자료구조 실습 보고서

[제06주] 리스트의 성능 측정

2021년 04월 11일

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.

➔ 리스트 성능 비교 프로그램에서의 각 클래스 별 MVC 구조 역할

Model :

- Experiment : 실험 자체를 위한 추상 자료형. 출력될 실험변수와 실험방법을 구성
- MeasuredResult : 실험 결과를 받고 합과 최대값을 구함.
- SortedArrayList : ArrayList로 구현된 정렬된 리스트
- UnsortedArrayList : ArrayList로 구현된 정렬되지 않은 리스트
- SortedLinkedList : LinkedList로 구현된 정렬된 리스트
- UnsortedLinkedList : LinkedList로 구현된 정렬되지 않은 리스트
- Coin : 코인의 변수, 속성을 구성. 비교함수도 존재
- LinkedNode : Node의 변수, 속성을 구성

View :

- AppView : 프로그램의 입/출력을 담당한다.

Controller :

- AppController : Model을 통해 결과물을 AppView를 통해 출력한다.

(2) 함수 설명서

➔ 주요 알고리즘

(1) SortedArrayList

```
public void add(E anElement) {
    if (this.size() == 0) { // size가 0일 때
        this.elements()[this.size()] = anElement; // 첫번째 배열에 anElement가 들어감
    } else {
        int order = 0; // 순서 order = 0 으로 선언
        for (int i = 0; i < this.size(); i++) { // this.size() 만큼 반복
            if (this.elements()[i].compareTo(anElement) > 0) { // 처음부터 마지막 배열을 anElement와 비교 후 +1이 나오면
                order++; // 순서를 + 1 해준다.
                break; // 종료
            }
        }
        if (order == this.size()) { // 찾은 순서가 마지막 순서라면
            this.elements()[order] = anElement; // order+1번째에 anElement 삽입
        } else {
            this.makeRoomAt(order); // order번째부터 한 칸씩 미루기
            this.elements()[order] = anElement; // order번 째 배열에 anElement 삽입
        }
    }
    this.setSize(this.size() + 1); // 사이즈 + 1
}
```

Add : Size를 확인하여 0이라면 첫번째 배열에 원소를 넣어준다. compareTo 함수를 이용하여 객체 값을 비교하여 삽입할 위치를 order에 저장한다. 만약 찾은 순서가 마지막이라면 그냥 넣어주고 중간 인덱스라면 makeRoomAt 함수를 이용하여 한 칸 씩 미루어 삽입한다.

```
private void makeRoomAt(int aPosition) {
    for (int i = this.size(); i > aPosition; i--) { // this.size()부터 aPosition보다 클 때까지 i --
        this.elements()[i] = this.elements()[i - 1]; // 배열을 한 칸씩 당긴다.
    }
}
```

주어진 인덱스 이후 한 칸 씩 뒤로 값을 밀어주는 함수이다.

```
public E max() {
    // SortedArrayList에서 가장 큰 값은 맨 뒤에 있다.
    if (this.isEmpty()) {
        return null;
    } else {
        return this.elements()[this.size() - 1]; // 마지막 원소를 반환
    }
}
```

Max : 오름차순으로 정렬된 리스트이기 때문에 배열의 마지막 원소를 반환한다.

(2) UnsortedArrayList

```
// 원소를 더하는 함수
public void add(E anElement) {
    this.elements()[this.size()] = anElement;
    this.setSize(this.size() + 1);
}
```

Add : 정열 되지 않는 리스트에 대한 구현이기 때문에 가장 마지막 순서에 원소를 삽입한다.

```
// max값을 반환하는 함수
public E max() {
    if (this.isEmpty()) {
        return null;
    } else {
        E maxElement = this.elements()[0];
        for (int i = 1; i < this.size(); i++) {
            if (maxElement.compareTo(this.elements()[i]) < 0)
                maxElement = this.elements()[i];
        }
        return maxElement;
    }
}
```

Max : 리스트가 비어 있다면 null을 반환. 배열의 크기만큼 max값 찾는 반복문을 이용하여 만약 max 변수에 저장된 값과 현재 배열 원소를 비교하여 더 큰 값을 max에 저장한다. 이 때도 객체의 비교는 compareTo 함수를 활용한다.

(3) SortedLinkedList

```
// 리스트에 원소를 삽입하는 함수
public boolean add(E anElement) {
    if (this.isFull()) { // size = capacity라면 false 반환
        return false;
    } else {
        LinkedNode<E> nodeForAdd = new LinkedNode<E>(anElement, null); // 주어진 anElement를 가진 노드생성
        if (this.isEmpty()) { // 리스트가 비어있다면
            this.setHead(nodeForAdd); // head에 삽입
        } else {
            LinkedNode<E> current = this.head(); // 현재 비교하는 노드
            LinkedNode<E> previous = null; // current의 앞 노드 삽입을 하려면 , 앞 노드를 알아야 한다
            while (current != null) { // 리스트의 끝에 도달할 때 까지 비교 검색한다
                if (current.element().compareTo(anElement) > 0) {
                    break; // 삽입할 위치를 찾은 것이므로 비교 검색 중지
                }
                previous = current;
                current = current.next();
            }
            if (previous == null) {
                nodeForAdd.setNext(this.head());
                this.setHead(nodeForAdd);
            } else {
                nodeForAdd.setNext(current);
                previous.setNext(nodeForAdd);
            }
        }
        this.setSize(this.size() + 1);
        return true;
    }
```

Add : 만약 리스트가 가득 찼다면 false를 반환. 삽입할 element를 가진 객체를 생성한다. 그리고 리스트가 비었다면 head에 삽입하고 그렇지 않다면 compareTo 함수를 이용하여 삽입할 위치를 검색한다. 위치를 검색했을 때 head라면 head로 삽입하고 중간 위치라면 previous의 next를 새로운 노드로 설정하여 삽입하여 준다.

```
public E max() {
    if (this.isEmpty()) {
        return null;
    } else {
        LinkedNode<E> currentNode = this._head; // 현재노드 생성 후 헤드로 설정
        while (currentNode.next() != null) { // 리스트 마지막까지 반복
            currentNode = currentNode.next();
        }
        return currentNode.element(); // 리스트 마지막 노드 element 반환
    }
}
```

Max : 리스트가 비어 있다면 null을 반환. 정렬 되어있는 LinkedList 이므로 while문을 이용하여 마지막 노드까지 걸어 나가 마지막 노드의 element를 반환한다.

(4) UnsortedLinkedList

```
// 노드 추가하는 함수
public void add(E anElement) {
    // head에 노드 추가하는 코드
    LinkedNode<E> nodeForAdd = new LinkedNode<E>(anElement, null);
    nodeForAdd.setNext(this.head());
    this.setHead(nodeForAdd);
    this.setSize(this.size() + 1);
}
```

Add : 정렬 되지 않은 리스트에 대한 구현이기 때문에 해당 element를 갖는 객체를 생성하여 head에 넣어준다. Size는 1 증가해준다.

```
// 최대값 반환하는 함수
public E max() {
    if (this.isEmpty()) { // 비어있다면 null 반환
        return null;
    } else {
        LinkedNode<E> currentNode = this.head(); // 현재노드 생성 후 헤드로 설정
        LinkedNode<E> maxNode = new LinkedNode<E>(this.head().element(), null);
        while (currentNode != null) { // 마지막 노드까지 반복
            if (maxNode.element().compareTo(currentNode.element()) < 0) {
                maxNode.setElement(currentNode.element());
            }
            currentNode = currentNode.next();
        }
        return maxNode.element();
    }
}
```

Max : 리스트가 비어 있다면 null을 반환. 현재 노드와 최대값을 저장할 노드 객체를 생성해주고 마지막 노드까지 반복하는 while문을 이용하여 max노드와 현재 노드를 비교해준다. 더 큰 값을 max노드에 저장하고 걸어 나가 반복한다. max값을 반환한다.

```
@Override
public int compareTo(Coin aCoin) {
    if (this.value() < aCoin.value()) {
        return -1;
    } else if (this.value() > aCoin.value()) {
        return 1;
    } else {
        return 0;
    }
}
```

객체 비교는 Coin 클래스의 campareTo 함수를 이용한다. 두 객체의 value의 차를 구하여 각각의 값을 비교하여 주어진 음수 양수 0 중 하나를 반환한다.

(3) 종합 설명서

➔ 프로그램 실행 순서대로 설명해보자.

```
public class DS06_201702039_오명주 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        AppController appcontroller = new AppController(); //appcontroller 선언
        appcontroller.run(); // start
    }

}
```

main에서 AppController의 객체를 생성하여 run 한다. 프로그램 실행.

```
// 공개 함수
public void run() {
    AppView.outputLine("<<<리스트의 성능 측정 프로그램을 시작합니다 .>>>");
    AppView.outputLine("! 리스트의 구현에 따른 성능의 차이를 알아봅니다 : (단위 : Micro Second)");
    AppView.outputLine("");
    AppView.outputLine("<Sorted Array List>");
    this.experiment().measureForSortedArrayList(); // 실험 객체에게 SortedArrayList에 대한 성능 측정을 실행하게 한다
    this.showExperimentResults(); // 실험 결과 출력
    AppView.outputLine("");
    AppView.outputLine("<Unsorted Array List>");
    this.experiment().measureForUnsortedArrayList(); // 실험 객체에게 UnSortedArrayList에 대한 성능 측정을 실행하게 한다
    this.showExperimentResults(); // 실험 결과 출력
    AppView.outputLine("");
    AppView.outputLine("<Sorted Linked List>");
    this.experiment().measureForSortedLinkedList(); // 실험 객체에게 SortedLinkedList에 대한 성능 측정을 실행하게 한다
    this.showExperimentResults(); // 실험 결과 출력
    AppView.outputLine("");
    AppView.outputLine("<Unsorted Linked List>");
    this.experiment().measureForUnsortedLinkedList(); // 실험 객체에게 UnsortedLinkedList에 대한 성능 측정을 실행하게 한다
    this.showExperimentResults(); // 실험 결과 출력
    AppView.outputLine("");
    AppView.outputLine("<<< 리스트의 성능 측정 프로그램을 종료합니다 >>>"); // 종료
}
```

AppConroller의 run 함수에서는 프로그램 시작과 진행을 알리는 출력문들과 experiment의 생성자를 만들어 각각의 리스트의 성능을 측정한 결과물을 출력해준다. 여기서 showExperimentResults 함수는 측정 결과를 출력하는 함수이다.

```
// 생성자
public AppController() {
    this.setExperiment(new Experiment()); // Experiment 객체를 생성
    this.experiment().generateData(); // experiment의 generateData 실행, 실험객체에게 성능 측정에 사용할 데이터를 생성하게 함
}
```

AppController은 Experiment 객체 선언과 experiment(), generateData() 함수를 실행한다.

```
// 생성자
public Experiment() {
    this.setNumberOfIteration(DEFAULT_NUMBER_OF_ITRATION); // 반복 5
    this.setFirstSize(DEFAULT_FIRST_SIZE); // 처음크기 10000
    this.setSizeIncrement(DEFAULT_SIZE_INCREMENT); // 증가량 10000
    this.setData(new Coin[this.maxSize()]); // Setter of Data
    this.setMeasuredResults(new MeasuredResult[this.numberOfIteration()]); // Setter of MeasuredResults

}
```

크기가 주어지지 않을 때의 Experiment 클래스의 생성자이다. 처음 리스트의 크기는 10000으로 초기화하고 증가 량을 10000씩 주어 10000, 20000, 30000 .. 50000까지 5번 반복하여 측정한다.

```
// 성능 측정에 필요한 데이터를 생성한다, 난수사용
public void generateData() {
    Random random = new Random(); // random 선언
    for (int i = 0; i < this.maxSize(); i++) { // this.maxSize()만큼 반복
        int randomCoinValue = random.nextInt(this.maxSize()); // 0~maxSize()의 난수를 생성 후 randomCoinValue에 대입
        this.data()[i] = new Coin(randomCoinValue); // data의 i번째 배열은 randomCoinValue를 갖는 코인
    }
}
```

난수 생성 - Random 클래스를 import 하여 random 객체를 만든다. maxSize() 만큼 반복하여 난수를 생성하고 data()[] 배열에 삽입한다. maxSize() = firstSize(10000) + sizeIncrement(10000) * Interation 4 = 50000

```
for (int i = 0; i < dataSize; i++) { // dataSize만큼 반복
    start = System.nanoTime(); // 시작
    list.add(this.data()[i]); // i번째 배열을 list에 add
    stop = System.nanoTime(); // 종료
    durationForAdd += (stop - start); // 종료시점 - 시작시점 을 계속 더해준다.

    start = System.nanoTime(); // 시작
    maxCoin = list.max(); // list에서 maxCoin을 찾기
    stop = System.nanoTime(); // 종료
    durationForMax += (stop - start); // 종료시점 - 시작시점을 계속 더해준다.
}
this.measuredResults()[iteration] = new MeasuredResult(dataSize, durationFo
dataSize += this.sizeIncrement();
```

측정을 진행하는 함수의 반복문. dataSize만큼 반복한다. Start 측정 시작 – 리스트에 add – Stop 측정 종료, Start 측정 시작 – 리스트에서 max 찾기 – Stop 측정 종료를 반복한다.

Experiment 클래스에서 결과를 AppController로 넘겨 주어 화면에 출력한다.

2. 프로그램 장단점 / 특이점 분석

➔ 장점

- MVC 모델을 이용하여 가독성과 생산성이 뛰어나다. 각 클래스, 함수의 역할이 분명해서 코드와 프로그램을 잘 이해할 수 있다.

- 구현 프로그램의 실행 시간이나 성능을 측정할 수 있다는 게 편리하다고 느껴졌다. 앞으로 구현할 때나 프로그램을 개발할 때 시간을 측정하면서 효율적으로 개발할 수 있을 것 같다.

- 각각의 리스트에 대한 성능을 확인했으니 목적에 맞게 리스트들을 이용할 수 있다. 만약 정보를 찾을 때 시간이 적은 경우가 필요하다면 SortedList를 이용하면 될 것이다.

➔ 단점

- 난수를 생성하여 사용하면서 코드 구현이 복잡해졌다. 실제로 정보를 저장할 때는 난수를 생성하지 않아도 되므로 해결될 것이다.

- SortedList는 삽입 할 때 시간이 오래 걸리는 것을 확인할 수 있었다.

- unSortedList는 max 값을 찾을 때 시간이 오래 걸리는 것을 확인할 수 있었다.

- LinkedList는 ArrayList에 비해 최대값 찾는 것에 시간이 걸리는 것을 확인하였다. 인덱스를 이용한 출력이 바로 가능한 Array에 비해 가장 마지막 노드까지 걸어 나가야 하기 때문이다.

3. 실행 결과 분석

(1) 입력과 출력 (화면 capture하여 제출)

```
<terminated> DS06_201702039_오명주 [Java Application] C:₩Program Files₩Java₩jdk-12.0.1₩bin₩javaw.exe (2021. 4. 11. 오후 7:57:40)
<<<리스트의 성능 측정 프로그램을 시작합니다 .>>>
! 리스트의 구현에 따른 성능의 차이를 알아봅니다 : (단위 : Micro Second)

<Sorted Array List>
[크기 : 10000] 삽입 :   215659 , 최대값 :     1386
[크기 : 20000] 삽입 :   600046 , 최대값 :     1887
[크기 : 30000] 삽입 :  1473069 , 최대값 :     3754
[크기 : 40000] 삽입 :  2989258 , 최대값 :     2831
[크기 : 50000] 삽입 :  5343057 , 최대값 :     2760

<Unsorted Array List>
[크기 : 10000] 삽입 :     1872 , 최대값 :   155291
[크기 : 20000] 삽입 :     3142 , 최대값 :   551595
[크기 : 30000] 삽입 :     4832 , 최대값 :  1144295
[크기 : 40000] 삽입 :     4868 , 최대값 :  2228448
[크기 : 50000] 삽입 :     3786 , 최대값 :  3693245

<Sorted Linked List>
[크기 : 10000] 삽입 :   290088 , 최대값 :   349311
[크기 : 20000] 삽입 :  1459325 , 최대값 :  1885487
[크기 : 30000] 삽입 :  3323542 , 최대값 :  4831103
[크기 : 40000] 삽입 :  6368700 , 최대값 :  9857633
[크기 : 50000] 삽입 :  9226921 , 최대값 : 14598294

<Unsorted Linked List>
[크기 : 10000] 삽입 :     1710 , 최대값 :   131626
[크기 : 20000] 삽입 :     3358 , 최대값 :   497188
[크기 : 30000] 삽입 :     5015 , 최대값 :  1128008
[크기 : 40000] 삽입 :     3914 , 최대값 :  1913050
[크기 : 50000] 삽입 :     4254 , 최대값 :  3124574

<<< 리스트의 성능 측정 프로그램을 종료합니다 >>>
```

(2) 결과 분석 (자신의 논리적 평가, 기타 느낀 점)

⇨ 정렬된 리스트, 정렬되지 않은 리스트, 배열, 노드 차이를 시간 복잡도를 통해 알 수 있었다.
SortedArrayList – add O(n), max O(1) (add가 반복문을 통해 값을 삽입하기 때문)
UnsortedArrayList – add O(1), max O(n) (max 값을 찾을 때 반복문 이용)
SortedLinkedList – add O(n), max O(n) (max 값 찾을 때도 반복문으로 끝 노드까지 가기때문)
UnsortedLinkedList – add O(1), max O(n) (max 값 찾을 때 반복문 이용)

⇨ UnsortedLinkedList를 구현할 때 처음엔 리스트의 마지막에 값을 삽입하는 것으로 구현하였는데 그러다 보니 반복문을 통해 마지막 노드까지 가야하는 복잡함이 있어 O(n) 만큼 걸리었다. 성능을 향상시키기 위해 head에 노드를 추가하는 형태로 바꾸었더니 O(1) 로 성능이 향상된 것을 확인할 수 있었다.

⇨ 정렬을 하면 더 좋은 배열이라고 판단될 수 있지만 삽입하고 정렬하는데 시간이 오래 걸린다는 것을 알게 되었다. 프로그램을 실행하면서 크기를 10만으로 변경하여 실행하였더니 1분정도 더 오래 걸리는 것을 확인하였다.

4. 소스코드

```
public class DS06_201702039_오명주 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        AppController appcontroller = new AppController(); //appcontroller 선언
        appcontroller.run(); // start
    }

}
```

```java
public class AppView {

    // 출력 관련 함수
    // 한줄을 출력하는 함수 (한줄이 띄워진다)
    public static void outputLine(String aMessage) {
        System.out.println(aMessage);
    }

    // 한줄을 출력하는 함수 (한줄이 띄워지지않는다)
    public void output(String aMessage) {
        System.out.print(aMessage);
    }

    // 포맷에 맞게 출력하는 함수
    public static void outputResults(int size, long durationForAdd, long durationForMax) {
        System.out.println("[크기 : " + String.format("%5d", size) + "] " + "삽입 : " + String.format("%8d", durationForAdd)
                + " , 최대값 : " + String.format("%8d ", durationForMax));
    }
}
```

```java
    // 비공개 인스턴스 변수
    private Experiment _experiment;

    // Getters/Setters
    private Experiment experiment() {
        return this._experiment;
    }

    private void setExperiment(Experiment newExperiment) {
        this._experiment = newExperiment;
    }

    // 생성자
    public AppController() {
        this.setExperiment(new Experiment()); // Experiment 객체를 생성
        this.experiment().generateData(); // experiment의 generateData 실행, 실험객체에게 성능 측정에 사용할 데이터를 생성하게 함
    }

    // 비공개 함수
    // 결과를 출력하는 함수
    private void showExperimentResults() {
        MeasuredResult[] results = this.experiment().measuredResults();
        // results 배열을 experiment().measuredResults()로 설정
        for (int i = 0; i < this.experiment().numberOfIteration(); i++) {
            AppView.outputResults(results[i].size(), results[i].durationForAdd() / 1000, // Nano 를 Micro로 변환
                    results[i].durationForMax() / 1000); // Nano 를 Micro로 변환
        }
    }

    // 공개 함수
    public void run() {
        AppView.outputLine("<<<리스트의 성능 측정 프로그램을 시작합니다 .>>>");
        AppView.outputLine("! 리스트의 구현에 따른 성능의 차이를 알아봅니다 : (단위 : Micro Second)");
        AppView.outputLine("");
        AppView.outputLine("<Sorted Array List>");
        this.experiment().measureForSortedArrayList(); // 실험 객체에게 SortedArrayList에 대한 성능 측정을 실행하게 한다
        this.showExperimentResults(); // 실험 결과 출력
        AppView.outputLine("");
        AppView.outputLine("<Unsorted Array List>");
        this.experiment().measureForUnsortedArrayList(); // 실험 객체에게 UnSortedArrayList에 대한 성능 측정을 실행하게 한다
        this.showExperimentResults(); // 실험 결과 출력
        AppView.outputLine("");
        AppView.outputLine("<Sorted Linked List>");
        this.experiment().measureForSortedLinkedList(); // 실험 객체에게 SortedLinkedList에 대한 성능 측정을 실행하게 한다
        this.showExperimentResults(); // 실험 결과 출력
        AppView.outputLine("");
        AppView.outputLine("<Unsorted Linked List>");
        this.experiment().measureForUnsortedLinkedList(); // 실험 객체에게 UnsortedLinkedList에 대한 성능 측정을 실행하게 한다
        this.showExperimentResults(); // 실험 결과 출력
        AppView.outputLine("");
        AppView.outputLine("<<< 리스트의 성능 측정 프로그램을 종료합니다 >>>"); // 종료
    }
}
```

```java
    private static final int DEFAULT_VALUE = 0; // Default 금액
    private int _value; // 동전 금액

    public Coin() {
        this._value = DEFAULT_VALUE; // 값을 받지 못한 Coin의 동전금액은 DEFAULT = 0
    }

    public Coin(int givenValue) {
        this._value = givenValue; // 값을 받은 Coin의 동전금액은 주어진 값
    }

    public int value() {
        return this._value; // getter of value
    }

    public void setValue(int newValue) {
        this._value = newValue; // setter of value
    }

    public interface Comparable {
        public int compareTo(Coin aCoin);

    }

    @Override
    public int compareTo(Coin aCoin) {
        if (this.value() < aCoin.value()) {
            return -1;
        } else if (this.value() > aCoin.value()) {
            return 1;
        } else {
            return 0;
        }
    }
}


public class LinkedNode<E> {
    // 비공개 인스턴스 변수
    private E _element; // 현재 노드에 있는 원소
    private LinkedNode<E> _next; // 다음노드

    // Getter/Setter
    public E element() {
        return this._element;
    }

    public void setElement(E newElement) {
        this._element = newElement;
    }

    public LinkedNode<E> next() {
        return this._next;
    }

    public void setNext(LinkedNode<E> newNext) {
        this._next = newNext;
    }

    // 생성자
    public LinkedNode() { // 비어 있을 때
        this.setElement(null);
        this.setNext(null);
    }

    public LinkedNode(E givenElement) { // 원소 하나일때 (head 원소)
        this.setElement(givenElement);
        this.setNext(null);
    }
```

```java
    private static final int DEFAULT_NUMBER_OF_ITRATION = 5;
    private static final int DEFAULT_FIRST_SIZE = 10000; // 첫 번째 실험 데이터 크기
    private static final int DEFAULT_SIZE_INCREMENT = 10000; // 실험 데이터 크기 증가량

    private int _numberOfIteration; // 반복횟수
    private int _firstSize; // 처음크기
    private int _sizeIncrement; // 증가량
    private Coin[] _data; // 데이터 배열필드
    private MeasuredResult[] _measuredResults; // 실험결과 배열필드 -> 측정결과저장

    // getter / setter 선언
    public int numberOfIteration() {
        return this._numberOfIteration;
    }

    public void setNumberOfIteration(int newNumberOfIteration) {
        this._numberOfIteration = newNumberOfIteration;
    }

    public int firstSize() {
        return _firstSize;
    }

    public void setFirstSize(int newFirstSize) {
        this._firstSize = newFirstSize;
    }

    public int sizeIncrement() {
        return _sizeIncrement;
    }

    public void setSizeIncrement(int newSizeIncrement) {
        this._sizeIncrement = newSizeIncrement;
    }

    public Coin[] data() {
        return this._data;
    }

    public void setData(Coin[] newData) {
        this._data = newData;
    }

    public MeasuredResult[] measuredResults() {
        return this._measuredResults;
    }

    public void setMeasuredResults(MeasuredResult[] newMeasuredResults) {
        this._measuredResults = newMeasuredResults;
    }

    // 실험 데이터의 최대 크기를 계산하여 돌려준다
    public int maxSize() { // maxSize의 합의
        return this.firstSize() + this.sizeIncrement() * (this.numberOfIteration() - 1);
    }

    // 생성자
    public Experiment() {
        this.setNumberOfIteration(DEFAULT_NUMBER_OF_ITRATION); // 반복 5
        this.setFirstSize(DEFAULT_FIRST_SIZE); // 처음크기 10000
        this.setSizeIncrement(DEFAULT_SIZE_INCREMENT); // 증가량 10000
        this.setData(new Coin[this.maxSize()]); // Setter of Data
        this.setMeasuredResults(new MeasuredResult[this.numberOfIteration()]); // Setter of MeasuredResults

    }

    public Experiment(int givenNumberOfIteration, int givenFirstSize, int givenSizeIncrement) {
        this.setNumberOfIteration(givenNumberOfIteration); // 반복횟 givenNumberOfIteration
        this.setFirstSize(givenFirstSize); // 처음크기 givenFirstSize
        this.setSizeIncrement(givenSizeIncrement); // 증가량 givenSizeIncrement
        this.setData(new Coin[this.maxSize()]); // Setter of Data
        this.setMeasuredResults(new MeasuredResult[this.numberOfIteration()]); // Setter of MeasuredResults

    }

    // 값는 측정에 필요한 데이터를 생성한다, 난수사용
    public void generateData() {
        Random random = new Random(); // random 선언
        for (int i = 0; i < this.maxSize(); i++) { // this.maxSize()만큼 반복
            int randomCoinValue = random.nextInt(this.maxSize()); // 0~maxSize()의 난수를 생성 후 randomCoinValue에 대입
            this.data()[i] = new Coin(randomCoinValue); // data의 i번째 배열은 randomCoinValue를 갖는 코인
        }
    }

    public void measureForUnsortedArrayList() {
        @SuppressWarnings("unused")
        Coin maxCoin;

        long durationForAdd, durationForMax;
        long start, stop;

        int dataSize = this.firstSize();
        for (int iteration = 0; iteration < this.numberOfIteration(); iteration++) {
            UnsortedArrayList<Coin> list = new UnsortedArrayList<Coin>(dataSize);
            durationForAdd = 0;
            durationForMax = 0;
            for (int i = 0; i < dataSize; i++) {
                start = System.nanoTime();
                list.add(this.data()[i]);
                stop = System.nanoTime();
                durationForAdd += (stop - start);

                start = System.nanoTime();
                maxCoin = list.max();
                stop = System.nanoTime();
                durationForMax += (stop - start);
            }
            this.measuredResults()[iteration] = new MeasuredResult(dataSize, durationForAdd, durationForMax);
            dataSize += this.sizeIncrement();
        }
    }

    // Sorted Array로 구현한 List의 성능을 측정한다
    public void measureForSortedArrayList() {
        @SuppressWarnings("unused")
        Coin maxCoin;

        long durationForAdd, durationForMax;
        long start, stop;

        int dataSize = this.firstSize();
        for (int iteration = 0; iteration < this.numberOfIteration(); iteration++) {
            SortedArrayList<Coin> list = new SortedArrayList<Coin>(dataSize);
            durationForAdd = 0;
            durationForMax = 0;
            // 생성되어 저장된 난수들을 하나씩 꺼내와 배열에 추가하는 반복문
            for (int i = 0; i < dataSize; i++) {
                start = System.nanoTime();
                list.add(this.data()[i]);
                stop = System.nanoTime();
                durationForAdd += (stop - start);

                start = System.nanoTime();
                maxCoin = list.max();
                stop = System.nanoTime();
                durationForMax += (stop - start);
            }
            // 측정된 결과 각각 저장
            this.measuredResults()[iteration] = new MeasuredResult(dataSize, durationForAdd, durationForMax);
            dataSize += this.sizeIncrement();
        }
    }

    public void measureForSortedLinkedList() {
        @SuppressWarnings("unused")
        Coin maxCoin; // 최대값 Coin 선언

        long durationForAdd, durationForMax; // 합과 최대값 - 결과 값
        long start, stop; // 시작과 종료 - 변수 측정

        int dataSize = this.firstSize(); // dataSize = firstSize = 10000
        for (int iteration = 0; iteration < this.numberOfIteration(); iteration++) { // 0~4까지 반복
            SortedLinkedList<Coin> list = new SortedLinkedList<Coin>(dataSize); // dataSize를 갖는 list생성
            durationForAdd = 0; // 초기값 0
            durationForMax = 0; // 초기값 0
            for (int i = 0; i < dataSize; i++) { // dataSize만큼 반복
                start = System.nanoTime(); // 시작
                list.add(this.data()[i]); // i번째 배열을 list에 add
                stop = System.nanoTime(); // 종료
                durationForAdd += (stop - start); // 종료시점 - 시작시점 을 계속 더해준다.

                start = System.nanoTime(); // 시작
                maxCoin = list.max(); // list에서 maxCoin을 찾기
                stop = System.nanoTime(); // 종료
                durationForMax += (stop - start); // 종료시점 - 시작시점을 계속 더해준다.
            }
            this.measuredResults()[iteration] = new MeasuredResult(dataSize, durationForAdd, durationForMax);
            dataSize += this.sizeIncrement();
        }
    }

    public void measureForUnsortedLinkedList() {
        @SuppressWarnings("unused")
        Coin maxCoin;

        long durationForAdd, durationForMax;
        long start, stop;

        int dataSize = this.firstSize();
        for (int iteration = 0; iteration < this.numberOfIteration(); iteration++) {
            UnsortedLinkedList<Coin> list = new UnsortedLinkedList<Coin>(dataSize);
            durationForAdd = 0;
            durationForMax = 0;
            for (int i = 0; i < dataSize; i++) {
                start = System.nanoTime();
                list.add(this.data()[i]);
                stop = System.nanoTime();
                durationForAdd += (stop - start);

                start = System.nanoTime();
                maxCoin = list.max();
                stop = System.nanoTime();
                durationForMax += (stop - start);
            }
            this.measuredResults()[iteration] = new MeasuredResult(dataSize, durationForAdd, durationForMax);
            dataSize += this.sizeIncrement();
        }
    }
}
```

```
    private int size;
    private long _durationForAdd; // 시간 측정결과는 long
    private long _durationForMax;

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

    public void setSize(int newSize) {
        this.size = newSize;
    }

    public long durationForAdd() {
        return this._durationForAdd;
    }

    public void setDurationForAdd(long newDurationForAdd) {
        this._durationForAdd = newDurationForAdd;
    }

    public long durationForMax() {
        return _durationForMax;
    }

    public void setDurationForMax(long newDurationForMax) {
        this._durationForMax = newDurationForMax;
    }

    public MeasuredResult() {
        this.setSize(0);
        this.setDurationForAdd(0);
        this.setDurationForMax(0);
    }

    public MeasuredResult(int givenSize, long givenDurationForAdd, long givenDurationForMax) {
        this.setSize(givenSize);
        this.setDurationForAdd(givenDurationForAdd);
        this.setDurationForMax(givenDurationForMax);
    }
}
```

◆ 상단에 보고서에 첨부된 소스코드는 따로 캡쳐 하지 않았습니다.

◆ Experiment Class 에 대한 소스코드는 코드를 참고 부탁드립니다.