Prepared by Linan Qiu <lq2137@columbia.edu>, adapted from Open Data Structures (opendatastructures.org)

# Doubly Linked Lists

In this class, you're not expected to implement a doubly linked list from scratch. However, you should still understand how a doubly linked list works.

Essentially, the only difference between singly linked lists and doubly linked lists is that each node keeps track of both its `prev` and `next` instead of only `next` for singly linked lists.

This makes each of the operations slightly more complex. I did not have time to write a `AwsmDoublyLinkedList.java` this week, so I included a version of a doubly linked list from last year's class. It does not implement `AwsmList` in entirety. However, I'd still encourage you to read through it.

*The source code is in `DoublyLinkedList.java` in this same directory. You may find that easier to read than this pdf.*

```java
// DoublyLinkedList.java

public class DoublyLinkedList<T> implements Iterable<T> {

  private int size;
  private Node<T> head;
  private Node<T> tail;

  /**
   * This is the doubly-linked list node.
   */
  public class Node<AnyType> {
    public AnyType data;
    public Node<AnyType> prev;
    public Node<AnyType> next;

    public Node(AnyType data, Node<AnyType> prev, Node<AnyType> next) {
      this.data = data;
      this.prev = prev;
      this.next = next;
    }
  }

  /**
   * Construct an empty LinkedList.
```

```
 */
public DoublyLinkedList() {
  head = new Node<>(null, null, null);
  tail = new Node<>(null, head, null);
  head.next = tail;

  size = 0;
}

public int size() {
  return size;
}

private Node<T> getNode(int idx, int lower, int upper) {
  Node<T> p;

  if (idx < lower || idx > upper)
    throw new IndexOutOfBoundsException("getNode index: " + idx + "; size: " + size());

  if (idx < size() / 2) { // Search through list from the beginning
    p = head.next;
    for (int i = 0; i < idx; i++)
      p = p.next;
  } else { // serch through the list from the end
    p = tail;
    for (int i = size(); i > idx; i--)
      p = p.prev;
  }

  return p;
}

/**
 * Gets the Node at position idx, which must range from 0 to size( ) - 1.
 *
 * @param idx
 *          index to search at.
 * @return internal node corresponding to idx.
 * @throws IndexOutOfBoundsException
 *           if idx is not between 0 and size( ) - 1, inclusive.
 */
private Node<T> getNode(int idx) {
  return getNode(idx, 0, size() - 1);
}

/**
```

```
 * Returns the item at position idx.
 *
 * @param idx
 *          the index to search in.
 * @throws IndexOutOfBoundsException
 *           if index is out of range.
 */
public T get(int idx) {
  return getNode(idx).data;
}

/**
 * Changes the item at position idx.
 *
 * @param idx
 *          the index to change.
 * @param newVal
 *          the new value.
 * @return the old value.
 * @throws IndexOutOfBoundsException
 *           if index is out of range.
 */
public T set(int idx, T newVal) {
  Node<T> p = getNode(idx);
  T oldVal = p.data;

  p.data = newVal;
  return oldVal;
}

/**
 * Adds an item to this collection, at specified position p. Items at or after
 * that position are slid one position higher.
 *
 * @param p
 *          Node to add before.
 * @param x
 *          any object.
 * @throws IndexOutOfBoundsException
 *           if idx is not between 0 and size(), inclusive.
 */
private void add(Node<T> p, T x) {
  Node<T> newNode = new Node<>(x, p.prev, p);
  newNode.prev.next = newNode;
  p.prev = newNode;
  size++;
```

```
}

/**
 * Adds an item to this collection, at specified position. Items at or after
 * that position are slid one position higher.
 *
 * @param x
 *          any object.
 * @param idx
 *          position to add at.
 * @throws IndexOutOfBoundsException
 *           if idx is not between 0 and size(), inclusive.
 */
public void add(int idx, T x) {
  add(getNode(idx, 0, size()), x);
}

/**
 * Adds an item to this collection, at the end.
 *
 * @param x
 *          any object.
 * @return true.
 */
public void add(T x) {
  add(size(), x);
}

/**
 * Removes the object contained in Node p.
 *
 * @param p
 *          the Node containing the object.
 * @return the item was removed from the collection.
 */
private T remove(Node<T> p) {
  p.next.prev = p.prev;
  p.prev.next = p.next;
  size--;

  return p.data;
}

/**
 * Removes an item from this collection.
 *
```

```
   * @param idx
   *          the index of the object.
   * @return the item was removed from the collection.
   */
  public T remove(int idx) {
    return remove(getNode(idx));
  }

  /**
   * Returns a String representation of this collection.
   */
  public String toString() {
    StringBuilder sb = new StringBuilder("[ ");

    for (T x : this)
      sb.append(x + " ");
    sb.append("]");

    return new String(sb);
  }

  /**
   * Obtains an Iterator object used to traverse the collection.
   *
   * @return an iterator positioned prior to the first element.
   */
  public java.util.Iterator<T> iterator() {
    return new LinkedListIterator();
  }

  /**
   * Return the index of the first occurence of o in the list or -1 if the index
   * is not in the list.
   */
  public int indexOf(Object o) {
    int i = 0;
    for (T e : this) {
      if (e.equals(o))
        return i;
      i++;
    }
    return -1;
  }

  /**
   * Reverse the linked list.
```

```
 */
public void reverse() {
  /**
   * The O(N) solution involves iterating through all nodes and swapping their
   * prev and next references.
   */
  Node<T> current = head;

  Node<T> temp = head;
  head = tail;
  tail = temp;

  while (current != null) {
    Node<T> oldNext = current.next;
    current.next = current.prev;
    current.prev = oldNext;
    current = oldNext;
  }
}

/**
 * This is the implementation of the LinkedListIterator. It maintains a notion
 * of a current position and of course the implicit reference to the
 * DoublyLinkedList.
 */
private class LinkedListIterator implements java.util.Iterator<T> {
  private Node<T> current = head.next;

  public boolean hasNext() {
    return current != tail;
  }

  public T next() {
    if (!hasNext())
      throw new java.util.NoSuchElementException();

    T nextItem = current.data;
    current = current.next;
    return nextItem;
  }
}

/**
 * Test the linked list.
 */
public static void main(String[] args) {
```

```
    DoublyLinkedList<Integer> lst = new DoublyLinkedList<>();

    for (int i = 0; i < 10; i++) {
      lst.add(i);
    }
    for (int i = 20; i < 30; i++) {
      lst.add(0, i);
    }

    lst.remove(0);
    lst.remove(lst.size() - 1);

    System.out.println(lst);

  }
}
```

I will highlight certain implementation quirks during the recitation.