Arrays and Linked Lists
Value vs. Reference Variables

Variables of primitive types are value variables
Variables of arrays and classes are reference variables
Reference variables are a safer form of pointers
(In C++, you can choose)

Value

i
10

int i=10;

Reference

j
new performs dynamic memory allocation

int[] j = new int[1];
j[0] = 10;

Or int[] j = {10};
Recall that all objects in Java are really references, i.e. what happens when I do

```
MyObject x = new MyObject(a,b,c);
MyObject y = x;
MyFunction(y);
```

```
x = new MyObject;
```
Arrays

Refer to several values of same type

Example:

```java
int[] myArray = new int[20];
```

Length field holds allocated number of elements

```java
myArray.length == 20
```

Indexed from 0..length-1

bounds checked dynamically

Initialized manually or in declaration

```java
int[] myArray2 = {18, 36};
```

```plaintext
myArray2 [0]  [1]
18 36
```
Predefined set of static methods operating on arrays

- equals(A, B)
- fill(A, x)
- sort(A)
- toString(A)
2D (and higher D) Arrays

May be allocated at once for rectangles

```java
int[][] i = new int[12][15];
```

Or deal with 1 dimension at a time

```java
int[][] i = new int[10][];
for (int j=0; j<10; j++)
    i[j] = new int[2*j + 1];
```

Note: multidimensional arrays appear as arrays of arrays
e.g. `i[3].length = 15` and `i.length = 12;`
Non-rectangular 2D Array
Ideally, a data type can be described only in terms of operations and how they act, not in terms of how it is actually implemented or represented. This allows us to prove properties of the data type which may be useful for testing, implementation, optimization ....
An Example: An Abstract Stack

Basic rules:
1. There is a unique constant $E$ (empty stack)
2. There are functions $\text{Push}$ and $\text{Pop}$ with rules
   1. $x.\text{Push}(x.\text{Pop}()) = x$
   2. $x.\text{Pop}(x.\text{Push}(z)) = z$
   3. $E.\text{Pop}() = \text{error}$
List ADT

What are the operations on a list of objects:

- print
- clear
- insert (where?)
- remove (what?)
- find
- access the kth element
A Warmup: Array Implementation

How long do these take on an array-based implementation?

- print
- clear
- insert
  - front
  - rear
  - at element
- remove at index
- find index of element
- access the kth element
The Idea of Linked List

- Singly linked lists
- Doubly linked lists
- Circular lists
public class Node{
    private Object element;
    private Node next;
    public Node(Object e, Node n) {
        element = e; next = n; }
    public Object getElement() { return element;}
    public Object setElement(nE) {element = nE;}
    public Node getNext() { return next;}
    public Node setNext(Node nN) {next = nN;}
}
SL List Implementation

How long do these take on an singly-linked list-based implementation?

• print
• clear
• insert
  • front
  • rear
  • at element
• remove at index
• find index of element
• access the kth element
Doubly Linked Lists

The limitation of the singly linked lists stems from the fact we cannot look back

Doubly linked lists overcome this problem

With doubly linked lists, it is inconvenient to have an asymmetry at the beginning and end of the list.

Sentinels are “dummy” nodes that do not hold data, but regularize the computation
public class DNode<T> {
    private T element;
    private DNode<T> next, prev;
    public DNode(T e, DNode<T> n, DNode<T> p) {
        element = e;
        next = n;
        prev = p
    }
}
Doubly Linked List Implementation

```java
public class MyList<T> {
    protected int numElts=0;
    protected Dnode<T> header, trailer;

    MyList () { header = new Dnode<T>(null,null,null);
        trailer = new Dnode<T>(null, header, null);
        header.setNext(trailer); }

    public Dnode<T> getPrev(Dnode<T> v) throws IllegalArgumentException {
        if (v == header) throw new IllegalArgumentException("At header");
        return v.getPrev();
    }

    public Dnode<T> addBefore(Dnode<T> p, Dnode<T> newNode) throws .... {
        numElts++;
        newNode.setPrev(p.getPrev());
        p.getPrev().setNext(newNode);
        p.setPrev(newNode);
        newNode.setNext(p);
        return newNode;
    }
}
```
Exceptions

Language-level support for managing run-time errors
You can define your own exception classes
Methods declare which exceptions they might possibly throw
Calling methods either catch these exceptions or pass them up the call stack
public void myMethod() throws BadThingHappened {
    if (someCondition)
        throw new BadThingHappened;
    
    ...}

try
    ...
    myMethod()
    ...
    catch (BadThingHappened BTH)
        block
    catch (exceptiontype id)
        block
    finally
        block // ALWAYS executed!!
public void remove(DNode v) {
    DNode u = getPrev(v);
    DNode w = getNext(v);
    w.setPrev(u);
    u.setNext(w);
    v.setPrev(null);
    v.setPrev(null);
    size--;
}
How long do these take on an doubly-linked list-based implementation?

- print
- clear
- insert
  - front
  - rear
  - at element
- remove at index
- find index of element
- access the kth element
Java Collections Framework

**Collection:** Any collection of objects; extends iterable and thus permits iteration with iterator() method

**Iterator:** Basic iterator ADT

**List:** Extends collection to include ArrayList and LinkedList; supports ListIterator (modification)

**ListIterator:** Forward/backward and modification

**Stack:** Add and remove from front

**Queue:** Add to end, take from front

*All in java.util*
Iterators

Once we have a linear data structure, a natural model of computation is *iteration*

An iterator abstracts this idea from the underlying data storage format

```java
public interface Iterator<E> {
    public boolean hasNext();
    public E next();
    void remove( );  // last item return by next
}
```
Each class using an iterator can support `Iterable<E>`

```java
public interface Iterable<E> {
    // return an iterator object
    public java.util.Iterator<E> iterator();
}
```
In Java.util

```java
public interface Iterator<AnyType> {
    boolean hasNext();
    AnyType next();
    void remove();  // last item returned by next
}

public interface Collection<AnyType> extends Iterable<AnyType> {
    int size();
    boolean isEmpty();
    void clear();
    boolean contains(AnyType x);
    boolean add(AnyType x);
    boolean remove(AnyType x);
    java.util.Iterator<AnyType> iterator();
}
```
Why Iterable?

Allows the use of special “:” syntax

```java
public static <T> void print( Collection<T> c )
{
    for (T item : c)
        System.out.println( item );
}

// VS

public static <T> void print( Collection<T> c )
{
    Iterator<T> itr = c.iterator( );
    while (itr.hasNext( )) {
        T item = itr.next( );
        System.out.println( item );
    }
}
```
The Dangers of Iterators

Iterators make sense if data structures are static while they are in use.

What if the DS changes?

Could make copy (O(n))

Positions continue to make sense (although difficult to guarantee behavior).

In Java, the list iterator in Java does not expose a “position” concept --- iterators are invalidated if container is changed.
What is the problem here?

```
for (Integer x : lst )
    if (x % 2 == 0)
        lst.remove(x)

FAILS!

Iterator itr = lst.iterator();

while (itr.hasNext())
    if (itr.next() % 2 == 0)
        itr.remove();
```
The List Interface

public interface List<T> extends Collection<T>
{
    T get (int i);
    T set (int i);
    void add (int i, T x);
    void remove (int i);
    ListIterator<AnyType> listIterator(int pos);
}

public interface ListIterator<T> extends Iterator<T>
{
    boolean hasPrevious( );
    T previous( );
    void add (T x);
    void set (T x);
}

Implemented by **ArrayList** and **LinkedList**
Example of ListIterator

I start with:
8 3 5 6 4
and execute

System.out.println( iter.next() );
iter.add( 1 ) ;
System.out.println( iter.next( ) );
iter.remove( ) ;
System.out.println( iter.next( ) );
iter.set( 7 ) ;
System.out.println( iter.next( ) );
x.set(4,20);

What is the result?
How long do each of these take with linked list or array?

```java
for (int i=0; i< N; i++)
    lst.add( i, i );

for (int i=0; i< N; i++)
    lst.add(0, i);

for (int i=0; i< lst.size(); i++)
    total += lst.get(i);
```
```java
int i = 0;
while (i < lst.size)
    if lst.get(i) % 2 == 0
        lst.remove(i);
    else
        i++;

for (Integer x : lst)
    if (x%2 == 0)
        lst.remove(x);

Iterator<Integer> itr = lst.iterator();
while (itr.hasNext())
    if (itr.next() % 2 == 0)
        itr.remove();
```
ArrayList class

Things to note:
- outer vs. inner vs. nested class for iterator
- inner good if 1-1 association
- nested allows non-associated
- exception handling
- automatic sizing
Note:

- The use of sentinel nodes
- Visibility of nested class members
- Use of modcount
Amortization

Useful analysis tool
When some calls are more expensive than others, average out the costs over the total number of calls.

After every n calls to add, 1 call takes $O(n)$ instead of $O(1)$.

Averaged out over n calls, each call is still $O(1)$.
Formal Amortization Analysis

Assume each `add()` costs $1 in compute time

Overcharge these `add()` operations
charge $3 each
store $2 in the bank for each operation

Now when the extend happens, use money from the bank to pay for the copy operations

We pay for all $n$ operations using a constant cost for each operation

implies $O(n)$ total cost, or average of $O(1)$ per operation
Work done in “extend” for $n$ pushes:

$$1 + 2 + 4 + 8 + \ldots + n = \sum_{i=0}^{\log_2 n} 2^i$$

$$= 2^{\log n + 1} - 1 = 2 \times 2^{\log n} - 1$$

$$= 2n - 1 = O(n)$$ for $n$ pushes
On to Stacks and Queues...
What is a Stack?

Stores a set of elements in a particular order
Accessed in Last-In-First-Out (LIFO) fashion
Real life examples:
  Pile of books
  PEZ dispenser
  Cup trays in cafeteria
CS examples: program execution stack, parsing/evaluating expressions
Stack Abstract Data Type

push(o): insert o on top of stack
pop( ): remove object from top of stack
top( ): look at object on top of stack (but don’t remove)
size( ): number of objects on the stack
isEmpty( ): does (size == 0)?
An Example: Call Stack Frames

Stack Pointer

Frame Pointer

Locals of DrawLine

Return Address

Parameters for DrawLine

Locals of DrawSquare

Return Address

Parameters for DrawSquare

::
The new HP 3000 minicomputer system from Hewlett-Packard Company ranges from the high-end Series 3000/3000 to the low-end Series 2000, which offers the advantages of RISC technology.
public class StackEmptyException
    extends RuntimeException {
    public EmptyStackException(String err) {
        super(err);
    }
}
Exceptions

Language-level support for managing run-time errors

You can define your own exception classes

Methods declare which exceptions they might possibly throw

Calling methods either catch these exceptions or pass them up the call stack
public void myMethod() throws BadThingHappened {
    ...
    if (someCondition)
        throw new BadThingHappened;
    ...
}

try
    ...
    myMethod()
    ...
catch (BadThingHappened BTH)
    block
catch (exceptiontype id)
    block
finally
    block // ALWAYS executed!!
public interface Stack<T> {
    public int size();
    public boolean isEmpty();
    public T top() throws StackEmptyException;
    public void push (T element);
    public T pop() throws StackEmptyException;
}

Johns Hopkins Department of Computer Science
Course 600.226: Data Structures, Professor: Greg Hager
Benefits
Avoids maximum stack size restriction
Only allocates memory for stack elements actually used

How
Allocate a node for each stack element
Nodes are chained together by reference to next node in the chain
public class Node<E> {
    private E element;
    private Node next;
    public Node(E e, Node n) {
        element = e; next = n; }
    public E getElement() { return element;}
    public Node<E> getNext() { return next;}
    public void setElement(E newE) {element = newE;}
    public void setNext(Node<E> newN) {next = newN;}
}
public class LinkedStack<E> implements Stack<E> {
    private Node<E> top = null;
    private int size = 0;

    public void push(E elem) {
        Node<E> v = new Node<E>(elem, top);
        top = v;
        size++;    }

    public E pop() throws StackEmptyException {
        if (isEmpty()) throw new StackEmptyException("empty");
        E temp = top.getElement();
        top = top.getNext();
        size --;
        return temp;
    }
}
Array-based Stack Implementation

Allocate array of some size
Maximum # of elements in stack
Bottom stack element stored at index 0
first index tells which element is the top
increment first when element pushed, decrement when pop’ d
public class ArrayStack<T> implements Stack<T> {
    private T[] S;
    private int topIndex = -1;

    public ArrayStack(int cap) {
        capacity cap;
        S = (T[]) new Object[capacity];
    }
}
public void push(T obj) throws StackFull {
    if (size() == S.length)
        throw new StackFullException("stack full");
    S[++topIndex] = obj; }

public T pop() throws StackEmpty {
    if (isEmpty())
        throw new StackEmptyException("empty");
    T elem = S[topIndex];
    S[topIndex--] = null;
    return elem; }
Analysis

Each operation is $O(1)$ running time
Independent of number of items in stack
push, pop, top, size, isEmpty

Space can be $O(n)$ or may be much more
depends if $n$ is known at initialization time
Analysis

All stack functions still $O(1)$
push, pop, top, size, isEmpty
Examples

Reversing an Array
Paren matching
HTML tags
postfix arithmetic
Converting to postfix
What is a Queue

Stores a set of elements in a particular order
Accessed in First-In-First-Out (FIFO) fashion
Real life examples:
  Waiting in line at cafeteria
  Waiting on hold for technical support
CS Example: Buffered I/O
Queue ADT

enqueue(o): Insert object o at rear of queue
dequeue( ): remove object from front of queue
size( ): number of elements
isEmpty( ): size == 0?
front( ): look at object at front of queue
public interface Queue<E> {
    public int size();
    public boolean isEmpty();
    public E front() throws QueueEmptyException;
    public void enqueue (E element);
    public E dequeue() throws QueueEmptyException;
}
Array-based Queue Implementation

Array of fixed size
Index array element for front and rear of queue
Indices “wrap around” when they cross end of array
Array Queue Implementation

```java
public class ArrayQueue<E> implements Queue<E> {
    private E[] Q;
    private int size=0;
    private int front=0, rear = 0;
    public void enqueue(E o) {
        if (size() == Q.length) throw new QueueFullException("full");
        Q[rear] = o;
        rear = (rear + 1) % Q.length; size++;
    }
    public E dequeue() {
        if (size() == 0) throw new QueueEmptyException("hungry");
        E tmp = Q[front];
        front = (front + 1) % Q.length; size--;
        return tmp;
    }
}
```
List Queue Implementation

Head and tail node references for front and rear of queue
Insert at tail, remove from head
Remove from tail too slow for singly linked list
  Updating tail reference with new tail takes full traversal
So use tail of list for rear of queue
public class ListQueue<E> implements Queue<E> {
    private Node<E> head = null;
    private Node<E> tail = null;
    private int size = 0;
    . . .
    public void enqueue(E elem)
        Node<E> node = new Node<E>(E,null);
        if (empty()) head=node;
        else tail.setNext(node);
        tail = node; size++;
    }

    public E dequeue() throws QueueEmptyException {
        if (empty()) throw QueueEmptyException("grr");
        E tmp = head.getElement();
        head = head.getNext(); size--;
        if (empty()) tail = null;
        return tmp;
    }
}
All queue operations are $O(1)$
size( ), isEmpty( )
enqueue( ), dequeue( ), front( )
Examples

Round-robin scheduling
Buffered I/O
Email
Printing

...
Example: The JVM

JVM divides memory into two parts: stack and heap
Stack is used for procedure calls and static allocation
Heap is used for dynamic allocation
In addition, the JVM maintains a queue of active threads
Double-ended Queue

Sometimes called “deque” (děk)
Similar to stack and queue
  Allows insertion and removal at both ends of the queue
Stack or queue is easily implemented using a deque
Deque ADT

insertFirst(e) : insert element at front
insertLast(e) : insert element at rear
removeFirst( ) : remove first element
removeLast( ) : remove element at rear
first( ) : examine first element
last( ) : examine last element
size(e) : number of elements in deque
isEmpty( ) : size == 0?
Summary

1. Basic data structures: arrays, singly linked lists, doubly linked lists
2. Iterators, Iterable, Collections, Lists
3. Stacks, Queues and their uses
4. Time-analysis of implementations