Doubly Linked List

Axel Månson Lokrantz

Spring Fall 2022

Introduction

In the previous assignment we developed different benchmarks for singly linked lists, in this assignment a new type of list was introduced, namely a doubly linked list. The differences between the two are very minor, a singly linked list only has a forward pointer while the doubly linked list has both a forward and a previous pointer. The following program was written i Java, together with Pontus Kinnmark.

The Benchmark

The assignment consisted of one task which was to setup a benchmark where we could measure the time of different operations on a singly and doubly linked list and then compare the results. To get started, we built the overall structure for the lists, with nested node classes and useful methods to do simple testing such as display and get methods. The benchmark had directives to hold a list of n elements and then perform k remove and add operations.

```
public void addNode(int data){
    Node newNode = new Node(data);
    newNode.next = head;
    head = newNode;
}
```

Add operation for singly Linked List.

```
public void addNodeHead(Node ref){
    ref.next = head;
    ref.prev = null;
    head = ref;
}
```

Add operation for doubly linked list.

To not make it too complicated the add operation was only required to do simple insertion of the removed node at the top of the list. As seen above the two implementations are very similar. The real difference however was for the deletion of a node at a randomly given index.

```
public int deleteNodeAtIndex (int index) throws NullPointerException,
IndexOutOfBoundsException
 {
     if (head == null){
         throw new NullPointerException("Invalid input.");
     }
     Node temp = head;
     if (index == 0){
         int value = head.getData();
         head = temp.next;
         return value;
     }

     for(int i = 0; temp != null && i < index - 1; i++){
         temp = temp.next;
     }
     int value = temp.next.getData();

     if (temp == null || temp.next == null)
         throw new IndexOutOfBoundsException("The specified index does not exists

     Node node = temp.next.next;
     temp.next = node;
     return value;
     }
 }
```

deleteNodeAtIndex singly linked list.

To break it down the code stores our head in a new temp node. If either the head is null or the position of our pointer is more than the number of nodes we return. We check for the corner case where our given index is 0 and if so, simply set our head to temp.next. To find the the previous node of the node to deleted we traverse the list. Once there we create a new node which is the node to be deleted and store the pointer to the next of node to be deleted.

```
public void deleteNode(Node ref){

        if (head == null || ref == null) {
            return;
        }
        if (head == ref) {
            head = ref.next;
        }
        if (ref.next != null) {
            ref.next.prev = ref.prev;
        }
        if (ref.prev != null) {
            ref.prev.next = ref.next;
        }
        return;
    }
```

deleteNode doubly linked list.

Deletion of a node in a doubly list is much more convenient. Through the parameter we receive a reference node with the address to the node we want to delete. In case either the head or the reference node is null we return. If head equals the reference node we simply set head to reference.next. For the last two if statements we only change the previous and next pointer if the node to be deleted is not the last or the first node of the list. Lastly we free the memory occupied by our reference node.

n	singly	doubly
32	300	120
64	500	120
128	900	110
256	1750	100
512	3600	110
1024	7500	100
2048	15000	120
4096	30500	130
8192	65000	130
16384	15000	150

Table 1: Benchmark for adding and deleting a node k number of times on a singly and doubly list of length n.

.

The above benchmark was done for deleting a node at a random in-

dex and inserting it at the beginning of the list. Our k, the number of remove/add operations, was set to 10. The benchmark measures the time it takes for the whole sequence to complete. The singly benchmark uses a random index position, while the doubly benchmark uses a random reference node from a Node[ ] list.

Conclusion

From the results in table 1, we can conclude that for a singly linked list k amount of remove/add operations, on a list of n length has the time complexity of $O(n)$. This is because the singly linked list does not have a previous pointer, meaning in order to find a node at a given index we have to traverse the list up until that node to delete it. As for the doubly linked list, all we need is a reference to the node that we want to delete. With that reference, we know exactly where that node is and can easily remove the node by changing our previous and next pointers accordingly. And since the add operation, for both of the lists is constant, we get a time complexity of $O(1)$. So why ever use a singly linked list? A singly linked list occupies less memory than the doubly linked list since it only has two fields, while the doubly linked list has three fields, data and previous / next link. A singly linked list is therefor preferred when there is a memory limitation and searching is not required and a doubly linked list is preferred when there is not a memory limitation and searching is required.

A Singly Linked List occupies less memory than the Doubly Linked List, as it has only 2 fields.