Fast Calculation of Factorials from 1 to 100 in Java

Factorial calculation represents one of the typical problems in Computer Science (CS) that is used to explain recursion. However, you might find this type of problems during interview sessions or even in programming challenges. With that said, in this post I will try to cover the traditional approach, as well as the optimized approach for calculating factorials.

The typical recursive implementation for factorial calculation as explained in CS textbooks is:

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

However, it is easy to see that this solution wouldn't suffice for calculating factorials ranging up to 100, as they can have values up to 9.3326215444×10^157. In order to overcome this restriction, we need to use Java's BigInteger class. This class will enable us to do basic arithmetic operations with very large numbers, which will clearly suffice for the needs of factorial calculation. With that said, we can rewrite our initial recursive implementation to use the BigInteger class as so:

```
BigInteger factorial(int n) {
    if (n == 0) {
        return BigInteger.ONE;
    } else {
        return new BigInteger("" + n).multiply(factorial(n - 1));
    }
}
```

It's clear, that you can use this BigInteger-based recursive approach for calculating factorials; however, if you plan to calculate factorials from 1 to 100 then using this approach would be costly; especially if we implement it in the following manner:

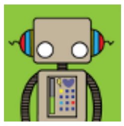
```
for (int i = 0; i < 100; i++) {
    System.out.println(factorial(i));
}
```

The faster approach of calculating the factorials from 1 to 100 would be to use the approach described in **SmallFactorials.java**, i.e.:

```
 import java.util.Scanner;
 import java.math.BigInteger;

 public class SmallFactorials {

     public static final int MAX_SIZE = 100;
     public static BigInteger[] factorials = new BigInteger[MAX_SIZE + 1];
     static {
         factorials[0] = BigInteger.ONE;
         init();
     }

     public static void init() {
         for (int i = 1; i <= MAX_SIZE; i++) {
             factorials[i] = factorials[i - 1].multiply(new BigInteger("" + i));
         }
     }

     public static void main(String[] args) {

         Scanner sc = new Scanner(System.in);
         int lineCount = sc.nextInt();

         for (int i = 0; i < lineCount; i++) {
             System.out.println(factorials[sc.nextInt()]);
         }
     }
 }
```

Yes, there are several things going on in this class, so please bear with me as I try to explain them First of all, the calculation of factorials is not done in a recursive manner. Since we want to calculate factorials from 1 to 100 we can use a traditional for loop, and just multiply the current index (i.e. *n*) with the previous result (i.e. *factorial(n - 1)*). This approach would be more efficient than by calling the recursive *factorial()* method every time. To further

About



Hello and welcome to the personal blog of Genc Doko. I'm a Research Programmer working on Intelligent Tutoring Systems at the Human-Computer Interaction Institute of Carnegie Mellon University in Pittsburgh, Pennsylvania. Continue reading.

Recent Posts

- Stack Implementation Using a Doubly-Linked List
- Another Doubly-Linked List Implementation in Java
- Finding the Middle Node of a Singly-Linked List
- Eeny, Meeny, Miny, Moe Selection with Circular Doubly-Linked Lists
- Circular Doubly-Linked List Implementation in Java

Tags

AJAX Algorithms Amazon.com Amazon Kindle Applets Audio Books Binary Search Tree Biography Books C Data Structures E-books Elgg File I/O Hacks Interview Questions Java Java Collections Framework JavaScript JDBC Linked Lists Mathematics Merge Sort MySQL Notetaking Object Oriented Design Optimizations Oracle Database PHP Physics Post-Apocalyptic Programming Challenges Science Fiction Software Engineering Sort Algorithms SPAM SPOJ SQL Stacks Stored Procedures String Algorithms Threads Ubuntu XML YouTube

Archives

- August 2011 (10)
- July 2011 (5)
- May 2011 (1)
- January 2011 (3)
- December 2010 (7)
- October 2010 (1)
- September 2010 (17)
- August 2010 (17)

This approach would be more efficient than by calling the recursive *factorial()* method every time. To further optimize our solution, we can make sure that all the factorials are stored in a static (i.e. class level) BigInteger array whose populated through static instantiation.

The final caveat of this approach is the integration of the [Scanner]() class. The instance of the Scanner class will enable us to take the input directly from the command line (i.e. stdin). Thus, in the command line we can specify the number of requests, and then proceed to make requests for pre-calculated factorials ranging from 1 to 100. E.g. we indicate on the first line that we will make 5-requests, and then proceed with requesting results for factorials 1, 3, 5, 10, 30:

```
gdoko@mars:~/Java$ java SmallFactorials
5
1
1
3
6
5
120
10
3628800
30
265252859812191058636308480000000
```

The speed of this approach can be tested by using the **SmallFactorialsInput.txt** (see below) which contains 100-requests for factorials ranging from 1 to 100. On my laptop (i.e. Lenovo T400, Intel Core 2 Duo P8600 with 4GB RAM), the runtime for this test is as follows:

```
gdoko@mars:~/Java$ time java SmallFactorials < SmallFactorialsInput.txt > SmallFactorialsOutput.txt

real    0m0.350s
user    0m0.320s
sys     0m0.040s
```

For your convenience, here are the input and output files that I've used for testing the SmallFactorials class:

SmallFactorialsInput.txt:

```
100
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
```

58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

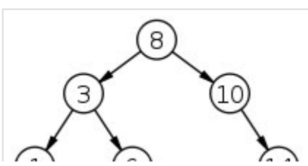
SmallFactorialsOutput.txt:

1
2
6
24
120
720
5040
40320
362880
3628800
39916800
479001600
6227020800
87178291200
1307674368000
20922789888000
355687428096000
6402373705728000
121645100408832000
2432902008176640000
51090942171709440000
1124000727777607680000
25852016738884976640000
620448401733239439360000
15511210043330985984000000
403291461126605635584000000
10888869450418352160768000000
304888344611713860501504000000
8841761993739701954543616000000
265252859812191058636308480000000
8222838654177922817725562880000000
263130836933693530167218012160000000
8683317618811886495518194401280000000
295232799039604140847618609643520000000
10333147966386144929666651337523200000000
371993326789901217467999448150835200000000
13763753091226345046315979581580902400000000
523022617466601111760007224100074291200000000
20397882081197443358640281739902897356800000000
815915283247897734345611269596115894272000000000
33452526613163807108170062053440751665152000000000
1405006117752879898543142606244511569936384000000000
60415263063373835637355132068513997507264512000000000
2658271574788448768043625811014615890319638528000000000
119622220865480194561963161495657715064383733760000000000
5502622159812088949850305428800254892961651752960000000000
258623241511168180642964355153611979969197632389120000000000
12413915592536072670862289047373375038521486354677760000000000
608281864034267560872252163321295376887552831379210240000000000
30414093201713378043612608166064768844377641568960512000000000000
1551118753287382280224243016469303211063259720016986112000000000000

```
806581751709438785716606368564037669752895054408832778240000000000000
4274883284060025564298013753389399649690343788366813724672000000000000
230843697339241380472092742683027581083278564571807941132288000000000000
12696403353658275925965100847566516959580321051449436762275840000000000000
710998587804863451854045647463724949736497978881168458687447040000000000000
40526919504877216755680601905432322134980384796226602145184481280000000000000
2350561331282878571829474910515074683828862318181142924420699914240000000000000
138683118545689835737939019720389406345902876772687432540821294940160000000000000
8320987112741390144276341183223364380754172606361245952449277696409600000000000000
507580213877224798800856812176625227226004528988036003099405939480985600000000000000
31469973260387937525653122354950764088012280797258232192163168247821107200000000000000
1982608315404440064116146708361898137544773690227268628106279599612729753600000000000000
126886932185884164103433389335161480802865516174545192198801894375214704230400000000000000
8247650592082470666723170306785496252186258551345437492922123134388955774976000000000000000
544344939077443064003729240247842752644293064388798874532860126869671081148416000000000000000
36471110918188685288249859096605464427167635314049524593701628500267962436943872000000000000000
2480035542436830599600990418569171581047399201355367672371710738018221445712183296000000000000000
171122452428141311372468338881272839092270544893520369393648040923257279754140647424000000000000000
11978571669969891796072783721689098736458938142546425857555362864628009582789845319680000000000000000
850478588567862317521167644239926010288584608120796235886430763388588680378079017697280000000000000000

61234458376886086861524070385274672740778091784697328983823014963978384987221689274204160000000000000000

4470115461512684340891257138125051110076800700282905015819080092370422104067183317016903680000000000000000

330788544151938641225953028221253782145683251820934971170611926835411235700971565459250872320000000000000000

24809140811395398091946477116594033660926243886570122837795894512655842677572867409443815424000000000000000

1885494701666050254987932260861146558230394535379329335672487982961844043495537923117729972224000000000000000

145183092028285869634070784086308284983740379224208358846781574688061991349156420080065207861248000000

11324281178206297831457521158732046228731749579488251990048962825668835325234200766245086213177344000000

894618213078297528685144171539831652069808216779571907213868063227837990693501860533361810841010176000

71569457046263802294811533723186532165584657342365752577109445058227039255480148842668944867280814080000

5797126020747367985879734231578109105412357244731625958745865049716390179693892056256184534249745940480000

475364333701284174842138206989404946643813294067993328617160934076743994734899148613007131808479167119

39455239697206586511897471180120610571436503407643446275224357528369751562996629334879591940103770870900

3314240134565353266999387579130131288000666286242049487118846032383059131291716864129885722968716753150

281710411438055027694947944226061159480056634330574206405101912752560026159795933451040286452340924018

24222709538367273238176552320344125971528487055242938175083876449672016224974245027678946463490131946550

210775729837952771721360051869938959522978373806135621232297251121465411572759317408068342323641479350

185482642257398439114796845645546284380220968949399346684421580986889562184028199319100141244804501828

165079551609084610812169192624536193098396662364965418549135207078331710343785097393999125707876006627

148571596448176149730952273362082573788556996128468876694221686370498539309406587654599213137088405964

135200152767840296255166568759495142147586866476906677791741734597153670771559994765685283954750449427

124384140546413072554753243258735530775779917158754143568402395829381377109835195184430461238370413473

115677250708164157475920516230624043621475322957641353518614228121324680712146731521520328951684484530

108736615665674308027365285256786601004186803580182872307497374434045199869417927630229109214583415458

103299784882390592625997020993947270953977463401173728692122505712342939875947031248717653753854244685

99167793487094968920957140154189380115818364865126779544437605483849222280909149999876894760370007489820

96192759682482119853328425949563698712343813919172976158104477319333745612481875498805879175589072651200

942689044888324774562618574305724247380969376407895166349423877729470707002322379888297615920772911982

933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862

933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862
```

Tagged as: Algorithms, Java, Mathematics, Optimizations, Programming Challenges, SPOJ 3 Comments

Java Implementation of Binary Search Tree Insert and Traversal Methods



If you never heard about the Binary Search Tree (BST) data structure, then you better get used to it, as it represents the basis of many interview questions regarding algorithms. According to Thomas H. Cormen et. al. Introduction to Algorithms (Third Edition), Chapter 12: Binary Search Trees, a BST is defined as a linked data structure in which each node is represented as an object. In addition to having a key (i.e. value), each node has a left child, right child, and parent field, which link the node to its child nodes and parent node. Thus, each node can have at most one parent, or no parent (if it's a root node); and each node can have at most two (thus the name binary) children (i.e. left and right child), or no children at all (if it's a leaf node). Finally, the left child node of each BST node must have a key (i.e. value) that is smaller; and the right child node of each BST node must have a key that is greater. Having said that, a BST which conforms to the aforementioned properties is represented in Figure 1: Binary Search Tree.

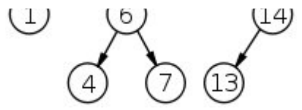


Figure 1: Binary Search Tree

The Java implementation of the BST data structure represented in this post, will consist of a **Node** class, and a **BinarySearchTree** class. The Node class will hold the basic information on a BST node, such as the key, and the fields for the left child, right child, and the parent node. The BinarySearchTree class on the other hand will hold the reference to the root node, as well as the methods for inserting and traversing the BST data structure (i.e. preorder, inorder, and postorder tree traversal).

Having said that, the Node class has the following structure:

```java
public class Node {

    private int key;
    private Node parent;
    private Node leftChild;
    private Node rightChild;

    public Node(int key, Node leftChild, Node rightChild) {
        this.setKey(key);
        this.setLeftChild(leftChild);
        this.setRightChild(rightChild);
    }

    public void setKey(int key) {
        this.key = key;
    }

    public int getKey() {
        return key;
    }

    public void setParent(Node parent) {
        this.parent = parent;
    }

    public Node getParent() {
        return parent;
    }

    public void setLeftChild(Node leftChild) {
        this.leftChild = leftChild;
    }

    public Node getLeftChild() {
        return leftChild;
    }

    public void setRightChild(Node rightChild) {
        this.rightChild = rightChild;
    }

    public Node getRightChild() {
        return rightChild;
    }
}
```

In order to proceed with the Java implementation for the BST insertion, we will look at Cormen et. al. pseudocode which has the following structure:

```
Tree-Insert(T, z)
    y = NIL
    x = T.root
    while x != NIL
        y = x
        if z.key < x.key
            x = x.left
        else x = x.right
    z.p = y
    if y == NIL
        T.root = z
    elseif z.key < y.key
        y.left = z
    else y.right = z
```

Thus, based on the above mentioned pseudocode, as well as taking in consideration few adjustments, we have the following Java implementation for BST insertion:

```java
public void insert(int key) {
    insert(new Node(key, null, null));
}

public void insert(Node z) {
```

```
        Node y = null;
        Node x = root;

        while (x != null) {
            y = x;

            if (z.getKey() < x.getKey()) {
                x = x.getLeftChild();
            } else {
                x = x.getRightChild();
            }
        }

        z.setParent(y);

        if (y == null) {
            root = z;
        } else if (z.getKey() < y.getKey()) {
            y.setLeftChild(z);
        } else {
            y.setRightChild(z);
        }
    }
```

As it can be seen, we can either insert an integer value in the tree (and thus automatically create a node) through the **insert(int)** method, or we can insert an existing node (or a binary search sub-tree), through the **insert(Node)** method.

The tree traversal methods on the other hand, are defined in such a manner that you can either start traversing from the root node, or you can start traversing from a specific node (i.e. sub-tree). Thus, for the preorder, inorder, and postorder traversal, the Java implementation is:

```
    public void preorderTraversal() {
        preorderTraversal(root);
    }

    public void preorderTraversal(Node node) {
        if (node != null) {
            System.out.print(node.getKey() + " ");
            preorderTraversal(node.getLeftChild());
            preorderTraversal(node.getRightChild());
        }
    }

    public void inorderTraversal() {
        inorderTraversal(root);
    }

    private void inorderTraversal(Node node) {
        if (node != null) {
            inorderTraversal(node.getLeftChild());
            System.out.print(node.getKey() + " ");
            inorderTraversal(node.getRightChild());
        }
    }

    public void postorderTraversal() {
        postorderTraversal(root);
    }

    private void postorderTraversal(Node node) {
        if (node != null) {
            postorderTraversal(node.getLeftChild());
            postorderTraversal(node.getRightChild());
            System.out.print(node.getKey() + " ");
        }
    }
```

All three tree traversal methods are implemented recursively, however they differ on their traversal approach, i.e.:
• **Preorder Traversal** proceeds with **parent node**, **left child node**, **right child node**.
• **Inorder Traversal** proceeds with **left child node**, **parent node**, **right child node**.
• **Postorder Traversal** proceeds with **left child node**, **right child node**, **parent node**.

It should be clear after testing the code that the inorder traversal produces a sorted output (in ascending order). With that said, the complete code for the BinarySearchTree class, is as follows:

```
    public class BinarySearchTree {

        private Node root;

        public void insert(int key) {
            insert(new Node(key, null, null));
        }

        public void insert(Node z) {

            Node y = null;
            Node x = root;
```

```java
        while (x != null) {
            y = x;

            if (z.getKey() < x.getKey()) {
                x = x.getLeftChild();
            } else {
                x = x.getRightChild();
            }
        }

        z.setParent(y);

        if (y == null) {
            root = z;
        } else if (z.getKey() < y.getKey()) {
            y.setLeftChild(z);
        } else {
            y.setRightChild(z);
        }
    }

    public void preorderTraversal() {
        preorderTraversal(root);
    }

    public void preorderTraversal(Node node) {
        if (node != null) {
            System.out.print(node.getKey() + " ");
            inorderTraversal(node.getLeftChild());
            inorderTraversal(node.getRightChild());
        }
    }

    public void inorderTraversal() {
        inorderTraversal(root);
    }

    private void inorderTraversal(Node node) {
        if (node != null) {
            inorderTraversal(node.getLeftChild());
            System.out.print(node.getKey() + " ");
            inorderTraversal(node.getRightChild());
        }
    }

    public void postorderTraversal() {
        postorderTraversal(root);
    }

    private void postorderTraversal(Node node) {
        if (node != null) {
            inorderTraversal(node.getLeftChild());
            inorderTraversal(node.getRightChild());
            System.out.print(node.getKey() + " ");
        }
    }
}
```

In order to test all the Java implementations displayed so far, we will use the **BinarySearchTreeTest** class, which will insert the integer sequence of **8, 3, 10, 1, 6, 14, 4, 7, 13**, and afterwards it will traverse the tree in a preorder, inorder, and postorder approach. With that said, the BinarySearchTreeTest implementation is as follows:

```java
public class BinarySearchTreeTest {
    public static void main(String[] args) {
        BinarySearchTree bst = new BinarySearchTree();
        int[] input = new int[] { 8, 3, 10, 1, 6, 14, 4, 7, 13 };

        for (int i : input) {
            bst.insert(i);
        }

        System.out.println("Preorder Traversal:");
        bst.preorderTraversal();

        System.out.println( "\nInorder Traversal:");
        bst.inorderTraversal();

        System.out.println("\nPostorder Traversal:");
        bst.postorderTraversal();
    }
}
```

After successful compilation and execution the BinarySearchTreeTest produces the following output:

```
Preorder Traversal:
8 3 1 6 4 7 10 14 13
Inorder Traversal:
1 3 4 6 7 8 10 13 14
Postorder Traversal:
1 4 7 6 3 13 14 10 8
```

As mentioned previously, the inorder traversal has produced a sorted output.

Do note that there is more than one way of implementing the BST insert and traversal methods. However, the implementation included in this post primarily follows the pseudocode of Cormen et. al. in the [Introduction to Algorithms (Third Edition)](#).

Finally, due to the vast implementation and importance of the BST data structure, I will follow-up with several other posts, with the next one being Java Implementation of the Binary Search Tree Search and Delete Methods. Needless to say, I look forward to your feedback.

Tagged as: Algorithms, Binary Search Tree, Data Structures, Java 2 Comments

3-Key Differences between LinkedList and ArrayList

The LinkedList and ArrayList represent some of the widely used data structures of the Java Collections Framework. You might have had a chance to already use both classes several times so far; however, if you have wondered on what makes them fundamentally different, then continue reading.

Although both the LinkedList and ArrayList classes implement the List interface, they differ on the following 3-key issues: 1) Data Storage, 2) Data Access, 3) Data Management.

DATA STORAGE: Linked-list versus Array

• **LinkedList** uses the design of a linked-list data structure (from where it get its name) to store data elements (i.e. objects). In this manner, each data element -- depending on the position of the list -- has a link to the previous and next element. This storage approach proves to be very efficient, as it uses only as much memory as it needs; no more, no less.
• **ArrayList** uses an Object array for storing the data elements. By default the array starts with a capacity of 10; unless otherwise specified at the time of instantiation. For example, if we want to instantiate an ArrayList of strings with an initial capacity of 1024 then we would use:

```
ArrayList<String> stringArrayList = new ArrayList<String>(1024);
```

Thus, by default the ArrayList can be more wasteful as far as the memory allocation goes.

DATA ACCESS: Sequential Access versus Random Access

• **LinkedList** due to the nature of its data storage, has a sequential access approach. Thus, if we want to retrieve an element at a specific index through the public E get(int index) method, we would have to iterate either from the beginning or the end, depending if the requested index is greater than or less than half the length of the list. It can easily be seen how this sequential access can become quite timely as we deal with longer and longer lists. As the **public E get(int index)** method uses the **Node node(int index)** method for data retrieval, we can see the implications of LinkedList's sequential access in the following code:

```
/**
 * Returns the (non-null) Node at the specified element index.
 */
Node<E> node(int index) {
    // assert isElementIndex(index);

    if (index < (size >> 1)) {
        Node<E> x = first;
        for (int i = 0; i < index; i++)
            x = x.next;
        return x;
    } else {
        Node<E> x = last;
        for (int i = size - 1; i > index; i--)
            x = x.prev;
        return x;
    }
}
```

• **ArrayList** supports random access as it has an internal array for data storage. Thus, when we call the public E get(int index) method in an ArrayList instance, we get forwarded to the **E elementData(int index)** method which simply returns the element located at the specified position within the array, i.e.:

```
// Positional Access Operations

@SuppressWarnings("unchecked")
E elementData(int index) {
    return (E) elementData[index];
}
```

Needless to say, ArrayList's random access approach is far more efficient, as it doesn't have to waste computing cycles on iteration.

DATA MANAGEMENT: Linking and Unlinking versus Array Copying

• **LinkedList** can very efficiently undergo any changes in its collection of data elements. For example, if we want to remove a specific data element from the LinkedList instance -- through the public boolean remove(Object o) method -- then we go ahead and link the neighbors (i.e. previous and next data elements) of the unwanted data element, and the Garbage Collector will take care of the rest. Thus, if we have a linked-list structure with three string data elements (with previous and next links), as follow:

```
["A"] <-> ["B"] <-> ["C"]
```

Then, when we request removal of the "B" string data element, "A" and "C" get linked together, and "B" is left alone, thus becoming eligible for Garbage Collection:

```
["A"].next = ["B"].next => ["A"].next = ["C"]
["C"].prev = ["B"].prev => ["C"].prev = ["A"]
```

The Java implementation behind this pseudo-code looks like the following:

```java
/**
 * Unlinks non-null node x.
 */
E unlink(Node<E> x) {
    // assert x != null;
    final E element = x.item;
    final Node<E> next = x.next;
    final Node<E> prev = x.prev;

    if (prev == null) {
        first = next;
    } else {
        prev.next = next;
        x.prev = null;
    }

    if (next == null) {
        last = prev;
    } else {
        next.prev = prev;
        x.next = null;
    }

    x.item = null;
    size--;
    modCount++;
    return element;
}
```

Do note that the code displayed above is for the **E unlink(Node x)** method on which LinkedList's **public boolean remove(Object o)** method relies. In addition to the efficiency of the remove method, LinkedList is very flexible when it comes to enlarging or shrinking its collection of data elements. In a LinkedList instance whenever we need to add to the collection, a call is made to the **void linkLast(E e)** method, which simply appends the new data element to the end of the list, as seen below:

```java
/**
 * Links e as last element.
 */
void linkLast(E e) {
    final Node<E> l = last;
    final Node<E> newNode = new Node<E>(l, e, null);
    last = newNode;
    if (l == null)
        first = newNode;
    else
        l.next = newNode;
    size++;
    modCount++;
}
```

• **ArrayList** due to the usage of its internal array it copies all of the elements to a new array every time there's a removal request, or when there's not enough room to add a new element. Thus, for every new addition or removal from our ArrayList collection a call is made to the public void ensureCapacity(int minCapacity) method, which carries out the necessary copying and shifting of the elements, as seen below:

```java
/**
 * Increases the capacity of this <tt>ArrayList</tt> instance, if
 * necessary, to ensure that it can hold at least the number of elements
 * specified by the minimum capacity argument.
 *
 * @param   minCapacity   the desired minimum capacity
 */
public void ensureCapacity(int minCapacity) {
    modCount++;
    int oldCapacity = elementData.length;
    if (minCapacity > oldCapacity) {
        int newCapacity = (oldCapacity * 3)/2 + 1;
        if (newCapacity < minCapacity)
            newCapacity = minCapacity;
        // minCapacity is usually close to size, so this is a win:
        elementData = Arrays.copyOf(elementData, newCapacity);
    }
}
```

Needless to say, you can imagine how much work these operations carry out, especially when dealing with very large arrays.

In conclusion, we can see that both the LinkedList and ArrayList have their positives and negatives; however, by having a good understanding of these issues you can make a better decision on what implementation should you use, based on the data size and usage requirements.

Tagged as: Algorithms, Data Structures, Interview Questions, Java, Java Collections Framework, Linked Lists

No Comments

3-Ways to Reverse a String in Java

Reversing a String in Java is a task that I came across in several interview experiences that I've read about. Although there are several ways to do this task, it seems that [potential] employers are especially interested in the solution involving recursion. Overall, (in case you're wondering) this task is used to get a better assessment on your understanding of the programming flow. Having said that, I will list in detail 3-ways to solve this task, from which you can pick and choose:

First Solution: StringBuffer
The easiest way to reverse a String in Java is by using an instance of the StringBuffer class as it already contains a reverse() method. Thus, with this approach our reverse method will look like:

```
public String reverse(String s) {
    return new StringBuffer(s).reverse().toString();
}
```

However, using this solution in an interview will only show how much you're familiar with the Java API which is not necessary the point of the task.

Second Solution: Reverse For Loop
You can also reverse a String by traversing it from the end in a traditional for loop. For this approach you can either use a char array which would be somewhat more efficient than by creating a large String pool as a result of continuous concatenation to a String variable. To better elaborate my point, here are both versions of this solution:

```
/**
 * Reverse For Loop: Char Array
 */
public String reverse(String s) {
    char[] reverseStringArray = new char[s.length()];
    for (int i = s.length() - 1, j = 0; i != -1; i--, j++) {
        reverseStringArray[j] = s.charAt(i);
    }
    return new String(reverseStringArray);
}
```

```
/**
 * Reverse For Loop: String Variable
 */
public String reverse(String s) {
    String reverseStringVariable = "";
    for (int i = s.length() - 1; i != -1; i--) {
        reverseStringVariable += s.charAt(i);
    }
    return reverseStringVariable;
}
```

Third Solution: Recursion
Finally, here's the solution that most likely your [potential] employer will like to see:

```
public String reverse(String s) {
    if (s.length() <= 1) {
        return s;
    }
    return reverse(s.substring(1, s.length())) + s.charAt(0);
}
```

To better understand this recursive approach, let's trace for example a call to the reverse(String) method with the String "ABC" as an argument:

```
Step 1: "ABC" does NOT have a length equal to or less than 1.
Step 2: Call reverse(String) with "BC" as argument and concatenate to its return value "A".
Step 3: "BC" does NOT have a length equal to or less than 1.
Step 4: Call reverse(String) with "C" as argument and  concatenate to its return value "B".
Step 5: "C" has a length of 1. Return "C" and unroll the stack.
```

Now, as the stack unrolls, the following concatenation occurs: **"C" + "B" + "A"** thus we end up getting the String

"CBA", which is the exact reverse

Needless to say, I hope that you get a chance to use these solutions, and I look forward to hear about any different approaches that you've taken for solving this task.

Tagged as: Algorithms, Interview Questions, Java, String Algorithms — No Comments

Iterating through a LinkedList Instance

LinkedList is the Java Collections Framework implementation of the List interface. This implementation represents a widely used elementary data structure on which other data structures such as Queues, Stacks, Double-ended Queues (Deques) are built upon. As per the definition of the linked-list data structures, a link is kept between the previous and next neighbors of each inserted element; thus by default the LinkedList implementation keeps all elements sorted by insertion order.

To instantiate and populate a LinkedList implementation which contains the names of planets of our Solar System we could use:

```
LinkedList<String> linkedList = new LinkedList<String>();
linkedList.add("Mercury");
linkedList.add("Venus");
linkedList.add("Earth");
linkedList.add("Mars");
linkedList.add("Jupiter");
linkedList.add("Saturn");
linkedList.add("Uranus");
linkedList.add("Neptune");
```

To iterate through this LinkedList instance we can either use a ListIterator, or we can use a traditional for loop approach. With the ListIterator approach the iteration is carried out in the following way:

```
// ListIterator approach
ListIterator<String> listIterator = linkedList.listIterator();
while (listIterator.hasNext()) {
    System.out.println(listIterator.next());
}
```

Whereas the iteration through a traditional for loop approach is carried out as:

```
// Traditional for loop approach
for (int i = 0; i < linkedList.size(); i++) {
    System.out.println(linkedList.get(i));
}
```

It is important to stress out that it is more efficient to use a ListIterator, as the traditional for loop approach traverses the list every time either from the beginning or the end, depending if the requested index is greater than or less than half the length of the list (i.e. *index < size() >> 1*).

Thus, to put everything in perspective the LinkedList iteration example application looks like:

```
import java.util.LinkedList;
import java.util.ListIterator;

public class LinkedListIterationExample {
    public static void main(String[] args) {

        LinkedList<String> linkedList = new LinkedList<String>();
        linkedList.add("Mercury");
        linkedList.add("Venus");
        linkedList.add("Earth");
        linkedList.add("Mars");
        linkedList.add("Jupiter");
        linkedList.add("Saturn");
        linkedList.add("Uranus");
        linkedList.add("Neptune");

        // ListIterator approach
        ListIterator<String> listIterator = linkedList.listIterator();
        while (listIterator.hasNext()) {
            System.out.println(listIterator.next());
        }

        // Traditional for loop approach
        for (int i = 0; i < linkedList.size(); i++) {
            System.out.println(linkedList.get(i));
        }
    }
}
```

Tagged as: Algorithms, Data Structures, Java, Java Collections Framework, Linked Lists — No Comments

Older Entries »

Go to top ↑