3 Advanced Programming Techniques

Topics

- Recursion
- Abstract Data Types
  - Stacks
  - Queues
  - Sequential and Linked Representation
  - Java Collections

What is Recursion?

- Recursion
  - Can be applied when the nature of the problem is repetitive
  - Less efficient than iteration but more elegant
  - Methods are allowed to call upon itself
  - Data from arguments are stored temporarily onto a stack until method calls have been completed

Recursion Vs. Iteration

- Iteration
  - Use repetition control structures
  - Process ends when the loop condition fails
  - Faster

- Recursion
  - Calling a method repetitively
  - Define the problem in terms of smaller instances of itself
  - Process ends once a particular condition called the base case is satisfied
    - Base case is simply the smallest instance of the problem
  - Encourage good programming practice

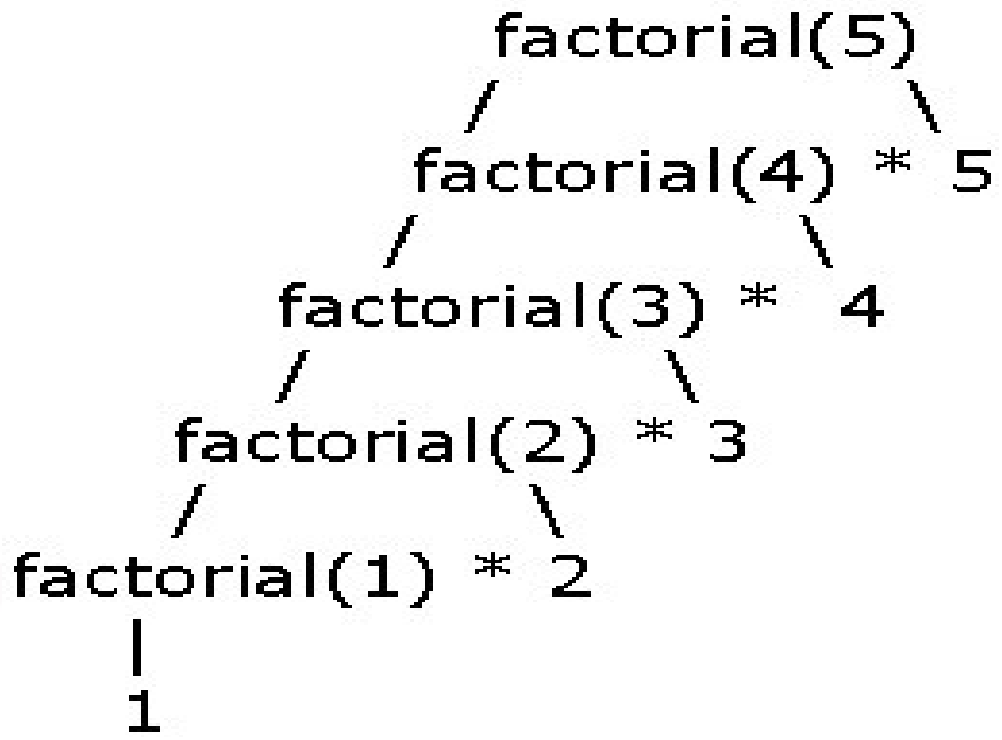
Recursion Vs. Iteration

- Both can lead to infinite loops
- Recursion or Iteration?

Factorials: An Example

- Recursive definition:
  - factorial(n) = factorial(n-1) * n, n is a positive integer
  - factorial(1) = 1
- Example:
  - factorial(2) = factorial(1)*2 = 2
  - factorial(3) = factorial(2)*3 = 2*3 = 6.

Factorials: An Example



Factorials: An Example

- Iterative code:

```
1  class FactorialIter {
2      static int factorial(int n) {
3          int result = 1;
4          for (int i = n; i > 1; i--) {
5              result *= i;
6          }
7          return result;
8      }
9      public static void main(String args[]) {
10         int n = Integer.parseInt(args[0]);
11         System.out.println(factorial(n));
12     }
13 }
```

Factorials: An Example

- Recursive code:

```
1  class FactorialRecur {
2      static int factorial(int n) {
3          if (n == 1) {     /* The base case */
4              return 1;
5          }
6          /* Recursive definition; Self-invocation */
7          return factorial(n-1)*n;
8      }
9      public static void main(String args[]) {
10         int n = Integer.parseInt(args[0]);
11         System.out.println(factorial(n));
12     }
13 }
```

Print n in any Base: Another Example

- Recall:
  - Use repetitive division and to write the remainders in reverse
  - The process terminates when the dividend is less than the base

- $10_{10} = \underline{12}_{8}$

8	10	2
8	1	1
	0	

- $165_{10} = \underline{A5}_{16}$

16	165	5
16	10	10
	0	

Print n in any Base: Another Example

- Iterative code:

```
1 class DecToOthers {
2     public static void main(String args[]) {
3         int num = Integer.parseInt(args[0]);
4         int base = Integer.parseInt(args[1]);
5         printBase(num, base);
6     }
7 //continued...
```

Print n in any Base: Another Example

```
8  static void printBase(int num, int base) {
9          int rem = 1;
10         String digits = "0123456789abcdef";
11         String result = "";
12         while (num!=0) {
13             rem = num%base;
14             num = num/base;
15             result = result.concat(digits.charAt(rem)
16                                    +"");
17         }
18         for(int i = result.length()-1; i >= 0; i--) {
19             System.out.print(result.charAt(i));
20         }
21     }
22 }
```

Print n in any Base: Another Example

- Recursive code:

```
1  class DecToOthersRecur {
2     static void printBase(int num, int base) {
3        String digits = "0123456789abcdef";
4        /* Recursive step*/
5        if (num >= base) {
6           printBase(num/base, base);
7        }
8        /* Base case: num < base */
9        System.out.print(digits.charAt(num%base));
10    }
11 //continued...
```

Print n in any Base: Another Example

```
12      public static void main(String args[]) {
13          int num = Integer.parseInt(args[0]);
14          int base = Integer.parseInt(args[1]);
15          printBase(num, base);
16      }
17  }
```

Abstract Data Types (ADT)

- Collection of data elements provided with a set of operations that are defined on the data elements

- Examples:
  - Stacks
  - Queues
  - Binary trees

ADT: Stacks

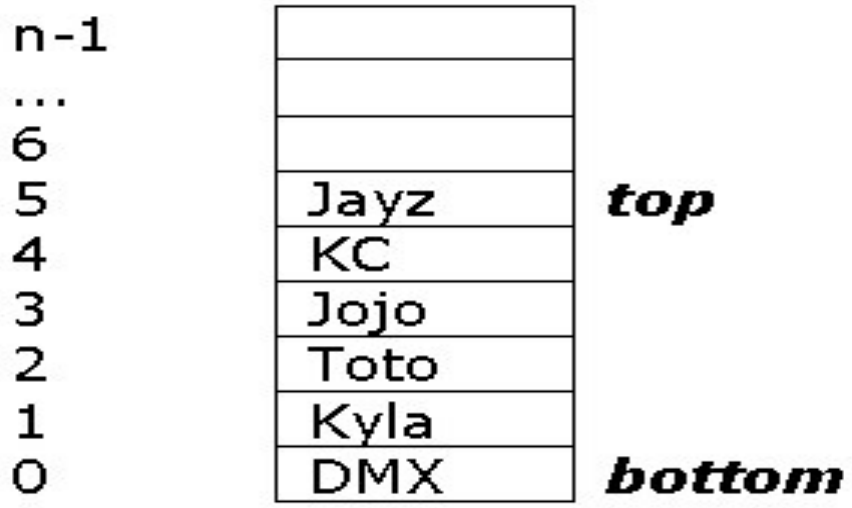
- Definition:
  - Linearly ordered collection of data on which the discipline of “last in, first out” (LIFO) is imposed
  - Manipulation of elements is allowed only at the top of the stack
- Applications
  - Pattern recognition
  - Conversion among infix, postfix and prefix notations

ADT: Stacks

- Two operations:
  - Push
    - Insert element at the top of the stack
  - Pop
    - Remove the element at the top of the stack.
- Analogy
  - Stack of Plates
  - Spindle case

ADT: Stacks

- Illustration:



- Condition for a full stack:
  - top == n-1.

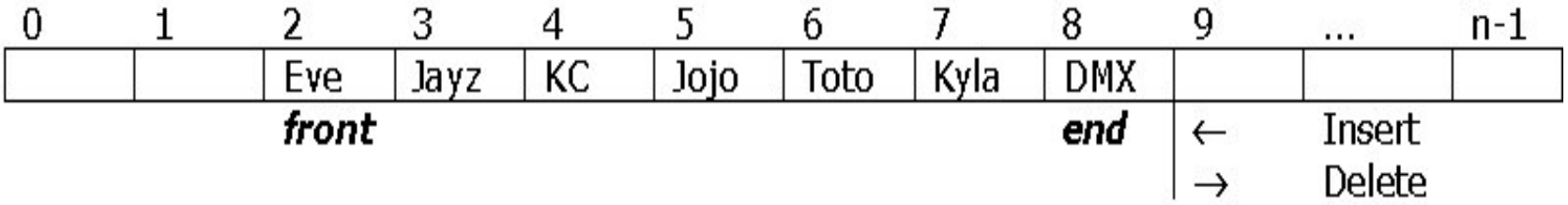
- Condition for an empty stack:
  - top == -1

ADT: Queues

- Definition:
  - Enqueue
    - Insert at the end of the queue
  - Dequeue
    - Removing front element of the queue
- Analogy:
  - Queue or a line

ADT: Queues

- Illustration:



- Condition for a full queue:
  - end == n-1

- Condition for an empty queue:
  - end < front

Sequential and Linked Representation

- Sequential representation
  - Easier to implement with the use of arrays
  - Inflexible size
    - Results in memory waste or not enough memory

- Linked representation
  - A little more difficult to implement
  - Flexible

Sequential Representation of an Integer Stack

```
1  class SeqStack {
2      int top = -1;        /* stack is empty */
3      int memSpace[];      /* storage for integers */
4      int limit;           /* size of memSpace */
5      SeqStack() {
6          memSpace = new int[10];
7          limit = 10;
8      }
9      SeqStack(int size) {
10         memSpace = new int[size];
11         limit = size;
12     }
13 //continued...
```

Sequential Representation of an Integer Stack

```
14      boolean push(int value) {
15          top++;
16          /* check if the stack is full */
17          if (top < limit) {
18              memSpace[top] = value;
19          } else {
20              top--;
21              return false;
22          }
23          return true;
24      }
25  //continued...
```

Sequential Representation of an Integer Stack

```
26      int pop() {
27          int temp = -1;
28          /* check if the stack is empty */
29          if (top >= 0) {
30              temp = memSpace[top];
31              top--;
32          } else {
33              return -1;
34          }
35          return temp;
36      }
37  //continued...
```

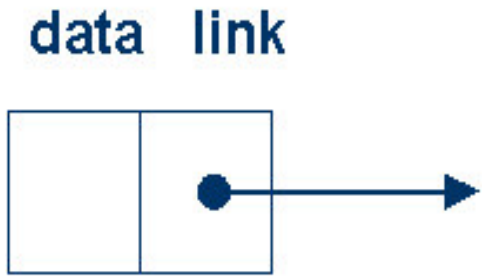
Sequential Representation of an Integer Stack

```
38      public static void main(String args[]) {
39          SeqStack myStack = new SeqStack(3);
40          myStack.push(1);
41          myStack.push(2);
42          myStack.push(3);
43          myStack.push(4);
44          System.out.println(myStack.pop());
45          System.out.println(myStack.pop());
46          System.out.println(myStack.pop());
47          System.out.println(myStack.pop());
48      }
49  }
```

Linked Lists

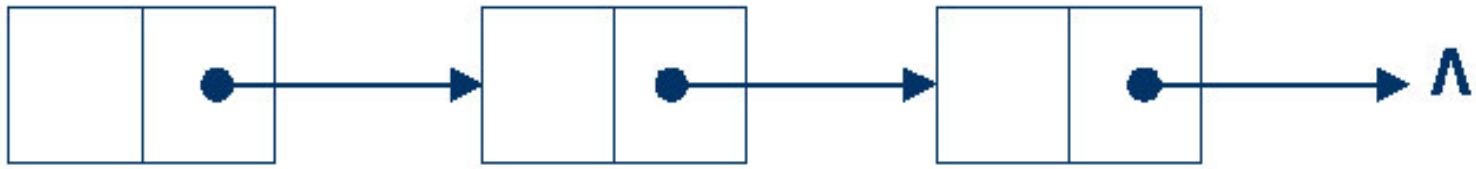
- Definition:
  - Dynamic structure
    - Can grow and shrink in size
  - Collection of nodes
    - A node consists of some data and a link or a pointer to the next node in the list

- Node:



Linked Lists

- Non-empty linked list with three nodes:



- Node class implementation:

```
1 class Node {
2     int data;           /* integer data in node */
3     Node nextNode;      /* next node in the list */
4 }
```

Linked Lists

- Testing the Node class:

```
1  class TestNode {
2     public static void main(String args[]) {
3        Node emptyList = null;     /* empty list */
4        /* head points to 1st node in the list */
5        Node head = new Node();
6        /* initialize 1st node in the list */
7        head.data = 5;
8        head.nextNode = new Node();
9        head.nextNode.data = 10;
10 //continued...
```

Linked Lists

```
11          /* null marks the end of the list */
12          head.nextNode.nextNode = null;
13          /* print elements of the list */
14          Node currNode = head;
15          while (currNode != null) {
16              System.out.println(currNode.data);
17              currNode = currNode.nextNode;
18          }
19      }
20  }
```

Linked Representation of an Integer Stack

```
1  class DynamicIntStack{
2     private IntStackNode top;      //head of the stack
3     class IntStackNode {           //Node class
4        int data;
5        IntStackNode next;
6        IntStackNode(int n) {
7           data = n;
8           next = null;
9        }
10    }
11 //continued...
```

Linked Representation of an Integer Stack

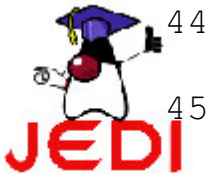
```
12    void push(int n){
13        /* no need to check for overflow */
14        IntStackNode node = new IntStackNode(n);
15        node.next = top;
16        top = node;
17    }
18 //continued...
```

Linked Representation of an Integer Stack

```
19    int pop() {
20       if (isEmpty()) {
21          return -1;
22          /* may throw a user-defined exception */
23       } else {
24          int n = top.data;
25          top = top.next;
26          return n;
27       }
28    }
29    boolean isEmpty(){
30       return top == null;
31    }
```

Linked Representation of an Integer Stack

```
32    public static void main(String args[]) {
33       DynamicIntStack stack = new DynamicIntStack();
34       stack.push(5);
35       stack.push(10);
36       /* print elements of the stack */
37       IntStackNode currNode = stack.top;
38       while (currNode!=null) {
39          System.out.println(currNode.data);
40          currNode = currNode.next;
41       }
42       System.out.println(stack.pop());
43       System.out.println(stack.pop());
44    }
45 }
```



Java Collections

- Java Collections
  - Java built-in collection classes and interfaces
    - Found in the *java.util* package
  - Examples of collection classes:
    - Stack
    - LinkedList
    - ArrayList
    - HashSet
    - TreeSet

Java Collections

- *Collection* interface
  - Root of all collection interfaces
  - No built-in implementations
  - Example:
    - Queue (J2SE 5.0)

- Definition of Collection:
  - Group of objects, which are also called elements
  - May allow duplicates and requires no specific ordering

Java Collections

- Built-in subinterfaces of *Collection* interface
  - *Set* Interface
    - Unordered collection that contains no duplicates
    - Some built-in implementing classes: HashSet, LinkedHashSet and TreeSet
  - *List* Interface
    - Ordered collection of elements where duplicates are permitted
    - Some built-in implementing classes: ArrayList, LinkedList and Vector

Java Collections

- Java Collections Hierarchy

<table>
<tr><td colspan="6"><root interface>
Collection</td></tr>
<tr><td colspan="3"><interface>
Set</td><td colspan="3"><interface>
List</td></tr>
<tr><td colspan="3"><implementing classes></td><td colspan="3"><implementing classes></td></tr>
<tr><td>HashSet</td><td>LinkedHashSet</td><td>TreeSet</td><td>ArrayList</td><td>LinkedList</td><td>Vector</td></tr>
</table>

Java *Collection* Methods:
Java 2 Platform SE v1.4.1

Collection Methods
`public boolean add(Object o)`
Inserts the *Object o* to this collection. Returns *true* if *o* was successfully added to the collection.
`public void clear()`
Removes all elements of this collection.
`public boolean remove(Object o)`
Removes a single instance of the *Object o* from this collection, if it is present. Returns *true* if *o* was found and removed from the collection.
`public boolean contains(Object o)`
Returns true if this collection contains the *Object o*.
`public boolean isEmpty()`
Returns true if this collection does not contain any object or element.

Java *Collection* Methods:
Java 2 Platform SE v1.4.1

Collection Methods
`public int size()`
Returns the number of elements in this collection.
`public Iterator iterator()`
Returns an iterator that allows us to go through the contents of this collection.
`public boolean equals(Object o)`
Returns true if the *Object o* is equal to this collection.
`public int hashCode()`
Returns the hash code value (i.e., the ID) for this collection. Same objects or collections have the same hash code value or ID.
Returns true if this collection contains the *Object o*.

Java Collections: *LinkedList*

```
1  import java.util.*;
2  class LinkedListDemo {
3     public static void main(String args[]) {
4        LinkedList list = new LinkedList();
5        list.add(new Integer(1));
6        list.add(new Integer(2));
7        list.add(new Integer(3));
8        list.add(new Integer(1));
9        System.out.println(list+", size = "+list.size());
10       list.addFirst(new Integer(0));
11       list.addLast(new Integer(4));
12       System.out.println(list);
13       System.out.println(list.getFirst() + ", " +
14                                          list.getLast());
```

Java Collections: *LinkedList*

```
15 //continuation...
16         System.out.println(list.get(2)+", "+list.get(3));
17         list.removeFirst();
18         list.removeLast();
19         System.out.println(list);
20         list.remove(new Integer(1));
21         System.out.println(list);
22         list.remove(3);
23         System.out.println(list);
24         list.set(2, "one");
25         System.out.println(list);
26     }
27 }
```

Java Collections: *ArrayList*

- Definition:
  - Resizable version an ordinary array
  - Implements the *List* interface

- Example:

```
1 import java.util.*;
2 class ArrayListDemo {
3    public static void main(String args[]) {
4       ArrayList al = new ArrayList(2);
5       System.out.println(al+", size = "+al.size());
6       al.add("R");
7 //continued...
```

Java Collections: *ArrayList*

```
8          al.add("U");
9          al.add("O");
10         System.out.println(al+", size = "+al.size());
11         al.remove("U");
12         System.out.println(al+", size = "+al.size());
13         ListIterator li = al.listIterator();
14         while (li.hasNext())
15            System.out.println(li.next());
16         Object a[] = al.toArray();
17         for (int i=0; i<a.length; i++)
18            System.out.println(a[i]);
19      }
20   }
```

Java Collections: *HashSet*

- Definition:
  - Implementation of the *Set* interface that uses a hash table
  - Hash table
    - Uses a formula to determine where an object is stored.
  - Benefits of using a hash table
    - Allows easier and faster look up of elements

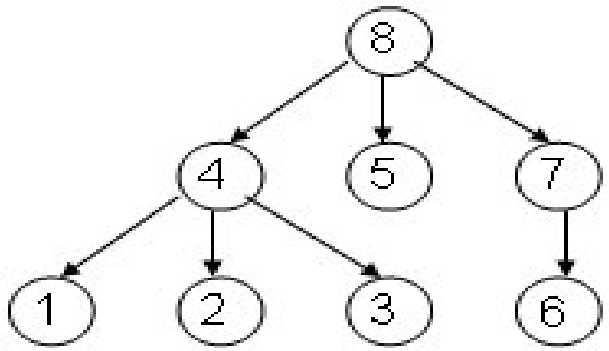
Java Collections: *HashSet*

```
1  import java.util.*;
2  class HashSetDemo {
3     public static void main(String args[]) {
4        HashSet hs = new HashSet(5, 0.5f);
5        System.out.println(hs.add("one"));
6        System.out.println(hs.add("two"));
7        System.out.println(hs.add("one"));
8        System.out.println(hs.add("three"));
9        System.out.println(hs.add("four"));
10       System.out.println(hs.add("five"));
11       System.out.println(hs);
12    }
13 }
```

Java Collections: *TreeSet*

- Definition:
  - Implementation of the *Set* interface that uses a tree
  - Tree
  - Ensures that the sorted set will be arranged in ascending order

- Tree representation



Java Collections: *TreeSet*

```
1  import java.util.*;
2  class TreeSetDemo {
3     public static void main(String args[]) {
4        TreeSet ts = new TreeSet();
5        ts.add("one");
6        ts.add("two");
7        ts.add("three");
8        ts.add("four");
9        System.out.println(ts);
10    }
11 }
```

Summary

- Recursion
  - Definition
  - Recursion Vs. Iteration
- Abstract Data Types
  - Definition
  - Stacks
  - Queues
  - Sequential and Linked Representation

Summary

- Java Collections
  - Collection
  - Linked List
  - ArrayList
  - HashSet
  - TreeSet