CSC 222: Computer Programming II

Spring 2005

Queues and simulation

- queue ADT
  - enqueue (offer), dequeue (remove), peek, empty, size
- `java.util.Queue` interface, `java.util.LinkedList` class
- application: bank simulation
- linked list structures
  - nodes & references/pointers, singly-linked lists, doubly-linked lists
- `LinkedList` vs. `ArrayList` tradeoffs

1

Lists & queues

recall: a *stack* is a simplified list

- add/delete/inspect at one end
- useful in many real-word situations (e.g., delimiter matching, run-time stack)

queue ADT

- a *queue* is another kind of simplified list
- add at one end (the back), delete/inspect at other end (the front)

DATA: sequence of items
OPERATIONS: enqueue (add/offer at back), dequeue (remove off front), peek at front, check if empty, get size

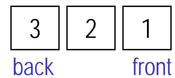
these are the ONLY operations allowed on a queue
— queues are useful because they are simple, easy to understand
— each operation is O(1)

2

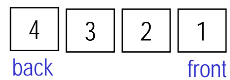
Queue examples

- line at bank, bus stop, grocery store, …
- printer jobs
- CPU processes
- voice mail

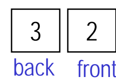
a queue is also known as
- first-in-first-out (FIFO) list



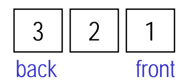
enqueue (offer): adds item at the back



dequeue (remove): removes item at the front



peek: returns item at the front → 1



Queue exercise

- start with empty queue
- ENQUEUE 1
- ENQUEUE 2
- ENQUEUE 3
- PEEK
- ENQUEUE 4
- DEQUEUE
- DEQUEUE
- PEEK
- ENQUEUE 5

`Queue` interface

a queue is a common data structure, with many variations

- Java provides a `Queue` interface
- also provides several classes that implement the interface (with different underlying implementations/tradeoffs)

`java.util.Queue<T>` interface

```
public boolean offer(T newItem);
public T remove();
public T peek();
public boolean isEmpty();
public int size();
```

`java.util.LinkedList<T>` implements the `Queue` interface

```
Queue<Integer> numQ = new LinkedList<Integer>();

for (int i = 1; i <= 10; i++) {
  numQ.offer(i);
}

while ( !numQ.isEmpty() ) {
  System.out.println(numQ.peek());
  numQ.remove();
}
```

```
Queue<Integer> q1 = new LinkedList<Integer>();
Queue<Integer> q2 = new LinkedList<Integer>();

for (int i = 1; i <= 10; i++) {
  q1.offer(i);
}

while ( !q1.isEmpty() ) {
  q2.offer(q1.peek());
  q1.remove();
}

while ( !q2.isEmpty() ) {
  System.out.println(q2.peek());
  q2.remove();
}
```

5

Queues and simulation

queues are especially useful for simulating events

e.g., consider simulating a small-town bank

- 1 teller, customers are served on a first-come-first-served basis
- at any given minute of the day, there is a constant probability of a customer arriving
- the length of a customer's transaction is a random int between 1 and some MAX

```
What is the time duration (in minutes) to be simulated? 10
What percentage of the time (0-100) does a customer arrive? 30

2: Adding customer 1 (job length = 4)
2:   Serving customer 1 (finish at 6)
4: Adding customer 2 (job length = 3)
5: Adding customer 3 (job length = 1)
6:     Finished customer 1
6:   Serving customer 2 (finish at 9)
9: Adding customer 4 (job length = 3)
9:     Finished customer 2
9:   Serving customer 3 (finish at 10)
10:     Finished customer 3
10:   Serving customer 4 (finish at 13)
13:     Finished customer 4
```

6

OOP design of bank simulation

what are the objects/actors involved in the bank simulation?
or more generally, in any service center?

Customer
- get the customer's ID
- get the customer's arrival time
- get the customer's job length

Server (Teller)
- get the server's ID
- start serving a customer
- get information on the customer being served
- check to see when the server will be free
- finish serving the customer

ServiceCenter
- get the current time
- add a customer
- do business (serve customer if available)
- check to see if any customers remain

7

Bank simulation classes

given this general scheme, can design the classes

will worry about the data fields & implementation details later

```
public class Customer
{
  . . .

  public Customer(int arrivalTime) { … }
  public int getID() { … }
  public int getArrivalTime() { … }
  public int getJobLength() { … }
}
```

```
public class Server
{
  . . .

  public Server() { … }
  public int getID() { … }
  public void startCustomer(Customer c, int time) { … }
  public Customer getCustomer() { … }
  public int busyUntil() { … }
  public void finishCustomer() { … }
}
```

```
public class ServiceCenter
{
  . . .

  public ServiceCenter() { … }
  public int getTime() { … }
  public void addCustomer() { … }
  public void doBusiness() { … }
  public boolean customersRemaining() { … }
}
```

8

BankSimulator class

```
import java.util.Scanner;

/**
 * Class that simulates a single-teller bank serving customers.
 *   @author Dave Reed
 *   @version 4/18/05
 */
public class BankSimulator
{
    public static void main(String[] args)
    {
              Scanner input = new Scanner(System.in);

              System.out.print("What is the time duration (in minutes) to be simulated?
");
              int maxTime = input.nextInt();
              System.out.print("What percentage of the time (0-100) does a customer
arrive? ");
              int arrivalProb = input.nextInt();

              ServiceCenter bank = new ServiceCenter();
        while (bank.getTime() <= maxTime || bank.customersRemaining()) {
            if (bank.getTime() <= maxTime && customerArrived(arrivalProb)) {
                bank.addCustomer();
            }
            bank.doBusiness();
        }
          }

          private static boolean customerArrived(int prob)
          {
              return (Math.random()*100 <= prob);
          }
}
```

9

Customer class

what data fields are needed to implement the methods?

let's do it!

```
public class Customer
{



  public Customer(int arrivalTime)
  {

  }

  public int getID()
  {

  }

  public int getArrivalTime()
  {

  }

  public int getJobLength()
  {

  }
}
```

10

Server class

what data fields are needed to implement the methods?

let's do it!

```
public class Server
{


  public Server() {

  }
  public int getID() {

  }
  public void startCustomer(Customer c, int time) {

  }
  public Customer getCustomer() {

  }
  public int busyUntil() {

  }
  public void finishCustomer() {

  }
}
```

11

ServiceCenter class

what data fields are needed to implement the methods?

```
public class ServiceCenter
{
  . . .

  public ServiceCenter() { … }
  public int getTime() { … }
  public void addCustomer() { … }
  public void doBusiness() { … }
  public boolean customersRemaining() { … }
}
```

HW6: implement this class

- constructor initializes the data fields
- `getTime` returns the current time in the simulation (starts at 0, increments on each step)
- `addCustomer` adds a new customer to the waiting queue (& displays a message)
- `customerRemaining` returns true if a customer is currently being served
- `doBusiness` does one step in the simulation
  - increments the current time
  - if the teller finishes with a customer, removes them (& displays a message)
  - if the teller is free and there is a customer waiting, starts serving them (& displays a message)

you will also add code for maintaining and displaying statistics on the simulation

12

`LinkedList` class

recall: the `LinkedList` class implements the `Queue` interface
- methods: `offer, remove, peek, size, isEmpty, …`

it also implements the `List` interface (as does `ArrayList`)
- methods: `add, remove, get, set, size, isEmpty, …`

`LinkedList` has a different underlying representation than `ArrayList`
→ different performance characteristics

- LinkedList: can add/remove/inspect at either front or back in O(1)
  but, add/remove/inspect in middle is O(N)
  (no direct access. must traverse to reach a value)
- ArrayList: can inspect anywhere, add/remove from back in O(1)
  but, add/remove from front or middle is O(N)
  (have direct access, but add/remove requires shifting)

13

In-class exercise

```
public class Person {
    int leftHand;
    Person rightHand
}
```

note: recursive definition
(a Person contains a Person)

is this OK?

idea: chain people together
- each person stores a number in their left hand
- right hand points to the next person in the chain

note: consecutive people in the chain do not have to be consecutive in the room

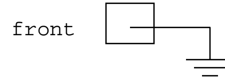
to construct a chain…
- need a reference to the front of the chain
- the end of the list must be clear – use a special value (null) to mark the end

`ADD TO FRONT, DELETE FROM FRONT, DISPLAY ALL, SEARCH, …`

14

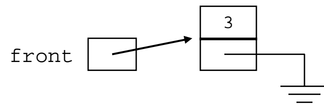
Simple linked list code

```
Person front = null;
```



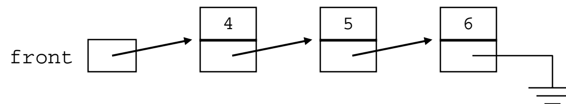
to add to the front of an empty list:

```
front = new Person();          // allocate new Person &
                               //   assign it to front
front.leftHand = 3;            // store 3 in left hand

front.rightHand = null;        // store null in right hand
```



the objects in a linked list (here, Persons) are known in general as *nodes*

Displaying the contents of a linked list



naïve attempt:

```
while (front != null) {
  System.out.println(front.leftHand);
  front = front.rightHand;
}
```

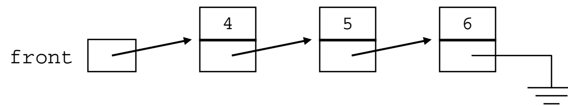
PROBLEMS?

if you lose the only pointer to a node, it becomes inaccessible! better:

```
Person step = front;
while (step != null) {
  System.out.println(step.leftHand);
  step = step.rightHand;
}
```

or could use a for loop

Adding to the front



in general, adding to the front of an arbitrary list:

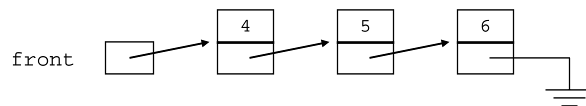
```
Person temp = new Person();    // allocate new Person

temp.leftHand = 3;             // store the data

temp.rightHand = front;        // assign front to right
                               //   hand of new Person
front = temp;                  // make new Person the
                               //   front
```

note: when you add a node to the linked list, the *physical* location of that node in memory is unimportant, the references/pointers between nodes *logically* connect them

→ so, no need to shift entries when you add at the front

17

Deleting from the front



simply:

```
front = front.rightHand;
```

what happens to the old front node?

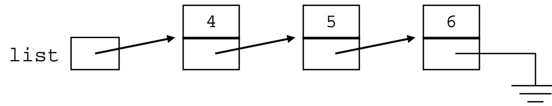
recall: if you lose the only reference to a node, that node is lost for good

- in this case, we don't need the deleted node anymore
- the Java interpreter will eventually reclaim lost memory, so no harm done

- again, no need for shifting when deleting from the front
  updating the front reference/pointer is sufficient to keep track of new front node

18

Adding/deleting in the middle



must find the location (traverse from the front)

- to delete a node from the middle, need a reference to the node before it.

```
previous.rightHand = previous.rightHand.rightHand;
```

- to add a node in the middle, need a reference to the node before it.

```
Person temp = new Person();
temp.leftHand = ???;
temp.rightHand = previous.rightHand;

previous.rightHand = temp;
```

19

`SimpleLinkedList` class

here, define a generic linked list class, uses inner class definition to hide `ListNode`

```
public class SimpleLinkedList<T>
{
    private class ListNode<T>
    {
        public T data;
        public ListNode<T> next;
    }

    private ListNode<T> front;
    private int numNodes;

    public SimpleLinkedList()
    {
        front = null;
        numNodes = 0;
    }

    public T getFirst()
    {
        return front.data;
    }

    public void addFirst(T item)
    {
        ListNode<T> newNode = new ListNode<T>();
        newNode.data = item;
        newNode.next = front;
        front = newNode;
        numNodes++;
    }

    . . .
```

```
    . . .

    public T removeFirst()
    {
        if (front == null) {
            return null;
        }
        else {
            T value = front.data;
            front = front.next;
            numNodes--;
            return value;
        }
    }

    public T get(int index)
    {
        ListNode<T> step = front;
        for (int i = 0; i < index; i++) {
            step = step.next;
        }
        return step.data;
    }

    public int size()
    {
        return numNodes;
    }
}
```

20

SimpleLinkedList, stacks & queues

note: our SimpleLinkedList class is sufficient for implementing a stack

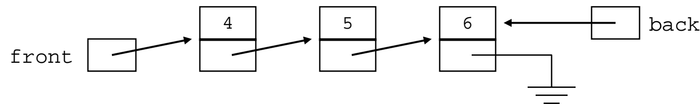
- can add, inspect, and remove at same end in O(1) time

technically, could also provide queue operations

- can already inspect and remove at the front
- need to be able to add at the back

- SOLUTION 1: traverse all the way to the back and add → O(N)

- SOLUTION 2: maintain a 2nd reference to the back of the list
  must initialize to null, update as nodes are added and removed
  advantage is that adding to the back can be done in O(1) time HOW?



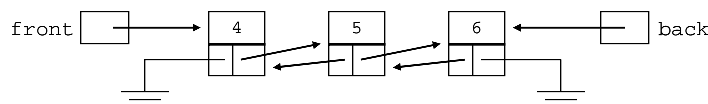
21

java.util.LinkedList class

Java's LinkedList is more general and robust than SimpleLinkedList

- recall: want to be able to add/remove/inspect from either end in O(1) time

- this is accomplished by making the list bidirectional (a.k.a., doubly-linked list)
- each node stores 2 references: one to the next node & one to the previous node



- *advantages:* can add/remove/inspect either end in O(1)
  given a reference to a node, can remove it or insert before/after in O(1)
  when accessing in middle, can start at either end (whichever is closer)

- *disadvantage:* must store, initialize and maintain twice as many references

22

MyLinkedList class

```
public class MyLinkedList<T>
{
    private class ListNode<T>
    {
        public T data;
        public ListNode<T> previous;
        public ListNode<T> next;
    }

    private ListNode<T> front;
    private ListNode<T> back;
    private int numNodes;

    public MyLinkedList()
    {
        front = null;
        back = null;
        numNodes = 0;
    }

    public T getFirst()
    {
        return front.data;
    }

    public T getLast()
    {
        return back.data;
    }

    . . .
```

```
    . . .

    public void addFirst(T item)
    {
        ListNode<T> newNode = new ListNode<T>();
        newNode.data = item;
        newNode.previous = null;
        newNode.next = front;

        if (front == null) {
            front = newNode;
            back = newNode;
        }
        else {
            front.previous = newNode;
            front = newNode;
        }
        numNodes++;
    }

    public T get(int index)
    {
        if (index <= size()/2) {
            ListNode<T> step = front;
            for (int i = 0; i < index; i++) {
                step = step.next;
            }
            return step.data;
        }
        else {
            ListNode<T> step = back;
            for (int i = size()-1; i > index; i--){
                step = step.previous;
            }
            return step.data;
        }
    }
}
```

23

List tradeoffs

recall that LinkedList and ArrayList implement the List interface

- in theory, could use either one whenever basic list operations were required

```
public displayAll(List<T> items)
{
  for (int i = 0; i < items.size(); i++) {
    System.out.println(items.get(i));
  }
}

--------------------------------------------------------

ArrayList<Integer> nums = new ArrayList<Integer>();
...
displayAll(nums);


LinkedList<String> words = new LinkedList<String>();
...
displayAll(words);
```

do both of these calls work?

are they equally efficient?

24

Tradeoffs (cont.)

the fact that both `LinkedList` and `ArrayList` implement the `List` interface guarantees that similar method calls will produce similar results

- it makes no guarantee about efficiency

`get(i)` is O(1) using an `ArrayList`
is O(`i`) using a `LinkedList`

```
public displayAll(List<T> items)
{
  for (int i = 0; i < items.size(); i++) {
    System.out.println(items.get(i));
  }
}
```

→ `displayAll` is O(N) when called with `ArrayList`

→ `displayAll` is O(N^2) when called with `LinkedList`

Tradeoffs (cont.)

of course, it can go the other way as well

- some `LinkedList` operations are more efficient, so a method can be more efficient when passed a `LinkedList`

```
public dupeFirst(List<T> items)
{
  items.add(0, items.get(0));
}

--------------------------------------------------------

ArrayList<Integer> nums = new ArrayList<Integer>();
...
dupeFirst(nums);


LinkedList<String> words = new LinkedList<String>();
...
dupeFirst(words);
```

Iterators

since traversing a list is a common task, Java provides a supplementary class that can be used to traverse any list efficiently

- an *iterator* is an object that can step through a list, keeps its place
  - `hasNext()` method determines whether any nodes/values left to visit
  - `next()` method returns reference to next node/value & advances

each `List` type must implement the iterator methods efficiently for that class

- `ArrayList` keeps track of an index into the list, advances via increment
- `LinkedList` keeps track of a reference into the list, advances via next field

```
public displayAll(List<T> items)
{
  iterator<T> iter = items.iterator();
  while (iter.hasNext()) {
    System.out.println(iter.next());
  }
}
```

this version will behave efficiently, O(N), regardless of whether `LinkedList` or `ArrayList`

27

FINAL EXAM: Thursday, May 5 10:00 – 11:40

similar format to previous tests

- true or false
- discussion/short answer
- explain/modify/write code

cumulative, but will emphasize material since Test 2

designed to take 60-75 minutes, will allow full 100 minutes

study hints:

- review lecture notes
- review text
- look for supplementary materials where needed (e.g., Web search)
- think big picture -- assimilate the material!
- use online review sheet as a study guide, *but not exhaustive*

28