CSE 12
Fundamental Data Structures: The Array and Linked Structures

- Implementations of the List ADT
- Properties of array and linked implementations
- Separate and nested node classes
- Singly and doubly linked lists
Introduction

• Any interface specification can be correctly implemented in many ways

• However, different correct implementations may have different performance characteristics

• It is important to know these characteristics, as they affect the performance of the resulting software
Collection ADT's

- An ADT specifies a range of values that instances of the type can have, and operations on those values.

- A collection ADT can use various data structures to implement its values.

- The data structure a collection ADT uses internally is sometimes called its *backing store*.

- For the List ADT, the backing store is typically chosen to be either an *array*, or a *linked list*. 
Array Characteristics

• An array is a homogeneous data structure: all elements are of the same type.

• The elements of an array are in adjacent memory locations.

• Because each cell has the same size, and the cells are adjacent in memory, it is possible to quickly calculate the address of any array cell, given the address of the first cell. So accessing any array cell is constant time: just one multiplication, one addition, and one memory access.

⇒ an array is a random (direct) access structure.
Linked Structure Characteristics

• Nodes in a linked structure are allocated and deallocated dynamically, as needed

• The nodes of a linked structure are created at different times, and are probably not adjacent in memory

• Even if the address of the first node in a linked structure is known, it is impossible to directly calculate the address of any other node; instead each node stores the address of the next node in the structure
  – so a node access requires visiting all previous nodes in the structure in sequence

  ⇒ a linked structure is a sequential access structure
Array Versus Linked Structures

A 10 cell array storing 3 elements.

A 3-node singly linked structure storing 3 elements.
Implementing Linked Lists

• In a *singly linked list*, each node X contains:
  – a pointer to the node immediately after X in the list; or null, or pointer to a dummy node, if X is the last node in the list
  – the data in X (or a pointer to data)

• In a *doubly linked list*, each node X contains:
  – a pointer to the node immediately after X in the list; or null, or pointer to a dummy node, if X is the last node in the list
  – a pointer to the node immediately before X in the list; or null, or pointer to a dummy node, if X is the first node in the list
  – the data in X (or a pointer to data)
Implementing Linked Lists

• Taking an object oriented approach, you will define (at least) two classes:
  • A node class
    – instances of this class will be nodes in the list
  • A list class
    – an instance of this class will contain (one or more pointers to) instances of the node class

• For each, consider the properties and behavior required
• We will first show an example of a separate node class, and later see how to define a nested node class
The structure of a node in a singly linked list

```java
public class SLNode<E> {
    private E data; // the data field
    private SLNode<E> next; // link to successor

    /**
     * Create an empty <tt>SLNode</tt> object.
     */
    public SLNode() {
        this.data = null;
        this.next = null;
    }

    /**
     * Create an <tt>SLNode</tt> that stores <tt>theElement</tt> and
     * whose successor is <tt>theSuccessor</tt>.
     * @param theElement the element to store in this node
     * @param theSuccessor this node’s successor
     */
    public SLNode( E theElement, SLNode<E> theSuccessor ) {
        this.data = theElement;
        this.next = theSuccessor;
    }
}
```
/**
 * Successor accessor
 */
public SLNode<E> getSuccessor() {
    return this.next;
}

/**
 * Successor mutator
 */
public void setSuccessor(SLNode<E> n) {
    this.next = n;
}

/**
 * Element accessor
 */
public E getData() {
    return this.data;
}

/**
 * Element mutator
 */
public void setData(E e) {
    this.data = e;
}
SLNode construction

```java
Integer i1 = new Integer(15);
SLNode<Integer> node1 = new SLNode<Integer>(i1, node2);
SLNode<Integer> node2 = new SLNode<Integer>();

Integer i1 = new Integer(15);
SLNode<Integer> node1 = new SLNode<Integer>(i1, node2);
```
Accessing the successor of a SLNode

```java
SLNode<Integer> temp = node1.getSuccessor();
```
A Singly-linked List using SLNode

public class SinglyLinkedList<E> implements java.util.List<E>{
    private SLNode<E> head; // used in options 1, 2, 3
    private SLNode<E> tail; // used only in option 2
    private int size; // number of data elements in the list

• One important issue: how to represent an empty list?
  - Need to be clear about this! It has implications for correctly coding several of the List operations

• We will consider some implementation options:
  1. head pointer points directly to first node, or null if none; no tail pointer
  2. head, tail always point to 'dummy' nodes
  3. head always points to 'dummy' node; no tail pointer
Option 1: direct head pointer

- Create initially empty singly linked list

```java
SinglyLinkedList<Integer> ls = new SinglyLinkedList<Integer>();
```

Null head pointer means empty list, so size must be 0
Option 1: direct head pointer

- After adding 4 data items to the list

Using head directly as pointer to the first node of the linked structure

Null next pointer means “end of the list”
Option 2: head and tail 'dummy' nodes

- Create initially empty singly linked list

```java
SinglyLinkedList<Integer> ls = new SinglyLinkedList<Integer>();
```

Dummy head and tail nodes always exist. Dummy head node next pointing to dummy tail node means empty list, with size 0.
Option 2: head and tail 'dummy' nodes

- After adding 4 data items to the list

Dummy head node next pointer points to first element of list
Last element of list next pointer points to same node as tail pointer
void addAtHead(E theElement) {
    1. Create the new SLNode
    SLNode<E> newnode = new SLNode<E>(theElement, null);
    2. Set the new node's next field to point where head points
    newnode.setSuccessor(head);
    3. Set head to point to the new node
    head = newnode;
    4. Increment size by 1
    size++;
}

Question: does this correctly handle cases both of adding at the head of an empty list, and of a nonempty list?
Adding at the beginning of an empty list: Step 1

1. Create the new SLNode

```
SLNode<E> newnode = new SLNode<E>( theElement, null );
```
Adding at the beginning of an empty list:
Step 2

2. Set the new node’s next field to point where head points
   newnode.setSuccessor( head );
Adding at the beginning of an empty list:

Step 3

3. Set head to point to the new node, increment size

```java
head = newnode;
size++;
```
1. Create the new SLNode

`SLNode<E> newnode = new SLNode<E>( theElement, null );`
Adding at the beginning of a nonempty list: Step 2

2. Set the new node’s next field to point where head points

```java
newnode.setSuccessor( head );
```
Adding at the beginning of a nonempty list: Step 3

3. Set head to point to the new node, increment size

```plaintext
head = newnode;
size++;
```
Operations on a singly-linked list

- Keep in mind the different options we’re considering in implementing a singly linked list:
  1. head pointer points directly to first node, or null if none; no tail pointer
  2. head, tail always point to 'dummy' nodes
  3. head always points to 'dummy' node; no tail pointer

- We have looked at adding a node at the beginning of a linked list in the context of option 1
- Next we will look at adding a node at an arbitrary index in a linked list, using option 1
Inserting an element at an index in the list (using option 1)

```java
public void add(int index, E theElement) {
    1. check index >=0 && index <= size
    if(index < 0 || index > size) throw new IOBException();
    2. advance cursor to point to node just before insertion point
    SLNode<E> cursor = head;
    while(--index > 0) cursor = cursor.getSuccessor();
    4. Create new node
    SLNode<E> newnode = new SLNode<E>(theElement, null);
    5. Set the new node's next to be same as cursor's next
    newnode.setSuccessor(cursor.getSuccessor());
    6. Set cursor's next to point to the new node
    cursor.setSuccessor(newnode);
    7. Increment size by 1
    size++;
}
```
4. Create the new SLNode

```
SLNode<E> newnode = new SLNode<E>( theElement, null );
```
Adding at index 2 in a list: Step 5

5. Set newnode's next to be same as cursor's next

newnode.setSuccessor(cursor.getSuccessor());
Adding at index 2 in a list: Step 6

6. Set cursor's next to point to the new node, increment size

```java
cursor.setSuccessor(newnode);
size++;
```
Adding at an index: cases to test

- Does that add method work in all cases:
  - index == 0 ?
  - index == size ?
  - index == 0 when list is empty (so index==size also)?
  - 0 < index < size ?

- If not, how can you fix it so it does work?
- Would “option 2” (using dummy head and tail nodes) make the implementation easier?
Inserting an element at the beginning of the list (using option 2)

void addAtHead(E theElement) {
    1. Create the new SLNode
    SLNode<E> newnode = new SLNode<E>(theElement, null);
    2. Set the new node's next to be same as dummy head node's next
    newnode.setSuccessor(head.getSuccessor());
    3. Set dummy head's next to point to the new node
    head.setSuccessor(newnode);
    4. Increment size by 1
    size++;
}

Question: does this correctly handle the special case of adding at the head of an empty list?
Deleting a node at an index: pseudocode

```java
1  public E remove ( int p ) {
2      1. Verify that p is within the linked list bounds
3      2. Move cursor to node with index p – 1
4      3. Let target = cursor's successor: the node to remove
5          SLNode<E> target = cursor.getSuccessor();
6          4. Save the target’s data, to return it
7          E element = target.getData();
8          5. Make cursor's next point to target's successor node.
9              (This removes target node from the list)
10         cursor.setSuccessor( target.getSuccessor() );
11         6. Decrement size
12         size--;
13         7. Return the element stored in the target node
14         return element;
15    }
```
Removing at index 2 in a list: Step 3

3. Let target = cursor's successor: the node to remove

```java
SLNode<E> target = cursor.getSuccessor();
```
Removing at index 2 in a list: Step 5

5. Make cursor's next point to target's successor node.

```java
cursor.setSuccessor( target.getSuccessor() );
size--;```

Deleting a Node: Cleanup

- “Splicing around” the node to be deleted, as shown, removes it from the linked list...
- …a traversal from the head of the list will no longer encounter the node at all
- Furthermore, consider the pointer variable `target`
  - It is a local variable in the remove method
  - It will be destroyed when the method returns
  - Then the program has no 'live' pointers to the deleted node, and Java's garbage collector can reclaim it
- Nothing more needs to be done...!
Using a nested class for the node class

• The previous example used a separate, top-level public class SLNode<E> to define a singly-linked node type, with accessor and mutator methods to manipulate the node

• Another approach is to use a private static nested class to define a node, and access instance variables of node objects directly, instead of using accessor/mutator methods
  – This is a good example of making implementation details (in this case, the entire definition of the list’s node class) private!
Using a nested class for the node class

```java
import java.util.*;

/**
 * Outer class: List<E>
 */
public class SinglyLinkedList<E> implements java.util.List<E> {
    private Node<E> head; // pointer to first element (not dummy)
    private int size; // number of nodes in this List

    /**
     * private static nested class: Node<T>
     */
    private static class Node<T> {
        private T data; // the data field
        private Node<T> next; // link to successor
        private Node(T data) { // nested class constructor
            this.data = data;
        }
    }

    public SinglyLinkedList() { // outer class constructor
        head = null; // null head means empty list
        size = 0;
    }
```

void addAtHead(E theElement) {
    1. Create the new Node
    Node<E> newnode = new Node<E>(theElement);
    2. Set the new node’s next field to point where head is pointing
    newnode.next = head;
    3. Set head to point to the new node
    head = newnode;
    4. Increment size by 1
    size++;
}

Question: does this correctly handle the special case of adding to an empty list?
Adding at the beginning of an empty list: Step 1

1. Create the new Node

Node<E> newnode = new Node<E>(theElement);
Adding at the beginning of an empty list: Step 2

2. Set the new Node’s next field to point where head points

newnode.next = head;
Adding at the beginning of an empty list: Step 3

3. Set head to point to the new Node, increment size

```
head = newnode;
size++;
```
Adding at the beginning of a nonempty list: Step 1

1. Create the new Node

Node<E> newNode = new Node<E>(theElement);
Adding at the beginning of a nonempty list: Step 2

2. Set the new Node’s next field to point where head points

newnode.next = head;
Adding at the beginning of a nonempty list: Step 3

3. Set head to point to the new Node, increment size

head = newnode;
size++;
Doubly-Linked Lists

- The singly-linked list is unidirectional.
- At the expense of an additional link (and the consequent code complexity) we can have a bidirectional list.

We need to add a second “link” attribute to the node definition.
Circular Doubly-Linked Lists

• We considered a singly-linked list implementation using dummy head and tail nodes

• Because of the bidirectional nature of a doubly-linked list, it is possible to use just one dummy node, which is both a head and a tail node!

• The first node in the list has its ‘predecessor’ pointer pointing to the dummy node, and the last node in the list has its ‘successor’ pointer pointing to the dummy node

• Thus the list is ‘circular’, in that following successor (or predecessor) pointers will eventually get you back to where you started
A Circular Doubly-Linked List

- This circular doubly linked list has size 3
- From first to last, the data elements are: ‘A’, ‘B’, ‘C’
Time Complexity for Array Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>read at an index, worst case</td>
<td>O(1)</td>
</tr>
<tr>
<td>add/remove (at the end of the array)</td>
<td>O(1)</td>
</tr>
<tr>
<td>add/remove (in the interior of the array)</td>
<td>O(n)</td>
</tr>
<tr>
<td>resize</td>
<td>O(length of new array)</td>
</tr>
<tr>
<td>find by index</td>
<td>O(1)</td>
</tr>
<tr>
<td>find by target</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
## Time Complexity for Linked Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Singly Linked</th>
<th>Doubly Linked</th>
</tr>
</thead>
<tbody>
<tr>
<td>read at an index, worst case</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>add/remove (at the head)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>add/remove (at the tail)</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>add/remove (in the interior of the list)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>resize</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>find by index</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>find by target</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
Next time

- Algorithm analysis vs. measurement
- Timing an algorithm
- Average and standard deviation
- Improving measurement accuracy

Reading: Gray, Ch 2