자료구조

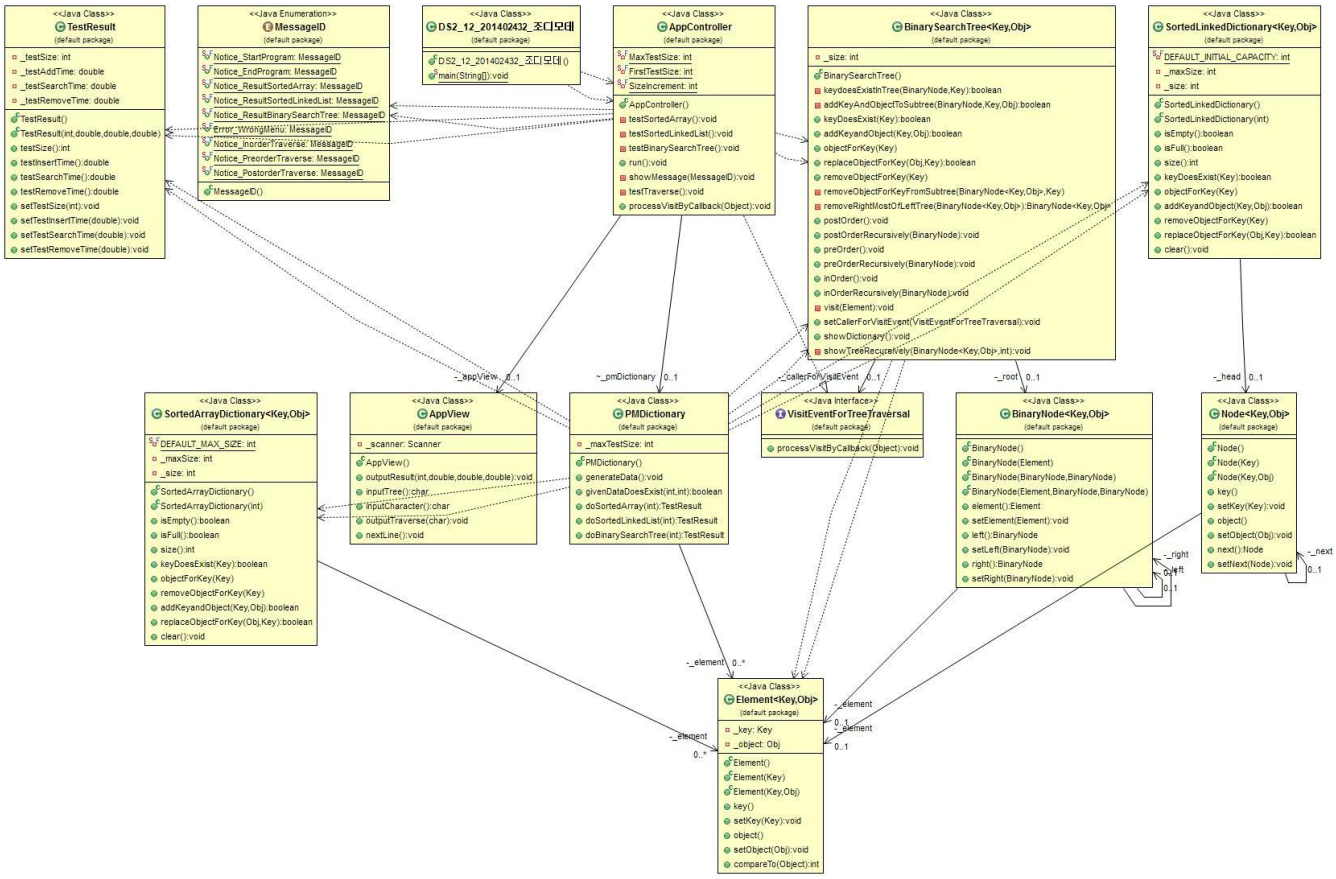
실습 보고서

[제 12주] 사전 – 성능분석

제출일 : 2015.12.1

201402432 / 조디모데

1.프로그램설명서

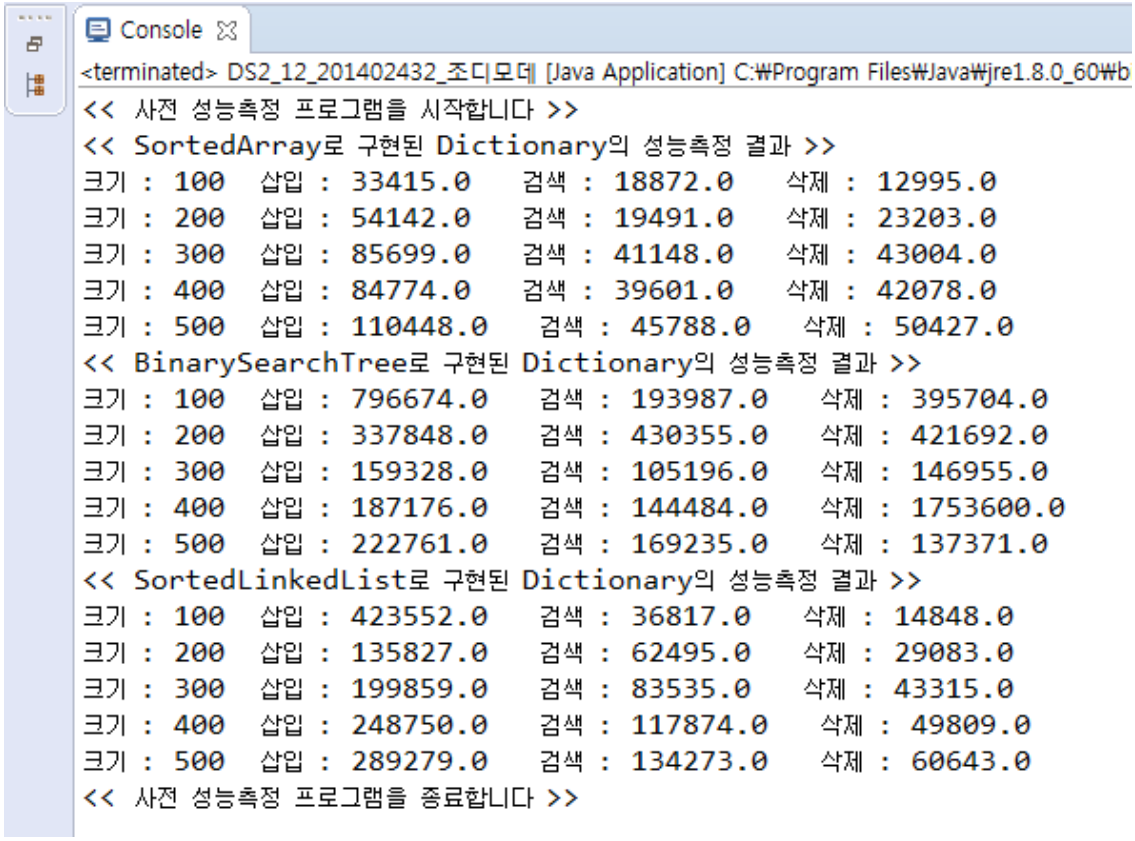


자료 구조 & 알고리즘 : ArryList, LinkedList, BinarySearchTree, Hash

2.실행 결과 분석

1.입력과출력

- 프로그램 실행 화면



3.소스 코드

<main>

```
public class DS2_12_201402432_조디모데 {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		AppController appController = new AppController() ;
		appController.run();


	}

}
```

<AppControllor>

```
public class AppController implements VisitEventForTreeTraversal {
	private AppView _appView;
	PMDictionary _pmDictionary;
	private static final int MaxTestSize = 500;
	private static final int FirstTestSize = 100;
	private static final int SizeIncrement = 100;

	public AppController() {
		this._appView = new AppView();
	}

	private void testSortedArray() {
		this.showMessage(MessageID.Notice_ResultSortedArray);
		for (int testSize = FirstTestSize; testSize <=
MaxTestSize; testSize += SizeIncrement) {
			TestResult testResult =
this._pmDictionary.doSortedArray(testSize);
			this._appView.outputResult(testResult.testSize(),
					testResult.testInsertTime(),
testResult.testSearchTime(),
					testResult.testRemoveTime());
		}
	}

	private void testSortedLinkedList() {

	this.showMessage(MessageID.Notice_ResultSortedLinkedList);
```

```
            for (int testSize = FirstTestSize; testSize <=
MaxTestSize; testSize += SizeIncrement) {
                  TestResult testResult = this._pmDictionary
                              .doSortedLinkedList(testSize);
                  this._appView.outputResult(testResult.testSize(),
                              testResult.testInsertTime(),
testResult.testSearchTime(),
                              testResult.testRemoveTime());
            }
      }

      private void testBinarySearchTree() {

      this.showMessage(MessageID.Notice_ResultBinarySearchTree);
            for (int testSize = FirstTestSize; testSize <=
MaxTestSize; testSize += SizeIncrement) {
                  TestResult testResult = this._pmDictionary
                              .doBinarySearchTree(testSize);
                  this._appView.outputResult(testResult.testSize(),
                              testResult.testInsertTime(),
testResult.testSearchTime(),
                              testResult.testRemoveTime());
            }
      }

      public void run() {
            this.showMessage(MessageID.Notice_StartProgram);
            this._pmDictionary = new PMDictionary();
            this._pmDictionary.generateData();
            this.testSortedArray();
            this.testBinarySearchTree();
            this.testSortedLinkedList();

            this.showMessage(MessageID.Notice_EndProgram);

            System.out.println();
            this.testTraverse();

      }

      private void showMessage(MessageID aMessageID) {
            switch (aMessageID) {
            case Notice_ResultSortedArray:
                  System.out.println("<< SortedArray로 구현된
Dictionary의 성능측정 결과 >>");
                  break;
            case Notice_StartProgram:
```

```
                System.out.println("<< 사전 성능측정 프로그램을
시작합니다 >>");
                break;
            case Notice_EndProgram:
                System.out.println("<< 사전 성능측정 프로그램을
종료합니다 >>");
                break;
            case Notice_ResultSortedLinkedList:
                System.out
                .println("<< SortedLinkedList로 구현된 Dictionary의
성능측정 결과 >>");
                break;
            case Notice_ResultBinarySearchTree:
                System.out
                .println("<< BinarySearchTree로 구현된 Dictionary의
성능측정 결과 >>");
                break;
            case Notice_InorderTraverse:
                System.out.print(">> INORDER TRAVERSE : ");
                break;
            case Notice_PreorderTraverse:
                System.out.print(">> PREORDER TRAVERSE : ");
                break;
            case Notice_PostorderTraverse:
                System.out.print(">> POSTORDER TRAVERSE : ");
                break;
            default:
                break;

            }
        }

        private void testTraverse() {
            BinarySearchTree<Integer, Character> binaryTree = new
BinarySearchTree<Integer, Character>();
            char value = 'A';
            int[] input = new int[] { 50, 30, 70, 10, 40, 60, 80, 0,
20 };
            for (int i = 0; i < input.length; i++)
                binaryTree.addKeyandObject(input[i], value++);

            this._appView.nextLine();
            binaryTree.showDictionary();
```

```
            binaryTree.setCallerForVisitEvent(this);
            this._appView.nextLine();

            this.showMessage(MessageID.Notice_InorderTraverse);
            binaryTree.inOrder();
            this._appView.nextLine();

            this.showMessage(MessageID.Notice_PreorderTraverse);
            binaryTree.preOrder();
            this._appView.nextLine();

            this.showMessage(MessageID.Notice_PostorderTraverse);
            binaryTree.postOrder();
            this._appView.nextLine();
      }
      @Override
      public void processVisitByCallback(Object anObj) {
            this._appView.outputTraverse((Character) anObj);
      }
}
```

<AppView>

```
import java.util.*;

public class AppView {
	private Scanner _scanner;

	public AppView() {
		this._scanner = new Scanner(System.in);
	}

	public void outputResult(int aTestSize, double aTestInsertTime,
			double aTestSearchTime, double aTestRemoveTime) {
		System.out.println("크기 : " + aTestSize + "  삽입 : " +
aTestInsertTime
				+ "   검색 : " + aTestSearchTime + "   삭제 :
" + aTestRemoveTime);
	}

	public char inputTree() {
		return this._scanner.nextLine().charAt(0);
	}

	public char inputCharacter() {
		return this._scanner.nextLine().charAt(0);
	}

	public void outputTraverse(char anObj) {
		System.out.println(anObj);
	}

	public void nextLine() {
		System.out.println();
	}
}
```

<PMDictionary>

```
import java.util.Random;

public class PMDictionary {
	private int _maxTestSize;
	private Element<Integer, Integer>[] _element;

	public PMDictionary() {
		this._maxTestSize = 500;
		this._element = new Element[this._maxTestSize];
	}

	public void generateData() {
		Random random = new Random();
		int currentSize = 0;
		while (currentSize < this._maxTestSize) {
			int newData = random.nextInt(this._maxTestSize);
		       if (!this.givenDataDoesExist(newData, currentSize))
{
				Element<Integer, Integer> newElement = new
Element<Integer, Integer>(
					newData, currentSize);
				this._element[currentSize] = newElement;
				currentSize++;
			}
		}
	}

	public boolean givenDataDoesExist(int newData, int
currentDataSize) {
		for (int i = 0; i < currentDataSize; i++)
			if (this._element[i].key() == newData)
				return true;

		return false;
	}

	public TestResult doSortedArray(int testSize) {
		SortedArrayDictionary<Integer, Integer> dic = new
SortedArrayDictionary<Integer, Integer>();
		long timeForAdd, timeForSearch, timeForRemove;
		long start, stop;

		timeForAdd = 0;
		for (int testCount = 0; testCount < testSize;
testCount++) {
			start = System.nanoTime();
```

```
                dic.addKeyandObject(this._element[testCount].key(),
                            this._element[testCount].object());
                stop = System.nanoTime();
                timeForAdd += (stop - start);
            }
            timeForSearch = 0;
            for (int testCount = 0; testCount < testSize;
testCount++) {
                Integer searchObj;
                start = System.nanoTime();
                searchObj =
dic.objectForKey(this._element[testCount].key());
                stop = System.nanoTime();
                timeForSearch += (stop - start);
            }
            timeForRemove = 0;
            for (int testCount = 0; testCount < testSize;
testCount++) {
                Integer removedObj;
                start = System.nanoTime();
                removedObj =
dic.removeObjectForKey(this._element[testCount].key());
                stop = System.nanoTime();
                timeForRemove += (stop - start);
            }
            return new TestResult(testSize, timeForAdd,
timeForSearch,
                        timeForRemove);
      }

      public TestResult doSortedLinkedList(int testSize) {
            SortedLinkedDictionary<Integer, Integer> dic = new
SortedLinkedDictionary<Integer, Integer>();
            long timeForAdd, timeForSearch, timeForRemove;
            long start, stop;

            timeForAdd = 0;
            for (int testCount = 0; testCount < testSize;
testCount++) {
                start = System.nanoTime();
                dic.addKeyandObject(this._element[testCount].key(),
                            this._element[testCount].object());

                stop = System.nanoTime();
                timeForAdd += (stop - start);
            }
            timeForSearch = 0;
            for (int testCount = 0; testCount < testSize;
```

```
testCount++) {
                  Integer searchObj;
                  start = System.nanoTime();
                  searchObj =
dic.objectForKey(this._element[testCount].key());
                  stop = System.nanoTime();
                  timeForSearch += (stop - start);
            }
            timeForRemove = 0;
            for (int testCount = 0; testCount < testSize;
testCount++) {
                  Integer removedObj;
                  start = System.nanoTime();
                  removedObj =
dic.removeObjectForKey(this._element[testCount].key());
                  stop = System.nanoTime();
                  timeForRemove += (stop - start);
            }
            return new TestResult(testSize, timeForAdd,
timeForSearch,
                        timeForRemove);
      }

      public TestResult doBinarySearchTree(int testSize) {
            BinarySearchTree<Integer, Integer> dic = new
BinarySearchTree<Integer, Integer>();
            double timeForAdd, timeForSearch, timeForRemove;
            long start, stop;

            timeForAdd = 0;
            for (int testCount = 0; testCount < testSize;
testCount++) {
                  start = System.nanoTime();
                 dic.addKeyandObject(this._element[testCount].key(),
                              this._element[testCount].object());
                  stop = System.nanoTime();
                  timeForAdd += (double) (stop - start);
            }
            timeForSearch = 0;
            for (int testCount = 0; testCount < testSize;
testCount++) {
                  Integer searchObj;
                  start = System.nanoTime();
                  searchObj =
dic.objectForKey(this._element[testCount].key());
                  stop = System.nanoTime();
                  timeForSearch += (double) (stop - start);
            }
```

```
            timeForRemove = 0;
            for (int testCount = 0; testCount < testSize;
testCount++) {
                  Integer removedObj;
                  start = System.nanoTime();
                  removedObj =
dic.removeObjectForKey(this._element[testCount].key());
                  stop = System.nanoTime();
                  timeForRemove += (double) (stop - start);
            }
            return new TestResult(testSize, timeForAdd,
timeForSearch,
                        timeForRemove);
      }
}
```

<MessageID>

```
public enum MessageID {
      Notice_StartProgram,
      Notice_EndProgram,
      Notice_ResultSortedArray,
      Notice_ResultSortedLinkedList,
      Notice_ResultBinarySearchTree,

      Error_WrongMenu,

      Notice_InorderTraverse,
      Notice_PreorderTraverse,
      Notice_PostorderTraverse,

}
```

<SortedLinkedDictionary>

```
public class SortedLinkedDictionary<Key extends Comparable, Obj> {
	private static final int DEFAULT_INITIAL_CAPACITY = 20;
	private int _maxSize;
	private int _size;
	private Node<Key, Obj> _head;

	public SortedLinkedDictionary() {
		this._maxSize = this.DEFAULT_INITIAL_CAPACITY;
		this._size = 0;
		this._head = null;
	}

	public SortedLinkedDictionary(int aMaxsize) {
		this._maxSize = aMaxsize;
		this._size = 0;
		this._head = null;
	}

	public boolean isEmpty() {
		return this._size == 0;
	}

	public boolean isFull() {
		return this._maxSize == this._size;
	}

	public int size() {
		return this._size;
	}

	public boolean keyDoesExist(Key aKey) {
		Node exNode = this._head;
		while (exNode != null) {
			if (exNode.key() == aKey)
				return true;
			exNode = exNode.next();
		}
		return false;
	}

	public Obj objectForKey(Key aKey) {
		Node exNode = this._head;
		Obj findObj = this._head.object();
		while (exNode != null) {
			if (exNode.key() == aKey)
				return findObj;
```

```
                exNode = exNode.next();
            }
            return null;
        }

        public boolean addKeyandObject(Key aKey, Obj anObject) {
            if (!this.keyDoesExist(aKey)) {
                Node newNode = new Node(aKey, anObject);
                Node previousNode = null;
                Node currentNode = this._head;
                if (!isEmpty()) {
                    while (currentNode != null) {
                        if (currentNode.key().compareTo(aKey)
> 0) {
                            previousNode.setNext(newNode);
                            newNode.setNext(currentNode);
                            return true;
                        }
                        previousNode = currentNode;
                        currentNode = currentNode.next();
                    }
                } else {
                    this._head = newNode;
                    return true;
                }
            }
            return false;
        }

        public Obj removeObjectForKey(Key aKey) {
            if (this.isEmpty())
                return null;
            else {
                Node previousNode = null;
                Node currentNode = this._head;
                Obj removeObj = this._head.object();

                while (currentNode != null) {
                    if (currentNode.key() == aKey) {
                        removeObj = (Obj) currentNode.object();
                        previousNode.setNext(currentNode);
                        return removeObj;
                    }
                    previousNode = currentNode;
                    currentNode = currentNode.next();
                }

                return null;
```

```
		}
	}

	public boolean replaceObjectForKey(Obj newObject, Key aKey) {
		if (!this.keyDoesExist(aKey))
			return false;
		else {
			Node exNode = this._head;
			while (exNode != null) {
				if (exNode.key() == aKey) {
					exNode.setObject(newObject);
					return true;
				}
				exNode = exNode.next();
			}
			return false;
		}
	}

	public void clear() {
		this._size = 0;
		this._head = null;
	}

}
```

<BinaryNode>

```
public class BinaryNode<Key extends Comparable, Obj> {
	private Element<Key, Obj> _element;
	private BinaryNode<Key, Obj> _left;
	private BinaryNode<Key, Obj> _right;

	public BinaryNode() {
		this._element = null;
		this._left = null;
		this._right = null;
	}

	public BinaryNode(Element anElement) {
		this._element = anElement;
		this._left = null;
		this._right = null;
	}
	public BinaryNode(BinaryNode aLeft, BinaryNode aRight) {
		this._element = null;
		this._left = aLeft;
		this._right = aRight;
	}
	public BinaryNode(Element anElement, BinaryNode aLeft,
BinaryNode aRight) {
		this._element = anElement;
		this._left = aLeft;
		this._right = aRight;
	}
	public Element element() {
		return this._element;
	}
	public void setElement(Element anElement) {
		this._element = anElement;
	}
	public BinaryNode left() {
		return this._left;
	}
	public void setLeft(BinaryNode aLeft) {
		this._left = aLeft;
	}
	public BinaryNode right() {
		return this._right;
	}
	public void setRight(BinaryNode aRight) {
		this._right = aRight;
	}
```

<BinarySearchTree>

```
public class BinarySearchTree<Key extends Comparable, Obj> {
	private BinaryNode<Key, Obj> _root;
	private int _size;
	private VisitEventForTreeTraversal _callerForVisitEvent;

	public BinarySearchTree() {
		this._root = null;
		this._size = 0;
	}

	private boolean keydoesExistInTree(BinaryNode currentRoot, Key
aKey) {
		if (currentRoot == null)
			return false;
		else {
			if (currentRoot.element().key() == aKey)
				return true;
			else if
(currentRoot.element().key().compareTo(aKey) > 0)
				return keydoesExistInTree(currentRoot.left(),
aKey);
			else
				return
keydoesExistInTree(currentRoot.right(), aKey);

		}
	}

	private boolean addKeyAndObjectToSubtree(BinaryNode
currentRoot, Key aKey,
			Obj anObject) {
		BinaryNode newNode = null;
		if (currentRoot.element().key().compareTo(aKey) == 0)
			return false;
		else if (currentRoot.element().key().compareTo(aKey) > 0)
{
			if (currentRoot.left() == null) {
				newNode = new BinaryNode(new Element(aKey,
anObject), null,
						null);
				currentRoot.setLeft(newNode);
				this._size++;
				return true;
			} else
				return
addKeyAndObjectToSubtree(currentRoot.left(), aKey,
```

```
                                        anObject);
                } else {
                        if (currentRoot.right() == null) {
                                newNode = new BinaryNode(new Element(aKey,
anObject), null,
                                        null);
                                currentRoot.setRight(newNode);
                                this._size++;
                                return true;
                        } else
                                return
addKeyAndObjectToSubtree(currentRoot.right(), aKey,
                                        anObject);
                }
        }

        public boolean keyDoesExist(Key aKey) {
                return this.keydoesExistInTree(this._root, aKey);
        }

        public boolean addKeyandObject(Key aKey, Obj anObject) {
                if (this._root == null) {
                        this._root = new BinaryNode(new Element(aKey,
anObject), null, null);
                        this._size++;
                        return true;
                } else
                        return addKeyAndObjectToSubtree(this._root, aKey,
anObject);
        }

        public Obj objectForKey(Key aKey) {
                boolean found = false;
                BinaryNode currentRoot = this._root;
                while ((!found) && (currentRoot != null)) {
                        if (currentRoot.element().key().compareTo(aKey) ==
0)
                                found = true;
                        else if
(currentRoot.element().key().compareTo(aKey) > 0)
                                currentRoot = currentRoot.left();
                        else
                                currentRoot = currentRoot.right();
                }
                if (found)
                        return (Obj) currentRoot.element().object();
                else
                        return null;
```

```
	}

	public boolean replaceObjectForKey(Obj newObject, Key aKey) {
		boolean found = false;
		BinaryNode currentRoot = this._root;
		while ((!found) && (currentRoot != null)) {
			if (currentRoot.element().key().compareTo(aKey) >
0)
				currentRoot = currentRoot.left();
			else if
(currentRoot.element().key().compareTo(aKey) < 0)
				currentRoot = currentRoot.right();
			else
				found = true;
		}
		if (found) {
			currentRoot.element().setObject(newObject);
			return true;
		} else
			return false;
	}

	public Obj removeObjectForKey(Key aKey) {
		Obj removedObject = null;
		if (this._root == null)
			return null;
		else if (this._root.element().key().compareTo(aKey) == 0)
{
			removedObject = (Obj)
this._root.element().object();
			if ((this._root.left() == null) &&
(this._root.right() == null))
				this._root = null;
			else if (this._root.left() == null)
				this._root = this._root.right();
			else if (this._root.right() == null)
				this._root = this._root.left();
			else {
				BinaryNode<Key, Obj> newRoot =
removeRightMostOfLeftTree(this._root);
				newRoot.setLeft(this._root.left());
				newRoot.setRight(this._root.right());
				this._root = newRoot;
			}
			this._size--;
			return removedObject;
		} else {
			return removeObjectForKeyFromSubtree(this._root,
```

```
aKey);
            }
      }

      private Obj removeObjectForKeyFromSubtree(BinaryNode<Key, Obj>
currentRoot,
                  Key aKey) {
            if (currentRoot.element().key().compareTo(aKey) > 0) {
                  BinaryNode<Key, Obj> child = currentRoot.left();
                  if (child == null)
                        return null;
                  else {
                        if (child.element().key().compareTo(aKey) ==
0) {
                              Obj removedObject = (Obj)
child.element().object();
                              if (child.left() == null &&
child.right() == null)
                                    currentRoot.setLeft(null);
                              else if (child.left() == null)

      currentRoot.setLeft(child.right());
                              else if (child.right() == null)

      currentRoot.setLeft(child.left());
                              else {
                                    BinaryNode<Key, Obj> newChild =
removeRightMostOfLeftTree(child);
                                    newChild.setLeft(child.left());
                                   newChild.setRight(child.right());
                                    currentRoot.setLeft(newChild);
                              }
                              this._size--;
                              return removedObject;
                        } else {
                              return
removeObjectForKeyFromSubtree(child, aKey);
                        }
                  }
            } else {
                  BinaryNode<Key, Obj> child = currentRoot.right();
                  if (child == null)
                        return null;
                  else {
                        if (child.element().key().compareTo(aKey) ==
0) {
                              Obj removedObject = (Obj)
child.element().object();
```

```
                                    if (child.left() == null &&
child.right() == null)
                                          currentRoot.setRight(null);
                                    else if (child.left() == null)

      currentRoot.setRight(child.left());
                                    else if (child.right() == null)

      currentRoot.setRight(child.left());
                                    else {
                                          BinaryNode<Key, Obj> newChild =
removeRightMostOfLeftTree(child);
                                          newChild.setLeft(child.left());
                                         newChild.setRight(child.right());
                                          currentRoot.setLeft(newChild);
                                    }
                                    this._size--;
                                    return removedObject;
                              } else
                                    return
removeObjectForKeyFromSubtree(child, aKey);

                        }
                  }
      }

      private BinaryNode<Key, Obj> removeRightMostOfLeftTree(
                  BinaryNode<Key, Obj> currentRoot) {

            BinaryNode<Key, Obj> leftOfCurrentRoot =
currentRoot.left();
            if (leftOfCurrentRoot == null)
                  return null;
            if (leftOfCurrentRoot.right() == null) {
                  currentRoot.setLeft(leftOfCurrentRoot.left());
                  return leftOfCurrentRoot;
            } else {
                  BinaryNode<Key, Obj> parentOfRightMost =
leftOfCurrentRoot;
                  BinaryNode<Key, Obj> rightMost =
leftOfCurrentRoot.right();
                  while (rightMost.right() != null) {
                        parentOfRightMost = rightMost;
                        rightMost = rightMost.right();
                  }
                  parentOfRightMost.setRight(rightMost.left());
                  rightMost.setLeft(null);
                  return rightMost;
```

```
		}
	}

	public void postOrder() {
		this.postOrderRecursively(this._root);
	}

	public void postOrderRecursively(BinaryNode aRoot) {
		if (aRoot != null) {
			this.postOrderRecursively(aRoot.left());
			this.postOrderRecursively(aRoot.right());
			this.visit(aRoot.element());
		} else
			return;
	}

	public void preOrder() {
		this.preOrderRecursively(this._root);
	}

	public void preOrderRecursively(BinaryNode aRoot) {
		if (aRoot != null) {
			this.visit(aRoot.element());
			this.preOrderRecursively(aRoot.left());
			this.preOrderRecursively(aRoot.right());
		}
	}

	public void inOrder() {
		this.inOrderRecursively(this._root);
	}

	public void inOrderRecursively(BinaryNode aRoot) {
		if (aRoot != null) {
			this.inOrderRecursively(aRoot.left());
			this.visit(aRoot.element());
			this.inOrderRecursively(aRoot.right());
		} else
			return;
	}

	private void visit(Element anElement) {
		System.out.print((Character) anElement.object() + "-");
	}

	public void setCallerForVisitEvent(VisitEventForTreeTraversal
aCaller) {
		this._callerForVisitEvent = aCaller;
```

```
	}

	public void showDictionary() {
		this.showTreeRecursively(this._root, 1);
	}

	private void showTreeRecursively(BinaryNode<Key, Obj>
currentNode, int depth) {
		if (currentNode != null) {
			showTreeRecursively(currentNode.right(), depth +
1);
			for (int i = 0; i < depth - 1; i++)
				System.out.print("\t");
			System.out.println("(" +
currentNode.element().key() + ", "
					+ (Character)
currentNode.element().object() + ")");
			showTreeRecursively(currentNode.left(), depth + 1);
		}
	}
}
```

<Element>

```
public class Element<Key extends Comparable, Obj> implements
Comparable {

	private Key _key;
	private Obj _object;

	public Element() {
		this._key = null;
		this._object = null;
	}

	public Element(Key aKey) {
		this._key = aKey;
		this._object = null;
	}

	public Element(Key aKey, Obj anObject) {
		this._key = aKey;
		this._object = anObject;
	}

	public Key key() {
		return this._key;
	}

	public void setKey(Key aKey) {
		this._key = aKey;
	}

	public Obj object() {
		return this._object;
	}

	public void setObject(Obj anObject) {
		this._object = anObject;
	}

	@Override
	public int compareTo(Object arg0) {
		return this._key.compareTo(arg0);
	}

}
```

<Node>

```
public class Node<Key extends Comparable, Obj> {
	private Element<Key, Obj> _element;
	private Node<Key, Obj> _next;

	public Node() {
		this._element = null;
		this._next = null;
	}

	public Node(Key aKey) {
		this._element = new Element(aKey);
		this._next = null;
	}

	public Node(Key aKey, Obj anObject) {
		this._element = new Element(aKey);
		this._element.setObject(anObject);
		this._next = null;
	}

	public Key key() {
		return this._element.key();
	}

	public void setKey(Key aKey) {
		this._element.setKey(aKey);
	}

	public Obj object() {
		return this._element.object();
	}

	public void setObject(Obj anObject) {
		this._element.setObject(anObject);
	}

	public Node next() {
		return this._next;
	}

	public void setNext(Node aNode) {
		this._next = aNode;
	}
}
```

<SortedArrayDictionary>

```
public class SortedArrayDictionary<Key extends Comparable, Obj> {
	private static final int DEFAULT_MAX_SIZE = 20;
	private int _maxSize;
	private int _size;
	private Element<Key, Obj>[] _element;

	public SortedArrayDictionary() {
		this._maxSize = this.DEFAULT_MAX_SIZE;
		this._size = 0;
		this._element = new Element[this._maxSize];
	}

	public SortedArrayDictionary(int aMaxSize) {
		this._maxSize = aMaxSize;
		this._size = 0;
		this._element = new Element[this._maxSize];
	}

	public boolean isEmpty() {
		return this._size == 0;
	}

	public boolean isFull() {
		return this._size == this._maxSize;
	}

	public int size() {
		return this._size;
	}

	public boolean keyDoesExist(Key aKey) {
		if (this.isEmpty())
			return false;
		else {
			for (int i = 0; i < this._size; i++)
				if (this._element[i].key() == aKey)
					return true;
		}
		return false;
	}

	public Obj objectForKey(Key aKey) {
		for (int i = 0; i < this._size; i++)
			if (this._element[i].key() == aKey)
				return this._element[i].object();
		return null;
```

```
	}

	public Obj removeObjectForKey(Key aKey) {
		Element removedElement = null;
		for (int i = 0; i < this._size; i++)
			if (this._element[i].key() == aKey) {

	removedElement.setObject(this._element[i].object());

				for (int j = i; j < this._size; j++)
					this._element[j] = this._element[j +
1];
				this._element[this._size - 1] = null;
				this._size--;
				break;
			}
		return (Obj) removedElement;
	}

	public boolean addKeyandObject(Key aKey, Obj anObject) {
		if (!this.keyDoesExist(aKey)) {
			for (int i = 0; i < this._size; i++) {
				if (this._element[i].compareTo(aKey) > 0) {
					for (int j = this._size; j > i; j--) {
						this._element[j] =
this._element[j - 1];
					}
					this._element[i].setObject(anObject);
					return true;
				}
			}
		}
		return false;
	}
	public boolean replaceObjectForKey(Obj newObject, Key aKey) {
		for (int i = 0; i < this._size; i++)
			if (this._element[i].key() == aKey) {
				this._element[i].setObject(newObject);
				return true;
			}
		return false;
	}
	public void clear() {
		this._size = 0;
		this._element = null;
	}
}
```

<TestResult>

```
public class TestResult {
	private int _testSize;
	private double _testAddTime;
	private double _testSearchTime;
	private double _testRemoveTime;

	public TestResult() {
		this._testAddTime = 0;
		this._testRemoveTime = 0;
		this._testSearchTime = 0;
		this._testSize = 0;
	}

	public TestResult(int testSize, double insertTime, double
searchTime,
			double removeTime) {
		this._testSize = testSize;
		this._testAddTime = insertTime;
		this._testRemoveTime = removeTime;
		this._testSearchTime = searchTime;
	}

	public int testSize() {
		return this._testSize;
	}

	public double testInsertTime() {
		return this._testAddTime;
	}

	public double testSearchTime() {
		return this._testSearchTime;
	}

	public double testRemoveTime() {
		return this._testRemoveTime;
	}

	public void setTestSize(int aTestSize) {
		this._testSize = aTestSize;
	}

	public void setTestInsertTime(double aTestInsertTime) {
		this._testAddTime = aTestInsertTime;

	}
```

```
	public void setTestSearchTime(double aTestSearchTime) {
		this._testSearchTime = aTestSearchTime;
	}

	public void setTestRemoveTime(double aTestRemoveTime) {
		this._testRemoveTime = aTestRemoveTime;
	}
}
```

<VisitEventForTreeTraversal>

```
public interface VisitEventForTreeTraversal {
	public void processVisitByCallback(Object anObj);
}
```