MACQUARIE University

WCOM125/ COMP125 Fundamentals of Computer Science
Workshop - Custom Linked Lists

Learning outcomes

By the end of this session, you will have learnt about custom built linked lists.

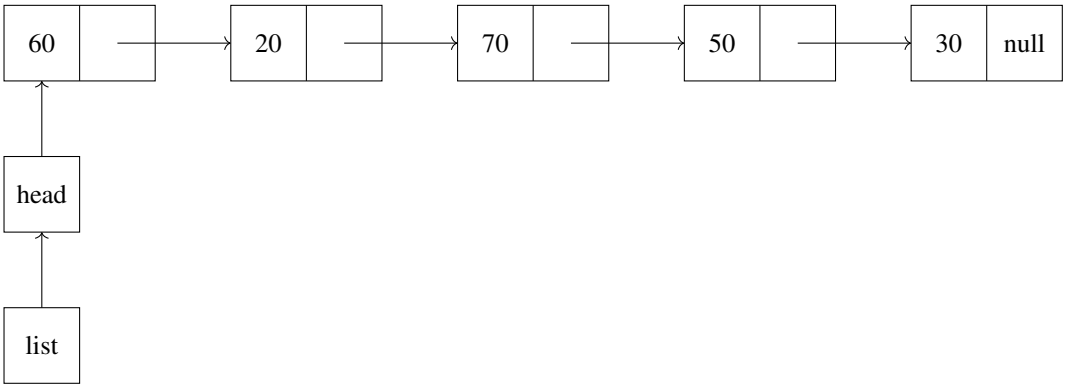
Questions

NOTE: All questions in this workshop relate to code in `Node.java` and `MyLinkedList.java`

1. Consider the following code:

```
1 MyLinkedList list = new MyLinkedList();
2 list.add(20);
3 list.add(50);
4 list.add(70, 1);
5 list.add(60, 0);
6 list.add(30, list.size());
```

   Draw a detailed memory diagram to represent the objects held in the memory.



2. What modifications do you have to make to classes `Node` and `MyLinkedList`, if, for two connected nodes `a` and `b`, you want to be able to go from `a` to `b` **AND** from `b` to `a`, and further be able to traverse list from left **AND** also from right or right to left?

   `Node` class needs to hold a `next` and `previous` reference.

   `MyLinkedList` class needs to hold a reference to the first node `head` (and traverse using `next`) and a reference to the last node `tail` (and traverse using `previous`).

   `add, remove` methods need to manipulate both `next` and `previous` links.

3. Implement as many as possible of the following methods in the `MyLinkedList` class. Read the javadoc specifications carefully before attempting.

   a. `public int countItemsMoreThan(int minValue);`

   b. `public int getMedianIndex();`

   c. `public void removeFirstOccurrence(int item);`

d. `public void removeAllOccurrences(int item);`

e. `public boolean containsAllDigits();`

f. `public MyLinkedList add(MyLinkedList other);`

```
public class MyLinkedList {
        private Node head;


        public boolean isEmpty() {
                return head==null;
        }

        public void addToFront(int item) {
                Node nodeToAdd = new Node(item, null);
                if(head == null)
                        head = nodeToAdd;
                else {
                        nodeToAdd.setNext(head); //link nodeToAdd to
                           head
                        head = nodeToAdd; //update head to point to
                           newly added node
                }
        }

        public void addToBack(int item) {
                Node nodeToAdd = new Node(item, null);
                if(head == null)
                        head = nodeToAdd;
                else {
                        Node cur = head;
                        while(cur.getNext() != null) {
                                cur = cur.getNext();
                        }
                        //now cur refers to the last node in the list
                        cur.setNext(nodeToAdd);
                }
        }

        public int size() {
                Node cur = head;
                int counter = 0;
                while(cur != null) {
                        //another non-null node
                        counter++;
                        cur = cur.getNext();
                }
                return counter;
        }

        public boolean isValidIndex(int idx) {
                return idx >= 0 && idx < size();
        }

        //add item at index supplied
        public void add(int item, int idx) {
                if(idx < 0 || idx > size())
                        return;
```

```
52                  Node nodeToAdd = new Node(item, null);
53
54                  if(head == null) {
55                          head = nodeToAdd;
56                          return;
57                  }
58
59                  if(idx == 0) { //add to front
60                          addToFront(item);
61                          return;
62                  }
63
64                  Node cur = head;
65                  for(int i=1; i < idx; i++) {
66                          cur = cur.getNext();
67                  }
68                  nodeToAdd.setNext(cur.getNext());
69                  cur.setNext(nodeToAdd);
70          }
71
72          //remove item from index supplied
73          public void remove(int idx) {
74                  if(idx < 0 || idx >= size())
75                          return;
76                  if(idx == 0) {
77                          head = head.getNext();
78                          return;
79                  }
80                  Node cur = head;
81                  for(int i=1; i < idx; i++) {
82                          cur = cur.getNext();
83                  }
84                  //cur refers to node before the node to be removed
85                  cur.setNext(cur.getNext().getNext());
86          }
87
88          //get item at index supplied
89          public Integer get(int idx) {
90                  if(idx < 0 || idx >= size())
91                          return null;
92                  Node cur = head;
93                  for(int i=0; i < idx; i++) {
94                          cur = cur.getNext();
95                  }
96                  return cur.getData();
97          }
98
99          public String toString() {
100                 if(head == null)
101                         return "[]";
102                 String result = "[";
103                 Node cur = head;
104                 while(cur!=null) {
105                         result = result + cur.getData() + ",␣";
106                         cur = cur.getNext();
107                 }
108                 result = result.substring(0, result.length()-2); //to
```

```
                remove the last ", "
            return result + "]";
        }

        /**
         * @return the number of items more than the passed value
         */
        public int countItemsMoreThan(int minValue) {
            int count = 0;
            Node cur = head;
            while(cur != null) {
                if(cur.getData() > minValue) {
                    count++;
                }
                cur = cur.getNext();
            }
            return count;
        }

        /**
         * @return the median index.
         * Return k if there are 2*k or 2*k+1 items.
         * think of methods you already have that will help you
        */
        public int getMedianIndex() {
            return size()/2; //YUPP!
        }

        //remove the first occurrence of passed item from the list
        public void removeFirstOccurrence(int item) {
            Node last = head;
            Node cur = head;
            while(cur != null && cur.getData() != item) {
                last = cur;
                cur = cur.getNext();
            }
            if(cur != null) { //got one :)
                last.setNext(cur.getNext());
            }
        }

        //remove all occurrences of passed item from the list
        public void removeAllOccurrences(int item) {
            while(head!= null && head.getData() == item) {
                head = head.getNext();
            }

            Node last = head;
            Node cur = head;

            while(cur != null) {
                while(cur != null && cur.getData() != item) {
                    last = cur;
                    cur = cur.getNext();
                }
                if(cur != null) { //got one :)
                    last.setNext(cur.getNext());
```

```
                }
                cur = cur.getNext();
            }
        }

        /**
         * For this method, assume that each item in the list
         * is a single digit item (that is either 0 or 1 or ... 9).
         * @return true if the list contains all digits, false
             otherwise
         */
        public boolean containsAllDigits() {
            boolean[] check = new boolean[10];
            int remainingCount = 10;

            Node cur = head;
            while(cur != null && remainingCount > 0) {
                if(check[cur.getData()] == false) {
                    remainingCount--;
                    check[cur.getData()] = true;
                }
            }
            return remainingCount == 0;
        }

        /**
         * For this method, assume that each item in the calling
         * object and the parameter object
         * is a single digit item (that is either 0 or 1 or ... 9).
         * thus the list itself represent a single arbirary-length
         * integer.
         * you may assume that both calling object and passed
         * object represent positive integers (integers > 0)
         * @return a list that represents an integer that is the sum
         * of the integer held by calling object and that held by
         * parameter object.
         * For example,
         * this.head  -> 9 -> 5 -> 0 -> 3 -> null (9503)
         * other.head -> 6 -> 9 -> 9 -> null (699)
         * return list such that
         * list.head -> 1 -> 0 -> 2 -> 0 -> 2 -> null (9503+699 =
             10202)
         */
        public MyLinkedList add(MyLinkedList other) {
            MyLinkedList result = new MyLinkedList();
            int idx1 = this.size()-1;
            int idx2 = other.size()-1;
            int carry = 0;
            while(idx1 >= 0 && idx2 >= 0) {
                int sumDigits = get(idx1) + other.get(idx2) +
                    carry;
                result.addToFront(sumDigits%10);
                carry = sumDigits/10;
                idx1--;
                idx2--;
            }
            while(idx1 >= 0) {
```

```
219                         int sumDigitCarry = get(idx1) + carry;
220                         result.addToFront(sumDigitCarry%10);
221                         carry = sumDigitCarry/10;
222                         idx1--;
223                 }
224                 while(idx2 >= 0) {
225                         int sumDigitCarry = other.get(idx2) + carry;
226                         result.addToFront(sumDigitCarry%10);
227                         carry = sumDigitCarry/10;
228                         idx2--;
229                 }
230                 if(carry != 0) {
231                         result.addToFront(carry);
232                 }
233                 return result;
234         }
235 }
```