CSCI 136
Data Structures & Advanced Programming

Jeannie Albrecht

Lecture 15

March 14, 2014

Administrative Details

- Midterm 1
  - Please don't discuss yet!
  - Probably won't get them back until after break…
- Labs 3 and 4 are graded
  - You'll get them back during lab next week
- Lab 6 will be posted soon (in case you want to get a head start)

2

Last Time

- Started talking about SinglyLinkedLists and SinglyLinkedListElements

3

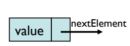
Today's Outline

- Wrap up SinglyLinkedLists
- Briefly discuss DoublyLinkedLists
- Begin learning about our next class of data structures: stacks and queues
  - You'll use a stack in lab next week…

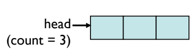
4

Recap: SinglyLinkedLists

- How would we implement SinglyLinkedListElement?
  - SinglyLinkedListElement = SLLE in my notes
  - SLLE = Node in the book (in Ch 9)
  - (previous slide)



- How about SinglyLinkedList?
  - SinglyLinkedList = SLL in my notes
  - (implement on board)



- What would addFirst(E value) look like? (on board)
- getFirst()? (on board)
- addLast(E value)? getLast()? (see next slide)

5

```
public void addLast(E value) {
    count++;

    //two cases to consider: empty list, or non-empty list
    //case 1: empty list
    if (head == null) {
        head =  new SLLE<E>(value);
    }

    //case 2: non-empty list.  Must follow "next" pointers to tail
    else {
        //"finger" is pointer to possible tail
        SLLE<E> finger = head;
        while (finger.next() != null) {
            //keep following next pointers until you find the tail
            finger = finger.next();
        }

         //finger is now pointing to the tail. Add new element to it
        finger.setNext( new SLLE<E>(value) );
    }
}
```

```
public E getLast() {
    SLLE<E> finger = head;
    //keep following next pointers until you find the tail
    while (finger != null && finger.next() != null) {
            finger = finger.next();
    }
    return finger.value();
}
```

6

More SLL Methods

- How would we implement:
  - get(int index), set(E value, int index)
  - add(E value, int index), remove(int index)
    - removeLast() is just remove(size() - 1)
    - removeFirst() is just remove(0)
- Left as an exercise:
  - contains(E value)
  - clear()

7

Get and Set

```
public E get(int index) {
   Assert.pre(index < size() - 1, "Index out of range");

   SLLE<E> finger = head;
   for (int i=0; i<index; i++){
        finger = finger.next();
   }
   return finger.value();
}
```

```
public E set(E value, int index) {
   Assert.pre(index < size() - 1, "Index out of range");

   SLLE<E> finger = head;
   for (int i=0; i<index; i++){
        finger = finger.next();
   }
   E old = finger.value();
   finger.setValue(value);
   return old;
}
```

8

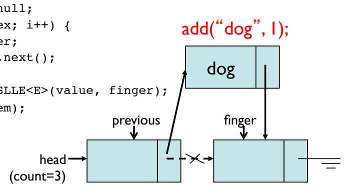
Add

```
public void add(E value, int index) {
   Assert.pre(index <= size(), "Invalid index");
   E old;

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

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

   else {
        SLLE<E> finger = head;
        SLLE<E> previous = null;
        for (int i=0; i<index; i++) {
            previous = finger;
            finger = finger.next();
        }
        SLLE<E> elem = new SLLE<E>(value, finger);
        previous.setNext(elem);
        count++;
   }
}
```



Remove

```
public E remove(int index) {
   Assert.pre(index < size() - 1, "Invalid index");
   E old;

   if (index==0) {
        old = head.value();
        head = head.next();
        count--;
        return old;
   }

   else {
        SLLE<E> 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;
   }
}
```

10

Linked Lists Summary

- Recursive data structures used for storing data
- Waste less space than vectors
  - Why?
  - Because they are not implemented using arrays
- Easy to add objects to front of list
- Components of SLL:
  - head, elementCount
- Components of SLLE:
  - next, value

11

Vectors vs. SLL

(Big-O runtime for Object o, int i)

Operation	Vector	SLL
size()	O(1)	O(1)
addLast(o)	O(1) or O(n)(if resize)	O(n)
removeLast()	O(1)	O(n)
getLast()	O(1)	O(n)
addFirst(o)	O(n)	O(1)
removeFirst()	O(n)	O(1)
getFirst()	O(1)	O(1)
get(i)	O(1)	O(n)
set(i, o)	O(1)	O(n)
remove(i)	O(n)	O(n)
contains(o)	O(n)	O(n)
remove(o)	O(n)	O(n)

12

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: O(n) overhead (but this is constant and predictable)

13

Food for Thought: SLL Traversal

- We need a way to iterate through Lists

```
for (int i=0; i<list.size(); i++) {
  System.out.println(list.get(i));
}
```

- What is runtime for Vectors?
- For SLL this is $O(n^2)$! (Why?)
- We'll learn about iterators soon…
  - Iterators provide O(n) traversal of SLLs
- For now, suppose we just want to efficiently access head **and** tail of list

14

SLL Improvements to Tail Ops

- We want to improve SLLs so tail ops are not O(n)
- In addition to SLLE head and int elementCount, add SLLE tail reference (instance variable) to SLL class
- Result
  - addLast is O(1), getLast is O(1)
  - removeLast is…
  - …still O(n)!
    - 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(Object d) and addLast(Object d)

15

DoublyLinkedLists

- Keep reference/links in **both** directions
  - previous and next
- DoubleLinkedListElement instance variables
  - DLLE next, DLLE prev, Object value
- Space overhead is still O(n)
- ALL operations on tail (including removeLast) are O(1)!
- Additional complexity in each list operation
  - Example: add(Object d, int index)
  - Four cases to consider now: empty list, add to front, add to tail, add in middle

16

List (SLL/Vector) Final Summary

- addFirst, removeFirst
- addLast, removeLast
- get/set
- contains
- Bottom line:
  - Choose list implementation based on needs of application!

17

Moving on…

18

Linear Structures

- What if we want to impose an **ordering** to our lists?
- I.e., provide only one way to add and remove elements from list
  - No longer provide access to middle
- Order of removal depends on the order elements were added
  - LIFO: Last In First Out
  - FIFO: First In First Out

19

Examples

- FIFO
  - Line (queue) at grocery store
  - Line at dining hall (hopefully)
- LIFO
  - Stack of trays at dining hall
  - Stack of cups
  - Deck of cards

20

Linear Interface

- We need another interface!
  - Should have less methods than List interface since we are limiting access a bit…
- Methods:
  - add(E value) - Add a value to the structure.
  - boolean empty() - Returns true iff the structure is empty.
  - E get() - Preview the next object to be removed.
  - E remove() – Remove the next value from the structure.
  - int size() - Returns the number of elements in the linear structure.

21

Linear Structures

- **No "random access" to list elements!**
  - This means no access to middle of list
- More restrictive than general List structures
  - More implementation freedom
  - More efficient for *some* uses
  - More choices to think about when building our programs

22

Stacks

- Examples: stack of trays, stack of cups
  - People can only take trays/cups 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)

23

Note 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" does not exist in Stack interface (it is inherited from Linear)
- Stack interface **extends** Linear interface
  - Interfaces *extend* other interfaces
  - Classes *implement* interfaces

24

Stack Implementations

- Stack array
  - int top, Object data[ ] — + all operations are O(1)
  - Add/remove from index top — – wasted/run out of space
- Stack Vector
  - Vector data — +/– most ops are O(1) (add is O(n) in worst case)
  - Add/remove from tail — – potentially wasted space
- Stack List
  - SLL data — + all operations are O(1)
  - Add/remove from head — +/– O(n) space overhead (no "wasted" space)

25

Stack Implementations

- structure5.StackArray
  - int top, Object data[ ] — + all operations are O(1)
  - Add/remove from index top — – wasted/run out of space
- structure5.StackVector
  - Vector data — +/– most ops are O(1) (add is O(n) in worst case)
  - Add/remove from tail — – potentially wasted space
- structure5.StackList
  - SLL data — + all operations are O(1)
  - Add/remove from head — +/– O(n) space overhead (no "wasted" space)

26

Evaluating Arithmetic Expressions

- Computer processes use stacks to evaluate arithmetic expressions
- Example: x*y+z
  - First rewrite as xy*z+ (we'll look at this rewriting process in more detail soon)
  - Then:
    - push x
    - push y
    - mult (pop twice, multiply, push result)
    - push z
    - add (pop twice, add, push result)

27