자료구조 보고서

[제 03주]

2018.3.29.

201502273/김현종

1. 코드

```
public class Bag {

	protected static final int DEFAULT_MAX_SIZE = 100;
	protected int maxSize;
	protected int size;
	protected int totalValue;

	protected Coin[] coins;

	/*
	 * Bag 클래스의 기본 생성자
	 * 기본 최대 크기를 Bag의 크기로 갖는다.
	 * 현재 크기를 0으로 초기화한다.
	 */
	public Bag() {
		this.maxSize = Bag.DEFAULT_MAX_SIZE;
		coins = new Coin [this.maxSize];
		this.size = 0;
	}

	/*
	 * Bag 클래스의 기본 생성자
	 * 기본 최대 크기를 Bag의 크기로 갖는다.
	 * 현재 크기를 0으로 초기화한다.
	 */
	public Bag(int maxSize) {
		this.maxSize = maxSize;
		coins = new Coin [this.maxSize];
		this.size = 0;
	}

	/*
	 * 현재 Bag이 차있는지를 확인한다.
	 */
	public boolean isFull() {
		return (this.maxSize == this.size);
	}

	/*
	 * 현재 Bag이 비어있는지를 확인한다.
	 */
	public boolean isEmpty() {
		return (this.size == 0);
	}

	/*
	 * 먼저 Bag이 꽉 차있는지 확인한 뒤 차있지 않으면 coins에 새로운 Coin을 할당한다.
	 * 그리고 Bag에 들어있는 총 가격을 coin 값 만큼 올려준다.
	 * 그리고 현재 Coin개수를 1 올린다.
	 */
	public boolean add() {
		if(this.isFull()) {
			System.out.print("\n실패\n");
			return false;
		}else {
			this.coins[this.size] = new Coin();
			this.totalValue += this.coins[this.size].getValue();
			this.size++;
			return true;
		}
	}

	/*
	 * 먼저 Bag이 꽉 차있는지 확인한 뒤 차있지 않으면 coins에 새로운 Coin을 할당한다.
	 * Coin의 가격은 입력된 value값이 된다.
	 * 그리고 Bag에 들어있는 총 가격을 coin 값 만큼 올려준다.
	 * 그리고 현재 Coin개수를 1 올린다.
	 */
	public boolean add(int value) {
```

```
                if(this.isFull()) {
                        System.out.print("\n실패\n");
                        return false;
                }else {
                        this.coins[this.size] = new Coin(value);
                        this.totalValue += this.coins[this.size].getValue();
                        this.size++;
                        return true;
                }
        }

        /*
         * 현재 Bag에 들어있는 Coin의 수를 반환한다.
         */
        public int getSize() {
                return this.size;
        }

        /*
         * 현재 Bag에 들어있는 총 가격을 반환한다.
         */
        public int getTotalValue() {
                return this.totalValue;
        }

        /*
         * Bag이 비어있는지 확인한 후 비어있지 않으면 입력된 값과 같은 가격을 갖는 Coin을
coins에서 찾는다.
         * 찾으면 총 가격을 해당 코인의 가격만큼 빼준다.
         * Bag에 들어있는 코인의 수를 하나 줄인다.
         * 어레이 상에서 맨 뒤에 들어있는 코인을 해당 코인의 자리로 할당하고 맨 뒤 자리는
null을 할당한다.
         */
        public  boolean remove(int value) {
                if(!this.isEmpty()) {
                        for(int i = 0; i < this.size; i++) {
                                if(this.coins[i].getValue() == value) {
                                        this.totalValue -= this.coins[i].getValue();
                                        this.size--;
                                        this.coins[i] = this.coins[this.size];
                                        this.coins[this.size] = null;
                                        System.out.print(value + " 코인이
삭제되었습니다.");
                                        return true;
                                }
                        }
                        System.out.print("\n없습니다\n");
                        return false;

                }
                System.out.print("\n실패\n");
                return false;
        }

        /*
         * 입력된 값과 같은 값을 갖는 코인을 일일히 비교해 찾은 후
         * 찾을때마다 수를 세어 그 수를 반환한다.
         */
        public int frequentCoin(int value) {
                int frequency = 0;
                for(int i = 0; i < this.size; i++) {
                        if(value == this.coins[i].getValue()) {
                                frequency++;
                        }
                }
                System.out.print(value + " 코인은 " + frequency + "개 존재합니다.\n");
                return frequency;
        }

        /*
         * 입력된 값과 같은 값을 갖는 코인이 Bag안에 존재하는지
         * Bag안에 존재하는 모든 코인의 값을 비교하여 확인하는 함수이다.
```

```
	 */
	public boolean doesContain(int value) {
		for(int i = 0; i < this.size; i++) {
			if(value == this.coins[i].getValue()) {
				return true;
			}
		}
		return false;
	}

	/*
	 * Bag의 현재 상태를 출력해주는 함수이다.
	 */
	public void print() {

		System.out.println("총 코인의 개수 : "+ this.size);
		System.out.println("가장 큰 코인 : "+ this.max());
		System.out.println("코인의 합 : "+ this.totalValue);

	}

	/*
	 * Bag의 내부에 있는 모든 코인들의 값을 비교하여 value보다 값이 더 크면 value에
값을 할당한다.
	 * 그리고 반복문이 끝나고 value값을 반환하므로써 가장 큰 값을 찾는다.
	 */
	public int max() {
		int value = Integer.MIN_VALUE;
		for(int i = 0; i < this.size; i++) {
			if(value < this.coins[i].getValue()) {
				value = this.coins[i].getValue();
			}
		}
		return value;
	}

	/*
	 * Bag에 들어있는 총 가격을 반환한다.
	 */
	public int sum() {
		return this.totalValue;
	}
}
```

Bag.java

```
public class LinkedBag {

	private int maxSize;
	protected int curSize;
	protected int totalValue;
	protected Node<Coin> next;


	/*
	 * LinkedBag의 생성자
	 * 인수인 maxSize를 LinkedBag의 최대 크기로 정한다.
	 * init() 메소드를 통해 필요한 값들을 초기화한다.
	 */
	public LinkedBag(int maxSize) {
		this.maxSize = maxSize;
		init();
	}

	/*
	 * 현재 담긴 코인의 개수와 총 가격, next포인터를 초기화한다.
```

```
	 */
	public void init() {
		this.curSize = 0;
		this.totalValue = 0;
		this.next = null;
	}

	/*
	 * 현재 사이즈와 최대 사이즈를 비교하여 둘이 같으면 꽉 찼다고 알려준다.
	 */
	public boolean isFull() {
		return (this.maxSize == this.curSize);
	}

	/*
	 * 현재 사이즈가 0이면 LinkedBag이 비었다고 알려준다.
	 */
	public boolean isEmpty() {
		return (this.curSize == 0);
	}

	/*
	 * 특정 위치에 value라는 값을 갖는 코인을 추가하는 함수이다.
	 * 먼저 꽉 찼는지 확인하고 꽉 차지 않았으면 현재 넣으려고 하는 인덱스가 음수이거나 혹은 현재 들어있는 개수보다 뒤인지 확인한다.
	 * 둘 다 아닐경우 이전 노드와 현재 노드, 두 개의 노드를 가지고 리스트를 탐색하며 index가 가리키는 자리까지 이동한다.
	 * 다 이동하면 새로 넣을 값인 newNode를 리스트 사이에 끼어 넣는다.
	 * prev.setNext(newNode);
	 * if(curr != null)
	 *		newNode.setNext(curr);
	 * 현재 사이즈를 1 올리고 총 가격을 value만큼 올린다.
	 */
	public boolean addAt(int value, int index) {
		if(this.isFull()) {
			System.out.print("\n실패\n");
			return false;
		}else if(index > this.curSize || index < 0){
			System.out.print("\n실패\n");
			return false;
		} else {
			Node<Coin> prev = null;
			Node<Coin> curr = this.next;
			Node<Coin> newNode = new Node<Coin>(new Coin(value));
			for(int i = 0; i < index; i++) {
				prev = curr;
				curr = curr.getNext();
			}
			if(prev == null) {
				this.next = newNode;
			}else {
				prev.setNext(newNode);
				if(curr != null)
					newNode.setNext(curr);
			}
			this.curSize++;
			this.totalValue += value;

			return true;
		}
	}

	/*
	 * value라는 값을 갖는 새로운 코인을 넣는 함수이다.
	 * 먼저 LinkedBag이 꽉 찼는지 확인해보고
	 * 꽉 차지 않았으면 addAt함수를 통해 현재 맨 뒷자리에 코인을 추가한다.
	 */
	public boolean add(int value) {
		if(this.isFull()) {
			System.out.print("\n실패\n");
		}else {
			if(this.addAt(value, this.curSize))
```

```
                                        return true;
                                }
                                return false;
                        }

                        /*
                         * 현재 사이즈를 반환해주는 함수이다.
                         */
                        public int getSize() {
                                return this.curSize;
                        }

                        /*
                         * 총 가격을 반환해주는 함수이다.
                         */
                        public int getTotalValue() {
                                return this.totalValue;
                        }

                        /*
                         * value라는 값을 갖는 코인을 리스트에서 제거해주는 함수이다.
                         * pre와 curr 두 개의 노드를 가지고 리스트를 탐색하고 현재 노드가 가지고 있는
value가 찾던것과 같으면 이전 노드에 next에 현재 노드의 next를 연결해준다.
                         * 그리고 현재 사이즈를 1 줄이고 총 가격도 value만큼 줄인다.
                         */
                        public  boolean remove(int value) {
                                if(!this.isEmpty()) {
                                        Node<Coin> pre = null;
                                        Node<Coin> curr = this.next;
                                        while(curr != null) {
                                                if(curr.getValue().getValue() == value) {
                                                        pre.setNext(curr.getNext());
                                                        this.curSize--;
                                                        this.totalValue -= value;
                                                        return true;
                                                }
                                                pre = curr;
                                                curr = curr.getNext();
                                        }
                                        return false;
                                }
                                System.out.print("\nFull\n");
                                return false;
                        }

                        /*
                         * 모든 코인을 삭제하는 함수이다.
                         * 먼저 코인을 담을 coins라는 Coin 어레이를 만들고 현재 들어있는 사이즈만큼의
크기를 준다.
                         * temp라는 노드로 리스트 전체를 탐색하면서 나오는 코인들을 coins에 순서대로
저장한다.
                         * 그리고 작업이 모두 끝나면 리스트를 다시 초기화하기 위해 init() 메소드를 호출한다.
                         */
                        public Coin[] removeAll() {
                                Coin[] coins = new Coin[this.curSize];
                                Node<Coin> temp = this.next;

                                for(int i = 0; i < this.curSize; i++) {
                                        coins[i] = temp.getValue();
                                        temp = temp.getNext();
                                }

                                init();

                                return coins;
                        }

                        /*
                         * 리스트에 있는 가장 큰 값을 삭제한다.
                         */
                        public boolean removeMax() {
                                return this.remove(this.max());
```

```
	}

	/*
	 * 입력된 값의 코인이 LinkedBag에 몇개나 들어있는지를 알려주는 메소드이다.
	 * frequency라는 변수를 만들고 temp라는 노드를 통해 리스트 전체를 탐색하며
value와 값이 같은 코인이 있을때마다 frequency를 1 올린다.
	 * 그리고 탐색이 끝나면 frequency를 반환한다.
	 */
	public int frequentCoin(int value) {
		int frequency = 0;
		Node<Coin> temp = this.next;
		while(temp != null) {
			if(temp.getValue().getValue() == value) {
				frequency++;
			}
		}
		return frequency;
	}

	/*
	 * temp노드를 사용해 리스트 전체를 탐색하다가 value와 같은 값이 나오면 true를
return한다.
	 * 찾지 못하면 false를 return한다.
	 */
	public boolean doesContain(int value) {
		Node<Coin> temp = this.next;
		while(temp != null) {
			if(temp.getValue().getValue() == value) {
				return true;
			}
		}
		return false;
	}

	/*
	 * next라는 노드가 null이 아니면 해당 노드의 print() 메소드를 호출한다.
	 * null이면 EMPTY를 출력한다.
	 */
	public void print() {
		if(this.next != null)
			this.next.print();
		else
			System.out.println("EMPTY");
	}

	/*
	 * 현재 가방에 들었는 코인들 중 가장 큰 값을 반환한다.
	 * temp라는 노드를 통해 리스트를 탐색하고 현재값(value)보다 더 큰값이 나올때마다
value값을 그 값으로 갱신한다.
	 * 탐색이 다 끝나면 value값을 반환한다.
	 */
	public int max() {
		int value = Integer.MIN_VALUE;
		Node<Coin> temp = this.next;
		while(temp != null) {
			if(temp.getValue().getValue() > value) {
				value = temp.getValue().getValue();
			}
			temp = temp.getNext();
		}
		return value;
	}

	/*
	 * 가방의 크기를 재 조정하는 함수이다.
	 * 새로 정하는 크기를 인수로 받아오며 새로 정하는 사이즈가 현재 사이즈보다
클경우에는 그냥 최댓값을 키운다.
	 * 그렇지 못할경우에는 새로 정하는 사이즈에 있는 노드까지 찾아가 리스트를
끊어버린다.
	 * 그리고 총 가격을 다시 계산하기 위해 totalValue를 0으로 초기화하고 리스트 전체를
탐색해서 모든 코인의 값을 totalValue에 더한다.
	 */
```

```
	public boolean resize(int size) {

		if(this.curSize > size) {
			Node<Coin> temp = this.next;
			for(int i = 1; i < size; i++) {
				temp = temp.getNext();
			}
			temp.setNext(null);
		}

		this.curSize = size;
		this.totalValue = 0;

		Node<Coin> temp = this.next;
		while(temp != null) {
			this.totalValue += temp.getValue().getValue();
			temp = temp.getNext();
		}

		return false;
	}

	/*
	 * 현재 가방에 들어있는 코인의 총 값을 반환한다.
	 */
	public int sum() {
		return this.totalValue;
	}

	/*
	 * 노드 클래스이다.
	 */
	class Node<T> {

		private Node<T> next; //해당 노드 다음으로 오는 노드이다. 없을경우 null;
		private T value; //해당 노드가 가지고 있는 value이다.

		/*
		 * value의 setter이다.
		 */
		public void setValue(T value) {
			this.value = value;
		}

		/*
		 * 기본 생성자로 다음 노드와 값 모두 null로 초기화한다.
		 */
		public Node() {
			this.next = null;
			this.value = null;
		}

		/*
		 * 생성자이다. 받은 인수로 노드의 값을 초기화한다.
		 */
		public Node(T value) {
			this.next = null;
			this.value = value;
		}

		/*
		 * 노드의 값을 반환하는 메소드이다.
		 */
		public T getValue() {
			return this.value;
		}

		/*
		 * 해당 노드 다음에 오는 노드를 설정하는 메소드이다.
		 * 다음에 오는 노드가 null일경우 null을 반환하고 그렇지 않으면 값을
반환한다.
		 */
```

```
		public T setNext(Node<T> node) {

			this.next = node;
			if(node == null) {
				return null;
			}
			return node.getValue();
		}

		/*
		 * 해당 노드 다음에 오는 노드를 반환하는 메소드이다.
		 */
		public Node<T> getNext() {
			return next;
		}

		/*
		 * 노드의 관계를 보여주는 메소드이다. 다음 노드가 null이 아니면 value의
값을 출력해준 뒤 다음 노드의 print() 메소드를 호출한다.
		 * 다음 노드가 null이면 현재 노드의 값을 출력한다.
		 */
		public void print() {
			if( this.next != null) {
				if(this.value.getClass() == Coin.class) {
					System.out.print( ((Coin)this.value).getValue() +
" -> ");
					next.print();
				}
			} else {
				if(this.value.getClass() == Coin.class) {
					System.out.println(
((Coin)this.value).getValue());
				}
			}
		}


	}
}
```

LinkedBag.java

```
public class SortedArrayBag extends Bag {

	/*
	 * SortedArrayBag의 생성자
	 * 부모클래스의 생성자를 호출한다.
	 */
	public SortedArrayBag(int i) {
		// TODO Auto-generated constructor stub
		super(i);
	}

	/*
	 * 삽입할 값을 받아서 순서에 맞게 삽입하는 함수
	 */
	@Override
	public boolean add(int value) {
		if(this.isFull()) {
			System.out.print("\n실패\n");		//꽉 차있으면 삽입 실패
			return false;						//함수 끝,
삽입 실패
		} else if(this.isEmpty()){
			this.coins[0] = new Coin(value); //비어있으면 제일 앞에 삽입
			this.size++;						//현재 길이
하나 증가
			return true;						//함수 끝,
```

```
삽입 성공
                } else {
//꽉 차지도 않고 비어있지도 않으면
                        if(this.coins[0].getValue() > value) {  //0번째 원소의 값이
입력값보다 크면
                                for(int i = 1; i <= this.size; i++)
                                        coins[i] = coins[i-1];
//어레이의 모든 값을 한칸씩 뒤로 미루고
                                coins[0] = new Coin(value);
//0번째 자리에 입력된 값의 코인 삽입
                                this.size++;
        //현재 길이 하나 증가
                                return true;
        //함수 끝, 삽입 성공
                        }
                        for(int i = 1; i < this.size; i++) {     //0번째를 제외한 나머지 부분
비교
                                if(this.coins[i].getValue() > value) {  //해당 값이 입력
값보다 클때
                                        for(int j = i+1; j <= this.size; j++)
                                                coins[j] = coins[j-1];
//어레이의 해당 위치 이후의 값을 한칸씩 뒤로 미루고
                                        coins[i] = new Coin(value);             //해당
자리에 입력된 값의 코인 삽입
                                        this.size++;
        //현재 길이 하나 증가
                                        return true;
        //함수 끝, 삽입 성공
                                }
                        }
                        //입력값이 현재 어레이에 있는 값보다 큰값인 경우
                        this.coins[this.size] = new Coin(value);//맨뒤 자리에 입력된 값의
코인 삽입
                        this.size++;
        //현재 길이 하나 증가
                        return true;
        //함수 끝, 삽입 성공
                }
        }


}
```

SortedArrayBag.java

```
public class MainClass_03_201502273 {

        public static void main(String[] args) {
                // TODO Auto-generated method stub
                PerformanceMeasurement p = new PerformanceMeasurement();

                p.generateData();
                p.testSortedArrayBag();
                p.testSortedLinkedBag();
        }

}
```

MainClass_03_201502273.java

```
public class SortedLinkedBag extends LinkedBag {

	public SortedLinkedBag(int maxSize) {
		super(maxSize);
		// TODO Auto-generated constructor stub
	}

	/*
	 * 삽입할 값을 받아서 순서에 맞게 삽입하는 함수
	 */
	@Override
	public boolean add(int value) {
		// TODO Auto-generated method stub
		if(this.isFull()) {
			System.out.print("\n실패\n");			//꽉 차있으면 삽입 실패
			return false;						//함수 끝, 삽입 실패
		} else if(this.isEmpty()){
			this.next = new Node<Coin>(new Coin(value)); //비어있으면 제일 앞에 삽입
			this.curSize++;						//현재 길이 하나 증가
			return true;						//함수 끝, 삽입 성공
		} else {								//꽉 차지도 않고 비어있지도 않으면
			Node<Coin> curr = this.next;		//현재 노드를 나타내는 노드
			Node<Coin> prev = null;				//이전 노드를 나타내는 노드
			Node<Coin> temp = new Node<Coin>(new Coin(value));//넣을 값이 들어있는 노드

			while(curr != null) {		//현재 노드가 null이 아니면
				if(curr.getValue().getValue() >= value) {	//받은 값보다 현재 값이 크면
					if(prev == null) { //이전값이 null이면, 즉 현개값이 head이면
						temp.setNext(curr); //삽입할 노드에 현재 노드를 연결
						this.next = temp; //head에 삽입할 노드를 할당
					} else {
						temp.setNext(curr);		//삽입할 노드에 현재 노드를 연결
						prev.setNext(temp);		//이전 노드에 삽입할 노드를 연결
					}
					this.curSize++;				//현재 길이 하나 증가
					return true;				//함수 끝, 삽입 성공
				}
				prev = curr;				//현재노드를 이전노드에 할당
				curr = curr.getNext();		//다음노드를 현재노드에 할당
			}
			prev.setNext(temp);			//이전노드에 삽입노드를 연결
			return true;				//함수 끝, 삽입 성공
		}
	}
}
```

SortedLinkedBag.java

```java
public class Coin {

	private static final int DEFAULT_VALUE = 0;
	private int value;

	/*
	 * 코인의 기본 생성자
	 * 코인의 값을 기본값으로 설정한다.
	 */
	public Coin() {
		this.value = Coin.DEFAULT_VALUE;
	}

	/*
	 * 코인의 생성자
	 * 코인의 값을 입력된 값으로 설정한다.
	 */
	public Coin(int value) {
		this.value = value;
	}

	/*
	 * 코인의 값을 반환해주는 메소드
	 */
	public int getValue() {
		return this.value;
	}
}
```

Coin.java

```java
import java.util.Random;

public class PerformanceMeasurement {
	private static final int size = 1000;
	private SortedArrayBag arrBag;
	private SortedLinkedBag linkedBag;
	private Random rand;
	private int[] data;
	private long startTime, endTime, insertingTime, findingMaxTime;

	public PerformanceMeasurement() {

	}

	public void generateData() {
		data = new int[5000];		//랜덤한 수를 위한 공간
		rand = new Random();		//랜덤 객체 생성
		rand.setSeed(System.currentTimeMillis());	//그때그때 다른 랜덤한 값을 받기위해 시드 설정
		for(int i = 0; i < data.length; i++) {
			data[i] = rand.nextInt();			//랜덤한 수를 data에 저장
		}
	}

	public void testSortedArrayBag() {
		System.out.println("{Sorted Array}");

		for(int i = 1; i < 6; i++) {			//i는 1에서 5까지
			this.arrBag = new SortedArrayBag(size * i); //SortedArrayBag 생성
			this.startTime = System.nanoTime(); //시작시간 기록
			for(int j = 0; j < i * size; j++) {
				this.arrBag.add(data[j]);		//data에 있는 값을 어레이 백에 저장
			}
			this.endTime = System.nanoTime(); //종료시간 기록
			this.insertingTime = this.endTime - this.startTime; //어레이 백에 저장하는데 걸린 시간
```

```
                        this.startTime = System.nanoTime(); //시작시간 기록
                        this.arrBag.max();                  //어레이 백에 들은 값 중
최댓값 서치
                        this.endTime = System.nanoTime(); //종료시간 기록
                        this.findingMaxTime = this.endTime - this.startTime; //어레이
백에서 최댓값 서치하는데 걸린 시간

                        System.out.println("Size " + i * size + ",\t insertion " +
this.insertingTime + "\t Search Max " + this.findingMaxTime);
                }
        }

        public void testSortedLinkedBag() {
                System.out.println("{Sorted Linked}");

                for(int i = 1; i < 6; i++) {                //i는 1에서 5까지
                        this.linkedBag = new SortedLinkedBag(size * i);
//SortedLinkedBag 생성
                        this.startTime = System.nanoTime(); //시작시간 기록
                        for(int j = 0; j < i * size; j++) {
                                this.linkedBag.add(data[j]);        //data에 있는 값을
링크드 백에 저장
                        }
                        this.endTime = System.nanoTime(); //종료시간 기록
                        this.insertingTime = this.endTime - this.startTime; //링크드 백에
저장하는데 걸린 시간

                        this.startTime = System.nanoTime(); //시작시간 기록
                        this.linkedBag.max();                       //링크드 백에 들은
값 중 최댓값 서치
                        this.endTime = System.nanoTime(); //종료시간 기록
                        this.findingMaxTime = this.endTime - this.startTime; //링크드
백에서 최댓값 서치하는데 걸린 시간

                        System.out.println("Size " + i * size + ",\t insertion " +
this.insertingTime + "\t Search Max " + this.findingMaxTime);
                }
        }
}
```

PerformanceMeasurement.java

2. 결과

```
{Sorted Array}
Size 1000,        insertion 2411176       Search Max 31928
Size 2000,        insertion 5784713       Search Max 55724
Size 3000,        insertion 22091250      Search Max 92170
Size 4000,        insertion 3306971       Search Max 106929
Size 5000,        insertion 5047355       Search Max 132833
{Sorted Linked}
Size 1000,        insertion 2786482       Search Max 122893
Size 2000,        insertion 7469675       Search Max 240666
Size 3000,        insertion 15612250      Search Max 360246
Size 4000,        insertion 15793578      Search Max 480729
Size 5000,        insertion 26194014      Search Max 607840
```

{Sorted Array} 밑에 있는 것은 각각 사이즈 1000, 2000, 3000, 4000, 5000의 랜덤하게 생성된 값들을 가지고 크기순으로 SortedArrayBag에 삽입하는데 걸린 시간을 ns단위로 insertion에 표기하고 Bag내부의 값들 중 최댓값을 찾는데 걸린 시간을 ns단위로 Search Max에 표기한 것이다.
{Sorted Linked} 밑에 표시된 것들은 위와 같은 것을 SortedLinkedBag으로 수행한 결과이다.

깨달은 점 및 결론

최댓값을 찾는 연산에서 어레이의 경우가 LinkedList의 경우보다 훨씬 빠른 것을 볼 수 있고 삽입시에 걸린 시간의 경우 어레이는 대체적으로 빠르지만 중간에 사이즈 3000에서 걸린 시간이 급격하게 늘어난 것을 볼 수 있었다. 이는 새로 넣는 값이 들어갈 자리를 찾은 후 그 뒤의 모든 값을 한 칸씩 뒤로 보내는 연산 때문에 랜덤 값에 따라 차이가 많이 날 수 있기 때문으로 보인다. 또한 LinkedList의 경우 삽입시의 시간이 삽입 양에 따라 증가하는 것을 볼 수 있었다.
SortedArrayBag의 경우 값을 넣을 때 중첩 for문이 있지만 if문으로 나뉘어져 한번만 실행되므로 O(n)이라고 볼 수 있고 SortedLinkedBag의 경우에도 while문 하나로 이루어져있기 때문에 O(n)으로 볼 수 있다.