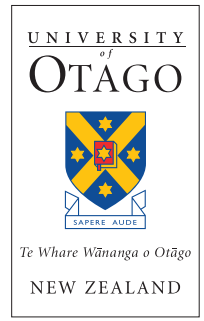
Cosc 241
Programming and Problem Solving
Lecture 9 (23/3/2020)
Links, lists, and linked lists

Michael Albert
michael.albert@cs.otago.ac.nz

Keywords: linked list, iterators, anonymous classes



References as links

- A Java reference is just that – it *refers* to a memory location in which the complete data for an object is stored.
- In other languages such references are sometimes called *pointers* and a clear distinction is drawn between “objects” and “references to objects” (cf. C, in COSC 242).
- A reference to an object of the same, or a closely related type is often called a *link*.
- A collection of linked objects is called a *linked data structure*.

Family Tree

```
public class Person{

   private String name;
   private Date dateOfBirth;

   private Person father;
   private Person mother;
   private Person[] children;

}
```

- A complex linked and recursive data structure!
- Where does it start? How do we navigate it? How do we maintain it?

Simple linked lists

- A *linked list* consists of a sequence of nodes connected by links.
- Each node, except the last, has a *successor*.
- Each node, except the first, has a *predecessor*
- Each node contains a single element and links (i.e., references) to its successor and/or predecessor.
- The key difference with an array is that by manipulating the links we can change the *structure* of the list – not just the values it stores.
- A linked list is a *dynamic* data structure.

Dynamic vs. static

- A *static* data structure is one whose structure (i.e., its disposition in the computer's memory, or the organisation of the data it contains) is fixed at the time it is created and cannot be changed.
- For instance, arrays – we cannot change the *structure* of an array, only its *contents*.
- A *dynamic* data structure is one whose structure can change.
- For instance, linked lists – by changing the links we change the actual structure of the list.
- Sometimes we use static structures (because they tend to be more efficient) in the background of apparently dynamic structures (for instance, using an array to model a stack).
- Generally this requires some finesse to cope with the clash between the static and dynamic requirements (e.g., when the stack grows beyond the capacity of its backing array).

Singly linked lists

- In a *singly linked list* each node stores only one link, to the next element of the list
- The actual list object contains a reference to the first node of the list (or a `null` reference if the list is empty).

```
public class SinglyLinkedList<T> {
   private SLLNode<T> first;
   public SinglyLinkedList() {
      this.first = null;
   }

   private class SLLNode<T> {
      private T value;
      private SLLNode<T> next;
      private SLLNode(T value, SLLNode<T> next) {
         this.value = value;
         this.next = next;
      }
   }
}
```

How do we get started?

- Shortly, we will examine how to insert elements into an existing singly linked list (including an empty one).
- First a `toString` method so that we can see what's going on.
- Now add a single node at the head of a singly linked list.
- And now at the end.

toString

```
public String toString() {
    StringBuilder result = new StringBuilder("<SLL>: ");
    SLLNode node = this.first;
    while (node != null) {
        result.append(node.value);
        result.append(" --> ");
        node = node.next;
    }
    return result.toString();
}
```

- Uses the idea of *list traversal*.
- Start at the first node.
- While the node is not null, do something.
- Set the node to the next node.
- Later we will add an *iterator* that allows us to do this generally.

Adding at either end

```
public void add(T value) {
    SLLNode<T> newNode = new SLLNode(value, this.first);
    this.first = newNode;
}

public void addLast(T value) {

    if (this.first == null) {
        add(value); return;
    }

    SLLNode current = this.first;
    while (current.next != null) current = current.next;
    current.next = new SLLNode(value, null);
}
```

First is $O(1)$, second $O(n)$ where n is the size of the list.

Making `addLast` constant time

- How could we change the implementation so that `add` and `addLast` both have $O(1)$ performance?
- The problem with `addLast` as it stands is that we need to traverse the list to find the last node.
- So what about adding a new data field which stores the last node of the list? Call it, oh I don't know, `last`.

```
public void addLast(T value) {
    if (this.first == null) {
        add(value);
        this.last = this.first;
        return;
    }
    this.last.next = new SLLNode(value, null);
    this.last = this.last.next;
}
```

Accessing by index

- We can think of the items in a linked list as being indexed, starting from 0 for the first node etc.
- In that case it makes sense to be able to `get` the value at a specified index.
- Because it will be useful later, we include and use a (private - why?) method for getting the node at a specific index.
- Note that if the index is too small or large, we throw an `IndexOutOfBoundsException`. As this is a run time exception, it does not need to be caught, or re-thrown by calling methods.

Accessing by index

```
public T get(int index) {
    return getNode(index).value;
}

private SLLNode<T> getNode(int index)
    throws IndexOutOfBoundsException {

    if (index < 0)
        throw new IndexOutOfBoundsException("Negative index");

    SLLNode current = this.first;
    while (current != null && index > 0) {
        current = current.next; index--;
    }

    if (current == null)
        throw new IndexOutOfBoundsException("Index too large");
    return current;

}
```

Adding in the middle

- Insertion in the middle of a linked list is a little tricky.
- The `next` reference of the inserted item should be the `next` reference of the preceding item.
- The `next` reference of the preceding item should be reset to the inserted item.
- In our structure, adding at the beginning is a special case.
- Note that `IndexOutOfBoundsException` might occur implicitly.

Adding in the middle

```
public void add(T value) {
    SLLNode<T> newNode = new SLLNode(value, this.first);
    this.first = newNode;
}

public void add(int index, T value) {
    if (index == 0) {
        this.add(value);
        return;
    }
    SLLNode prev = this.getNode(index-1);
    SLLNode newNode = new SLLNode(value, prev.next);
    prev.next = newNode;
}
```

Iterable

- One of the useful features of many classes in the `Collections` framework is that they can be the targets of "foreach" statements, e.g.,
  `for(String s : ArrayDeque<String> titles)`
  `...`
- Any class can do this providing that it implements the `Iterable` interface.
- This in turn requires an `iterator()` method which returns an object.
- That implements the `Iterator` interface (but relax, it stops there).

Iterators

- Iterators must support three methods:

  `hasNext()` returns a boolean indicating if there's anything more to return,

  `next()` returns an object from the iterator,

  `remove()` removes the last item returned (frequently just throws an `UnsupportedOperation` exception).

- These are often defined *anonymously*, i.e., the code that defines the behaviour of the iterator is given directly where the iterator is constructed (not as a separate named type).
- Iterators over dynamic data structures are generally allowed, even expected, to behave unpredictably if the structure is modified while the iterator is active.

An iterator for singly linked lists

```
public java.util.Iterator<T> iterator() {
  return new java.util.Iterator<T>() {
    private SLLNode<T> current = first;

    public boolean hasNext() {
      return current != null;
    }

    public T next() {
    T result = current.value;
      current = current.next;
      return result;
    }

    public void remove() {
      throw new UnsupportedOperationException();
    }
  };
}
```

Things to think about

- Linked list code can be cryptic – pictures help.
- Searching
  - Remember the difference between `a == b` and `a.equals(b)` for reference types.
  - What if the list contains `null` elements?
- Converting to an array (pretty easy).
- Converting from an array (ditto).
- Sorting.