Linked List

mardi 25 juin 2024 9:43 AM

SINGLY LINKED LIST

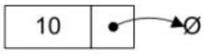
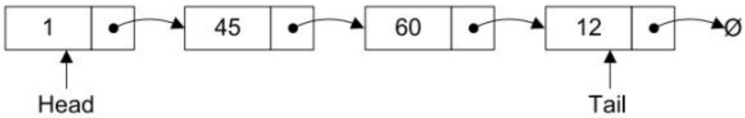


Figure 2.1: Singly linked list node



Insertion :

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 ← node(value)
    if head = ∅
        head ← n
        tail ← n
    else
        tail.Next ← n
        tail ← 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.

Searching

```
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 ← head
    while n ≠ ∅ and n.Value ≠ value
        n ← n.Next
    end while
    if n = ∅
        return false
    end if
    return true
end Contains
```

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

```
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
```

Traversing the List :

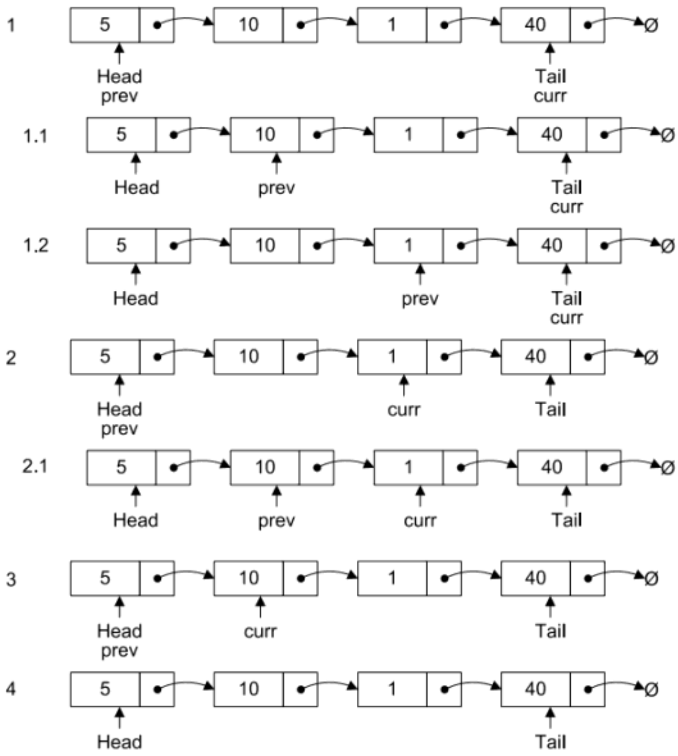
1. $node = \emptyset$, we have exhausted all nodes in the linked list; or
2. we must update the $node$ reference to be $node$.Next.

```
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
```

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 $O(n^2)$.

```
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
```



Doubly Linked List

Figure 2.4: Doubly linked list node

Insertion

The only major difference between the algorithm in §2.1.1 is that we need to remember to bind the previous pointer of n to the previous tail node if n was not the first node to be inserted into the list.

```
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 ← node(value)
    if head = ∅
        head ← n
        tail ← n
    else
        n.Previous ← tail
        tail.Next ← n
        tail ← n
    end if
end Add
```

Figure 2.5 shows the doubly linked list after adding the sequence of integers defined in §2.1.1.

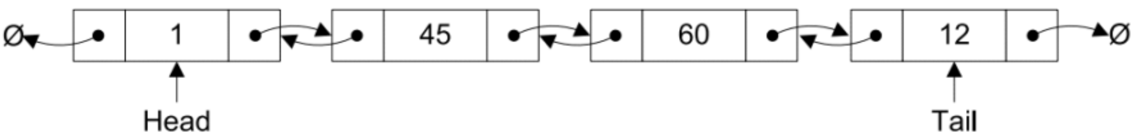


Figure 2.5: Doubly linked list populated with integers

DELETION

```
algorithm Remove(head, value)
    Pre: head is the head node in the list
         value is the value to remove from the list
    Post: value is removed from the list, true; otherwise false
    if head = ∅
      return false
    end if
    if value = head.Value
      if head = tail
        head ← ∅
        tail ← ∅
      else
        head ← head.Next
        head.Previous ← ∅
      end if
      return true
    end if
    n ← head.Next
    while n ≠ ∅ and value ≠ n.Value
      n ← n.Next
    end while
    if n = tail
      tail ← tail.Previous
      tail.Next ← ∅
      return true
    else if n ≠ ∅
      n.Previous.Next ← n.Next
      n.Next.Previous ← n.Previous
      return true
    end if
    return false
end Remove
```

Reverse Traversal

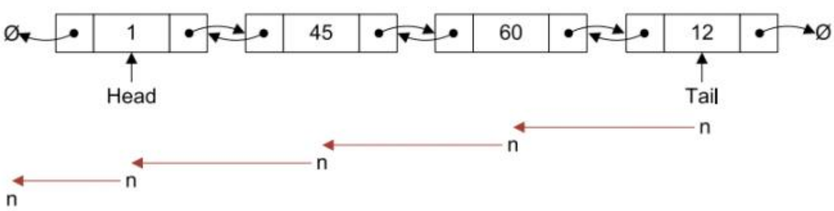


Figure 2.6: Doubly linked list reverse traversal

```
algorithm ReverseTraversal(tail)
    Pre: tail is the tail node of the list to traverse
    Post: the list has been traversed in reverse order
    n ← tail
    while n ≠ ∅
        yield n.Value
        n ← n.Previous
    end while
end ReverseTraversal
```

SUMMARY

Linked lists are good to use when you have an unknown number of items to store. Using a data structure like an array would require you to specify the size up front; exceeding that size involves invoking a resizing algorithm which has a linear run time. You should also use linked lists when you will only remove nodes at either the head or tail of the list to maintain a constant run time. This requires maintaining pointers to the nodes at the head and tail of the list but the memory overhead will pay for itself if this is an operation you will be performing many times.

What linked lists are not very good for is random insertion, accessing nodes by index, and searching. At the expense of a little memory (in most cases 4 bytes would suffice), and a few more read/writes you could maintain a *count* variable that tracks how many items are contained in the list so that accessing such a primitive property is a constant operation - you just need to update *count* during the insertion and deletion algorithms.

Singly linked lists should be used when you are only performing basic insertions. In general doubly linked lists are more accommodating for non-trivial operations on a linked list.

We recommend the use of a doubly linked list when you require forwards and backwards traversal. For the most cases this requirement is present. For example, consider a token stream that you want to parse in a recursive descent fashion. Sometimes you will have to backtrack in order to create the correct parse tree. In this scenario a doubly linked list is best as its design makes bi-directional traversal much simpler and quicker than that of a singly linked

.