CS4092 – All Classes

```
public interface Deque<EltType> {

	public int size();
	public boolean isEmpty();
	public void insertFirst(EltType e);
	public void insertLast(EltType e);
	public EltType removeFirst();
	public EltType removeLast();

}
```

```
public class DequeArray<EltType> implements Deque<EltType> {

	// Declarations
	private static final int INIT_CAPACITY = 100;
	private int capacity;
	private int numEntries;
	private EltType[] entries;


	public DequeArray() {
	capacity = INIT_CAPACITY;
	entries = (EltType[]) (new Object[capacity]);
	numEntries = 0;
  }


	public int size() {
		return numEntries;
	}

	public boolean isEmpty() {
		return (numEntries == 0);
	}



	public static void main(String args[]) {

		DequeArray<Integer> map = new DequeArray<Integer>();

		map.insertFirst(1);
		map.insertFirst(2);
		map.insertFirst(3);
		map.insertLast(4);
		System.out.println(map.removeFirst());
		System.out.println(map.removeLast());

		System.out.println(map.size());
		System.out.println(map.isEmpty());

		map.showMap();
	}
```

```
public void insertFirst(EltType newElement) {
		expandIfNecessary();

		for(int i = numEntries - 1; i >= 0; i--) {
			entries[i+1] = entries[i];
		}

		entries[0] = newElement;
		numEntries++;
}



public void insertLast(EltType newElement) {
		expandIfNecessary();

		entries[numEntries] = newElement;
		numEntries++;
}



public EltType removeFirst() {

		EltType removedEntry = entries[0];

		for(int i = 0; i < numEntries; i++) {
			entries[i] = entries[i+1];
		}


		numEntries--;
		return removedEntry;
}



public EltType removeLast() {

		EltType removedEntry = entries[numEntries-1];
		numEntries--;
		return removedEntry;

}



/************************
* Helper Methods
************************/
public void showMap() {
		System.out.println("\n****Start Map Structure****");


		// loop through map
		for(int i = 0; i < numEntries; i++) {

			System.out.println("element at position " + i + ": value = " + entries[i]);


		}
```

```
                System.out.println("****End Map Structure****");
        }


        private void expandIfNecessary() {
                if (size() == capacity)  {
                        // copy array into one of larger size
                        EltType temp[] = (EltType[])(new Object[2*capacity]);
                        for (int i = 0; i < capacity; i++) {
                                temp[i] = entries[i];
                                entries = temp;
                                capacity = 2*capacity;
                        }
                }
        }


        private void flagError(String errmsg) {
        System.out.println("LinkedStack: "+errmsg);
    System.exit(1);
  }
}
```

```
import helpers.LLNode;


public class DequeDLinkedList<EltType> implements Deque<EltType> {

        // Declarations
        private int size;
        private LLNode<EltType> head;
        private LLNode<EltType> tail;


        public DequeDLinkedList() {
        size = 0;
        head = new LLNode<EltType>(null, null, null);
        tail = new LLNode<EltType>(head, null, null);
        head.setNext(tail);
  }


        public int size() {
                return size;
        }

        public boolean isEmpty() {
                return size == 0;
        }




        public static void main(String args[]) {

                /*DequeLinkedList<Integer> map = new DequeLinkedList<Integer>();*/
                DequeDLinkedList<String> map = new DequeDLinkedList<String>();
```

```
		// add elements to map, where one element is duplicate of previous to test put method
		/*map.insertFirst(0);
		map.insertFirst(1);
		map.insertFirst(2);
		map.insertFirst(3);
		map.insertFirst(4);
		map.insertLast(7);
		map.insertLast(8);
		System.out.println("RemoveFirst: " + map.removeFirst());
		System.out.println("RemoveLast: " + map.removeLast());*/
		map.insertFirst("one");
		//map.insertFirst("two");
		//map.insertFirst("three");
		//map.insertFirst("three");
		//map.insertFirst("four");
		map.insertLast("zero");
		//map.insertLast("eight");
		//System.out.println("RemoveFirst: " + map.removeFirst());
		//System.out.println("RemoveLast: " + map.removeLast());


		map.showMap();
	}



	public void insertFirst(EltType newElement) {

		LLNode<EltType> newFirst;
		LLNode<EltType> oldFirst;


		oldFirst = head.getNext();
		newFirst = new LLNode<EltType>(head, oldFirst, newElement);


		head.setNext(newFirst);
		oldFirst.setPrev(newFirst);


		size++;
	}



	public void insertLast(EltType newElement) {

		LLNode<EltType> newLast;
		LLNode<EltType> oldLast;


		oldLast = tail.getPrev();
		newLast = new LLNode<EltType>(oldLast, tail, newElement);


		tail.setPrev(newLast);
		oldLast.setNext(newLast);


		size++;
	}
```

```
	public EltType removeFirst() {

		if (isEmpty()) {
		flagError("illegal operation: Deque empty");
	}


		LLNode<EltType> oldFirst;
		oldFirst = head.getNext();


		head.setNext(oldFirst.getNext());
		oldFirst.getNext().setPrev(head);


		size--;
		return oldFirst.getElement();
	}



	public EltType removeLast() {

		if (isEmpty()) {
		flagError("illegal operation: Deque empty");
	}


		LLNode<EltType> oldLast;


		oldLast = tail.getPrev();
		tail.setPrev(oldLast.getPrev());
		oldLast.getPrev().setNext(tail);


		size--;
		return oldLast.getElement();
	}



/************************
  * Helper Methods
  ************************/
	public void showMap() {

		// check head and tail
		System.out.println("Headnext: " + head.getNext().getElement());
		System.out.println("Tailprev: " + tail.getPrev().getElement());


		System.out.println("\n****Start Map Structure****");
		System.out.println("head: " + head.getElement());


		// get starting node
		LLNode<EltType> currentElement;
		currentElement = head.getNext();
```

```
                // loop through map
                for(int i = 0; i < size; i++) {

                        System.out.println("element at position " + i + ": value = " + currentElement.getElement());

                        // go to next node
                        currentElement = currentElement.getNext();

                }

                System.out.println("tail: " + tail.getElement());
                System.out.println("****End Map Structure****");
        }


        private void flagError(String errmsg) {
        System.out.println("LinkedStack: "+errmsg);
    System.exit(1);
  }

}
```

```
import helpers.LLNode;


public class DequeLinkedList<EltType> implements Deque<EltType> {

        // Declarations
        private int size;
        private LLNode<EltType> head;
        private LLNode<EltType> tail;



        public DequeLinkedList() {
        size = 0;
        head = null;
        tail = null;
  }


        public int size() {
                return size;
        }

        public boolean isEmpty() {
                return size == 0;
        }




        public static void main(String args[]) {

                /*DequeLinkedList<Integer> map = new DequeLinkedList<Integer>();*/
                DequeLinkedList<String> map = new DequeLinkedList<String>();


                // add elements to map, where one element is duplicate of previous to test put method
                /*map.insertFirst(0);
                map.insertFirst(1);
```

```
		map.insertFirst(2);
		map.insertFirst(3);
		map.insertFirst(4);
		map.insertLast(7);
		map.insertLast(8);
		System.out.println("RemoveFirst: " + map.removeFirst());
		System.out.println("RemoveLast: " + map.removeLast());*/
		map.insertFirst("zero");
		map.insertFirst("one");
		map.insertFirst("two");
		map.insertFirst("three");
		map.insertFirst("four");
		map.insertLast("seven");
		map.insertLast("eight");
		//map.insertLast("eight");
		//System.out.println("RemoveFirst: " + map.removeFirst());
		System.out.println("RemoveLast: " + map.removeLast());


		map.showMap();
	}



	public void insertFirst(EltType e) {


		LLNode<EltType> newFirst = new LLNode<EltType>(head, e);

		head = newFirst;
		if(size == 0) {
			tail = newFirst;
		}


		size++;
	}



	public void insertLast(EltType e) {


		LLNode<EltType> newLast = new LLNode<EltType>(null, e);


		if(size == 0) {
			tail = newLast;
			head = newLast;
		}
		else {
			tail.setNext(newLast);
			tail = newLast;
		}


		size++;
	}



	public EltType removeFirst() {
```

```
                if (isEmpty()) {
                flagError("illegal operation: Deque empty");
        }


                LLNode<EltType> oldFirst = head;
                head = head.getNext();


                size--;
                if(isEmpty()) {
                        head = null;
                        tail = null;
                }


                return oldFirst.getElement();
        }



        public EltType removeLast() {

                if (isEmpty()) {
                flagError("illegal operation: Deque empty");
        }


                LLNode<EltType> oldLast = tail;
                tail = nodeAtIndex(size-2);


                size--;
                if(isEmpty()) {
                        head = null;
                        tail = null;
                }


                return oldLast.getElement();
        }



/************************
  * Helper Methods
  ************************/
        public void showMap() {

                // check head and tail
                //System.out.println("Headnext: " + head.getNext().getElement());
                //System.out.println("Tailprev: " + tail.getPrev().getElement());


                System.out.println("\n****Start Map Structure****");
                System.out.println("head: " + head.getElement());

                // get starting node
                head.getElement();
```

```
                        // loop through map
                        for(int i = 0; i < size; i++) {

                                System.out.println("element at position " + i + ": value = " + head.getElement());

                                // go to next node
                                head = head.getNext();

                        }

                        System.out.println("tail: " + tail.getElement());
                        System.out.println("****End Map Structure****");
                }


                private LLNode<EltType> nodeAtIndex(int index) {
                LLNode<EltType> node = head;

                for (int i = 0; i < index; i++) {
                        node = node.getNext();
                }
                return node;
          }


                private void flagError(String errmsg) {
                System.out.println("LinkedStack: "+errmsg);
             System.exit(1);
          }

        }
```

```
public interface List<EltType> {

   public int size();
   public boolean isEmpty();
   public EltType get(int inx);
   public EltType set(int inx, EltType newElt);
   public void add(EltType newElt);
   public void add(int inx, EltType newElt);
   public EltType remove(int inx);

}
```

```
import helpers.IntegerComparator;



public class ListArray<EltType> implements List<EltType> {

                private static final int INIT_CAP = 100;
                private int capacity;
                protected EltType entries[];
                protected int numEntries;


                public ListArray() {
                        numEntries = 0;
                        capacity =  INIT_CAP;
                        entries = (EltType[]) (new Object[capacity]);
                }
```

```
public int size() {
	return numEntries;
}

public boolean isEmpty() {
	return (numEntries == 0);
}



public static void main(String args[]) {

	ListArray<Integer> map = new ListArray<Integer>();

	map.add(1);
	map.add(2);
	map.add(1, 3);
	System.out.println(map.remove(1));

	System.out.println(map.get(1));

	map.showMap();
}


public EltType get(int inx) {
	checkIndex(inx);
	return entries[inx];
}



public EltType set(int inx, EltType newElt) {
	checkIndex(inx);

	EltType oldElt = entries[inx];
	entries[inx] = newElt;
	return oldElt;
}



public void add(EltType newElt) {
	expandIfNecessary();
	entries[numEntries] = newElt;
	numEntries++;
}



public void add(int inx, EltType newElt) {
	checkIndex(inx);     // so not adding newElt too far ahead in array (no gaps)
	expandIfNecessary();

	for (int i = numEntries-1; i >= inx; i--) {
		entries[i+1] = entries[i];
	}

	entries[inx] = newElt;
	numEntries++;
}
```

```
	public EltType remove(int inx) {
		checkIndex(inx);
		EltType retElt = entries[inx];

		entries[inx] = entries[numEntries-1];

		numEntries--;
		return retElt;
	}



/************************
  * Helper Methods
  ************************/
	public void showMap() {
		System.out.println("\n****Start Map Structure****");


		// loop through map
		for(int i = 0; i < numEntries; i++) {

			System.out.println("element at position " + i + ": value = " + entries[i]);


		}


		System.out.println("****End Map Structure****");
	}


	private void checkIndex(int inx) {
		if ((inx < 0) || (inx >= size())) {
			flagError("index out of bounds");
		}
	}


	private void expandIfNecessary() {
		if (size() == capacity) {
			EltType temp[] = (EltType[])(new Object[2*capacity]);


			for (int i = 0; i < capacity; i++) {
				temp[i] = entries[i];
			}
			entries = temp;
			capacity = 2*capacity;
		}
	}


	private void flagError(String errmsg) {
		System.out.println("ArrayBasedList: "+errmsg);
		System.exit(1);
	}
}
```

```
import helpers.LLNode;


public class ListDLinkedList<EltType> implements List<EltType> {

	// Declarations
	protected int size;
  protected LLNode<EltType> head;
  protected LLNode<EltType> tail;


  public ListDLinkedList() {
	size = 0;
	head = new LLNode<EltType>(null, null, null);
	tail = new LLNode<EltType>(head, null, null);
	head.setNext(tail);
  }


  public int size() {
	return size;
  }

  public boolean isEmpty() {
	return (size == 0);
  }



  public static void main(String args[]) {

	ListDLinkedList<Integer> map = new ListDLinkedList<Integer>();
		/*ListDLinkedList<String> map = new ListDLinkedList<String>();*/


		// add elements to map
		map.add(1);
		map.add(2);
		map.add(3);
		map.add(0, 0);
		//System.out.println(map.get(1));
		//System.out.println(map.set(2, 4));
		System.out.println(map.remove(0));


		map.showMap();
	}




  public EltType get(int inx) {
	checkIndex(inx);
	LLNode<EltType> node = nodeAtIndex(inx);
	return node.element();
  }



  public EltType set(int inx, EltType newElt) {
	checkIndex(inx);
	EltType temp = remove(inx);
	add(inx, newElt);
```

```
        return temp;
}



public void add(EltType newElt) {

        LLNode<EltType> oldLast = tail.getPrev();
        LLNode<EltType> newLast = new LLNode<EltType>(oldLast, tail,  newElt);


        oldLast.setNext(newLast);
        tail.setPrev(newLast);


        size++;
}



public void add(int inx, EltType newElt) {
        if (inx == size()) {
                add(newElt);
        }
        else {
                checkIndex(inx);


                LLNode<EltType> next = nodeAtIndex(inx);
                LLNode<EltType> prev = next.getPrev();
                LLNode<EltType> newNode = new LLNode<EltType>(prev, next, newElt);


                next.setPrev(newNode);
                prev.setNext(newNode);


                size++;
        }
}



public EltType remove(int inx) {

        checkIndex(inx);


        LLNode<EltType> node = nodeAtIndex(inx);
        LLNode<EltType> next = node.getNext();
        LLNode<EltType> prev = node.getPrev();


        prev.setNext(next);
        next.setPrev(prev);


        size--;
        return node.element();
}
```

```
  /************************
   * Helper Methods
   ************************/
        public void showMap() {

                // check head and tail
                System.out.println("Headnext: " + head.getNext().getElement());
                System.out.println("Tailprev: " + tail.getPrev().getElement());

                System.out.println("\n****Start Map Structure****");
                System.out.println("head: " + head.getElement());


                // get starting node
                LLNode<EltType> currentElement;
                currentElement = head.getNext();


                // loop through map
                for(int i = 0; i < size(); i++) {

                        System.out.println("element at position " + i + ": value = " + currentElement.getElement());

                        // go to next node
                        currentElement = currentElement.getNext();

                }

                System.out.println("tail: " + tail.getElement());
                System.out.println("****End Map Structure****");
        }



  private void checkIndex(int index) {
        if (index < 0 || index >= size) {
                flagError("Rank " +index+ " is invalid for this sequence of " + size+ " elements.");
        }
  }


  private LLNode<EltType> nodeAtIndex(int index) {
        checkIndex(index);
        LLNode<EltType> node = head.getNext();

        for (int i = 0; i < index; i++) {
                node = node.getNext();
        }
        return node;
  }


  private void flagError(String errmsg) {
        System.out.println("LinkedList: "+errmsg);
        System.exit(1);
  }


}
```

```
import helpers.LLNode;
```

```
public class ListLinkedList<EltType> implements List<EltType> {

	// Declarations
	protected int size;
  protected LLNode<EltType> head;


  public ListLinkedList() {
	size = 0;
	head = null;
  }


  public int size() {
	return size;
  }

  public boolean isEmpty() {
	return (size == 0);
  }



  public static void main(String args[]) {

		ListLinkedList<Integer> map = new ListLinkedList<Integer>();
		/*ListLinkedList<String> map = new ListLinkedList<String>();*/


		// add elements to map
		map.add(1);
		map.add(2);
		map.add(3);
		map.add(0, 0);
		map.add(2, 22);
		//System.out.println(map.remove(3));
		//System.out.println(map.get(2));
		//System.out.println(map.set(0, 1));


		map.showMap();
	}



  public EltType get(int inx) {
	checkIndex(inx);
	LLNode<EltType> node = nodeAtIndex(inx);
	return node.element();
  }



  public EltType set(int inx, EltType newElt) {
	checkIndex(inx);
	EltType temp = remove(inx);
	add(inx, newElt);
	return temp;
  }
```

```
public void add(EltType newElt) {


        LLNode<EltType> newLast = new LLNode<EltType>(null, newElt);


        if( isEmpty() ) {

        head = newLast;
        }
        else {

                LLNode<EltType> oldLast = nodeAtIndex(size-1);
                oldLast.setNext(newLast);

        }

        size++;
}



public void add(int inx, EltType newElt) {



        if (inx == size()) {  // add at end of list
                add(newElt);
        }
        else if (inx == 0) {  // add at head of list

                LLNode<EltType> newFirst = new LLNode<EltType>(head, newElt);


                head = newFirst;


                size++;
        }
        else {

                checkIndex(inx);


                LLNode<EltType> prev = nodeAtIndex(inx-1);
                LLNode<EltType> next = nodeAtIndex(inx);
                LLNode<EltType> newNode = new LLNode<EltType>(next, newElt);


                prev.setNext(newNode);


                size++;
        }
}



public EltType remove(int inx) {

        checkIndex(inx);
        LLNode<EltType> oldNode = nodeAtIndex(inx);
```

```
        if (inx == 0) { // remove at head of list
                head = head.getNext();
        }
        else {
                LLNode<EltType> prev = nodeAtIndex(inx-1);
                LLNode<EltType> next = oldNode.getNext();


                prev.setNext(next);
        }

        size--;

        return oldNode.element();
}



/************************
 * Helper Methods
 ************************/
        public void showMap() {

                // check head and tail
                //System.out.println("Headnext: " + head.getNext().getElement());
                //System.out.println("Tailprev: " + tail.getPrev().getElement());

                System.out.println("\n****Start Map Structure****");
                System.out.println("head: " + head.getElement());


                // get starting node
                LLNode<EltType> currentElement;
                currentElement = head;


                // loop through map
                for(int i = 0; i < size(); i++) {

                        System.out.println("element at position " + i + ": value = " + currentElement.getElement());

                        // go to next node
                        currentElement = currentElement.getNext();

                }

                //System.out.println("tail: " + tail.getElement());
                System.out.println("****End Map Structure****");
        }


private void checkIndex(int index) {
        if (index < 0 || index >= size) {
                flagError("Rank " +index+ " is invalid for this sequence of " + size+ " elements.");
        }
}


private LLNode<EltType> nodeAtIndex(int index) {
        checkIndex(index);
        LLNode<EltType> node = head;
```

```
		for (int i = 0; i < index; i++) {
			node = node.getNext();
		}
		return node;
	}


	private void flagError(String errmsg) {
		System.out.println("LinkedList: "+errmsg);
		System.exit(1);
	}


}
```

```
public interface Map<KeyType, ValueType> {

		public int size();
		public boolean isEmpty();
		public ValueType get(KeyType k);
		public ValueType put(KeyType k, ValueType e);
		public ValueType remove(KeyType k);

}
```

```
import helpers.Comparator;
import helpers.IntegerComparator;
import helpers.MapEntry;


public class MapArray<KeyType, ValueType> implements Map<KeyType, ValueType> {

		protected static final int INIT_CAPACITY = 100;			/* the initial capacity of the map */
		protected int capacity ;					/* the current capacity of the map */
		protected int numEntries;
		protected MapEntry<KeyType, ValueType>[] entries;
		protected Comparator<KeyType> comparator;			/* comparator that defines equality
between keys */
		protected static final int NO_SUCH_KEY = -1;			/* value denoting unsuccessful search */



		public MapArray(Comparator<KeyType> myComparator) {
			entries = new MapEntry[INIT_CAPACITY];
			capacity = INIT_CAPACITY;
			comparator = myComparator;
			numEntries = 0;
		}


		public int size() {
			return numEntries;
		}


		public boolean isEmpty() {
			return (numEntries == 0);
		}
```

```
	public ValueType get(KeyType k) {
		int indexWithKey = findEntry(k);

		if (indexWithKey != NO_SUCH_KEY) {
			return (entries[indexWithKey]).getValue();
		}
		return null;
	}



	public ValueType put(KeyType k, ValueType e)  {
		MapEntry<KeyType, ValueType> newEntry = new MapEntry<KeyType, ValueType>(k, e);
		int indexWithKey = findEntry(k);

		if (indexWithKey != NO_SUCH_KEY) {
			ValueType oldVal = entries[indexWithKey].getValue();
			entries[indexWithKey] = newEntry;
			return oldVal;
		}
		else {
			expandIfNecessary();
			entries[numEntries++] = newEntry;
		}
		return null;
	}



	public ValueType remove(KeyType k) {
		int indexWithKey = findEntry(k);

		if (indexWithKey != NO_SUCH_KEY) {
			MapEntry<KeyType, ValueType> removedEntry = entries[indexWithKey];
			entries[indexWithKey] = entries[numEntries-1];
			numEntries--;
			return removedEntry.getValue();
		}
		return null;
	}



/************************
  * Helper Methods
  ************************/
	private int findEntry(KeyType key) {
		for (int i = 0; i < numEntries; i++) {
			if ( comparator.compare(key, entries[i].getKey()) == 0 ) {
				return i;
			}
		}
		return NO_SUCH_KEY;
	}


	protected void expandIfNecessary() {
		if (numEntries == capacity) {
			int newCapacity = 2*capacity;
			MapEntry<KeyType, ValueType>[] temp =  new MapEntry[newCapacity];
```

```
                                for (int i = 0; i < capacity; i++) {
                                        temp[i] = entries[i];
                                }
                                entries = temp;
                                capacity = newCapacity;
                        }
                }


}
```

```
import helpers.Comparator;
import helpers.IntegerComparator;
import helpers.LLNode;
import helpers.MapEntry;


public class MapDLinkedList<KeyType, ValueType> implements Map<KeyType, ValueType> {


        /* Declarations */
  private int size;
        private ValueType entry;
  private Comparator<KeyType> comparator;
        private LLNode<MapEntry<KeyType, ValueType>> head;
  private LLNode<MapEntry<KeyType, ValueType>> tail;
        private LLNode<MapEntry<KeyType, ValueType>> node;



  public MapDLinkedList(Comparator<KeyType> myComparator) {
        comparator = myComparator;
        size = 0;
        head = new LLNode<MapEntry<KeyType, ValueType>>(null, null, null);
        tail = new LLNode<MapEntry<KeyType, ValueType>>(head, null, null);
        head.setNext(tail);
  }



        public int size() {
                return size;
        }

        public boolean isEmpty() {
                return (size == 0);
        }



  public static void main(String args[]) {

                IntegerComparator myComparator = new IntegerComparator();
                MapDLinkedList<Integer, Integer> map = new MapDLinkedList<Integer, Integer>(myComparator);


                // add elements to map, where one element is duplicate of previous to test put method
                map.put(1, 11);
                map.put(2, 22);
                map.put(3, 33);
                map.put(4, 44);
                System.out.println(map.put(4, 44));
                System.out.println("old value: " + map.put(1, 12));
```

```
		System.out.println("mapGet: " + map.get(1));
		map.remove(3);


		map.showMap();
	}



	public ValueType get(KeyType k) {
		node = head.getNext();

		for(int i = 0; i < size; i++) {

			if( comparator.compare( node.getElement().getKey(), k ) == 0 ) {

				return node.getElement().getValue();
			}
			node = node.getNext();
		}
		return null;
	}



	public ValueType put(KeyType k, ValueType e) {

		MapEntry newEntry = new MapEntry(k, e);
		LLNode<MapEntry<KeyType, ValueType>> oldFirst;
		LLNode<MapEntry<KeyType, ValueType>> newFirst;


		if(get(k) == null) {

			oldFirst = head.getNext();
			newFirst = new LLNode<MapEntry<KeyType, ValueType>>(head, oldFirst, newEntry);


			oldFirst.setPrev(newFirst);
			head.setNext(newFirst);


			size++;
		}
		else {
			node = head.getNext();

			for(int i = 0; i < size; i++) {

				if( comparator.compare( node.getElement().getKey(), k ) == 0 ) {

					entry = node.getElement().getValue();
					node.getElement().setValue(e);
					return entry;


				}
				node = node.getNext();
			}
		}
		return null;
```

```
	}



	public ValueType remove(KeyType k) {
		node = head.getNext();

		for(int i = 0; i < size; i++) {

			if( comparator.compare( node.getElement().getKey(), k ) == 0 ) {

				LLNode<MapEntry<KeyType, ValueType>> next = node.getNext();
				LLNode<MapEntry<KeyType, ValueType>> prev = node.getPrev();
				prev.setNext(next);
				next.setPrev(prev);
				size--;


				return node.getElement().getValue();
			}
			node = node.getNext();
		}
		return null;
	}



/************************
  * Helper Methods
  ************************/
	public void showMap() {


		System.out.println("\n****Start Map Structure****");
		System.out.println("Headnext: " + ((MapEntry<KeyType, ValueType>)
		head.getNext().getElement()).getKey());

		// get starting node
		node = head.getNext();


		// loop through map
		for(int i = 0; i < size; i++) {

			System.out.println("element at position " + i + ": key = " + node.getElement().getKey()
				+ ", value = " + node.getElement().getValue());

			// go to next node
			node = node.getNext();

		}


		System.out.println("Tailprev: " + ((MapEntry<KeyType, ValueType>)
			tail.getPrev().getElement()).getKey());
		System.out.println("****End Map Structure****");
	}



}
```

```
import helpers.Comparator;
import helpers.IntegerComparator;
import helpers.LLNode;
import helpers.MapEntry;


public class MapLinkedList<KeyType, ValueType> implements Map<KeyType, ValueType> {

	/* Declarations */
  private int size;
	private ValueType entry;
  private Comparator<KeyType> comparator;
	private LLNode<MapEntry<KeyType, ValueType>> head;
	private LLNode<MapEntry<KeyType, ValueType>> node;


	// constructor for empty map
	public MapLinkedList(Comparator<KeyType> myComparator) {
		comparator = myComparator;
		size = 0;
		head = null;
	}


	// constructor for one entry map
	/*public MapLinkedList(Comparator<KeyType> myComparator) {
		comparator = myComparator;
		size = 1;
		MapEntry entry = new MapEntry(9, 99);
		head = new LLNode<MapEntry<KeyType, ValueType>>(null, entry);
	}*/



	public static void main(String args[]) {

		IntegerComparator myComparator = new IntegerComparator();
		MapLinkedList<Integer, Integer> map = new MapLinkedList<Integer, Integer>(myComparator);


		// add elements to map, where one element is duplicate of previous to test put method
		map.put(1, 11);
		map.put(2, 22);
		map.put(3, 33);
		map.put(4, 44);
		System.out.println("old value: " + map.put(1, 12));


		System.out.println("get: " + map.get(1));


		System.out.println("Removed: " + map.remove(1));


		map.showMap();
	}



	public int size() {
		return size;
	}
```

```
public boolean isEmpty() {
        if(size == 0) { return true; }
        else { return false; }
}


public ValueType get(KeyType k) {
        node = head;

        for(int i = 0; i < size; i++) {

                if( comparator.compare( node.getElement().getKey(), k ) == 0 ) {

                        entry = node.getElement().getValue();
                        return entry;

                }
                node = node.getNext();
        }
        return null;
}



public ValueType put(KeyType k, ValueType e) {

        // declarations
        MapEntry newEntry = new MapEntry(k, e);
        LLNode<MapEntry<KeyType, ValueType>> newFirst;


        if(get(k) == null) {

                newFirst = new LLNode<MapEntry<KeyType, ValueType>>(head, newEntry);


                // set links
                head = newFirst;


                size++;
        }
        else {
                node = head;

                for(int i = 0; i < size; i++) {

                        if( comparator.compare( node.getElement().getKey(), k ) == 0 ) {

                                entry = node.getElement().getValue();
                                node.getElement().setValue(e);
                                return entry;


                        }
                        node = node.getNext();
                }
        }
        return null;
}
```

```
	public ValueType remove(KeyType k) {
		node = head;


		for(int i = 0; i < size; i++) {

			if( comparator.compare( node.getElement().getKey(), k ) == 0 ) {


				LLNode<MapEntry<KeyType, ValueType>> next = node.getNext();
				LLNode<MapEntry<KeyType, ValueType>> prev = head;


				for(int j = 0; j < i-1; j++) {
					prev = prev.getNext();
				}


				prev.setNext(next);
				size--;


				entry = node.getElement().getValue();
				return entry;

			}

			node = node.getNext();
		}
		return null;
	}



/************************
  * Helper Methods
  ************************/
	public void showMap() {

		System.out.println("\n****Start Map Structure****");
		System.out.println("head: " + head.getElement().getValue());

		// get starting node
		node = head;


		// loop through map
		for(int i = 0; i < size; i++) {

			System.out.println("element at position " + i + ": key = " + node.getElement().getKey()

		+ ", value = " + node.getElement().getValue());

			// go to next node
			node = node.getNext();

		}

		System.out.println("****End Map Structure****");
	}
```

```
}
```

```
import helpers.MapEntry;


public interface PriorityQueue<KeyType, ValueType> {

	public int size();
	public boolean isEmpty();


	public MapEntry insert(KeyType k, ValueType e);
	public MapEntry min();
	public MapEntry removeMin();

}
```

```
import helpers.Comparator;
import helpers.IntegerComparator;
import helpers.MapEntry;



public class PriorityQueueArray<KeyType, ValueType> implements PriorityQueue<KeyType, ValueType> {

	public static final int INIT_CAPACITY = 100;
	private int capacity;
	private MapEntry<KeyType, ValueType>[] entries;
	protected int numEntries;
	protected Comparator<KeyType> comparator;
	protected static final int NO_SUCH_KEY = -1;



	public PriorityQueueArray(Comparator<KeyType> myComparator) {
		capacity = INIT_CAPACITY;
		entries = new MapEntry[capacity];
		numEntries = 0;
		comparator = myComparator;
	}



	public int size() {
		return numEntries;
	}


	public boolean isEmpty() {
		return (numEntries == 0);
	}


	public static void main(String args[]) {

		IntegerComparator myComparator = new IntegerComparator();
		PriorityQueueArray<Integer, Integer> map = new PriorityQueueArray<Integer,
                                                                Integer>(myComparator);

		map.insert(1, 11);
		map.insert(2, 22);
		map.insert(0, 00);
		map.insert(3, 33);
```

```
		map.insert(4, 44);
		System.out.println("Min: " + map.min().getValue());
		System.out.println("REmoved Min: " + map.removeMin().getValue());

		map.showMap();
	}



	public MapEntry insert(KeyType k, ValueType e) {

		expandIfNecessary();
		MapEntry<KeyType, ValueType> newEntry = new MapEntry<KeyType, ValueType>(k, e);
		entries[numEntries] = newEntry;
		numEntries++;

		return newEntry;
	}



	public MapEntry min() {

		if(isEmpty()) {
			flagError("Queue is empty");
		}

		MapEntry<KeyType, ValueType> minKey = entries[0];

		for(int i = 0; i < numEntries; i++) {
			if( comparator.compare(entries[i].getKey(), (KeyType) minKey.getKey()) < 0 ) {
				minKey = entries[i];
			}
		}
		return minKey;
	}



	public MapEntry removeMin() {

		if(isEmpty()) {
			flagError("Queue is empty");
		}

		KeyType minKey = (KeyType) min().getKey();
		MapEntry<KeyType, ValueType> removedMin = null;


		for(int i = 0; i < numEntries; i++) {
			if( comparator.compare(entries[i].getKey(), minKey) == 0 ) {
				removedMin = entries[i];
				entries[i] = entries[numEntries-1];
				numEntries--;
			}
		}
		return removedMin;
	}
```

```
/************************
  * Helper Methods
  ************************/
        public void showMap() {
                System.out.println("\n****Start Map Structure****");


                // loop through map
                for(int i = 0; i < numEntries; i++) {

                        System.out.println("element at position " + i + ": key = " + entries[i].getKey()

                + ", value = " + entries[i].getValue());


                }


                System.out.println("****End Map Structure****");
        }


        protected void expandIfNecessary() {
                if (numEntries == capacity) {
                        int newCapacity = 2*capacity;
                        MapEntry<KeyType, ValueType>[] temp =  new MapEntry[newCapacity];

                        for (int i = 0; i < capacity; i++) {
                                temp[i] = entries[i];
                        }
                        entries = temp;
                        capacity = newCapacity;
                }
        }


        private void flagError(String errmsg) {
        System.out.println("LinkedStack: "+errmsg);
     System.exit(1);
   }


}
```

```
import helpers.Comparator;
import helpers.IntegerComparator;
import helpers.LLNode;
import helpers.MapEntry;


public class PriorityQueueDLinkedList<KeyType, ValueType> implements PriorityQueue<KeyType, ValueType> {

        /* Declarations */
        private int size;
        private MapEntry<KeyType, ValueType> entry;
         private Comparator<KeyType> comparator;
        private LLNode<MapEntry<KeyType, ValueType>> head;
         private LLNode<MapEntry<KeyType, ValueType>> tail;
        private LLNode<MapEntry<KeyType, ValueType>> node;
```

```
	public PriorityQueueDLinkedList(Comparator<KeyType> myComparator) {
	comparator = myComparator;
	size = 0;
	head = new LLNode<MapEntry<KeyType, ValueType>>(null, null, null);
	tail = new LLNode<MapEntry<KeyType, ValueType>>(head, null, null);
	head.setNext(tail);
}


	public int size() {
		return size;
	}

	public boolean isEmpty() {
		return size == 0;
	}



	public static void main(String args[]) {

		IntegerComparator myComparator = new IntegerComparator();
		PriorityQueueDLinkedList<Integer, Integer> map = new PriorityQueueDLinkedList<Integer,
									Integer>(myComparator);
		/*StringComparator myComparator = new StringComparator();
		PriorityQueueDLinkedList<String, String> map = new PriorityQueueDLinkedList<String,
									String>(myComparator);*/


		map.insert(0, 22);
		map.insert(1, 11);
		map.insert(2, 23);
		map.insert(3, 33);
		map.insert(4, 44);
		/*map.insert("b", "bb");
		map.insert("a", "aa");
		map.insert("c", "cc");*/
		System.out.println("min element: " + map.min().getKey());
		System.out.println("removedMin element: " + map.removeMin().getKey());


		map.showMap();
	}


	public MapEntry insert(KeyType k, ValueType e) {

		MapEntry newEntry = new MapEntry(k, e);
		LLNode<MapEntry<KeyType, ValueType>> oldFirst;
		LLNode<MapEntry<KeyType, ValueType>> newFirst;


		oldFirst = head.getNext();
		newFirst = new LLNode<MapEntry<KeyType, ValueType>>(head, oldFirst, newEntry);


		oldFirst.setPrev(newFirst);
		head.setNext(newFirst);


		size++;
		return newEntry;
```

```
	}



	public MapEntry min() {

		if (isEmpty()) {
		flagError("illegal operation: queue empty");
	}


		node = head.getNext();
		entry = node.getElement();

		for(int i = 0; i < size; i++) {

			if( comparator.compare( node.getElement().getKey(), entry.getKey() ) < 0 ) {

				entry = node.getElement();
			}
			node = node.getNext();
		}
		return entry;
	}



	public MapEntry removeMin() {

		if (isEmpty()) {
		flagError("illegal operation: queue empty");
	}


		entry = min();  // placed first as min() uses some elements from removeMin() i.e. need variable values to
				be fresh
		node = head.getNext();

		for(int i = 0; i < size; i++) {

			if( comparator.compare( node.getElement().getKey(), entry.getKey() ) == 0 ) {

				LLNode<MapEntry<KeyType, ValueType>> next = node.getNext();
				LLNode<MapEntry<KeyType, ValueType>> prev = node.getPrev();
				prev.setNext(next);
				next.setPrev(prev);


				size--;
				return node.getElement();
			}
			node = node.getNext();
		}
		return null;
	}



/************************
  * Helper Methods
  ************************/
	public void showMap() {
```

```
                System.out.println("\n****Start Map Structure****");
                System.out.println("head: " + head.getElement());

                // get starting node
                node = head.getNext();


                // loop through map
                for(int i = 0; i < size; i++) {

                        System.out.println("element at position " + i + ": key = " + node.getElement().getKey()

                + ", value = " + node.getElement().getValue());

                        // go to next node
                        node = node.getNext();

                }


                System.out.println("tail: " + tail.getElement());
                System.out.println("****End Map Structure****");
        }


        private void flagError(String errmsg) {
        System.out.println("LinkedStack: "+errmsg);
    System.exit(1);
  }

}
```

```
import helpers.Comparator;
import helpers.IntegerComparator;
import helpers.LLNode;
import helpers.MapEntry;
import helpers.StringComparator;


public class PriorityQueueLinkedList<KeyType, ValueType> implements PriorityQueue<KeyType, ValueType> {

        /* Declarations */
        private int size;
        private MapEntry<KeyType, ValueType> entry;
         private Comparator<KeyType> comparator;
        private LLNode<MapEntry<KeyType, ValueType>> head;
        private LLNode<MapEntry<KeyType, ValueType>> node;



        public PriorityQueueLinkedList(Comparator<KeyType> myComparator) {
        comparator = myComparator;
        size = 0;
        head = null;
  }


        public int size() {
                return size;
        }

        public boolean isEmpty() {
                return size == 0;
```

```
	}



	public static void main(String args[]) {

		IntegerComparator myComparator = new IntegerComparator();
		PriorityQueueLinkedList<Integer, Integer> map = new PriorityQueueLinkedList<Integer,
									Integer>(myComparator);
		/*StringComparator myComparator = new StringComparator();
		PriorityQueueLinkedList<String, String> map = new PriorityQueueLinkedList<String,
									String>(myComparator);*/


		// add elements to map, where one element is duplicate of previous to test put method
		/*System.out.println("0: " + map.insert(0, 00).getKey());
		System.out.println("1: " + map.insert(1, 11).getKey());
		System.out.println("2: " + map.insert(2, 22).getKey());*/
		map.insert(0, 00);
		map.insert(1, 11);
		map.insert(2, 22);
		//map.insert(3, 33);
		//map.insert(4, 44);
		/*map.insert("a", "aa");
		map.insert("b", "bb");
		map.insert("c", "cc");*/
		System.out.println("min element: " + map.min().getKey());
		System.out.println("removedMin element: " + map.removeMin().getKey());


		map.showMap();
	}


	public MapEntry insert(KeyType k, ValueType e) {


		MapEntry newEntry = new MapEntry(k, e);
		LLNode<MapEntry<KeyType, ValueType>> newFirst;


		newFirst = new LLNode<MapEntry<KeyType, ValueType>>(head, newEntry);


		head = newFirst;


		size++;
		return newEntry;
	}



	public MapEntry min() {

		if(isEmpty()) {
		flagError("min(): The PriorityQueue is empty!");
}


		node = head;
		entry = head.getElement();
```

```
		for(int i = 0; i < size; i++) {

			// if node map entry key is less than current min (node mapEntry key)
			if( comparator.compare( node.getElement().getKey(), entry.getKey() ) < 0 ) {

				entry = node.getElement();

			}
			node = node.getNext();
		}
		return entry;
	}



	public MapEntry removeMin() {

		if(isEmpty()) {
			System.out.println("min(): The PriorityQueue is empty!");
		System.exit(1);
	}


		entry = min();  // placed first as min() uses some elements from removeMin() i.e. need variable values to
				be fresh
		node = head;

		for(int i = 0; i < size; i++) {

			if( comparator.compare( node.getElement().getKey(), entry.getKey() ) == 0 ) {


				LLNode<MapEntry<KeyType, ValueType>> next = node.getNext();
				LLNode<MapEntry<KeyType, ValueType>> prev = head;


				for(int j = 0; j < i-1; j++) {
					prev = prev.getNext();
				}


				prev.setNext(next);
				size--;


				entry = node.getElement();
				return entry;

			}
			node = node.getNext();
		}
		return null;
	}



/************************
  * Helper Methods
  ************************/
	public void showMap() {
		System.out.println("\n****Start Map Structure****");
```

```
                System.out.println("head: " + head.getElement().getKey());

                // get starting node
                node = head;


                // loop through map
                for(int i = 0; i < size; i++) {

                        System.out.println("element at position " + i + ": key = " + node.getElement().getKey()
                                        + ", value = " + node.getElement().getValue());

                        // go to next node
                        node = node.getNext();

                }

                System.out.println("****End Map Structure****");
        }



        private void flagError(String errmsg) {
                System.out.println("LinkedStack: "+errmsg);
                System.exit(1);
        }

}
```

```
public interface Queue<EltType> {

  public int size();
  public boolean isEmpty();
  public EltType front();
  public void enqueue (EltType element);
  public EltType dequeue();

}
```

```
public class QueueArray<EltType> implements Queue<EltType> {

        public static final int INIT_CAPACITY = 100;        // default capacity of the queue
        private int capacity;                               // maximum capacity of the queue.
        private EltType entries[];                          // Q holds the queue elements
        private int f = 0;                                  // the front element of the queue.
        private int r = 0;                                  // the next available queue slot


        public QueueArray() {
                capacity = INIT_CAPACITY;
                entries = (EltType[])(new Object[capacity]);
        }



        public int size() {
                return ((capacity - f + r) % capacity);
        }
```

```
	public boolean isEmpty() {
		return (f == r);
	}



	public EltType front() {
		if (isEmpty()) {
			flagError("Queue is empty");
		}
		return entries[f];
	}


	public void enqueue (EltType element) {
		if (size() == capacity-1) {
			// copy queue contents into array of greater size
			EltType temp[] = (EltType[])(new Object[2*capacity]);
			int indexIntoQ = f;
			int indexIntoTemp = 0;

			for (indexIntoQ = f; indexIntoQ != r; indexIntoQ = (indexIntoQ + 1) % capacity) {
				temp[indexIntoTemp] = entries[indexIntoQ];
				indexIntoTemp++;
			}

			f = 0;
			r = indexIntoTemp;
			entries = temp;
			capacity = 2*capacity;
		}
		entries[r] = element;
		r = (r+1) % capacity;
	}


	public EltType dequeue() {

		if (isEmpty()) {
			flagError("Queue is empty");
		}


		EltType oldElement;
		oldElement = entries[f];
		entries[f] = null;
		f = (f+1) % capacity;
		return oldElement;
  }



/************************
  * Helper Methods
  ************************/
	private void flagError(String errmsg) {
		System.out.println("ArrayBasedQueue: "+errmsg);
		System.exit(1);
	}


}
```

```
import helpers.LLNode;


public class QueueDLinkedList<EltType> implements Queue<EltType> {

	// Declarations
	private int size;
	private LLNode<EltType> head;
	private LLNode<EltType> tail;


  public QueueDLinkedList() {
	size = 0;
	head = new LLNode<EltType>(null, null, null);
	tail = new LLNode<EltType>(head, null, null);
	head.setNext(tail);
  }


  public int size() {
	return size;
  }

  public boolean isEmpty() {
			return (size == 0);
  }


  public static void main(String args[]) {

			QueueDLinkedList<Integer> map = new QueueDLinkedList<Integer>();


			map.enqueue(1);
			map.enqueue(2);
			map.enqueue(3);
			map.enqueue(4);
			System.out.println("Front: " + map.front());
			System.out.println(map.dequeue());


			map.showMap();
	}




  public EltType front() {

	if (size == 0) {
			flagError("illegal queue op");
	}


	LLNode<EltType> firstNode = head.getNext();
	return firstNode.getElement();
  }



  public void enqueue(EltType obj) {
	LLNode<EltType> oldLast = tail.getPrev();
```

```
        LLNode<EltType> newLast = new LLNode<EltType>(oldLast, tail, obj);


        oldLast.setNext(newLast);
        tail.setPrev(newLast);


        size++;
    }



    public EltType dequeue() {

        if (size == 0) {
                flagError("illegal queue op");
        }


        LLNode<EltType> oldFirst = head.getNext();
        //head = head.getNext();
        head.setNext(oldFirst.getNext());


        size--;
        return oldFirst.getElement();
    }



/************************
     * Helper Methods
     ************************/
        public void showMap() {


                System.out.println("\n****Start Map Structure****");
                System.out.println("Headnext: " + head.getNext().getElement());

                // get starting node
                LLNode<EltType> currentElement = head.getNext();


                // loop through map
                for(int i = 0; i < size; i++) {

                        System.out.println("element at position " + i + ": value = " + currentElement.getElement());

                        // go to next node
                        currentElement = currentElement.getNext();

                }

                System.out.println("Tailprev: " + tail.getPrev().getElement());
                System.out.println("****End Map Structure****");
        }


    private void flagError(String errmsg) {
        System.out.println("LinkedQueue: "+errmsg);
       System.exit(1);
    }
```

```
}
```

```
import helpers.LLNode;


public class QueueLinkedList<EltType> implements Queue <EltType> {

	// Declarations
	private int size;
	private LLNode<EltType> head;
	private LLNode<EltType> tail;


  public QueueLinkedList() {
	size = 0;
	head = null;
	tail = null;
  }


  public int size() {
	return size;
  }

  public boolean isEmpty() {
			return (size == 0);
  }



  public static void main(String args[]) {

	QueueLinkedList<Integer> map = new QueueLinkedList<Integer>();


	map.enqueue(0);
	map.enqueue(1);
	map.enqueue(2);

	map.dequeue();
	map.dequeue();
	map.dequeue();

	map.enqueue(0);
	map.enqueue(1);
	map.enqueue(2);

	System.out.println(map.front());
	map.showMap();
	}




  public EltType front() {

	if (size == 0) {
		flagError("illegal queue op");
	}
```

```
        return head.getElement();
    }



    public void enqueue(EltType obj) {
        LLNode<EltType> newFirst = new LLNode<EltType>(null, obj);

        if (size == 0) {
            head = newFirst;
        }
        else {
            tail.setNext(newFirst);
        }

        tail = newFirst;
        size++;
    }


    public EltType dequeue() {

        if (size == 0) {
            flagError("illegal queue op");
        }


        EltType oldFirst = head.getElement();
        head = head.getNext();
        size--;


        return oldFirst;
    }




/************************
     * Helper Methods
     ************************/
        public void showMap() {
            System.out.println("\n****Start Map Structure****");
            System.out.println("head: " + head.getElement());

            // get starting node
            LLNode<EltType> node = head;


            // loop through map
            for(int i = 0; i < size; i++) {

                System.out.println("element at position " + i + ": value = " + node.getElement());

                // go to next node
                node = node.getNext();

            }

            System.out.println("****End Map Structure****");
        }
```

```
   private void flagError(String errmsg) {
            System.out.println("LinkedQueue: "+errmsg);
      System.exit(1);
   }


}
```

```
import java.util.HashSet;



public interface Set<EltType> extends java.lang.Iterable<EltType> {

  public boolean isEmpty();
  public int size();
  public void add(EltType newElement);
  public boolean contains(EltType checkElement);
  public void remove(EltType remElement);
  public void addAll(Set<EltType> addSet);
  public boolean containsAll(Set<EltType> checkSet);

}
```

```
import helpers.Comparator;
import helpers.IntegerComparator;

import java.util.HashSet;
import java.util.Iterator;


public class SetArray<EltType> implements Set<EltType> {

            // Declarations
            private final static int INIT_CAPACITY = 100;
            private int capacity;
            private int numEntries;
            private EltType[] entries;
            private Comparator<EltType> comparator;


            public SetArray(Comparator<EltType> myComparator) {
                        capacity = INIT_CAPACITY;
                        numEntries = 0;
                        comparator = myComparator;
                        entries = (EltType[])(new Object[capacity]);
            }


            public int size() {
                        return numEntries;
            }


            public boolean isEmpty() {
                        return (size() == 0);
            }



            public static void main(String args[]) {
```

```
		IntegerComparator myComparator = new IntegerComparator();
		SetArray<Integer> map = new SetArray<Integer>(myComparator);
		HashSet<Integer> map2 = new HashSet<Integer>();

		map.add(1);
		map.add(2);
		map.add(3);
		map.remove(1);


		map2.add(3);
		map2.add(4);
		/*map2.add(10);
		map2.add(11);
		map2.add(12);
		map.addAll(map2);*/

		System.out.println(map.size());
		System.out.println(map.containsAll(map2));


		map.showMap();
}



public void add(EltType newElement) {

		if(contains(newElement) == false) {
			entries[numEntries] = newElement;
			numEntries++;
		}

}



public boolean contains(EltType checkElement) {

		for(int i = 0; i < numEntries; i++) {
			if( comparator.compare(entries[i], checkElement) == 0 ) {
				return true;
			}
		}
		return false;
}



public void remove(EltType remElement) {

		for(int i = 0; i < numEntries; i++) {
			if( comparator.compare(entries[i], remElement) == 0 ) {
				entries[i] = entries[numEntries-1];
				numEntries--;
			}
		}
}
```

```
	public void addAll(HashSet<EltType> addSet) {

		for (EltType element : addSet) {

			add(element);
		}
	}



	public boolean containsAll(HashSet<EltType> checkSet) {

		for (EltType element : checkSet) {

			if(contains(element) == false) {
				return false;
			}
		}
		return true;
	}



/************************
  * Helper Methods
  ************************/
	public void showMap() {
		System.out.println("\n****Start Map Structure****");


		// loop through map
		for(int i = 0; i < numEntries; i++) {

			System.out.println("element at position " + i + ": value = " + entries[i]);


		}


		System.out.println("****End Map Structure****");
	}


	private void expandIfNecessary() {
		if (size() == capacity)  {
			// copy array into one of larger size
			EltType temp[] = (EltType[])(new Object[2*capacity]);
			for (int i = 0; i < capacity; i++) {
				temp[i] = entries[i];
				entries = temp;
				capacity = 2*capacity;
			}
		}
	}


	private void flagError(String errmsg) {
	System.out.println("LinkedStack: "+errmsg);
    System.exit(1);
  }
```

```
	public Iterator<EltType> iterator() {
		return null;
	}


	@Override
	public void addAll(Set<EltType> addSet) {
		// TODO Auto-generated method stub

	}


	@Override
	public boolean containsAll(Set<EltType> checkSet) {
		// TODO Auto-generated method stub
		return false;
	}



}
```

```
import java.util.Iterator;
import helpers.Comparator;
import helpers.IntegerComparator;
import helpers.LLNode;


public class SetDLinkedList<EltType> implements Set<EltType> {

	// Declarations
	private int size;
	private LLNode<EltType> head;
	private LLNode<EltType> tail;
	private LLNode<EltType> node;
	private Comparator<EltType> comparator;


	public SetDLinkedList(Comparator<EltType> myComparator) {
	comparator = myComparator;
	size = 0;
	head = new LLNode<EltType>(null, null, null);
	tail = new LLNode<EltType>(head, null, null);
	head.setNext(tail);
  }


	public int size() {
		return size;
	}

	public boolean isEmpty() {
		return size == 0;
	}



	public static void main(String args[]) {

		IntegerComparator myComparator = new IntegerComparator();
		SetDLinkedList<Integer> map = new SetDLinkedList<Integer>(myComparator);
		SetDLinkedList<Integer> map2 = new SetDLinkedList<Integer>(myComparator);
```

```
		/*StringComparator myComparator = new StringComparator();
		SetDLinkedList<String> map = new SetDLinkedList<String>(myComparator);*/


		// add elements to map
		map.add(1);
		map.add(2);
		map.add(3);
		map.add(3);
		System.out.println(map.contains(1));
		System.out.println(map.contains(4));
		map.remove(2);

		//map2.add(10);
		//map.addAll(map2);

		map.showMap();
	}



	public void add(EltType newElement) {

		if(contains(newElement) == false) {

			/*LLNode<EltType> oldFirst;
			LLNode<EltType> newFirst;


			oldFirst = head.getNext();
			newFirst = new LLNode<EltType>(head, oldFirst, newElement);


			head.setNext(newFirst);
			oldFirst.setPrev(newFirst);
			if(isEmpty()) {
				tail.setPrev(newFirst);
			}
			size++;*/

			LLNode<EltType> oldLast;
			LLNode<EltType> newLast;


			oldLast = tail.getPrev();
			newLast = new LLNode<EltType>(oldLast, tail, newElement);


			tail.setPrev(newLast);
			oldLast.setNext(newLast);


			size++;
		}
	}



	public boolean contains(EltType checkElement) {

		node = head.getNext();
```

```
		for(int i = 0; i < size; i++) {

			if( comparator.compare( node.getElement(), checkElement) == 0 ) {
				return true;
			}
			node = node.getNext();
		}
		return false;
	}



	public void remove(EltType remElement){

		if(contains(remElement) == false) {
			flagError("illegal operation: set empty");
		}


		node = head.getNext();


		for(int i = 0; i < size; i++) {

			if( comparator.compare( node.getElement(), remElement ) == 0 ) {

				LLNode<EltType> next = node.getNext();
				LLNode<EltType> prev = node.getPrev();
				prev.setNext(next);
				next.setPrev(prev);
				size--;
			}
			node = node.getNext();
		}
	}



	public void addAll(Set<EltType> addSet) {


		for (EltType e: addSet) {

			add(e);
		}
	}



	public boolean containsAll(Set<EltType> checkSet) {

		return false;
	}



/************************
  * Helper Methods
  ************************/
	public void showMap() {
```

```
                System.out.println("\n****Start Map Structure****");
                System.out.println("Headnext: " + head.getNext().getElement());

                // get starting node
                node = head.getNext();


                // loop through map
                for(int i = 0; i < size; i++) {

                        System.out.println("element at position " + i + ": value = " + node.getElement());

                        // go to next node
                        node = node.getNext();

                }

                System.out.println("Tailprev: " + tail.getPrev().getElement());
                System.out.println("****End Map Structure****");
        }


        private void flagError(String errmsg) {
        System.out.println("LinkedStack: "+errmsg);
    System.exit(1);
  }


        public Iterator<EltType> iterator() {

                return null;
        }

}
```

```
import java.util.Iterator;
import helpers.Comparator;
import helpers.IntegerComparator;
import helpers.LLNode;


public class SetLinkedList<EltType> implements Set<EltType> {

        // Declarations
        private int size;
        private LLNode<EltType> head;
        private LLNode<EltType> node;
        private Comparator<EltType> comparator;


        public SetLinkedList(Comparator<EltType> myComparator) {
        comparator = myComparator;
        size = 0;
        head = null;
  }


        public int size() {
                return size;
        }

        public boolean isEmpty() {
                return size == 0;
```

```
}


public static void main(String args[]) {

	IntegerComparator myComparator = new IntegerComparator();
	SetLinkedList<Integer> map = new SetLinkedList<Integer>(myComparator);
	SetLinkedList<Integer> map2 = new SetLinkedList<Integer>(myComparator);

	/*StringComparator myComparator = new StringComparator();
	SetDLinkedList<String> map = new SetDLinkedList<String>(myComparator);*/


	// add elements to map
	map.add(1);
	map.add(2);
	map.add(3);
	System.out.println(map.contains(1));
	//System.out.println(map.contains(4));
	map.remove(3);

	//map2.add(10);
	//map.addAll(map2);

	map.showMap();
}



public void add(EltType newElement) {

	if(contains(newElement) == false) {

		LLNode<EltType> oldHead = head;
		LLNode<EltType> newHead = new LLNode<EltType>(oldHead, newElement);


		head = newHead;


		size++;
	}
}



public boolean contains(EltType checkElement) {
	node = head;

	for(int i = 0; i < size; i++) {

		if( comparator.compare( node.getElement(), checkElement) == 0 ) {
			return true;
		}
		node = node.getNext();
	}
	return false;
}
```

```
public void remove(EltType remElement){

		if(contains(remElement) == false) {
			flagError("illegal operation: set does not contain element");
		}


		node = head;


		for(int i = 0; i < size; i++) {

			if( comparator.compare( head.getElement(), remElement ) == 0 ) {
				// if removing head
				head = head.getNext();


				size--;
			}
			else if( comparator.compare( node.getElement(), remElement ) == 0 ) {

				LLNode<EltType> next = node.getNext();
				LLNode<EltType> prev = head;


				for(int j = 0; j < i-1; j++) {
					prev = prev.getNext();
				}


				prev.setNext(next);


				size--;
			}
			node = node.getNext();
		}
}



public void addAll(Set<EltType> addSet) {


		/*for (EltType e: addSet) {

			if( !contains(e) == false) {

				add(e);

			}
		}*/
}



public boolean containsAll(Set<EltType> checkSet) {

		return false;
}
```

```
/************************
   * Helper Methods
   ************************/
            public void showMap() {

                        // check head and tail
                        System.out.println("Headnext: " + head.getNext().getElement());
                        //System.out.println("Tailprev: " + tail.getPrev().getElement());

                        System.out.println("\n****Start Map Structure****");
                        System.out.println("head: " + head.getElement());

                        // get starting node
                        node = head;


                        // loop through map
                        for(int i = 0; i < size; i++) {

                                    System.out.println("element at position " + i + ": value = " + node.getElement());

                                    // go to next node
                                    node = node.getNext();

                        }

                        System.out.println("****End Map Structure****");
            }


            private void flagError(String errmsg) {
            System.out.println("LinkedStack: "+errmsg);
       System.exit(1);
     }


            public Iterator<EltType> iterator() {

                        return null;
            }

}
```

```
public interface Stack<EltType> {

            public int size();
            public boolean isEmpty();
            public EltType top();
            public void push (EltType element);
            public EltType pop();

}
```

```
public class StackArray<EltType> implements Stack <EltType> {

            private static final int INIT_CAP = 100;            // default initial capacity of the stack
            private int capacity;                               // maximum capacity of the stack.
            private EltType entries[];                          // S holds the elements of the stack
```

```
        private int top = -1;                                   // the top element of the stack.


        public StackArray() {
                capacity = INIT_CAP;
                entries = (EltType[]) (new Object[ INIT_CAP]);
        }


        public int size() {
                return (top + 1);
        }

        public boolean isEmpty() {
                return (top < 0);
        }



        public EltType top() {
                if (isEmpty()) {
                        flagError("Stack is empty.");
                }
                return entries[top];
        }



        public void push(EltType obj)  {
                entries[++top] = obj;
        }



        public EltType pop() {
                if (isEmpty()) {
                        flagError("Stack is Empty.");
                }

                EltType elem;
                elem = entries[top];
                entries[top--] = null;
                return elem;
        }



/************************
  * Helper Methods
  ************************/
        private void expandIfNecessary() {
                if (size() == capacity)  {
                        // copy array into one of larger size
                        EltType temp[] = (EltType[])(new Object[2*capacity]);
                        for (int i = 0; i < capacity; i++) {
                                temp[i] = entries[i];
                                entries = temp;
                                capacity = 2*capacity;
                        }
                }
        }
```

```
	private void flagError(String errmsg) {
		System.out.println("ArrayBasedStack: "+errmsg);
		System.exit(1);
	}


}
```

```
import helpers.LLNode;


public class StackLinkedList<EltType> implements Stack<EltType> {

  // Declarations
  private LLNode<EltType> top;
  private int size;


  public StackLinkedList() {
	size = 0;
	top = null;
  }


  public int size() {
	return size;
  }

  public boolean isEmpty() {
	return size == 0;
  }




  public EltType top() {
	if (isEmpty()) {
		flagError("illegal operation: stack empty");
	}
	return top.getElement();
  }


  public void push(EltType obj) {
	LLNode<EltType> newFirst = new LLNode<EltType>(top, obj);
    top = newFirst;
    size++;
  }


  public EltType pop() {

	if (isEmpty()) {
		flagError("illegal operation: stack empty");
	}


	EltType oldFirst;
	oldFirst = top.getElement();
	 top = top.getNext();
	size--;
	return oldFirst;
  }
```

```
   private void flagError(String errmsg) {
            System.out.println("LinkedStack: "+errmsg);
      System.exit(1);
   }

}
```