자료구조 실습 보고서

[제 6주] : 리스트 성능비교

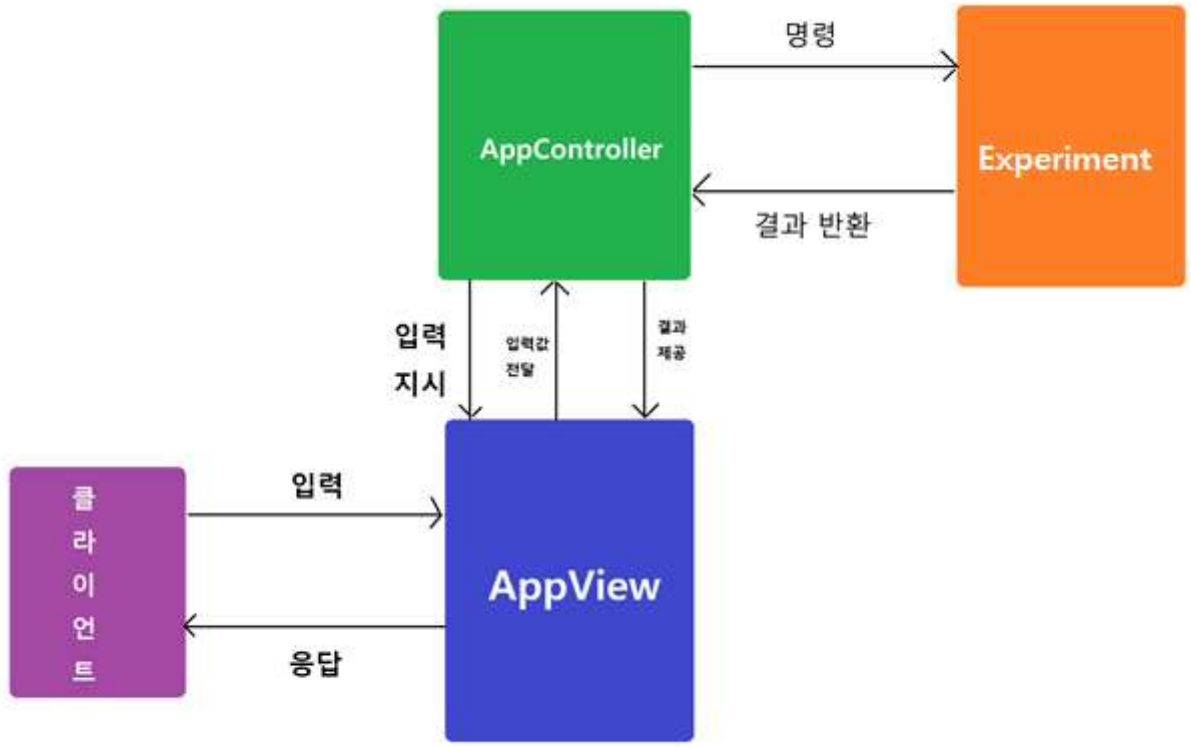
제출일: 2022-04-17(일)

201902708 신동훈

프로그램 설명

1. 프로그램의 전체 설계 구조

간단하게 표현하면 다음과 같다.



MVC(Model - View - Controller)구조를 따르며, 각각의 역할은 다음과 같다.

AppController

AppView에게 출력 정보를 제공하며, 출력을 요청한다.
Experiment에게 여러 List의 성능 측정을 실행하게 한 뒤, 결과를 받아와 AppView에게 전달한다.

AppView

화면(콘솔)에 메시지를 출력하는 역할과, 화면으로부터 입력값을 받아와 다른 객체에게 전달하는 역할을 담당한다.

Experiment

UnsortedArrayList와 SortedArrayList, UnsortedLinkedList, SortedALinkedList 에게 동일한 개수의 랜덤한 Coin 객체를 생성하여 넘겨준 후, add와 max의 성능을 측정하여 반환한다.

2. 함수 설명(주요 알고리즘/ 자료구조의 설명 포함)

2-1. 사용된 자료구조

ArrayList는 내부적으로 배열을 사용하여 구현하였다.
배열은 메모리를 연속적인 공간에 할당하여, 컴파일 시 배열의 타입과 크기가 고정되어 있다는 단점이 있으나, 임의의 인덱스에 대한 접근에는 매우 빠른 속도로 연산이 가능하다는 장점이 있다.
이에 비해 LinkedList는 내부적으로 LinkedNode를 사용하여 구현하였다.
LinkedNode는 데이터와, 다음 순서의 LinkedNode의 주소를 가지고 있으며, 이를 통해 순서를 가지고 자료를 저장한다.
따라서 LinkedNode는 자료의 크기가 한정되어 있지 않고 계속해서 늘어날 수 있으며, 중간 원소의 삽입과 삭제 시에도 단지 주소만 바꾸어주면 되므로 연산이 빠르다.
그러나 임의의 순서의 요소에 대한 접근은 처음 노드부터 순차적으로 탐색하여 접근해야 하기 때문에 오랜 시간이 걸리게 된다.

2-2. 주요 함수

Experiment

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

```
 /**
     * 성능 측정에 필요한 데이터를 생성한다.
     * 난수를 사용하며, 생성한 난수 값을 갖는 Coin 객체를 생성하여 저장한다.
     */
    public void generateData() {
        Random random = new Random() ;
        for (int i = 0 ; i < this.maxSize() ; i++ ) {
            int randomCoinValue = random.nextInt(this.maxSize()) ;
            this.data()[i] = new Coin(randomCoinValue);
        }
    }

    /**
     * 성능 측정 결과를 돌려준다.
     * @return 성능 측정 결과
     */
    public MeasuredResult[] measuredResults() {
        return _measuredResults;
    }

    /**
     * SortedArrayList 의 성능을 측정한다.
     * 이후 결과를 MeasuredResult[]에 담아 저장한다.
     */
    public void measureForSortedArrayList() {

        @SuppressWarnings("unused")
        Coin maxCoin ;//가장 값이 큰 코인을 저장할 변수

        long durationForAdd, durationForMax ;//add 와 max 에 소모된 시간을 저장할 객체
        long start, stop ;//시작 시간과 끝 시간을 저장할 객체

        int dataSize = this.firstSize() ;

        for ( int iteration = 0 ; iteration < this.numberOfIteration() ; iteration ++ ) {

            SortedArrayList<Coin>listOfCoins = new SortedArrayList<Coin>(dataSize) ;
            durationForAdd = 0 ;
            durationForMax = 0 ;
            for ( int i = 0 ; i < dataSize ; i++ ) {
                start = System.nanoTime() ;//add 측정 시작
                listOfCoins.add (this.data()[i]) ;
                stop = System.nanoTime() ;//add 측정 종료
                durationForAdd += (stop -start) ;
                start = System.nanoTime() ;//max 측정 시작
                maxCoin = listOfCoins.max() ;
                stop = System.nanoTime() ;//max 측정 종료
            }
            this.measuredResults()[iteration] = new MeasuredResult(dataSize, durationForAdd, durationForMax)
;
            dataSize += this.sizeIncrement() ;
        }
    }
```

```
/**
 * UnsortedArrayList 의 성능을 측정한다.
 * 이후 결과를 MeasuredResult[]에 담아 저장한다.
 */
public void measureForUnsortedArrayList() {
    Coin maxCoin ;
    long durationForAdd, durationForMax ;
    long start, stop ;

    int dataSize = this.firstSize() ;
    for ( int iteration = 0 ; iteration < this.numberOfIteration() ; iteration ++ ) {
        UnsortedArrayList<Coin>listOfCoins = new UnsortedArrayList<Coin>(dataSize) ;
        durationForAdd = 0 ;
        durationForMax = 0 ;
        for ( int i = 0 ; i < dataSize ; i++ ) {
            start = System.nanoTime() ;//add 측정 시작
            listOfCoins.add (this.data()[i]) ;
            stop = System.nanoTime() ;//add 측정 종료
            durationForAdd += (stop -start) ;
            start = System.nanoTime() ;//max 측정 시작
            maxCoin = listOfCoins.max() ;
            stop = System.nanoTime() ;//max 측정 종료
            durationForMax+= (stop -start) ;
        }
        this.measuredResults()[iteration] = new MeasuredResult(dataSize, durationForAdd, durationForMax);
        dataSize += this.sizeIncrement() ;
    }
}

/**
 * UnsortedLinkedList 의 성능을 측정한다.
 * 이후 결과를 MeasuredResult[]에 담아 저장한다.
 */
public void measureForUnsortedLinkedList() {
    Coin maxCoin ;
    long durationForAdd, durationForMax ;
    long start, stop ;
    int dataSize = this.firstSize() ;
    for ( int iteration = 0 ; iteration < this.numberOfIteration() ; iteration ++ ) {
        UnsortedLinkedList<Coin>listOfCoins = new UnsortedLinkedList<Coin>() ;
        durationForAdd = 0 ;
        durationForMax = 0 ;
        for ( int i = 0 ; i < dataSize ; i++ ) {
            start = System.nanoTime() ;//add 측정 시작
            listOfCoins.add (this.data()[i]) ;
            stop = System.nanoTime();//add 측정 종료
            durationForAdd += (stop -start) ;
            start = System.nanoTime() ;//max 측정 시작
            maxCoin = listOfCoins.max() ;
            stop = System.nanoTime() ;//max 측정 종료
            durationForMax+= (stop -start) ;
        }
        this.measuredResults()[iteration] = new MeasuredResult(dataSize, durationForAdd, durationForMax) ;
        dataSize += this.sizeIncrement() ;
    }
}
```

```
    /**
     * SortedLinkedList 의 성능을 측정한다.
     * 이후 결과를 MeasuredResult[]에 담아 저장한다.
     */
    public void measureForSortedLinkedList() {
        Coin maxCoin ;
        long durationForAdd, durationForMax ;
        long start, stop ;
        int dataSize = this.firstSize() ;
        for ( int iteration = 0 ; iteration < this.numberOfIteration() ; iteration ++ ) {
            SortedLinkedList<Coin>listOfCoins = new SortedLinkedList<Coin>() ;
            durationForAdd = 0 ;
            durationForMax = 0 ;
            for ( int i = 0 ; i < dataSize ; i++ ) {
                start = System.nanoTime() ;//add 측정 시작
                listOfCoins.add (this.data()[i]) ;
                stop = System.nanoTime() ;//add 측정 종료
                durationForAdd += (stop -start) ;
                start = System.nanoTime() ;//max 측정 시작
                maxCoin = listOfCoins.max() ;
                stop = System.nanoTime() ;//max 측정 종료
                durationForMax+= (stop -start) ;
            }
            this.measuredResults()[iteration] = new MeasuredResult(dataSize, durationForAdd, durationForMax)
;
            dataSize += this.sizeIncrement() ;
        }
    }
```

SortedArrayList <E extends Comparable<E>>

```
/**
 * 원소를 오름차순으로 정렬하여 삽입한다.
 * @param anElement 삽입할 원소
 * @return 삽입에 성공하면 true
 */
public boolean add(E anElement){
    if (this.isFull()) {
        return false;
    }

    //이분 탐색 시작
    int start = 0;
    int end = this.size()-1;
    while (start < end){
        int mid = (start + end) >>> 1;

        if (this.elements()[mid].compareTo(anElement) > 0){//중간값이 더 큰 경우
            end = mid -1;
        }else {
            start = mid + 1;
        }

    }//이분 탐색 종료
    int anOrder = end+1;

    this.makeRoomAt(anOrder);
    this.elements()[anOrder] = anElement;
    this.setSize(this.size() +1);
    return true;
}

public E max(){
    if ( this.isEmpty() ) {
        return null ;
    }
    else {
        return this.elements()[this.size()-1] ;//정렬되어 있으므로 가장 마지막 원소를 반환한다.
    }
}
```

UnsortedArrayList <E extends Comparable<E>>

```
/**
 * 배열의 맨 끝에 원소를 삽입한다.
 * @param anElement 주어진 원소
 * @return 삽입에 성공하면 true
 */
public boolean add(E anElement) {
    if (this.isFull()) {
        return false;
    }
    this.elements()[this.size()] = anElement;
    this.setSize(this.size() + 1);
    return true;

}

public E max(){
    if ( this.isEmpty() ) {
        return null ;
    }

    E max = elements()[0];//가장 큰 원소를 저장할 변수
    for (int i = 0; i < size(); i++) {//순차적으로 탐색
        if(elements()[i].compareTo(max) > 0){//max값보다 더 크다면
            max = elements()[i];//max 갱신
        }
    }
    return max;
}
```

SortedLinkedList<E extends Comparable<E>>

```
/**
 * 원소를 오름차순으로 정렬하여 삽입한다.
 * @param anElement 삽입할 원소
 * @return 삽입에 성공하면 true
 */
public boolean add(E anElement) {
    if(this.isFull()) {return false;}
    else {
        LinkedNode<E> nodeForAdd = new LinkedNode<E>(anElement, null);//추가할 데이터를 가진 Node
        if (this.isEmpty()) {
            this.setHead(nodeForAdd);//만약 List가 비어있으면 head로 설정
        }
        else {
            LinkedNode<E> current = this.head();//head를 current로 설정
            LinkedNode<E> previous = null;//이전 노드는 아직 null
            while ( current != null ) {
                if ( current.element().compareTo(anElement) > 0 ) {
                //정렬되어 있으므로 들어온 Coin보다 더 큰 객체가 존재한다면 while문 종료
                    break;
                }
                previous = current;
                current = current.next();
            }
            if ( previous == null ) {//이전 노드가 없다는 것은 current가 head node
                nodeForAdd.setNext(this.head());
                this.setHead(nodeForAdd);
            }
            else {
                previous.setNext(nodeForAdd);
                nodeForAdd.setNext(current);
            }
        }
        this.setSize(this.size() + 1);
        return true;
    }
}

public E max() {
    if (this.isEmpty()) {
        return null;
    }
    else {
        LinkedNode<E> current = this.head();
        LinkedNode<E> previous = null;
        while(current != null) {
            previous = current;
            current = current.next();
        }
        return previous.element();//정렬된 Linked List이므로 가장 마지막 값 반환
    }
}
```

UnsortedLinkedList<E extends Comparable<E>>

```
/**
 * 원소를 삽입한다.
 * @param anElement 삽입할 원소
 * @return 삽입에 성공하면 true
 */
public boolean add(E anElement){
    LinkedNode<E> nodeForAdd = new LinkedNode<>(anElement, null);
    nodeForAdd.setNext(this.head());//추가할 노드의 다음 노드에 현재의 head Node 를 설정한다.
    this.setHead(nodeForAdd);//LinkedList 의 Head Node 를 추가할 노드로 설정해준
    this.setSize(this.size() + 1);
    return true;
}


public E max() {
    if (this.isEmpty()) {
        return null;
    }
    else {
        LinkedNode<E> current = this.head();
        LinkedNode<E> previous = null;

        E max = current.element();//최댓값을 저장할 변수

        while(current != null) {
            previous = current;
            if(current.element().compareTo(max) > 0){
                max = current.element();
            }
            current = current.next();
        }
        return max;
    }
}
```

3. 종합 설명

해당 프로그램은 여러 가지 List의 add와 max에 대한 성능을 비교해주는 프로그램이다.
앱을 실행하기 위해서는 '_DS06_Main_201902708_신동훈' 이라는 이름의 클래스에 속한 main 메서드를 실행시켜주면 된다.

해당 프로그램은 따로 입력받지 않으며, 실행 직후 여러 List의 성능을 측정하여 화면에 보여준다.

프로그램 장단점/특이점

1. 프로그램의 장점

MVC 패턴을 사용하였기에 유지보수성과, 이후 요구사항의 변경 시 코드의 변경의 파급효과가 적을 뿐더러 여러 List 클래스는 제네릭을 사용하여 다양한 객체들을 가방에 담을 수 있음과 동시에 잘못된 객체를 담게되었을 때 런타임이 아닌 컴파일 시예 예외가 발생하여 실수를 방지할 수 있다는 장점이 있다. 또한 제네릭의 상한 제한(extends)을 통해, Comparable을 구현한 객체만 사용할 수 있게끔 제한을 걸어 실수를 방지하는 것도 장점이라 볼 수 있다.

2 프로그램의 단점

여러 리스트의 성능을 측정할 때, 실행되는 운영체제의 환경이 멀티태스킹 환경이다 보니, OS가 CPU 할당시간을 얼마나 할당하는지 등의 여러 요인에 의해 결과가 많이 달라질 수 있다.

실행 결과 분석

1. 입력과 출력

```
<<<리스트 성능 측정 프로그램을 시작합니다.>>>
! 리스트의 구현에 따른 시간의 차이를 알아봅니다: (단위: Micro Second)

<Unsorted Array List>
[크기:10000] 삽입:     2856, 최대값:   239835
[크기:20000] 삽입:     3492, 최대값:   795067
[크기:30000] 삽입:     4338, 최대값:  1512675
[크기:40000] 삽입:     4146, 최대값:  2671150
[크기:50000] 삽입:     2907, 최대값:  4160876

<Sorted Array List>
[크기:10000] 삽입:    90757, 최대값:      948
[크기:20000] 삽입:   374389, 최대값:     1820
[크기:30000] 삽입:   899150, 최대값:     3129
[크기:40000] 삽입:  1690616, 최대값:     1792
[크기:50000] 삽입:  2185332, 최대값:     1809
```

```
<Unsorted Linked List>
[크기:10000] 삽입:     2385, 최대값:   174529
[크기:20000] 삽입:     2796, 최대값:   679563
[크기:30000] 삽입:     4338, 최대값:  1482662
[크기:40000] 삽입:     3060, 최대값:  2656278
[크기:50000] 삽입:     3403, 최대값:  4159296

<Sorted Linked List>
[크기:10000] 삽입:   182433, 최대값:   221991
[크기:20000] 삽입:  1096711, 최대값:  1532325
[크기:30000] 삽입:  2782008, 최대값:  4281466
[크기:40000] 삽입:  5361512, 최대값:  8672525
[크기:50000] 삽입:  7642062, 최대값: 12150474
<<<리스트 성능 측정 프로그램을 종료합니다.>>>
```

2. 결과 분석

UnsortedLinkedList와 UnsortedArrayList는 add를 수행하는 데 걸리는 시간이 매우 짧았고, SortedArrayList와 SortedLinkedList는 많은 시간이 소요되었다.
max는 SortedArrayList에서 가장 빨랐으며, UnsortedLinkedList와 UnsortedArrayList가 비슷했고, SortedLinkedList에서 가장 느렸다.
이에 대한 원인 추측은 아래 '생각해 볼 점'에서 언급하도록 하겠다.

생각해 볼 점에 대한 의견

1. 리스트의 구현에 따른 성능 차이에 대한 결과 분석과 나의 의견

우선 성능 예측을 해보도록 하겠다.

add 메서드의 경우

UnsortedArrayList의 경우, 단순히 배열의 마지막 값에 값을 추가해주는 작업을 수행하기 때문에, 배열의 특성 상 O(1)의 시간이 걸리게 된다고 예측할 수 있다.

SortedArrayList의 경우, add 메서드를 순차 탐색으로 구현하는지, 이분 탐색으로 구현하는지에 따라서 소모되는 시간이 달라진다.
순차 탐색의 경우 배열의 처음부터 순차적으로 탐색하기에, 배열의 길이가 길어지면, 이에 비례해서 시간이 증가한다. 즉 O(n)의 시간 복잡도를 가지게 된다고 예측할 수 있다.
그에 비해 이분 탐색으로 구현할 경우, 탐색에 걸리는 시간 복잡도가 O(log(n))으로 줄어들게 된다.
위 '함수 설명'부분을 보면 이분 탐색 코드를 확인할 수 있다고 예측할 수 있다.
그러나 makeRoomAt 메서드의 시간 복잡도가 O(n)이기 때문에, 결과적으로 탐색을 구현하는 방법과는 상관없이 O(n)의 시간복잡도를 가진다고 할 수 있다.
그러나 시간복잡도는 동일하지만, 실제 실행 속도는 이분탐색으로 구현한 부분이 조금 더 빠를 것이라 예상할 수 있다.

UnsortedLinkedList의 경우, 단순히 맨 앞 노드에 새로운 노드를 추가하며, 새로운 노드의 nextNode로 기존 head 노드의 주소를 가리키도록 설정한 후, 새로운 노드를 head 노드로 설정해주면 끝나기에 O(1)의 시간이 걸린다고 예측할 수 있다.

SortedLinkedList의 경우, 해당 자료구조의 특성 상, 항상 순차 탐색만이 가능하기 때문에 SortedArrayList의 경우처럼 이분 탐색을 사용하여 구현할 수 없다. 따라서 O(n)의 시간 복잡도를 가지며, 메서드를 이용해서 계속 다음 Node를 호출하기 때문에, 단순 순차탐색으로 구현하는 SortedArrayList보다도 더 오랜 시간이 걸릴 것이라고 예측할 수 있다.

리스트별 실행 성능만을 본다면 위 예측이 정확했으나, 하나의 리스트 내부에서의 결과를 보면 조금 의아한 부분이 있다.
그 예시로 UnsortedArrayList를 보자.
이번 실습을 따라 했다면, 데이터의 개수는 1만개, 2만개, 3만개, 4만개, 5만개 순으로 증가하여 각각에 대해 성능을 측정하게 했다.
UnsortedArrayList의 Add는 시간 복잡도가 O(1)이라 했었다.
따라서 데이터의 개수가 1만개인 경우에는 O(1)의 시간 복잡도를 가지는 메서드를 1만번 수행하게 된다. 마찬가지로 데이터의 개수가 2만개의 경우에는 같은 메서드를 2만번 수행하게 된다.
즉 메서드 자체의 시간 복잡도는 데이터 개수에 영향을 받지 않지만, 해당 메서드가 호출되고 실행되는 시간은 데이터 개수의 영향을 받을 것이다.

정말 간단한 예시로 데이터 개수가 1개일 때 add를 하는것과, 천만개일 때 add를 하는 것은 당연하게도 천만개일 때 더 오래 걸릴 것이다.
그런데 실험 결과를 다시 보면,

```
<Unsorted Array List>
[크기:10000] 삽입:     2456, 최대값:    83462
[크기:20000] 삽입:     5996, 최대값:   353837
[크기:30000] 삽입:     4149, 최대값:   572586
[크기:40000] 삽입:     3588, 최대값:   941946
[크기:50000] 삽입:     2499, 최대값:  1514529
```

다음과 같이 데이터의 크기에 비례하여 시간이 증가하지 않는다.

그래서 필자는 다음과 같은 예측을 해보았다.
'데이터의 개수가 적어서 연산 속도에 거의 차이가 없었을 것이다.'
따라서 필자는 데이터의 초기 크기와, 크기 증가량을 1만에서 1000만으로 늘려보았다.

```
<Unsorted Array List>
[크기:10000000] 삽입:   311526,
[크기:20000000] 삽입:   646275,
[크기:30000000] 삽입:  1049638,
[크기:40000000] 삽입:  1228810,
[크기:50000000] 삽입:  1651736,
```

(최대값은 시간이 너무 오래 걸려 이는 주석처리한 후 코드를 수행하였다.)
데이터의 개수를 1000만개씩 늘렸을 경우, 데이터의 수가 2배, 3배 증가함에 따라 연산에 걸리는 시간도 2배, 3배 증가하였다.

위 결과를 통해 데이터의 개수가 적어 연산 속도를 구하는 데 별 차이가 없었을 것이라는 생각이 틀리지 않았다는 것을 알 수 있다.
(추가로 실습자료 ppt의 실험 결과는 데이터의 개수가 적었는데도 불구하고 데이터의 개수에 비례하여 결과가 나왔는데, 이에 대해서는 조금 의아한 부분이 있다.)
또한 데이터의 개수를 1000만개로 늘렸는데도 불구하고 데이터의 크기가 천만인 경우와 4천만인 경우, 실행 시간이 4배에는 근접하지만, 4배보다는 적게 나왔는데, 이는 프로그램의 문제가 아닌 운영체제의 CPU 스케줄러의 CPU 할당하는 방식에 의해, 중간중간 발생하는 컨텍스트 스위치(Context Switch)나 컴퓨터 자체가 받는 부하등의 영향을 받아 측정값들 사이에 오차가 생기는 것으로 예상을 해볼 수 있다. 그러나 확실하게는 알 수 없다.

이어서 max 메서드의 경우를 살펴보겠다.

max 메서드의 경우

UnsortedArrayList의 경우, List가 정렬되어 있지 않기 때문에, 최댓값을 찾기 위해 List를 순차적으로 탐색해야 하며 따라서 소모되는 시간이 데이터의 개수에 비례하는, 즉 O(n)의 시간복잡도를 가진다고 예측할 수 있다.

SortedArrayList의 경우, 이미 정렬되어 있으므로, 최댓값의 인덱스는 List의 size -1이며, 배열의 특성 상 특정 원소를 인덱스를 통해 조회하는 연산은 O(1)의 시간 복잡도를 가지므로, max 메서드 역시 O(1)의 시간 복잡도를 가진다고 예측할 수 있다.

SortedLinkedList의 경우, LinkedNode를 사용하는 자료구조의 특성 상, 항상 순차 탐색만이 가능하기 때문에 SortedArrayList의 경우처럼 이분 탐색을 사용하여 구현할 수 없다. 또한 정렬되어 있기 때문에, 최댓값은 항상 가장 마지막 위치에 존재하며, 따라서 마지막 원소까지 순차적으로 탐색하기 때문에 O(n)의 시간 복잡도를 가지며, 메서드를 이용해서 계속 다음 Node를 호출하기 때문에, max를 단순 순차탐색으로 구현하는 UnsortedArrayList보다도 더 오랜 시간이 걸릴 것이라 예상할 수 있다.

UnsortedLinkedList의 경우에도, LinkedNode를 사용하는 자료구조의 특성 상, 항상 순차 탐색만이 가능하기 때문에 SortedArrayList의 경우처럼 이분 탐색을 사용하여 구현할 수 없다. 또한 무엇이 최댓값인지 정해진 것이 없기 때문에, 항상 List의 끝까지 모든 원소들을 순회하며 각각의 원소들과 현재까지의 최댓값을 계속해서 비교해주고 최댓값을 갱신해야 하기 때문에, 비교하는 연산을 수행하지 않고 가장 마지막 원소까지 순회하는 SortedLinkedList보다 더 오랜 시간이 걸릴 것이라 예상할 수 있다.

결과를 보면 대부분 예상과 일치하지만 SortedLinkedList와 UnsortedLinkedList의 경우 차이가 발생한다.
UnsortedLinkedList가 더 오랜 시간이 걸릴 것이라 예상하였지만, 실제로는 SortedLinkedList가 훨씬 더 오랜 시간이 걸렸다.
그 이유를 한번 분석해 보려고 한다.

우선 원인을 예측해 보려 한다.
SortedLinkedList와 UnsortedLinkedList는 최댓값을 찾기 전 우선 데이터를 삽입하는 과정을 거친다.
그런데 SortedLinkedList는 데이터를 정렬해서 삽입해야 하기 때문에, 이에 걸리는 시간이 UnsortedLinkedList에 비해 훨씬 오랜 시간이 걸리게 된다.
데이터를 정렬하는 데 많은 CPU를 사용하였고, 많은 부하가 걸린 상태에서 곧바로 최댓값을 찾기 위해 데이터를 처음부터 끝까지 순회하기에, 최댓값을 찾는 과정에서 CPU를 적게 할당받는다던가 하는 등의 영향으로 UnsortedLinkedList에 비해 오랜 시간이 걸렸을 것이라 예측해보았다.

예측을 정확하게 검증하기에는 현재 나의 능력으로는 무리가 있다 판단하여, 다른 방법을 사용하여 일반적인 상황에서 max 메서드의 소요시간이 SortedLinkedList보다 UnsortedLinkedList에서 더 오래 걸리는 것을 보이려 한다.

우선 처음으로는 SortedLinkedList의 데이터를 정렬하지 않고 집어넣은 후, 가장 마지막 값을 찾도록 할 것이다.
당연하게도 이는 최댓값이 아닐 가능성이 매우 높다. 그러나 나는 SortedLinkedList의 max 메서드를

```
public E max() {
    if (this.isEmpty()) {
        return null;
    }
    else {
        LinkedNode<E> current = this.head();
        LinkedNode<E> previous = null;
        while(current ≠ null) {
            previous = current;
            current = current.next();
        }
        return previous.element();
    }
}
```

위와 같이 항상 마지막 Node까지 탐색하여 반환하도록 구현하였기에, 리스트가 정렬되었는지의 여부와 max 메서드의 소요시간에는 관계가 없다.
SortedLinkedList의 max는 그대로 두고, 데이터를 정렬하지 않은 상태로 add를 수행한 후 max를 구한 결과는 다음과 같다.

```
<Sorted Linked List>
[크기:10000] 삽입:     2306, 최대값:   102904
[크기:20000] 삽입:     2761, 최대값:   416868
[크기:30000] 삽입:     5160, 최대값:   996298
[크기:40000] 삽입:     3974, 최대값:  1699095
[크기:50000] 삽입:     4209, 최대값:  2703018

<Unsorted Linked List>
[크기:10000] 삽입:     1251, 최대값:   118493
[크기:20000] 삽입:     2622, 최대값:   469446
[크기:30000] 삽입:     4112, 최대값:  1054299
[크기:40000] 삽입:     3786, 최대값:  1890455
[크기:50000] 삽입:     4300, 최대값:  4105528
<<<리스트 성능 측정 프로그램을 종료합니다.>>>
```

(코드의 순서를 조금 바꾸어서 SortedLinkedList가 먼저 실행되도록 하였다.)
삽입의 경우 당연히 정렬을 하지 않고 넣었기에 비슷한 시간이 걸리는 것을 확인할 수 있으며, 최댓값을 구하는 부분을 보면 UnsortedLinkedList의 max는 맨 마지막 원소까지 순회하며 최댓값과 비교하는 연산을 추가로 수행하기 때문에 SortedLinkedList보다 더 오랜 시간이 걸리는 것을 알 수 있다.

다음으로는 UnsortedLinkedList의 데이터를 삽입할 때 정렬을 하며 삽입한 후, max를 구하는 메서드를 그대로 사용하여 시간을 측정해 보도록 하겠다.
위와 같이 실행하는 이유는, 데이터를 정렬하는데 소요되는 메모리 사용률 등이 max값을 구하는 데에도 영향을 미칠 것이라는 예측이 맞는지 확인해보기 위해서이다.
결과는 다음과 같다.

```
<Sorted Linked List>
[크기:10000] 삽입:   185348, 최대값:   222963
[크기:20000] 삽입:  1115475, 최대값:  1506507
[크기:30000] 삽입:  3119190, 최대값:  4629768
[크기:40000] 삽입:  5516286, 최대값:  8624997
[크기:50000] 삽입:  8781174, 최대값: 14254317

<Unsorted Linked List>
[크기:10000] 삽입:   194430, 최대값:   376085
[크기:20000] 삽입:  1146453, 최대값:  2386714
[크기:30000] 삽입:  2923253, 최대값:  6287969
[크기:40000] 삽입:  5509308, 최대값: 12156206
[크기:50000] 삽입:  8911644, 최대값: 20088965
```

분명 UnsortedLinkedList의 max 메서드의 코드는 동일함에도 불구하고 add 메서드의 코드를 수정하였을 뿐인데도 메서드의 실행 시간이 확연히 차이나는 것을 알 수 있다.
즉 이를 통해 '결과 분석'에서의 결과에서 SortedLinkedList의 max에서 많은 시간이 소요되었던 것이 add 메서드를 수행하며, 원소들을 정렬하는 과정에서 발생한 부하등의 영향을 받았다는 것을 알 수 있으며, 외부의 요인이 없는 경우 max 메서드를 수행하는 데 있어 UnsortedLinkedList가 SortedLinkedList보다 조금 더 오랜 시간이 소요된다고 할 수 있다.