**NAME :- ABHI METHA**

**BRANCH A :- IT**

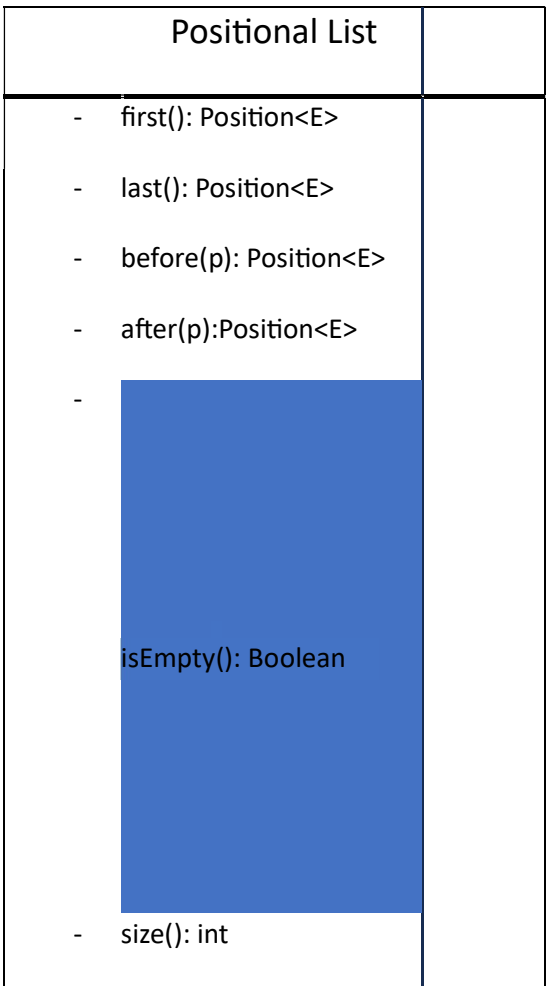
**REG NO – 221080001**

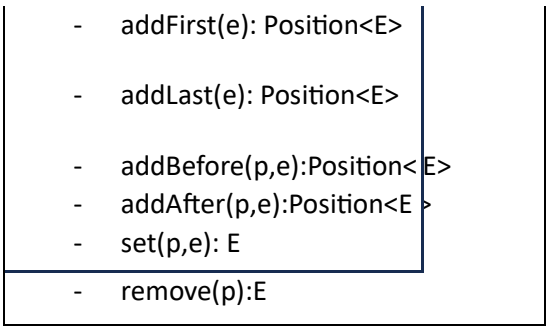
**FDS LAB ASSIGNMENT 5**

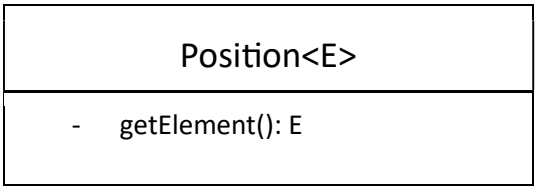
# Laboratory-5

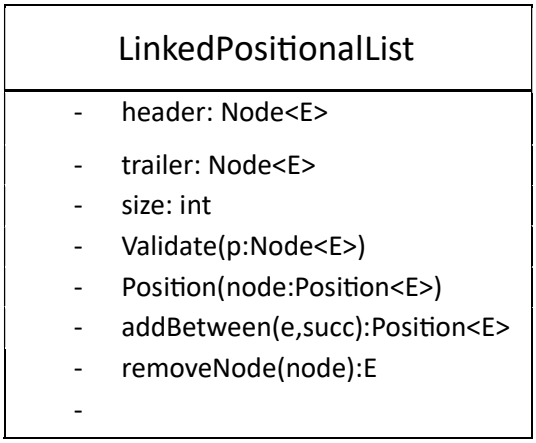
**Aim**-Implement the positional list (all accessor and update methods) using linked list and implement an iterator interface for the same with the help of position interface and iterator interface.

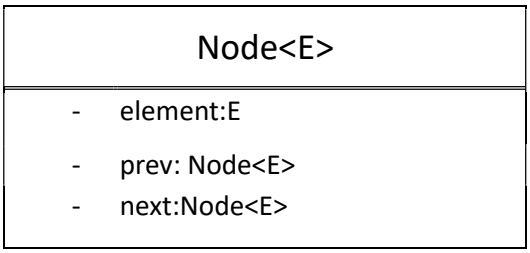
## Class Diagram-






## Description of Java constructs used and the methods-

- Packages- We need to import packages such as Iterator which allows you to perform operations like iterating over the elements and NoSuchElementException which is important to handle when working with iterators.

- ‘implements’ keyword- The ‘implements’ keyword is used when a class is used to implement an interface i.e all the methods declared in the interface.
- Constructors- Constructors are special method functions having the same name as the class and are automatically called when the object is created.In our experiment we used the constructorsLinkedPositionalList,
- ‘instanceof’ keyword- The ‘instanceof’ keyword is used to check if the object is an instance of the class or interface or not.
- Exceptions- Exceptions such as ‘IllegalArgumentException’ or ‘IllegalStateException’ are thrown after checking current conditions.
- Incrementation/Decrementation- We increment or decrement the size of the positional list on addition or deletion of elements.
- Access Specifiers- We specify the access of certain variables and methods as public or private depending on our need.
- Interfaces- Interfaces in java are a way to define or specify the set or group of method functions that a class must perform (can say is required to perform).
- Conditional statements- We use conditional statements like do while loop ,for loop(for traversing the list),if else statements and switch case in order display the menu and perform operations based on the choice chosen.

## CODE-

**Position.java**

```
/**
* Position interface
 *
* @param <E> the type of element stored in the position
*/
```

```
public interface Position<E> {

   /**
* @returns the element stored at this position.
    *
* @throws IllegalStateException if position not valid
    */
   E getElement() throws IllegalStateException;

}
```

**Node.java**

```
/**
* Node class of a doubly linked list which provides implementation of Position
  interface
* @param <E> the type of element stored in the node
*
 */
public class Node<E> implements Position<E> {

   private E element;      private
Node<E> prev;     private
Node<E> next;

   /**
* Constructs a new Node object
* @param e the element to be stored in the node
* @param p the previous node
```

```
* @param n the next node
     */
    public Node(E e, Node<E> pr, Node<E> nx) {
this.element = e;        this.prev = pr;        this.next
= nx;
    }

    /**
     *
* @return the element stored in this node
* @throws IllegalStateException if the position is invalid
*      */
    public E getElement() throws IllegalStateException {        if
(next == null)
          throw new IllegalStateException("Position is no longer valid");            else
return element;
    }

    /**
* Returns the previous node
* @return the previous node
     */
    public Node<E> getPrev() {
return prev;
    }
```

```
   /**
* Returns the next node
* @return the next node
    */
   public Node<E> getNext() {
return next;
   }

   /**
* Sets the element stored in the node to the given element.
* @param e the new element to be stored
    */
   public void setElement(E e) {
      this.element = e;
   }

   /**
* Sets the next node in the list to the given node.
* @param n the new next node in the list
    */
   public void setNext(Node<E> nx) {        this.next
= nx;
   }

   /**
* Sets the previous node in the list to the given node.
```

```
* @param p the new previous node in the list
     */
    public void setPrev(Node<E> pr) {        this.prev
= pr;
    }
}
```

**PositionalListInterface.java**

```
/**
* Positional List Interface
* @param <E> the type of element IN the list
 */
public interface PositionalListInterface<E> {

    /**
* Returns the number of elements in the list.
*      */     int size();

    /**
* Checks whether the list is empty.
*      */
    boolean isEmpty();

    /**
* Returns the first position in the list.
     *
     */
```

```
  Position<E> first();

  /**
* Returns the last position in the list.
   *
   */
  Position<E> last();

  /**
* Returns the position before the given position.
   *
* @throws IllegalArgumentException if the position is invalid
   */
  Position<E> before(Position<E> p) throws IllegalArgumentException;

  /**
* Returns the position after the given position.
   *
* @throws IllegalArgumentException if the position is invalid
   */
  Position<E> after(Position<E> p) throws IllegalArgumentException;

  /**
* Inserts an element at the beginning of the list.
   *
   */
```

```
  Position<E> addFirst(E e);

  /**
* Inserts an element at the end of the list.
   *
   */
  Position<E> addLast(E e);

  /**
* Inserts an element before the given position.
   *
* @throws IllegalArgumentException if the position is not valid
   */
  Position<E> addBefore(Position<E> p, E e) throws IllegalArgumentException;

  /**
* Inserts an element after the given position.
   *
* @throws IllegalArgumentException if the position is not valid
   */
  Position<E> addAfter(Position<E> p, E e) throws IllegalArgumentException;

  /**
* Replaces the element at the given position with the given element.
   *
* @throws IllegalArgumentException if the position is not valid
```

```
     */
    E set(Position<E> p, E e) throws IllegalArgumentException;

    /**
* Removes the element at the given position.
     *
* @throws IllegalArgumentException if the position is not valid
     */
    E remove(Position<E> p) throws IllegalArgumentException;
}
```

**LinkedPositionalList.java** import

```
java.util.Iterator;
import java.util.NoSuchElementException; import
java.lang.Iterable;



/**
* PositionalListInterface implementation using linked list
 *
* @author MethaAbhi
* @version 1.0
 *
 */

/**
* A doubly linked list implementation of the PositionalList interface.
```

```
* @param <E> the type of elements held in this collection
 */
public class LinkedPositionalList<E> implements PositionalListInterface<E> {



   private int size;     private
Node<E> header;      private
Node<E> trailer;


   /**
* Constructs an empty linked positional list.
    */
   public LinkedPositionalList() {        this.size
= 0;
       this.trailer = new Node<>(null, null, null);        this.header
= new Node<>(null, null, this.trailer);
this.trailer.setPrev(this.header);
   }


   /**
* Validates that the given position is a valid node in the list.
* @param p the position to be validated
* @return the node corresponding to the given position
* @throws IllegalArgumentException if the position is invalid
    */
```

```
    private Node<E> validateNode(Position<E> p) throws
IllegalArgumentException {        if (!(p instanceof Node))

          throw new IllegalArgumentException("Passed position is invalid");

Node<E> node = (Node<E>) p;         if (node.getNext() == null)

          throw new IllegalArgumentException("The passed node is
defunct");        return node;

   }


   /**

* Returns the position corresponding to the given node

* @param node the node to be converted to a position

* @return the position corresponding to the given node or null if the node is the
  header or trailer

    */

   private Position<E> position(Node<E> node) {        if
(node == this.header || node == this.trailer)

return null;        else            return node;

   }


   /**

* Adds a new node with the given element between the given predecessor and
  successor nodes.

* @param t the element to be stored in the new node

* @param predecessor the node that will be the predecessor of the new node

* @param successor the node that will be the successor of the new node

* @return the position of the newly added node

    */
```

```
   private Position<E> addBetween(E e, Node<E> predecessor, Node<E>
successor) {
      Node<E> newNode = new Node<>(e, predecessor, successor);
predecessor.setNext(newNode);        successor.setPrev(newNode);
      ++this.size;
      return position(newNode);
   }

   /**
* Returns the number of elements in the list.
* @return the number of elements in the list
    */
   public int size() {
return this.size;
   }

    /**
* Returns true if the list is empty, false otherwise.
* @return true if the list is empty, false otherwise
    */
   public boolean isEmpty() {
return this.size == 0;
   }

   /**
* Returns the position of the first element in the list, or null if the list is empty.
* @return the position of the first element in the list, or null if the list is empty
```

```
    */
   public Position<E> first() {
      return position(this.header.getNext());
   }

   /**
* Returns the position of the last element in the list, or null if the list is empty.
* @return the position of the last element in the list, or null if the list is empty
    */
   public Position<E> last() {
      return position(this.trailer.getPrev());
    }
   /**
* Returns the position of the element immediately before the given position.
* @param p the position whose predecessor is to be returned
* @return the position of the element immediately before the given position
* @throws IllegalArgumentException if the given position is not a valid node in
  the list
    */
   public Position<E> before(Position<E> p) throws IllegalArgumentException {
      Node<E> node = validateNode(p);        return
position(node.getPrev());
   }

   /**
* Returns the position of the element immediately after the given position.
```

```
* @param p the position whose successor is to be returned
* @return the position of the element immediately after the given position
* @throws IllegalArgumentException if the given position is not a valid node in
  the list
   */
    public Position<E> after(Position<E> p) throws
IllegalArgumentException {       Node<E>
node = validateNode(p);       return
position(node.getNext());
  }

  /**
* Adds a new element to the beginning of the list.
* @param t the element to be added
* @return the position of the newly added element
   */
  public Position<E> addFirst(E e) {
     return addBetween(e, this.header, this.header.getNext());
  }

  /**
* Adds a new element to the end of the list.
* @param t the element to be added
* @return the position of the newly added element
   */
  public Position<E> addLast(E e) {
     return addBetween(e, this.trailer.getPrev(), this.trailer);
```

```
    }

    /**
* Adds a new element immediately before the given position.
* @param p the position before which the new element is to be added
* @param t the element to be added
* @return the position of the newly added element
* @throws IllegalArgumentException if the given position is not a valid node in
   the list
     */

    public Position<E> addBefore(Position<E> p, E e) throws
IllegalArgumentException {
        Node<E> node = validateNode(p);        return
addBetween(e, node.getPrev(), node);
    }

    /**
* Adds a new element immediately after the given position.
* @param p the position after which the new element is to be added
* @param t the element to be added
* @return the position of the newly added element
* @throws IllegalArgumentException if the given position is not a valid node in
   the list
     */
    public Position<E> addAfter(Position<E> p, E e) throws
IllegalArgumentException {
```

```
      Node<E> node = validateNode(p);        return
addBetween(e, node, node.getNext()); }
   /**
* Replaces the element stored at the given position with the given element.
* @param p the position of the element to be replaced
* @param t the new element to be stored at the given position
* @return the old element that was replaced
* @throws IllegalArgumentException if the given position is not a valid node in
  the list
    */
   public E set(Position<E> p, E e) throws IllegalArgumentException {
      Node<E> node = validateNode(p);
E temp = node.getElement();
node.setElement(e);        return temp;
   }

   /**
* Removes the element stored at the given position from the list.
* @param p the position of the element to be removed
* @return the element that was removed
* @throws IllegalArgumentException if the given position is not a valid node in
  the list
    */
   public E remove(Position<E> p) throws IllegalArgumentException {
Node<E> node = validateNode(p);        E temp = node.getElement();
node.getPrev().setNext(node.getNext());
node.getNext().setPrev(node.getPrev());        size--;
```

```
        // unlink node
node.setPrev(null);
node.setNext(null);
node.setElement(null);        return
temp;
    }

    /**
* Returns an iterable representation of the list's positions.
     *
     */

   private class PositionIterator implements Iterator<Position<E>> {

        /**
* The cursor is the position of the next element to be returned.
* The recent is the position of the last element returned.
         */
        private Position<E> cursor = first();
        Position<E> recent = null;



        /**
* Returns true if the iterator has a next element, false otherwise.
* @return true if the iterator has a next element, false otherwise
         */
```

```
        public boolean hasNext() {
return cursor != null;
        }

        /**
* Returns the next element in the iteration.
* @return the next element in the iteration
* @throws NoSuchElementException if there are no more elements in the
  iteration
         */
        public Position<E> next() throws NoSuchElementException {            if
(cursor == null)
                throw new NoSuchElementException("No element found");            else
{
                recent = cursor;
cursor = after(cursor);
return recent;
           }
        }
        /**
* Removes the element returned by the most recent call to next().
* @throws IllegalStateException if there is no element to be removed
         */
        public void remove() throws IllegalStateException {            if
(recent == null)
```

```
            throw new IllegalStateException("No element can be removed");
LinkedPositionalList.this.remove(recent);           recent = null;
      }
   }
   /**
* Position Iterable is a private class that implements the Iterable interface and is
   used to iterate over the positions in the list.
    */

   private class PositionIterable implements Iterable<Position<E>> {

      public Iterator<Position<E>> iterator() {
return new PositionIterator();
      }
   }
   /**
* Returns an iterator of the elements stored in the list.
* @return an iterator of the elements stored in the list
    */
   public Iterable<Position<E>> positions() {        return
new PositionIterable();
   }
   /**
* Element Iterator is a private class that implements the Iterator interface and is
   used to iterate over the elements in the list.
    */
   private class ElementIterator implements Iterator<E> {
```

```
    /**
* The position iterator used to iterate over the positions in the list.
     */
    Iterator<Position<E>> posIterator = new PositionIterator();
    /**
* Returns true if the iterator has a next element, false otherwise.
* @return true if the iterator has a next element, false otherwise
     */
    public boolean hasNext() {
return posIterator.hasNext();
    }
    /**
* Returns the next element in the iteration.
* @return the next element in the iteration
* @throws NoSuchElementException if there are no more elements in the
  iteration
     */
    public E next() {
       return posIterator.next().getElement();
    }
    /**
* Removes the element returned by the most recent call to next().
* @throws IllegalStateException if there is no element to be removed
     */
    public void remove() {
posIterator.remove();
```

```
        }
    }
    /**
* Returns an iterator of the elements stored in the list.
* @return an iterator of the elements stored in the list
     */
    public Iterator<E> iterator() {
return new ElementIterator();
    }
}
```

**TestApplication.java** import

java.util.Iterator; import

java.util.Scanner;

```
/**
* The class TestApplication displays a menu for the operations to be performed.
* The operations such as inserting at the beginning,middle, end and deleting from
  the beginning ,middle,end
 */
class TestApplication {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        LinkedPositionalList<String> PositionList = new LinkedPositionalList<>();
        int choice;
        do {display(PositionList);
```

```
System.out.println("Menu\n"+"1. First element\n"+ "2. Last element\n"+
"3. Add first\n"+ "4. Add last\n"+ "5. Add before\n"+ "6. Add after\n" + "7.
Remove\n"+ "8. Set\n" + "9. Display\n"+ "10. Exit\n");
System.out.println("Enter your choice: ");           choice = sc.nextInt();
sc.nextLine();           switch (choice) {


case 1: if
(PositionList.isEm
pty()) {
                System.out.println("Positional list is empty");
            } else {
                System.out.println("First element is: " +
PositionList.first().getElement());
            }
break;              case
2:
            if (PositionList.isEmpty()) {
                System.out.println("Positional list is empty");
            } else {
                System.out.println("Last element is: " +
PositionList.last().getElement());
            }
break;              case
3:
            // enters an element at the front of the list returning the position of
the new element
            System.out.println("Enter the element you want to add First: ");
```

```
                String temp = sc.nextLine();
                PositionList.addFirst(temp);
// display(PositionList);                    break;

case 4:

                        System.out.println("Enter the element you want to add Last: ");
                temp = sc.nextLine();
                PositionList.addLast(temp);


                break;
case 5:
                System.out.println("Enter the element you want to add: ");
                String element = sc.next();
                System.out.println("Enter the position before which you want to add :
");
                int position = sc.nextInt();
                if(position<1 || position>PositionList.size()) {
System.out.println("Invalid position");                    break;
                }
else{
                  Position<String>    p    =    PositionList.first();
for (int i = 1; i < position; i++) {                              p =
PositionList.after(p);
                  }
```

```
                PositionList.addBefore(p, element);
break;
            }
case 6:
            System.out.println("Enter the element you want to add: "); element =
sc.next();
            System.out.println("Enter the position after which you want to add : ");
            position = sc.nextInt();
            if(position<1 || position>PositionList.size()) {
System.out.println("Invalid position");                    break;
            }
else{
                Position<String>    p    =    PositionList.first();
for (int i = 1; i < position; i++) {                          p =
PositionList.after(p);
                }
                PositionList.addAfter(p, element);
            }
break;            case
7:
            System.out.println("Enter the position: ");
int pos = sc.nextInt();
            if(pos<1 || pos>PositionList.size()) {
System.out.println("Invalid position");                    break;
            }
```

```
                Position<String> p = PositionList.first();
for (int i = 1; i < pos; i++) {
                   p = PositionList.after(p);
                }
                PositionList.remove(p);
break;              case 8:
                System.out.println("Enter the position: ");
pos = sc.nextInt();
                if(pos<1 || pos>PositionList.size()) {
System.out.println("Invalid position");                    break;
                }
                p = PositionList.first();
for (int i = 1; i < pos; i++) {                  p
= PositionList.after(p);
                }
                System.out.println("Enter the element you want to add: ");
element = sc.next();
                PositionList.set(p, element);
break;              case 9:
                // display is already implemented
break;              default:
                System.out.println("Exiting the program ...");
                System.out.println("Thank you for using the program");
break;


        }
```

```
        } while (choice < 10 && choice > 0);

        Iterable<Position<String>> posIter = PositionList.positions();        for
(Position<String> p : posIter) {
            System.out.print(p.getElement() + " --> ");
        }
        System.out.println("\n");

        System.out.println("Searching the positional list for a node");
        System.out.println("Enter the Element you want to search: ");
String to_search = sc.nextLine();        int pos = 0;
        for (Position<String> p : posIter) {
pos++;
            if (p.getElement().equals(to_search)) {
                System.out.println("Element " + to_search + " found at position "
+ pos);
                return;
            }
        }
        System.out.println("Element not found");
    }
    public static <E> void display(LinkedPositionalList<E> posList) {        if
(posList.isEmpty()) {
            System.out.println("Positional list is empty" + "\n");
        } else {
```

```
        System.out.println("Elements of the positional list are: ");
Iterator<E> i = posList.iterator();          while (i.hasNext()) {
           System.out.print(" -- "+ i.next());
        }
        System.out.println("\nSize of the positional list is: " + posList.size());
        System.out.println("First node is :" + posList.first().getElement());
        System.out.println("Last node is :" + posList.last().getElement() + "\n");
     }
  }
}
```

## Errors encountered-

- Declaring the variable with wrong data type gave error which was resolved by rechecking the needs of the variable such as element and position.
- Implementing two wrongly connected classes gave error which required proper connection depending on the need of execution.
- Not taking input wherever required gave error.
- Passing correct parameters with the correct data type as per the need of the function resolved some errors.
- Basic indentation and semicolon error were also encountered to some extent.

## LEARNINGS-

I learned that in order to implement positional list it requires a lot of interfaces such as that of node, position and doubly linked list. It was a very difficult data structure to learn and explore as it included so many accessor and update methods to be implemented having its own technicalities which were not easy to understand initially but with the help of a classmate I could understand them slowly. It required to link each of the interfaces very carefully for proper execution of the code step by step. I also learned that proper knowledge of each interface is required.

## CONCLUSION-

Thus we conclude that a Doubly Linked List, when combined with an Iterator interface, forms a robust foundation for creating a Positional List. This implementation depends on the meticulous handling of nodes and positions. In return, the Positional List provides an extensive array of techniques for accessing, altering, inserting, and removing elements at designated positions.