CS1102/B
Data Structures and Algorithms
Practical Exam

Semester 1 2002/2003
School of Computing
National University of Singapore

Date: 26 October 2002
Time Allowed: 3 Hours

READ THE FOLLOWING INSTRUCTIONS CAREFULLY:

- This is an OPEN book exam. You may refer to your notes, books or any resource online (such as the Java API).
- You are not allowed to communicate with each other in any way, by word of mouth or otherwise.
- Turn off your handphone.
- Login to the PC using your NUSNET account as usual.
- You should have received a user id and a password for your special exam account from your invigilator. You must login to `sunfire` (also known as `sf3`) using this account. Your usual account will be disabled during the exam.
- After login, you should find two files Undoable.java and Main.java in your account. Do a directory list ("`ls`") to verify.
- Answer the questions in Undoable.java. You may modify Main.java for testing but anything written in Main.java will be ignored during grading.
- Your code must remain in your special exam account all the time. Transferring of your code to other systems (including laptop, desktop, mail server, web server, file server) is strictly prohibited.
- Write your name and matriculation number on top of the file UndoableList.java
- Compile your code with "`javac UndoableList.java Main.java`"
- Run your code with "`java Main`"
- You are encouraged to save your work often.
- Programs that cannot be compiled will be penalized heavily. If you cannot complete all methods, you should make sure that your program at least compiles.
- At the end of this exam, save your files, leave them in your account, and log off. You DO NOT have to type "submit". We will collect your files from your exam account after the exam.
- This paper contains SIX printed pages, including this cover page.

The objective of this exam is to implement a new data structure called UndoableList[1]. An UndoableList is a linear (non-circular), singly linked list without dummy nodes. It contains integers in non-decreasing sorted order and supports the usual find, insert and delete operations, plus two new operations called undo and redo. We will refer to insert and delete operations collectively as the *edit* operations. A list is considered to be the *most recent version* after we apply an edit operation.

The undo operation allows us to change the list back to a previous version, by reversing the last edit operation. We can call undo *n* times to reverse the last *n* edit operations. If there is no previous edit operation (because the list is new, or because all edit operations has been undone), undo does nothing.

A redo operation cancels the previous undo operation and is used to update the list back to the next more recent version. We can call redo multiple times consecutively to cancel multiple undo operations. redo does not have any effect on the most recent version of the list.

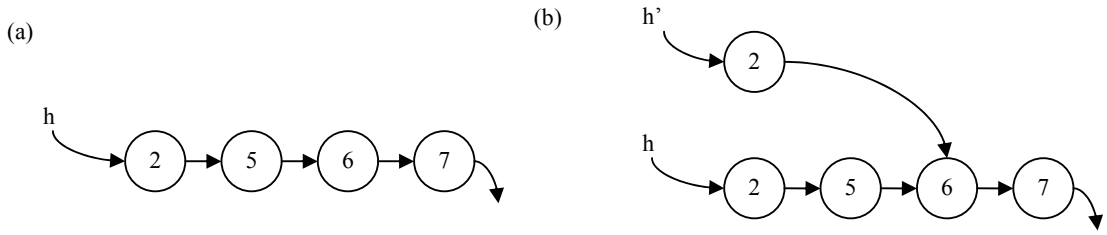
As an example, consider a list L containing 2, 5, and 7, denoted as [2,5,7]. The following shows the sequence of operations executed and the list after each operation.

1. insert(6) // [2,5,6,7]
2. delete(5) // [2,6,7]
3. undo // [2,5,6,7], delete(5) cancelled.
4. undo // [2,5,7], insert(6) cancelled.
5. redo // [2,5,6,7], last undo cancelled, re-apply insert(6).
6. insert(4) // [2,4,5,6,7]
7. redo // [2,4,5,6,7], nothing to redo.
8. delete(3) // [2,4,5,6,7], 3 is not in the list, L remains unchanged.
9. undo // [2,4,5,6,7], delete(3) cancelled.
10. undo // [2,5,6,7], insert(4) cancelled.
11. undo // [2,5,7], insert(6) cancelled (delete(5) has been cancelled by line 3).
12. redo // [2,5,6,7], reapply insert(6).
13. redo // [2,4,5,6,7] reapply insert(4).

To support undo and redo operations in constant time, you need to keep copies of previous versions of the list. Then, undo and redo can just switch between different versions of the list. To reduce memory requirements, you should keep only a finite number of most recent versions. Attempting to undo beyond the oldest version kept will not have any effect on the list.

Creating a copy of every node in the list after an edit operation is not necessary. You should share nodes among different versions of the list as much as possible. The following figure illustrates this with an example. Figure (a) shows the original list [2,5,6,7], pointed to by head pointer *h*. Figure (b) shows the new list [2,6,7] after delete(5), pointed to by head pointer *h'*. Since node 6 and 7 remains unchanged, they are shared by both [2,5,6,7] and [2,6,7].

[1] "Undoable" should be read as "Undo-able" (can undo), not "Un-doable" (cannot do).



We provide you with the following three classes.

Node: Implements a node in UndoableList. Each node contains an int value, and a next reference. We also tag each node with a unique integer nodeId, so that you can easily verify if you are sharing the correct nodes. All members are declared public for your convenience. Therefore, you can access a member of a node n, by using either "n.next" or the usual "n.getNext()/n.setNext()".

UndoableList: Implements an UndoableList. Some methods are incomplete. You are required to complete all seven incomplete methods. A method toString() is given to you. You can call toString() to convert your list into a string "[*value*:*nodeId, value*:*nodeId*, …]" for printing and debugging. This class also defines a constant `MAX_VERSION` that defines how many previous versions of the list to keep.

Main: This class defines the main() method used to test your Node and UndoableList. You are free to modify this class for testing purposes, but any modification in this class will not be graded.

YOUR TASK

You are to complete all seven methods in class UndoableList listed below using the scheme described previously:

UndoableList( ) : Constructor for an UndoableList object.

getHead() : Returns the head of the list.

find(int v): Returns true if value v exists in the list, returns false otherwise.

insert(int v): Inserts value v into the list, maintaining sorted order.

delete(int v): Deletes the first occurrence of value v from the list. If v does not exist in the list, do nothing.

undo(): Revert the list back to the previous version by canceling the last edit operation. If no previous version exists (either we have reverted to the oldest version available or the list is new), it has no effect on the list.

redo(): Reapply the operation cancelled by the last undo operation. It has no effect on the most recent version of the list.

You are free to add any methods or members to the class UndoableList and use the classes provided by the Java API. Some of these methods can be very easy if you do it the right way. Therefore, think and plan before you start coding. Grading will be based on three independent features: (i) operation insert/delete/find, (ii) operation undo/redo, and (iii) sharing of nodes among different list versions.

File: UndoableList.java

```
import java.util.*;

/**
 * UndoableList is a ordered, singly-linked, linear list that supports
 * insert(), delete(), find(), undo() and redo() operation.
 */
public class UndoableList {
  // MAX_VERSION is the maximum number of versions to keep.
  private static final int MAX_VERSION = 10;

  // ADD MEMBERS HERE.

  /**
   * Constructor for an UndoableList object
   */
  public UndoableList() {
    // COMPLETE THIS METHOD IN UndoableList.java.
  }

  /**
   * Return the head of the list.
   */
  private Node getHead() {
    // COMPLETE THIS METHOD IN UndoableList.java.
    return null; // For compiling only.  You may want to remove this.
  }

  /**
   * Inserts value v into this list, maintaining the sorted order.
   */
  public void insert(int v) {
    // COMPLETE THIS METHOD IN UndoableList.java.
  }

  /**
   * Deletes the first occurrence of value v from this list.  If v does
   * not exist in the list, do nothing.
   */
  public void delete(int v) {
    // COMPLETE THIS METHOD IN UndoableList.java.
  }

  /**
   * Returns true if value v exists in this list. Returns false
   * otherwise.
   */
  public boolean find(int v) {
    // COMPLETE THIS METHOD IN UndoableList.java.
    return false; // For compiling only.  You may want to remove this.
  }

  /**
   * Revert the list back to the previous version by canceling the last
   * edit operation.  If no previous version exists (either we have reverted
   * to the oldest version available, or the list is new), undo() has no
   * effect on the list.
   */
  public void undo() {
    // COMPLETE THIS METHOD IN UndoableList.java.
  }

  /**
   * Reapply the operation cancelled by the last undo.  It has no effect on
   * the most recent version of the list.
```

```
   */
  public void redo() {
    // COMPLETE THIS METHOD IN UndoableList.java.
  }

  /**
   * Returns a string containing the values in the list for printing.
   */
  public String toString()
  {
    Node head = getHead();
    if (head == null)
      return "[ ]";
    else
      return "[" + toString(head) + "]";
  }

  /**
   * Returns a string containing values from the list starting at
   * node n.
   */
  private String toString(Node n) {
    if (n == null)
      return "";
    else
      if (n.getNext() == null)
        return "" + n.getValue() + ":" + n.getNodeId();
      else
        return "" + n.getValue() + ":" + n.getNodeId() +", "+toString(n.getNext());
  }
}

/**
 * A Node represents a node in a linked list.  It contains: (i) value,
 * which is an int, is the data stored inside the node, and (ii) next, which
 * is a reference to the next node in the list.  next is null if it is the
 * last node. (iii) nodeId, a unique identifier to identify a Node object for
 * debugging.
 */
class Node {
  public static int lastId = 0;  // last node id.
  public int value;  // value stored inside this node.
  public int nodeId; // unique id to identify a node.
  public Node next;  // reference to the next node.

  public Node(int v) {
    value = v;
    next  = null;
    nodeId = lastId++;
  }
  public int getValue() {
       return value;
  }
  public Node getNext() {
       return next;
  }
  public int getNodeId() {
       return nodeId;
  }
  public void setValue(int v) {
       value = v;
  }
  public void setNext(Node n) {
       next = n;
  }
}
```

File: Main.java

```
// This file provides a simple set of test routines for testing your methods,
// you may modify it to test your methods as you like.  This file will not
// be looked at during grading.  You should code your answer for the exam
// questions in the file UndoableList.java.

import java.util.*;

public class Main {
  public static void main(String args[]) {
    UndoableList l = new UndoableList();
    Random r = new Random();

    // TEST insert
    for (int i = 0; i < 12; i++)
    {
      l.insert(r.nextInt(20));
      System.out.println(l);
    }

    // TEST find
    for (int i = 0; i < 12; i++)
    {
      int v = r.nextInt(20);
      boolean b = l.find(v);
      System.out.println("find " + v + ":" + b);
    }

    // TEST delete
    for (int i = 0; i < 5; i++)
    {
      int v = r.nextInt(20);
      l.delete(v);
      System.out.println("after deleting " + v + ":\n" + l);
    }

    // TEST undo and redo
    for (int i = 0; i < 12; i++)
    {
      l.undo();
      System.out.println("after undo:\n" + l);
    }

    for (int i = 0; i < 12; i++)
    {
      l.redo();
      System.out.println("after redo:\n" + l);
    }

    // TEST redo after modification.
    l.insert(30);
    System.out.println("after insert 30:\n" + l);
    l.redo();
    System.out.println("after redo:\n" + l);
  }
}
```

END OF PAPER