CS61B, Spring 2003 **Discussion #5** **Amir Kamil**
UC Berkeley **2/24/03**

Topics: Arrays, Linked Lists, Queues, Stacks

1 Arrays

Arrays are the simplest data structure possible, and are just aggregates of homogeneous items. They are very similar to variables, except that all array elements share the same variable name, but have unique indices. Each array position acts just like a variable. It may be assigned to, passed as an argument (by value), or accessed like a variable. Arrays are typed, and all positions in an array have the same type.

Since array positions behave like variables, primitive arrays hold primitive values and object arrays hold pointers, like their variable counterparts. Unlike variables (at least local variables), it is possible to use array positions before they are explicitly initialized. When an array is constructed, the positions are initialized to zero or false for primitive arrays, and `null` for object arrays.

Arrays are actually objects, so you can call methods on them and access their fields. The only methods they have are those inherited from Object, so the methods are not very useful. All arrays have one field of particular importance, their `length` field. Arrays are actually fixed-size, so if you initialize a size k array, k will be its size for its entire life. The `length` field tells you this size.

1.1 Array Usage

- Declaration:

  ```
  <type>[] <name>;
  ```

  Examples:

  ```
  int[] numbers;
  String[] sentences;
  ```

- Creation:

  ```
  new <type>[<size>]
  ```

  Examples:

  ```
  int[] numbers = new int[4];
  String[] sentences = new String[18];
  ```

  Another way:

  ```
  int[] numbers = {1, 2, 3};
  ```

  This creates an int array of size three with the above elements. Note that this form can only be used when declaring the array.

- Access:
  To access the nth element in an array:

  ```
  <name>[n]
  ```

  Example:

  ```
  numbers[2] = 4;
  ```

- As shown above, array elements can be assigned to.
- Array indexing starts at 0, i.e. the first element of `numbers` is `numbers[0]`.
- Legal array access is limited to n between 0 and one less than the size of the array, inclusive. To get the size of `numbers`, use `numbers.length`.
- Arrays are objects and therefore have methods and fields. They inherit all methods and fields from the `Object` class.
- Primitive arrays (e.g. `int[]`) store values of their respective types, while object arrays (e.g. `Computer[]`) store references to objects of their respective types. Positions in object arrays that don't point to an object hold `null` pointers.

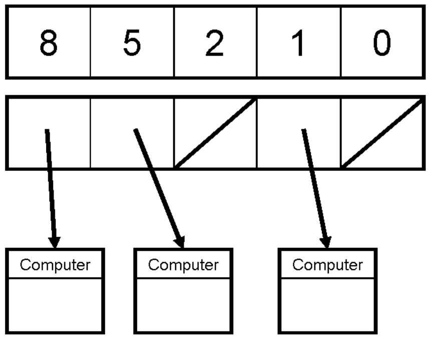

Figure 1: An `int` and a `Computer` array of size 5.

1.2 Multi-Dimensional Arrays

It is possible to declare "multi-dimensional" arrays, like `int[][] x`. Such an array is actually an array of arrays. Arrays can hold any types, and multi-dimensional arrays are the special case when the type happens to be another array. For a general array `x`, `x[i]` returns the `i`th element in the array. The same is true for two dimensional arrays: `x[i]` would return the `i`th one dimensional array that it contains. This can be generalized to higher dimensions. For example, an `int[][][]` is an array that holds arrays that hold `int` arrays.

The most common two-dimensional arrays are those in which the subarray all have the same length. Such an array can be created so that both the outer array and the inner arrays are created. For example, an initialization `int[][] x = new int[3][4];` would create an array that holds three `int` arrays, each of size four. Those arrays do not have to be independently initialized.

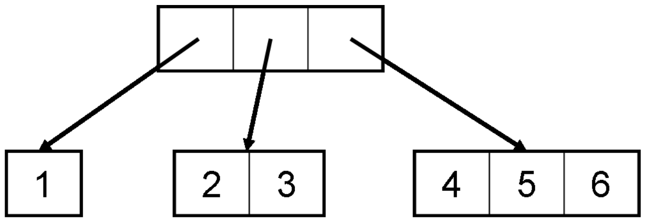

Figure 2: A two-dimensional array.

2 Lists

A list is a sequence of data items, and is implemented using multiple `nodes` that each contain a single object and references to other nodes in the list. Lists can be *singly-linked* or *doubly-linked*. In singly-linked lists, each node only points to the next node in the list. In doubly-linked lists, each node points to both the previous and next nodes in the list. The following could be an implementation of a node in a singly-linked list:

```
public class ListNode {
  public ListNode next; // the next node in the list
  public int data;
  public ListNode(int data, List next) {
    this.data = data;
    this.next = next;
  }
}
```

Some lists abstract away their implementation details and only provide a container class that has access to individual nodes, but doesn't provide the user access to those nodes. The container has direct access to the first node in the list, and indirect access to the rest of them through the first one. Here's an example:

```
public class SList {

  private ListNode head;

  public SList() {
    head = null;
  }

  // insert a value at the front of the list
  public void insertFront(int value) {
    head = new ListNode(value, head);
  }

  // remove the front of the list, return its value
  public int removeFront() {
    int value = head.value;
    head = head.next;
    return value;
  }

  // other methods

}
```

With such a container, a list can also be made *circular*, meaning that the last node points to the head of the list rather than to `null`. This makes some of the linked list operations easier to implement. The Java `LinkedList` class is a container for a doubly-linked circular list.

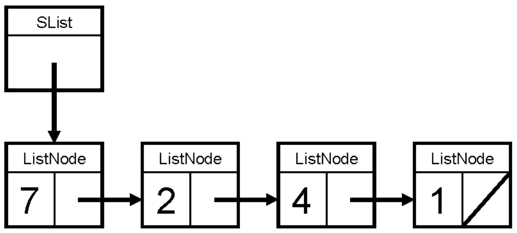

Figure 3: An `SList` of size 4.

There are some basic data structures that can be derived from linked lists, such as the *queue* and the *stack*. Most linked lists can actually be used as such structures without implementing a new class, but we will do so to only allow the operations that these data structures allow.

2.1 Queues

Queues are *FIFO* (first in, first out) data structures that support two operations: `enqueue` adds an element into a queue, and `dequeue` removes the first element to be added among the elements in the queue. A queue can easily be implemented by using a `LinkedList`:

```
public class Queue {

  private LinkedList list;

  public Queue() {
    list = new LinkedList();
  }

  // Adds an element to this queue.
  public void enqueue(Object o) {
    list.addLast(o);
  }

  // Removes and returns the first element of all those in this queue to be added
  // to this queue.
  public Object dequeue() {
    return list.removeFirst();
  }

  // Returns the size of this queue.
  public int size() {
    return list.size();
  }

  // Returns whether or not this queue is empty.
  public boolean isEmpty() {
    return list.isEmpty();
  }

}
```

2.2 Stacks

Stacks are *FILO* (first in, last out) data structures that also support two operations: `push` adds an element to a stack, and `pop` removes the last element to be added among the elements in the stack. The structure is named as such because it is like a stack of books. You can add a book to the top, or remove the top book, but you can't change anything in the middle of the stack (at least, for a large stack of books, or a stack of large books). Like a queue, a stack can easily be implemented using a `LinkedList`:

```
public class Stack {

  private LinkedList list;

  public Stack() {
    list = new LinkedList();
  }

  // Adds an element to this stack.
  public void push(Object o) {
    list.addFirst(o);
  }

  // Removes and returns the last element of all those in this stack to be added
  // to this stack.
  public Object pop() throws EmptyStackException {
    if (list.isEmpty()) {
      throw new EmptyStackException();
    }
    return list.removeFirst();
  }

  // Returns the size of this stack.
  public int size() {
    return list.size();
  }

  // Returns whether or not this stack is empty.
  public boolean isEmpty() {
    return list.isEmpty();
  }
}
```