Sequential Data Structures

In this lecture we introduce the basic data structures for storing *sequences* of objects. These data structures are based on *arrays* and *linked lists*, which you met in first year (you were also introduced to stacks and queues). In this course, we give abstract descriptions of these data structures, and analyse the asymptotic running time of algorithms for their operations.

3.1 Abstract Data Types

The first step in designing a data structure is to develop a mathematical model for the data to be stored. Then we decide which methods we need to access and modify the data. Such a model, together with the methods to access and modify it, is an *abstract data type (ADT)*. An ADT completely determines the functionality of a data structure (*what* we want it to), but it says nothing about the implementation of the data structure and its methods (*how* the data structure is organised in memory, or which algorithms we implement for the methods). Clearly we will be very interested in which algorithms are used but not at the stage of defining the ADT. The particular algorithms/data structures that get used will influence the running time of the methods of the ADT. The definition of an ADT is something done at the beginning of a project, when we are concerned with the *specification*[1] of a system.

As an example, if we are implementing a dictionary ADT, we will need to perform operations such as look-up(w), where w is a word. We would know that this is an essential operation at the specification stage, before deciding on data structures or algorithms.

A *data structure* for *realising* (or *implementing*) an ADT is a structured set of variables for storing data. On the implementation level and in terms of JAVA, an ADT corresponds to a JAVA interface and a data structure realising the ADT corresponds to a class implementing the interface. The ADT determines the functionality of a data structure, thus an algorithm requiring a certain ADT works *correctly* with any data structure realising the ADT. Not all methods are equally efficient in the different possible implementations, and the choice of the right one can make a huge difference for the efficiency of an algorithm.

3.2 Stacks and Queues

A *Stack* is an ADT for storing a collection of elements, with the following methods:

- push(e): Insert element e (at the "top" of the stack).
- pop(): Remove the most recently inserted element (the element on "top") and return it; an error occurs if the stack is empty.
- isEmpty(): Return TRUE if the stack is empty and FALSE otherwise.

[1]You will learn about specification in Software engineering courses.

A stack obeys the LIFO (*Last-In, First-Out*) principle. The *Stack* ADT is typically implemented by building either on *arrays* (in general these need to be *dynamic arrays*, discussed in 3.4) or on *linked lists*[2]. Both types of implementation are straightforward, and efficient, taking $O(1)$ time (the (dynamic) array case is more involved, see 3.4) for any of the three operations listed above.

A *Queue* is an ADT for storing a collection of elements that retrieves elements in the opposite order to a stack. The rule for a queue is FIFO (*First-In, First-Out*). A queue supports the following methods:

- enqueue(e): Insert element e (at the "rear" of the queue).
- dequeue(): Remove the element inserted the longest time ago (the element at the "front") and return it; an error occurs if the queue is empty.
- isEmpty(): Return TRUE if the queue is empty and FALSE otherwise.

Like stacks, queues can easily be realised using (dynamic) arrays or linked lists. Again, whether we use arrays or linked lists, we can implement a queue so that all operations can be performed in $O(1)$ time.

*Stack*s and *Queue*s are very simple ADTs, with very simple methods—and this is why we can implement these ADTs so the methods all run in $O(1)$ time.

3.3 ADTs for Sequential Data

In this section, our mathematical model of the data is a linear *sequence* of elements. A sequence has well-defined *first* and *last* elements. Every element of a sequence except the first has a unique *predecessor* while every element except the last has a unique *successor*[3]. The *rank* of an element e in a sequence S is the number of elements before e in S.

The two most natural ways of storing sequences in computer memory are *arrays* and *linked lists*. We model the memory as a sequence of memory cells, each of which has a unique *address* (a 32 bit non-negative integer on a 32-bit machine). An *array* is simply a contiguous piece of memory, each cell of which stores one object of the sequence stored in the array (or rather a reference to the object). In a *singly linked list*, we allocate two successive memory cells for each object of the sequence. These two memory cells form a *node* of a sequence. The first stores the object and the second stores a reference to the next node of the list (i.e., the address of the first memory cell of the next node). In a *doubly linked list* we not only store a reference to the successor of each element, but

[2]Do not confuse the two structures. Arrays are by definition contiguous memory cells giving us efficiency both in terms of memory usage and speed of access. Linked lists do not have to consist of contiguous cells and for each cell we have to pay not only the cost of storing an item but also the location of the next cell. The disadvantage of arrays is that we cannot be sure of being able to grow them in situ whereas of course we can always grow a list (subject to memory limitations). Confusing the two things is inexcusable.

[3]A sequence can consist of a single element in which case the first and last elements are identical and of course there are no successor or predecessor elements. In some applications it also makes sense to allow the empty sequence in which case it does not of course have a first or last element.

also to its predecessor. Thus each node needs three successive memory cells. Figure 3.1 illustrates how an array, a singly linked list, and a doubly linked list storing the sequence $o1, o2, o3, o4, o5$ may be located in memory.[4] Figure 3.2 gives a more abstract view which is how we usually picture the data.

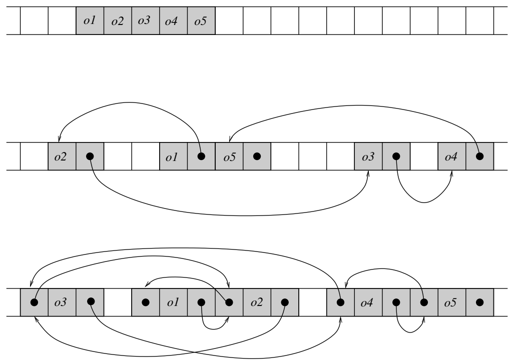

Figure 3.1. An array, a singly linked list, and a doubly linked list storing $o1, o2, o3, o4, o5$ in memory.

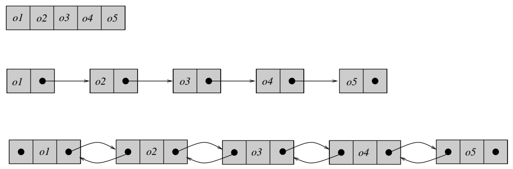

Figure 3.2. An array, a singly linked list, and a doubly linked list storing $o1, o2, o3, o4, o5$.

The advantage of storing a sequence in an array is that elements of the sequence can be accessed quickly in terms of rank. The advantage of linked lists is that they are flexible, of unbounded size (unlike arrays) and easily allow the insertion of new elements.

We will discuss two ADTs for sequences. Both can be realised using linked lists or arrays, but arrays are (maybe) better for the first, and linked lists for the second.

[4]This memory model is simplified, but it illustrates the main points.

Vectors

A *Vector* is an ADT for storing a sequence S of n elements that supports the following methods:

- elemAtRank(r): Return the element of rank r; an error occurs if $r < 0$ or $r > n - 1$.
- replaceAtRank(r, e): Replace the element of rank r with e; an error occurs if $r < 0$ or $r > n - 1$.
- insertAtRank(r, e): Insert a new element e at rank r (this increases the rank of all following elements by 1); an error occurs if $r < 0$ or $r > n$.
- removeAtRank(r): Remove the element of rank r (this reduces the rank of all following elements by 1); an error occurs if $r < 0$ or $r > n - 1$.
- size(): Return n, the number of elements in the sequence.

The most straightforward data structure for realising a vector stores the elements of S in an array A, with the element of rank r being stored at index r (assuming that the first element of an array has index 0). We store the length of the sequence in a variable n, which must always be smaller than or equal to $A.length$. Then the methods elemAtRank, replaceAtRank, and size have trivial algorithms[5] (cf. Algorithms 3.3–3.5) which take $O(1)$ time.

```
Algorithm elemAtRank($r$)
  1. return $A[r]$
```

Algorithm 3.3

```
Algorithm replaceAtRank($r, e$)
  1. $A[r] \leftarrow e$
```

Algorithm 3.4

```
Algorithm size()
  1. return $n$        // $n$ stores the current length of the sequence,
                          which may be different from the length of $A$.
```

Algorithm 3.5

By our general assumption that each line of code only requires a constant number of computation steps, the running time of Algorithms 3.3–3.5 is $\Theta(1)$.

[5]We don't worry about implementation issues such as error handling.

The implementation of insertAtRank and removeAtRank are much less efficient (see Algorithms 3.6 and 3.7). Also, there is a problem with insertAtRank if $n = A.length$ (we will consider this issue properly in § 3.4 on *dynamic arrays*), but for now we assume that the length of the array A is chosen to be large enough to never fill up. In the worst case the loop of insertAtRank is iterated n times and the loop of removeAtRank is iterated $n-1$ times. Hence $T_{\text{insertAtRank}}(n)$ and $T_{\text{removeAtRank}}(n)$ are both $\Theta(n)$.

```
Algorithm insertAtRank(r, e)
 1. for $i \leftarrow n$ downto $r+1$ do
 2.     $A[i] \leftarrow A[i-1]$
 3. $A[r] \leftarrow e$
 4. $n \leftarrow n+1$
```

Algorithm 3.6

```
Algorithm removeAtRank(r)
 1. for $i \leftarrow r$ to $n-2$ do
 2.     $A[i] \leftarrow A[i+1]$
 3. $n \leftarrow n-1$
```

Algorithm 3.7

The *Vector* ADT can also be realised by a data structure based on linked lists. Linked lists do not properly support the access of elements based on their rank. To find the element of rank r, we have to step through the list from the beginning for r steps. This makes all methods required by the *Vector* ADT quite inefficient, with running time $\Theta(n)$.

Lists

Suppose we had a sequence and wanted to remove every element satisfying some condition. This would be possible using the *Vector* ADT, but it would be inconvenient and inefficient (for the standard implementation of *Vector*). However, if we had our sequence stored as a linked list, it would be quite easy: we would just step through the list and remove nodes holding elements with the given condition. Hence we define a new ADT for sequences that abstractly reflects the properties of a linked list—a sequence of nodes that each store an element, have a successor, and (in the case of doubly linked lists) a predecessor. We call this ADT *List*. Our abstraction of a node is a *Position*, which is itself an ADT associated with *List*. The basic methods of the *List* are:

- element(p): Return the element at position p.

- first(): Return the position of the first element; an error occurs if the list is empty.
- isEmpty(): Return TRUE if the list is empty and FALSE otherwise.
- next(p): Return the position of the element following the one at position p; an error occurs if p is the last position.
- isLast(p): Return TRUE if p is the last position of the list and FALSE otherwise.
- replace(p, e): Replace the element at position p with e.
- insertFirst(e): Insert e as the first element of the list.
- insertAfter(p, e): Insert element e after position p.
- remove(p): Remove the element at position p.

List also has methods last(), previous(p), isFirst(p), insertLast(e), and insertBefore(p, e). These methods correspond to first(), next(p), isLast(p), insertFirst(e), and insertAfter(p, e) if we reverse the order of the list; their functionality should be obvious.

The natural way of realising the *List* ADT is by a data structure based on a doubly linked list. *Position*s are realised by *nodes* of the list, where each node has fields *previous*, *element*, and *next*. The list itself stores a reference to the first and last node of the list. Algorithms 3.8–3.9 show implementations of insertAfter and remove.

```
Algorithm insertAfter(p, e)
 1. create a new node $q$
 2. $q.element \leftarrow e$
 3. $q.next \leftarrow p.next$
 4. $q.previous \leftarrow p$
 5. $p.next \leftarrow q$
 6. $q.next.previous \leftarrow q$
```

Algorithm 3.8

```
Algorithm remove(p)
 1. $p.previous.next \leftarrow p.next$
 2. $p.next.previous \leftarrow p.previous$
 3. delete $p$        (done automatically in JAVA by garbage collector.)
```

Algorithm 3.9

The asymptotic running time of Algorithms 3.8–3.9 is $\Theta(1)$. It is easy to see that all other methods can also be implemented by algorithms of asymptotic running time $\Theta(1)$ but only because we assume that p *is given as a direct "pointer"* (to the relevant node). An operation such as insertAtRank, or asking to insert into the element's "sorted" position, would take $\Omega(n)$ worst-case running time on a list.

Given the trade-off between using *List* and *Vector* for different operations, it makes sense to consider combining the two ADTs into one ADT *Sequence*, which will support all methods of both *Vector* and *List*. For *Sequence*, both arrays and linked lists are more efficient on some methods than on others. The data structure used in practice would depend on which methods are expected to be used most frequently in the application.

3.4 Dynamic Arrays

There is a real problem with the array-based data structures for sequences that we have not considered so far. What do we do when the array is full? We cannot simply extend it, because the part of the memory following the block where the array sits may be used for other purposes at the moment. So what we have to do is to allocate a sufficiently large block of memory (large enough to hold both the current array and the additional element we want to insert) somewhere else, and then copy the whole array there. This is not efficient, and we clearly want to avoid doing it too often. Therefore, we always choose the length of the array to be a bit larger than the number of elements it currently holds, keeping some extra space for future insertions. In this section, we will see a strategy for doing this surprisingly efficiently.

Concentrating on the essentials, we shall only implement the very basic ADT *VeryBasicSequence*. It stores a sequence of elements and supports the methods elemAtRank(r), replaceAtRank(r,e) of *Vector* and the method addLast(e) of *List*. So it is almost like a queue without the dequeue operation.

Our data structure stores the elements of the sequence in an array A. We store the current size of the sequence in a variable n and let N be the length of A. Thus we must always have $N \geq n$. The *load factor* of our array is defined to be n/N. The load factor is a number between 0 and 1 indicating how much space we are wasting. If it is close to 1, most of the array is filled by elements of the sequence and we are not wasting much space. In our implementation, we will always maintain a load factor of at least $1/2$.

The methods elemAtRank(r) and replaceAtRank(r,e) can be implemented as for *Vector* by algorithms of running time $\Theta(1)$. Consider Algorithm 3.10 for insertions. As long as there is room in the array, insertLast simply inserts the element at the end of the array. If the array is full, a new array of length twice the length of the old array plus the new element is created. Then the old array is copied to the new one and the new element is inserted at the end.

```
Algorithm insertLast($e$)
 1. if $n < N$ then
 2.     $A[n] \leftarrow e$
 3. else                  // $n = N$, i.e., the array is full
 4.     $N \leftarrow 2(N+1)$
 5.     Create new array $A'$ of length $N$
 6.     for $i = 0$ to $n-1$ do
 7.         $A'[i] \leftarrow A[i]$
 8.     $A'[n] \leftarrow e$
 9.     $A \leftarrow A'$
10. $n \leftarrow n+1$
```

Algorithm 3.10

Amortised Analysis

By letting the length of the new array be at most twice the number of elements it currently holds, we guarantee that the load factor is always at least $1/2$. Unfortunately, the worst-case running time of inserting one element in a sequence of size n is $\Theta(n)$, because we need $\Omega(n)$ steps to copy the old array into the new one in lines 6–7. However, we only have to do this copying phase occasionally, and in fact we do it less frequently as the sequence grows larger. This is reflected in the following theorem, which states that if we average the worst-case performance of *a sequence of insertions* (this is called *amortised analysis*), we only need an average of $O(1)$ time for each.

Theorem 3.11. *Inserting m elements into an initially empty VeryBasicSequence using the method* insertLast *(Algorithm 3.10) takes $\Theta(m)$ time.*

If we only knew the worst-case running time of $\Theta(n)$ for a single insertion into a sequence of size n, we might conjecture a worst-case time of $\sum_{n=1}^{m} \Theta(n) = \Theta(m^2)$ for m insertions into an initially empty *VeryBasicSequence*. Our $\Theta(m)$ bound is therefore a big improvement. Analysing the worst-case running-time of a total *sequence* of operations, is called *amortised analysis*.

PROOF Let $I(1), \ldots, I(m)$ denote the m insertions. For most insertions only lines 1,2, and 10 are executed, taking $\Theta(1)$ time. These are called *cheap* insertions. Occasionally on an insertion we have to create a new array and copy the old one into it (lines 4-9). When this happens for an insertion $I(i)$ it requires time $\Theta(i)$, because all of the elements $l(1), \ldots, l(i-1)$ need to be copied into a new array (lines 6-7). These are called *expensive* insertions. Let $I(i_1), \ldots, I(i_\ell)$, where $1 \leq i_1 < i_2 < \ldots < i_\ell \leq m$, be all the expensive insertions.

Then the overall time we need for all our insertions is

$$\sum_{j=1}^{\ell} \Theta(i_j) + \sum_{\substack{1 \le i \le m \\ i \ne i_1, \dots, i_\ell}} \Theta(1) \tag{3.1}$$

We now split this into two parts, one for O and one for Ω (recall that $f = \Theta(g)$ iff $f = O(g)$ and $f = \Omega(g)$). First consider O.

$$\sum_{j=1}^{\ell} O(i_j) + \sum_{\substack{1 \le i \le m \\ i \ne i_1, \dots, i_\ell}} O(1) \le \sum_{j=1}^{\ell} O(i_j) + \sum_{i=1}^{m} O(1) \le O\Big(\sum_{j=1}^{\ell} i_j\Big) + O(m), \tag{3.2}$$

where at the last stage we have repeatedly applied rule (2) of Theorem 2.3 (lecture notes 2). To give an upper bound on the last term in (3.2), we have to determine the i_j. This is quite easy: We start with $n = N = 0$. Thus the first insertion is expensive, and after it we have $n = 1, N = 2$. Therefore, the second insertion is cheap. The third insertion is expensive again, and after it we have $n = 3, N = 6$. The next expensive insertion is the seventh, after which we have $n = 7, N = 14$. Thus $i_1 = 1, i_2 = 3, i_3 = 7$. The general pattern is

$$i_{j+1} = 2i_j + 1.$$

Now an easy induction shows that $2^{j-1} \le i_j < 2^j$, and this gives us

$$\sum_{j=1}^{\ell} i_j \le \sum_{j=1}^{\ell} 2^j = 2^{\ell+1} - 2 \tag{3.3}$$

(summing the *geometric series*). Since $2^{\ell-1} \le i_\ell \le m$, we have $\ell \le \lg(m) + 1$. Thus

$$2^{\ell+1} - 2 \le 2^{\lg(m)+2} - 2 = 4 \cdot 2^{\lg(m)} - 2 = 4m - 2 = O(m). \tag{3.4}$$

Then by (3.1)–(3.4), the amortised running time of $I(1), \dots, I(m)$ is $O(m)$.

Next we consider Ω. By (3.1) and by properties of Ω , we know

$$\sum_{j=1}^{\ell} \Omega(i_j) + \sum_{\substack{1 \le i \le m \\ i \ne i_1, \dots, i_\ell}} \Omega(1) \ge \sum_{\substack{1 \le i \le m \\ i \ne i_1, \dots, i_\ell}} \Omega(1). \tag{3.5}$$

Recall that we have shown that $i_1 = 1$ and for $j \ge 1$, $i_{j+1} = 2i_j + 1$. We claim that for any $k \in \mathbb{N}$, with $k \ge 4$, the set $\{1, 2, 3, \dots, k\}$ contains at most $k/2$ occurrences of i_j indices (This can be proven by induction, left as an exercise). Hence there at least $m/2$ elements of the set $\{1 \le i \le m \mid i \ne i_1, \dots, i_\ell\}$. Hence

$$\sum_{\substack{1 \le i \le m \\ i \ne i_1, \dots, i_\ell}} \Omega(1) \ge (m/2)\Omega(1) = \Omega(m), \tag{3.6}$$

whenever $m \ge 4$. Combining (3.5) and (3.6), the running time for $I(1), \dots, I(m)$ is $\Omega(m)$. Together with our $O(m)$ result, this implies the theorem. □

Note that although the theorem makes a statement about the "average time" needed by an insertion, it is *not* a statement about average running time. The reason is that the statement of the theorem is completely independent of the input, i.e., the elements we insert. In this sense, it is a statement about the worst-case running time, whereas average running time makes statements about "average", or random, inputs. Since an analysis such as the one used for the theorem occurs quite frequently in the theory of algorithms, there is a name for it: *amortised analysis*. A way of rephrasing the theorem is saying that the *amortised (worst case) running time* of the method insertLast is $\Theta(1)$.

Finally, suppose that we want to add a method removeLast() for removing the last element of our ADT *VeryBasicSequence*. We can use a similar trick for implementing this—we only create a new array of, say, size $3/4$ of the current one, if the load factor falls below $1/2$. With this strategy, we always have a load factor of at least $1/2$, and it can be proved that the amortised running time of both insertLast and removeLast is $\Theta(1)$.

The JAVA Collections Framework contains the ArrayList implementation of List (in the class java.util.ArrayList), as well as other list implementations. ArrayList, though not implemented as a dynamic array, provides most features of a dynamic array.

3.5 Further reading

If you have [GT]: The chapters "Stacks, Queues and Recursion" and "Vectors, Lists and Sequences".

Exercises

1. Implement the method removeLast() for removing the last element of a dynamic array as described in the last paragraph of Section 3.4.

2. Prove by induction that in the proof of Theorem 3.11, the following claim (towards the end of the proof) is true:
For any $k \in \mathbb{N}$, $k \ge 4$, the set $\{1, 2, 3, \dots, k\}$ contains at most $k/2$ occurrences of i_j indices.

3. What is the amortised running time of a sequence $P = p_1 p_2 \dots p_n$ of operations if the running time of p_i is $\Theta(i)$ if i is a multiple of 3 and $\Theta(1)$ otherwise?
What if the running time of p_i is $\Theta(i)$ if i is a square and $\Theta(1)$ otherwise?
Hint: For the second question, use the fact that $\sum_{i=1}^{m} i^2 = \Theta(m^3)$.

4. Implement in JAVA a `Stack` class based on dynamic arrays.