자료구조 실습 보고서

[제14주] 이진검색트리 사전에서의

삽입, 삭제, Call Back

2021년 06월 08일

201702039 오명주

1. 프로그램 설명서

(1) 프로그램의 전체 설계 구조

➔ MVC (Model – View – Controller) 구조

Model : 프로그램이 "무엇"을 할 것인지 정의. 사용자의 요청에 맞는 알고리즘을 처리하고 DB와 상호작용하여 결과물을 산출하고 Controller에게 전달.

View : 화면에 무엇인가를 "보여주기 위한" 역할. 최종 사용자에게 "무엇"을 화면으로 보여줌.

Controller : 모델이 "어떻게" 처리할 지 알려주는 역할. 사용자로부터 입력을 받고 중개인 역할. Model과 View는 서로 직접 주고받을 수 없음. Controller을 통해 이야기함.

➔ 정렬 결과 검증 프로그램에서의 각 클래스 별 MVC 구조 역할

Model:

- DictionaryElement<Key, Obj> : Key와 Obj에 대한 속성과 getter/setter가 존재
- Dictionary : 추상클래스로, 사전 기능에 대한 정의
- DictionaryByBinarySearchTree<Key, Obj> : String과 Student가 쌍으로 저장되는 이진탐색트리
- BinaryNode<E> : 트리에 대한 BinaryNode를 정의
- Stack<E> : 인터페이스로 스택 기본 기능 구성
- LinkedStack<E> : 스택을 LinkedList로 구현한 클래스
- LinkedNode<E> : LinkedList를 구성하는 LinkedNode에 대한 속성, 변수를 정의
- Iterator<E> : 반복자 인터페이스. 반복자 기능을 구성
- VisitDelegateForTraversal<E> : Visit 할 때마다 실행할 CallBack 함수사용법 정의한 인터페이스.

View :

- AppView : 프로그램의 입/출력을 담당한다.

Controller :

- AppController : Model을 통해 생성된 결과물을 AppView를 통해 출력한다.

(2) 함수 설명서

➔ 주요 알고리즘

1) addKeyAndObject

```
@Override
public boolean addKeyAndObject(Key aKey, Obj anObject) { // Key와 Object를 쌍으로 삽입
    if (aKey == null) {
        return false; // In any case, "aKey" cannot be null for add
    }
    DictionaryElement<Key, Obj> elementForAdd = new DictionaryElement<Key, Obj>(aKey, anObject); // 삽입을 위한 객체 생성
    BinaryNode<DictionaryElement<Key, Obj>> nodeForAdd = new BinaryNode<DictionaryElement<Key, Obj>>(elementForAdd,
            null, null);
    if (this.root() == null) { // 만약 root가 null이면
        this.setRoot(nodeForAdd); // 해당 원소를 root로 설정
        this.setSize(1); // size 1로 설정
        return true;
    }
```

⇨ 이진 탐색 트리에 Key와 Object를 삽입하는 함수

⇨ 삽입을 위한 DictionaryElement 타입의 변수를 생성하고 해당 aKey와 anObject를 값으로 넣는다.

⇨ 만약 root가 null이면 해당 원소를 root로 설정하고 size를 1로 설정한다.

```
BinaryNode<DictionaryElement<Key, Obj>> current = this.root(); // root가 null이 아니면 current 원소 지정
while (aKey.compareTo(current.element().key()) != 0) { // current 원소의 key가 aKey가 같지않으면 while문 실행
    if (aKey.compareTo(current.element().key()) < 0) { // aKey가 root보다 작다면 (left로)
        if (current.left() == null) { // left가 null이면
            current.setLeft(nodeForAdd); // left로 설정
            this.setSize(this.size() + 1); // size 증가
            return true;
        } else { // left가 null이 아니면
            current = current.left(); // left로 이동
        }
    } else { // aKey가 root보다 크다면 (right로)
        if (current.right() == null) { // right가 null이면
            current.setRight(nodeForAdd); // right로 설정
            this.setSize(this.size() + 1); // size 증가
            return true;
        } else { // right가 null이 아니면
            current = current.right(); // right로 이동
        }
    }
} // End of while
```

⇨ 만약 원소가 하나라도 존재한다면 current 노드를 root로 설정하고 삽입할 원소의 key와 비교한다.

⇨ while문 종료조건 : aKey와 current 원소의 키가 같은경우

- 만약 aKey가 root보다 작다면 (leftsubtree로)
- 만약 left()가 null이라면 left로 설정하고 size 증가 후 true를 return한다.
- left()가 null이 아니면 current 원소를 current.left()로 이동
- 만약 aKEy가 root보다 크다면 (rightsubtree로)
- 만약 right()가 null이라면 right로 설정하고 size 증가 후 true를 return한다.
- right()가 null이 아니면 current 원소를 current.right()로 이동

2) removeObjectForKey

```
public Obj removeObjectForKey(Key aKey) { // aKey에 해당하는 object를 삭제
    if (aKey == null) { // aKey가 null이면 null반환
        return null;
    }
    if (this.root() == null) { // root가 null이면 null 반환
        return null;
    }
```

주어진 aKey에 해당하는 object를 삭제하는 removeObjectForKey 함수

- aKey가 null이라면 null을 반환
- this.root()가 null이라면 null을 반환(비어있는 경우)

```
if (aKey.compareTo(this.root().element().key()) == 0) { // this.root() is the node to be removed.
    Obj objectForRemove = this.root().element().object(); // root()의 object()를 저장
    if ((this.root().left() == null) && (this.root().right() == null)) { // root()에 자식이 없다면
        this.setRoot(null); // root삭제
    } else if (this.root().left() == null) { // root()의 right()가 존재하면
        this.setRoot(this.root().right()); // right()를 root로 설정
    } else if (this.root().right() == null) { // root()의 left()가 존재하면
        this.setRoot(this.root().left()); // left()를 root로 설정
    } else { // root()의 left(), right() 모두 존재
        this.root().setElement(this.removeRightMostElementOfLeftSubTree(this.root())); // left()에서 가장 큰 element로 설정
    }
    this.setSize(this.size() - 1); // size--
    return objectForRemove; // 삭제한 object 반환
```

- aKey와 this.root().element().key()를 비교하여 같다면 반환해주기 위해 objectForRemove 변수에 root의 object를 저장한다.
- 만약 삭제할 root 원소에 자식이 하나도 없다면 root를 null로 설정한다.
- 만약 삭제할 root 원소에 left()가 null이라면 right()는 존재하므로 root에 root.right() 원소를 설정한다. 마찬가지로 root 원소에 right()가 null이라면 root에 root.left() 원소를 설정한다.
- 만약 삭제할 root 원소에 모든 자식이 존재하면 left-subtree에서 가장 큰 값을 찾아서 반환해주는 removeRightMostElementOfLeftSubTree를 호출한다.
- size를 하나 감소하고 삭제한 object 반환한다.

```
BinaryNode<DictionaryElement<Key, Obj>> current = this.root(); // root()를 current로 설정
BinaryNode<DictionaryElement<Key, Obj>> child = null;
do {
    if (aKey.compareTo(current.element().key()) < 0) { // aKey < current.element().key()
        child = current.left(); // child를 left()로 지정
        if (child == null) { // child가 null이면 aKey가 존재하지않음
            return null; // "aKey" does not
        }
        if (aKey.compareTo(child.element().key()) == 0) { // Found. "child" is to be removed
            Obj objectForRemove = child.element().object(); // child의 object 저장
            if (child.left() == null && child.right() == null) { // child의 자식이 없다면
                current.setLeft(null); // "child" is a leaf.
            } else if (child.left() == null) { // "child" has no left.
                current.setLeft(child.right());
            } else if (child.right() == null) { // "child" has no right
                current.setLeft(child.left());
            } else { // child의 left(), right() 모두 존재
                child.setElement(this.removeRightMostElementOfLeftSubTree(child)); // left()에서 가장 큰 element로 설정
            }
            this.setSize(this.size() - 1); // size--
            return objectForRemove; // 삭제한 object 반환
        }
```

- root가 aKey에 해당하지 않는다면 subtree에서 찾아야하기 때문에 root를 의미하는 current와 실제 삭제할 값을 나타낼 child BinaryNode 변수를 선언한다.
- 찾을 때까지 반복하기 위해 do-while문을 사용한다.
- 만약 aKey가 current의 key보다 작다면 child를 current의 left()로 저장한다. 이때 child가 null이라면 찾으려는 값이 존재하지 않기 때문에 null을 반환한다.
- 만약 child가 aKey에 해당하면 root가 aKey인 경우와 마찬가지로 left(), right() 모두 존재하는 경우, left()만 존재하는 경우, right()만 존재하는 경우 자식 없는 경우로 나누어 처리한다.
- aKey가 current의 key보다 크면 child를 current의 right()로 저장하고 동일하게 진행한다.

```
                this.setSize(this.size() - 1);
                return objectForRemove;
            }
        }
        current = child; // 찾을때까지 내려감

    } while (true); // return 될때까지 반복
}
```

- 만약 값을 찾지 못했다면 current를 child로 설정하여 내려간다.
- while문은 return될때까지 반복한다.

```
private DictionaryElement<Key, Obj> removeRightMostElementOfLeftSubTree(
        BinaryNode<DictionaryElement<Key, Obj>> root) { // 삭제할 element에게 자식이 모두 존재할 때 left()에서 가장 큰 element로 대체하는 함수
    // At this point, "root" has non-empty left subtree, "root" is to be removed
    BinaryNode<DictionaryElement<Key, Obj>> leftOfRoot = root.left(); // root의 left를 leftOfRoot로 저장
    if (leftOfRoot.right() == null) { // leftOfRoot에게 right-subtree가 없으므로 leftOfRoot가 rightmost가 됨
        root.setLeft(leftOfRoot.left()); // leftOfRoot를 삭제하여 반환
        return leftOfRoot.element();
```

- 삭제하려는 노드에 자식이 모두 존재할 때 left-subtree에서 가장 right 값을 반환하는 removeRightMostElementOfLeftSubTree 함수.
- 전달받은 root가 삭제될 노드이며, left-subtree의 root를 leftOfRoot 변수에 저장한다.
- 만약 이 leftOfRoot 의 right()가 null이라면 leftOfRoot가 가장 큰 값이 되는 것이므로 root.left()에 leftOfRoot의 left()를 저장하고 leftOfRoot의 element를 반환한다.

```
} else { // leftOfRoot에게 right-subtree가 하나이상 존재
    BinaryNode<DictionaryElement<Key, Obj>> parentOfRightMost = leftOfRoot; // leftOfRoot가 parentOfRightMost가 됨
    BinaryNode<DictionaryElement<Key, Obj>> rightMost = parentOfRightMost.right(); // parentOfRightMost의 right()를 rightMost로 저장
    while (rightMost.right() != null) { // rightMost가 null이라면 종료
        parentOfRightMost = rightMost; // 내려감
        rightMost = rightMost.right(); // 내려감
    }
    parentOfRightMost.setRight(rightMost.left()); // rightMost를 삭제하여 반환
    return rightMost.element();
}
```

- leftOfRoot에게 right-subtree가 하나이상 존재하면 leftOfRoot를 parentOfRightMost에 저장하고 rightMost에 parentOfRightMost의 right()를 저장한다. rightMost가 실제 반환될 값을 의미한다.
- rightmost.right()가 null이 될 때까지 내려간다.
- parentOfRightMost의 right에 rightMost의 left를 설정하고 rightMost의 element를 반환한다.

(3) 종합 설명서

➔ 프로그램 실행 순서대로 설명해보자.

```
public class _DS14_201702039_오명주 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        AppController appController = new AppController() ;
        // AppController가 실질적인 main class 이다
        appController.run();
        //여기 main() 에서는 앱 실행이 시작되도록 해주는 일이 전부이다
```

main에서 AppController 의 객체를 생성하여 run 한다. 프로그램을 실행한다.

```java
public void run() {
    AppView.outputLine("<<< 이진검색트리로 구현된 사전에서의 삽입과 삭제 >>>");
    AppView.outputLine("");
    this.setDictionary(new DictionaryByBinarySearchTree<Integer, Integer>()); // DictionaryBST정의
    this.dictionary().setVisitDelegate(this); // visitDelegate에게 객체가 누구인지 알려준다
    this.setList(DataGenerator.randomListWithoutDuplication(DEFAULT_DATA_SIZE)); // 난수생성
    this.addToDictionaryAndShowShape(); // addTo실행 + 삽입과정 출력
    this.showDictionaryInSortedOrderByCallBack(); // 재귀로 SortedOrder 출력
    this.showDictionaryInSortedOrderByIterator(); // 반복자로 SortedOrder 출력
    this.setList(DataGenerator.randomListWithoutDuplication(DEFAULT_DATA_SIZE)); // 난수생성
    this.removeFromDictionaryAndShowShape(); // removeFrom실행
    AppView.outputLine("<<< 종료 >>>");
```

- 프로그램이 실행되는 AppController의 run() 함수.
- 새로운 DictionaryByBST객체 생성한다.
- dictionary 객체에게 VisitDelegate의 주체가 AppController(this)임을 알려준다.

```java
public interface VisitDelegate<Key, Obj> {
    public void visitForSortedOrder(Key aKey, Obj aObj, int aLevel);
    public void visitForReverseOfSortedOrder(Key aKey, Obj aObj, int aLevel);
```

- Visit일을 실행하는 VisitDelegate 인터페이스

```java
@Override
public void visitForSortedOrder(Integer aKey, Integer anObj, int aLevel) { // VisitDelegate()에서 정의한 함수
    AppView.outputLine(String.format("%3d (%2d)", aKey, anObj)); // Visit했을 때 할 일을 사용자가 정의
}

@Override
public void visitForReverseOfSortedOrder(Integer aKey, Integer anObj, int aLevel) { // VisitDelegate()에서 정의한 함수
    if (aLevel == 1) { // aLevel에 따른 출력을 다르게 하고 있음
        AppView.output(String.format("%7s", "Root: "));
    } else {
        AppView.output(String.format("%7s", ""));
    }
    for (int i = 1; i < aLevel; i++) {
        AppView.output(String.format("%7s", ""));
    }
    AppView.outputLine(String.format("%3d (% 2d)", aKey, anObj));
}
```

- VisitDelegate 인터페이스의 구현은 사용자 객체인 AppController에서 하고있다.
- Visit 했을 때, 형식에 맞는 출력을 한다.

```
private void inorderRecursively(BinaryNode<DictionaryElement<Key, Obj>> aRootOfSubtree, int aLevel) { // 중위순회를 recursive하게 구현
    if (aRootOfSubtree != null) {
        this.inorderRecursively(aRootOfSubtree.left(), aLevel + 1); // left-subtree
        DictionaryElement<Key, Obj> visitedElement = aRootOfSubtree.element(); // visit 수행할 element visitElement에 저장
        this.visitDelegate().visitForSortedOrder(visitedElement.key(), visitedElement.object(), aLevel); // visit 수행
        this.inorderRecursively(aRootOfSubtree.right(), aLevel + 1); // right-subtree
    }
}

@Override
public void scanInSortedOrder() {
    this.inorderRecursively(this.root(), 1);
```

- Visit가 필요한 함수는 AppController에서 호출한 showDictionaryInSortedOrderByCallBack 함수에서 scanInSortedOrder 함수를 호출하면서 호출된다.
- 다시 AppController의 visitForSortedOrder() 함수를 호출하고 있다 => Call Back

2. 프로그램 장단점 / 특이점 분석

➔ 장점

- VisitDelegate를 인터페이스로 만들어 사용자가 Visit함수를 구현할 수 있도록 함으로써 사용자가 Visit함수에 접근하면서도 MVC 구조를 유지할 수 있었다.

➔ 단점

- 기능이 세분화 되어있어 코드가 길어질 수 있다.

3. 실행 결과 분석

(1) 입력과 출력

```
<terminated> _DS14_201702039_오명주 [Java Application] C:₩Program Files₩Java₩jdk-12.0.1₩bin₩javaw.exe (2021. 6. 8. 오후 9:23:51)
<<< 이진검색트리로 구현된 사전에서의 삽입과 삭제 >>>

[삽입 과정에서의 이진검색트리 사전의 변화 ]
> 삽입을 시작하기 전의 이진검색트리 사전 :
  EMPTY

> Key=6 (Object=0) 원소를 삽입한 후의 이진검색트리 사전 :
 Root:   6 ( 0)

> Key=9 (Object=1) 원소를 삽입한 후의 이진검색트리 사전 :
                9 ( 1)
 Root:   6 ( 0)

> Key=5 (Object=2) 원소를 삽입한 후의 이진검색트리 사전 :
                9 ( 1)
 Root:   6 ( 0)
                5 ( 2)

> Key=4 (Object=3) 원소를 삽입한 후의 이진검색트리 사전 :
                9 ( 1)
 Root:   6 ( 0)
                5 ( 2)
                        4 ( 3)

> Key=2 (Object=4) 원소를 삽입한 후의 이진검색트리 사전 :
                9 ( 1)
 Root:   6 ( 0)
                5 ( 2)
                        4 ( 3)
                                2 ( 4)


> Key=0 (Object=5) 원소를 삽입한 후의 이진검색트리 사전 :
                9 ( 1)
 Root:   6 ( 0)
                5 ( 2)
                        4 ( 3)
                                2 ( 4)
                                        0 ( 5)

> Key=1 (Object=6) 원소를 삽입한 후의 이진검색트리 사전 :
                9 ( 1)
 Root:   6 ( 0)
                5 ( 2)
                        4 ( 3)
                                2 ( 4)
                                                1 ( 6)
                                        0 ( 5)

> Key=8 (Object=7) 원소를 삽입한 후의 이진검색트리 사전 :
                9 ( 1)
                        8 ( 7)
 Root:   6 ( 0)
                5 ( 2)
                        4 ( 3)
                                2 ( 4)
                                                1 ( 6)
                                        0 ( 5)
```

```
> Key=7 (Object=9) 원소를 삽입한 후의 이진검색트리 사전 :
                9 ( 1)
                       8 ( 7)
                              7 ( 9)
 Root:   6 ( 0)
                5 ( 2)
                       4 ( 3)
                                     3 ( 8)
                              2 ( 4)
                                            1 ( 6)
                                     0 ( 5)

["Call Back"를 사용하여 보여준 사전의 내용 ]
  0 ( 5)
  1 ( 6)
  2 ( 4)
  3 ( 8)
  4 ( 3)
  5 ( 2)
  6 ( 0)
  7 ( 9)
  8 ( 7)
  9 ( 1)

["Iterator"를 사용하여 보여준 사전의 내용 ]
  0 ( 5)
  1 ( 6)
  2 ( 4)
  3 ( 8)
  4 ( 3)
  5 ( 2)
  6 ( 0)
  7 ( 9)
  8 ( 7)
  9 ( 1)

[삭제 과정에서의 이진검색트리 사전의 변화 ]
> 삭제를 시작하기 전의 이진검색트리 사전 :
                9 ( 1)
                       8 ( 7)
                              7 ( 9)
 Root:   6 ( 0)
                5 ( 2)
                       4 ( 3)
                                     3 ( 8)
                              2 ( 4)
                                            1 ( 6)
                                     0 ( 5)

> Key=0 (Object=5) 원소를 삭제한 후의 이진검색트리 사전 :
                9 ( 1)
                       8 ( 7)
                              7 ( 9)
 Root:   6 ( 0)
                5 ( 2)
                       4 ( 3)
                                     3 ( 8)
                              2 ( 4)
                                     1 ( 6)

> Key=6 (Object=0) 원소를 삭제한 후의 이진검색트리 사전 :
                9 ( 1)
                       8 ( 7)
                              7 ( 9)
 Root:   5 ( 2)
                4 ( 3)
                              3 ( 8)
                       2 ( 4)
                              1 ( 6)
```

```
> Key=8 (Object=7) 원소를 삭제한 후의 이진검색트리 사전 :
                9 ( 1)
                        7 ( 9)
 Root:    5 ( 2)
                4 ( 3)
                                3 ( 8)
                        2 ( 4)
                                1 ( 6)

> Key=1 (Object=6) 원소를 삭제한 후의 이진검색트리 사전 :
                9 ( 1)
                        7 ( 9)
 Root:    5 ( 2)
                4 ( 3)
                                3 ( 8)
                        2 ( 4)

> Key=9 (Object=1) 원소를 삭제한 후의 이진검색트리 사전 :
                7 ( 9)
 Root:    5 ( 2)
                4 ( 3)
                                3 ( 8)
                        2 ( 4)

> Key=2 (Object=4) 원소를 삭제한 후의 이진검색트리 사전 :
                7 ( 9)
 Root:    5 ( 2)
                4 ( 3)
                        3 ( 8)

> Key=7 (Object=9) 원소를 삭제한 후의 이진검색트리 사전 :
 Root:    5 ( 2)
                4 ( 3)
                        3 ( 8)


> Key=5 (Object=2) 원소를 삭제한 후의 이진검색트리 사전 :
 Root:    4 ( 3)
                3 ( 8)

> Key=3 (Object=8) 원소를 삭제한 후의 이진검색트리 사전 :
 Root:    4 ( 3)

> Key=4 (Object=3) 원소를 삭제한 후의 이진검색트리 사전 :
   EMPTY

<<< 종료 >>>
```

(2) 결과 분석 (자신의 논리적 평가, 기타 느낀 점)

➔ VisitDelegate 인터페이스를 통해 사용자가 AppController를 통해 구현할 수 있으면서 MVC 중 Model에 해당하는 DictionaryBST에서 Visit를 수행할 수 있도록 구현함으로써 MVC 구조를 사용할 수 있었다.

➔ 재귀함수를 구현할때마다 느끼는 것이지만 짧은 코드로 구현할 수 있어서 효율적이라고 생각하였다.(구현자 입장에서, 메모리는 아니지만)

4. 소스코드(대부분의 클래스 예전 과제 클래스이기 때문에 수정 부분만 캡쳐)

```
    // Constants
    private static final int DEFAULT_DATA_SIZE = 10;
    private Dictionary<Integer, Integer> _dictionary;
    private Integer[] _list;

    // Getters/Setters
    private Dictionary<Integer, Integer> dictionary() {
        return this._dictionary;
    }
    private void setDictionary(Dictionary<Integer, Integer> newDictionary) {
        this._dictionary = newDictionary;
    }
    private Integer[] list() {
        return this._list;
    }
    private void setList(Integer[] newList) {
        this._list = newList;
    }

    @Override
    public void visitForSortedOrder(Integer aKey, Integer anObj, int aLevel) { // VisitDelegate()에서 정의한 함수
        AppView.outputLine(String.format("%3d (%2d)", aKey, anObj)); // Visit했을 때 할 일을 사용자가 정의
    }

    @Override
    public void visitForReverseOfSortedOrder(Integer aKey, Integer anObj, int aLevel) { // VisitDelegate()에서 정의한 함수
        if (aLevel == 1) { // aLevel에 따른 출력을 다르게 하고 있음
            AppView.output(String.format("%7s", "Root: "));
        } else {
            AppView.output(String.format("%7s", ""));
        }
        for (int i = 1; i < aLevel; i++) {
            AppView.output(String.format("%7s", ""));
        }
        AppView.outputLine(String.format("%3d (% 2d)", aKey, anObj));
    }

    public void run() {
        AppView.outputLine("<<< 이진검색트리로 구현된 사전에서의 삽입과 삭제 >>>");
        AppView.outputLine("");
        this.setDictionary(new DictionaryByBinarySearchTree<Integer, Integer>()); // DictionaryBST정의
        this.dictionary().setVisitDelegate(this); // visitDelegate에게 객체가 누구인지 알려준다
        this.setList(DataGenerator.randomListWithoutDuplication(DEFAULT_DATA_SIZE)); // 난수생성
        this.addToDictionaryAndShowShape(); // addTo실행 + 삽입과정 출력
        this.showDictionaryInSortedOrderByCallBack(); // 재귀로 SortedOrder 출력
        this.showDictionaryInSortedOrderByIterator(); // 반복자로 SortedOrder 출력
        this.setList(DataGenerator.randomListWithoutDuplication(DEFAULT_DATA_SIZE)); // 난수생성
        this.removeFromDictionaryAndShowShape(); // removeFrom실행
        AppView.outputLine("<<< 종료 >>>");
    }

    private void showDictionary(String aTitlePrefix) {
        AppView.outputLine("> " + aTitlePrefix + "이진검색트리 사전 :");
        if (this.dictionary().isEmpty()) {
            AppView.outputLine("  EMPTY");
        } else {
            this.dictionary().scanInReverseOfSortedOrder();
        }
        AppView.outputLine("");
    }

    private void addToDictionaryAndShowShape() { // addTo실행
        AppView.outputLine("[삽입 과정에서의 이진검색트리 사전의 변화 ]");
        this.showDictionary("삽입을 시작하기 전의 ");
        for (int i = 0; i < this.list().length; i++) {
            Integer currentKey = this.list()[i];
            Integer currentObj = Integer.valueOf(i);
            this.dictionary().addKeyAndObject(currentKey, currentObj);
            this.showDictionary(String.format("Key=%d (Object=%d) 원소를 삽입한 후의 ", currentKey, currentObj));
        }

    }

    private void removeFromDictionaryAndShowShape() { // removeFrom실행
        AppView.outputLine("[삭제 과정에서의 이진검색트리 사전의 변화 ]");
        this.showDictionary("삭제를 시작하기 전의 ");
        for (int i = 0; i < this.list().length; i++) {
            Integer currentKey = this.list()[i];
            Integer currentObj = this.dictionary().removeObjectForKey(currentKey);
            this.showDictionary(String.format("Key=%d (Object=%d) 원소를 삭제한 후의 ", currentKey, currentObj));
        }
    }

    private void showDictionaryInSortedOrderByCallBack() {
        AppView.outputLine("[\"Call Back\"을 사용하여 보여준 사전의 내용 ]");
        this.dictionary().scanInSortedOrder(); // Visit을 포함하는 callBack 함수 호출
        AppView.outputLine("");
    }

    private void showDictionaryInSortedOrderByIterator() {
        AppView.outputLine("[\"Iterator\"를 사용하여 보여준 사전의 내용 ]");
        Iterator<DictionaryElement<Integer, Integer>> iterator = this.dictionary().iterator(); // 반복자 생성
        while (iterator.hasNext()) { // hasNext-> next 원소가 있다면
            DictionaryElement<Integer, Integer> dictionaryElement = iterator.next();
            AppView.outputLine(String.format("%3d (%2d)", dictionaryElement.key(), dictionaryElement.object()));
        }
        AppView.outputLine("");
    }
}
```

[AppController]

```
    private int _size; // "private" 임에 유의 . 상속 받는 class 에서는 getter/setter 를 통해서만 접근한다
    // Getter/Setter

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

    protected void setSize(int newSize) {
        this._size = newSize;
    }
    // setSize()는 사전의 크기를 변경시킨다 . 삽입과 삭제의 행위가 실행될 때 변경된다
    // 따라서 "public" 함수가 아니어야 한다 . 외부에 공개되면 안 된다
    // "private"이 아니고 "protected"인 것은 , 상속 받는 class 에서만은 사용할 수 있게 하기 위함이다
    // 상속 받는 class 에서는 , 인스턴스 변수 "_size" 를 직접 사용할 수 없게 설계하는 것이 적절한 방법이다

    // Constructors
    public Dictionary() {
        this.setSize(0);
        // 상속받는 class 의 생성자는 암묵적으로 상위 class 의 생성자를 call 한다는 것을 잊지 말 것
    }

    // Public non-abstract method: 이 class 에서 구현되어야 한다
    public boolean isEmpty() {
        return (this.size() == 0);
    }

    // Public abstract methods
    public abstract boolean isFull();

    public abstract boolean keyDoesExist(Key aKey);

    public abstract Obj objectForKey(Key aKey);

    public abstract boolean addKeyAndObject(Key aKey, Obj anObject);

    public abstract Obj removeObjectForKey(Key aKey);

    public abstract void clear();

    public abstract Iterator<DictionaryElement<Key, Obj>> iterator();

    public abstract void scanInSortedOrder();

    public abstract void scanInReverseOfSortedOrder();

    private VisitDelegate<Key, Obj> _visitDelegate;

    public VisitDelegate<Key, Obj> visitDelegate() {
        return this._visitDelegate;
    }

    public void setVisitDelegate(VisitDelegate<Key, Obj> newVisitDelegate) {
        this._visitDelegate = newVisitDelegate;
    }
}
```

[Dictionary]

```
public interface VisitDelegate<Key, Obj> {
    public void visitForSortedOrder(Key aKey, Obj aObj, int aLevel);
    public void visitForReverseOfSortedOrder(Key aKey, Obj aObj, int aLevel);
}
```

[VisitDelegate]

```java
    // Instance Variables
    private int _size;
    private LinkedNode<E> _top;

    // 생성자
    public LinkedStack() {
        this.setSize(0);
        this.setTop(null);
    }

    // Getters/Setters
    private void setSize(int newSize) {
        this._size = newSize;
    }

    private LinkedNode<E> top() {
        return this._top;
    }

    private void setTop(LinkedNode<E> newTop) {
        this._top = newTop;
    }

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

    @Override
    public boolean isFull() {
        return false;
    }

    @Override
    public boolean isEmpty() {
        return (this.size() == 0);
    }

    @Override
    public boolean push(E anElement) {
        if (this.isFull()) {
            return false;
        } else {
            LinkedNode<E> nodeForAdd = new LinkedNode<E>(anElement, null);
            nodeForAdd.setNext(this.top());
            this.setSize(this.size() + 1);
            this.setTop(nodeForAdd);
            return true;
        }
    }

    @Override
    public E pop() {
        if (this.isEmpty()) {
            return null;
        } else {
            LinkedNode<E> topNode = this.top();
            E poppedElement = topNode.element();
            this.setTop(topNode.next());
            this.setSize(this.size() - 1);
            return poppedElement;
        }
    }

    @Override
    public E peek() {
        if (this.isEmpty()) {
            return null;
        } else {
            LinkedNode<E> topNode = this.top();
            return topNode.element();
        }
    }

    @Override
    public void clear() {
        this.setSize(0);
        this.setTop(null);

    }

}
```

[LinkedStack]

```
    private BinaryNode<DictionaryElement<Key, Obj>> _root;

    // Getter/Setter
    protected BinaryNode<DictionaryElement<Key, Obj>> root() {
        return this._root;
    }

    private void setRoot(BinaryNode<DictionaryElement<Key, Obj>> newRoot) {
        this._root = newRoot;
    }

    // Constructor
    public DictionaryByBinarySearchTree() {
        this.clear();
    }

    // Private methods
    private DictionaryElement<Key, Obj> elementForKey(Key aKey) { // aKey에 해당하는 element 반환
        if (aKey != null) { // aKey가 null이 아니라면
            BinaryNode<DictionaryElement<Key, Obj>> current = this.root(); // root를 current로 지정
            while (current != null) // current(root)가 null이 아닌동안 반복
                if (current.element().key().compareTo(aKey) == 0) { // element의 key와 aKey가 같으면
                    return current.element(); // element 반환
                } else if (current.element().key().compareTo(aKey) > 0) { // aKey가 더 작으면
                    current = current.left(); // left-subtree로 이동
                } else { // aKey가 더 크면
                    current = current.right(); // right-subtree로 이동
                }
        }
        return null; // aKey가 null이거나 element가 없으면 null 반환
    }

    @Override
    public boolean isFull() {
        return false; // Always false
    }

    @Override
    public boolean keyDoesExist(Key aKey) {
        return (this.elementForKey(aKey) != null);
    }

    @Override
    public Obj objectForKey(Key aKey) { // aKey에 해당하는 object를 반환
        DictionaryElement<Key, Obj> element = this.elementForKey(aKey); // aKey에 해당하는 element
        if (element != null) { // element가 null이 아니면
            return element.object(); // element의 object를 반환
        } else { // element가 null이면
            return null; // null 반환
        }
    }

    @Override
    public boolean addKeyAndObject(Key aKey, Obj anObject) { // Key와 Object를 쌍으로 삽입
        if (aKey == null) {
            return false; // In any case, "aKey" cannot be null for add
        }
        DictionaryElement<Key, Obj> elementForAdd = new DictionaryElement<Key, Obj>(aKey, anObject); // 삽입을 위한 객체 생성
        BinaryNode<DictionaryElement<Key, Obj>> nodeForAdd = new BinaryNode<DictionaryElement<Key, Obj>>(elementForAdd,
                null, null);
        if (this.root() == null) { // 만약 root가 null이면
            this.setRoot(nodeForAdd); // 해당 원소를 root로 설정
            this.setSize(1); // size 1로 설정
            return true;
        }
        BinaryNode<DictionaryElement<Key, Obj>> current = this.root(); // root가 null이 아니면 current 원소 지정
        while (aKey.compareTo(current.element().key()) != 0) { // current 원소의 key가 aKey가 같지않으면 while문 실행
            if (aKey.compareTo(current.element().key()) < 0) { // aKey가 root보다 작다면 (left로)
                if (current.left() == null) { // left가 null이면
                    current.setLeft(nodeForAdd); // left로 설정
                    this.setSize(this.size() + 1); // size 증가
                    return true;
                } else { // left가 null이 아니면
                    current = current.left(); // left로 이동
                }
            } else { // aKey가 root보다 크다면 (right로)
                if (current.right() == null) { // right가 null이면
                    current.setRight(nodeForAdd); // right로 설정
                    this.setSize(this.size() + 1); // size 증가
                    return true;
                } else { // right가 null이 아니면
                    current = current.right(); // right로 이동
                }
            }
        } // End of while
        return false;
    }

    @Override
    public Obj removeObjectForKey(Key aKey) { // aKey에 해당하는 object를 삭제
        if (aKey    null) { // aKey가 null이면 null반환
```

[DictionaryBST]-잘린코드는 코드 참조 바랍니다.