ALP2 Blatt 11

(Mi14h)
Tutor: Can Göktas

Tobias Lohse
Okan Biskin

```java
import java.util.*;
import java.lang.*;

public class BinLinkedTree<T extends Comparable<T>,D>
implements Iterable<T>
{
  // ATTRIBUTES
  private TreeNode root = null;
  private int size = 0;

  class TreeNode {
    // ATTRIBUTES
    D data;
    T key;
    TreeNode left;
    TreeNode right;
    TreeNode parent;

    // CONSTRUCTOR
    TreeNode(T key, D data, TreeNode left, TreeNode right, TreeNode parent) {
      this.data = data;
      this.key = key;
      this.left = left;
      this.right = right;
      this.parent = parent;
    }
    TreeNode(T key, D data, TreeNode parent) {
      this(key, data, null, null, parent);
    }

    // METHODS
    public String toString(){
      return "" + this.key;
    }
  }

  // INSERT, DELTE METHODS
  public boolean empty() {
  /* Return true if tree is empty. */
  // O{1}
    return this.size == 0;
  }
  public void insert(T key, D data) {
  /* Insert new node with given key and data. */
  // O{log(n)}
    TreeNode node = this.root;
    if( this.empty() ){
      this.root = new TreeNode(key, data, null);
    } else {
      while( true ){
        if( key.compareTo(node.key) > 0 ){
          if( node.right != null ){
            node = node.right;
          } else {
            node.right = new TreeNode(key, data, node);
            break;
          }
        } else if( key.compareTo(node.key) < 0 ) {
          if( node.left != null ){
            node = node.left;
          } else {
```

```
            node.left = new TreeNode(key, data, node);
            break;
          }
        } else {
          throw new DuplicateKeyException();
        }
      }
    }
    size++;
  }
  public static class DuplicateKeyException extends RuntimeException {
    public DuplicateKeyException() { super(); }
  }
  public boolean delete(T key) {
  /* Delete the node with the given key and return true, if it does not exists, return
false. */
  // O{(log(n))^2}
    try {
      TreeNode node = this.getNode(key);
      boolean isroot = node.key.equals(this.root.key);
      boolean rightchild = !isroot && node.key.compareTo(node.parent.key) > 0;
      if( node.right == null && node.left == null ){
        if( isroot ){
          this.root = null;
        } else if( rightchild ){
          node.parent.right = null;
        } else {
          node.parent.left = null;
        }
      } else if( node.right != null && node.left == null ){
        if( isroot ){
          this.root = node.right;
        } else if( rightchild ){
          node.parent.right = node.right;
        } else {
          node.parent.left = node.right;
        }
      } else if( node.left != null && node.right == null ){
        if( isroot ){
          this.root = node.left;
        } else if( rightchild ){
          node.parent.right = node.left;
        } else {
          node.parent.left = node.left;
        }
      } else {
        TreeNode successor = this.succ(node);
        this.delete(successor.key);
        this.size++;
        successor.left = node.left;
        successor.right = node.right;
        if( isroot ){
          this.root = successor;
        } else if( rightchild ){
          node.parent.right = successor;
        } else {
          node.parent.left = successor;
        }
      }
      this.size--;
      return true;
    } catch( NoSuchElementException e ){
      return false;
    }
  }
```

```
// GET, MIN, MAX, SUCC, PRED METHODS
public TreeNode getNode(T key) {
/* Return the node with the given key */
// O{log(n)}
  TreeNode node = this.root;
  while( node != null ){
    if( key.compareTo(node.key) > 0 ){
      node = node.right;
    } else if( key.compareTo(node.key) < 0 ) {
      node = node.left;
    } else {
      return node;
    }
  }
  throw new NoSuchElementException();
}
public D getData(T key){
/* Return the data with the given key */
// O{log(n)}
  return this.getNode(key).data;
}
public TreeNode minNode(TreeNode node) {
/* Return the node with the smallest key in the subtree of the given node. */
// O{log(n)}
  while( node.left != null ){
    node = node.left;
  }
  return node;
}
public TreeNode minNode() {
/* Return the node with the smallest key. */
// O{log(n)}
  return this.minNode(this.root);
}
public TreeNode maxNode(TreeNode node) {
/* Return the node with the largest key in the subtree of the given node. */
// O{log(n)}
  while( node.right != null ){
    node = node.right;
  }
  return node;
}
public TreeNode maxNode() {
/* Return the node with the largest key. */
// O{log(n)}
  return this.maxNode(this.root);
}
public TreeNode pred(TreeNode node) {
/* Return the node with the next smaller key of the given node. */
// O{log(n)}
  if( node.left != null ){
    return maxNode(node.left);
  } else {
    TreeNode elder = node.parent;
    while( elder != null && node.key.compareTo(elder.key) < 0 ){
      elder = elder.parent;
    }
    return elder;
  }
}
public TreeNode succ(TreeNode node) {
/* Return the node with the next bigger key of the given node. */
// O{log(n)}
  if( node.right != null ){
    return minNode(node.right);
  } else {
```

```
    TreeNode elder = node.parent;
    while( elder != null && node.key.compareTo(elder.key) > 0 ){
      elder = elder.parent;
    }
    return elder;
  }
}

// DEPTH METHOD
public int deep() {
/* Return depth of tree. */
// O{n*log(n)}
  Stack<TreeNode> stack = new Stack<TreeNode>();
  stack.push(this.root);
  int max = 0;
  while( !stack.isEmpty() ){
    TreeNode node = stack.pop();
    if( node.left != null ) {
      stack.push(node.left);
      max = StrictMath.max(max, depth(node.left.key));
    }
    if( node.right != null ){
      stack.push(node.right);
      max = StrictMath.max(max, depth(node.right.key));
    }
  }
  return max;
}
public int depth(T key){
/* Return depth of tree from given node to the root. */
// O{log(n)}
  int depth = 0;
  TreeNode node = this.getNode(key);
  while( node.parent != null ){
    node = node.parent;
    depth++;
  }
  return depth;
}

// BALANCED METHOD
public boolean perfectBalanced(TreeNode node) {
/* Return true, if subtree of given node is perfectly balanced. */
// O{log(n)}
  int leftSize = getSize(node.left.key);
  int rightSize = getSize(node.right.key);
  if( leftSize > rightSize ){
    return leftSize == rightSize+1;
  } else if( leftSize < rightSize ){
    return leftSize+1 == rightSize;
  } else {
    return true;
  }
}
public boolean perfectBalanced(T key) {
  TreeNode node = this.getNode(key);
  return perfectBalanced(node);
}
public boolean perfectBalanced() {
  return perfectBalanced(this.root);
}
public int getSize(T key) {
/* Return number of nodes in subtree of given node */
// O{log(n)}
  Stack<TreeNode> stack = new Stack<TreeNode>();
  stack.push(this.getNode(key));
```

```
  int size = 0;
  while( !stack.isEmpty() ){
    TreeNode node = stack.pop();
    size++;
    if( node.left != null ){
      stack.push(node.left);
    }
    if( node.right != null ){
      stack.push(node.right);
    }
  }
  return size;
}
public int getSize() {
  return this.size;
}

// STRING, ARRAY METHODS
public D[] toArray() {
/* Return array with the data from all elements of the tree, sorted by the keys. */
// O{n*log(n)}
  D[] array = (D[]) new Object[this.size];
  int i = 0;
  for( T key : this ){
    array[i] = this.getNode(key).data;
    i++;
  }
  return array;
}
public String toStringOrdered() {
// O{n*log(n)}
  String out = "";
  for( T key : this ){
    out += key + "  ";
  }
  return out;
}
public String toString() {
// O{n*log(n)}
  String out = "";
  int depth = this.deep();
  for( int row=0; row<=depth; row++ ){
    String space = getSpaces( 2 * ((int)Math.pow(2,depth-row)-1) );
    out += space;
    for( int i=(int)Math.pow(2,row); i<(int)Math.pow(2,row+1); i++ ){
      try {
        out += this.getNode(i).key + space + space + "  ";
      } catch( NoSuchElementException e ){
        out += " ." + space + space + "  ";
      }
    }
    out += "\n";
  }
  return out;
}
public static String getSpaces(int n){
  String out = "";
  for( int i=0; i<n; i++ ){
    out += " ";
  }
  return out;
}
public static char[] getNodePath(int n) {
  String binString = Integer.toBinaryString(n);
  char[] binChars = binString.substring(1, binString.length()).toCharArray();
  return binChars;
```

```
}
public TreeNode getNode(int n) {
/** Returns node at given position. */
// O{log(n)}
  TreeNode node = this.root;
  for( char c : getNodePath(n) ){
    if( c == '0' && node.left != null ){
      node = node.left;
    } else if( c == '1' && node.right != null ){
      node = node.right;
    } else {
      throw new NoSuchElementException();
    }
  }
  return node;
}

// ITERATOR METHOD
public Iterator<T> iterator() {
  return iteratorIO();
}

// IN ORDER ITERABLE METHODS
public Iterator<T> iteratorIO() {
  return new InOrderIterator();
}
private class InOrderIterator implements Iterator<T> {
  // ATTRIBUTES
  private Stack<TreeNode> stack = new Stack<TreeNode>();

  // CONSTRUCTOR
  InOrderIterator() {
    pushLeftTree(root);
  }

  // METHODS
  public boolean hasNext() {
  // O{1}
    return !stack.isEmpty();
  }
  public T next() {
  // O{log(n)}
    if( !hasNext() ){
      throw new NoSuchElementException();
    }
    TreeNode node = stack.pop();
    pushLeftTree(node.right);
    return node.key;
  }
  private void pushLeftTree(TreeNode node) {
  // O{log(n)}
    while (node != null) {
      stack.push(node);
      node = node.left;
    }
  }
  public void remove() {
    throw new UnsupportedOperationException();
  }
}

// POST ORDER ITERABLE METHODS
public Iterator<T> iteratorPO() {
  return new PostOrderIterator();
}
private class PostOrderIterator implements Iterator<T> {
```

```
    // ATTRIBUTES
    Stack<TreeNode> stack = new Stack<TreeNode>();
    Stack<Boolean> rightChild = new Stack<Boolean>();

    // CONSTRUCTOR
    PostOrderIterator() {
      pushLeftTree(root);
    }

    // METHODS
    public boolean hasNext() {
    // O{1}
      return !stack.isEmpty();
    }
    public T next() {
    // O{log(n)}
      if( stack.peek().right == null || rightChild.peek() ){
        rightChild.pop();
        return stack.pop().key;
      } else {
        rightChild.pop();
        rightChild.push(true);
        pushLeftTree(stack.peek().right);
        return next();
      }
    }
    public void remove() {
      throw new UnsupportedOperationException();
    }
    private void pushLeftTree(TreeNode node) {
    // O{log(n)}
      if (node != null) {
        stack.push(node);
        rightChild.push(false);
        pushLeftTree(node.left);
      }
    }
  }

}


public class TestBinLinkedTree {
  public static class Num implements Comparable<Num> {
    int num;
    public Num(int num) {
      if( num < 100 ){
        this.num = num;
      } else {
        throw new RuntimeException("Num just excepts Elements < 100");
      }
    }
    public int compareTo(Num other) {
      if( this.num < other.num ){
        return -1;
      } else if( this.num > other.num ){
        return 1;
      } else {
        return 0;
      }
    }
    public String toString() {
      if( this.num < 10 ){
        return "0" + this.num;
      } else {
        return "" + this.num;
```

```
    }
  }
}

public static void print(Object o) {
  System.out.println(o);
}
public static BinLinkedTree<Num,String> newTree(int[] ints) {
  BinLinkedTree<Num,String> tree = new BinLinkedTree<Num,String>();
  for( int i : ints ) {
    tree.insert(new Num(i), "a");
  }
  return tree;
}

public static void main(String[] args) {

  // GET, MIN, MAX, SUCC, PRED
  int[] ints1 = {6,4,10,2,5,8,9,7,1,3,12,11};
  BinLinkedTree<Num,String> tree1 = newTree(ints1);
  print(tree1);
  print("depth: " + tree1.deep());
  print("node 06: " + tree1.getNode(new Num(6)));
  print("node 10: " + tree1.getNode(new Num(10)));
  print("");
  print("min: " + tree1.minNode());
  print("min 10: " + tree1.minNode(tree1.getNode(new Num(10))));
  print("max: " + tree1.maxNode());
  print("max 04: " + tree1.maxNode(tree1.getNode(new Num(4))));
  print("");
  print("pred 06: " + tree1.pred(tree1.getNode(new Num(6))));
  print("succ 06: " + tree1.succ(tree1.getNode(new Num(6))));
  print("pred 08: " + tree1.pred(tree1.getNode(new Num(8))));
  print("succ 09: " + tree1.succ(tree1.getNode(new Num(9))));
  print("pred 07: " + tree1.pred(tree1.getNode(new Num(7))));
  print("succ 05: " + tree1.succ(tree1.getNode(new Num(5))));
  print("pred 01: " + tree1.pred (tree1.getNode(new Num(01))));
  print("succ 12: " + tree1.succ(tree1.getNode(new Num(12))));

  // DELETING
  int[] ints2 = {53,27,69,13,34,63,95,8,17,30,46,66,5,9,15,18,32,
                 50,29,68,71,64,98,99,97};
  BinLinkedTree<Num,String> tree2 = newTree(ints2);
  print(tree2);
  print("delete 15 =>");
  tree2.delete(new Num(15));
  print(tree2);
  print("delete 46 =>");
  tree2.delete(new Num(46));
  print(tree2);
  print("delete 27 =>");
  tree2.delete(new Num(27));
  print(tree2);
  print("delete 53 =>");
  tree2.delete(new Num(53));
  print(tree2);
  print("");

  // KOMMUTATIVE DELETING
  print("Ist komutativ, da der Knoten durch seinen Nachfolger \nersetzt wird
        und die Suche nach Nachfolgern Kommutativ \nist.\n");
  BinLinkedTree<Num,String> tree2a = newTree(ints2);
  BinLinkedTree<Num,String> tree2b = newTree(ints2);
  print(tree2a);
  print("delete 53 & 63 =>");
  tree2a.delete(new Num(53));
```

```
    tree2a.delete(new Num(63));
    print(tree2a);
    print("delete 63 & 53 =>");
    tree2b.delete(new Num(63));
    tree2b.delete(new Num(53));
    print(tree2b);
    print("");

    // BALANCED DELETING
    print("Tree size:");
    print(tree2b.getSize());
    print(tree2b.getSize(new Num(64)));
    print("13 size:");
    print(tree2b.getSize(new Num(13)));
    print("");
    print("Tree perfectly balanced: " + tree2b.perfectBalanced());
    print("perfectly balanced at 13: " + tree2b.perfectBalanced(new Num(13)));
    print("perfectly balanced at 27: " + tree2b.perfectBalanced(new Num(27)));
    print("perfectly balanced at 95: " + tree2b.perfectBalanced(new Num(95)));
    print("");

    // ARRAY AND POST ORDER ITERATION
    print(tree1);
    print("Element in order:");
    String elementsIO = "";
    for( Iterator<Num> iter = tree1.iteratorIO(); iter.hasNext(); ){
      elementsIO += iter.next() + "  ";
    }
    print(elementsIO);
    print("");
    // print("Data in order:");
    // String dataString = "";
    // for( String data : tree1.toArray() ){
    //  dataString += data + "   ";
    // }
    // print(dataString);
    print("Java can not cast `(String[]) Object[n]`, so no tests for `toArray` here.");
    print("");
    print("Element post order:");
    String elementsPO = "";
    for( Iterator<Num> iter = tree1.iteratorPO(); iter.hasNext(); ){
      elementsPO += iter.next() + "  ";
    }
    print(elementsPO);
  }
}
```