Data Structures - Linked Lists 02

Dr. TGI Fernando [1] [2]

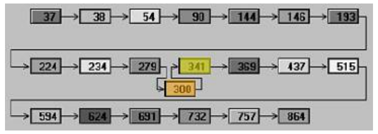
[1]Email: tgi.fernando@gmail.com
[2]URL: http://tgifernando.wordpress.com/

Sorted Lists

- In some applications it's useful to maintain the data in sorted order within the list.
- Items are arranged in sorted order by key value.
- Deletion is often limited to the smallest (or the largest) item in the list, which is at the start of the list.
- Sometimes `find()` and `delete()` methods are used in sorted lists.
- Sorted lists can be used in most situations in which you use a sorted array.
- Advantages over sorted array
  - * speed of insertion (no need to move elements)
  - * list can expand to fill available memory.
- However, a sorted list is difficult to implement than a sorted array.

Inserting an item

- The algorithm must search the appropriate place to put the item.
- This is just before the first item that's larger than the item to be inserted.



- Need two references current and previous to hold current node and the previously visited node respectively.
- To insert the new node

```
previous.next = newNode;
newNode.next = current;
```

Inserting an item (Contd.)

- However, it is needed to consider the following special cases.
  - Inserting the node at the beginning of the list.
  - Inserting the node at end of the list.
- Initially,

```
current = head;
previous = null;
```

- This set up is important because this `null` value determines whether we are still at the beginning of the list.
- `while` loop
  - The loop terminates when the key of the current node being examined is no longer smaller than the key of the node to be inserted.
  - The loop also terminates if `current` is `null`. This happens at the end of the list or if the list is empty.

Efficiency of sorted linked list

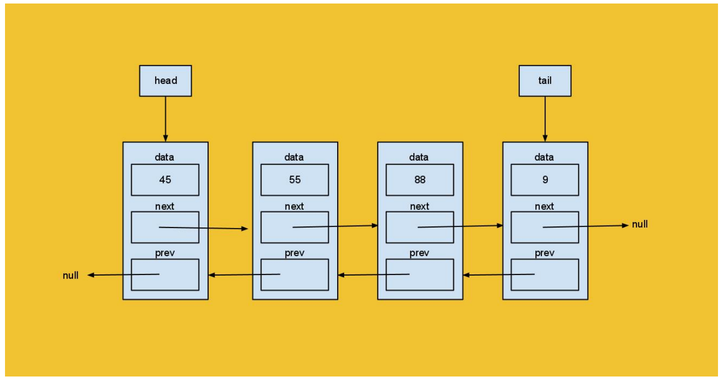
- Insertion and deletion of arbitrary item - `O(N)` comparisons (`N/2` on the average)
- The minimum value can be found, or deleted, in `O(1)` time.
- If an application frequently accesses the minimum item, and fast insertion isn't critical, then a sorted linked list is an effective choice.

Doubly Linked List

- Not a double-ended list.
- In an ordinary linked list, it's difficult to traverse backward along the list.
- **E.g.** Text editor - a linked list is used to store each text line.
  - Each text line is stored as a `String` object embedded in a node.
  - When a user move the cursor downward on the screen, the program steps to the next node to display the next line.
  - But what happens if the user moves the cursor upward?
  - In an ordinary linked list, the `current` must be moved back to the beginning of the list and then step all the way down to the required node.
  - This is not efficient.
  - You want to make a single step upward.

Doubly Linked List (Contd.)

- Doubly linked list allows you to traverse backward as well as forward through the list.
- Each node has two references: one to the next node (`next`) and other to the previous node (`prev`).



Doubly Linked List (Contd.)

- Doubly linked list's node class

```
class Node {
    public int data;     // data item
    public Node next;    // next node in list
    public Node prev;    // previous node in list
    ...
}
```

- Downside
  * To insert or delete a node, you must deal with four references: two references to the previous node and two references to the next node.
  * Each node is a little bigger because of the extra reference `prev`.
- Keeping a reference to the last node on the list (`tail`) is not necessary.

Traversal

- `displayForward()` method is the same as the `displayList()` method in an ordinary linked list.
- `displayBackward()` method is similar but starts at the last node on the list and proceeds towards the start of the list, visiting to each node's previous field (`prev`).

```
Node current = last;     // start at end
while (current != null) {   // until start of list,
    ...
    current = current.prev; // move to previous node
}
```

Insertion

<u>`insertFirst()`</u>

1. If the list is empty, change the tail field to new node.
   Else change the previous field in the old head node to point to the new node.
2. Change the next field in the new node to point to the old head node.
3. Finally, it sets head to point to the new node.

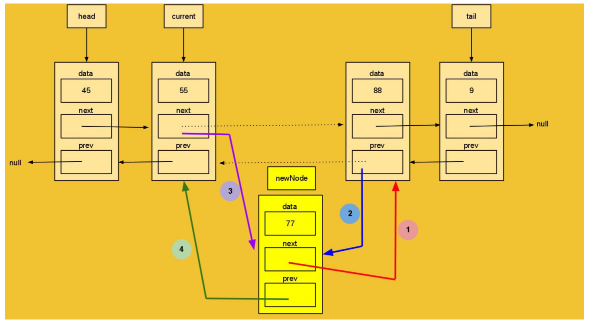
```
// step 1
if (isEmpty())  // if empty list
    tail = newNode;
else
    head.prev = newNode;
newNode.next = head;     // step 2
head = newNode; // step 3
```

<u>`insertLast()`</u> - same process applied to the end of the list; it's a mirror image of `insertFirst()`

Insertion (Contd.)

<u>`insertAfter()`</u> - inserts a new node following the node with a specified key value.

- First, the link with the specified key value must be found (same way as `find()`).
- Four connections need to be changed; two connections between the new node and the next node, two more between current and the new node.



Insertion (Contd.)

<u>`... insertAfter()`</u>

- If the new node will be inserted at the end of the list, its next field must point to `null` and
- the `tail` must point to the new node.

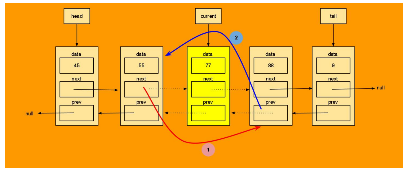
```
if (current == tail){   // new node inserted at end of
 list
    newNode.next = null;
    tail = newNode;
}
else{   // new node inserted at an arbitrary location
    newNode.next = current.next;     // step 1
    current.next.prev = newNode;     // step 2
}
newNode.prev = current;              // step 3
current.next = newNode;              // step 4
```

Deletion

Three deletion routines: `deleteFirst()`, `deleteLast()` and `deleteKey()`.

`deleteKey()`

- If the node to be deleted is neither the head nor the tail in the list.
  - **Step 1:** the `next` field of `current.prev` (the link before the one being deleted), and
  - **Step 2:** the `prev` field of `current.next` is set to point to `current.prev`.



- The `deleteKey()` must separately handle the cases if the node to be deleted is either the head node or tail node.

Iterators

Suppose you wanted to traverse a list, performing some operation on certain nodes.

E.g. Imagine a personnel file stored as a linked list. You might want to increase the wages of all employees who were being paid minimum wage, without affecting employees already above the minimum.

Arrays - This type of operations are easy because you can use an array index to keep track of your position. You can operate on one item, then increment the index to point to the next item, and see if that item is a suitable candidate for the operation.

Linked Lists

- Nodes don't have fixed index numbers.
- You could repeatedly use the `find()` to look for appropriate nodes.
- Need many comparisons and inefficient.

Iterators (Contd.)

A reference in the list itself?

- As users of the list class, we need to access a reference (in the linked list class) that can point to any arbitrary node.
- We should be able to increment the reference so we can traverse along the list.
- We should be able to access the node pointed to by the reference.

Where will it be placed?

- One possibility is to use a field in the list itself (say `current`).
- But we need more than one such reference, just as we can use several array indices at the same time.
- So it is implemented as a separate class in which its objects containing a references to nodes in data structures.

An Iterator class

Objects containing references to items in data structures, used to traverse these structures, are commonly called **iterators** (or sometimes, as in certain Java classes, **enumerators**).

```
class ListIterator {
    private Node current;
    ...
}
```

The current field contains a reference to the node the iterator currently points to.

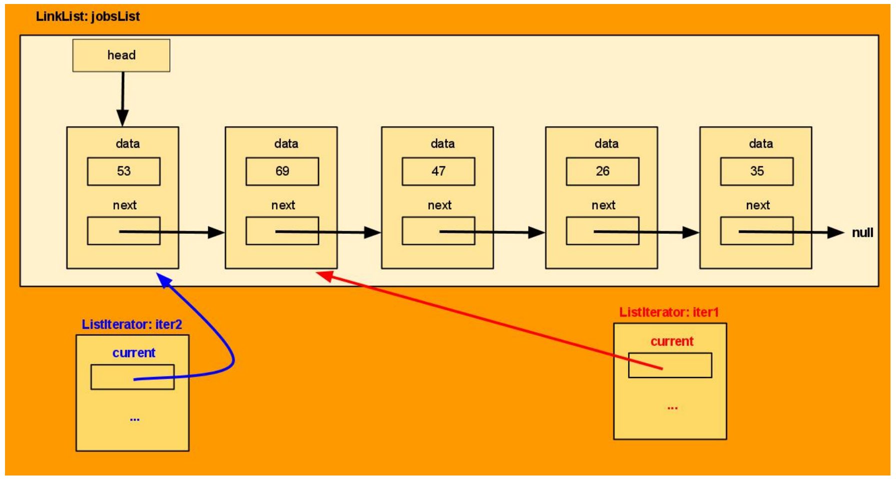
To use such an iterator, the user might create a list and then create an iterator object associated with the list.

main() method

```
public static void main(String[] args) {
    LinkList jobsList = new LinkList ();
    ...
    ListIterator iter1 = new ListIterator (jobsList); //
     iter1 associated with jobsList
    ListIterator iter2 = new ListIterator (); // calls
     default constructor
    iter2.setList (jobsList); // iter2 too associated with
     jobsList

    Node cur = iter1.getCurrent (); // gets current node
    cur.displayNode ();
    iter1.moveToNextNode; // now iterator iter1 point to
     second node in jobsList
    ...
}
```

Linked list and iterators



The ListIterator class

```
class ListIterator {
    private Node current;        // current node
    private Node prev;           // previous node
    private LinkList iList;      // iterate through 'list'
    //--------------------------------------------
    public void reset () {  // reset current to iList's head
       and prev to  null
        current = iList.getFirst(); // LinkList method -
         returns a reference to head
        prev = null;
    }
    //--------------------------------------------
    public ListIterator () {...} // default constructor
    public ListIterator (LinkList list) {...} // constructor
    public void setList (LinkList list) {...} // assign list
       to iList
    public boolean isAtEnd () {...} // true if current is
     last node
    public void moveToNextNode () {...} // go to the next
     node
    public Node getCurrent () {...} // get current node
    public Node getPrevious () {...} // get previous node
    //--------------------------------------------
}
```

Lab Work 05 - ListIterator class

1. Add getFirst() method to LinkList class.
2. Complete the coding of the ListIterator class.
3. Write a class (ListIteratorApp) to use the methods defined in LinkList and ListIterator classes.
   - Create a linked list with any ten numbers.
   - Display all elements in the list.
   - Find the sum of odd numbers in the list.