Chapter 2

Linked Lists

Linked lists can be thought of from a high level perspective as being a series of nodes. Each node has at least a single pointer to the next node, and in the last node's case a null pointer representing that there are no more nodes in the linked list.

In DSA our implementations of linked lists always maintain head and tail pointers so that insertion at either the head or tail of the list is a constant time operation. Random insertion is excluded from this and will be a linear operation. As such, linked lists in DSA have the following characteristics:

1. Insertion is $O(1)$
2. Deletion is $O(n)$
3. Searching is $O(n)$

Out of the three operations the one that stands out is that of insertion. In DSA we chose to always maintain pointers (or more aptly references) to the node(s) at the head and tail of the linked list and so performing a traditional insertion to either the front or back of the linked list is an $O(1)$ operation. An exception to this rule is performing an insertion before a node that is neither the head nor tail in a singly linked list. When the node we are inserting before is somewhere in the middle of the linked list (known as random insertion) the complexity is $O(n)$. In order to add before the designated node we need to traverse the linked list to find that node's current predecessor. This traversal yields an $O(n)$ run time.

This data structure is trivial, but linked lists have a few key points which at times make them very attractive:

1. the list is dynamically resized, thus it incurs no copy penalty like an array or vector would eventually incur; and
2. insertion is $O(1)$.

2.1 Singly Linked List

Singly linked lists are one of the most primitive data structures you will find in this book. Each node that makes up a singly linked list consists of a value, and a reference to the next node (if any) in the list.

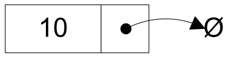

Figure 2.1: Singly linked list node

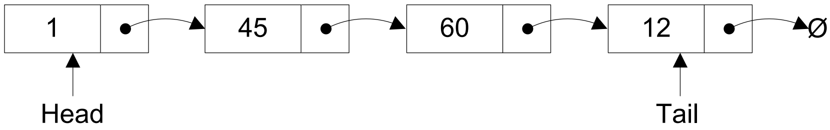

Figure 2.2: A singly linked list populated with integers

2.1.1 Insertion

In general when people talk about insertion with respect to linked lists of any form they implicitly refer to the adding of a node to the tail of the list. When you use an API like that of DSA and you see a general purpose method that adds a node to the list, you can assume that you are adding the node to the tail of the list not the head.

Adding a node to a singly linked list has only two cases:

1. $head = \emptyset$ in which case the node we are adding is now both the *head* and *tail* of the list; or

2. we simply need to append our node onto the end of the list updating the *tail* reference appropriately.

algorithm Add($value$)
    Pre: $value$ is the value to add to the list
    Post: $value$ has been placed at the tail of the list
    $n \leftarrow$ node($value$)
    if $head = \emptyset$
        $head \leftarrow n$
        $tail \leftarrow n$
    else
        $tail$.Next $\leftarrow n$
        $tail \leftarrow n$
    end if
end Add

As an example of the previous algorithm consider adding the following sequence of integers to the list: 1, 45, 60, and 12, the resulting list is that of Figure 2.2.

2.1.2 Searching

Searching a linked list is straightforward: we simply traverse the list checking the value we are looking for with the value of each node in the linked list. The algorithm listed in this section is very similar to that used for traversal in §2.1.4.

algorithm Contains(*head*, *value*)
    Pre: *head* is the head node in the list
        value is the value to search for
    Post: the item is either in the linked list, true; otherwise false
    $n \leftarrow head$
    while $n \neq \emptyset$ **and** n.Value $\neq value$
        $n \leftarrow n$.Next
    end while
    if $n = \emptyset$
        return false
    end if
    return true
end Contains

2.1.3 Deletion

Deleting a node from a linked list is straightforward but there are a few cases we need to account for:

1. the list is empty; or
2. the node to remove is the only node in the linked list; or
3. we are removing the head node; or
4. we are removing the tail node; or
5. the node to remove is somewhere in between the head and tail; or
6. the item to remove doesn't exist in the linked list

The algorithm whose cases we have described will remove a node from anywhere within a list irrespective of whether the node is the *head* etc. If you know that items will only ever be removed from the *head* or *tail* of the list then you can create much more concise algorithms. In the case of always removing from the front of the linked list deletion becomes an $O(1)$ operation.

```
1) algorithm Remove(head, value)
2)     Pre: head is the head node in the list
3)          value is the value to remove from the list
4)     Post: value is removed from the list, true; otherwise false
5)     if head = ∅
6)        // case 1
7)        return false
8)     end if
9)     n ← head
10)    if n.Value = value
11)       if head = tail
12)          // case 2
13)          head ← ∅
14)          tail ← ∅
15)       else
16)          // case 3
17)          head ← head.Next
18)       end if
19)       return true
20)    end if
21)    while n.Next ≠ ∅ and n.Next.Value ≠ value
22)       n ← n.Next
23)    end while
24)    if n.Next ≠ ∅
25)       if n.Next = tail
26)          // case 4
27)          tail ← n
28)       end if
29)       // this is only case 5 if the conditional on line 25 was false
30)       n.Next ← n.Next.Next
31)       return true
32)    end if
33)    // case 6
34)    return false
35) end Remove
```

2.1.4 Traversing the list

Traversing a singly linked list is the same as that of traversing a doubly linked list (defined in §2.2). You start at the head of the list and continue until you come across a node that is ∅. The two cases are as follows:

1. *node* = ∅, we have exhausted all nodes in the linked list; or
2. we must update the *node* reference to be *node*.Next.

The algorithm described is a very simple one that makes use of a simple *while* loop to check the first case.

```
algorithm Traverse(head)
    Pre: head is the head node in the list
    Post: the items in the list have been traversed
    n ← head
    while n ≠ 0
        yield n.Value
        n ← n.Next
    end while
end Traverse
```

2.1.5 Traversing the list in reverse order

Traversing a singly linked list in a forward manner (i.e. left to right) is simple as demonstrated in §2.1.4. However, what if we wanted to traverse the nodes in the linked list in reverse order for some reason? The algorithm to perform such a traversal is very simple, and just like demonstrated in §2.1.3 we will need to acquire a reference to the predecessor of a node, even though the fundamental characteristics of the nodes that make up a singly linked list make this an expensive operation. For each node, finding its predecessor is an $O(n)$ operation, so over the course of traversing the whole list backwards the cost becomes $O(n^2)$.

Figure 2.3 depicts the following algorithm being applied to a linked list with the integers 5, 10, 1, and 40.

```
algorithm ReverseTraversal(head, tail)
    Pre: head and tail belong to the same list
    Post: the items in the list have been traversed in reverse order
    if tail ≠ ∅
      curr ← tail
      while curr ≠ head
          prev ← head
          while prev.Next ≠ curr
              prev ← prev.Next
          end while
          yield curr.Value
          curr ← prev
      end while
      yield curr.Value
    end if
end ReverseTraversal
```

This algorithm is only of real interest when we are using singly linked lists, as you will soon see that doubly linked lists (defined in §2.2) make reverse list traversal simple and efficient, as shown in §2.2.3.

2.2 Doubly Linked List

Doubly linked lists are very similar to singly linked lists. The only difference is that each node has a reference to both the next and previous nodes in the list.