Berufsakademie Mannheim
Information Technology International

Lecture Notes

Programming I
Part 3 Algorithms and Data Structures

Dipl.-Betriebswirt (BA) Volkan Yavuz
{xmachina GmbH

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

Contents

1 The Array Datastructure **1**

2 Sorting **2**
2.1 Selection Sort . . . 2
2.2 Bubble Sort . . . 3
2.3 Quick Sort . . . 3

3 Elementary Data Structures **5**
3.1 Linked Lists . . . 5
3.1.1 Implementation . . . 6
3.1.2 Adding to Linked Lists . . . 8
3.1.3 Traversing Linked Lists . . . 8
3.1.4 Searching in Linked Lists . . . 8
3.1.5 Building Sorted Linked Lists . . . 9

References **10**

1 The Array Datastructure

Based upon an earlier example, we would like to implement an dynamically sized array datastructure called '`DynamicArray`', that holds elements of class '`Element`'. This data structure should implement the following operations:

- '`public int add(Element element)`' to add an element at the end
- '`public Element remove()`' to get the last element, and remove it
- '`public Element getElement(int pos)`' to get the element at the given position

- '`public void setElement(int pos, Element element)`' to get an element at the given position
- '`public boolean hasMoreElements()`' to check whether the data structure is empty or not
- '`public int getSize()`' to return the number of elements in the data structure

The contained *elements* do have the following interface:

- '`public Element(int key, String value)`' A constructor to create an element with the given attributes of '`key`' and '`value`'.
- '`public int getKey()`' a simple accessor method the read the '`key`' attribute.
- '`public String getValue()`' a simple accessor method the read the '`value`' attribute.
- '`public String toString()`' a simple function to return a string containing an elements attributes.

2 Sorting

Sorting algorithms have some properties:

performance Performance of sorting algorithms is dominated by

- the number of comparisons that have to be made, and
- how many times elements have to be exchanged.

memory needed Sorting algorithms may be able to sort

- *in place* (without the need to allocate additional storage locations), or
- need to allocate some extra memory for auxiliary data structures, or
- may even need to work on wholes copies of the data elements to perform the sorting.

stable, unstable A sorting method is said to be *stable* if it preserves the relative order of elements with duplicated keys in the file.

indirect sorting If the elements to be sorted are large, it might be resonable not to exchange the elements, but instead to rearrange an array of pointers to the elements to be sorted.

access Based upon *how* (random access, sequential, traversal) the elements to be sorted can be accessed, different sorting algorithms exist.

adaptive, nonadaptive A sorting algorithm is said to be *adaptive* if it performs different sequences of operations, depending on the comparisons performed.

2.1 Selection Sort

Selection Sort is one of the simplest sorting algorithms. First, find the smallest element in the array, and exchange it with the element in the first position. Then, find the second smallest element, and exchange it with the element in the second position. Continue this way until the entire array is sorted. It is called *selection* sort because it work by repeatedly selecting the smallest remaining element.

An implementation of *selection sort* is shown below. For each '`i`' from '`0`' to '`count - 2`', exchange '`element[i]`' with the minimum element in '`a[i], ..., a[count]`'. As the index '`i`' travels from left to right, the elements to its left are in their final position in the array (and will not be touched again), so the array is fully sorted when '`i`' reaches the right end.

```
1      public void selectionSort()
2      {
3          // loop over all elements
4          for (int i = 0; i < count - 1; i++) {
5              int min_pos = i;
6              // now look for a key smaller than the current one in the rest of elements
7              for (int j = i + 1; j < count; j++)
8                  if (elements[j].getKey() < elements[min_pos].getKey())
9                      min_pos = j;
10             exchangeElements(i, min_pos);
11         }
12     }
```

Table 1: Characteristics of Selection Sort

Comment	Description
Number of comparisons	$N^2/2$
Number of exchanges	N
Advantages	Simple. Algorithm that needs the least number of exchanges.
Disadvantages	Evidently brute-force. Its running time depends only slightly on the amount of order. Running time for an already sorted array is about the same as for a randomly ordered array.

2.2 Bubble Sort

Most people learn *bubble sort* as their first sorting algorithm because of its simplicity. Keep passing through the array, exchanging adjacent elements that are out of order, continuing until the file is sorted. Bubble sort's prime virtue is that it is easy to implement, but whether it is actually easier to implement than insertion or selection sort is arguable.

An implementation of *bubble sort* is shown below. For each '`i`' from '`0`' to '`count - 2`', the inner '`j`' loop puts the minimum element among the elements in '`a[i], ..., a[count-1]`' into '`a[i]`' by passing from right to left though the elements, compare-exchanging successive elements. The smallest one moves on all such comparisons, so it "bubble" to the beginning. As in selection sort, as the index '`i`' travels from left to right through the array, the elements to its left are in their final position in the array.

```
13     public void bubbleSort()
14     {
15         // loop over all elements
16         for (int i = 0; i < count - 1; i++) {
17             for (int j = count - 1; j > i; j--) {
18                 if (elements[j].getKey() < elements[j-1].getKey())
19                     exchangeElements(j-1, j);
20             }
21         }
22     }
```

2.3 Quick Sort

Quicksort is the sorting algorithm that is probably used more widely than any other one. The basic algorithm was invented in 1960 by C. A. R. Hoare, and it has been studied by many people since that time.

Table 2: Characteristics of Bubble Sort

Comment	Description
Number of comparisons	$N^2/2$
Number of exchanges	$N^2/2$
Advantages	Simple, both in understanding and in implementation.
Disadvantages	Evidently brute-force. Its running time depends only slightly on the amount of order. Running time for an already sorted array is about the same as for a randomly ordered array. Longer running time that *selection sort*.

Quicksort is a *divide-and-conquer* method for sorting. It works by *partitioning* an array into two parts, then sorting the parts independently. The partitioning process rearranges the array to make the following three conditions hold:

- The element'`a[i]`' is in its final place
- None of the elements in '`a[l],...a[i-1]`' is greater than '`a[i]`'
- None of the elements in '`a[i+1],...a[r]`' is less than '`a[i]`'

If the array has one or fewer elements, do nothing. Otherwise, the array is processed by a '`partition`' method, which puts '`a[i]`' into position for some '`i`' between '`l`' and '`r`' inclusive, and rearranges the other elements such that the recursive calls properly finish the sort.

```
23    private void quickSort(int l, int r)
24    {
25        int i;
26        if (r <= l)
27            return;
28        i = partition(l, r);
29        quickSort(l, i - 1);
30        quickSort(i + 1, r);
31    }
```

To implement partitioning, the following general strategy is used. First, we arbitrarily choose '`a[r]`' to be the *partitioning element* – the one that will go into its final position. Next, we scan from the left end of the array until we find an element greater than the partitioning element, and we scan from the right end of the array until we find an element less than the partitioning element. The two elements that stopped the scans are obviously out of place in the final partitioned array, so we exchange them. Continuing in this way, we ensure that no array elements to the left of the left pointer are greater than the partitioning element, and no array elements to the right of the right pointer are less than the partitioning element.

The variable '`v`' holds the value of the partitioning element '`a[r]`', and '`i`' and '`j`' are the left and right scan pointers, respectively. The partitioning loop increments '`i`' and descrements '`j`', while maintaining the invariant property that no elements to the left of '`i`' are greater than '`v`' and no elements to the right of '`j`' are smaller than '`v`'. Once the pointers meet, we complete the partitioning by exchanging '`a[i]`' and '`a[r]`', which puts '`v`' into '`a[i]`', with no larger elements to '`v`'s rioght and no smaller elements to its left. The partitioning loop is implemented as an infinite loop, with a '`break`' when the pointers cross. The test '`j == l`' protects against the case that the partitioning element is the smallest element in the file.

```
32    private int partition(int l, int r)
33    {
34        int i = l - 1;
```

```
35          int j = r;
36          Element v = elements[r];
37  
38          for (;;) {
39              while (elements[++i].getKey() < v.getKey());
40              while (v.getKey() < elements[--j].getKey())
41                  if (j == l)
42                      break;
43              if (i >= j)
44                  break;
45              exchangeElements(i, j);
46          }
47          // put the partitioning element into its final position
48          exchangeElements(i, r);
49          return i;
50      }
```

Table 3: Characteristics of Quick Sort

Comment	Description
Number of comparisons	
Number of exchanges	
Advantages	Not diffucult to implement, well known and studied. Works well for different kinds of input data. Consumes fewer resources than any other sorting method in many situations. It works inplace, only using a small auxiliary stack
Disadvantages	It is not stable. It is fragile in the sense that a simple mistake in the implementation can go unnoticed and can cause it to perform badly for some files.

3 Elementary Data Structures

"Organizing the data for processing is an essential step in the development of a computer program. For many applications, the choice of the proper data structure is the only major decision involved in the implementation: once the choice has been made, the necessary algorithms are simple. For the same data, some data structures require more or less space than others; for the same operations on the data, some data structures lead to more or less efficient algorithms than others. This choices of algorithm and of data structures are closely intertwined, and we continually seek ways to save time or space by makeing the choice properly."Sedgewick [1998]

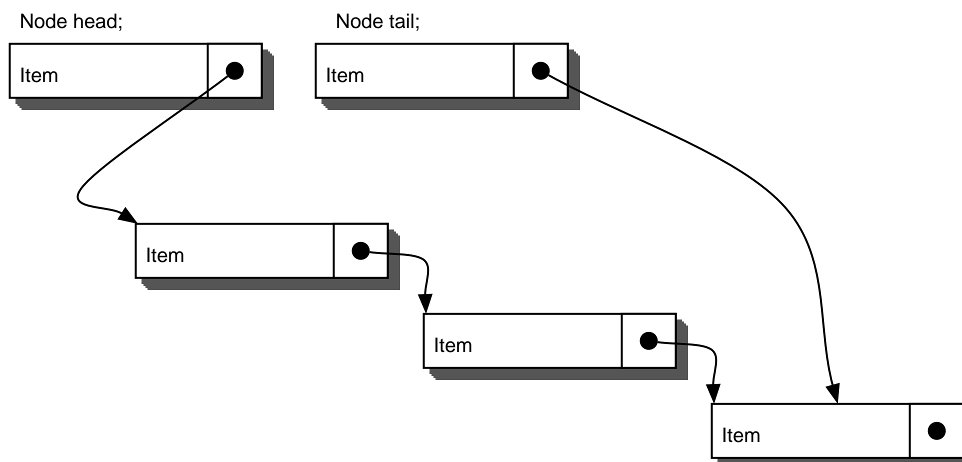
The *array* data structure has already been discussed. The next data structure to be discussed are *linked lists*.

3.1 Linked Lists

Whenever our primary interest is in traversing a collection of items sequentially, one by one, we can organize the items as a linked list. Items in a linked list can be rearranged efficiently, at the cost of quick access to any arbitryry item in the list. Because the only way to get to an item in the list is to follow all links from the beginning until the desired item is found.

Definition 1 A *linked list* is a set of items where each item is part of a *node* that also contains a *link* to a node. □

Figure 1: A linked list



A typical linked list is depicted in 1. It consists of *nodes*, that point to '`item`'s (instances of class '`Item`', that hold the data to be stored. The member '`next`' points to the next node in the list. The list structure itself consists of the members '`head`' and '`tail`', which point to the first and last nodes of the list, respectively.

3.1.1 Implementation

We use instances of class '`Item`' to store data that we want to process:

```
51 class Item
52 {
53     // Private Members
54     private int key;
55     private String value;
56
57     // Constructor
58     public Item(int key, String value)
59     {
60         this.key = key;
61         this.value = value;
62     }
63
64     public String toString()
65     {
66         return "(key=" + this.key + ", value=" + this.value + ")";
67     }
68
69     public int getKey()
70     {
71         return key;
72     }
73
74     public String getValue()
75     {
76         return value;
77     }
78 } // Item
```

And to build up the linked list, we use instances of class '`Node`':

```
 79 class Node
 80 {
 81     Node next;
 82     Item item;
 83
 84     Node(Item item)
 85     {
 86         this.next = null;
 87         this.item = item;
 88     }
 89
 90     Node(Node next, Item item)
 91     {
 92         this.next = next;
 93         this.item = item;
 94     }
 95
 96     public void setNext(Node next)
 97     {
 98         this.next = next;
 99     }
100
101     public Node getNext()
102     {
103         return next;
104     }
105
106     public Item getItem()
107     {
108         return item;
109     }
110 } // Node
```

A linked list, finally, is represented by instances of class ‘LinkedList’: The private members hold a reference to the first element in the list, the ‘head’, and also maintains a counter ‘size’, i. e. the number of nodes in that list.

```
111 class LinkedList
112 {
113     // private members
114     Node head;
115     int size;
116
117     // constructor
118     LinkedList()
119     {
120         head = null;
121         size = 0;
122     }
```

As a driver program, we use a slightly modified version of ‘App.java’. The first task is to read in items, and add them as new nodes into an already existing linked list:

```
123     private void processLine(String line)
124     {
125         StringTokenizer tokenizer = new StringTokenizer(line, ";");
126
127         int key = Integer.parseInt(tokenizer.nextToken());
128         String value = tokenizer.nextToken();
```

```
129
130         Item new_item = new Item(key, value);
131
132         list.add(new_item);
133     }
```

3.1.2 Adding to Linked Lists

Implementation of method 'add' looks like this:

```
134     // adds a new Item as node at the beginning of the list
135     public void add(Item item)
136     {
137         Node new_node = new Node(item);
138
139         if (head == null) {
140             // list is empty
141             new_node.setNext(null);
142             head = new_node;
143         } else {
144             // list is not empty
145             new_node.setNext(head);
146             head = new_node;
147         }
148
149         ++size;
150         //System.out.println("+" + toString());
151     }
```

3.1.3 Traversing Linked Lists

Traversing a list would be implementedd as follows. Starting with the node pointed to by 'head', we visit each node, printing out its item, and get the next node.

```
152         // traverse list and output contents
153         Node pos = list.getHead();
154         if (pos != null) {
155             do {
156                 System.out.println(pos.getItem().toString());
157             } while ((pos = pos.getNext()) != null);
158         }
```

3.1.4 Searching in Linked Lists

Searching can also be implemented, although not very efficiently, straight-forward. The search works by traversing the whole list, visiting each node, and comparing the items key with the looked for value.

```
159     public Node search(int value)
160     {
161         Node pos = head;
162
163         if (pos != null) {
164             while ((pos.getItem().getKey() != value) &&
165                    (next(pos) != null)) {
166                 pos = next(pos);
167             }
168             if (pos.getItem().getKey() != value)
169                 return null;
170             else
```

```
171                 return pos;
172         } else
173             return null;
174     }
```

3.1.5 Building Sorted Linked Lists

Unlike arrays, linked lists make it easy to sort them why building them up:

```
175     // inserts the given item as a new node at the correct position
176     // with respect to sorting.
177     public void insertSorted(Item item)
178     {
179         Node new_node = new Node(item);
180
181         if (head == null) {
182             // list is empty
183             new_node.setNext(null);
184             head = new_node;
185         } else if (head.getItem().getKey() > item.getKey()) {
186             // insert at head
187             new_node.setNext(head);
188             head = new_node;
189         } else {
190             boolean found = false;
191             Node pos = head;
192
193             // search for position to insert
194             while ((next(pos) != null) && !found) {
195                 if (next(pos).getItem().getKey() > item.getKey())
196                     found = true;
197                 else
198                     pos = next(pos);
199             }
200
201             if (found) {
202                 // insert into list
203                 new_node.setNext(pos.getNext());
204                 pos.setNext(new_node);
205             } else {
206                 // append to list
207                 pos.setNext(new_node);
208                 new_node.setNext(null);
209             }
210         }
211
212         ++size;
213         //System.out.println("+" + toString());
214     }
```

List of Tables

1 Characteristics of Selection Sort 3
2 Characteristics of Bubble Sort 4
3 Characteristics of Quick Sort 5

List of Figures

1 A linked list 6

List of Acronyms

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.