자료구조: 2022년 1학기 [실습]

제 11 주:

정렬 결과 검증

강 지 훈
jhkang@cnu.ac.kr
충남대학교 컴퓨터융합학부

실습 목표

실습 목표

- 이론적 관점
  - 삽입 정렬과 퀵 정렬 알고리즘 각각의 개념과 특성 이해
- 구현적 관점
  - 정렬 알고리즘의 구현 방법을 이해
  - 실험 데이터 생성 방법을 이해
  - 결과 검증 방법을 이해

과제에서 해결할 문제

문제

- 정렬 알고리즘들이 정렬을 정상적으로 실행하고 있는지를 검증한다.
  - 삽입 정렬과 퀵 정렬 알고리즘을 각각 구현한다.
  - 검증용 데이터 리스트로, 다음의 세 가지를 생성한다.
    - 오름차순 (ascending) 데이터 리스트
    - 내림차순 (descending) 데이터 리스트
    - 무작위 (Random) 데이터 리스트
  - 이 리스트를 정렬을 한 후, 정상적으로 정렬이 되었는지를 검증한다.

입출력

- 입력:
  - 없음
  - 필요한 데이터는 프로그램에서 생성
- 출력:
  - 성능 검증 결과

이 과제에서 필요한 Class 는?

- Class "AppController"
- Class "AppView"

- Model
  - Class "DataGenerator"
  - Abstract Class "Sort<E>"
  - Class "InsertionSort<E>" extends "Sort<E>"
  - Class "QuickSort<E>" extends "Sort<E>"
  - Enum "ListOrder"

출력의 예

```
<<< 정렬 알고리즘의 정렬 결과를 검증하는 프로그램을 시작합니다 >>>

> 정렬 결과의 검증:

[오름차순 리스트] 의 앞 부분: 0  1  2  3  4
- [오름차순 리스트]를 [InsertionSort] 한 결과는 올바릅니다.
- [오름차순 리스트]를 [QuickSort] 한 결과는 올바릅니다.

[내림차순 리스트] 의 앞 부분: 9999  9998  9997  9996  9995
- [내림차순 리스트]를 [InsertionSort] 한 결과는 올바릅니다.
- [내림차순 리스트]를 [QuickSort] 한 결과는 올바릅니다.

[무작위 리스트] 의 앞 부분: 7586  2682  2308  5320  1499
- [무작위 리스트]를 [InsertionSort] 한 결과는 올바릅니다.
- [무작위 리스트]를 [QuickSort] 한 결과는 올바릅니다.

<<< 정렬 알고리즘의 정렬 결과를 검증하는 프로그램을 종료합니다 >>>
```

구현할 내용

main()

main()을 위한 class

```
public class _DS11_학번_이름 {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        AppController  appController= new AppController() ;
        appController.run () ;
    }
}
```

Class “AppView”

AppView:

- 지난 실습의 것을 그대로 사용한다.

Enum “ListOrder”

Enum "ListOrder"

```
public enum  ListOrder
{
    // 이번 실험에서는, 3 가지 유형의 데이터 리스트를 구분하고 있다.
    // 이 구분을 표현할 목적으로 Enum "ListOrder" 를 사용한다.
    Ascending ,     // 오름차순 데이터 리스트의 유형을 표현
    Descending ,    // 내림차순 데이터 리스트의 유형을 표현
    Random ;        // 무작위 데이터 리스트의 유형을 표현

    // Enum 은 Class 의 특수한 경우로 간주된다.
    // 따라서 아래와 같이 상수를 선언할 수 있다.

    public static final String[]  ORDER_NAMES = { "오름차순", "내림차순", "무작위" } ;

    // 또한 아래와 같이 member method 역시 선언할 수 있다.
    // 즉, Enum 안에 선언된 값들은 Enum 의 객체 인스턴스로 인식된다.
    // Ascending, Descending, Random 각각이 객체 인스턴스이다.

    public String  orderName () {
        return  ListOrder.ORDER_NAMES [ this.ordinal() ] ;
            // "ordinal()"은 모든 Enum 에 미리 정의되어 있는 함수로,
            // 선언된 값의 Enum 안에서의 순서를 정수로 얻을 수 있다.
            // 즉. Ascending.ordinal() 은 0, Descending.ordinal() 은 1,
            // Random.ordinal() 은 2 를 얻는다.
    }
}
```

ListOrder: 공개 인스턴스 함수 "orderName()"

- public String orderName ()
  - Enum 의 각 값에 문자열 이름을 부여하여, Enum 값에 해당하는 문자열을 얻으려는 것이다.
  - 사용 예:

```
ListOrder  order = ListOrder.Random ;
AppView.outputLine (order.orderName()) ;
```

  - 미리 부여한 이름은 상수 배열 ORDER_NAEMS[] 로 정의해 두고 있다:

```
public static final String[] ORDER_NAMES =
                                { "오름차순", "내림차순", "무작위" } ;
```

    - this.ordinal() 값을 인덱스로 사용하여 ORDER_NAMES[] 배열에서 해당 문자열 이름을 얻을 수 있다.
      (앞 쪽의, 함수 "orderName()" 의 구현 코드를 볼 것)

Class “AppController”

AppController: 개요[1]

```
<<< 정렬 알고리즘의 정렬 결과를 검증하는 프로그램을 시작합니다 >>>

> 정렬 결과의 검증:

[오름차순 리스트] 의 앞 부분: 0  1  2  3  4
- [오름차순 리스트]를 [InsertionSort] 한 결과는 올바릅니다.
- [오름차순 리스트]를 [QuickSort] 한 결과는 올바릅니다.

[내림차순 리스트] 의 앞 부분: 9999  9998  9997  9996  9995
- [내림차순 리스트]를 [InsertionSort] 한 결과는 올바릅니다.
- [내림차순 리스트]를 [QuickSort] 한 결과는 올바릅니다.

[무작위 리스트] 의 앞 부분: 7586  2682  2308  5320  1499
- [무작위 리스트]를 [InsertionSort] 한 결과는 올바릅니다.
- [무작위 리스트]를 [QuickSort] 한 결과는 올바릅니다.

<<< 정렬 알고리즘의 정렬 결과를 검증하는 프로그램을 종료합니다 >>>
```

프로그램 시작 메시지

실험 내용 설명 메시지

오름차순 데이터를 정렬한 결과의 검증:
validateWithAscendingOrderList()

내림차순 데이터를 정렬한 결과의 검증:
validateWithDescendingOrderList()

무작위 데이터를 정렬한 결과의 검증:
validateWithRandomOrderList()

프로그램 종료 메시지

AppController: 개요[2]

```
<<< 정렬 알고리즘의 정렬 결과를 검증하는 프로그램을 시작합니다 >>>

> 정렬 결과의 검증:

[오름차순 리스트] 의 앞 부분: 0  1  2  3  4
- [오름차순 리스트]를 [InsertionSort] 한 결과는 올바릅니다.
- [오름차순 리스트]를 [QuickSort] 한 결과는 올바릅니다.

[내림차순 리스트] 의 앞 부분: 9999  9998  9997  9996  9995
- [내림차순 리스트]를 [InsertionSort] 한 결과는 올바릅니다.
- [내림차순 리스트]를 [QuickSort] 한 결과는 올바릅니다.

[무작위 리스트] 의 앞 부분: 7586  2682  2308  5320  1499
- [무작위 리스트]를 [InsertionSort] 한 결과는 올바릅니다.
- [무작위 리스트]를 [QuickSort] 한 결과는 올바릅니다.

<<< 정렬 알고리즘의 정렬 결과를 검증하는 프로그램을 종료합니다 >>>
```

정렬할 데이터 리스트의 앞 부분 출력:
showFirstPartOfDataList()

검증 결과 출력:
validateSortsAndShowResult()

정렬할 데이터 리스트의 앞 부분 출력
showFirstPartOfDataList()

검증 결과 출력:
validateSortsAndShowResult()

정렬할 데이터 리스트의 앞 부분 출력:
showFirstPartOfDataList()

검증 결과 출력:
validateSortsAndShowResult()

AppController: 상수, 인스턴스 변수, 생성자

```
public class AppController {
   // 상수
   private static final int  TEST_SIZE = 10000 ;
   private static final int  FIRST_PART_SIZE = 5 ;
   private static final InsertionSort<Integer>    INSERTION_SORT = new InsertionSort<Integer>() ;
   private static final QuickSort<Integer>        QUICK_SORT = new QuickSort<Integer>() ;

   // 비공개 변수들
   private Integer[]   _list ;
   private ListOrder  _listOrder ;

   // Getter/Setter
   private Integer[] list() {…}
   private void setList (Integer[] newList) {…}
   private ListOrder listOrder() {…}
   private void setListOrder (ListOrder newListOrder) {…}

   // 생성자
   public  AppController() {
   }

   // 비공개함수의 구현
   ……

   // 공개함수의 구현
   ……
} // End of class "AppController"
```

AppController: 공개 함수 run()

```
public void  run () {
    AppView.outputLine
        ("<<< 정렬 알고리즘의 정렬 결과를 검증하는 프로그램을 시작합니다 >>>") ;
    AppView.outputLine ("") ;

    AppView.outputLine ( "> 정렬 결과의 검증:" ) ;
    AppView.outputLine ("") ;

    this.validateWithAscendingOrderList () ;
    this.validateWithDescendingOrderList () ;
    this.validateWithRandomOrderList () ;

    AppView.outputLine
        ("<<< 정렬 알고리즘의 정렬 결과를 검증하는 프로그램을 종료합니다 >>>") ;
}
```

AppController: validateWithAscendingOrderList(), validateWithDescendingOrderList (), validateWithRandomOrderList ()

```
private void  validateWithAscendingOrderList () {
      this.setListOrder (ListOrder.Ascending) ;
      this.setList (DataGenerator.ascendingList (AppController.TEST_SIZE)) ;
      this.showFirstPartOfDataList () ;
      this.validateSortsAndShowResult () ;
}

private void  validateWithDescendingOrderList () {
      this.setListOrder (ListOrder.Descending) ;
      this.setList (DataGenerator.descendingList (AppController.TEST_SIZE)) ;
      this.showFirstPartOfDataList () ;
      this.validateSortsAndShowResult () ;
}

private void  validateWithRandomOrderList () {
      this.setListOrder (ListOrder.Random) ;
      this.setList (DataGenerator.randomList (AppController.TEST_SIZE)) ;
      this.showFirstPartOfDataList () ;
      this.validateSortsAndShowResult () ;
}
```

AppController: validateSortsAndShowResult(), showFirstPartOfDataList(), validateSort()

```
private void  showFirstPartOfDataList () {
    AppView.output (
        "[" + this.listOrder().orderName() + " 리스트] 의 앞 부분: " ) ;
    ..... // 여기를 채우시오 (출력화면 참고)
    AppView.outputLine ("") ;
}

private void  validateSortsAndShowResult () {
    this.validateSort (AppController.INSERTION_SORT) ;
    this.validateSort (AppController.QUICK_SORT) ;
    AppView.outputLine ("") ;
}

private void validateSort (Sort<Integer> aSort) {
    Integer[]  list = this.copyList (this._list) ;
        // 동일한 리스트로 여러 번 (실제로는 2 번) 정렬하게 된다.
        // 매번 원본 리스트를 복사하여 정렬한다.
    aSort.sort (list, list.length) ;
    this.showValidationMessage (aSort, list) ;
}
```

AppController: copyList()

```
private Integer[]  copyList (Integer[] aList) {
    // 주어진 배열 객체 aList[] 의 복사본을 만들어 돌려준다.
    // aList[] 자체는 복사하지만,
    // 배열의 원소 객체 자체는 복사하지 않고 공유한다.

    Integer[]  copiedList = new Integer[aList.length] ;
    ..... // 여기를 채우시오

    return  copiedList ;
}
```

AppController: validateSort(), showValidationMessage()

```
private boolean  sortedListIsValid (Integer[] aList) {
    // 주어진 aList 의 원소들이 오름차순으로 되어 있으면 true 를 돌려준다.
    for ( int i = 0 ; i < (aList.length-1) ; i++ ) {
        if ( aList[i].compareTo (aList[i+1]) > 0 ) {
            return  false ; // 오름차순이 아닌 순서를 발견
        }
    }
    return  true ; // 리스트 전체가 오름차순으로 되어 있다.
}

private void  showValidationMessage (Sort<Integer> aSort, Integer[] aList) {
    AppView.output (
        "[" + this.listOrder().orderName() + " 리스트]를 [" +
        aSort.getClass().getSimpleName() + "] 한 결과는 " ) ;
    if ( this.sortedListIsValid (aList) ) {
        ..... // 여기를 채우시오 (출력화면 참고)
    }
    else {
        ..... // 여기를 채우시오 (출력화면 참고)
    }
}
```

Class “DataGenerator”

"Static" Class

- Static Class:
  - 객체 인스턴스를 필요로 하지 않는 class. 즉 객체를 생성할 필요가 없는 class.
  - 예: Math
  - 오로지 static constant, static variable, 그리고 static method 만 존재하게 된다.
- Java 에서는 "Static" class 를 선언하는 방법을 직접적으로 제공하지는 않는다. 즉 다음과 같이 선언할 수 없다.
  - public **static** class Math { // Compile Error!!

    ……

    }
- 그러나, 실질적인 static class 를 만들 수는 있다:

Java 에서 static class 선언 방법

- class 의 선언은 "final":
  - public **final** class **DataGenerator** {
    ……
    }
  - 일반적으로 "Math" 와 같은 라이브러리를 만들 때 사용하게 된다. 라이브러리를 상속받는다?
  - 그럼에도, Static class 자체를 확장하고 싶다면?
    - static variable 이나 static method 를 추가한 확장된 static class 를 만들고 싶다면?
- 생성자는 "private" 으로 선언:
  - **private** DataGenerator() { }
  - 객체 인스턴스를 생성할 수 없게 막아 놓는다.
- 모든 상수, 변수, 함수는 "static" 으로 선언:
  - public **static** Integer[] ascendingOrderList (int aSize) {
    ……
    }
  - 사용법:
    Integer[] myAscendingList = DataGenerator.ascendingOrderList (100) ;

Static class "DataGenerator" [1]

```
public final class  DataGenerator {
        // Static class.
        // 더 이상 상속할 필요가 없으므로 "final" 로 선언

        // 생성자는 private. 따라서 외부에서 객체를 만들 수 없다.
        private  DataGenerator() {} ;

        // 모든 공개함수는 static
        public static Integer[]  ascendingList (int aSize) {
              Integer[]  list = null ;
              if (  aSize > 0 ) {
                    list = new Integer[aSize] ;
                    for ( int i = 0 ;  i < aSize ; i++ ) {
                          list[i] = i ;
                    }
              }
              return  list ;
        }

        public static Integer[]  descendingList (int aSize) {
              ..... // 여기를 작성하시오
        }
```

Static class "DataGenerator" [2]

```
public final class  DataGenerator {
        ......

        public static Integer[]  randomList (int aSize) {
            // 겹치는 원소가 없는 무작위 리스트를 생성하여, 돌려준다.
            Integer[]  list = null ;
            if (  aSize > 0 ) {
                // 일단 Ascending order list 를 만든다
                list = new Integer[aSize] ;
                for ( int i = 0 ;  i < aSize ; i++ ) {
                    list[i] = i ;
                }

                // 각 원소 list[i] 에 대해 무작위 위치 r 을 생성하여 list[i] 와 list[r] 를 맞바꾼다.
                Random  random = new Random() ;
                for ( int i = 0 ;  i < aSize ; i++ ) {
                    int  r = random.nextInt (aSize) ;
                    Integer temp = list[i] ;
                    list[i] = list[r] ;
                    list[r] = temp ;
                }
            }
            return  list ;
        }
}
```

Abstract Class “Sort<E>”

Abstract Class:

- Abstract (추상) class 로는 객체를 생성할 수 없다.
  - 반드시 상속받는 class 가 있어서, 그 상속받은 class 를 사용하여 객체를 생성할 수 있다. (상속 받는 class 는 물론 abstract class 가 아니어야 한다.)
- 인스턴스 변수:
  - "private" 으로 선언하면 상속이 되지 않는다.
  - 상속받는 class 에서 사용하려면 반드시 "protected" 로 선언되어야 한다.
  - 인스턴스 변수이므로 "public" 으로 선언하는 것은 피해야 할 것이다.
- 생성자:
  - "protected" 로 선언한다. 외부에서 사용되지 않지만, 상속은 가능해야 한다.
- 멤버 함수:
  - Abstract class 자체에서 이미 구현되는 함수:
    - 상속받는 class 에서 사용하려면 반드시 "protected", 또는 공개적으로 사용하려면 "public" 으로 선언되어야 한다.
  - Abstract class 에서는 선언만 하고 구현되지는 않아, 상속받는 class 에서 override 하여 구현할 함수:
    - Abstract method 로 선언해야 한다.

Abstract Class:

- InsertionSort 와 QuickSort 는 정렬 구현 방법은 다르다.
- 그러나, 사용자의 관점에서 사용법은 동일할 수 있다.
- 사용법과 구현 방법이 동일한 멤버 함수는 공통으로 만들어 사용하려면?
  - "swap()" 과 같은 일부 함수는 사용법은 물론, 구현 방법도 동일하다. 이러한 기능을 각각의 정렬 마다 구현하는 것은 낭비적이다.

- 두 정렬의 동일한 사용법을 뽑아서 하나의 abstract class "sort" 로 정의:
  - 구현자의 관점:
    - InsertionSort 와 QuickSort 각각의 정렬은 이 abstract class "Sort" 를 상속받도록 하는 것이 **효과적인 코드 관리 방법**.
    - 예:
      ```
      public class  InsertionSort<E>  extends Sort<E> {...}
      ```
  - 사용자의 관점:
    - 객체 변수의 선언은 abstract class "Sort" 로 하고, 객체의 생성만 필요에 맞추어 특정 정렬 방법의 객체로 생성한다, 나중에 InsertionSort 를 QuickSort 로 바꾸고 싶으면, 객체 생성만 QuickSort 객체로 생성하도록 바꾸면 되며, 다른 부분은 수정할 필요가 없다.
    - 예:
      ```
      Sort<Integer>  mySort = new InsertionSort<Integer>() ;
      mySort.sort (list) ;
      ```

Abstract class "Sort" : 구현

```
public abstract class Sort<E extends Comparable<E>> {

    // Protected Method
    protected void  swap (E[] aList, int p, int q) {
        E  tempElement = aList[p] ;
        aList[p] = aList[q] ;
        aList[q] = tempElement ;
    }

    // Constructor
    protected  Sort () {
    }

    // Public Method
    public abstract boolean  sort (E[] aList, int aSize) ;

} // End of "Sort"
```

Class “InsertionSort”

InsertionSort: 개요

- 맨 앞 원소의 값을 (–∞) 로:
  - 이렇게 하면, 원소 삽입 위치를 찾는 while loop 조건이 비교를 한 번만 하게 되어 시간적으로 유리하다.
  - 구현 방법:
    - 가장 작은 원소를 찾아 맨 앞에 갖다 놓기로 한다.
    - 이 원소는 실질적인 –∞ 값의 역할을 한다.

- Abstract class "Sort" 를 상속받는다:
  - "swap()" 은 상속받은 것을 그대로 사용한다.
  - Abstract method 인 "sort()" 는 override 하여 구현한다.

InsertionSort: 구현

```
public class InsertionSort<E extends Comparable<E>> extends Sort<E>  {
    // Constructor
    public  InsertionSort () { }

    // Public Method
    public boolean  sort (E[] aList, int aSize) {
        if ( (aSize < 1) || (aSize > aList.length) ) {
            return  false ;
        }
        int  minLoc = 0 ;
        for ( int  i = 1 ; i < aSize ; i++ ) {
            if ( aList[i].compareTo(aList[minLoc]) < 0 ) {
                minLoc = i ;
            }
        }
        this.swap (aList, 0, minLoc) ; // Abstract class "Sort" 에 구현된 것을 그대로 사용하고 있다.

        for ( int  i = 2 ; i < aSize ; i++ ) {
            E  insertedElement = aList[i] ;
            int  insertionLoc = i – 1 ;
            while ( aList[insertionLoc].compareTo(insertedElement) > 0 ) {
                aList[insertionLoc+1] = aList[insertionLoc] ;
                insertionLoc--;
            }
            // While loop 조건이 false 라서 loop 종료. 따라서, (insertionLoc+1) 이 원소 삽입 위치.
            aList[insertionLoc+1] = insertedElement ;
        }
        return  true ;
    }
} // End of "Sort"
```

Class “QuickSort”

QuickSort: 개요

- 맨 뒤 원소의 값을 (+∞) 로:
  - 퀵 정렬에서는 맨 뒤에 +∞ 가 존재해야 한다.
    - 그렇지 않다면?
  - 구현 방법:
    - 정렬 구간: [0, aSize-1]
    - 가장 큰 원소를 찾아 맨 뒤인 (aSize-1) 위치에 갖다 놓는다.
    - 이 원소는 실질적인 +∞ 값의 역할을 한다.
    - 실제 퀵 정렬 구간은 [0, aSize-2] 가 된다.

- Abstract class "Sort"를 상속받는다:
  - swap() 은 상속받은 것을 그대로 사용한다.
  - Abstract method 인 sort() 는 override 하여 구현한다.

QuickSort: 필요한 함수들

- private int pivot (E[] aList, int left, int right) ;
  - 주어진 구간에서 pivot 원소 위치를 결정하여 얻는다.
  - 현재는, 구간의 가장 왼쪽 원소 위치를 사용하기로 한다.

- private int partition (E[] aList, int left, int right) ;
  - 주어진 구간을 partition 한다.
  - Partition을 실행한 후에, pivot 원소가 최종적으로 놓인 위치를 얻는다.

- private void quickSortRecursively (E[] aList, int left, int right) ;
  - 주어진 구간을 재귀적으로 퀵 정렬 한다.

- public boolean sort (E[] aList, int aSize) ;
  - Abstract class Sort<E> 의 sort()를 override 하여 구현한다.

QuickSort: 구현

```
public class QuickSort<E extends Comparable<E>> extends Sort<E>  {
    // Constructor
    public  QuickSort () { }

    // Private methods
    private int  pivot (E[] aList, int left, int right) {
        return  left ;
    }
    private int  partition (E[] aList, int left, int right) {
        ..... // 여기를 작성하시오
    }
    private void  quickSortRecursively (E[] aList, int left, int right) {
        ..... // 여기를 작성하시오
    }

    @Override
    public boolean  sort (E[] aList, int aSize) {
        if ( (aSize < 1) || (aSize > aList.length) ) {
            return  false;
        }
        int  maxLoc = 0 ;
        for ( int i = 1 ;  i < aSize ; i++) {
            if ( aList[i].compareTo(aList[maxLoc] > 0 ) {
                maxLoc = i ;
            }
        }
        this.swap (aList, maxLoc, aSize-1) ;

        this.quickSortRecursively (aList, 0 , aSize-2) ;
        return  true ;
    }
} // End of "Sort"
```

요약

확인하자

- 정렬 알고리즘의 기본 작동 개념
  - 삽입 정렬
  - 퀵 정렬

- 다음의 개념을 이해하자
  - Static Class
  - Abstract Class
  - Interface “Comparable”
  - Enum Class

생각해 볼 점

- 자료형 “int” 와 Class (추상자료형) “Integer” 의 차이점은?
- Static Class 란?
- Abstract Class 란?

⇒ 생각해 볼 점에 대해, 자신의 의견을 보고서에 작성하시오.

[실습 끝]