Data Structures - Linked Lists 01

Dr. TGI Fernando [1] [2]

[1] Email: tgi.fernando@gmail.com
[2] URL: http://tgifernando.wordpress.com/

Introduction

Problems with arrays

- Unordered array - searching is slow, deletion is slow
- Ordered array - insertion is slow, deletion is slow
- Arrays have a fixed length

Linked lists solves some of these problems.

Linked lists - general purpose storage structures after arrays

Applicable in many cases in which arrays can be used UNLESS you need frequent random access to individual items using an index.

Node

- Each data item is embedded in a node.
- Each node has
  - an item
  - a **reference** to next node in the list

```
public class Node {
    public int item;     // Data
    public Node next;    // reference to next node
}
```

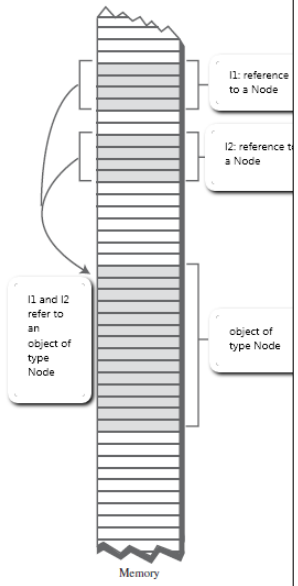
- Recursive data structure: Definition of the Node class referred to Node itself.

- Each node object contains a reference (next) to the next node in the list.

Reference

A reference is a number that refers to an object. It's the object's address in the computer's memory, but you don't need to know its value; you just treat it as a magic number that tells you where the object is.

```
public static void main(String[] args) {
    Node l1, l2;     // references to Node
    Node objNode = new Node (); // Node
     object
    objNode.item = 10;
    l1 = l2 = objNode;  // l1 and l2
     refers to objNode
    System.out.println("l2 item = " + l1.
     item);
    System.out.println("l2 item = " + l2.
     item);
}
```



Null pointer

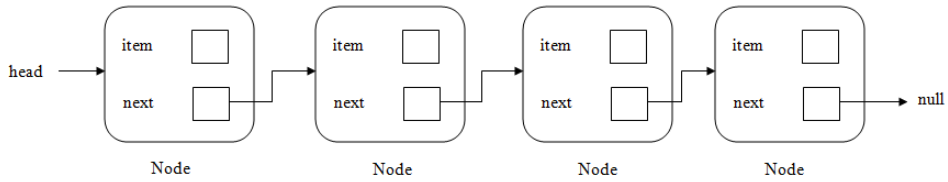
- has a value reserved for indicating that the pointer does not refer to a valid object.
- routinely used to represent conditions such as the end of a list of unknown length or the failure to perform some action.

Null reference

In Java programming, null can be assigned to any variable of a reference type (that is, a non-primitive type) to indicate that the variable does not refer to any object or array.

Linked List

- A linked list (or simply a list) made up of nodes.
- Each node is connected to the next node by the reference 'next.'
- Last node's next is set to 'null.'



Relationship, Not position

- In an array each item occupies a particular position.
- This position can be directly accessed by using an index.
- In a list the only way to find a particular element is to follow along the chain of elements.

The complete `Node` class

In addition to data, this class contains a constructor and a method, `displayNode()`, that displays the node's data item.

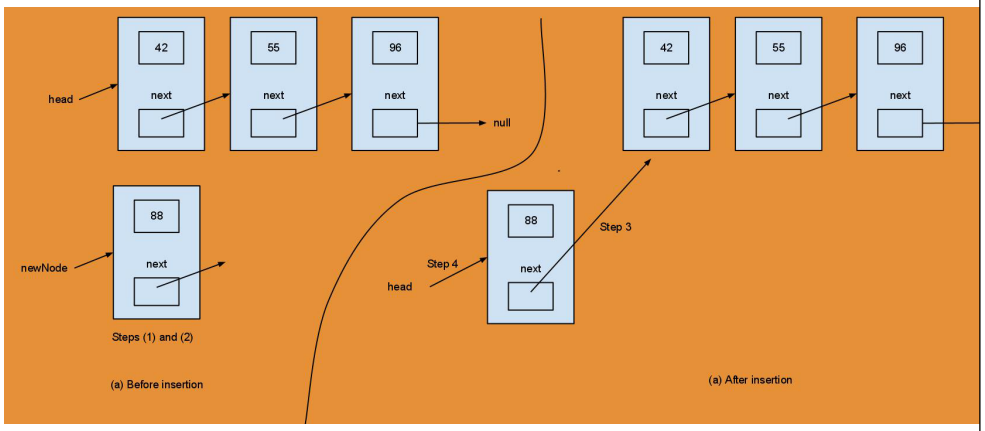
```
class Node {
    public int item;    // data
    public Node next;   // reference to next node
    //-----------------------------------------
    public Node (int i) { // constructor
        item = i; // initialize data
        next = null; // not necessary; automatically set to
         null
    }
    //-----------------------------------------
    public void displayNode () { // display node item
        System.out.print(item);
    }
} // end class Node
```


The `LinkList` class

```
class LinkList {
    private Node head; // reference to first node on list
    //-----------------------------------------
    public void LinkList () { // constructor
        head = null; // empty list - no items on list yet
    }
    //-----------------------------------------
    public boolean isEmpty () { // true if list is empty
        return (head == null);
    }
    //-----------------------------------------
    // other methods go here
}
```

- Contains only one data item: a reference (`head`) to the first node on the list.
- The reference "`head`" finds the other nodes by following the chain of references from `head`, using each node's next field:

Inserting an item at the beginning of the list



Inserting an item at the beginning of the list (Contd.)

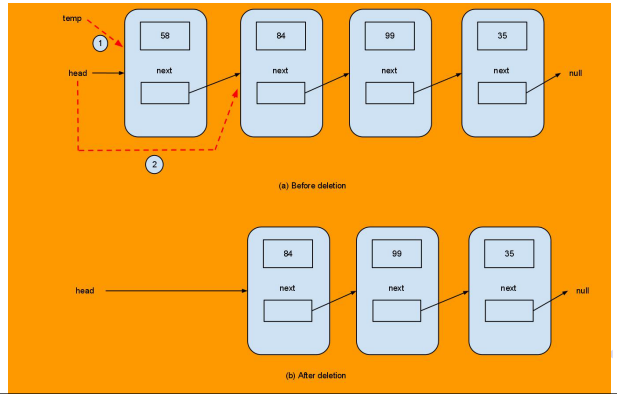
This is the simplest way to add an item to a linked list.

1. allocate space for a new node,
2. copy the item into it,
3. make the new node's next pointer point to the current head of the list and
4. make the head of the list point to the newly allocated node.

```
public void insertFirst (int i) { // inserts at start of
list
    Node newNode = new Node (i); // make a new node
    newNode.next = head; // newNode's next pointer to point
     current head
    head = newNode; // head points to the newNode
}
```


Deleting the first node

- Reverse of `insertFirst ()` method.
- Removes the first node from the list and
- `head` points to second node on the list.
- This second node is found by looking at the `next` field in the first node.
- Returns the reference of the deleted node.
- `deleteFirst()` method assumes the list is not empty.



The `displayList()` method

- Start from `head` and follow the chain of references from node to node.
- A reference variable (to a node) "`current`" points to each node in turn.
- Initially "`current`" points to "`head`," which holds a reference to the first node.
- The statement

  `current = current.next;`

  changes "`current`" to point to the next node on the list.
- The end of list is detected by by the "`next`" field of the last node pointing to `null` rather than another node.

```
while (current != null) { // until end of list
    ...
}
```

The LinkList class

```
class LinkList {
    private Node head; // reference to first node on list
    //--------------------------------------------
    public void LinkList () { // constructor
        head = null; // empty list - no items on list yet
    }
    //--------------------------------------------
    public boolean isEmpty () { // true if list is empty
        return (head == null);
    }
    //--------------------------------------------
    public void insertFirst (int i) { // inserts at start of
      list
        // ...
    }
    //--------------------------------------------
    public Node deleteFirst () { // delete first item (
     assumes list not empty)
        // ...
    }
    //--------------------------------------------
    public void displayList () { // displays items on the
     list
        // ...
    }
}
```


Lab Work 04 - LinkList class

1. Complete the coding of the `LinkList` class.
2. Include two methods `find()` and `delete()` described in slides 15 and 16.
3. Write a class (`LinkListApp`) to use the methods defined in the `LinkList` class appropriately.

The `find()` method

- Searches for a key in the list.
- Work much like `displayList()` method.
- The reference "`current`" initially points to `head` and in each turn it moves to the next node.
- At each node, `find()` checks whether that node's key is the one it's looking for.
- **Output**
  - If the key is found, it returns with a reference to that node.
  - If `find()` reaches to the end of the list without finding the desired node, it returns "`null`."

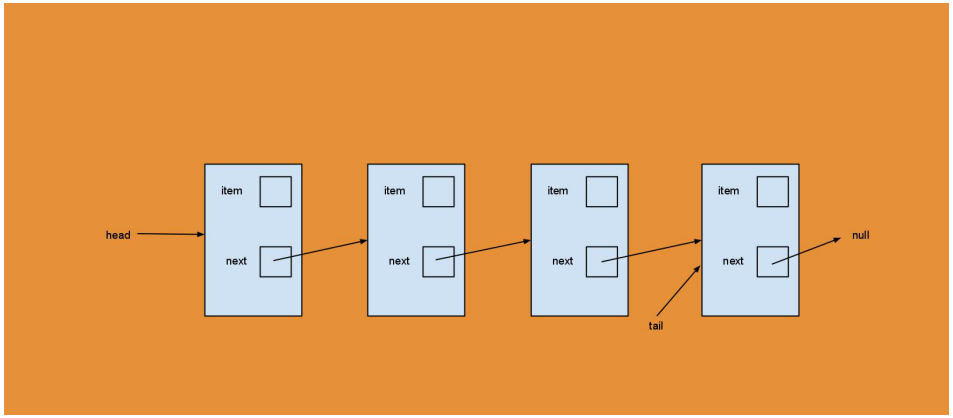
The `delete()` method

- Similar to `find()` method in the way it searches for the node to be deleted.
- Needs to maintain two references: current node ("`current`") and to the node preceding the current node ("`previous`").
- At each cycle thorough a while loop, just before "`current`" is set to "`current.next`", "`previous`" is set to "`current`."
- To delete the current node (once found it)

  `previous.next = current.next;`
- **Special case:** If the node to be deleted is the first node, the node is deleted by changing "`head`" to "`head.next`."
- The code that covers these two possibilities:

```
if (current == head)                    // if first node,
    head = head.next;                   // change head
else                                    // otherwise
    previous.next = current.next;       // bypass node
```

Double-ended list



- Similar to an ordinary linked list; but it has one additional reference ("`tail`") to the last node in addition to the reference "`head`."

Double-ended list (Contd.)

- This tail node allows to insert a new node directly at the end of the list.
- You can insert a new node at the end of an ordinary single-ended list by iterating through the entire list until you reach the end.
- But this approach is **inefficient**.
- Suitable for some applications (**E.g.** queue).

The `FirstLastList` class

```
class FirstLastList { // implements a double-ended list
    private Node head;  // ref to first node
    private Node tail;  // ref to last node
    //--------------------------------------
    public FirstLastList () {...} // constructor
    //--------------------------------------
    public boolean isEmpty () {...} // true if no nodes
    //--------------------------------------
    public void insertFirst(int i) {...} // insert at front
     of list
    //--------------------------------------
    public void insertLast(int i) {...} // insert at end of
     list
    //--------------------------------------
    public Node deleteFirst() {...}// delete first link
    //--------------------------------------
    public void displayList() {...}
    //--------------------------------------
}
```


Efficiency of a linked list

- Insertion and deletion at the beginning
  - Very fast
  - Need only changing one or two references.
  - Takes O(1) time.
- Finding, deleting or inserting next to a specific item
  - Requires searching through, on average, half the items in the list.
  - O(N) comparisons (Arrays also O(N) for these operations).
  - Linked list - nothing needs to be moved when an item is inserted or deleted.
- A linked list uses exactly as much memory as it needs.
- A linked list can grow or shrink without wasting the memory or running out of room.