Last Time

• More Comparable Examples
• Better Sorting Methods
  • MergeSort
  • QuickSort
• More Flexible Comparing: Comparator Interface
Today: Linked Lists

- Class extension
  - Abstract base classes
  - Concrete extension classes
- List: A general-purpose structure
- Implementing Lists with linked structures
  - Singly and Doubly Linked Lists
Class Specialization

- Classes can extend other classes
  - Inherit fields and method bodies
- By extending other classes, we can create specialized sub-classes
- Java supports class extension/specialization
- Java enforces type-safety: Objects behave according to their type
  - Some checks are made at compile-time
  - Some checks are made at run-time
- We’ll first use this feature to factor out code
Abstract Classes

• Note: All of our Card implementations code toString() in identical fashion.
• It’s good to be able to “factor out” common code so that it only has to be maintained in one place
• Abstract classes to the rescue….

• An abstract class allows for a partial implementation
• We can then extend it to a complete implementation
• Let’s do this with our cards.
  • Examine CardAbstract.java….
• Note: Can’t use “new“ with abstract types!
Abstract Classes

Notes from CardAbstract class example

• CardAbstract implements Card (partially)
• CardAbstract is declared to be abstract
  • It contains the implementation of toString(), equals(), and compareTo()

How do the full implementations (CardRankSuit, etc) change?

• They are declared to extend CardAbstract
• They don’t need to say “implements Card”
• They don’t contain the toString() method
  • They inherit that method from CardAbstract
  • But could override that method if desired
Extending Concrete Classes

Let’s call a class *concrete* if it is not abstract

We can extend concrete classes

Example: Adding a point count to a Card

- Suppose we wanted to add a point value to each of the playing cards in CardRankSuit

- We extend that class

  ```java
  class CardRankSuitPoints extends CardRankSuit {
  ... }
  ```

- This new class can now contain additional instance variables and methods

- Let’s look at the code for CardRankSuitPoints.java...
CardRankSuitPoints Notes

• Constructor calls CardRankSuit constructor using super
• We can override methods---e.g., toString()
• Can use a CardRankSuitPoints object wherever we use a Card
  • But! Can only use new features (getPoints()) if the object is declared to be of type CardRankSuitPoints

CardRankSuitPoints c1 = new CardRankSuitPoints(Rank.ACE, Suit.CLUBS, 4);
int p1 = c1.getPoints(); // Legal
Card c2 = new CardRankSuitPoints(Rank.ACE, Suit.CLUBS, 4);
int p2 = c2.getPoints(); // Bad! c2 is of type Card
int p3 = ((CardRankSuitPoints) c2).getPoints(); // Legal

• Java enforces type-safety: An variable of type X can only be assigned a value of type X or of a type that extends X
The Card Classes Hierarchy
Pros and Cons of Vectors

**Pros**
- Good general purpose list
- Dynamically Resizeable
- Fast access to elements
  - `vec.get(387425)` finds item 387425 in the same number of operations regardless of vec’s size

**Cons**
- Slow updates to front of list (why?)
- Hard to predict time for add (depends on internal array size)
- Potentially wasted space

Today we look at another way to store data: Linked Lists
interface List {
    size()
    isEmpty()
    contains(e)
    get(i)
    set(i, e)
    add(i, e)
    remove(i)
    addFirst(e)
    getLast()
    .
    .
    .
}

• Flexible interface
• Can be used to describe many different types of lists
• It’s an interface…therefore it provides no implementation
• Vector implements List
• Other implementations are possible
  • SinglyLinkedList
  • CircularlyLinkedList
  • DoublyLinkedList
Linked List Basics

- There are two key aspects of Lists
  - Elements of the list
  - The list itself

- Visualizing lists
Linked List Basics

• List nodes are recursive data structures
• Each “node” has:
  • A data value
  • A “next” value that identifies the next element in the list
  • Can also have “previous” that identifies the previous element (“doubly-linked” lists)
• What methods does Node class need?
SinglyLinkedLists

• Terminology alert!
  • SinglyLinkedListNode = SLLE in these notes
  • SLLE = Node in structure5 (and in Ch 9)
  • Let’s look at SLLE.java
  • How about SinglyLinkedList?
    • SinglyLinkedList = SLL in my notes

• What would addFirst(E d) look like?
  • getFirst()?
  • addLast(E d)? (more interesting)
  • getLast()?
More SLL Methods

- How would we implement:
  - `get(int index)`, `set(E d, int index)`
  - `add(E d, int index)`, `remove(int index)`

- Left as an exercise:
  - `contains(E d)`
  - `clear()`

- Note: E is value type
Get and Set

public E get(int index) {  
    Assert.pre(index < size() - 1, "Index out of range");  
    // or should we return null in above case?  
    SLLN finger = head;  
    for (int i=0; i<index; i++) {  
        finger = finger.next();  
    }  
    return finger.value();  
}

public E set(E d, int index) {  
    Assert.pre(index < size() - 1, "Index out of range");  
    // Same question!  
    SLLN finger = head;  
    for (int i=0; i<index; i++) {  
        finger = finger.next();  
    }  
    E old = finger.value();  
    finger.setValue(d);  
    return old;  
}
public E remove(int index) {
    if (index >= size()) return null;

    E old;

    if (index == 0) return removeFirst();
    else if (index == size() - 1) return removeLast();

    else {
        SLLN finger = head;
        for (int i = 0; i < index - 1; i++) { // stop one before index
            finger = finger.next();
        }
        old = finger.next().value();
        finger.setNext(finger.next().next());
        count--;
        return old;
    }
}
public void add(E d, int index) {
    if(index > size()) return null;
    E old;

    if (index==0) { addFirst(d); }

    else if (index==size()) { addLast(d); }

    else {
        SLLN finger = head;
        SLLN previous = null;
        for (int i=0; i<index; i++) {
            previous = finger;
            finger = finger.next();
        }
        SLLN elem = new SLLN(d, finger);
        previous.setNext(elem); // new "ith" item added after i-1
        count++;
    }
}
Linked Lists Summary

- Recursive data structures used for storing data
- More control over space use than Vectors
- Easy to add objects to front of list
- Components of SLL (SinglyLinkedList)
  - head, elementCount
- Components of SLLN (Node):
  - next, value
Vectors vs. SLL

- Compare performance of
  - size
  - addLast, removeLast, getLast
  - addFirst, removeFirst, getFirst
  - get(int index), set(E d, int index)
  - remove(int index)
  - contains(E d)
  - remove(E d)
SLL Summary

- SLLs provide methods for efficiently modifying front of list
  - Modifying tail/middle of list is not quite as efficient
- SLL runtimes are consistent
  - No hidden costs like Vector.ensureCapacity()
  - Avg and worst case are always the same
- Space usage
  - No empty slots like vectors
  - But keep extra reference for each value
    - overhead proportional to list length
      - (but this is constant and predictable)
Food for Thought: SLL Improvements to Tail Ops

• In addition to Node head and int elementCount, add Node tail reference to SLL

• Result
  • addLast and getLast are fast
  • removeLast is not improved
    • We need to know element before tail so we can reset tail pointer

• Side effects
  • We now have three cases to consider in method implementations: empty list, head == tail, head != tail
  • Think about addFirst(E d) and addLast(E d)
CircularlyLinkedLists

- Use `next` reference of last element to reference head of list
- Replace `head` reference with `tail` reference
- Access head of list via `tail.next`
- **ALL** operations on head are fast!
- `addLast()` is still fast
- Only modest additional complexity in implementation
- Can “cyclically reorder” list by changing `tail` node
- Question: What’s a circularly linked list of size 1?
DoublyLinkedLists

- Keep reference/links in **both** directions
  - previous and next
- DoublyLinkedListListNode instance variables
  - DLLN next, DLLN prev, E value
- Space overhead is proportional to number of elements
- **ALL** operations on tail (including removeLast) are fast!
- Additional work in each list operation
  - Example: add(E d, int index)
  - Four cases to consider now: empty list, add to front, add to tail, add in middle
```java
public class DoublyLinkedNode<E>
{
    protected E data;
    protected DoublyLinkedNode<E> nextElement;
    protected DoublyLinkedNode<E> previousElement;

    // Constructor inserts new node between existing nodes
    public DoublyLinkedNode(E v, DoublyLinkedNode<E> next, DoublyLinkedNode<E> previous)
    {
        data = v;
        nextElement = next;
        if (nextElement != null)  // point next back to me
            nextElement.previousElement = this;
        previousElement = previous;
        if (previousElement != null) // point previous to me
            previousElement.nextElement = this;
    }
```
public void add(int i, E o) {
    Assert.pre((0 <= i) && (i <= size()),
            "Index in range.");
    if (i == 0) addFirst(o);
    else if (i == size()) addLast(o);
    else {
        // Find items before and after insert point
        DoublyLinkedNode<E> before = null;
        DoublyLinkedNode<E> after = head;
        // search for ith position
        while (i > 0) {
            before = after;
            after = after.next();
            i--;
        }
        // before, after refer to items in slots i-1 and i
        // continued on next slide
DoublyLinkedList Add Method

// Note: Still in “else” block!
// before, after refer to items in slots i-1 and i

// create new value to insert in correct position
// Use DLN constructor that takes parameters
// to set its next and previous instance variables
DoublyLinkedListNode<E> current =
    new DoublyLinkedListNode<E>(o, after, before);

count++; // adjust size

public E remove(E value) {
    DoublyLinkedNode<E> finger = head;
    while (finger != null &&
           !finger.value().equals(value))
        finger = finger.next();
    if (finger == null) return null;

    // fix next field of previous element
    if (finger.previous() != null)
        finger.previous().setNext(finger.next());
    else head = finger.next();

    // fix previous field of next element
    if (finger.next() != null)
        finger.next().setPrevious(finger.previous());
    else tail = finger.previous();
    count--;
    return finger.value();
}
The structure5 package has a hierarchical structure

• A collection of interfaces that describe---but do not implement---the functionality of one or more data structures

• A collection of abstract classes provide partial implementations of one or more data structures
  • To factor out common code or instance variables

• A collection of concrete (fully implemented) classes to provide full functionality of a data structure
AbstractList Superclass

abstract class AbstractList<E> implements List<E> {
    public void addFirst(E element) { add(0, element); }
    public E getLast() { return get(size() - 1); }
    public E removeLast() { return remove(size() - 1); }
}

• AbstractList provides some of the list functionality
  • Code is shared among all sub-classes (see Ch. 7 for more info)
    public boolean isEmpty() { return size() == 0; }
  • Concrete classes (SLL, DLL) can override the code implemented in AbstractList

• Abstract classes in general do not implement every method
  • For example, size() is not defined although it is in the List interface

• Can’t create an “AbstractList” directly

• Other lists extend AbstractList and implement missing functionality as needed
class Vector extends AbstractList {
    public int size() { return elementCount; }
}
The Structure5 Universe (almost)
The Structure5 Universe (so far)