Class #11: Linked Lists

Software Design III (CS 340): M. Allen, 17 Feb. 16

The Linked List ADT

- General idea: an implementation of List interface/ADT, using a set of linked nodes, with size() == # nodes
- Each node two basic attributes:
  - Data: object that it stores (simple or complex)
  - Next link: a variable reference to the next node in list
- End of list links to null object (can be used as a marker)

Wednesday, 17 Feb. 2016

Software Design III (CS 340)

Doubly Linked Lists

- Actual java.util.LinkedList class is slightly more complex, since each node has two separate links
- A bi-directional, doubly-linked list
- Nodes link to both the next and previous nodes in list order
- Front: prev links to null object
- Back: next links to null object

Wednesday, 17 Feb. 2016

Software Design III (CS 340)

One Last Complication: Sentinel Nodes

- For convenience, implementations will use one/more sentinel nodes
- These track the start (head) or end (tail) of the list (and have no data)
- Can make code easier to write, but don’t generally change basic behavior

Wednesday, 17 Feb. 2016

Software Design III (CS 340)
LinkedList Nodes

```java
private static class Node<T>
{
    private T data;
    private Node<T> prev;
    private Node<T> next;
    public Node(T d, Node<T> p, Node<T> n)
    {
        data = d;
        prev = p;
        next = n;
    }
}
```

Making the class `static` separates it from the parent class, so it has no requirement for a linked instance of the parent to exist.

- This simplifies things, and can improve memory management.
- This is a good approach, so long as the nested class doesn’t need direct access to other non-static members (variables, methods) of the containing class.
- Containing List class does have direct access to `Node` elements, whether `Node` is static or not.

Creating the LinkedList

```java
public class MyLinkedList<T> implements Iterable<T>
{
    private int theSize;
    private Node<T> head;
    private Node<T> tail;
    public MyLinkedList()
    {
        clear();
    }
    public void clear()
    {
        head = new Node<T>(null, null, null);
        tail = new Node<T>(null, head, null);
        head.next = tail;
        theSize = 0;
    }
}
```

Use `head/tail` nodes to indicate start/end of list.

Creating the LinkedList Iterators

```java
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;
    }
    public void remove() {
        MyLinkedList.this.remove(current.prev);
    }
}
```

This private inner class is not `static`. It can easily directly access any other elements of the List class it needs.
Adding Nodes to Linked Lists

With array-based lists, adding/removing involves shifting objects around to open/close space in array, leading to linear $O(n)$ complexity.

With linked lists, we can add nodes by simply creating the new node, then adding/deleting links appropriately (a constant-time $O(1)$ process).

Note: Order is important! We can lose data if we delete links to a node before adding a new one to keep track of it.

Deleting Nodes from Linked Lists

Can also remove from linked structure in constant-time $O(1)$.

We first change over the links for the remaining list elements, and then delete the node and its own links.

```java
public boolean add(T x) {
    add(theSize, x);
    return true;
}

public void add(int idx, T x) {
    addBefore(getNode(idx, 0, theSize), x);
}

private void addBefore(Node<T> p, T x) {
    Node<T> newNode = new Node<T>(x, p.prev, p);
    newNode.prev.next = newNode;
    p.prev = newNode;
    theSize++;
}
```

```java
public T remove(int idx) {
    return remove(getNode(idx));
}

private T remove(Node<T> p) {
    p.next.prev = p.prev;
    p.prev.next = p.next;
    p.next = null;
    p.prev = null;
    theSize--;

    return p.data;
}
```

User calls public remove(int) and gives a position at which to remove data.

That method uses a helper to get the list node to remove, sends it to another helper.

The first helper, getNode(), throws an IndexOutOfBoundsException if idx < 0 or idx >= theSize.

Second helper, remove(Node<T>), does the work of re-arranging the node links, adjusting the list-size, and returning removed data (in case needed at the original calling site).
Unlike array-based lists, however, we cannot directly access a list element in constant time. Instead, we must start from either the head or tail, and iterate inwards to the position we want in the list. This gives worst-case \( \Theta(n/2) \) operations == \( \Theta(n) \).
This Week

- **Topic:** Linear Structures
- **Read:** Text, chapter 03
- **In Lab:** Friday, 19 Feb.
- **Homework 02:** due Wednesday, 24 Feb. (5:00 PM)
- **Office Hours:** Wing 210
  - Tuesday & Thursday: 10:00–11:30 AM
  - Tuesday & Thursday: 4:00–5:30 PM