자료구조 실습 보고서

[제13주] 이진검색트리로 구현된 사전_성적처리

2021년 05월 31일

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(Dictionary) :

- 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

Model(Ban) :

- Student : 학생 객체 생성할 수 있는 클래스
- Ban : 학생 배열 받을 학급을 의미하는 클래스
- GradeCounter : 성적에 따른 학점을 정하고 학생 수 관리

Model(Sort) :

- Sort<E> (Abstract) : 정렬의 공통 기능.
- QuickSort<E> : 퀵정렬의 기능을 구현한다.

Model(Stack) :

- Stack<E> : 인터페이스로 스택 기본 기능 구성

- ArrayList<E> : 스택을 Array로 구현한 클래스

Model(Iterator) :

- Iterator<E> : 반복자 인터페이스. 반복자 기능을 구성

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) Iterator (중위탐색)

```
@Override
public boolean hasNext() { // 다음노드가 있는지 확인하는 함수
    return ((this.nextNode() != null) || (!this.stack().isEmpty())); // nextNode가 null이 아니면 true
}
```

⇨ Iterator의 hasNext() 함수. Interface Iterator를 구현

⇨ stack이 Empty가 아니거나 nextNode가 null이 아니라면 true를 반환

```java
@Override
public DictionaryElement<Key, Obj> next() { // 중위탐색
    if (!this.hasNext()) { // hasNext가 false라면
        return null;
    } else {
        while (this.nextNode() != null) { // nextNode가 null이 아닌동안 반복
            this.stack().push(this.nextNode()); // stack에 nextNode를 push
            this.setNextNode(this.nextNode().left()); // nextNode의 left를 nextNode로 설정
        }
        BinaryNode<DictionaryElement<Key, Obj>> poppedNode = this.stack().pop(); // stack을 pop하여 저장
        DictionaryElement<Key, Obj> nextElement = poppedNode.element(); // pop한 원소를 element에 저장
        this.setNextNode(poppedNode.right()); // poppedNode의 right를 nextNode로 설정
        return nextElement;
    }
}
```

⇨ 원소를 탐색하는 next() 함수.

⇨ hasNext가 false라면 null을 반환.

⇨ nextNode()가 null이 아니면 while문을 반복

- stack에 nextNode()를 push한다.
- nextNode의 left를 nextNode로 설정한다. (중위탐색에는 left->root->right순이기 때문)

⇨ stack을 pop하여 BinaryNode로 저장하고 element를 반환한다.

(3) 종합 설명서

➔ 프로그램 실행 순서대로 설명해보자.

```java
public class _DS13_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("");
    AppView.outputLine("<<< 성적 처리를 시작합니다 >>>");

    this.inputAndStoreStudents(); // 성적 입력받아서 element 객체로 저장
    if (this.ban().isEmpty()) {
        AppView.outputLine("");
        AppView.outputLine("(오류) 학생 정보가 전혀 입력되지 않았습니다.");
    } else {
        this.showStudentList(); // 학생 리스트 출력
        this.showStatistics(); // 통계정보 출력
        this.showGradeCounts(); // 학점별로 학생 수 출력
        this.showStudentSortedByScore(); // 성적순으로 학생 출력
    }
    AppView.outputLine("");
    AppView.outputLine("<<< 성적 처리를 종료합니다 >>>");
}
```

⇨ AppController의 run 함수이다. 프로그램 실행

⇨ inputAndStroeStudents() 함수로 학번과 성적을 입력받아 element 객체로 저장한다.

⇨ 아무 정보도 없다면 오류문을 출력한다.

⇨ 그렇지 않다면 학생리스트, 통계정보, 학점별로 학생수, 성적순 리스트를 출력한다.

```java
// Student 생성하여 score 저장하는 함수
private static DictionaryElement<String, Student> inputStudent() {
    DictionaryElement<String, Student> element = new DictionaryElement<String, Student>(); // 정보를 저장할 element 선언
    String studentId = AppView.inputStudentId(); // 학번 입력
    int score = AppView.inputScore(); // 학생 점수 입력

    if (AppController.studentIDIsValid(studentId) && AppController.scoreIsValid(score)) { // 입력받은 studentId, score 모두 true라면
        Student student = new Student(); // student 객체 생성
        student.setScore(score); // 점수 저장
        element.setKey(studentId); // 학번 저장
        element.setObject(student);
    }
```

⇨ 정보를 저장할 element를 선언하고 사용자로부터 학번과 학생 점수를 입력받는다.

⇨ 입력받은 학번과 점수의 유효성을 확인하여 모두 true라면 Student 객체를 생성하여 score를 저장하고 element에 key와 object를 설정한다.

```java
if (!AppController.studentIDIsValid(studentId)) { // 학번의 길이를 확인하여 오류문 출력
    AppView.outputLine("(오류) 학번의 길이가 너무 깁니다. 최대 " + AppController.VALID_STUDENTID_LENGTH + " 입니다.");
}
if (!AppController.scoreIsValid(score)) { // 성적 숫자를 확인하여 오류문 출력
    AppView.outputLine("(오류) 성적이 " + AppController.VALID_MIN_SCORE + " 보다 작거나 " + AppController.VALID_MAX_SCORE
            + " 보다 커서, 정상적인 점수가 아닙니다.");
```

⇨ 학번과 성적이 유효하지 않다면 오류문을 출력한다.

```
while (storingAStudentWasSuccessful && AppView.doesContinueToInputStudent()) { // 정상처리 &&'Y'입력확인
    DictionaryElement<String, Student> element = AppController.inputStudent(); // 입력 받아온 element 변수에 저장
    this.ban().addKeyAndObject(element.key(), element.object()); // ban에 element의 key와 object를 삽입
    AppView.outputLine("");
}
```

⇨ 해당 원소를 삽입한다.

```
private void showStudentList() {
    AppView.outputLine("");
    AppView.outputLine("[학생 목록]");

    Iterator<DictionaryElement<String, Student>> iterator = this.ban().iterator(); // 반복자 생성
    DictionaryElement<String, Student> element = new DictionaryElement<String, Student>();
    Student student = null;
    while (iterator.hasNext()) { // hasNext()가 null이 아닌동안 반복
        element = iterator.next();
        student = element.object(); // 다음 student를 저장
        AppView.outputStudentList(element.key(), student.score());
```

⇨ 학생 리스트를 출력하는 showStudentList() 함수

⇨ 반복자를 생성하고 원소를 저장할 element 변수 선언

⇨ iterator가 null이 아닌동안 while문 반복

- element에 next() 원소를 저장
- student 객체에 element의 object를 저장 , 출력문 출력

⇨ next가 중위탐색을 실행하므로 해당 반복문은 학번순으로 출력하게 된다.

```
// 학급 성적 통계를 출력해주는 함수
private void showStatistics() {
    AppView.outputLine("");
    AppView.outputLine("[학급 성적 처리 결과]");

    AppView.outputTotalNumberOfStudents(this.ban().size()); // 전체 학생 수 출력
    AppView.outputAverageScore(this.ban().average()); // 학급 평균점수 출력
    AppView.outputNumberOfStudentsAboveAverage(this.ban().numberOfStudentsAboveAverage()); // 평균이상인 학생 수 출력
    AppView.outputHighestScore(this.ban().highest().score()); // 학급 최고점 출력
    AppView.outputLowestScore(this.ban().lowest().score()); // 학급 최저점 출력
```

⇨ 학급 성적 통계를 출력하는 showStatistics() 함수

⇨ 전체 학생 수, 평균점수, 평균 이상의 학생 수, 학급 최고점, 최저점을 출력한다.

```java
private DictionaryElement<String, Student> lowestRecursively(BinaryNode<DictionaryElement<String, Student>> aRoot) {
    DictionaryElement<String, Student> lowest = aRoot.element(); // aRoot의 element()를 lowest 변수로 저장
    if (aRoot.left() != null) { // aRoot의 left가 null이 아니라면
        DictionaryElement<String, Student> lowestOfLeftSubtree = this.lowestRecursively(aRoot.left()); // LeftSubtree를 재귀로 검사
        if (lowestOfLeftSubtree.object().score() < lowest.object().score()) { // LeftSubtree의 가장 낮은 값과 aRoot의 점수를 비교
            lowest = lowestOfLeftSubtree; // 더 낮은 값을 저장
        }
    }
    if (aRoot.right() != null) { // aRoot의 right가 null이 아니라면
        DictionaryElement<String, Student> lowestOfRightSubtree = this.lowestRecursively(aRoot.right()); // RightSubtree를 재귀로 검사
        if (lowestOfRightSubtree.object().score() < lowest.object().score()) { // RightSubtree의 가장 낮은 값과 aRoot의 점수를 비교
            lowest = lowestOfRightSubtree; // 더 낮은 값을 저장
        }
```

⇨ 최저점을 재귀로 구하는 lowestRecursively 함수

⇨ aRoot의 left, right를 재귀적으로 가장 작은 값을 구한다.

⇨ aRoot의 left가 null이 아니라면 LeftSubtree를 재귀적으로 검사하여 가장 낮은 값과 aRoot 값을 비교하여 subtree의 가장 낮은 값을 저장한다. RightSubtree도 재귀적으로 검사하여 작은 값을 저장하여 반환한다.

```java
// 학생들의 성적을 순서대로 출력하는 함수
private void showStudentSortedByScore() {
    AppView.outputLine("");
    AppView.outputLine("[학생들의 성적순 목록]");
```

⇨ 학생들을 성적순으로 출력하는 showStudetnSortedByScore() 함수

```java
DictionaryElement<String, Student> element = new DictionaryElement<String, Student>();
Student student = null;
Student students[] = this.ban().studentsSortedByScore(); // 학생 성적 오름차순으로 정렬
for (int i = students.length - 1; i >= 0; i--) {
    Iterator<DictionaryElement<String, Student>> iterator = this.ban().iterator(); // 반복자 생성
    while (iterator.hasNext()) { // iterator의 hasNext가 존재한다면
        element = iterator.next(); // element에 원소 저장
        student = element.object(); // element의 object를 student 변수에 저장
        if (students[i].score() == student.score()) { // 만약 원소를 찾는다면
            AppView.outputStudentInfo(element.key(), student.score(), Ban.scoreToGrade(student.score())); // 출력
```

⇨ 원소를 저장할 element 변수와 Student 객체 변수를 선언한다.

⇨ students 배열 객체에 학생 성적을 오름차순으로 정렬한 결과를 저장한다.

⇨ students 배열은 오름차순이므로 마지막 idx-0까지 for문을 반복한다.
- iterator을 통해 ban의 원소들과 students[i]를 비교하여 같은 점수를 발견하면 key와 score를 출력한다.

2. 프로그램 장단점 / 특이점 분석

➔ 장점

- 삽입, 삭제, 검색 모두 평균적으로 시간 복잡도가 O(logN)으로 트리의 높이에 비례하여 다소 빠르다.
-

➔ 단점

- 최악의 경우, 시간 복잡도는 O(N)이 될 수도 있다. 한쪽으로만 기울어진 형태는 노드의 개수에 비례하여 삽입, 삭제, 검색이 이루어진다.

3. 실행 결과 분석

(1) 입력과 출력

```
<terminated> _DS13_201702039_오명주 [Java Application] C:₩Program Files₩Java₩jdk-12.0.1₩bin₩javaw.exe (2021. 5. 31. 오후 9:35:03)

<<< 성적 처리를 시작합니다 >>>

? 성적을 입력하려면 'Y'또는 'y'를, 종료하려면 다른 아무 키나 치시오: y
- 학번을 입력하시오 : 20119999
- 점수를 입력하시오 : 82

? 성적을 입력하려면 'Y'또는 'y'를, 종료하려면 다른 아무 키나 치시오: y
- 학번을 입력하시오 : 20118888
- 점수를 입력하시오 : 45

? 성적을 입력하려면 'Y'또는 'y'를, 종료하려면 다른 아무 키나 치시오: y
- 학번을 입력하시오 : 20112222
- 점수를 입력하시오 : 93

? 성적을 입력하려면 'Y'또는 'y'를, 종료하려면 다른 아무 키나 치시오: Y
- 학번을 입력하시오 : 20115555
- 점수를 입력하시오 : 66

? 성적을 입력하려면 'Y'또는 'y'를, 종료하려면 다른 아무 키나 치시오: Y
- 학번을 입력하시오 : 20113333
- 점수를 입력하시오 : 87

? 성적을 입력하려면 'Y'또는 'y'를, 종료하려면 다른 아무 키나 치시오: N
<성적 입력을 마칩니다.>

[학생 목록]
학번 : 20112222, 점수 : 93
학번 : 20113333, 점수 : 87
학번 : 20115555, 점수 : 66
학번 : 20118888, 점수 : 45
학번 : 20119999, 점수 : 82

[학급 성적 처리 결과]
전체 학생 수 : 5
학급 평균 점수 : 74.6
평균 이상인 학생 수 : 3
학급 최고 점수 : 93
학급 최저 점수 : 45

[학점별 학생수]
A 학점은 모두 1명 입니다.
B 학점은 모두 2명 입니다.
C 학점은 모두 0명 입니다.
D 학점은 모두 1명 입니다.
F 학점은 모두 1명 입니다.

[학생들의 성적순 목록]
학번 : 20112222, 점수 : 93, 학점 : A
학번 : 20113333, 점수 : 87, 학점 : B
학번 : 20119999, 점수 : 82, 학점 : B
학번 : 20115555, 점수 : 66, 학점 : D
학번 : 20118888, 점수 : 45, 학점 : F

<<< 성적 처리를 종료합니다 >>>
```

```
<terminated> _DS13_201702039_오명주 [Java Application] C:₩Program Files₩Java

<<< 성적 처리를 시작합니다 >>>

? 성적을 입력하려면 'Y'또는 'y'를, 종료하려면 다른 아무 키나 치시오: y
- 학번을 입력하시오 : 1234567890
- 점수를 입력하시오 : 1000
(오류) 학번의 길이가 너무 깁니다. 최대 9 입니다.
(오류) 성적이 0 보다 작거나 100 보다 커서, 정상적인 점수가 아닙니다.

? 성적을 입력하려면 'Y'또는 'y'를, 종료하려면 다른 아무 키나 치시오: N
<성적 입력을 마칩니다.>

(오류) 학생 정보가 전혀 입력되지 않았습니다.

<<< 성적 처리를 종료합니다 >>>
```

- 학번의 길이가 9보다 긴 경우, 성적이 정상적이지 않은 경우, 학생 정보가 입력되지 않고 종료된 경우 -> 오류처리

(2) 결과 분석 (자신의 논리적 평가, 기타 느낀 점)

➔ 이론적 시간 복잡도

- 이진 탐색 트리는 평균적으로 O(logN)으로 트리의 높이에 비례하지만 Left 혹은 Right 한쪽으로 쏠린 트리인 경우 시간복잡도가 O(N)으로 노드에 비례하여 이진탐색트리의 장점을 살리지 못 한다.
- 이 경우 AVL Tree로 해결할 수 있다.

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

```
public class AppView {
    private static Scanner scanner = new Scanner(System.in);

    // 생성자
    public AppView() {}

    // 출력 관련 함수
    public static void output(String message) { // 한줄을 출력하는 함수 (한줄이 띄워지지않는다)
        System.out.print(message); // 입력받은 message를 출력한다
    }
    public static void outputLine(String message) { // 한줄을 출력하는 함수 (한줄이 띄워진다)
        System.out.println(message); // 입력받은 message를 출력한다
    }
    public static void outputTotalNumberOfStudents(int numberOfStudents) {
        System.out.println("전체 학생 수 : " + numberOfStudents);
    }
    public static void outputHighestScore(int aScore) {
        System.out.println("학급 최고 점수 : " + aScore);
    }
    public static void outputLowestScore(int aScore) {
        System.out.println("학급 최저 점수 : " + aScore);
    }
    public static void outputAverageScore(double average) {
        System.out.println("학급 평균 점수 : " + average);
    }
    public static void outputNumberOfStudentsAboveAverage(int numberOfStudents) {
        System.out.println("평균 이상인 학생 수 : " + numberOfStudents);
    }
    public static void outputNumberOfStudentsForGrade(char aGrade, int numberOfStudenst) {
        System.out.println(aGrade + " 학점은 모두 " + numberOfStudenst + "명 입니다.");
    }
    public static void outputStudentList(String aStudentID, int aScore) {
        System.out.println("학번 : " + aStudentID + ", 점수 : " + aScore);
    }
    public static void outputStudentInfo(String aStudentID, int aScore, char aGrade) {
        System.out.println("학번 : " + aStudentID + ", 점수 : " + aScore + ", 학점 : " + aGrade);
    }

    // 입력 관련 함수
    public static int inputInt() throws NumberFormatException {
        return Integer.parseInt(AppView.scanner.next());
    }
    public static boolean doesContinueToInputStudent() {
        AppView.output("? 성적을 입력하려면 'Y'또는 'y'를, 종료하려면 다른 아무 키나 치시오: ");
        String line = null;
        do { // 빈 줄이 아닐때까지 입력받는다
            line = AppView.scanner.nextLine();
        } while (line.equals(""));
        char answer = line.charAt(0);
        return ((answer == 'Y') || (answer == 'y'));
    }
    public static String inputStudentId() {
        while (true) {
            AppView.output("- 학번을 입력하시오 : ");
            String studentId = scanner.next();
            return studentId;
        }
    }
    public static int inputScore() {
        while (true) {
            try {
                AppView.output("- 점수를 입력하시오 : ");
                int score = AppView.inputInt();
                return score;
            } catch (NumberFormatException e) {
                AppView.outputLine("(오류) 정수가 입력되지 않았습니다");
            }
        }
    }
}
```

[AppView]

```java
    private BinaryNode<DictionaryElement<Key, Obj>> _root;
    // private int _size ;
    // 필요 없는 이유는 ? Getter/Setter 는?

    // 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) {
        if (aKey != null) {
            BinaryNode<DictionaryElement<Key, Obj>> current = this.root();
            while (current != null)
                if (current.element().key().compareTo(aKey) == 0) {
                    return current.element();
                } else if (current.element().key().compareTo(aKey) > 0) {
                    current = current.left();
                } else {
                    current = current.right();
                }
        }
        return null;
    }

    @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) {
        DictionaryElement<Key, Obj> element = this.elementForKey(aKey);
        if (element != null) {
            return element.object();
        } else {
            return null;
        }
    }

    @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) {
        // TODO Auto-generated method stub
        return null;
    }

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

    }

    @Override
    public Iterator<DictionaryElement<Key, Obj>> iterator() {
        return (new DictionaryIterator());
    }

    private class DictionaryIterator implements Iterator<DictionaryElement<Key, Obj>> {
        private BinaryNode<DictionaryElement<Key, Obj>> _nextNode;
        private Stack<BinaryNode<DictionaryElement<Key, Obj>>> _stack;

        private BinaryNode<DictionaryElement<Key, Obj>> nextNode() {
            return this._nextNode;
        }

        private void setNextNode(BinaryNode<DictionaryElement<Key, Obj>> newNextNode) {
            this._nextNode = newNextNode;
        }

        private Stack<BinaryNode<DictionaryElement<Key, Obj>>> stack() {
            return this._stack;
        }

        private void setStack(Stack<BinaryNode<DictionaryElement<Key, Obj>>> newStack) {
            this._stack = newStack;
        }

        // Constructor
        private DictionaryIterator() {
            this.setStack(new ArrayList<BinaryNode<DictionaryElement<Key, Obj>>>());
            this.setNextNode(DictionaryByBinarySearchTree.this.root());
        }

        @Override
        public boolean hasNext() { // 다음노드가 있는지 확인하는 함수
            return ((this.nextNode() != null) || (!this.stack().isEmpty())); // nextNode가 null이 아니면 true
        }

        @Override
        public DictionaryElement<Key, Obj> next() { // 주의할점
            if (!this.hasNext()) { // hasNext가 false라면
                return null;
            } else {
                while (this.nextNode() != null) { // nextNode가 null이 아닌동안 반복
                    this.stack().push(this.nextNode()); // stack에 nextNode를 push
                    this.setNextNode(this.nextNode().left()); // nextNode의 left를 nextNode로 설정
                }
                BinaryNode<DictionaryElement<Key, Obj>> poppedNode = this.stack().pop(); // stack을 pop하여 저장
                DictionaryElement<Key, Obj> nextElement = poppedNode.element(); // pop한 원소를 element에 저장
                this.setNextNode(poppedNode.right()); // poppedNode의 right를 nextNode로 설정
                return nextElement;
            }
        }
    }
}
```

[DictionaryByBST]

```
public class AppController {

    // 상수
    private static final int VALID_MAX_SCORE = 100;
    private static final int VALID_MIN_SCORE = 0;
    private static final int VALID_STUDENTID_LENGTH = 9;

    // 비공개 인스턴스 변수들
    private Ban _ban;
    private GradeCounter _gradeCounter;

    // Getters/Setters
    private Ban ban() {
        return this._ban;
    }

    private void setBan(Ban newBan) {
        this._ban = newBan;
    }

    private GradeCounter gradeCounter() {
        return this._gradeCounter;
    }

    private void setGradeCounter(GradeCounter newGradeCounter) {
        this._gradeCounter = newGradeCounter;
    }

    // 생성자
    public AppController() {
        setBan(new Ban());
    }

    // 입력받은 학생 점수가 유효한지 확인하는 함수
    private static boolean scoreIsValid(int aScore) {
        return (aScore >= AppController.VALID_MIN_SCORE && aScore <= AppController.VALID_MAX_SCORE);
    }

    // 입력받은 학번이 유효한지 확인하는 함수
    private static boolean studentIDIsValid(String aStudentId) {
        return (aStudentId.length() <= AppController.VALID_STUDENTID_LENGTH);
    }

    // Student 생성하여 score 저장하는 함수
    private static DictionaryElement<String, Student> inputStudent() {
        DictionaryElement<String, Student> element = new DictionaryElement<String, Student>(); // 정보를 저장할 element 선언
        String studentId = AppView.inputStudentId(); // 학번 입력
        int score = AppView.inputScore(); // 학생 점수 입력

        if (AppController.studentIDIsValid(studentId) && AppController.scoreIsValid(score)) { // 입력받은 studentId, score 모두 true라면
            Student student = new Student(); // student 객체 생성
            student.setScore(score); // 점수 저장
            element.setKey(studentId); // 학번 저장
            element.setObject(student);
        }
        if (!AppController.studentIDIsValid(studentId)) { // 학번의 길이를 확인하여 오류문 출력
            AppView.outputLine("(오류) 학번의 길이가 너무 깁니다. 최대 " + AppController.VALID_STUDENTID_LENGTH + " 입니다.");
        }
        if (!AppController.scoreIsValid(score)) { // 성적 숫자를 확인하여 오류문 출력
            AppView.outputLine("(오류) 성적이 " + AppController.VALID_MIN_SCORE + " 보다 작거나 " + AppController.VALID_MAX_SCORE
                    + " 보다 커서, 정상적인 점수가 아닙니다.");

        }
        return element;
    }

    private void inputAndStoreStudents() {
        AppView.outputLine("");
        boolean storingAStudentWasSuccessful = true; // 정상적으로 저리가 되었는지 확인하는 변수
        while (storingAStudentWasSuccessful && AppView.doesContinueToInputStudent()) { // 정상처리 && 'Y' 입력확인
            DictionaryElement<String, Student> element = AppController.inputStudent(); // 입력 받아온 element 변수에 저장
            this.ban().addKeyAndObject(element.key(), element.object()); // ban에 element의 key와 object를 삽입
            AppView.outputLine("");
        }
        AppView.outputLine("<성적 입력을 마칩니다.>");
    }

    // 학급 성적 통계를 출력해주는 함수
    private void showStatistics() {
        AppView.outputLine("");
        AppView.outputLine("[학급 성적 처리 결과]");

        AppView.outputTotalNumberOfStudents(this.ban().size()); // 전체 학생 수 출력
        AppView.outputAverageScore(this.ban().average()); // 학급 평균점수 출력
        AppView.outputNumberOfStudentsAboveAverage(this.ban().numberOfStudentsAboveAverage()); // 평균이상인 학생 수 출력
        AppView.outputHighestScore(this.ban().highest().score()); // 학급 최고점 출력
        AppView.outputLowestScore(this.ban().lowest().score()); // 학급 최저점 출력

    }

    // 학점별로 학생수를 출력해주는 함수
    private void showGradeCounts() {
        AppView.outputLine("");
        AppView.outputLine("[학점별 학생수]");
        this.setGradeCounter(this.ban().countGrades());
        AppView.outputNumberOfStudentsForGrade('A', this.gradeCounter().numberOfA()); // A 학생 수 출력
        AppView.outputNumberOfStudentsForGrade('B', this.gradeCounter().numberOfB()); // B 학생 수 출력
        AppView.outputNumberOfStudentsForGrade('C', this.gradeCounter().numberOfC()); // C 학생 수 출력
        AppView.outputNumberOfStudentsForGrade('D', this.gradeCounter().numberOfD()); // D 학생 수 출력
        AppView.outputNumberOfStudentsForGrade('F', this.gradeCounter().numberOfF()); // F 학생 수 출력

    }

    // 학생들의 성적을 순서대로 출력하는 함수
    private void showStudentSortedByScore() {
        AppView.outputLine("");
        AppView.outputLine("[학생들의 성적순 목록]");

        DictionaryElement<String, Student> element = new DictionaryElement<String, Student>();
        Student student = null;
        Student students[] = this.ban().studentsSortedByScore(); // 학생 성적 오름차순으로 정렬
        for (int i = students.length - 1; i >= 0; i--) {
            Iterator<DictionaryElement<String, Student>> iterator = this.ban().iterator(); // 반복자 생성
            while (iterator.hasNext()) { // iterator의 hasNext가 존재한다면
                element = iterator.next(); // element에 next 저장
                student = element.object(); // element의 object를 student 변수에 저장
                if (students[i].score() == student.score()) { // 만약 점수를 찾는다면
                    AppView.outputStudentInfo(element.key(), student.score(), Ban.scoreToGrade(student.score())); // 출력
                }
            }
        }
    }

    private void showStudentList() {
        AppView.outputLine("");
        AppView.outputLine("[학생 목록]");

        Iterator<DictionaryElement<String, Student>> iterator = this.ban().iterator(); // 반복자 생성
        DictionaryElement<String, Student> element = new DictionaryElement<String, Student>();
        Student student = null;
        while (iterator.hasNext()) { // hasNext()가 null이 아닐동안 반복
            element = iterator.next();
            student = element.object(); // 다음 student를 저장
            AppView.outputStudentList(element.key(), student.score());
        }
    }

    public void run() {
        AppView.outputLine("");
        AppView.outputLine("<<< 성적 처리를 시작합니다 >>>");

        this.inputAndStoreStudents(); // 성적 입력받아서 element 객체로 저장
        if (this.ban().isEmpty()) {
            AppView.outputLine("");
            AppView.outputLine("(오류) 학생 정보가 전혀 입력되지 않았습니다.");
        } else {
            this.showStudentList(); // 학생 리스트 출력
            this.showStatistics(); // 통계정보 출력
            this.showGradeCounts(); // 학점별로 학생 수 출력
            this.showStudentSortedByScore(); // 성적순으로 학생 출력
        }
        AppView.outputLine("");
        AppView.outputLine("<<< 성적 처리를 종료합니다 >>>");
    }
}
```

[AppController]

```java
public class DictionaryElement<Key extends Comparable<Key>, Obj extends Comparable<Obj>> {
    private Key _key;
    private Obj _object;

    public DictionaryElement() {
        this.setKey(null);
        this.setObject(null);
    }

    public DictionaryElement(Key givenKey, Obj givenObject) {
        this.setKey(givenKey);
        this.setObject(givenObject);
    }

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

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

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

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

[DictionaryElement]

```java
    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 nonabstract 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();

}
```

[Dictionary]