

CS211, LECTURE 17 FOUNDATIONAL DS REVISITED

ANNOUNCEMENTS:

READING:

DS&SD: Chap 4, 7, 13.1-13.4

DS&A: Chap 4

OVERVIEW:

- Variables
- Arrays
- Lists
- **ArrayLists** and **Vectors**
- Trees (actually, this happens next lecture)

1. Motivation

- ☐ Introduce more operations
 - so far, mostly searching
 - want to add/delete/reorganize/clone...
- ☐ Rethink design
 - alternative approaches to foundational ADTs
 - ex) lists: just head or head & tail?
 - more practice!
- ☐ Generic Programming
 - clean up, standardize code
 - demonstrate API again
- ☐ Implement Other ADTs
 - ADTs will use foundational data structures
 - ex) **StackAsArray**, **StackAsList**

2. Variables

- ☐ Simplest Data Structure
 - PrimitiveType var = primitive*
ClassType var = object
- ☐ Time and Space?
 - constant
- ☐ Not really an ADT
 - ADT is info and its operations
 - So, is an entire program an ADT?
- ☐ API
 - Collection interface

3. Arrays

- ☐ Syntax
 - type[][].... var = new type[size1][size2]....*
- ☐ Aspects
 - static structure, so space is linear
 - time to access an element: constant
- ☐ Some API info
 - build array with reflection: see **Array** in **java.lang.reflect**
 - predefined array operations: see **Arrays** in **java.util.Arrays**:
 - **asList**: convert to API's **List**
 - **binarySearch**: binary search (many types)
 - **equals**: tests if 2 arrays are “deeply” equal
 - **fill**: put a value in specified places in array
 - **sort**: modified merge sort, stable

3.1 Using java.lang.Object and java.lang.System (Arrays1)

```
int[] x1 = {1,2,3};
int[] x2 = (int[]) x1.clone();
int[] x3 = {1,2,3};
int[] x4 = new int[3];
System.arraycopy(x1,0,x4,0,3);

System.out.println("Test 1: "+x1.equals(x1));
System.out.println("Test 2: "+x1.equals(x2));
System.out.println("Test 3: "+x1.equals(x3));
System.out.println("Test 4: "+x1.equals(x4));

System.out.println("Test 5: "+x1);
System.out.println("Test 6: "+x2);
System.out.println("Test 7: "+x3);
System.out.println("Test 8: "+x4);

System.out.println("Test 9: "+x1.getClass());

/* output:
Test 1: true
Test 2: false
Test 3: false
Test 4: false
Test 5: [I@1ba34f2
Test 6: [I@1ea2dfe
Test 7: [I@17182c1
Test 8: [I@13f5d07
Test 9: class [I
*/
```

3.2 Using java.util.Arrays (Arrays2)

```
import java.util.Arrays;

int[] x1 = {1,2,3};
int[] x2 = (int[]) x1.clone();
int[] x3 = {1,2,3};
int[] x4 = new int[3];
System.arraycopy(x1,0,x4,0,3);
Integer[] x5 = { new Integer(3), new Integer(1),
                 new Integer(2)};

System.out.println("Test 1: "+Arrays.equals(x1,x1));
System.out.println("Test 2: "+Arrays.equals(x1,x2));
System.out.println("Test 3: "+Arrays.equals(x1,x3));
System.out.println("Test 4: "+Arrays.equals(x1,x4));
System.out.println("Test 5: "+Arrays.asList(x5));
System.out.println("Test 6: "+
                  Arrays.binarySearch(x5,new Integer(1)));

Arrays.sort(x5);
for (int i=0;i<x5.length;i++)
    System.out.print(x5[i]+" ");
System.out.println();

/* output:
Test 1: true
Test 2: true
Test 3: true
Test 4: true
Test 5: [3, 1, 2]
Test 6: 1
1 2 3
*/
```

4. Lists

☐ Motivation/Issues

- arrays are great, but are only static
- need dynamic structure, which changes size
- random access (only linear search)

☐ List

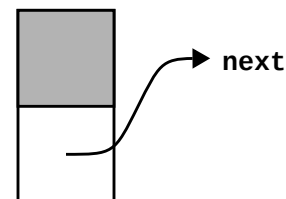
- linear DS, collections of linked cells
- cells contain info (Object) and link(s) to other cells

☐ Related Types

- singly-linked: link to another cell (**prev** or **next**)
- doubly-linked: link to both **prev** and **next** (easier to work with costs more heap storage)
- circular: head connects to tail
- degenerate tree (tree with only 1 “side”)
- related ADTs: set (no order, all unique), bag (also called multiset: no order, maybe repeats)

4.1 List Node

☐ “Standard” Node or Cell



☐ Node Code (see **TestListNode**)

```
public class ListNode {

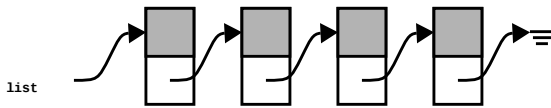
    private Object item;
    private ListNode next;

    public ListNode(Object o, ListNode n) {
        item = o;
        next = n;
    }

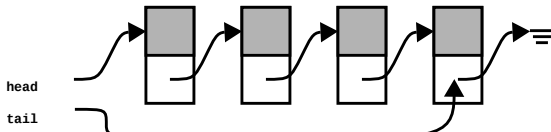
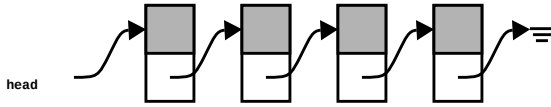
    public Object getItem() { return item; }
    public ListNode getNext() { return next; }
    public void setItem(Object o) { item = o; }
    public void setNext(ListNode n) { next = n; }
    public String toString() {
        String temp = item.toString();
        if (next != null)
            temp += " " + next.toString();
        return temp;
    }
}
```

4.2 Singly-Linked Lists

☐ The Gist



☐ Variations (see DS&A, 4.3 for more)



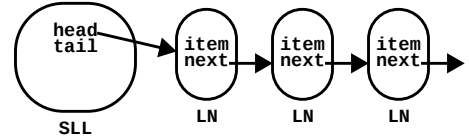
☐ How to choose?

- space considerations
- time consideration

4.3 List Class

☐ Structure

- List class: contain links to **head** and/or **tail**, sentinels
- List node: represent each item in list



☐ Choice of list class?

- DS&SD: see Figs 7.9, 7.10; DS&A: see 4.3.1

☐ Code Snippet

```
public class SLL {  
  
    private ListNode head;  
    private ListNode tail;  
  
    public SLL() { }  
  
    // getters, setters  
    // operations (remove, add, ... )  
}
```

4.4 List Operations

☐ List as ADT...

☐ Common operations from API:

void add(int index, Object o)

Inserts the specified element at the specified position in this list

boolean add(Object o)

Appends the specified element to the end of this list

void clear()

Removes all of the elements from this list

boolean contains(Object o)

Returns true if this list contains the specified element.

boolean equals(Object o)

Compares the specified object with this list for equality.

Object get(int index)

Returns the element at the specified position in this list.

int indexOf(Object o)

Returns the index in this list of the first occurrence of the specified element, or -1 if this list does not contain this element.

boolean isEmpty()

Returns true if this list contains no elements.

Object remove(int index)

Removes the element at the specified position in this list

boolean remove(Object o)

Removes the first occurrence in this list of the specified element

int size()

Returns the number of elements in this list.

4.5 Example Code (SLL)

```
public class SLL {  
    // see other code from (10)  
  
    // Add elem to end of list:  
    public boolean add(Object o) {  
        ListNode tmp = new ListNode(o, null);  
        if (head==null) head = tmp;  
        else tail.setNext(tmp);  
        tail = tmp;  
        return true;  
    }  
  
    // Remove 1st occurrence of o:  
    public boolean remove(Object o) {  
        ListNode current = head;  
        ListNode prev = null;  
        while(current != null &&  
            !o.equals(current.getItem())) {  
            prev = current;  
            current = current.getNext();  
        }  
  
        if (current==null)  
            return false;  
        if (current==head)  
            head = current.getNext();  
        else  
            prev.setNext(current.getNext());  
        if (current == tail)  
            tail = prev;  
        return true;  
    }  
}
```

```

// Purge all elements from list:
public void clear() {
    head = tail = null;
}

// Check if a node contains o:
public boolean contains(Object o) {
    ListNode current = head;
    while (current != null) {
        if (o.equals(current.getItem()))
            return true;
        else
            current = current.getNext();
    }
    return false;
}

// Return number of nodes:
public int size() {
    int count = 0; // start as empty
    ListNode current = head;
    while (current != null) {
        count++;
        current = current.getNext();
    }
    return count;
}

// Check if list is empty:
public boolean isEmpty() {
    return (head==null);
}

```

4.6 Using SLL (TestSLL)

```

public class TestSLL {
    public static void main(String[] args) {

        SLL list = new SLL();
        list.add("D");
        list.add("C");
        list.add("B");
        list.add("A");

        System.out.println("List: " + list);
        System.out.println("Size: " + list.size());
        System.out.println("E? " + list.contains("E"));
        System.out.println("A? " + list.contains("A"));
        System.out.println("Del A: " + list.remove("A"));
        System.out.println("List: " + list);
        System.out.print ("Gone! ");
                                list.clear();

        list.display();
        System.out.println("Empty? " + list.isEmpty());
    }
}

/* Output:
List:  D C B A
Size:  4
E?     false
A?     true
Del A: true
List:  D C B
Gone!  null
Empty? true
*/

```

4.7 Alternative (ListRecursion)

```

class List2 {
    private ListNode head;
    public List2() { }
    public String toString() { return head.toString(); }

    public void add(Object o) { head = add(o, head); }
    private ListNode add(Object o, ListNode n) {
        if (n == null) return new ListNode(o, n);
        else { n.next = add(o, n.next); return n; }
    }

    public void del(Object o) { head = del(o, head); }
    private ListNode del(Object o, ListNode n) {
        if (n == null) return n;
        if (o.equals(n.item)) return n.next;
        else { n.next=del(o,n.next); return n;}
    }

    private class ListNode {
        public Object item;
        public ListNode next;
        public ListNode(Object o, ListNode n) {
            item = o;
            next = n;
        }
        public String toString() {
            String temp = item.toString();
            if (next != null) temp+=" "+next.toString();
            return temp;
        }
    }
}

```

```

public class ListRecursion {
    public static void main(String[] args) {

        List2 list = new List2();
        list.display();

        list.add("A"); System.out.println(list);
        list.add("B"); System.out.println(list);
        list.add("C"); System.out.println(list);

        list.del("A"); System.out.println(list);
        list.del("B"); System.out.println(list);
        list.del("C"); System.out.println(list);
    }
}

/* Output:
null
A
A B
A B C
B C
C
null
*/

```

4.8 API for Lists

- ❑ package? import `java.util.*`
- ❑ Interfaces

- **Collection, List**



- see also **ListIterator**
- ❑ Abstract classes
 - **AbstractList** (partially implements **List** interface)
 - **AbstractSequentialList** (extends A.L.)
- ❑ Concrete Classes
 - **LinkedList** (extends A.S.L.)
 - **ArrayList, Vector** (extends A.L.)
 - see API for complete specs

5. ArrayLists and Vectors

- ❑ Motivation
 - want something with dynamic size of lists
 - want something with random access of arrays
- ❑ **ArrayList** vs. **Vector**
 - **Vector** is synchronized, **ArrayList** isn't
 - huh? see threads; prevent chaotic access to data structure
- ❑ API
 - **Vector** and **ArrayList** implement **List** so, refer there for method headers
 - Why bother with Vector?
Legacy code: early concept of collection in Java
- ❑ Example
 - see Lecture 3

6. Strings

- ❑ **String** as ADT?
- ❑ Object: **toString**
 - default: `getClass().getName() + '@' + Integer.toHexString(hashCode())`
 - **hashCode**: coming up in Search Structures
- ❑ **CharSequence** interface
 - implemented by **String**
 - `charAt(index), length(), toString(), subSequence(start, end)`
- ❑ **String**
 - `java.lang`
 - implements **Comparable, CharSequence**
- ❑ **StringBuffer**
 - mutable **Strings**
 - **insert, append** are common operations

❑ StringTokenizer

- `java.util`
- breaks **String** into collection of substrings
- uses delimiters: special characters to indicate breaks
- default delimiters: `" \t\r\n\f"`
- see also **StreamTokenizer**

```
// StringTokenizer example:
String s = "abc2de fg.hi";
String delims = "1234567890.\\\"\\' ";

StringTokenizer st1 = new StringTokenizer(s);
StringTokenizer st2 = new StringTokenizer(s, delims);
while (st1.hasMoreTokens())
    System.out.println(st1.nextToken());
while (st2.hasMoreTokens())
    System.out.println(st2.nextToken());
```

❑ StringCharacterIterator

- `java.text`
- iterator for **Strings**

❑ IO

- **StringBuffer, StringWriter, ...**

7. Suggested Exercises

Write a class that implements a singly-linked list as an array.

Write list classes for variations (a), (c), and (d) in DS&A, 4.3.

Implement the API's **List** interface methods for the given **SLL** class. See if you can write recursive and iterative versions for the methods instead of using just one approach.

Write a binary search method for an **ArrayList** and/or **Vector**.

Create a **MyArray** class (see previous notes) that includes a useful **toString** method.

Write a program that takes a body of text and counts the number of letters (without spaces, numbers, and punctuation), words, and sentences.