Tutorial 9

Topics

Linked lists
Trees
Tutorial Problems
Design Exercises

Linked lists

From earlier experiences in this course, you have already worked with two of the most basic data structures, arrays and vectors. In this tutorial we cover more advanced data structures: linked lists, trees and graphs.

Although arrays are good for storing a set number of the same type of object, and vectors allow us to dynamically size the storage space, both these data structures still have a major drawback. If you want to insert or delete an entry you have to move all the elements “below” the entry up or down by copying them into adjacent elements. While there are ways around this (like just setting unused elements to null) the amount of wasted memory in such a system can be huge and you must modify code to deal with all the empty spaces. A more efficient way of storing data that will require numerous additions and deletions is the linked list.

A definition

A linked list is a series of objects who “know” where the next member of the list resides in the computer’s memory. The last member of the list typically indicates the next member is a "null". To accomplish this, each object in the list must have a member that can store the memory location of the next object in the list. This type of list is singly linked because the code can move along the list in only one direction. Doubly linked lists are also possible. A reference to the list object normally marks the beginning of the list. Operations on linked lists

Some of the basic operations on a linked list include:

- Creating a list
- Finding an element in the list
- Inserting an element at the end of the list
- Deleting an element from the list

Java implementation
The following code is a simple implementation of linked list: (NOTE: The code for finding an element in the list is included in the delete method for brevity): Here is the class that declares the basic data in each link of the list:

class Check {
   private static int totalChecks=0; // one variable to keep track of the total checks
   private int checkNumber;
   private double checkAmount;
   private Check nextCheck; // reference to the next check in the list of checks

   public Check ( double amount ) {
      checkNumber = ++totalChecks;
      checkAmount = amount;
      nextCheck = null;
   }
   public static int getTotalChecks() {
      return totalChecks;
   }
   public int getCheckNumber() {
      return checkNumber;
   }
   public Check getNextCheck() {
      return nextCheck;
   }
   public void setNextCheck(Check c) {
      nextCheck = c;
   }
   public void print() {
      System.out.println("Check Number : #" + checkNumber + "; Check Amount : $"+ checkAmount);
   }
}

Here is the code that provides the operators for the list (add, delete, is empty, etc.) An exception is included as an example in the deleteChecks method. It would be better if the code checked for null before searching for another link rather than relying on an exception to handle the NullPointerException that is generated by trying to read an object that does not exist.

class ListOfChecks {
   // create a reference to our list
   private Check head;    public ListOfChecks() {
      // set the list reference to null since the list
      // has no members
      Check head = null;
   }
   public boolean isEmpty() {
      return (head == null); // if list is empty return true
   }
   public boolean addToEnd ( double amount ) {
      if ( isEmpty() )
         // If the list is empty, start a new one

head = new Check(amount);
else {
    // "Walk down the list" until you find the end (i.e. where next check is null)
    Check temp = head;
    while (temp.getNextCheck() != null)
        temp = temp.getNextCheck();
    temp.setNextCheck(new Check(amount));    // Put a new check at the end

    return (true);           // ... and return true
}
public boolean deleteCheck ( int number ) {
    if ( isEmpty() )
        return (false);      // If there is no list, nothing is deleted
    else if (head.getCheckNumber() == number)
        head = head.getNextCheck();
        // The 'deleted' memory is now lost because there is no reference to it, but
        // collection returns the memory to the system.
    else {
        Check temp = head;
        Check previous = null;
        // I need to remember the check before the one I want to delete
        while (temp.getCheckNumber() != number) {
            // Before you move on, remember where you came from
            previous = temp;
            temp = temp.getNextCheck();
        }
        // When the loop ends temp will be the check
        // to be deleted and previous will be the
        // check before that. So we want the check
        // that now follows temp to follow previous
        System.out.println("Check " + temp.getCheckNumber() + " has been just
dele. ");
        previous.setNextCheck(temp.getNextCheck());
    }
    return (true);
}
public void print() {
    int total = 0;
    if (!isEmpty())
    {
        Check temp = head;
        while (temp!= null){
            temp.print();
            temp = temp.getNextCheck();
            total++;
        }
    }
    System.out.println("Total number of checks : " +total);
System.out.println("Total checks has been written : " + Check.getTotalChecks());
}
} The following code tests the above two classes.

public class ListOfChecksTest {

    public static void main(String args[]) {
        ListOfChecks myList = new ListOfChecks();
        myList.addToEnd(1830.50);
        myList.addToEnd(255.68);
        myList.addToEnd(99.99);
        myList.print();
        myList.deleteCheck(2);
        myList.addToEnd(140.50);
        myList.print();
    }
}

Trees

Trees are data structures that look exactly like the way they sound, except most of the time they are drawn upside down, like roots. They are the next significant way of organizing data after lists.

Terminology

Here is a list of the terminology used when talking about trees:

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>The smallest piece of a tree, that stores some abstract data.</td>
</tr>
<tr>
<td>Root node</td>
<td>The tree starts from one root node and branches out into one or more child nodes, which branch out to one or more other child nodes. A root node may have no child node; it is the single node in the tree and it is a leaf node.</td>
</tr>
<tr>
<td>Child node</td>
<td>A node that branches from a parent node. A child node has one parent.</td>
</tr>
<tr>
<td>parent node</td>
<td>A node that branches into child nodes. Parent nodes can have one or more child node.</td>
</tr>
<tr>
<td>leaf node</td>
<td>A node without children.</td>
</tr>
<tr>
<td>subtree</td>
<td>The part of a tree that stems from a particular node, i.e. that node and all its descendants.</td>
</tr>
<tr>
<td>ancestor</td>
<td>An ancestor of a node A is any node in between and including the root node and the parent of A.</td>
</tr>
<tr>
<td>descendant</td>
<td>A descendant of a node A is any node that is a child of A or a descendant thereof. All the nodes in the subtree stemming from A are descendants of A.</td>
</tr>
<tr>
<td>depth</td>
<td>How deep the tree is; the greatest number of steps required to reach any leaf node from the root node.</td>
</tr>
<tr>
<td>sorted tree</td>
<td>A sorted tree is a tree in which the children of a node are ordered in a particular way.</td>
</tr>
<tr>
<td>key</td>
<td>Some data used to identify another piece of data. For example, your student ID is a key for all your MIT records.</td>
</tr>
</tbody>
</table>
Data are stored in nodes. The tree starts from one root node and branches out into one or more nodes, which branch out to one or more other nodes and so forth until eventually terminating in leaf nodes. Parent nodes branch out into child nodes, the parent.

There are several possible sequences for traversing a tree. The three most common are:

**Preorder:**

```
root
left subtree
right subtree
```

**Inorder:**

```
left subtree
root
right subtree
```

**Postorder:**

```
left subtree
right subtree
root
```

Below is a recursive algorithm to perform the Inorder traversal of a binary tree.

```java
public class BinaryTree
{
    private Object value;
    private BinaryTree left = null;
    private BinaryTree right = null;
    public Vector getInorder()
    {
        Vector vec = new Vector();
        return traverseInorder( this, vec );
    }
    private Vector traverseInorder( BinaryTree b, Vector v )
    {
        if ( b != null )
        {
            traverseInorder( left, v );
            v.addElement( value );
            traverseInorder( right, v );
        }
    }
}
```
Tutorial Problems

Problem 1

All elements in every linked list always have:

a. A reference to another element of the list or a NULL pointer.
b. No references.
c. A reference to the first element of the list.
d. A reference to the last element of the list.

Problem 2

If a binary tree has the parent's key value less than the left child's but greater than the right child's, which of the following traversals will print data out in descending order?

a. Pre-order.
b. In-order.
c. Post-order.
d. None of the above.

Design Exercises

1. Complete the BinaryTree class with three different ways of traverse (i.e., preorder, inorder, and postorder) and a main() method for testing. In your main(), first construct your binary tree, than traverse them in three different way and print out the traverse result. 2. The 1.00 TAs have decided to write a Java application to keep track of students' scores for Quiz2. In particular, they have decided to use a linked list as the data structure for storing all students' scores. Help them to write a method called count() which returns the number of students who received scores higher than or equal to a certain threshold.

First, help the TAs to define a class called Student to store each student's ID and score, and the access that returns the grade of the student. Then, help the TAs to write a StudentListTest class with a main function and a count() method. In the main function, please create a list that store several students' information. You then call the count() method to find out those students whose scores are higher than the threshold score. Your count() method has the following signature:

public static int count(List s, int threshold)
The first argument s is a reference to a linked list of Student objects. The second argument specifies the score threshold. We wish to count the number of students who received grade higher than or equal to the threshold. The method returns the count result as an integer. You should not modify List s.

Hint: You are recommended to use java.util.LinkedList class and java.util.Iterator interface for this problem.