Sequential Data Structures

In this lecture, we start our introduction into the most important data structures by studying data structures for storing sequences of objects. These data structures are based on arrays and linked lists, which should be familiar to you from CS1.

3.1 Abstract Data Types

The first step in designing a data structure is to develop a mathematical model for the data to stored in the data structure. The next step is to decide which methods we need to access and modify the data. A mathematical model together with methods to access and modify it form what we call an abstract data type (ADT). An ADT completely determines the functionality of a data structure, but it does not yet say anything about how the data described by the mathematical model are organised in computer memory, and which algorithms are used to implement the methods. A data structure realizing, or implementing, an ADT consists collections of variables, which may be connected in various ways, for storing the data, and algorithms implementing the methods of the ADT.

On the implementation level, an ADT corresponds to a JAVA interface and a data structure realising the ADT corresponds to a class implementing the interface.

Usually there are many different data structures realising an ADT. 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. But usually not all methods are implemented equally efficient in the different data structures, and the choice of the right one can make a huge difference for the efficiency of an algorithm.

As an example, consider Practical 5 with the three different data structures implementing the Dictionary ADT.

3.2 Stacks and Queues

A Stack is an ADT for storing a collection of elements that supports 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” of the stack) and return it; an error occurs if the stack is empty.

• isEmpty(): Return TRUE if the stack is empty and FALSE otherwise.

The two principal ways of realising the stack ADT is by data structures building on arrays or linked lists. Both are straightforward and can be found in any algorithms textbook.

A Queue is an ADT for storing a collection of elements that is very similar to a stack, but has a different access/removal policy (first-in-first-out, or FIFO, instead of last-in-first-out, or LIFO, for stacks). 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” of the queue) and return it; an error occurs if the queue is empty.

• isEmpty(): Return TRUE if the queue is empty and FALSE otherwise.

As stacks, queues can easily be realised using arrays or linked lists.

In a sense, Stacks and Queues are particularly simple ADTs for sequential data. We shall study more complicated ones next.

### 3.3 ADTs for Sequential Data

In this section, our mathematical model of the data is a linear sequence of elements. Note that a sequence has well-defined first and last elements. Moreover, every element of a sequence except the first and last has a unique predecessor and successor. 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 simply is 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 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 \( o_1, o_2, o_3, o_4, o_5 \) may be located in memory.\(^1\) Figure 3.2 gives a more abstract view.

\[ \begin{array}{ccccccc}
| & o_1 & o_2 & o_3 & o_4 & o_5 & & \\
\end{array} \]

\[ \begin{array}{ccccccc}
| & & & & & & \\
\end{array} \]

\[ \begin{array}{ccccccc}
| & & & & & & \\
\end{array} \]

\[ \begin{array}{ccccccc}
| & & & & & & \\
\end{array} \]

\[ \begin{array}{ccccccc}
| & & & & & & \\
\end{array} \]

\[ \begin{array}{ccccccc}
| & & & & & & \\
\end{array} \]

\[ \begin{array}{ccccccc}
| & & & & & & \\
\end{array} \]

Figure 3.1. An array, a singly linked list, and a doubly linked list storing \( o_1, o_2, o_3, o_4, o_5 \) in memory.

The advantage of storing a sequence in an array is that it gives us fast access to every single element of the sequence if we know its rank. The advantage of linked lists is that they are quite flexible and easily allow for inserting new elements.

In the following, we shall discuss two ADTs for sequences. Both can be realised using either linked lists or arrays, but arrays are more suitable for the first and linked lists for the second.

Vectors

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

\(^1\)Of course this memory model is simplified, but it illustrates the main points.
The most straightforward data structure realising a vector stores the elements of $S$ in an array $A$, with the element of rank $r$ being stored at index $r$. We store the length of the sequence in a variable $n$, which always must be smaller than or equal to $A.length$. Then the methods elemAtRank, replaceAtRank, and size have trivial algorithms (cf. Algorithms 3.3–3.5).\footnote{When describing algorithms here, we usually don’t worry about implementation issues such as error handling.}

**Algorithm** elemAtRank($r$)

1. return $A[r]$
Algorithm \text{replaceAtRank}(r, e)

1. \( A[r] \leftarrow e \)

\textbf{Algorithm 3.4}

Algorithm \text{size}()

1. \text{return } n \quad // \ n \text{ stores the current length of the sequence, which may be different of the length of } A.

\textbf{Algorithm 3.5}

The implementation of the insert and remove methods is only slightly more difficult, but much less efficient (see Algorithms 3.6 and 3.7). There is a problem with \text{insertAtRank} if \( n = A.\text{length} \). For now, let us just assume that the length of the array \( A \) is chosen big enough in advance that it never fills up. We will reconsider the issue in Section 3.4 on \textit{dynamic arrays}. In the worst case the loop of \text{insertAtRank} is iterated \( n \) times, and the loop of \text{removeAtRank} is iterated \( n - 1 \) times. Thus the time complexity of both algorithms is \( \Theta(n) \).

Algorithm \text{insertAtRank}(r, e)

1. \textbf{for } i \leftarrow n \textbf{ downto } r + 1 \textbf{ do}
2. \hspace{1em} \( A[i] \leftarrow A[i - 1] \)
3. \hspace{1em} \( A[r] \leftarrow e \)
4. \hspace{1em} \( n \leftarrow n + 1 \)

\textbf{Algorithm 3.6}

The vector ADT can also be realised by a data structure based on linked lists. However, linked lists do not support access of elements based on their rank in the sequence very well. To find the element of rank \( r \), one has to step through the list from the beginning for \( r \) steps. This makes all methods required by the vector ADT quite inefficient, their time complexity is \( \Theta(n) \).

\textbf{Lists}

Suppose we had a sequence and wanted to remove every element satisfying some condition. Though in principle this would be possible using just the methods of our \textit{Vector ADT}, it would be inconvenient and also inefficient for the standard implementations of vectors. However, if we had our sequence stored as a linked list, it would be quite easy: We could just step through the list and remove the nodes holding elements with the specified property. However, we can only do
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

this if we are allowed to manipulate the list directly and not only through the methods of Vector.

We define a new ADT for sequences that abstractly reflects the property of a linked list of consisting of 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:

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

In addition, List has methods last(), previous(), isFirst(), 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. Positions 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. It is straightforward to implement all the methods. Algorithms 3.8–3.10 are three examples.

The asymptotic time complexity of Algorithms 3.8–3.10 is \( \Theta(1) \), and it is easy to see that all other methods can also be implemented by algorithms of asymptotic time complexity \( \Theta(1) \).
**Algorithm** first()

1. return first

**Algorithm 3.8**

**Algorithm** insertAfter(p, e)

1. create a new node q
2. q.element ← e
3. q.next ← p.next
4. q.previous ← p
5. p.next ← q

**Algorithm 3.9**

We could also implement the List ADT by an array based data structure. Then positions would simply be indices of the array. However, the methods for inserting and removing elements (except insertLast) would require time Θ(n).

So it seems that implementing the Vector ADT by linked lists or the List ADT by arrays does not make much sense. However, it could make sense if we combine the two ADTs into one ADT Sequence supporting all methods of both Vector and List. For a Sequence, both arrays and linked lists are more efficient on some methods, and the decision which data structure to use would depend on which methods can be expected to be used most frequently in the application at hand.

### 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 if the array is full? We cannot simply extend it, because the part of the memory following the chunk where the array sits may be used for other purposes at the moment. So what we have to do is allocate a sufficiently large piece of memory (sufficiently large to hold both the current array and the additional element we want to insert) somewhere else and then copy the whole array there. Of course this is not very efficient, and we clearly want to avoid doing it too often. Therefore, we always choose the length of the array 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.
**Algorithm** `remove(p)`

1. `p.previous.next ← p.next`
2. `p.next.previous ← p.previous`
3. `delete p` // In JAVA, this is done automatically by the garbage collector.

**Algorithm 3.10**

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 always must 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 time complexity $\Theta(1)$. Consider Algorithm 3.11 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 in the end.

**Algorithm** `insertLast(e)`

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

**Algorithm 3.11**

By letting the length of the new array be at most twice the number of elements it holds, we guarantee that the load factor is always at least $1/2$. So we are reasonably space efficient. What about the time complexity? Unfortunately, the worst-case time complexity of inserting one element in a sequence of size $n$ is
\( \Theta(n) \), because this is what it costs to copy the old array into the new one in lines 6–7. However, we only have to create a new array and copy the old one into it once in a while, and actually we have to do it less frequently as the sequence gets larger (and therefore the copying more expensive). This is reflected in the following theorem, which states that if we average over a sequence of insertions, we only need time \( O(1) \) for each of them.

**Theorem 3.12.** Inserting \( m \) elements into an initially empty \( \text{VeryBasicSequence} \) using the method \texttt{insertLast} (Algorithm 3.11) requires time \( O(m) \).

Before proving this theorem, let me emphasis that from the worst-case time complexity of \( \Theta(n) \) for one insertion into a sequence of size \( n \) we would expect a time \( \sum_{n=1}^{m} \Theta(n) = \Theta(m^2) \) for \( m \) insertions into an initially empty \( \text{VeryBasicSequence} \). So this is really an improvement.

**Proof (of the theorem):** Let \( I(1), \ldots, I(m) \) be the \( m \) insertions. For most of them, only lines 1, 2, and 10 are executed, which requires time \( \Theta(1) \). Let us call such an insertion \textit{cheap}. Only for some of the insertions we have to create a new array and copy the old one into it (lines 4-9); if this happens for insertion \( I(i) \) it requires time \( \Theta(i) \). We call such an insertion \textit{expensive}. Let \( I(i_1), \ldots, I(i_\ell) \), where \( 1 \leq i_1 < i_2 < \cdots < i_\ell \leq m \), be all expensive insertions.

Then the overall time we need for all our insertions is

\[
\sum_{j=1}^{\ell} \Theta(i_j) + \sum_{1 \leq i \leq m \atop i \neq i_1, \ldots, i_\ell} \Theta(1) \leq \sum_{j=1}^{\ell} \Theta(i_j) + \sum_{i=1}^{m} \Theta(1) \leq \Theta \left( \sum_{j=1}^{\ell} i_j \right) + \Theta(m). \tag{3.1}
\]

To give an upper bound on the last term in (3.1), 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 \( 2^{j-1} \leq i_j < 2^j \), and this is good enough for us. It gives us

\[
\sum_{j=1}^{\ell} i_j \leq \sum_{j=1}^{\ell} 2^j = 2^{\ell+1} - 2 \tag{3.2}
\]

(by the summation rule for \textit{geometric series}). Since \( 2^{\ell-1} \leq i_\ell \leq m \), we have \( \ell \leq \lg(m) + 1 \). Thus

\[
2^{\ell+1} - 2 \leq 2^{\lg(m)+2} - 2 = 4 \cdot 2^{\lg(m)} - 2 = 4m - 2 \in O(m). \tag{3.3}
\]

Together, (3.1)–(3.3) imply the statement of the theorem. \( \square \)

Note that although the theorem makes a statement about the “average time” needed by an insertion, it is \textit{not} a statement about average time complexity. 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 time complexity, whereas average time complexity 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) time complexity of the method insertLast is $O(1)$.

Finally, suppose that we want to add a method removeLast() for removing the last element to our ADT VeryBasicSequence. We can use a similar trick for implementing this — we only create a new array, say, of 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 time complexity of both insertLast and removeLast is $\Theta(1)$.

**Reading Assignment**

Sections 1.5, 2.1, and 2.2 of [GT]. Chapter 6 of [AU] covers some, but not all the material of this lecture.

*Martin Grohe*