

BERUFSAKADEMIE MANNHEIM
Information Technology International

Lecture Notes

Programming I
Part 3b Further Algorithms and Data Structures

Dipl.-Betriebswirt (BA) VOLKAN YAVUZ
{xmachina GmbH

November 2001–February 2002
Classes: TIM01AGR && TIT01EGR
Term: 1

Contents

1 Discussion of Primitive Operations on Arrays and Linked Lists	1
2 Trees in General and Binary Trees in Special	3
2.1 Implementation of Binary Trees	3
2.2 Searching and Inserting	5
2.3 Searching and Inserting Recursively	6
2.4 Traversal	7
2.5 Degenerated Binary Trees	7
2.6 Mathematical Properties of Binary Trees	8
3 Stack (Last In, First Out)	8
4 Queue (First In, First Out)	9
5 Further Algorithms and Data Structures	11
References	12

1 Discussion of Primitive Operations on Arrays and Linked Lists

A discussion of of the classical operations like *search*, *insert*, and *remove* performed on the already described data structures (arrays and linked lists) shall provide a motivation to create and use more sophisticated data structures:

searching In general, searching is done by iterating over all elements of either an array or a linked list, and comparing the element visited with the searched for element. If they are equal, the search can terminate, if they are not, the traversal is continued, either until the element is found, or the end of the data structure is reached and the search ends without success.

In the *worst case*, i. e. when the searched element is in the last position or not in the data structure at all, the whole data structure have to be traversed. This can be avoided if the data structure is sorted. In this case, two improved search strategies can be implemented:

- stopping, that means, as soon as a value greater then the searched value is found, the search operation can be terminated. This strategy works for both arrays and linked lists, as it is based upon a *sequential* traversal of the data structure.
- binary search is based upon a random access to elements of the data structure, so it can not be implemented on linked lists. Search starts by inspecting the element at the middle position. If that element is the searched one, search terminates. If that element is greater than the searched one, searching continues in the lower half of the array, otherwise in the upper half. The algorithm continues by accessing the middle of the remaining array, until the searched for element is found, or it is determined that the element is not contained within the array.

insert Inserting means either adding a new element at the beginning or end of the data structure, or inserting it between already occupied positions.

- Arrays: Inserting into an array in the middle of the array involves copying all already elements one position to the top of the array, to create space for the new element. If the array is not big enough to hold place for the newly added element, it has to be expanded. This might involve creating a new (empty) array with enough space and copying the *complete* array to the new one. The same holds true if the new element is to be added to the top of the array.
- Linked Lists: Inserting a new node at any place within the list just involves updating the internal pointer structures, which is not a very costly operation, but it may be a costly operation to determine the node after which to add the new element. This involves searching.

delete Deleting means removing an element either from the beginning or the end of the data structure, or removing an element from between already occupied positions.

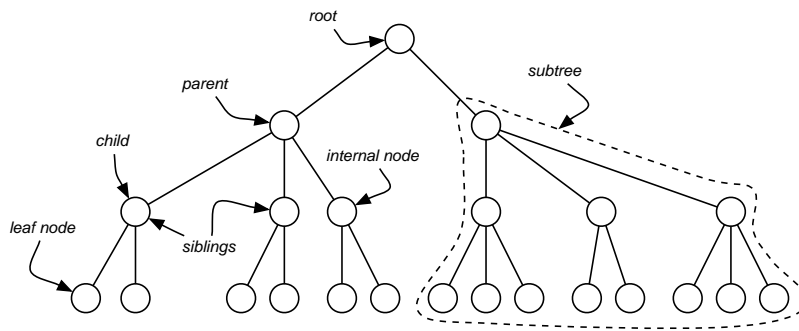
- Arrays: Removing involves copying the remaining elements one position to the bottom of the array, to fill the empty position caused by the removal of the element. It might also involve creating a new array with the reduced size, and copying all elements from the old array into the newly created one.
- Linked Lists: Removal of a node just involves updating the pointer structure at the immediately preceding node. The cost here lies again in finding the node to be removed, which is based on the search strategy implemented.

iteration Iteration means visiting, one by one, all elements of the data structure. Iterating linked lists is very easy, which involves just following the pointer structure. Iteration arrays is as easy, as it just involves maintaining a position counter which is to be incremented at each step.

random access Random Access means arbitrarily accessing any element within the data structure. As elements of an array can be accessed directly by means of an (integer) index, implementation is trivial. Accessing an arbitrary node of a linked list involves traversing all preceding nodes of that list.

sorting Sorting means manipulating the order in which elements are stored in the data structure in a way so that they are “in order” according to a given criterion. Differences are greatly due to the sorting algorithms implemented. Most sorting algorithm assume that random access to any element can be performed easily, so those sorting algorithms can only be performed sensibly on arrays. On the other hand, as inserting elements in a linked list is not a very costly operation, building sorted linked lists is easy.

Figure 1: Tree



min/max Finding extreme values, i. e. finding the smallest or largest value depends on iteration and on whether the elements are already sorted. There are no differences in iteration over either arrays or linked lists. If the data structure is known to be unsorted, the min/max operations also do not differ. If the data structure is known to be sorted, using arrays, the extreme values are to be found at the beginning and the end of the array, which can be accessed directly (random access by an integer index). Using linked lists, the first element can be accessed directly, whereas accessing the last element involves traversing the complete list.

2 Trees in General and Binary Trees in Special

Definition 1 A *tree* is a nonempty collection of *nodes* and *edges* that satisfies certain requirements. A *node* is a simple object that can have a name and/or carry other associated information. An *edge* is a connection between two nodes. A *path* in a tree is a list of distinct nodes in which successive nodes are connected by edges in the tree. The defining property of a tree is that there is precisely *one* path connecting any two nodes. If there is more than one path between some pair of nodes, or if there is no path between some pair of nodes, then we have a *graph*, but not a tree. A disjoint set of trees is called a *forest*.

A *rooted tree* is one where one node is designated as the *root* of the tree, and there is exactly one path between the root and each of the other nodes in the tree. Each node (except for the root) has exactly one node above it, which is called its *parent*; the nodes directly below a node are called its *children*. Nodes which have the same parent node are called *siblings* of each other.

The definitions were taken almost verbatim from [Sedgewick, 1998, pp. 216–229]. □

A typical example is show in figure 1.

Definition 2 A *binary tree* is a tree where each node has at most two child nodes, called the *left* and *right* subtree, respectively. □

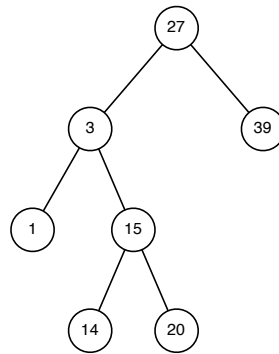
Definition 3 A *binary sort tree* is a binary tree, where the following conditions hold true:

- for every node p in the tree, the value stored in node p is greater than every value stored in nodes in the left subtree,
- and it is less than or equal to every value stored in nodes in the right subtree. □

Figure 2 on the next page shows a representation of the set 1, 3, 14, 15, 20, 27, 39.

2.1 Implementation of Binary Trees

Our implementation of binary trees is based upon the already used ‘Item’ class, of which a fragment is shown below:

Figure 2: A Binary Search Tree

```
1 public class Item
2 {
3     private int key;
4     private String value;
5
6     public Item(int key, String value)
7     {
8         this.key = key;
9         this.value = value;
10    }
11    ...
12 }
```

Binary trees consists of nodes, with references to the containing 'Item' instance, and referencing their left and right subtrees. Shown below is a fragemtn of that class 'Node', omitting the trivial *getter* and *setter* methods for the private member, 'item', 'left', and 'right', respectively.

```
13 public class Node
14 {
15     private Item item;
16     private Node left;
17     private Node right;
18
19     public Node(Item item)
20     {
21         this.item = item;
22         this.left = null;
23         this.right = null;
24     }
25     ...
26 }
```

Finally, the container implementation of a binary tree data structure is shown as class 'BinaryTree'. Except for the constructor and member declarations (which are trivial), all further methods will be discussed in the following sections:

```
27 public class BinaryTree
28 {
29     private Node root;
30
31     public BinaryTree()
32     {
33         root = null;
```

```
34     }
35
36     public BinaryTree(Node root)
37     {
38         this.root = root;
39     }
40     ...
41 }
```

2.2 Searching and Inserting

Given the properties of a binary search tree, searching can be performed using the same *binary search* strategy as is applicable on a sorted array. First, the looked for value is compared with the value stored in the root node. If the value is smaller, then search continues in the left subtree, otherwise in the right subtree. This is applied recursively until the value is found or it can be determined that it is not in the tree. The same strategy can be used to insert new items in sorted order into the tree. The implementation of an iterative algorithm is shown below:

```
42 public Node findKey(int key)
43 {
44     boolean found = false;
45     Node pos = root;
46
47     while ((pos != null) && !found) {
48         if (pos.getItem().getKey() == key) {
49             found = true;
50         } else {
51             if (key < pos.getItem().getKey())
52                 pos = pos.getLeft();
53             else
54                 pos = pos.getRight();
55         }
56     } // while
57
58     if (found)
59         return pos;
60     else
61         return null;
62 }
```

Inserting is based upon searching, since the correct subtree has to be determined prior to insertion of the new node. By definition, binary search trees contain each key only exactly once, so in the case that the value to be inserted already existst, nothing in done. A fragment of the implementation is shown below:

```
63 public void insert(Item item)
64 {
65     ...
66     if (!found) {
67         Node node = new Node(item);
68         node.setLeft(null);
69         node.setRight(null);
70
71         if (root == null) {
72             root = node;
73         } else {
74             if (item.getKey() < parentPos.getItem().getKey()) {
```

```
75         parentPos.setLeft(node);
76     } else {
77         parentPos.setRight(node);
78     }
79 }
80 }
81 }
```

2.3 Searching and Inserting Recursively

As binary trees are an inherently recursive data structure, recursive algorithms for search and insertion seem feasible. Below, a recursive search is shown. The algorithm can be described as follows, and the elegance of the implementation is striking.

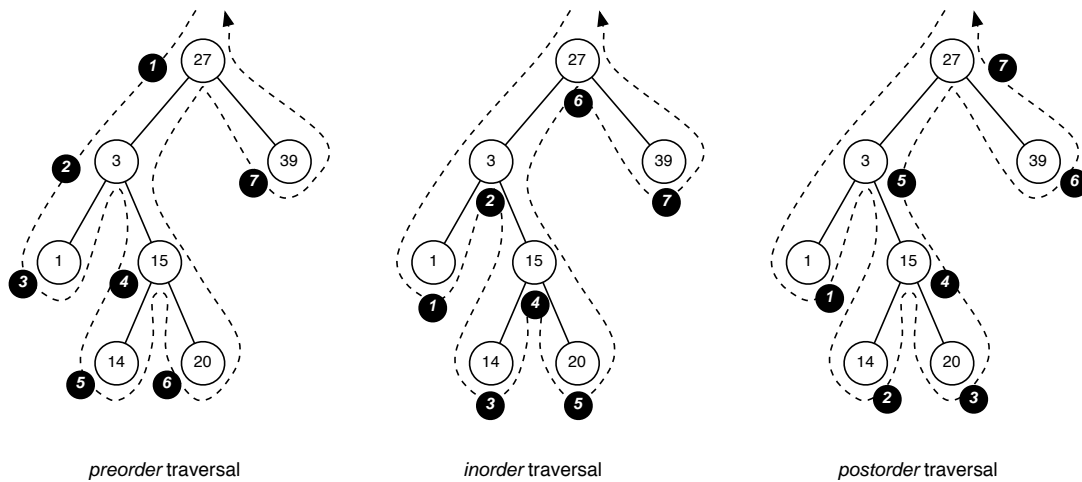
- If the key is smaller than the current root, consider the left subtree
- If the key is larger than the current root, consider the right subtree
- If the subtree is empty, the key is not contained in the tree
- Else, the node is found

```
82 private Node findKeyRecursively(int key, Node root)
83 {
84     Node pos = root;
85
86     if (root != null)
87         if (key < root.getItem().getKey())
88             pos = findKeyRecursively(key, root.getLeft());
89         else if (key > root.getItem().getKey())
90             pos = findKeyRecursively(key, root.getRight());
91
92     return pos;
93 }
```

Almost the same holds true for the insertion, which is, in principle, based on the search algorithm:

```
94 private Node insertRecursively(Item item, Node root)
95 {
96     if (root == null) {
97         Node node = new Node(item);
98         node.setLeft(null);
99         node.setRight(null);
100        return node;
101    } else {
102        Node node;
103        if (item.getKey() < root.getItem().getKey()) {
104            if ((node = insertRecursively(item, root.getLeft())) != null)
105                root.setLeft(node);
106        } else if (item.getKey() > root.getItem().getKey()) {
107            if ((node = insertRecursively(item, root.getRight())) != null)
108                root.setRight(node);
109        }
110        return node;
111    }
112 }
```

Figure 3: Tree Traversal



2.4 Traversal

A binary tree can be traversed in several ordered ways, which of all are show in figure 3.

preorder If the tree is empty, nothing needs to be done, else visit the root. Then, traverse the left subtree in preorder, and then traverse the right subtree in preorder.

Given the example, visiting the nodes yields the values 27, 3, 1, 15, 14, 20, 39.

inorder If the tree is empty, nothing needs to be done, else traverse the left subtree in inorder, then visit the root, and finally, traverse the right subtree in inorder.

Given the example, visiting the nodes yields the values 1, 3, 14, 15, 20, 27, 39.

postorder If the tree is empty, nothing needs to be done, else traverse the left subtree in inorder, then traverse the right subtree in inorder, and finally, visit the root.

Given the example, visiting the nodes yields the values 1, 14, 20, 15, 3, 39, 27.

Using recursion, implementation of tree traversal is almost trivial. An iterative solution would need a runtime stack managed by the programmer. This again shows the close relationship between algorithms and the data structures they operate on. Because trees are an inherently recursive data structure, recursive implementation of algorithms is very elegant. Shown below is the implementation of a pre order traversal.

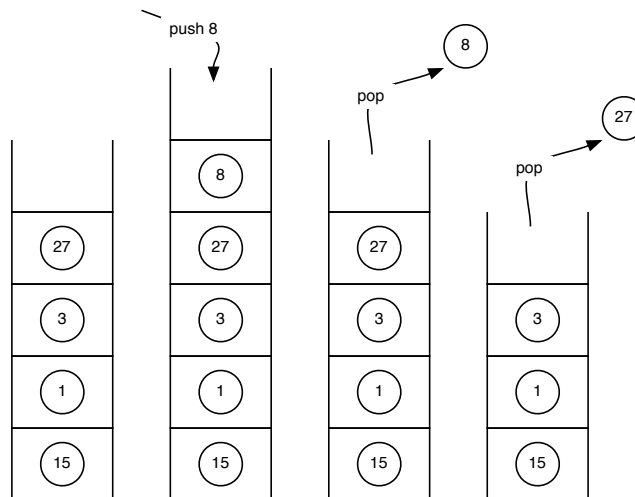
```

113 private void traversePreOrder(Node root)
114 {
115     if (root != null) {
116         System.out.println("pre order:" + root.getItem().toString());
117         traversePreOrder(root.getLeft());
118         traversePreOrder(root.getRight());
119     }
120 }
```

2.5 Degenerated Binary Trees

Unfortunately, binary trees can *degenerate*. A typical cause is building up a binary tree from an already ordered set of values. In that case, all left subtrees remain empty (or the right subtrees, if the set is sorted in reverse order), and the resulting tree becomes a linked list, with all it's properties.

Figure 4: Stack



To prevent this from happening, *balanced* binary trees are used, whose discussion is beyond the scope in this lecture.

2.6 Mathematical Properties of Binary Trees

- A binary tree with N internal nodes has $N + 1$ external nodes.
- A binary tree with N internal nodes has $2N$ edges: $N - 1$ edges to internal nodes and $N + 1$ edges to external nodes.

In the worst case (given a degenerated binary tree), a search in a tree with N keys can require N comparisons. On the average, search hits require about $2 \ln N$ comparisons. Compare this with a search miss in an unsorted array, which requires N comparisons, and a search hit in the average case, which requires $N/2$ comparisons.

3 Stack (Last In, First Out)

Definition 4 A *stack* is an ADT (Abstract Data Type) that comprises two basic operations: insert (*push*) a new item, and delete (*pop*) the item that was most recently inserted. \square

A *stack* is a simple but frequently needed data structure, that operates on the LIFO (Last-In, First-Out) principle. The sequence of one 'push' and two 'pop' operations and its effects onto the stack are shown in figure 4.

We will implement an ADT stack, which provides these basic operations. However, as underlying data structure, we will be using the 'DynamicArray' class developed earlier.

```

121 public class Stack
122 {
123     private DynamicArray storage;
124
125     public Stack() ...
126     public void push(Element element) ...
127     public Element pop() ...
128     public boolean isEmpty() ...
129 }
```

Given the functionality of 'DynamicArray', implementation is almost trivial:

```
130 public class Stack
131 {
132     private DynamicArray storage;
133
134     public Stack() {
135         storage = new DynamicArray();
136     }
137
138     public void push(Element element) {
139         storage.add(element)
140     }
141
142     public Element pop() {
143         return storage.remove();
144     }
145
146     public boolean isEmpty() {
147         return !(storage.hasMoreElements());
148     }
149 }
```

4 Queue (First In, First Out)

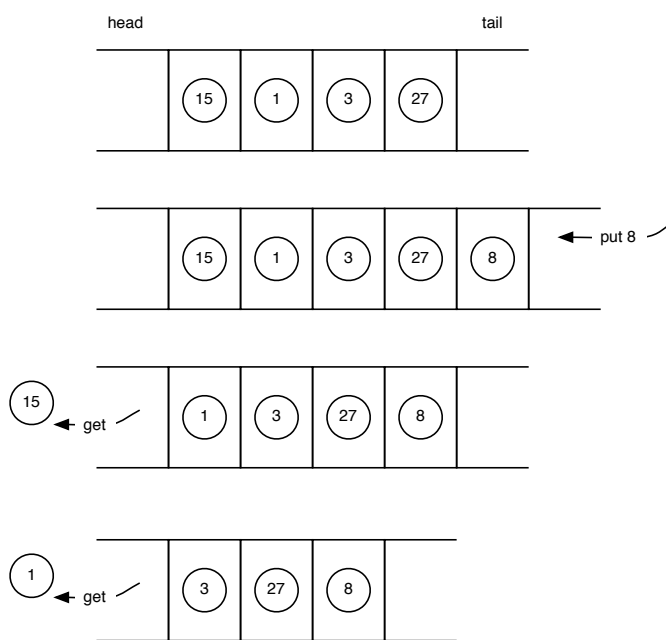
Definition 5 A FIFO (First-In, First-Out) *queue* is an ADT that comprises two basic operations: insert (*put*) a new item, and delete (*get*) the item that was least recently inserted. □

A *queue* is another simple but frequently used data structure, that operates on the FIFO principle. The sequence of one 'put' and two 'get' operations and its effects onto the queue are shown in figure 5 on the following page. Typical applications for queues are situations, where more than one item is eligible for processing, but they have to be processed one after the other, and not simultaneously.

We will implement an ADT queue, which provides these basic operations. However, as underlying data structure, we will be using the slightly modified 'LinkedList' class developed earlier.

```
150 public class Queue
151 {
152     private LinkedList storage;
153
154     public Queue() {
155         storage = new LinkedList();
156     }
157
158     public void put(Item item) {
159         storage.add(item);
160     }
161
162     public Item get() {
163         return storage.remove();
164     }
165
166     public boolean isEmpty() {
167         if (storage.getSize() == 0)
168             return true;
169         else
170             return false;
171     }
172 }
```

Figure 5: Queue



```
171     }
172 }
```

First, we have to complete the 'LinkedList' implementation, because a remove operation was missing. We also add a private member 'tail' to be able to add new nodes at the *end*, and remove them from the beginning (*head*).

```
173 class LinkedList
174 {
175     ...
176     Node tail;
177     int size;
178     ...
179     ...
180
181     public void add(Item item)
182     {
183         Node new_node = new Node(item);
184
185         if (tail == null) {
186             new_node.setNext(null);
187             head = new_node;
188             tail = new_node;
189         } else {
190             new_node.setNext(null);
191             tail.setNext(new_node);
192             tail = new_node;
193         }
194         ++size;
195     }
196
197     public Item remove()
```

```

198     {
199         if (tail != null) {
200             if (head == tail) {
201                 Item item = head.getItem();
202                 head.setNext(null);
203                 head = null;
204                 tail = null;
205                 --size;
206                 return item;
207             } else {
208                 Item item = head.getItem();
209                 Node old_head = head;
210                 head = old_head.getNext();
211                 old_head.setNext(null);
212                 --size;
213                 return item;
214             }
215         } else
216             return null;
217     }
218     ...
219 } // LinkedList
    
```

5 Further Algorithms and Data Structures

Further algorithms and data structures, that will not be discussed in this lecture include:

- Insertion Sort
- Shellsort
- Mergesort
- Heapsort
- Radix Sort
- doubly linked lists
- priority queues
- balanced trees
- n -ary trees
- graphs
- hash tables

List of Tables

List of Figures

1	Tree	3
2	A Binary Search Tree	4

3	Tree Traversal	7
4	Stack	8
5	Queue	10

List of Acronyms

- ADT Abstract Data Type
- FIFO First-In, First-Out
- LIFO Last-In, First-Out

A thorough treatment of the material presented here can be found in Sedgewick [1998].

References

Robert Sedgewick. *Algorithms in C*, volume 1. Addison-Wesley Publishing Company, Inc., AW:adr, 3rd edition, August 1998. ISBN 0-201-31452-5.