Announcements

• Lab today!
  • After mid-term we’ll have some “non-partner” labs
  • It’s Lab5 not Lab 4

• Mid-term exam is Wednesday, October 17
  • During your normal lab session
  • You’ll have approximately 1 hour & 45 minutes (if you come on time!)
  • Closed-book: Covers Chapters 1-7 & 9, handouts, and all topics up through Linked Lists
  • A “sample” mid-term and study sheet will be available online
  • Review session: Monday, Oct. 15, 7:00-8:00pm TCL 203
Last Time

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

- Linked List Wrap-Up
- The structure hierarchy so far
- Linear Structures
  - The Linear Interface (LIFO & FIFO)
  - The AbstractLinear and AbstractStack classes
- Stack Implementations
  - StackArray, StackVector, StackList,
- Stack applications
  - Expression Evaluation
  - PostScript: Page Description & Programming
  - Mazerunning (Depth-First-Search)
DoublyLinkedLists

- Keep reference/links in **both** directions
  - previous and next
- DoublyLinkedListNode 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
public class DoublyLinkedListNode<E>
{
    protected E data;
    protected DoublyLinkedListNode<E> nextElement;
    protected DoublyLinkedListNode<E> previousElement;

    // Constructor inserts new node between existing nodes
    public DoublyLinkedListNode(E v,
                                  DoublyLinkedListNode<E> next,
                                  DoublyLinkedListNode<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
// 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
DoublyLinkedList<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();
}
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 still fast: $O(1)$ time
- `addLast()` is now fast – $O(1)$ time
- Only modest additional complexity in implementation
- Can “cyclically reorder” list by changing `tail` node
- Question: What’s a circularly linked list of size 1?
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
- Concrete list classes extend AbstractList, implementing missing functionality
  class Vector extends AbstractList {
      public int size() { return elementCount; }
  }

• 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
• Concrete list classes extend AbstractList, implementing missing functionality
  class Vector extends AbstractList {
      public int size() { return elementCount; }
  }
The Structure5 Universe (almost)
The Structure5 Universe (so far)
The Structure5 Universe (soon)
Linear Structures

• What if we want to impose *access restrictions* on our lists?
  • I.e., provide only one way to add and remove elements from list
  • No longer provide access to middle

• Key Examples: Order of removal depends on order elements were added
  • LIFO: Last In First Out
  • FIFO: First In First Out
Examples

• **FIFO: First In – First Out (Queue)**
  • Line at dining hall
  • Data packets arriving at a router

• **LIFO: Last In – First Out (Stack)**
  • Stack of trays at dining hall
  • Java Virtual Machine stack
The Structure5 Universe (next)
Linear Interface

• How should it differ from List interface?
  • Should have fewer methods than List interface since we are limiting access …

• Methods:
  • Inherits all of the Structure interface methods
    • add(E value) – Add a value to the structure.
    • E remove(E o) – Remove value o from the structure.
      – But this is awkward---why?
    • int size(), isEmpty(), clear(), contains(E value), …
  • Adds
    • E get() – Preview the next object to be removed.
    • E remove() – Remove the next value from the structure.
    • boolean empty() – same as isEmpty()
Linear Structures

• Why no “random access”?  
  • I.e., no access to middle of list
• More restrictive than general List structures  
  • Less functionality can result in  
    • Simpler implementation  
    • Greater efficiency
• Approaches  
  • Use existing structures (Vector, LL), or  
  • Use underlying organization, but simplified
Stacks

• Examples: stack of trays or cups
  • Can only take tray/cup from top of stack
• What methods do we need to define?
  • Stack interface methods
• New terms: push, pop, peek
  • Only use push, pop, peek when talking about stacks
  • Push = add to top of stack
  • Pop = remove from top of stack
  • Peek = look at top of stack (do not remove)
Notes about Terminology

• When using stacks:
  • pop = remove
  • push = add
  • peek = get

• In Stack interface, pop/push/peek methods call
  add/remove/get methods that are defined in Linear
  interface

• But “add” is not mentioned in Stack interface (it is
  inherited from Linear)

• Stack interface extends Linear interface
  • Interfaces extend other interfaces
  • Classes implement interfaces
Stack Implementations

- Array-based stack
  - int top, Object data[ ]
  - Add/remove from index top
  + all operations are $O(1)$
  - wasted/run out of space

- Vector-based stack
  - Vector data
  - Add/remove from tail
  +/- most ops are $O(1)$ (add is $O(n)$ in worst case)
  - potentially wasted space

- List-based stack
  - SLL data
  - Add/remove from head
  + all operations are $O(1)$
  +/- $O(n)$ space overhead
  (no “wasted” space)
Stack Implementations

- **structure5.StackArray**
  - int top, Object data[ ]
  - Add/remove from index top
  - + all operations are O(1)
  - – wasted/run out of space

- **structure5.StackVector**
  - Vector data
  - Add/remove from tail
  - +/- most ops are O(1) (add is O(n) in worst case)
  - – potentially wasted space

- **structure5.StackList**
  - SLL data
  - Add/remove from head
  - + all operations are O(1)
  - +/- O(n) space overhead
  - (no “wasted” space)
Summary Notes on The Hierarchy

- Linear interface extends Structure
  - add(E val), empty(), get(), remove(), size()
- AbstractLinear (partially) implements Linear
- AbstractStack class (partially) extends AbstractLinear
  - Essentially introduces “stack-ish” names for methods
    - push(E val) is add(E val), pop() is remove(), peek() is get()
- Now we can extend AbstractStack to make “concrete” Stack types
  - StackArray<E>: holds an array of type E; add/remove at high end
  - StackVector<E>: similar, but with a vector for dynamic growth
  - StackList<E>: A singly-linked list with add/remove at head
  - We implement add, empty, get, remove, size directly
    - push, pop, peek are then indirectly implemented
The Structure5 Universe (so far)