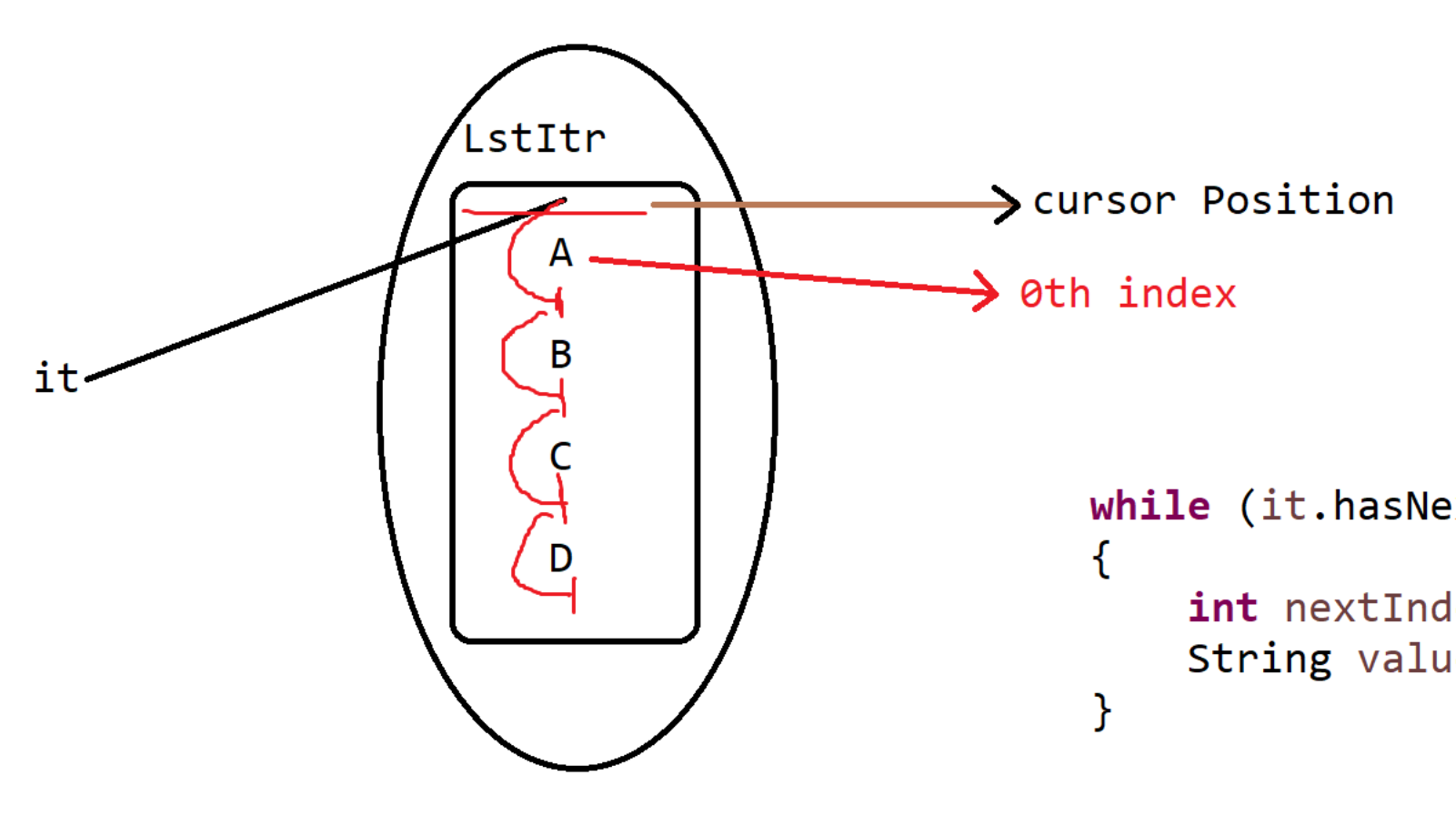
How ListIterator works internally ?
----
* ListIterator works on the basis of index that is the reason It will only working with List<E> interafce.

Diagram for the below program :

```
list.add("A");
list.add("B");
list.add("C");
list.add("D");

// Start ListIterator from index 0
ListIterator<String> it = list.listIterator();
```



[Cursor and index both are having different location]

```
while (it.hasNext())
{
    int nextIndex = it.nextIndex();
    String value = it.next();
}
```

ListIterator Methods :
----
1) public int nextIndex() : It will return the index position of the next position element which is available after the cursor.

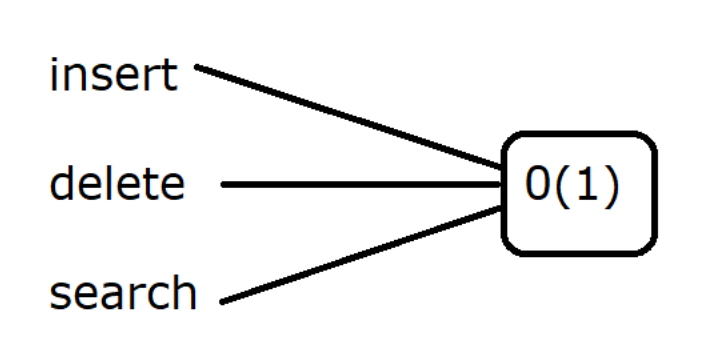
1) public int previousIndex() : It will return the index position of the previous position element which is available before the cursor.

```java
package com.ravi.hash_set_demo;


import java.util.LinkedList;
import java.util.ListIterator;

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

        list.add("A");
        list.add("B");
        list.add("C");
        list.add("D");

        // Start ListIterator from index 0
        ListIterator<String> it = list.listIterator();

        System.out.println("Forward Traversal:");
        while (it.hasNext())
        {
            int nextIndex = it.nextIndex();
            String value = it.next();
            System.out.println("Index " + nextIndex + " → " + value);
        }

        System.out.println("\nBackward Traversal:");
        while (it.hasPrevious())
        {
            int prevIndex = it.previousIndex();
            String value = it.previous();
            System.out.println("Index " + prevIndex + " → " + value);
        }


        System.out.println("\nStart Iterator from index 2:");

        ListIterator<String> it2 = list.listIterator(2);
        while (it2.hasNext())
        {
            System.out.println("Index " + it2.nextIndex() + " → " + it2.next());
        }
```

Difference between Iterator<E> and ListIterator<E> :

Iterator<E>	ListIterator<E>
1) Can move in forward direction only.	1) Can move in both forward and backward direction.
2) During Iteration, We can only remove the elements.	2) During Iteration, We can remove, add and replace the elements.
3) Can work with List<E>, Set<E> and Queue<E> interface.	3) Can work with only List<E> interface.
4) We cannot get index position of the element.	4) We can get index position of the element by using nextIndex() and previousIndex()
5) Does not provide index technique.	5) It supports indexing technique.

==
What is hashing algorithm ?
----
* By using hashing algorithm technique, We can insert, delete and search an element with 0(1) time complexity.



* It is more efficient than our classical indexing technique, when the data size is very large.

* It uses **Hashtable data structure and to get 0(1) time complexity It uses hashing function technique.**

* Hashtable data structure internally uses one more data structure i.e **Bucket data structure which is implemented by Node[] array class.**

Example of Hashtable data structure :

Let suppose, We want to insert the following data by using Hashtable data structure using hashing function. [no of buckets = table length]

92, 18, 23, 37, 55, 64

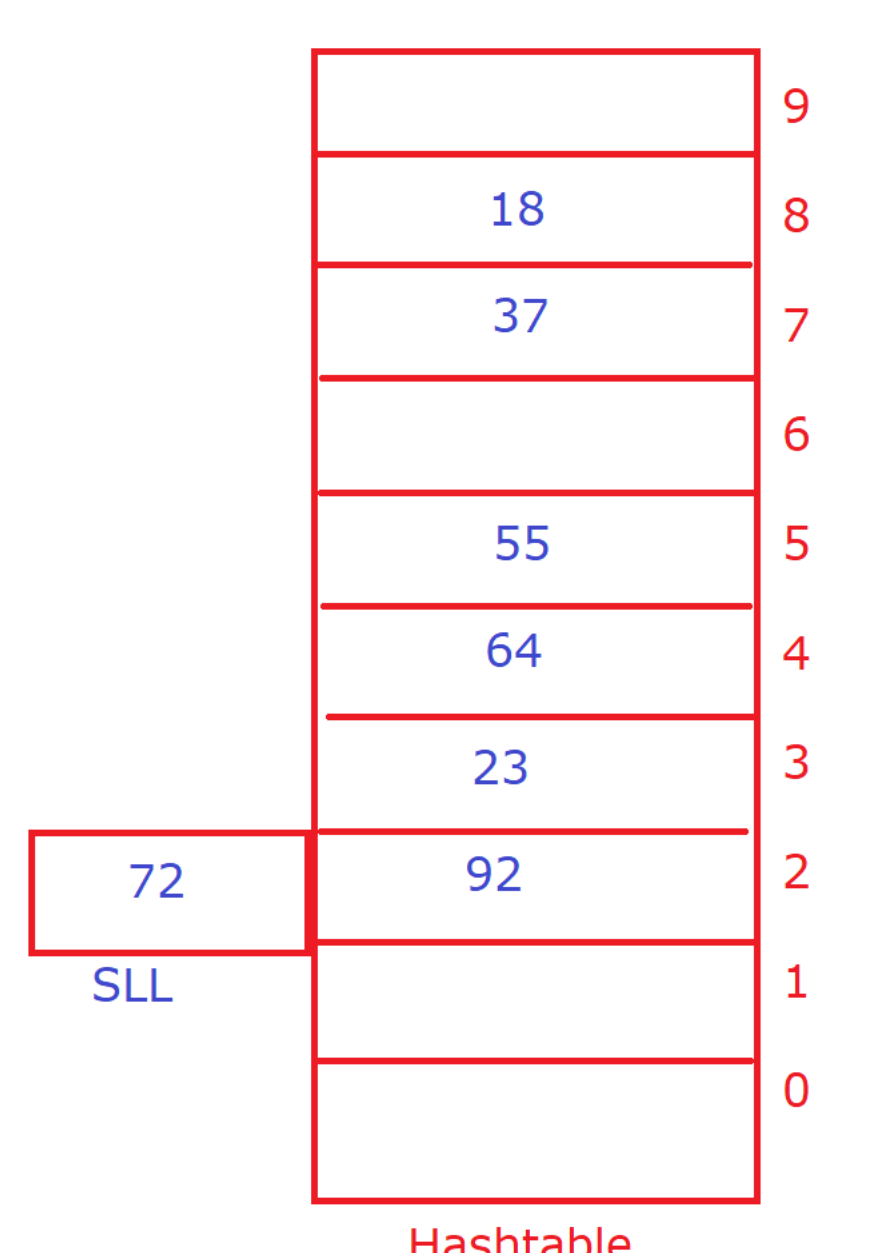
Hashing Function :

bucket index = key % table length

92 % 10 = 2

I want to insert one more element i.e 72
72 % 10 = 2

But in the 2nd Bucket index already 92 element is available It is called **Hash Collision.** In order to solve this collison we should use SLL (Singly LinkedList)



HashSet<E> [UNORDERED, UNSORTED, NO DUPLICATS]
----
public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, Serializable

* It is an implemented class of Set<E> interface available from JDK 1.2V.
* It does not accept duplicate value.
* It can accept null, homogeneous and hetrogeneous elements.
* It is an un-ordered and un-sorted set.
* It uses **Hashtable data structure**.
* THE ELEMENTS ARE INSERTED BASED ON THE **hashCode()** of the object.
* The default capacity is 16 so initially 16 buckets will be created in the Hashtable.
* Methdos are not synchronized.
* Ieartor is Fail Fast Iterator
* HashSet internally uses HashMap object.
* It is mainly used to search an element in the large amount of data.
* Here hashCode() and equals() contarct playing a major role.
* This class offers constant time performance for the basic operations (add, remove, contains and size

We have 4 types of constructor :
----
1) HashSet hs1 = new HashSet();
It will create the HashSet Object with default capacity is 16 (Internally HashMap object). The default load fator or Fill Ratio is 0.75 (75% of HashSet is filled up then new HashSet Object will be created having double capacity and elements are inserted by using Re-Hashing)

2) HashSet hs2 = new HashSet(int initialCapacity);
will create the HashSet object with user specified capacity.

3) HashSet hs3 = new HashSet(int initialCapacity, float loadFactor);
we can specify our own initialCapacity and loadFactor(by default load factor is 0.75)

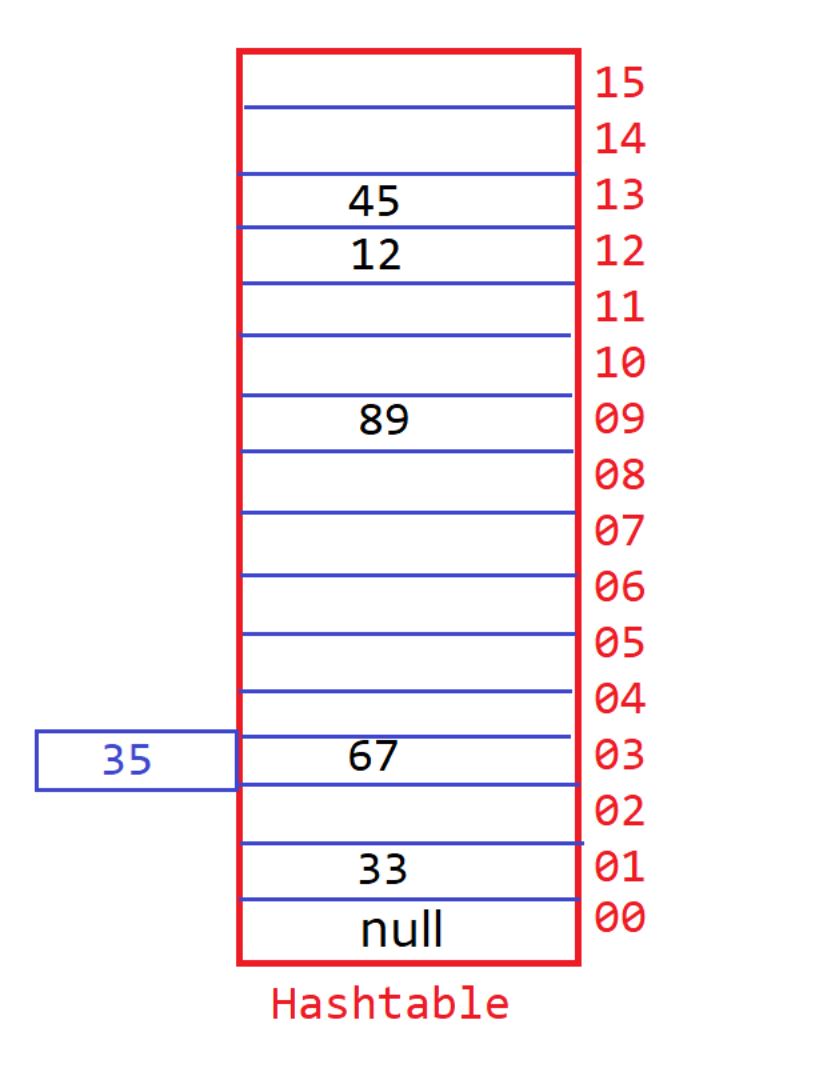
4) HashSet hs4 = new HashSet(Collection c);
Interconversion of Collection.

//Program on HashSet :

```java
package com.ravi.hash_set_demo;

import java.util.HashSet;

public class HashSetDemo1
{
 public static void main(String args[])
 {
        HashSet<Integer> hs = new HashSet<>();
        hs.add(67);
        hs.add(89);
        hs.add(33);
        hs.add(45);
        hs.add(12);
        hs.add(35);

        hs.forEach(num-> System.out.println(num));
    }
}
```

Diagram for the above program :

```
HashSet<Integer> hs = new HashSet<>();
      hs.add(67);
      hs.add(89);
      hs.add(33);
      hs.add(45);
      hs.add(12);
      hs.add(35);
      hs.add(null);

      Bucket = key % Table length

               67 % 16

 Output : Bottom to top and right to left
          [33, 67, 35, 89, 12, 45]
```



Two important concept with this :
----
1) Ravi % 16
true % 16
false % 16
new Employee() % 16

then they introduced **hashCode()** method

key.hashCode() % Table length

2) What about null ?

null.hashCode() -> NullPointerException

They have provided a fixed hash code value for null i.e 0