Lecture 4: SLLists, Nested Classes, Sentinel Nodes

9/2/2020

```
public class IntList {
    public int first;
    public IntList rest;

    public IntList(int f, IntList r) {
        first = f;
        rest = r;
    }
}
```

- While functional, "naked" linked lists like the one above are hard to use
    - Users of this class are probably going to need to know references very well, and be able to think recursively. Let's make our users' lives easier

Improvement #1: Rebranding and Culling

```
public class IntNode {
    public int item;
    public IntNode next;

    public IntNode(int i, IntNode n) {
        item = i;
        next = n;
    }
}
```

Improvement #2: Bureaucracy

```
public class IntNode {
    public int item;
    public IntNode next;

    public IntNode(int i, IntNode n) {
        item = i;
        next = n;
    }
}

// An SLList is a list of integers, which hides the terrible truth of the
nakedness within
public class SLList {
```

```
    public IntNode first;

    public SLList(int x) {
        first = new IntNode(x, null);
    }

    // Adds x to the front of the list
    public void addFirst(int x) {
        first = new IntNode(x, first);
    }

    // Returns the first item in the list
    public int getFirst() {
        return first.item;
    }

    public static void main(String[] args) {
        // Creates a list of one integer, namely 10
        SLList L = new SLList(10);
        L.addFirst(10); // Adds 10 to front of list
        L.addFirst(5); // Adds 5 to front of list
        int x = L.getFirst();
    }
}
```

- SLList is easier to instantiate (no need to specify `null`)

The Basic SLList and Helper IntNode Class

- While functional, "naked" linked lists like the IntList class are hard to use
  - Users of `IntList` need to know Java references well, and be able to think recursively
  - `SLList` is much simpler to use

A Potential SLList Danger

```
SLList L = new SLList(15);
L.addFirst(10);
L.first.next.next = L.first.next;
```

```
public class SLList {
    private IntNode first;

    public SLList(int x) {
        first = new IntNode(x, null);
    }

    // Adds x to the front of the list
    public void addFirst(int x) {
        first = new IntNode(x, first);
```

```
    }

    // Returns the first item in the list
    public int getFirst() {
        return first.item;
    }

    public static void main(String[] args) {
        // Creates a list of one integer, namely 10
        SLList L = new SLList(10);
        L.addFirst(10); // Adds 10 to front of list
        L.addFirst(5); // Adds 5 to front of list
        int x = L.getFirst();
    }
}
```

- Use the `private` keyword to prevent code in other classes from using members (or constructors) of a class

Why Restrict Access?

- Hide implementation details from users of your class
  - Less for user of class to understand
  - Safe for you to change private methods (implementation)
- Despite the term 'access control':
  - Nothing to do with protection against hackers, spies, or other evil entities

Improvement #4: Nesting a Class

```
public class SLList {

    private static class IntNode { // static: never looks outwards
        public int item;
        public IntNode next;

        public IntNode(int i, IntNode n) {
            item = i;
            next = n;
        }
    }

    private IntNode first;

    public SLList(int x) {
        first = new IntNode(x, null);
    }

    // Adds x to the front of the list
    public void addFirst(int x) {
        first = new IntNode(x, first);
    }
```

```
    // Returns the first item in the list
    public int getFirst() {
        return first.item;
    }

    public static void main(String[] args) {
        // Creates a list of one integer, namely 10
        SLList L = new SLList(10);
        L.addFirst(10); // Adds 10 to front of list
        L.addFirst(5); // Adds 5 to front of list
        int x = L.getFirst();
    }
}
```

- Could have made `IntNode` private if we wanted to

Why Nested Classes?

- Nested Classes are useful when a class doesn't stand on its own and is obviously subordinate to other classes
    - Make the nested class private if other classes should never use the nested class

Static Nested Classes

- If the nested class never uses any instance variables or methods of the outside class, declare it static
    - Static classes cannot access outer class's instance variables or methods
    - Results in a minor savings of memory

Adding More SLList Functionality

- To motivate our remaining improvements, ad to give more functionality to our SLList class, let's add:
    - `.addLast(int x)`
    - `.size()`

```
public class SLList {

    private static class IntNode { // static: never looks outwards
        public int item;
        public IntNode next;

        public IntNode(int i, IntNode n) {
            item = i;
            next = n;
        }
    }

    private IntNode first;

    public SLList(int x) {
```

```
        first = new IntNode(x, null);
    }

    // Adds x to the front of the list
    public void addFirst(int x) {
        first = new IntNode(x, first);
    }

    // Returns the first item in the list
    public int getFirst() {
        return first.item;
    }

    // Adds an item to the end of the list
    public void addLast(int x) {
        IntNode p = first;
        while (p.next != null) {
            p = p.next;
        }
        p.next = new IntNode(x, null);
    }

    // Returns the size of the list that starts at IntNode p
    private static int size(IntNode p) {
        if (p.next == null) {
            return 1;
        }
        return 1 + size(p.next);
    }

    public int size() {
        return size(first);
    }

    public static void main(String[] args) {
        // Creates a list of one integer, namely 10
        SLList L = new SLList(10);
        L.addFirst(10); // Adds 10 to front of list
        L.addFirst(5); // Adds 5 to front of list
        L.addLast(20);
        System.out.println(L.size())
    }
}
```

Efficiency of Size

- How efficient is size?
  - Suppose size takes 2 seconds on a list of size 1000
  - How long will it take on a list of size 1000000?
    - 2000 seconds!

Improvement #5: Fast size()

- Your goal:
  - Modify SLList so that the execution time of size() is always fast (i.e. independent of the size of the list)

```
public class SLList {

    private static class IntNode { // static: never looks outwards
        public int item;
        public IntNode next;

        public IntNode(int i, IntNode n) {
            item = i;
            next = n;
        }
    }

    private IntNode first;
    private int size;

    public SLList(int x) {
        first = new IntNode(x, null);
        size = 1;
    }

    // Adds x to the front of the list
    public void addFirst(int x) {
        first = new IntNode(x, first);
        size += 1;
    }

    // Returns the first item in the list
    public int getFirst() {
        return first.item;
    }

    // Adds an item to the end of the list
    public void addLast(int x) {
        size += 1;
        IntNode p = first;
        while (p.next != null) {
            p = p.next;
        }
        p.next = new IntNode(x, null);
    }

    public int size() {
        return size;
    }

    public static void main(String[] args) {
        // Creates a list of one integer, namely 10
        SLList L = new SLList(10);
```

```
        L.addFirst(10); // Adds 10 to front of list
        L.addFirst(5); // Adds 5 to front of list
        L.addLast(20);
        System.out.println(L.size())
    }
}
```

- Solution: Maintain a special size variable that caches the size of the list
    - Caching: putting aside data to speed up retrieval
- TANSTAAFL: There ain't no such thing as a free lunch
    - But spreading the work over each add call is a net win in almost any case
- The SLList class allows us to store meta information about the entire list, e.g. `size`

Improvement #6a: Representing the Empty List

```
public class SLList {

    private static class IntNode { // static: never looks outwards
        public int item;
        public IntNode next;

        public IntNode(int i, IntNode n) {
            item = i;
            next = n;
        }
    }

    private IntNode first;
    private int size;

    // Creates an empty SLList
    public SLList() {
        first = null;
        size = 0;
    }

    public SLList(int x) {
        first = new IntNode(x, null);
        size = 1;
    }

    // Adds x to the front of the list
    public void addFirst(int x) {
        first = new IntNode(x, first);
        size += 1;
    }

    // Returns the first item in the list
    public int getFirst() {
        return first.item;
    }
```

```
    // Adds an item to the end of the list
    public void addLast(int x) {
        size = size + 1;

        if (first == null) {
            first = new IntNode(x, null);
            return;
        }

        IntNode p = first;
        while (p.next != null) {
            p = p.next;
        }
        p.next = new IntNode(x, null);
    }

    public int size() {
        return size;
    }

    public static void main(String[] args) {
        // Creates a list of one integer, namely 10
        SLList L = new SLList();
        L.addFirst(10); // Adds 10 to front of list
        L.addFirst(5); // Adds 5 to front of list
        L.addLast(20);
        System.out.println(L.size())
    }
}
```

Tip for Being a GOod Programmer: Keep Code Simple

- As a human programmer, you only have so much working memory
  - You want to restrict the amount of complexity in your life!
  - Simple code is (usually) good code
    - Special cases are not "simple"

addLast's Fundamental Problem

- The fundamental problem:
  - The empty list has a null `first`, can't access `first.next`
- Our fix is a bit ugly:
  - Requires a special case
  - More complex data structures will have many more special cases
- How can we avoid special cases?
  - Make all `SLLists` (even empty) the "same"

Improvement #6b: Representing the Empty List Use a Sentinel Node

- Create a special node that is always there! Let's call it a "sentinel node"

- The empty list is just the sentinel node
- A list with 3 numbers has a sentinel node and 3 nodes that contain real data
- Let's try reimplementing SLList with a sentinel node

```
public class SLList {

    private static class IntNode { // static: never looks outwards
        public int item;
        public IntNode next;

        public IntNode(int i, IntNode n) {
            item = i;
            next = n;
        }
    }

    // The first item (if it exists) is at sentinel.next
    private IntNode sentinel;
    private int size;

    // Creates an empty SLList
    public SLList() {
        sentinel = new IntNode(69, null);
        size = 0;
    }

    public SLList(int x) {
        sentinel = new IntNode(69, null);
        sentinel.next = new IntNode(x, null);
        size = 1;
    }

    // Adds x to the front of the list
    public void addFirst(int x) {
        sentinel.next = new IntNode(x, sentinel.next);
        size = size + 1;
    }

    // Returns the first item in the list
    public int getFirst() {
        return sentinel.next.item;
    }

    // Adds an item to the end of the list
    public void addLast(int x) {
        size = size + 1;

        IntNode p = sentinel;
        while (p.next != null) {
            p = p.next;
        }
        p.next = new IntNode(x, null);
```

```
    }

    public int size() {
        return size;
    }

    public static void main(String[] args) {
        // Creates a list of one integer, namely 10
        SLList L = new SLList();
        L.addFirst(10); // Adds 10 to front of list
        L.addFirst(5); // Adds 5 to front of list
        L.addLast(20);
        System.out.println(L.size())
    }
}
```

Sentinel Node

- The sentinel node is always there for you
- Notes:
  - I've renamed `first` to be `sentinel`
  - `sentinel` is never null, always points to sentinel node
  - Sentinel node's `item` needs to be some integer, but doesn't matter what value we pick
  - Had to fix constructors and methods to be compatible with sentinel nodes

addLast (with Sentinel Node)

- Bottom line: Having a sentinel simplifies our `addLast` method
  - No need for a special case to check if `sentinel` is null

Invariants

- An invariant is a condition that is guaranteed to be true during code execution (assuming there are no bugs in your code)
- An `SLList` with a sentinel node has at least the following invariants
  - The `sentinel` reference always points to a sentinel node
  - The first node is always at `sentinel.next`
  - The `size` variable is always the total number of items that have been added
- Invariants make it easier to reason about code
  - Can assume they are true to simplify code (e.g. addLast doesn't need to worry about nulls)
  - Must ensure that methods preserve invariants