Amortised Analysis (CLR 18)

- Running time analysis:
  - Best case
  - Average case
  - Worse case
- Amortised running time is the average time of an operation in a sequence for the worse case for that sequence.

2 Sep 99 CS410 Lecture 3 1

Queue as 2 Stacks

```
class QueueEmpty extends Exception {}
class Queue {
    Stack front=new Stack();
    Stack back=new Stack();
    void enqueue(Object item) { back.push(item); }
    Object dequeue() throws QueueEmpty {
        if (!front.isEmpty()) return front.pop;
        while(!back.empty()) front.push(back.pop());
        if (front.isEmpty()) throw new QueueEmpty();
        else return front.pop();
    }
    bool isEmpty() {
        return front.isEmpty() && back.isEmpty();
    }
}
```

2 Sep 99 CS410 Lecture 3 2

Queue Analysis

- Assume stack operations are constant
- enqueue and isEmpty are constant
- dequeue could require moving all of back to front so is $O(n)$
- Over n operations, size of queue is n in worse case, operations are worse case $O(n)$, so worse case time is $O(n^2)$

2 Sep 99 CS410 Lecture 3 3

Amortised Analysis

- An item is pushed onto back, popped off back, pushed onto front, and popped off front
- For n items that are enqueue there are $O(n)$ operations in total to get them out
- In fact, over n operations, all operations have a $\Theta(1)$ average running time even for worse case sequence of operations

2 Sep 99 CS410 Lecture 3 4

Amortised Analysis

- Amortised analysis:
  - Average running time for operation
  - Actual running time could be worse
  - Not probabilistic - running time holds even for worse inputs
- See CLR chapter 18 for more details
  - Aggregate method, Accounting method, Potential method, three good examples

2 Sep 99 CS410 Lecture 3 5

Exceptions

- Occasionally unexpected or unusual things happen
  - Pop empty stack
  - Divide by zero
  - Open nonexistent file
- Several ways to deal with this, exceptions are generally the best way

2 Sep 99 CS410 Lecture 3 6

Error Handling 1

- One way is to return special error value

```
Object dequeue() {
  if (isEmpty()) return null; ...
}
```

- Cons: client code must check for error value

```
Object item=q.dequeue();
if (item==null) // handle error
```


Error Handling 2

- Set global variable

```
Object dequeue() {
   if (isEmpty()) {
     error=true; return null;
   } ...
}
```

- Cons:
  - Client code must check global variables
  - Global variables are bad

Error Handling 3

- Ignore error
- Cons:
  - Makes code very hard to debug when error does arise
  - Bad programming style

Error Handling 4

- Immediately exit program on error
- Cons:
  - Client code cannot handle error
  - Less robust code, no graceful degradation

Exceptions

- Quick and transparent control flow transfer from error site to error handling site
- Signal an error with throw:
  - `throw <expression>;`
  - control does not reach following statement
  - but goes straight to nearest handler

Exceptions (cont)

- Handle an error with try catch

```
try <statement>
catch (Exception v) {
    <statements>
}
```

  - All exceptions that are thrown in <statement> are matched against catch clauses. First one that is matched is executed

Exception Packets

- Indicate which error with information in the exception packet
- Just an object like any other object, in the class throwable or its subclasses
- Can have fields to store information about error

2 Sep 99 CS410 Lecture 3 13

Declaring an Exception

- Declare a new exception by subclassing Excpetion:

```
class MyException extends Exception {}
```

- To catch just that exception:

```
try ... catch (MyException v) {...}
```

- To carry data put fields in class:

```
class MyException extends Exception {
   string description;
}
```

2 Sep 99 CS410 Lecture 3 14

The Need for Abstraction

- Y2K
- Programmers wanted dates
- Coded all date variables to have two decimal digits, (or 1 byte), for the year
- All code is reliant upon this choice of representation
- All code is hard to change

2 Sep 99 CS410 Lecture 3 15

Alternative

- Small piece of code that knows how dates are represented-date module
- Date module includes many date operations
- All other code uses date operations
- Change code by changing date module

2 Sep 99 CS410 Lecture 3 16

Abstract and Datastructures

- Particularly important for databases
- Most university software just needs to lookup, add, and delete students
- How students records are stored and indexed is not relevant
- For efficiency need to be able to change the actual structure used

2 Sep 99 CS410 Lecture 3 17

Abstract Datatypes

- Queue is an example of an ADT
- Type: queues
- Operations:
  - create empty, enqueue, dequeue, isEmpty
- In object-oriented style ADTs are classes, operations are public, implementation is private.

2 Sep 99 CS410 Lecture 3 18

Queue Implementation

```
class QueueEmpty extends Exception {}
class Queue {
    private Stack front=new Stack();
    private Stack back=new Stack();
    public void enqueue(Object item) { back.push(item); }
    public Object dequeue() throws QueueEmpty {
        if (!front.isEmpty()) return front.pop;
        while(!back.empty()) front.push(back.pop());
        if (front.isEmpty()) throw new QueueEmpty();
        else return front.pop();
    }
    public bool isEmpty() {
        return front.isEmpty() && back.isEmpty();
    }
}
```

2 Sep 99 CS410 Lecture 3 19

Another Queue

```
class QueueEmpty extends Exception {}
class Queue {
    private Vector items=new Vector();
    public void enqueue(Object item) {
        items.addElement(item);
    }
    public Object dequeue() throws QueueEmpty {
        if (isEmpty()) throw new QueueEmpty();
        Object i=items.firstElement();
        items.removeElementAt(0);
        return i;
    }
    public bool isEmpty() {
        return items.isEmpty();
    }
}
```

2 Sep 99 CS410 Lecture 3 20

Another Queue

```
// From Course Page
class QNode
{
    Object data;
    QNode next;
}
public class Queue
{   // Last valid when head nonnull
    private QNode head, last;
    public void put (Object item)
    {
        QNode node = new QNode();
        node.data = item;
        if (head == null) head = node;
        else last.next = node;
        last = node;
    }
    public Object get () {
        Object result = head.data;
        head = head.next;
        return result;
    }
    public boolean isEmpty () {
        return head == null;
    }
    public void clear () {
        head = null;
    }
}
```

2 Sep 99 CS410 Lecture 3 21

Stack ADT

- Type: Stack
- Operations:
  – create empty
  – is empty?
  – push
  – pop
  – top

2 Sep 99 CS410 Lecture 3 22

Stack Implementation

```
class StackEmpty extends Exception {}
class Stack {
        private Vector items=new Vector();
        public bool isEmpty() { return items.isEmpty(); }
        public void push(Object i) { items.addElement(i); }
        public Object top() {
                if (isEmpty()) throw new StackEmpty();
                return items.lastElement;
        }
        public Object pop() {
                Object i=top();
                items.removeElementAt(items.size-1);
                return i;
        }
}
```

2 Sep 99 CS410 Lecture 3 23

ADT Design

- Important to determine the needed operations
- Too few operations and cannot write client code
- Too many operations constrains implementation
- Arbitrary delete operations are often very hard, for fast insert and find

2 Sep 99 CS410 Lecture 3 24