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. 9 CHAPTER 2. LINKED LISTS 10 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 = ∅ 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. 1) algorithm Add(value) 2) Pre: value is the value to add to the list 3) Post: value has been placed at the tail of the list 4) n ← node(value) 5) if head = ∅ 6) head ← n 7) tail ← n 8) else 9) tail.Next ← n 10) tail ← n 11) end if 12) 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. CHAPTER 2. LINKED LISTS 11 1) algorithm Contains(head, value) 2) Pre: head is the head node in the list 3) value is the value to search for 4) Post: the item is either in the linked list, true; otherwise false 5) n ← head 6) while n 6= ∅ and n.Value 6= value 7) n ← n.Next 8) end while 9) if n = ∅ 10) return false 11) end if 12) return true 13) 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. CHAPTER 2. LINKED LISTS 12 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 6= ∅ and n.Next.Value 6= value 22) n ← n.Next 23) end while 24) if n.Next 6= ∅ 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 f alse 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. CHAPTER 2. LINKED LISTS 13 1) algorithm Traverse(head) 2) Pre: head is the head node in the list 3) Post: the items in the list have been traversed 4) n ← head 5) while n 6= 0 6) yield n.Value 7) n ← n.Next 8) end while 9) 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. 1) algorithm ReverseTraversal(head, tail) 2) Pre: head and tail belong to the same list 3) Post: the items in the list have been traversed in reverse order 4) if tail 6= ∅ 5) curr ← tail 6) while curr 6= head 7) prev ← head 8) while prev.Next 6= curr 9) prev ← prev.Next 10) end while 11) yield curr.Value 12) curr ← prev 13) end while 14) yield curr.Value 15) end if 16) 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. CHAPTER 2. LINKED LISTS 14 Figure 2.3: Reverse traveral of a singly linked list Figure 2.4: Doubly linked list node CHAPTER 2. LINKED LISTS 15 The following algorithms for the doubly linked list are exactly the same as those listed previously for the singly linked list: 1. Searching (defined in §2.1.2) 2. Traversal (defined in §2.1.4) 2.2.1 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. 1) algorithm Add(value) 2) Pre: value is the value to add to the list 3) Post: value has been placed at the tail of the list 4) n ← node(value) 5) if head = ∅ 6) head ← n 7) tail ← n 8) else 9) n.Previous ← tail 10) tail.Next ← n 11) tail ← n 12) end if 13) 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 2.2.2 Deletion As you may of guessed the cases that we use for deletion in a doubly linked list are exactly the same as those defined in §2.1.3. Like insertion we have the added task of binding an additional reference (P revious) to the correct value. CHAPTER 2. LINKED LISTS 16 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) return false 7) end if 8) if value = head.Value 9) if head = tail 10) head ← ∅ 11) tail ← ∅ 12) else 13) head ← head.Next 14) head.Previous ← ∅ 15) end if 16) return true 17) end if 18) n ← head.Next 19) while n 6= ∅ and value 6= n.Value 20) n ← n.Next 21) end while 22) if n = tail 23) tail ← tail.Previous 24) tail.Next ← ∅ 25) return true 26) else if n 6= ∅ 27) n.Previous.Next ← n.Next 28) n.Next.Previous ← n.Previous 29) return true 30) end if 31) return false 32) end Remove 2.2.3 Reverse Traversal Singly linked lists have a forward only design, which is why the reverse traversal algorithm defined in §2.1.5 required some creative invention. Doubly linked lists make reverse traversal as simple as forward traversal (defined in §2.1.4) except that we start at the tail node and update the pointers in the opposite direction. Figure 2.6 shows the reverse traversal algorithm in action. CHAPTER 2. LINKED LISTS 17 Figure 2.6: Doubly linked list reverse traversal 1) algorithm ReverseTraversal(tail) 2) Pre: tail is the tail node of the list to traverse 3) Post: the list has been traversed in reverse order 4) n ← tail 5) while n 6= ∅ 6) yield n.Value 7) n ← n.Previous 8) end while 9) end ReverseTraversal