## 1 Linked Data Structures

The other type of data structures are what's called "Linked Data Structures". These revolve around an object called a "Node", which are used to store chunks of data. The node itself is a data structure that usually contains data, and links to other nodes.

Many people think that there are no pointers in Java, but just because you don't see them directly, doesn't mean they're not there. In fact, you can treat any object as a pointer (or more correctly, a reference).

Thus, the Node structure should have a data element, and a reference to another node (or nodes). Those other nodes which are referenced to, are called *child* nodes. The node itself is called the *parent* node (or sometimes a "father" node) in reference to its children (nice big happy family)

The node we create will be a one child node (it will have only one pointer), and we will later use it to build really cool data structures. The source for our one child node follows:

Listing 1: Node with one child reference.

```
/*
 * A list node with 1 child reference.
 */
public class pOneChildNode {

    protected Object data;
    protected pOneChildNode next;

    public pOneChildNode(){
        next = null;
        data = null;
    }
    public pOneChildNode(Object d,pOneChildNode n){
        data = d;
        next = n;
    }
    public void setNext(pOneChildNode n){
        next = n;
    }
    public void setData(Object d){
        data = d;
    }
    public pOneChildNode getNext(){
        return next;
    }
    public Object getData(){
        return data;
    }
    public String toString(){
        return ""+data;
    }
}
```

Go over the source, notice that it's nothing more than just set and get functions (pretty simple). The two data members are the data and next. The data member holds the data of the node, and next holds the pointer to the next node. Notice that next is of the same type as the class itself; it effectively points to the object of same class!

The String toString() method is Java's standard way of printing things. If an object wants to be printed in a special way it will define this method, with instructions on how to print itself. In our case, we just print the data. Adding data to a bunch of quotation marks automatically converts it to type String (hopefully, our data will also have a toString() method defined on it). Without this method, we get the default (defined in java.lang.Object class) toString, which isn't very readable.

Node based data structures provide for dynamic growing and shrinking, and are the key to some complex algorithms (as you'll see later). Now that we know how to implement a 'Node', lets get to something cool.

## 2 Linked Lists

A linked list is just a chain of nodes, with each subsequent node being a child of the previous one. Many programs rely on linked lists for their storage because these don't have any evident restrictions. For example, the array list we did earlier could not grow or shrink (if we wanted to make it grow or shrink, we would have to allocate a new array of bigger capacity, and copy all data into the new array), but node based ones can! This means there is no limit (other than the amount of memory) on the number of elements they can store.

A linked list has just one node, the "head", that node has links to subsequent nodes. So, the entire list can be referenced from that one node. The last node in the chain of nodes usually has some special feature to let us know that it's last. That feature, most of the time is a **null** pointer to the next node.

```
[node0]->[node1]->[node2]->[node3]->[node4]->null
```

The example above illustrates the node organization inside the list. In it, node0 is the head node, and node4 is the last node, because its pointer points to **null**. Well, now that you know how it's done, and what is meant by a linked list, lets write one.

Listing 2: Singly linked list.

```
/*
 * singly linked list
 */
public class pLinkedList{

    protected pOneChildNode head;
    protected int number;

    public pLinkedList(){
        head = null;
        number = 0;
    }
    public boolean isEmpty(){
        return head == null;
    }
    public int size(){
        return number;
    }
    public void insert(Object obj){
        head = new pOneChildNode(obj,head);
        number++;
    }
    public Object remove(){
        if(isEmpty())
            return null;
        pOneChildNode tmp = head;
        head = tmp.getNext();
        number--;
        return tmp.getData();
    }
    public void insertEnd(Object obj){
        if(isEmpty())
            insert(obj);
        else{
            pOneChildNode t = head;
            while(t.getNext() != null)
                t=t.getNext();
            pOneChildNode tmp =
                new pOneChildNode(obj,t.getNext());
            t.setNext(tmp);
            number++;
        }
    }
    public Object removeEnd(){
        if(isEmpty())
            return null;
        if(head.getNext() == null)
            return remove();
        pOneChildNode t = head;
        while(t.getNext().getNext() != null)
            t = t.getNext();
        Object obj = t.getNext().getData();
        t.setNext(t.getNext().getNext());
        number--;
        return obj;
    }
    public Object peek(int n){
        pOneChildNode t = head;
        for(int i = 0;i<n && t != null;i++)
            t = t.getNext();
        return t.getData();
    }
}
```

Before we move on, lets go over the source. There are two data members, one named head, and the other named number. Head is the first node of the list, and number is the total number of nodes in the list. Number is primarily used for the size() method. The constructor, pLinkedList() is self explanatory. The size() and isEmpty() methods are also pretty easy.

Here comes the hard part, the insertion and removal methods. Method insert(Object) creates a new pOneChildNode object with next pointer pointing to the current head, and data which is inserted. It then sets the head to that new node. Notice that the old head is still saved, and the new node points to it.

Method Object remove() works in a very similar fashion, but instead of inserting, it is removing (obviously). It first checks to see if the list is isEmpty() or not, if it is, it returns a **null**. It then saves the current head node, and then changes it to accommodate the removal, decrements the number, and returns the data from the previously saved node.

In the method insertEnd(Object), we are actually inserting at the end of the list. We first check to see if the list is isEmpty(), if it is, we do a regular insertion (since it really doesn't matter which direction we're coming from if the list is empty). We then setup a loop to search for the end. The end is symbolized by the next pointer of the node being **null**. When we get to the end, we create a new node, and place it at the end location. Incrementing number before we return.

Method Object removeEnd() works in a similar fashion as insertEnd(Object) method. It also goes through the whole list to look for the end. At the beginning, we check if the list is not isEmpty(), if it is, we return a **null**. We then check to see if there is only one element in the list, if there is only one, we remove it using regular remove(). We then setup a loop to look for the node whose child is the last node. It is important to realize that if we get to the last node, we won't be able to erase it; we need the last node's *parent— node. When we find it, we get the data, setup necessary links, decrement number, and return the data.*

*The Object peek(**int**) method simply goes through the list until it either reaches the element requested, or the end of the list. If it reaches the end, it should return a **null**, if not, it should return the correct location inside the list.*

*A note about performance: Normally, you wouldn't use operations that require you to loop through the whole list. So those inserts/removals with loops are generally never used (in fact, they're generally not even implemented). For every data structure you design, you should strive to make manipulation times constant or at worst logarithmic. Linear (or worse) performance is not acceptable!*

*To test, we convert pArrayListTest driver to accommodate this class:*

Listing 3: Test driver for singly linked list.

```
/*
 * linked list test driver.
 */
public class pLinkedListTest{
     public static void main(String[] args){
          pLinkedList l = new pLinkedList();
          Integer j = null;
          int i;
          System.out.println("starting...");
          for(i=0;i<5;i++){
               j = new Integer((int)(Math.random() * 100));
               l.insert(j);
               System.out.println("insert: " + j);
          }
          for(;i<10;i++){
               j = new Integer((int)(Math.random() * 100));
               l.insertEnd(j);
               System.out.println("insertEnd: " + j);
          }
          for(i=0;i<l.size();i++)
               System.out.println("peek "+i+": "+l.peek(i));
          for(i=0;i<5;i++)
               System.out.println("remove: " + ((Integer)l.remove()));
          while(!l.isEmpty())
               System.out.println("removeEnd: " + ((Integer)l.removeEnd()));
          System.out.println("Done :-)");
     }
}
```

*The test driver is nothing special, it just a simple conversion of the old test driver; wont spend any time discussing it. The output follows.*

```
starting...
insert: 65
insert: 78
insert: 21
insert: 73
insert: 62
insertEnd: 82
insertEnd: 63
insertEnd: 6
insertEnd: 95
insertEnd: 57
peek 0: 62
peek 1: 73
peek 2: 21
peek 3: 78
peek 4: 65
peek 5: 82
peek 6: 63
peek 7: 6
peek 8: 95
peek 9: 57
remove: 62
remove: 73
remove: 21
remove: 78
remove: 65
removeEnd: 57
removeEnd: 95
removeEnd: 6
removeEnd: 63
removeEnd: 82
Done ;-)
```

*Look over the output, make sure you understand why you get what you get. Linked lists are one of the most important data structures you'll ever learn, and it really pays to know them well. Don't forget that you can always experiment. One exercise you shold do is create a circular list. In a circular list, the last node is not pointing to* **null**, *but to the first node (creating a circle). Sometimes, lists are also implemented using two pointers; and there are many other variations you should consider and try yourself. You can even make it faster by having a "dummy" first node and/or "tail" node. This will eliminate most special cases, making it faster on insertions and deletions.*