CSCI 136
Data Structures & Advanced Programming

Jeannie Albrecht
Lecture 13
March 10, 2014

Administrative Details

- Lab 4 due today
- Midterm on Wednesday at 1pm in Wege
  - You'll have ~120 minutes to complete exam (designed for 90 minutes)
  - Review session tomorrow night: 9:30pm-10:30pm - TCL 202
  - Covers book, lab, lecture material **through today**
  - Closed book and notes
  - Study guide posted on Handouts page
- Optional Lab 5 will be posted later today
  - No TAs this week
- Sample exam (and solns) posted on Handouts page
- I'll be in my office from 1:30 – 3:00 today and during lecture time on Wednesday
- No class on Wednesday!

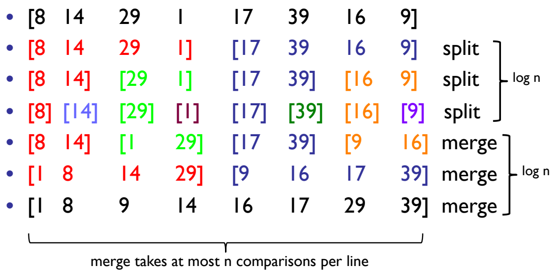
Last Time

- (Almost) finished discussing sorting
  - BubbleSort, InsertionSort, SelectionSort – O(n^2)
  - MergeSort – O(n log n)

Today's Outline

- Wrap up QuickSort
- Begin learning about Lists

Recall Merge Sort = O(n log n)



- [8 14 29 1 17 39 16 9]
- [8 14 29 1] [17 39 16 9] split
- [8 14] [29 1] [17 39] [16 9] split
- [8] [14] [29] [1] [17] [39] [16] [9] split
- [8 14] [1 29] [17 39] [9 16] merge
- [1 8 14 29] [9 16 17 39] merge
- [1 8 9 14 16 17 29 39] merge

log n

log n

merge takes at most n comparisons per line

Quick Sort

- Quick sort is designed to behave much like Merge sort, without requiring extra storage space

Merge Sort	Quick Sort
Split list in half	Split list into sublists
Sort halves	Sort sublists
Merge halves	Combine sorted sublists

Recall Merge Sort

```
private static void mergeSortRecursive(Comparable data[],
                     Comparable temp[], int low, int high) {
   int n = high-low+1;
   int middle = low + n/2;
   int i;

   if (n < 2) return;
   // move lower half of data into temporary storage
   for (i = low; i < middle; i++) {
       temp[i] = data[i];
   }
   // sort lower half of array
   mergeSortRecursive(temp,data,low,middle-1);
   // sort upper half of array
   mergeSortRecursive(data,temp,middle,high);
   // merge halves together
   merge(data,temp,low,middle,high);
}
```

Quick Sort

```
public void quickSortRecursive(Comparable data[],
                       int low, int high) {
    // pre: low <= high
    // post: data[low..high] in ascending order
        int pivot;
        if (low >= high) return;

       /* 1 - place pivot */
        pivot = partition(data, low, high);
       /* 2 - sort small */
       quickSortRecursive(data, low, pivot-1);
       /* 3 - sort large */
       quickSortRecursive(data, pivot+1, high);
}
```

We no longer need to merge!

Partition

1. Put first element (pivot) into sorted position
2. All to the left of "pivot" are smaller and all to the right are larger
3. Return index of "pivot"

Look at an example…

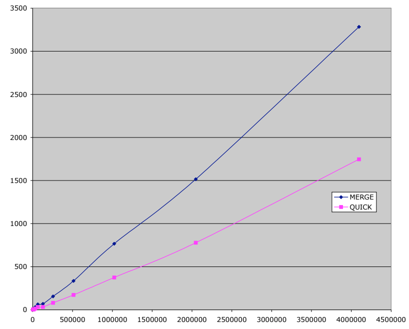
Partition

```
int partition(int data[], int left, int right) {
  while (true) { //continue until we return right or left
    while (left < right && data[left] < data[right])
      right--;
    if (left < right) {
      swap(data,left++,right);
    } else {
      return left;
    }

    while (left < right && data[left] < data[right])
      left++;
    if (left < right) {
      swap(data,left,right--);
    } else {
      return right;
    }
  }
}
```

Complexity

- Time:
  - Partition is O(n)
  - If partition breaks list exactly in half, same as merge sort, so O(n log n) on average
  - If data is already sorted, partition splits list into groups of 1 and n-1, so O(n^2) in worst case
- Space:
  - O(n) (which is better than O(2n) for merge sort)

Merge vs. Quick



Food for Thought…

- How to avoid picking a bad pivot value?
  - Pick median of 3 elements for pivot
- Combine selection sort with quick sort
  - For small n, selection sort is faster
  - Switch to selection sort when elements is <= 7
  - Switch to selection/insertion sort when the list is almost sorted (and partitions are very unbalanced)

Sorting Wrapup

	Time	Space
Bubble	$O(n^2)$ Best: O(n) - if it stops	O(n)
Insertion	$O(n^2)$ Best: O(n)	O(n)
Selection	$O(n^2)$	O(n)
Merge	O(n log n)	O(2n) = O(n)
Quick	O(n log n) Worst: $O(n^2)$	O(n)

Vector Review:
Pros and Cons of Vectors

Pros

- Good general purpose list
- Fast access to elements
  - Vec.get(15) finds element 15 in O(1) time
- Dynamically resizable

Cons

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

Today we're going to look at another way to store data using **LinkedLists.**

List Interface

```
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 (vectors, linked lists, etc)
- It's an interface…therefore it provides no implementation

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 general purpose list functionality
  - Code is shared among all sub-classes (i.e., classes that extend it, see Ch. 7)
  - For example, Vector and SinglyLinkedList both extend AbstractList
- Abstract classes in general do not implement every method in an interface
  - Abstract classes are *partial* implementations of interfaces
  - For example, size() is not defined in AbstractList although it is in the List interface
  - Can always override methods in AbstractList later if necessary
- Can't create an "AbstractList" directly
- Other lists extend AbstractList and implement missing functionality as needed

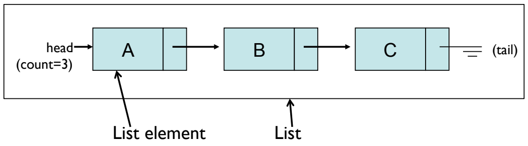
```
public class Vector<E> extends AbstractList<E> {
    public int size() { return elementCount; }
}
```

Lists

- General concept for storing/organizing data
- Vectors are good, but not perfect in all classes
  - Some updates are slow
    - i.e., Add to beginning of vector
  - Wasted space
  - Hard to know exact performance
    - (Resizing can be expensive)
- We are going to explore other types of Lists
  - SinglyLinkedList
  - DoublyLinkedList

LinkedList Basics

- There are two key concepts in LinkedLists
  - Elements of the list (data, next)
  - The list itself (head, count, maybe tail)
- Visualizing lists

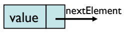


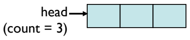
LinkedList Basics

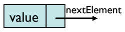
- LinkedLists are a recursive data structure
- Each "node" (LinkedListElement) has:
  - A data value
  - A "next" pointer that points to the next element in the list
  - Sometime a "previous" pointer that points to previous element, but not always (only in doubly linked lists)

SinglyLinkedLists

- How would we implement SinglyLinkedListElement?
  - SinglyLinkedListElement = SLLE in my notes
  - SLLE = Node in the book (in Ch 9)
  - (next slide)
- How about SinglyLinkedList?
  - SinglyLinkedList = SLL in my notes
  - (implement on board)
- What would addFirst(E value) look like?
- getFirst()?
- addLast(E value)?
- getLast()?



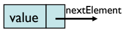


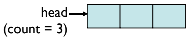


```
public class SinglyLinkedListElement<E> {

  protected E data; // value stored in this element
  protected SinglyLinkedListElement<E> nextElement; // ref to next

  public SinglyLinkedListElement(E v, SinglyLinkedListElement<E> next) {
        data = v;
        nextElement = next;
  }

  public SinglyLinkedListElement(E v) {
        this(v,null);
  }

  public SinglyLinkedListElement<E> next() {
        return nextElement;
  }

  public void setNext(SinglyLinkedListElement<E> next) {
        nextElement = next;
  }

  public E value() {
        return data;
  }

  public void setValue(E value) {  //doesn’t return old value this time
        data = value;
  }
}
```

SinglyLinkedLists

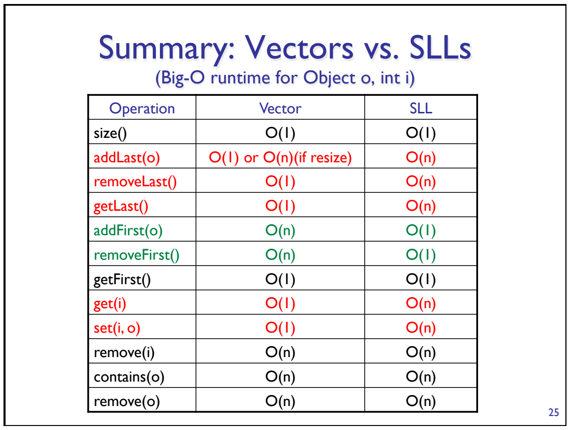
- How would we implement SinglyLinkedListElement?
  - SinglyLinkedListElement = SLLE in my notes
  - SLLE = Node in the book (in Ch 9)
  - (previous slide)
- How about SinglyLinkedList?
  - SinglyLinkedList = SLL in my notes
  - (implement on board)
- What would addFirst(E value) look like? (on board)
- getFirst()? (on board)
- addLast(E value)? getLast()? (see next slide)





```
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();
}
```



Summary: Vectors vs. SLLs

(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)