جامعة ديالى \كلية التربية الاساسية
قسم الحاسبات

Data Structures

اعداد : م.م ياسر علي

المحاضرة الاولى

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 ¯nd 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 ¯nd 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.

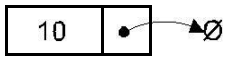

Figure 2.1: Singly linked list node

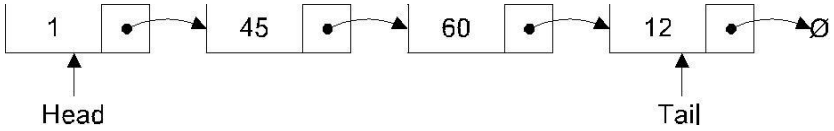

Figure 2.2: A singly linked list populated with integers

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 = ;* 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 Ã node(value)
5)     if head = ;
6)         head Ã n
7)         tail Ã n
8)     else
9)         tail.Next Ã n
10)        tail Ã 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 *x*2.1.4.

```
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 Ã head
6)     while n 6= ; and n.Value 6= value
7)         n Ã 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 any-where 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 Ã head
10)    if n.Value = value
11)       if head = tail
12)          // case 2
13)          head Ã ;
14)          tail Ã ;
15)       else
16)          // case 3
17)          head Ã head.Next
18)       end if
19)       return true
20)    end if
21)    while n.Next 6= ; and n.Next.Value 6= value
22)       n Ã n.Next
23)    end while
24)    if n.Next 6= ;
25)       if n.Next = tail
26)          // case 4
27)          tail Ã n
28)       end if
29)       // this is only case 5 if the conditional on line 25 was f alse
30)       n.Next Ã 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 (de¯ned in *x*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 ¯rst case.

```
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 Ã head
5)     while n =6 0
6)         yield n.Value
7)         n Ã 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 *x*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 *x*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, ¯nding 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 Ã tail
6)         while curr 6= head
7)             prev Ã head
8)             while prev.Next 6= curr
9)                 prev Ã prev.Next
10)            end while
11)            yield curr.Value
12)            curr Ã 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 (de¯ned in *x*2.2) make reverse list traversal simple and e±cient, as shown in *x*2.2.3.

2.2 Doubly Linked List

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

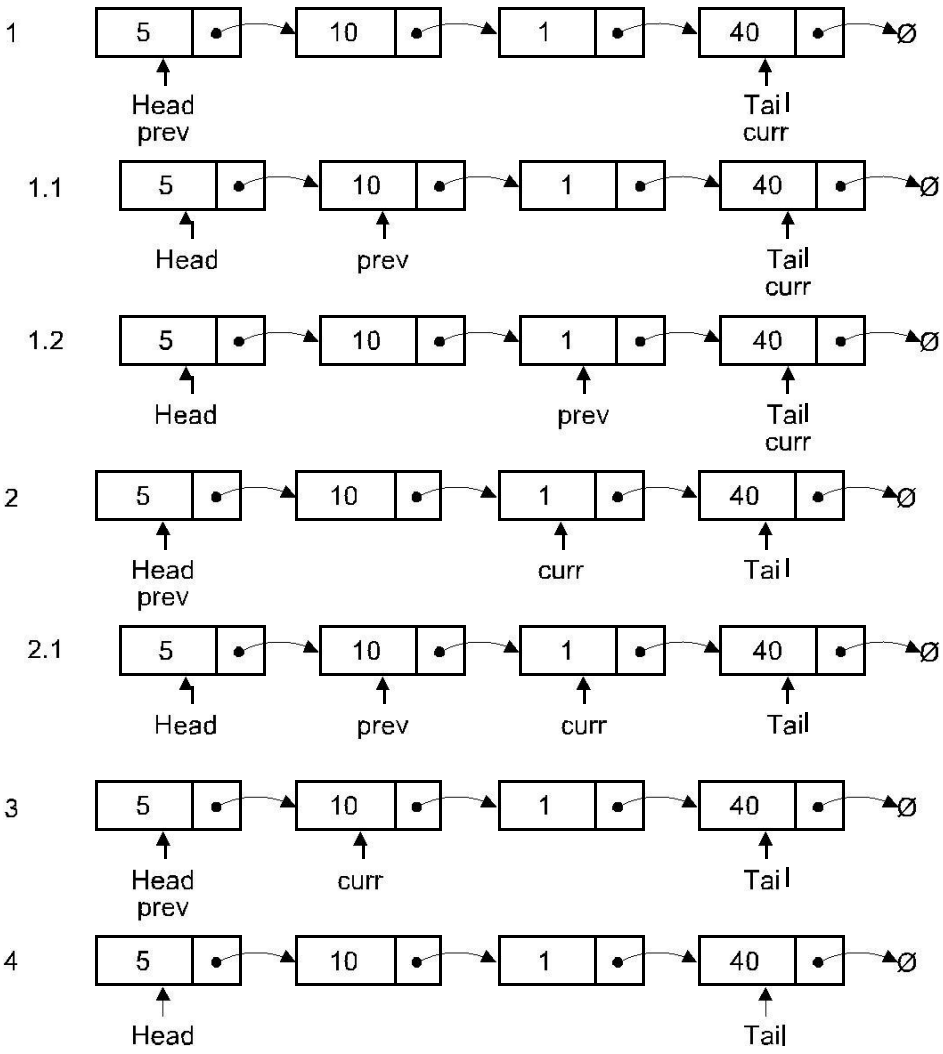

Figure 2.3: Reverse traveral of a singly linked list

Figure 2.4: Doubly linked list node

The following algorithms for the doubly linked list are exactly the same as those listed previously for the singly linked list:

1. Searching (de¯ned in *x*2.1.2)
2. Traversal (de¯ned in *x*2.1.4)

المحاضرة الثانية

2.2.1 Insertion

The only major di®erence between the algorithm in *x*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 ¯rst 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 Ã node(value)
5)     if head = ;
6)         head Ã n
7)         tail Ã n
8)     else
9)         n.Previous Ã tail
10)        tail.Next Ã n
11)        tail Ã n
12)    end if
13) end Add
```

Figure 2.5 shows the doubly linked list after adding the sequence of integers de¯ned in *x*2.1.1.

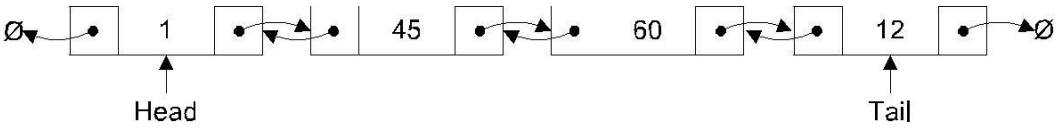

Figure 2.5: Doubly linked list populated with integers

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 de¯ned in *x*2.1.3. Like insertion we have the added task of binding an additional reference (*P revious*) to the correct value.

```
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 Ã ;
11)          tail Ã ;
12)       else
13)          head Ã head.Next
14)          head.Previous Ã ;
15)       end if
16)       return true
17)    end if
18)    n Ã head.Next
19)    while n 6= ; and value 6= n.Value
20)       n Ã n.Next
21)    end while
22)    if n = tail
23)       tail Ã tail.Previous
24)       tail.Next Ã ;
25)       return true
26)    else if n 6= ;
27)       n.Previous.Next Ã n.Next
28)       n.Next.Previous Ã 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 de¯ned in *x*2.1.5 required some creative invention. Doubly linked lists make reverse traversal as simple as forward traversal (de¯ned in *x*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.

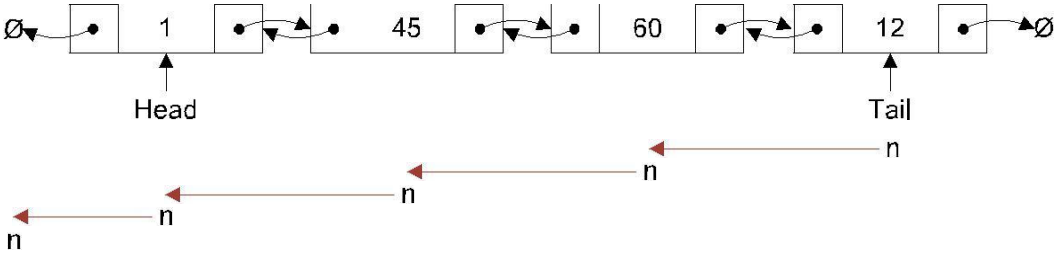

Figure 2.6: Doubly linked list reverse traversal

```
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 Ã tail
5)     while n =6 ;
6)         yield n.Value
7)         n Ã n.Previous
8)     end while
9) end ReverseTraversal
```

2.3 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 su±ce), 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

المحاضرة الثالثة

Chapter 3

Binary Search Tree

Binary search trees (BSTs) are very simple to understand. We start with a root node with value *x*, where the left subtree of *x* contains nodes with values < *x* and the right subtree contains nodes whose values are ¸ *x*. Each node follows the same rules with respect to nodes in their left and right subtrees.

BSTs are of interest because they have operations which are favourably fast: insertion, look up, and deletion can all be done in *O*(*log n*) time. It is important to note that the *O*(*log n*) times for these operations can only be attained if the BST is reasonably balanced; for a tree data structure with self balancing properties see AVL tree de¯ned in *x*7).

In the following examples you can assume, unless used as a parameter alias that *root* is a reference to the root node of the tree.

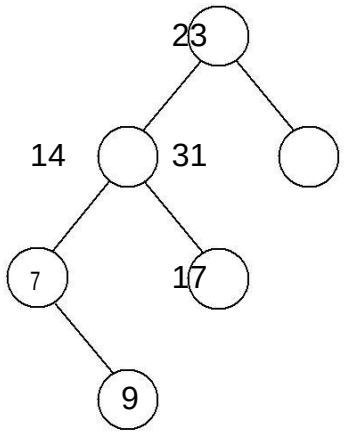

Figure 3.1: Simple unbalanced binary search tree

3.1 Insertion

As mentioned previously insertion is an *O*(*log n*) operation provided that the tree is moderately balanced.

```
1) algorithm Insert(value)
2)     Pre: value has passed custom type checks for type T
3)     Post: value has been placed in the correct location in the tree
4)     if root = ;
5)         root Ã node(value)
6)     else
7)         InsertNode(root, value)
8)     end if
9) end Insert
```

```
1) algorithm InsertNode(current, value)
2)     Pre: current is the node to start from
3)     Post: value has been placed in the correct location in the tree
4)     if value < current.Value
5)         if current.Left = ;
6)             current.Left Ã node(value)
7)         else
8)             InsertNode(current.Left, value)
9)         end if
10)    else
11)        if current.Right = ;
12)            current.Right Ã node(value)
13)        else
14)            InsertNode(current.Right, value)
15)        end if
16)    end if
17) end InsertNode
```

The insertion algorithm is split for a good reason. The ¯rst algorithm (non-recursive) checks a very core base case - whether or not the tree is empty. If the tree is empty then we simply create our root node and ¯nish. In all other cases we invoke the recursive *InsertN ode* algorithm which simply guides us to the ¯rst appropriate place in the tree to put *value*. Note that at each stage we perform a binary chop: we either choose to recurse into the left subtree or the right by comparing the new value with that of the current node. For any totally ordered type, no value can simultaneously satisfy the conditions to place it in both subtrees.

3.2 Searching

Searching a BST is even simpler than insertion. The pseudocode is self-explanatory but we will look brie°y at the premise of the algorithm nonetheless.

We have talked previously about insertion, we go either left or right with the right subtree containing values that are ¸ *x* where *x* is the value of the node we are inserting. When searching the rules are made a little more atomic and at any one time we have four cases to consider:

1. the *root* = *;* in which case *value* is not in the BST; or
2. *root*.Value = *value* in which case *value* is in the BST; or
3. *value* < *root*.Value, we must inspect the left subtree of *root* for *value*; or
4. *value* > *root*.Value, we must inspect the right subtree of *root* for *value*.

```
1) algorithm Contains(root, value)
2)     Pre: root is the root node of the tree, value is what we would like to locate
3)     Post: value is either located or not
4)     if root = ;
5)         return false
6)     end if
7)     if root.Value = value
8)         return true
9)     else if value < root.Value
10)        return Contains(root.Left, value)
11)    else
12)        return Contains(root.Right, value)
13)    end if
14) end Contains
```

3.3 Deletion

Removing a node from a BST is fairly straightforward, with four cases to consider:

1. the value to remove is a leaf node; or
2. the value to remove has a right subtree, but no left subtree; or
3. the value to remove has a left subtree, but no right subtree; or
4. the value to remove has both a left and right subtree in which case we promote the largest value in the left subtree.

There is also an implicit ¯fth case whereby the node to be removed is the only node in the tree. This case is already covered by the ¯rst, but should be noted as a possibility nonetheless.

Of course in a BST a value may occur more than once. In such a case the ¯rst occurrence of that value in the BST will be removed.

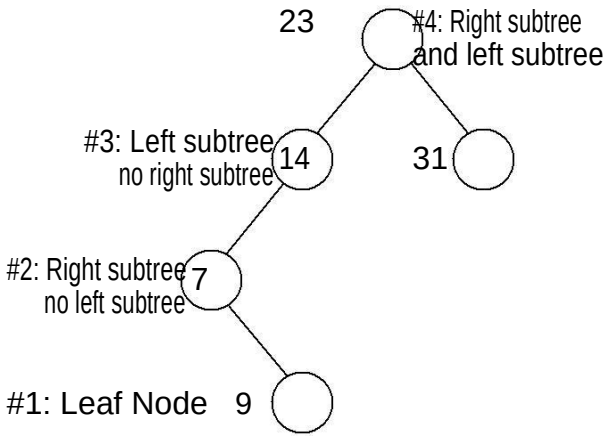

Figure 3.2: binary search tree deletion cases

The *Remove* algorithm given below relies on two further helper algorithms named *F indP arent*, and *F indN ode* which are described in *x*3.4 and *x*3.5 respectively.

```
1) algorithm Remove(value)
2)     Pre: value is the value of the node to remove, root is the root node of the BST
3)             Count is the number of items in the BST
3)     Post: node with value is removed if found in which case yields true, otherwise false
4)     nodeT oRemove Ã FindNode(value)
5)     if nodeT oRemove = ;
6)        return false // value not in BST
7)     end if
8)     parent Ã FindParent(value)
9)     if Count = 1
10)       root Ã ; // we are removing the only node in the BST
11)    else if nodeT oRemove.Left = ; and nodeT oRemove.Right = null
12)       // case #1
13)       if nodeT oRemove.Value < parent.Value
14)          parent.Left Ã ;
15)       else
16)          parent.Right Ã ;
17)       end if
18)    else if nodeT oRemove.Left = ; and nodeT oRemove.Right 6= ;
19)       // case # 2
20)       if nodeT oRemove.Value < parent.Value
21)          parent.Left Ã nodeT oRemove.Right
22)       else
23)          parent.Right Ã nodeT oRemove.Right
24)       end if
25)    else if nodeT oRemove.Left 6= ; and nodeT oRemove.Right = ;
26)       // case #3
27)       if nodeT oRemove.Value < parent.Value
28)          parent.Left Ã nodeT oRemove.Left
29)       else
30)          parent.Right Ã nodeT oRemove.Left
31)       end if
32)    else
33)       // case #4
34)       largestV alue Ã nodeT oRemove.Left
35)       while largestV alue.Right 6= ;
36)          // ¯nd the largest value in the left subtree of nodeT oRemove
37)          largestV alue Ã largestV alue.Right
38)       end while
39)       // set the parents' Right pointer of largestV alue to ;
40)       FindParent(largestV alue.Value).Right Ã ;
41)       nodeT oRemove.Value Ã largestV alue.Value
42)    end if
43)    Count Ã Count ¡1
44)    return true
45) end Remove
```

3.4 Finding the parent of a given node

The purpose of this algorithm is simple - to return a reference (or pointer) to the parent node of the one with the given value. We have found that such an algorithm is very useful, especially when performing extensive tree transforma-tions.

```
1) algorithm FindParent(value, root)
2)     Pre: value is the value of the node we want to ¯nd the parent of
3)          root is the root node of the BST and is ! = ;
4)     Post: a reference to the parent node of value if found; otherwise ;
5)     if value = root.Value
6)        return ;
7)     end if
8)     if value < root.Value
9)        if root.Left = ;
10)          return ;
11)       else if root.Left.Value = value
12)          return root
13)       else
14)          return FindParent(value, root.Left)
15)       end if
16)    else
17)       if root.Right = ;
18)          return ;
19)       else if root.Right.Value = value
20)          return root
21)       else
22)          return FindParent(value, root.Right)
23)       end if
24)    end if
25) end FindParent
```

A special case in the above algorithm is when the speci¯ed value does not exist in the BST, in which case we return ;. Callers to this algorithm must take account of this possibility unless they are already certain that a node with the speci¯ed value exists.

3.5 Attaining a reference to a node

This algorithm is very similar to *x*3.4, but instead of returning a reference to the parent of the node with the speci¯ed value, it returns a reference to the node itself. Again, ; is returned if the value isn't found.

```
1) algorithm FindNode(root, value)
2)     Pre: value is the value of the node we want to ¯nd the parent of
3)          root is the root node of the BST
4)     Post: a reference to the node of value if found; otherwise ;
5)     if root = ;
6)        return ;
7)     end if
8)     if root.Value = value
9)        return root
10)    else if value < root.Value
11)       return FindNode(root.Left, value)
12)    else
13)       return FindNode(root.Right, value)
14)    end if
15) end FindNode
```

Astute readers will have noticed that the *FindNode* algorithm is exactly the same as the *Contains* algorithm (de¯ned in *x*3.2) with the modi¯cation that we are returning a reference to a node not *true* or *f alse*. Given *FindNode*, the easiest way of implementing *Contains* is to call *FindNode* and compare the return value with ;.

المحاضرة الرابعة

3.6 Finding the smallest and largest values in the binary search tree

To ¯nd the smallest value in a BST you simply traverse the nodes in the left subtree of the BST always going left upon each encounter with a node, terminating when you ¯nd a node with no left subtree. The opposite is the case when ¯nding the largest value in the BST. Both algorithms are incredibly simple, and are listed simply for completeness.

The base case in both *F indM in*, and *F indM ax* algorithms is when the Left (*F indM in*), or Right (*F indM ax*) node references are *;* in which case we have reached the last node.

```
1) algorithm FindMin(root)
2)     Pre: root is the root node of the BST
3)             root =6 ;
4)     Post: the smallest value in the BST is located
5)     if root.Left = ;
6)        return root.Value
7)     end if
8)     FindMin(root.Left)
9) end FindMin
```

```
1) algorithm FindMax(root)
2)     Pre: root is the root node of the BST
3)          root =6 ;
4)     Post: the largest value in the BST is located
5)     if root.Right = ;
6)        return root.Value
7)     end if
8)     FindMax(root.Right)
9) end FindMax
```

3.7 Tree Traversals

There are various strategies which can be employed to traverse the items in a tree; the choice of strategy depends on which node visitation order you require. In this section we will touch on the traversals that DSA provides on all data structures that derive from *BinarySearchT ree*.

3.7.1 Preorder

When using the preorder algorithm, you visit the root ¯rst, then traverse the left subtree and ¯nally traverse the right subtree. An example of preorder traversal is shown in Figure 3.3.

```
1) algorithm Preorder(root)
2)     Pre: root is the root node of the BST
3)     Post: the nodes in the BST have been visited in preorder
4)     if root 6= ;
5)        yield root.Value
6)        Preorder(root.Left)
7)        Preorder(root.Right)
8)     end if
9) end Preorder
```

3.7.2 Postorder

This algorithm is very similar to that described in *x*3.7.1, however the value of the node is yielded after traversing both subtrees. An example of postorder traversal is shown in Figure 3.4.

```
1) algorithm Postorder(root)
2)     Pre: root is the root node of the BST
3)     Post: the nodes in the BST have been visited in postorder
4)     if root 6= ;
5)        Postorder(root.Left)
6)        Postorder(root.Right)
7)        yield root.Value
8)     end if
9) end Postorder
```

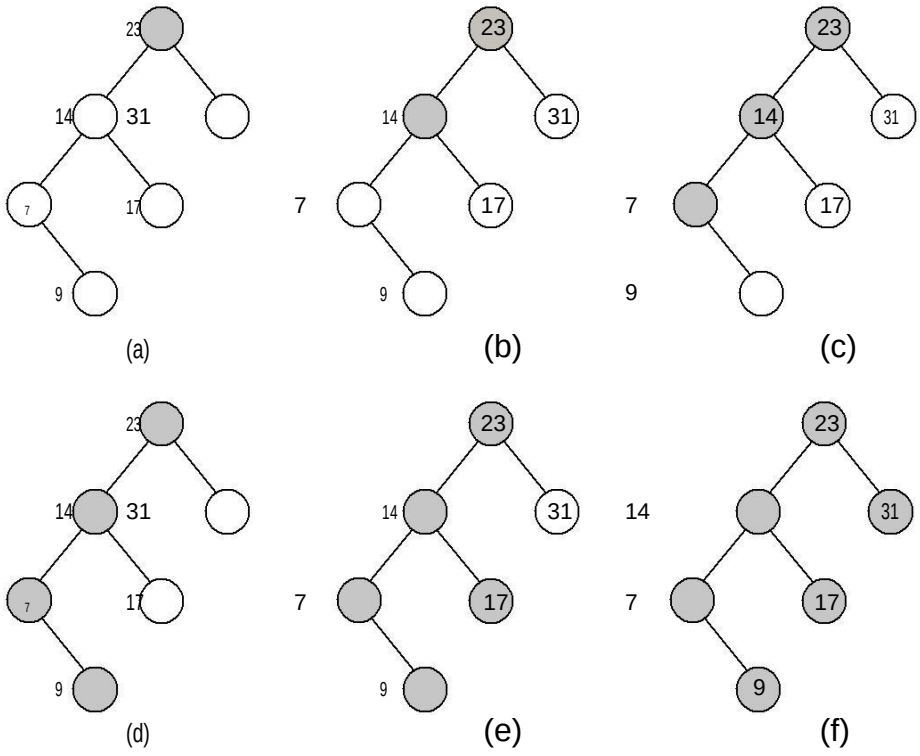

Figure 3.3: Preorder visit binary search tree example

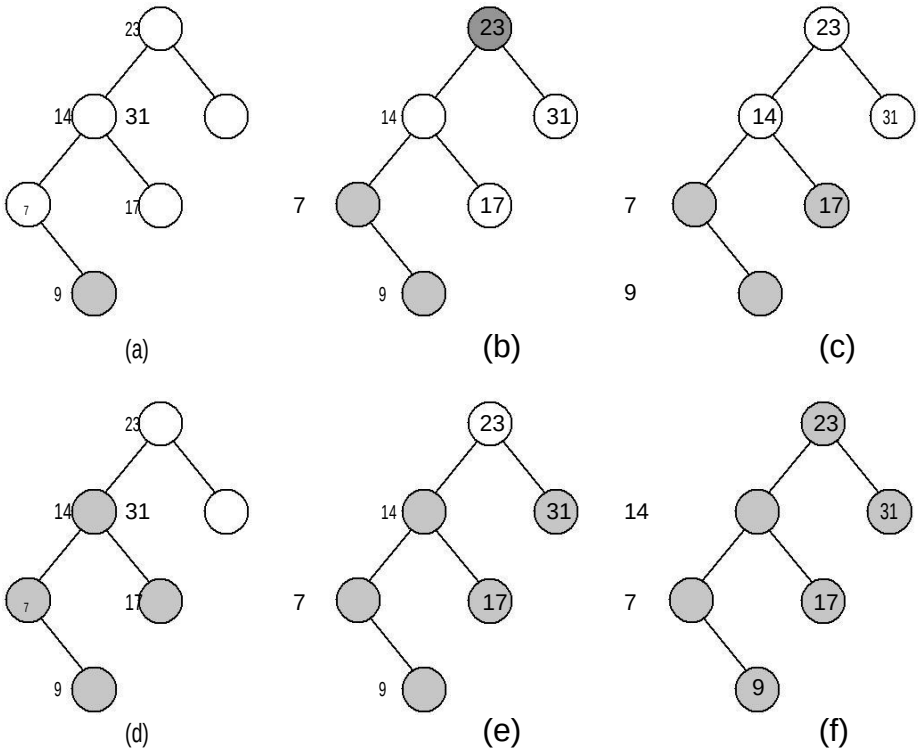

Figure 3.4: Postorder visit binary search tree example

3.7.3 Inorder

Another variation of the algorithms de¯ned in *x*3.7.1 and *x*3.7.2 is that of inorder traversal where the value of the current node is yielded in between traversing the left subtree and the right subtree. An example of inorder traversal is shown in Figure 3.5.

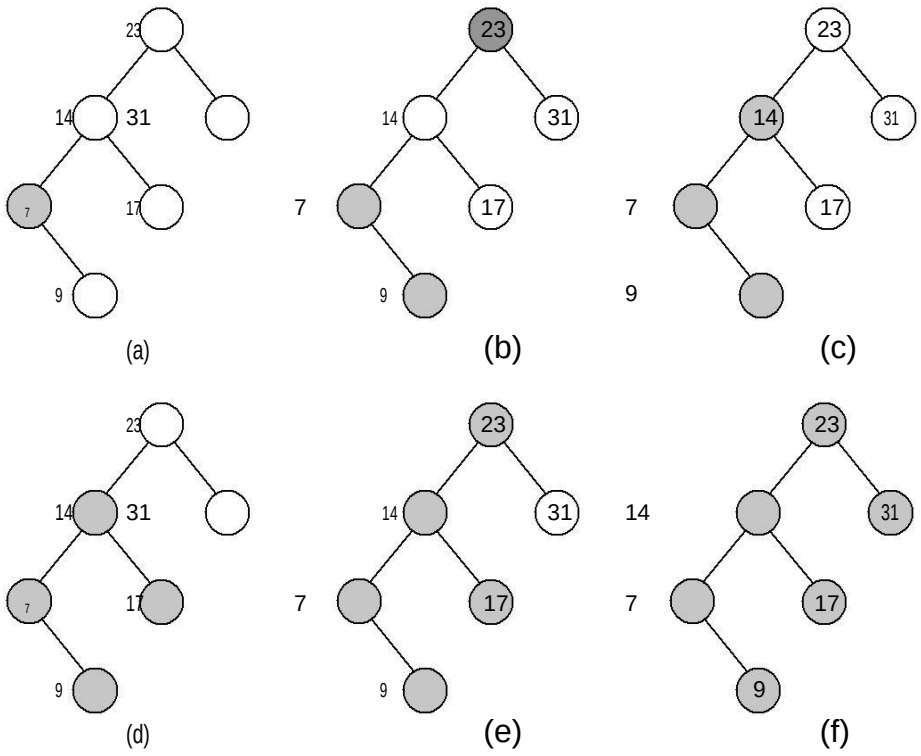

Figure 3.5: Inorder visit binary search tree example

```
1) algorithm Inorder(root)
2)     Pre: root is the root node of the BST
3)     Post: the nodes in the BST have been visited in inorder
4)     if root 6= ;
5)         Inorder(root.Left)
6)         yield root.Value
7)         Inorder(root.Right)
8)     end if
9) end Inorder
```

One of the beauties of inorder traversal is that values are yielded in their comparison order. In other words, when traversing a populated BST with the inorder strategy, the yielded sequence would have property $x_i \cdot x_{i+1} 8i$.

3.7.4 Breadth First

Traversing a tree in breadth ¯rst order yields the values of all nodes of a particular depth in the tree before any deeper ones. In other words, given a depth *d* we would visit the values of all nodes at *d* in a left to right fashion, then we would proceed to *d* + 1 and so on until we hade no more nodes to visit. An example of breadth ¯rst traversal is shown in Figure 3.6.

Traditionally breadth ¯rst traversal is implemented using a list (vector, resizeable array, etc) to store the values of the nodes visited in breadth ¯rst order and then a queue to store those nodes that have yet to be visited.

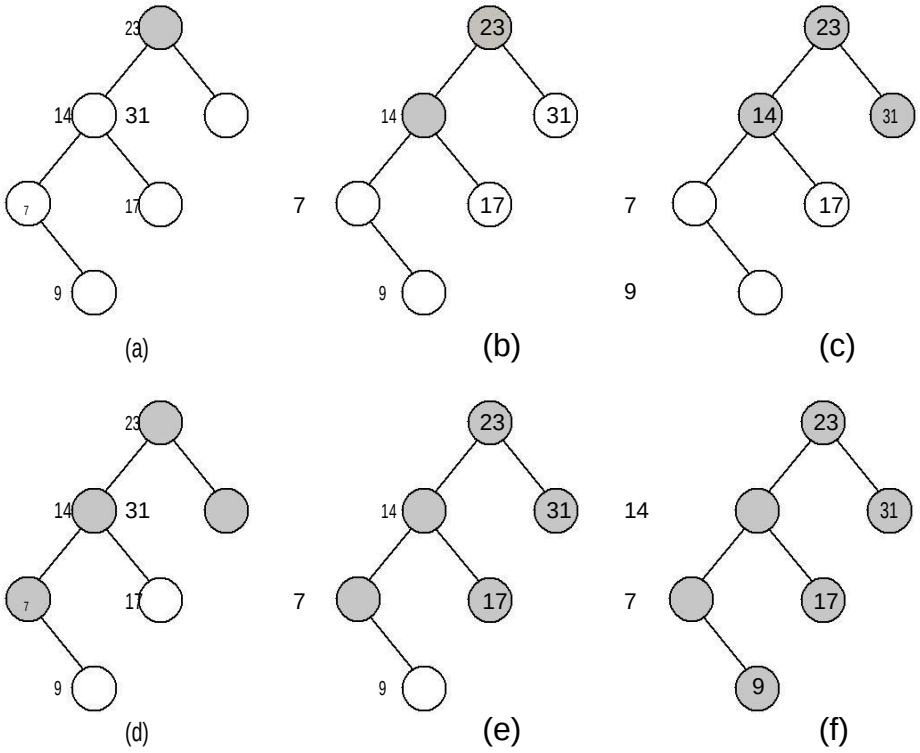

Figure 3.6: Breadth First visit binary search tree example

```
1) algorithm BreadthFirst(root)
2)     Pre: root is the root node of the BST
3)     Post: the nodes in the BST have been visited in breadth ¯rst order
4)     q Ã queue
5)     while root 6= ;
6)         yield root.Value
7)         if root.Left 6= ;
8)             q.Enqueue(root.Left)
9)         end if
10)        if root.Right 6= ;
11)            q.Enqueue(root.Right)
12)        end if
13)        if !q.IsEmpty()
14)            root Ã q.Dequeue()
15)        else
16)            root Ã ;
17)        end if
18)    end while
19) end BreadthFirst
```

3.8 Summary

A binary search tree is a good solution when you need to represent types that are ordered according to some custom rules inherent to that type. With logarithmic insertion, lookup, and deletion it is very e®ecient. Traversal remains linear, but there are many ways in which you can visit the nodes of a tree. Trees are recursive data structures, so typically you will ¯nd that many algorithms that operate on a tree are recursive.

The run times presented in this chapter are based on a pretty big assumption - that the binary search tree's left and right subtrees are reasonably balanced. We can only attain logarithmic run times for the algorithms presented earlier when this is true. A binary search tree does not enforce such a property, and the run times for these operations on a pathologically unbalanced tree become linear: such a tree is e®ectively just a linked list. Later in *x*7 we will examine an AVL tree that enforces self-balancing properties to help attain logarithmic run times.