");
```

```java
        animals.push("Cat");
        System.out.println("Initial Stack: " + animals);

        // Remove element stacks
        String element = animals.pop();
        System.out.println("Removed Element: " + element);
    }
}
```

Output

```
Initial Stack: [Dog, Horse, Cat]
Removed Element: Cat
```

peek() Method

The peek() method returns an object from the top of the stack. For example,

Program 6.4.c

```java
import java.util.Stack;

class Main {
    public static void main(String[] args) {
        Stack<String> animals= new Stack<>();

        // Add elements to Stack
        animals.push("Dog");
        animals.push("Horse");
        animals.push("Cat");
        System.out.println("Stack: " + animals);

        // Access element from the top
        String element = animals.peek();
        System.out.println("Element at top: " + element);

    }
}
```

```
Output

Stack: [Dog, Horse, Cat]

Element at top: Cat
```

search() Method

To search an element in the stack, we use the search() method. It returns the position of the element from the top of the stack. For example,

Program 6.4.d

```java
import java.util.Stack;

class Main {
    public static void main(String[] args) {
        Stack<String> animals= new Stack<>();

        // Add elements to Stack
        animals.push("Dog");
        animals.push("Horse");
        animals.push("Cat");
        System.out.println("Stack: " + animals);

        // Search an element
        int position = animals.search("Horse");
        System.out.println("Position of Horse: " + position);
    }
}
```

```
Output

Stack: [Dog, Horse, Cat]
Position of Horse: 2
```

empty() Method

To check whether a stack is empty or not, we use the empty() method. For example,

Program 6.4.e

```
import java.util.Stack;

class Main {
    public static void main(String[] args) {
        Stack<String> animals= new Stack<>();

        // Add elements to Stack
        animals.push("Dog");
        animals.push("Horse");
        animals.push("Cat");
        System.out.println("Stack: " + animals);

        // Check if stack is empty
        boolean result = animals.empty();
        System.out.println("Is the stack empty? " + result);
    }
}

Output
Stack: [Dog, Horse, Cat]
Is the stack empty? false
```

6.4.3 ALL in the same program

Program 6.4.f:

```
// Java code for stack implementation

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

```java
class Test
{
    // Pushing element on the top of the stack
    static void stack_push(Stack<Integer> stack)
    {
        for(int i = 0; i < 5; i++)
        {
            stack.push(i);
        }
    }

    // Popping element from the top of the stack
    static void stack_pop(Stack<Integer> stack)
    {
        System.out.println("Pop Operation:");

        for(int i = 0; i < 5; i++)
        {
            Integer y = (Integer) stack.pop();
            System.out.println(y);
        }
    }

    // Displaying element on the top of the stack
    static void stack_peek(Stack<Integer> stack)
    {
        Integer element = (Integer) stack.peek();
        System.out.println("Element on stack top: " + element);
    }

    // Searching element in the stack
    static void stack_search(Stack<Integer> stack, int element)
    {
        Integer pos = (Integer) stack.search(element);

        if(pos == -1)
            System.out.println("Element not found");
        else
            System.out.println("Element is found at position: " + pos);
    }
    public static void main (String[] args)
    {
        Stack<Integer> stack = new Stack<Integer>();

        stack_push(stack);
        stack_pop(stack);
        stack_push(stack);
        stack_peek(stack);
        stack_search(stack, 2);
        stack_search(stack, 6);
    }
}
Output:
Pop Operation:
4
3
2
1
0
Element on stack top: 4
Element is found at position: 3
Element not found
```

6.4.5 Exercise:

1. Write a Java program to iterate through all elements in a stack.
2. Write a Java program to iterate through all elements in a stack starting at the specified position.
3. Write a Java program to clone a stack to another stack.
4. Can you implement a stack with a generic type ? Hint: Could be a student type