## Topics

1. Create Position Interface
2. Create Tree Interface
3. Create AbstractTree Class.
4. Create BinaryTree Interface
5. Create AbstractBinaryTree Class
6. Create Linked Binary Tree Using Linked List structure (Nodes)
7. Implement Basic Methods of LinkedBinaryTree

* addRoot( E e)
* addLeft(Position<E> p ,E e)
* addRight(Position<E> p ,E e)
* set(Position<E> p ,E e)
* remove(Position<E> p)

import java.util.Iterator;

import java.util.ArrayList;

import java.util.List;

// 1. واجهة Position

public interface Position<E> {

E getElement() throws IllegalStateException;

}

// 2. واجهة Tree

public interface Tree<E> extends Iterable<E> {

Position<E> root();

Position<E> parent(Position<E> p) throws IllegalArgumentException;

Iterable<Position<E>> children(Position<E> p) throws IllegalArgumentException;

int numChildren(Position<E> p) throws IllegalArgumentException;

boolean isInternal(Position<E> p) throws IllegalArgumentException;

boolean isExternal(Position<E> p) throws IllegalArgumentException;

boolean isRoot(Position<E> p) throws IllegalArgumentException;

int size();

boolean isEmpty();

Iterator<E> iterator();

Iterable<Position<E>> positions();

}

// 3. كلاس AbstractTree

public abstract class AbstractTree<E> implements Tree<E> {

public boolean isInternal(Position<E> p) { return numChildren(p) > 0; }

public boolean isExternal(Position<E> p) { return numChildren(p) == 0; }

public boolean isRoot(Position<E> p) { return p == root(); }

public int size() {

int count = 0;

for (Position<E> p : positions()) count++;

return count;

}

public boolean isEmpty() { return size() == 0; }

public int depth(Position<E> p) {

if (isRoot(p)) return 0;

else return 1 + depth(parent(p));

}

public int height(Position<E> p) {

int h = 0;

for (Position<E> c : children(p))

h = Math.max(h, 1 + height(c));

return h;

}

private void preorderSubtree(Position<E> p, List<Position<E>> snapshot) {

snapshot.add(p);

for (Position<E> c : children(p))

preorderSubtree(c, snapshot);

}

public Iterable<Position<E>> preorder() {

List<Position<E>> snapshot = new ArrayList<>();

if (!isEmpty())

preorderSubtree(root(), snapshot);

return snapshot;

}

}

// 4. واجهة BinaryTree

public interface BinaryTree<E> extends Tree<E> {

Position<E> left(Position<E> p) throws IllegalArgumentException;

Position<E> right(Position<E> p) throws IllegalArgumentException;

Position<E> sibling(Position<E> p) throws IllegalArgumentException;

}

// 5. كلاس AbstractBinaryTree

public abstract class AbstractBinaryTree<E> extends AbstractTree<E> implements BinaryTree<E> {

public Position<E> sibling(Position<E> p) {

Position<E> parent = parent(p);

if (parent == null) return null;

if (p == left(parent))

return right(parent);

else

return left(parent);

}

public int numChildren(Position<E> p) {

int count = 0;

if (left(p) != null) count++;

if (right(p) != null) count++;

return count;

}

public Iterable<Position<E>> children(Position<E> p) {

List<Position<E>> snapshot = new ArrayList<>(2);

if (left(p) != null) snapshot.add(left(p));

if (right(p) != null) snapshot.add(right(p));

return snapshot;

}

}

// 6. كلاس LinkedBinaryTree

public class LinkedBinaryTree<E> extends AbstractBinaryTree<E> {

protected static class Node<E> implements Position<E> {

private E element;

private Node<E> parent;

private Node<E> left;

private Node<E> right;

public Node(E e, Node<E> above, Node<E> leftChild, Node<E> rightChild) {

element = e;

parent = above;

left = leftChild;

right = rightChild;

}

public E getElement() { return element; }

public Node<E> getParent() { return parent; }

public Node<E> getLeft() { return left; }

public Node<E> getRight() { return right; }

public void setElement(E e) { element = e; }

public void setParent(Node<E> p) { parent = p; }

public void setLeft(Node<E> l) { left = l; }

public void setRight(Node<E> r) { right = r; }

}

protected Node<E> createNode(E e, Node<E> parent, Node<E> left, Node<E> right) {

return new Node<>(e, parent, left, right);

}

protected Node<E> root = null;

private int size = 0;

public LinkedBinaryTree() {}

protected Node<E> validate(Position<E> p) throws IllegalArgumentException {

if (!(p instanceof Node))

throw new IllegalArgumentException("Not valid position type");

Node<E> node = (Node<E>) p;

if (node.getParent() == node)

throw new IllegalArgumentException("p is no longer in the tree");

return node;

}

public int size() { return size; }

public Position<E> root() { return root; }

public Position<E> parent(Position<E> p) throws IllegalArgumentException {

Node<E> node = validate(p);

return node.getParent();

}

public Position<E> left(Position<E> p) throws IllegalArgumentException {

Node<E> node = validate(p);

return node.getLeft();

}

public Position<E> right(Position<E> p) throws IllegalArgumentException {

Node<E> node = validate(p);

return node.getRight();

}

public Position<E> addRoot(E e) throws IllegalStateException {

if (!isEmpty()) throw new IllegalStateException("Tree is not empty");

root = createNode(e, null, null, null);

size = 1;

return root;

}

public Position<E> addLeft(Position<E> p, E e) throws IllegalArgumentException {

Node<E> parent = validate(p);

if (parent.getLeft() != null)

throw new IllegalArgumentException("p already has a left child");

Node<E> child = createNode(e, parent, null, null);

parent.setLeft(child);

size++;

return child;

}

public Position<E> addRight(Position<E> p, E e) throws IllegalArgumentException {

Node<E> parent = validate(p);

if (parent.getRight() != null)

throw new IllegalArgumentException("p already has a right child");

Node<E> child = createNode(e, parent, null, null);

parent.setRight(child);

size++;

return child;

}

public E set(Position<E> p, E e) throws IllegalArgumentException {

Node<E> node = validate(p);

E temp = node.getElement();

node.setElement(e);

return temp;

}

public E remove(Position<E> p) throws IllegalArgumentException {

Node<E> node = validate(p);

if (numChildren(p) == 2)

throw new IllegalArgumentException("p has two children");

Node<E> child = (node.getLeft() != null ? node.getLeft() : node.getRight());

if (child != null)

child.setParent(node.getParent());

if (node == root)

root = child;

else {

Node<E> parent = node.getParent();

if (node == parent.getLeft())

parent.setLeft(child);

else

parent.setRight(child);

}

size--;

E temp = node.getElement();

node.setElement(null);

node.setLeft(null);

node.setRight(null);

node.setParent(node);

return temp;

}

}

## Homework

1. Implement the Array Based Binary Tree Data structure as it is described in chapter 8.

public class ArrayBinaryTree<E> {

private E[] tree;

private int size;

public ArrayBinaryTree(int capacity) {

tree = (E[]) new Object[capacity];

size = 0;

}

public int size() { return size; }

public boolean isEmpty() { return size == 0; }

public Position<E> root() {

if (isEmpty()) return null;

return new ArrayPosition(0);

}

public Position<E> left(Position<E> p) throws IllegalArgumentException {

ArrayPosition pos = validate(p);

int leftIndex = 2 \* pos.index + 1;

if (leftIndex >= tree.length || tree[leftIndex] == null) return null;

return new ArrayPosition(leftIndex);

}

public Position<E> right(Position<E> p) throws IllegalArgumentException {

ArrayPosition pos = validate(p);

int rightIndex = 2 \* pos.index + 2;

if (rightIndex >= tree.length || tree[rightIndex] == null) return null;

return new ArrayPosition(rightIndex);

}

public Position<E> parent(Position<E> p) throws IllegalArgumentException {

ArrayPosition pos = validate(p);

if (pos.index == 0) return null;

return new ArrayPosition((pos.index - 1) / 2);

}

public Position<E> addRoot(E e) throws IllegalStateException {

if (!isEmpty()) throw new IllegalStateException("Tree is not empty");

tree[0] = e;

size = 1;

return new ArrayPosition(0);

}

public Position<E> addLeft(Position<E> p, E e) throws IllegalArgumentException {

ArrayPosition pos = validate(p);

int leftIndex = 2 \* pos.index + 1;

if (leftIndex >= tree.length) throw new IllegalArgumentException("No space left");

if (tree[leftIndex] != null) throw new IllegalArgumentException("p already has a left child");

tree[leftIndex] = e;

size++;

return new ArrayPosition(leftIndex);

}

public Position<E> addRight(Position<E> p, E e) throws IllegalArgumentException {

ArrayPosition pos = validate(p);

int rightIndex = 2 \* pos.index + 2;

if (rightIndex >= tree.length) throw new IllegalArgumentException("No space left");

if (tree[rightIndex] != null) throw new IllegalArgumentException("p already has a right child");

tree[rightIndex] = e;

size++;

return new ArrayPosition(rightIndex);

}

public E set(Position<E> p, E e) throws IllegalArgumentException {

ArrayPosition pos = validate(p);

E temp = tree[pos.index];

tree[pos.index] = e;

return temp;

}

public E remove(Position<E> p) throws IllegalArgumentException {

ArrayPosition pos = validate(p);

if (left(p) != null && right(p) != null)

throw new IllegalArgumentException("p has two children");

E temp = tree[pos.index];

tree[pos.index] = null;

size--;

return temp;

}

protected class ArrayPosition implements Position<E> {

private int index;

public ArrayPosition(int index) {

this.index = index;

}

public E getElement() { return tree[index]; }

}

private ArrayPosition validate(Position<E> p) throws IllegalArgumentException {

if (!(p instanceof ArrayPosition))

throw new IllegalArgumentException("Not valid position type");

ArrayPosition pos = (ArrayPosition) p;

if (pos.index < 0 || pos.index >= tree.length || tree[pos.index] == null)

throw new IllegalArgumentException("Invalid position");

return pos;

}

}