Administrative Details

• Lab 1
  • Feedback on GitHub as a “Pull Request”
    • In a separate `TA-feedback` branch
    • `//` and `/*` */` comments are from TAs/instructors.
  • Comment on any of the PR lines if you have any questions!

• Lab 4
  • Optional partners again: please fill out form whether working alone or in pairs!
Last Time

• Induction
• List: A general-purpose interface
• Implementing Lists with linked structures
  • Singly Linked Lists
Today

• Implementing Lists with linked structures
  • Singly Linked Lists – methods and implementation
  • Circularly Linked Lists (more details in book)
  • Doubly Linked Lists – Lab 4
Linked List Basics

- There are two key aspects of Lists
  - Elements of the list
    - Store data, point to the “next” element
  - The list itself
    - Includes head (sometimes tail) member variable

Visualizing lists

![Diagram of a linked list showing head, tail, and list elements.](image)
Linked List Basics

- List nodes are recursive data structures
- Each “node” has:
  - A data value
  - A next variable that identifies the next element in the list
  - Can also have “previous” that identifies the previous element (“doubly-linked” lists)
- What methods does the Node class need?
  - next(), setNext(), value(), setValue()
**SinglyLinkedLists**

- How would we implement `SinglyLinkedListNode`?
  - `SinglyLinkedListNode = SLLN` in my notes
  - `SLLN = Node in the book (in Ch 9)`

![Diagram of SinglyLinkedListNode](image)

- How about `SinglyLinkedList`?
  - `SinglyLinkedList = SLL` in my notes

![Diagram of SinglyLinkedList](image)
Let’s Draw and Implement

**In SinglyLinkedListNode:**
- public SLLN(E v, SLLN<E> next)
- SLLN<E> next(),
  void setNext(SLLN<E> next)
- E value(), setValue(E value)

**In SinglyLinkedList:**
- public SLL()
- public void addFirst(E value),
  public E getFirst()
- public void addLast(E value),
  public E getLast()
More SLL Methods

• How would we implement:
  • get(int index), set(E d, int index)
  • add(E d, int index), remove(int index)
    • removeLast() is just remove(size() - 1)
    • removeFirst() is just remove(0)

• Left as an exercise:
  • contains(E d)
  • clear()

• Note: E is value type (generic)
Get and Set

//pre: index < size() – 1, size() > 0
public E get(int index) {
    SLLNode finger = head;
    for (int i=0; i<index; i++) {
        finger = finger.next();
    }
    return finger.value();
}

//pre: index < size() – 1, size() > 0
public E set(E d, int index) {
    SLLNode finger = head;
    for (int i=0; i<index; i++) {
        finger = finger.next();
    }
    E old = finger.value();
    finger.setValue(d);
    return old;
}
public void add(E d, int index) {
    if(index > size()) return;
    E old;

    if (index==0) { addFirst(d); }  

    else if (index==size()) { addLast(d); }  

    else {
        SLLN finger = head;  
        SLLN previous = null;  
        for (int i=0; i<index; i++) {
            previous = finger;  
            finger = finger.next();  
        }
        SLLN elem = new SLLN(d, finger);  
        previous.setNext(elem); // new “ith” item added after i-1  
        count++;
    }
}
public E remove(int index) {
    if (index >= size()) return null;

    E old;

    if (index == 0) { // Special case: remove from head
        old = head.value();
        head = head.next();
        count--;
        return old;
    }

    else {
        SLLN finger = head;
        for (int i=0; i < index-1; i++) { //stop one before index
            finger = finger.next();
        }
        old = finger.next.value();
        finger.setNext(finger.next().next());
        count--;
        return old;
    }
}
Linked Lists Summary

• Recursive data structures used for storing data
• More control over space use than Vectors
• Easy to add objects to front of list
• Components of SLL (SinglyLinkedList)
  • SLLN<E> head, int elementCount
• Components of SLLN (Node):
  • SLLN<E> next, SLLN<E> value
Vectors vs. SLL

• Compare performance of:
  • size()
  • addLast(), removeLast(), getLast()
  • addFirst(), removeFirst(), getFirst()
  • get(int index), set(E d, int index)
  • remove(int index)
  • contains(E d)
  • remove(E d)
<table>
<thead>
<tr>
<th>Operation</th>
<th>Vector</th>
<th>SLL</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>addLast</td>
<td>O(1) or O(n)(if resize)</td>
<td>O(n)</td>
</tr>
<tr>
<td>removeLast</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>getLast</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>addFirst</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>removeFirst</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>getFirst</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>get(i)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>set(i)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>remove(i)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>contains</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>remove(o)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
SLL Summary

- SLLs provide methods for efficiently modifying front of list
  - Modifying tail/middle of list is not quite as efficient
- SLL runtimes are consistent
  - No hidden costs like `Vector.ensureCapacity()`
  - Avg and worst case are always the same
- Space usage
  - No empty slots like vectors
  - But keep extra reference for each value
    - overhead proportional to list length
      - (but this is constant and predictable)
DoublyLinkedLists

- Nodes keep reference/links in both directions
- DLL keeps head and tail references
- `DoublyLinkedListNode` instance variables:
  - `DLLN<E> next;`
  - `DLLN<E> prev;`
  - `E value;`

Figure 9.7, Bailey pg. 202
DoublyLinkedLists

• Space overhead is proportional to number of elements
  • Still O(n) like SLL and Vector
• **ALL** operations on tail (including removeLast) are fast!
• Additional complexity in each list operation
  • Example: add(E d, int index)
    • Four cases to consider now:
      – empty list
      – add to front
      – add to tail
      – add in middle
public class DoublyLinkedNode<E> {
    protected E data;
    protected DoublyLinkedNode<E> nextElement;
    protected DoublyLinkedNode<E> previousElement;

    // Constructor inserts new node between existing nodes
    public DoublyLinkedNode(E v,
                              DoublyLinkedNode<E> next,
                              DoublyLinkedNode<E> previous)
    {
        data = v;
        nextElement = next;
        if (nextElement != null)
            nextElement.previousElement = this;
        previousElement = previous;
        if (previousElement != null)
            previousElement.nextElement = this;
    }
}
DoublyLinkedList

- We will implement a modified version of DLL in Lab 4
- See LinkedList.java on course webpage
- What is the purpose of the lab?
Lab Question: What are the advantages of adding dummy nodes?