Lecture 5: Linked Structures
Lists, Stacks and Queues

Reading materials
Dale, Joyce, Weems: 2.5-2.7, 3.7, 5.6
OpenDSA: 5.4-5.12
Liang: only in Comprehensive edition

Topics Covered

1 Review of Java Reference Variables
2 Linked Structures
   2.1 Linking Nodes Together
3 Linked Lists
   3.1 insert method
   3.2 remove method
   3.3 clear method
   3.4 contains method
   3.5 indexOf method
   3.6 get method
   3.7 size method
   3.8 Implementation summary
4 Stacks
   4.1 Wrapper Implementation
   4.2 Another Inefficient Implementation
   4.3 An Efficient Stack Implementation
5 Queues
6 Performance Analysis

6.1 Experimentally Measuring Performance ........................................... 23
6.2 Theoretically Measuring Performance ............................................. 23
6.3 Singly Linked List <=> Array ......................................................... 25
1 Review of Java Reference Variables

In Java, when you allocate a variable of a primitive data type (see http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html for listing of types and descriptions) the memory location associated with that variable contains the value assigned to that variable.

Example:
The following lines of code

```java
int x;
x = 5;
```

result in a 32-bit block of memory being associated with the variable \( x \) and then the bits are used to represent the value of 5 (for simplicity, I will just write the number 5 instead of its binary equivalent).

With reference type variables the memory allocation is different. When you create a reference variable it does not have any room (bits/bytes in memory) for storing data. It only contains enough memory to store a reference or an address of memory location in which the actual data is stored.

The following line of code creates a reference variable, unless it is followed by creation of an actual object, there is no room to store data.

```java
Person p;
```

In fact, you should make a habit or always assigning the null value to a reference variable that does not actually reference any valid data:

```java
Person p = null;
```

The `new` operator is used to create the actual object and assign its memory address to a reference variable.

```java
p = new Person();
```

And since we do not really care about the hexadecimal value of the memory address (nor do we really have a way of knowing it in Java), you’ll see arrows used to show that a reference variable refers to or "points to" a memory location that contains the actual object.
2 Linked Structures

The idea of creating linked structures is based on the reference variables. When you create an array, a consecutive block of memory locations is assigned to it. This allows to easy access using subscript operators.

Linked structures are composed of individual objects that are linked to one another in some way. For example, the following six objects are connected, but they do not necessarily exist in consecutive memory locations.

reference to the first object

Notice that each object has an extra "box" with an arrow coming out of it. The linked structures are composed of special kind of objects, usually referred to as nodes. Each node contains the actual data, but also a reference (possibly more than one) to another node. Linked structures get connected through these references to the next node.

2.1 Linking Nodes Together

We want to connect several nodes together to create a linked structure. Assume that our node will store String objects as the data. This means that each node contains data part (String object), and the reference to another node:

```java
class StringNode {
```
private String data;
private StringNode next;

public String getData () {
    return data;
}
public String getNext () {
    return next;
}
public void setData (String data ) {
    this.data = data;
}
public void setNext (StringNode next ) {
    this.next = next;
}

The table below shows the steps needed to create several nodes and link them together into a linked structure.

<table>
<thead>
<tr>
<th>next instruction</th>
<th>memory view</th>
</tr>
</thead>
</table>
| StringNode sn = null; StringNode current = null; | sn
| current | sn
| current | null |
| sn = new StringNode(); | sn
| current | null | null |
| | null | null | null |
current = new StringNode();

sn
null

null
current

sn.setNext( current );

sn
null

null
current

current.setNext(new StringNode() );

sn
null

null
current

null

current = current.getNext();

sn
null

null
current

null
The implementation details and algorithms assume that we have a collection of connected nodes (as shown in previous section) with a reference to the first node called head.

### 3.1 insert method

We want to insert a new node into an existing list of nodes. The steps differ slightly depending on where the new node needs to be inserted: front, back or middle of the list.
**Inserting an element at the beginning**  Assume that we have a linked list with four nodes:

![Image of a linked list with four nodes: head, null, and three nodes in between.]

We want to add a new node at the beginning of this list, i.e., in front of the very first node.

![Image of the linked list with the new node added at the beginning.]

This requires making the new node point to the current first element and then changing where the head reference points to.

![Image of the linked list with the new node added at the beginning and the head reference updated.]

**WARNING:** If you change where the head reference points to first, then the rest of the list will be lost!

These steps in code:

```java
//create new node
StringNode newNode = new StringNode();
//make the new node point to the current first node
newNode.setNext(head);
//make the head reference point to the new first node
head = newNode;
```
**Inserting an element at the end**  
Again, assume that we start with a linked list with four nodes:

![Linked List Diagram](null)

We want to add a new node at the end of this list, i.e., the current last node should point to our new node.

![Linked List Diagram](null)

Since, we only have a reference to the first node, we need to find the last node (easy to find since it is the only one pointing to null). So we have a new reference, called `current` that now points to the last node.

![Linked List Diagram](null)

Our new node should point to null, since it will become the last node in the list,
and the current last node should point to the new node.

![Diagram showing linked list with head, current, and null nodes]

In this case, the order of the last two steps does not matter. (In fact, if the StringNode constructor automatically sets the next reference to null that step can be omitted.)

These steps in code:

```java
//create new node
StringNode newNode = new StringNode();

//create the current reference and point it to the last node
StringNode current = head;
while (current.next != null)
    current = current.getNext();

//make the new node point to null
newNode.setNext(null);

//make the last node point to the new last node
current.setNext(newNode);
```
Inserting an element in the middle The previous two scenarios covered special cases. Now, we have the same initial linked list, but we want to add a node in the middle. At this point it does not matter how the actual location is selected, so let’s add it after the second node.

At this point it does not matter how the actual location is selected, so let’s add it after the second node. We need the current reference to point to the node that should be before the new node after the insertion (the exact details of how to advance the current reference from head to that particular position depend on how the position is selected).

As with inserting at the front, we need to make sure not to disconnect the rest of the list during the insertion. The new node has to point to its successor.
and then its predecessor is pointed to the new node.

These steps in code:

```java
//create new node
StringNode newNode = new StringNode();

//create the current reference and advance it to
//the node after which we want to insert
StringNode current = head;
... //depends on specifics of the problem

//make the new node point to its successor
newNode.setNext(current.getNext());

//make the current node point to the new node
current.setNext(newNode);
```
3.2 remove method

As with the insertion, the removal of a node is slightly different depending on where the node is located in the list.

Removing an element at the beginning  We start with our four-node linked list

```
head
null
null
null
```
and we want to remove the very first node (the one pointed to by the head reference).

```
head
null
null
null
```
This involves pointing the head reference to the second node. Once the first node is no longer pointed to by anything, it becomes unreachable and memory used by it will be returned to available memory set by Java garbage collector.

```
head
null
null
null
```
These steps in code:

```java
//point the head reference to the second node
if ( head != null )
    head = head.getNext();
```
Removing an element at the end  We start with our four-node linked list

and we want to remove the very last node (the one that points to null). This means that we first need to find the node right before it ...

... and make it point to null (make it the last node).

These steps in code:

```
//create a current reference and advance it to the one
//before the last node
StringNode current;
current = head;
//assume that there are at least two nodes in the list
//(otherwise we are deleting the first node)
while ( current.getNext().getNext() != null )
    current = current.getNext();

//disconnect the last node by pointing the one
//before it to null
current.setNext(null);
```
Removing an element in the middle  We start with our four-node linked list

and we want to remove a node in the middle (not the first one or the last one). We need the current reference to point to the node before the one that needs to be removed (how this is done depends on the specifics of how the node to be deleted is selected).

The node pointed to be current needs to point to the node "one away" in the list (the one that the node to be deleted is pointing to).

These steps in code:

```java
//create a current reference and advance it to the one before the node to be deleted
StringNode current;
current = head;
... //depends on specifics of the problem

//connect the node pointed to by current with the node pointed to be the node being deleted
current.setNext(current.getNext().getNext());
```
3.3 clear method

The clear() method of the list ADT empties the list. This can be accomplished by simply pointing the head reference of our list to null.

3.4 contains method

The contains() method determines if a particular data item is in the list or not. With linked lists the data items are stored inside the nodes. When implementing contains method we need to make sure that we are "looking at" the actual data items, not at the nodes.

Here is an algorithm for searching in the linked list:

make current reference equal to head
while current != null
    check if current.data is what we are after
    if yes
        return true
    advance current to the next node
    (current = current.next )
    if we reach this point we did not find what we were after so return false

3.5 indexOf method

This method may be considered inappropriate for the liked list - after all we do not use indexes to access the elements. But since the get method expects such index, we will provide it anyway. The index returned by this method does not provide any information regarding ordering between the elements. In fact, at this point we are dealing with unsorted lists.

The implementation of this method is very similar to the implementation of contains method: we need to locate the element and return its position rather than returning Boolean value indicating it was found or not.

Here is an algorithm for searching in the linked list and returning the position information:

make current reference equal to head
set index to zero
while current != null
    check if current.data is what we are after
    if yes
        return the index
    advance current to the next node
    (current = current.next )
    increment the index by one
    if we reach this point we did not find what we were after so return null (or -1 if return type is int)
3.6 get method

The get method returns the data element of the list at a particular location, call it index. Since we only have a reference to the first node, we need to jump over the first index–1 nodes and then return the data element in the index\textsuperscript{th} node.

Here is an algorithm for finding and returning index\textsuperscript{th} node in a linked list:

```java
make current reference equal to head
set counter to zero
while current != null and counter != index
    increment the counter by one
if counter == index
    return current.data
otherwise (we reached the end of the list)
    return null
```

3.7 size method

The most efficient implementation of this method would make use of an auxiliary data field that keeps track of the number of elements in the list: its value is adjusted on every deletion and insertion. The method then simply returns a value of this variable.

But as an exercise we will develop this method to compute a size of a list in two different ways: iteratively and recursively.

**Iterative algorithm** The details of an iterative algorithm were discussed in previous parts of this section. We need to move through all the nodes incrementing a counter until we find a node that is pointing to null. Here is an iterative algorithm for determining the size of a linked list:

```java
make current reference equal to head
set counter to zero
while current != null
    increment the counter by one
return the value of counter
```

**Recursive algorithm** We need to decide on the base case and the recursive step of the recursive algorithm. The smallest possible linked list is an empty list. In that case the head points to null and the size is zero. This is the base case of the algorithm. If the head reference does not point to null than we know that the size is at least 1 (since head points to an actual node) plus the size of the list that the first node points to. This is the recursive step.

Here is a recursive algorithm for determining the size of a linked list:
size ( StringNode current )
    if current == null
        return 0;
    else return 1 + size(current.next);

3.8 Implementation summary

This section discussed several different algorithms for various operations on linked list. The listing below provides a summary of what the actual class implementation could look like with a few details that we did not pay attention to before.

The Node class is copied here from before just for completeness:

class StringNode {
    private String data;
    private StringNode next;

    public String getData () {
        return data;
    }

    public String getNext () {
        return next;
    }

    public void setData (String data) {
        this.data = data;
    }

    public void setNext (StringNode next) {
        this.next = next;
    }
}
The LikedStringList has to implement all methods provided by the StringList interface, but can add its own methods.

```java
public class LinkedStringList implements StringList {
    protected StringNode head;
    protected int numOfElements;

    /* default constructor sets the head reference to null */
    public LinkedStringList () {
        head = null;
        size = 0;
    }

    /* the insert method adds an item to the end of the list */
    public void insert ( String item ) {
        // the code was described before
        // numOfElements variable needs to be updated
    }

    /* the remove method removes a node whose data matches the parameter */
    public void remove ( String item ) {
        // how can this be implemented give the removal
        // algorithms described before?
        // numOfElements variable needs to be updated
    }

    /* empties the whole list */
    public void clear ( ) {
        head = null;
        numOfElements = 0;
    }

    /* determines if a node containing the data matching the parameter is in the list */
    public boolean contains ( String item ) {
        // the algorithm described in text
    }

    /* determines the index of a node containing the data matching the parameter is in the list */
    public int indexOf ( String item );

    /* returns data item located in the index^th node */
    public String get ( int index );

    /* returns number of elements in the list */
    public int size ( ) {
        return numOfElements;
    }

    /* returns String representation of the list */
    public String toString ( ) {
        // implementation depends on the application
    }
}
```
The remaining methods are not part of the implemented interface, but will be useful in our later development of stacks and queues.

```java
/* the insert method that adds an item to the front of the list */
public void insertFront ( String item ) {
    // the code was described before
    // numOfElements variable needs to be updated
}

/* the insert method that adds an item to the back of the list */
public void insertBack ( String item ) {
    // the code was described before
    // numOfElements variable needs to be updated

    // NOTE: this is a duplicate of insert() method above
}

/* the remove method that removes an item from the front of the list */
public void removeFront ( String item ) {
    // the code was described before
    // numOfElements variable needs to be updated
}

/* the remove method that removes an item from the back of the list */
public void removeBack ( String item ) {
    // the code was described before
    // numOfElements variable needs to be updated

    // NOTE: this is a duplicate of remove() method above
}

/* determines if the list is empty or not */
public boolean isEmpty () {
    return numOfElements == 0;
}
```
4 Stacks

In implementing liked structure based stacks we are going to reuse the code developed in the previous section. The stack class implementation needs to implement the interface that we have been using previously (but this time it is written for String objects):

```java
public interface StringStack {
    public void push (String item);
    public String pop () ;
    public String peek () ;
    public String toString ();
}
```

4.1 Wrapper Implementation

The quick and dirty solution is to define a stack class that has our LinkedListStringList as a data field and calls its respective methods. This approach reuses methods of the LinkedListStringList class and wraps them in its own methods appropriate for a stack.

```java
public class LinkedListStringStack implements StringStack {
    private LinkedListStringList stack;

    /* constructor needs to create a LinkedListStringList object */
    public LinkedListStringStac () {
        stack = new LinkedListStringList ();
    }

    /* uses the insertBack() method of the list */
    public void push (String item) {
        if (item != null)
            stack.insertBack( item );
    }

    /* uses the removeBack() method of the list, but needs to store the removed item to return it */
    public String pop () {
        String item = null;
        if ( ! stack.isEmpty () ) {
            item = stack.get( stack.size () - 1);
            stack.removeBack();
        }
        return item;
    }

    /* this uses the get() method of the list */
    public String peek () {
        String item = null;
        if ( ! stack.isEmpty () )
            item = stack.get( stack.size () - 1);
        return item;
    }
}
```
4.2 Another Inefficient Implementation

We can also implement the stack class with its own head reference, numOfElements variable and the methods

- push() that is identical to insertBack(),
- pop() that is identical to removeBack(),
- peek() that used get() method to obtain the last element,
- toString().

This approach avoids making duplicate method calls, but is still very inefficient: we need to traverse the entire list for every insertion, deletion and get operation.

4.3 An Efficient Stack Implementation

Since a stack data structure only requires accesses (push, pop, peek) on one end of the liked list, the top of a stack should be at the very beginning of the list. This way every access takes only one step and there is never a reason to traverse the entire list.

An efficient linked list based stack implementation should have its own head reference, (optional) numOfElements variable and the methods

- push() that is identical to insertFront(),
- pop() that is identical to removeFront(),
- peek() that used get() method to obtain first element (without removing it),
- toString().

5 Queues

As with the stack, in implementing liked structure based queues we are going to reuse the code developed for the linked list. The queue class implementation needs to implement the interface that we have been using previously (but this time it is written for String objects):
A queue requires accesses to both ends of the list, so on one of its operations the entire list has to be traversed. Assuming that the queue is somewhat balanced (i.e. number of insertions is approximately the same as number of removals), it does not really matter which end of the list is considered the front of the queue and which is considered the back of the queue.

A possible liked list based queue implementation would have its own head reference, (optional) numOfElements variable and the methods

- enqueue() that is identical to insertBack(),
- dequeue() that is identical to removeFront(),
- toString().

We will return to a liked list based implementation of a queue when we discuss doubly linked lists. This will provide a more efficient solution than the one above that uses singly liked lists.

6 Performance Analysis

At this point we informally talked about advantages and disadvantages of both array based and linked list based implementation of things. But how the relative performance is decided in practice?

6.1 Experimentally Measuring Performance

We could run different implementations of algorithms using different data structures and compare their actual running times. This is a commonly used practice when one needs to decide the performance of a particular design on a particular machine. But this approach has many drawback:

- Machine dependent. The running times will depend on the computer’s architecture.
- Programming language dependent. Different programming languages have different efficiency of the same concepts (for example a method call overhead).
- Compiler dependent. Compilers provide different levels of optimization of the code.
- Implementer dependent. Different programmers implementing the same algorithms and data structures may make choices that result in drastically different performance (for example two different ways of using linked list based implementation of the stack discussed in earlier).
- Size dependent. Experiments may not provide information about how the algorithm handles inputs of different sizes (i.e. if the input size doubles, does the running time double, quadruple, or ??? ).

6.2 Theoretically Measuring Performance

The theoretical approach considers how the computation grows with the problem size based on the algorithm design and the chosen data structure.
**Problem Size.** We use a problem of size \( N \). This general \( N \) usually refers to the number of items that we are operating on (this may be something else depending on the problem, but for most of the application in this class it will be the number of elements).

**Computation Time.** We are interested in some general notion of how many steps it takes (number of comparisons, number of loop iterations, etc.) to apply the particular algorithm to a problem of size \( N \). The result should be a function of \( N \). There are several possibilities:

- bounded by a constant - this is a computation time that is independent of the problem size,
- proportional to \( N \): \( 0.5N, 2N, 100N, 100,000N \),
- proportional to \( \log(N) \),
- proportional to \( N^2 \),
- proportional to \( N^3 \),
- ...  
- proportional to \( 2^N \),
- ...  
- and any products and sums of the above.

The computation time also depends on luck to some extent. We want to consider three different possibilities:

- best case - usually too optimistic, but sometimes equal to the other cases,
- average case - might be hard to compute,
- worst case - might be too pessimistic and in practice occurs very rarely.

**Standard Measure of Computation Time** If the problem size \( N \) is very large, then we really do not care about the constant that may be present in one of the functions mentioned above. Order of magnitude notation \( O(f) \) tells us that the problem and/or solution is “order \( f \)” where \( f \) is one the function mentioned above after omitting the constant. Algorithm that is \( O(N^2) \) requires \( cN^2 \) number of steps in which \( c \) is some constant. The problem with order of magnitude analysis is that in some cases (especially small problem sizes) the constant may be larger than the problem size and for practical reasons should not be ignored.
### 6.3 Singly Linked List <=> Array

Assume the problem size is $N$ and that we are working with an unsorted list.

<table>
<thead>
<tr>
<th>operation</th>
<th>array implementation</th>
<th>linked list implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>access an element at location $i$</td>
<td>$O(1)$</td>
<td>$O(1)$ best case</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(N)$</td>
</tr>
<tr>
<td>find an element</td>
<td>$O(1)$ best case</td>
<td>$O(1)$ best case</td>
</tr>
<tr>
<td></td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>insert an element at the beginning</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insert an element at the end (assume we know the current number of elements)</td>
<td>$O(1)$</td>
<td>$O(N)$ if need to resize the array</td>
</tr>
<tr>
<td></td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>insert an element in the middle based on index (somewhere inside, not first or last)</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>remove an element in the front</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>remove an element in the end (assume we know the current number of elements)</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>remove an element in the middle based on index (somewhere inside, not first or last)</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>