

Abstract Data Types

COM2404 *Foundations of Computer Science*

2002–03

1 Lists

Up to now we have talked about lists in a rather abstract way; we have said nothing about how list structures and operations on them should be implemented in a program. In principle there could be many ways of implementing list structures; in practice there are two main methods: we may use either *arrays* or *pointers*.

The easiest way of implementing a list is to use an array structure. In Java we could declare a list-of-integers identifier using `int[]`. If we do this, we have to make sure that we implement operations like *cons*, *head*, *tail*, and *Append* in the right way. For example, to *cons* a new element into a list we have to move all the existing elements along one place and then slot the new element into the vacated position at the beginning. The following code works:

```
public static int[] cons(int a, int[] list)
{
    int[] temp = new int[list.length+1];
    temp[0]=a;
    for (int i=1; i<temp.length; i++) temp[i]=list[i-1];
    return temp;
}
```

Exercise: Write suitable code for the functions *head*, *tail*, and *Append* using the array representation.

Another way of implementing lists is using pointers. This can be set up using the following code:

```
public static class list
{
    int head;
    list tail;

    public list(int h, list t)
    {
        head = h;
        tail = t;
    }
}
```

The code for *cons* is now almost trivial:

```
public static list cons(int h, list t)
{
    list l = new list(h,t);
    return l;
}
```

Exercise: Write code for *Append* using the pointer implementation of lists.

Both these methods implement a common underlying concept of lists and their associated operations. Suppose someone asked ‘Have you implemented lists correctly?’. This question is only meaningful because we have the underlying concept which exists separately from any particular implementations of it. The underlying concept can be thought of as setting the standard for possible implementations; it defines what is to count as a correct implementation. A disciplined way of programming will always start with the abstract concepts and then seek to implement them correctly.

In this section we shall look at a standard way of presenting these underlying abstract concepts, which are known as *abstract data types*. An abstract data type specifies a set of *operations* on elements of the type, and also lays down some *relations* which these operations must satisfy.

Here is a minimal specification of the abstract data type *List(Item)*. We assume that the type *Item* has already been specified (if, for example, we are interested in lists of integers, then we’d replace *Item* by *Integer* throughout):

Type <i>List(Item)</i>		
Sorts	Operations	Relations
<i>Item</i>	<i>empty</i> : <i>List</i>	<i>head(cons(x, y))</i> = <i>x</i>
<i>List</i>	<i>head</i> : <i>List</i> → <i>Item</i> ∪ { <i>error</i> }	<i>tail(cons(x, y))</i> = <i>y</i>
	<i>tail</i> : <i>List</i> → <i>List</i> ∪ { <i>error</i> }	<i>head(empty)</i> = <i>error</i>
	<i>cons</i> : <i>Item</i> × <i>List</i> → <i>List</i>	<i>tail(empty)</i> = <i>error</i>

Of the operations, *empty* is a *constant*, *cons* is a *constructor function* (because it is used for building up new lists out of old), and *head* and *tail* are *selector functions* (because they are used for picking out the components from which a list is built). The relations ensure that these all bear the correct relationship to each other, i.e., that they behave as we want them to.

Note that the specification is abstract in the sense that it says nothing about what lists actually *are*; it merely says what you can do to them and how they behave when you do it. *Any* collection of objects, abstract or concrete, which behave in the way specified will do as a way of implementing lists.

Using this minimal specification of *list* we can go on to introduce a richer repertoire of operations, such as *Length*, *Append*, and *Reverse*. These can be defined using recurrence relations as in the previous section, and theorems can be proved about how they behave.

We might want to specify *list* in such a way that these extra operations are included in the type definition. A richer *List* ADT is given below:

Type <i>List⁺(Item)</i>		
Sorts	Operations	Relations
<i>Item</i>	<i>empty</i> : <i>List</i>	<i>head(cons(x, y))</i> = <i>x</i>
<i>List</i>	<i>head</i> : <i>List</i> → <i>Item</i> ∪ { <i>error</i> }	<i>tail(cons(x, y))</i> = <i>y</i>
<i>Integer</i>	<i>tail</i> : <i>List</i> → <i>List</i> ∪ { <i>error</i> }	<i>head(empty)</i> = <i>error</i>
	<i>cons</i> : <i>Item</i> × <i>List</i> → <i>List</i>	<i>tail(empty)</i> = <i>error</i>
	<i>append</i> : <i>List</i> × <i>List</i> → <i>List</i>	<i>append(empty, x)</i> = <i>x</i>
	<i>length</i> : <i>List</i> → <i>Integer</i>	<i>append(cons(x, y), z)</i> = <i>cons(x, append(y, z))</i>
	<i>reverse</i> : <i>List</i> → <i>List</i>	<i>length(empty)</i> = 0
		<i>length(cons(x, y))</i> = <i>length(y)</i> + 1
		<i>reverse(empty)</i> = <i>empty</i>
		<i>reverse(cons(x, y))</i> = <i>append(reverse(y), cons(x, empty))</i>

You might think that we should add some further relations such as:

$$\begin{aligned} \text{append}(x, \text{empty}) &= x \\ \text{append}(x, \text{append}(y, z)) &= \text{append}(\text{append}(x, y), z) \\ \text{length}(\text{append}(x, y)) &= \text{length}(x) + \text{length}(y) \\ \text{reverse}(\text{reverse}(x)) &= x \\ \text{length}(\text{reverse}(x)) &= \text{length}(x) \\ \text{reverse}(\text{append}(x, y)) &= \text{append}(\text{reverse}(y), \text{reverse}(x)) \end{aligned}$$

but these would be redundant because they can all be proved to hold using the relations already given. This means that they will be automatically satisfied in any correct implementation of the data type as given.

Exercise. We've already proved the second and third of these relations in the section on Induction on Lists. See if you can prove the others.

2 Binary trees

The technique illustrated here for specifying lists can be used for other data types commonly used in computing. A frequently encountered example is the *binary tree*. An example is shown in Figure 1. This tree contains eleven *nodes*, each labelled with a letter. Nodes are of two types, *leaf nodes* (in the figure, these are d, e, g, h, j and k), and *branch nodes* (a, b, c, f , and i). Each branch node has two branches hanging down from it, and there is a node at the end of each of these branches. Node a is the *root node* of the tree, from which everything else hangs. Any of the other nodes can be regarded as the root node of a *subtree* of the tree. This gives us a way of describing the structure of the tree. At the highest level of description, the tree consists of the root node a , which is a branch node with branches to b and c , each of which is in turn the root node of a subtree. These subtrees can then be described similarly: the left subtree consists of its root node b , with branches to nodes d and e which are both leaf nodes; and similarly with the right subtree. A leaf node can be regarded as a minimal subtree; thus a node all on its own, with no branching, counts as a binary tree, albeit rather a trivial one.

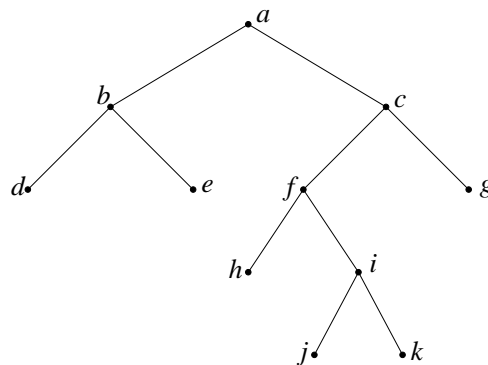


Figure 1: A binary tree

We can therefore describe the recursive construction of the type $Bintree(Item)$ of binary trees with labels of type $Item$ as follows: given item x , we can form a minimal tree $leaf(x)$; and given item x and trees L and R we can form a tree $tree(x, L, R)$ which has x as its root node, with branches to L on the left

and R on the right. Thus the tree in Figure 1 can be constructed as follows:

$$\begin{aligned}
T &= \text{tree}(a, T_1, T_2) \\
T_1 &= \text{tree}(b, T_3, T_4) \\
T_2 &= \text{tree}(c, T_5, T_6) \\
T_3 &= \text{leaf}(d) \\
T_4 &= \text{leaf}(e) \\
T_5 &= \text{tree}(f, T_7, T_8) \\
T_6 &= \text{leaf}(g) \\
T_7 &= \text{leaf}(h) \\
T_8 &= \text{tree}(i, T_9, T_{10}) \\
T_9 &= \text{leaf}(j) \\
T_{10} &= \text{leaf}(k)
\end{aligned}$$

T_1, T_2, \dots, T_{10} can be eliminated from these equations to give the single rather hard-to-read equation

$$T = \text{tree}(a, \text{tree}(b, \text{leaf}(d), \text{leaf}(e)), \text{tree}(c, \text{tree}(f, \text{leaf}(h), \text{tree}(i, \text{leaf}(j), \text{leaf}(k))), \text{leaf}(g))).$$

We now specify *bintree* as an abstract data type:

Type <i>Bintree</i> (<i>Item</i>)		
Sorts	Operations	Relations
<i>Item</i>	$\text{leaf} : \text{Item} \rightarrow \text{Bintree}$	$\text{root}(\text{leaf}(x)) = x$
<i>Bintree</i>	$\text{tree} : \text{Item} \times \text{Bintree} \times \text{Bintree} \rightarrow \text{Bintree}$	$\text{left}(\text{leaf}(x)) = \text{error}$
	$\text{root} : \text{Bintree} \rightarrow \text{Item}$	$\text{right}(\text{leaf}(x)) = \text{error}$
	$\text{left}, \text{right} : \text{Bintree} \rightarrow \text{Bintree} \cup \{\text{error}\}$	$\text{root}(\text{tree}(x, y, z)) = x$
		$\text{left}(\text{tree}(x, y, z)) = y$
		$\text{right}(\text{tree}(x, y, z)) = z$

As with lists, we can go on to define various other functions on the *Bintree* data type. Two examples are *depth*, which gives the length of the longest path from the root to a leaf in the tree; and *leaves*, which gives the total number of leaves. These can be defined by the following recurrence relations:

$$\text{depth}(\text{leaf}(x)) = 0 \tag{D1}$$

$$\text{depth}(\text{tree}(x, L, R)) = \max(\text{depth}(L), \text{depth}(R)) + 1 \tag{D2}$$

$$\text{leaves}(\text{leaf}(x)) = 1 \tag{Lvs1}$$

$$\text{leaves}(\text{tree}(x, L, R)) = \text{leaves}(L) + \text{leaves}(R) \tag{Lvs2}$$

To see how these work, we compute $\text{depth}(T)$ for the tree T of Figure 1.

Since $T_3, T_4, T_6, T_8, T_9, T_{10}$ are leaves, they all have depth 0. Then we have

$$\text{depth}(T_1) = \max(\text{depth}(T_3), \text{depth}(T_4)) + 1 = \max(0, 0) + 1 = 1$$

$$\text{depth}(T_8) = \max(\text{depth}(T_9), \text{depth}(T_{10})) + 1 = \max(0, 0) + 1 = 1$$

$$\text{depth}(T_5) = \max(\text{depth}(T_7), \text{depth}(T_8)) + 1 = \max(0, 1) + 1 = 2$$

$$\text{depth}(T_2) = \max(\text{depth}(T_5), \text{depth}(T_6)) + 1 = \max(2, 0) + 1 = 3$$

$$\text{depth}(T) = \max(\text{depth}(T_1), \text{depth}(T_2)) + 1 = \max(1, 3) + 1 = 4$$

so T has depth 4 (the length of a path from the root to leaf j or leaf k).

Exercise. Compute $leaves(T)$ for this tree. (Your answer should be 6.)

There is an important relationship between the depth of a binary tree and the number of leaves it has. This is expressed by the following theorem:

Theorem. For any binary tree T , $leaves(T) \leq 2^{depth(T)}$.

Proof. We use induction on $depth(T)$.

Base case ($depth(T) = 0$). In this case $T = leaf(x)$ for some x , and we have $leaves(T) = 1$. Since $1 \leq 2^0$, the base case is established.

Induction step (from $depth(T) = n$ to $depth(T) = n + 1$). Assume as induction hypothesis that the result holds for all trees of depth n or less, and let $tree(x, L, R)$ be any tree of depth $n + 1$. Then $depth(L)$ and $depth(R)$ must both be n or less, so by the induction hypothesis we have

$$leaves(L) \leq 2^{depth(L)}, \quad leaves(R) \leq 2^{depth(R)}.$$

Hence, by (Lvs2),

$$\begin{aligned} leaves(tree(x, L, R)) &= leaves(L) + leaves(R) \\ &\leq 2^{depth(L)} + 2^{depth(R)} \\ &\leq 2^n + 2^n \\ &= 2^{n+1} = 2^{depth(tree(x, L, R))} \end{aligned}$$

as required. □

2.1 Minimum complexity for a sorting algorithm

We can use the theorem just proved to determine the minimum complexity for any sorting algorithm. Assume that the sorting is achieved solely by means of comparisons between elements. Then the algorithm must follow a path through a binary tree whose leaves are all the possible permutations of the initial list. Each branch point represents a comparison between two elements of the list, with a branch corresponding to each of the two possible orderings. This is shown in Figure 2 for a list containing three elements $[a, b, c]$. The worst-case complexity is given by the length of the longest path down through the tree, in this case 3.

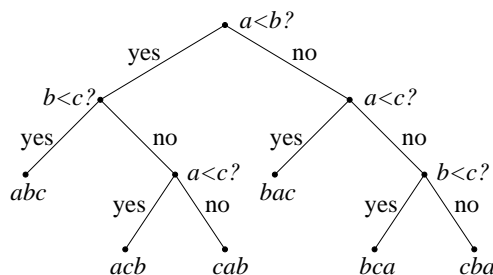


Figure 2: Decision tree for sorting three elements

For n elements to be sorted, the tree has $n!$ leaves (one for each possible ordering of the elements). If the maximum path length is k , then, by our theorem, the number of leaves is at most 2^k , so $n! \leq 2^k$, hence $k \geq \log_2(n!)$. Now k is a measure of the complexity of sorting n elements, so we need an estimate of its size. We have, first,

$$\log_2(n!) = \log_2 n + \log_2(n - 1) + \log_2(n - 2) + \cdots + \log_2(1) < n \log_2 n,$$

and second, assuming n is even,

$$\begin{aligned}
 \log_2(n!) &= \left[\log_2 n + \log_2(n-1) + \cdots + \log_2\left(\frac{n}{2} + 1\right) \right] + \left[\log_2\left(\frac{n}{2}\right) + \log_2\left(\frac{n}{2} - 1\right) + \cdots + \log_2(1) \right] \\
 &> \frac{n}{2} \log_2 \frac{n}{2} + \frac{n}{2} - 1 && \text{(since each term in the left-hand bracket is } > \log_2 \frac{n}{2} \text{ and} \\
 & && \text{each term in the right-hand bracket, apart from the last, is } \geq 1) \\
 &= \frac{n}{2} \left(\log_2 \frac{n}{2} + 1 \right) - 1 \\
 &= \frac{n}{2} \log_2 n - 1
 \end{aligned}$$

Hence we have $\frac{1}{2}n \log_2 n - 1 < \log(n!) < n \log_2 n$, so $\log(n!) = \Theta(n \log n)$. Hence $k \geq \Theta(n \log n)$. This gives us a *lower bound* for the complexity of sorting n elements. It tells us that no sorting algorithm using comparisons between elements can have a worst case complexity better than $\Theta(n \log n)$. Since algorithms of this complexity do exist (e.g., Mergesort, Quicksort), we can say that this is the complexity of the *problem* (not just of this or that algorithm).

Not all sorting algorithms do involve comparisons between elements. If the number of distinct elements that can occur in a list is finite and known in advance (say every element is known to be in the set $\{1, 2, 3, \dots, 100\}$) then we can use a highly efficient **bucket-sort** algorithm: simply set up an array of “buckets”, one for each possible element, and then run through the list putting each element in its correct bucket. Finally you read off the sorted list by running through the buckets in order and retrieving their contents. This can be done in *linear* time, i.e., $\Theta(n)$. However, this does not solve the *general* sorting problem.

3 Text files

We shall take a somewhat simple-minded approach to the definition of text files, which is nonetheless powerful enough to allow the definition of a wide range of word-processing operators.

A text-file may be regarded as a (possibly empty) string of characters, but from a word-processing view, an important concept is *where you are in the file*, i.e., the current cursor position. To model this we represent the file as a *pair* of character strings $\langle L, R \rangle$, where L is the string of characters occurring before the cursor, and R is the string occurring at or after the cursor. For technical convenience, L is given backwards, i.e., running right to left from the position immediately to the left of the cursor to the beginning of the file. This may seem odd, but it actually makes the word-processing operations much easier to specify. The following table shows a selection of files, and their pair representations. In the files the character at the cursor position is underlined. Recall that we use Λ to represent the empty string, containing no characters.

File	L	R	Comments
<u>abc</u> defg	cba	defg	Typical file
<u>a</u> bcde	Λ	abcde	Cursor at start of file
abc <u>d</u> e	dcba	e	Cursor at last character
abcde <u>_</u>	edcba	Λ	Cursor at end of file
<u>_</u>	Λ	Λ	Empty file

We must assume we have a data type *Char*, for the characters that can occur in a file, and *String*, for strings of characters. Then the type *Text* will be identified as $String \times String$. The type *String* is essentially the same as $List(Char)$, but for convenience we introduce a couple of small modifications: first, we'll use Λ rather than $[\]$ to denote the empty string, and second, we'll define $tail(Lambda)$ to be equal to Λ rather than leaving it undefined. The reason for this move is that it enables us to define operations and

prove theorems about them much more simply. Thus the type-specification for *string* is

Type <i>String</i>		
Sorts	Operations	Relations
<i>String</i>	$\Lambda : \textit{String}$	$\textit{head}(\textit{cons}(a, X)) = a$
<i>Char</i>	$\textit{cons} : \textit{Char} \times \textit{String} \rightarrow \textit{String}$	$\textit{tail}(\textit{cons}(a, X)) = X$
	$\textit{head} : \textit{String} \rightarrow \textit{Char} \cup \{\textit{error}\}$	$\textit{head}(\Lambda) = \textit{error}$
	$\textit{tail} : \textit{String} \rightarrow \textit{String}$	$\textit{tail}(\Lambda) = \Lambda$

The functions *Append* and *Reverse* are defined for strings exactly as for lists.

We now define the data-type *Text* to be simply $\textit{String} \times \textit{String}$. Thus the text

The quick brown fox jumps over the lazy dog

(with the cursor at the ‘m’) would be represented as the string pair

$\langle \textit{uj xof nworb kciuq ehT}, \textit{mps over the lazy dog} \rangle$

We go on to define various familiar word-processing operations in the context of the data-type *Text*.

Insertion. Insertion of a character at the cursor position. For example, inserting **h** into **My name is Antony** should result in **My name is Anthony**. We introduce a new operation $\textit{INS} : \textit{Char} \times \textit{Text} \rightarrow \textit{Text}$, with the relation:

$$\textit{INS}(a, \langle L, R \rangle) = \langle \textit{cons}(a, L), R \rangle$$

The insertion operation allows us to build up text-files of any length, starting with the empty file, e.g.,

$$\begin{aligned} \textit{INS}(\textit{T}, \langle \Lambda, \Lambda \rangle) &= \langle \textit{T}, \Lambda \rangle \\ \textit{INS}(\textit{h}, \langle \textit{T}, \Lambda \rangle) &= \langle \textit{hT}, \Lambda \rangle \\ \textit{INS}(\textit{e}, \langle \textit{hT}, \Lambda \rangle) &= \langle \textit{ehT}, \Lambda \rangle \end{aligned}$$

giving us the text-file **The_**.

Overwrite. This has the same type as *insert*, i.e., $\textit{OVR} : \textit{Char} \times \textit{Text} \rightarrow \textit{Text}$, but its effect is a little different, as all users of word-processors and text-editors know:

$$\textit{OVR}(a, \langle L, R \rangle) = \langle \textit{cons}(a, L), \textit{tail}(R) \rangle$$

Compare:

$$\begin{aligned} \textit{INS}(\textit{r}, \textit{He fished a cap}_\textit{ out of the lake}) &= \textit{He fished a car}_\textit{p out of the lake} \\ \textit{OVR}(\textit{r}, \textit{He fished a cap}_\textit{ out of the lake}) &= \textit{He fished a car}_\textit{ out of the lake} \end{aligned}$$

Deletion. There are two forms of deletion commonly used in word-processors. Starting with the file

The quick brown fox jumps over the lazy dog

a deletion operation could result in either of

The quick brown fox jups over the lazy dog
The quick brown fox jmps over the lazy dog

(for example, in Word, the first type of deletion occurs when you press the Del key, the second type occurs when you press the backspace key). We shall call the first type ‘delete’ (DEL), and the second ‘back-delete’ (BKD). They are operations of type $Text \rightarrow Text$. Note that if the cursor is at the end of the file, then *delete* has no effect, whereas if the cursor is at the beginning of the file, then *backdelete* has no effect. The definitions of these operations are

$$\begin{aligned} \text{DEL}(\langle L, R \rangle) &= \langle L, \text{tail}(R) \rangle \\ \text{BKD}(\langle L, R \rangle) &= \langle \text{tail}(L), R \rangle \end{aligned}$$

Cursor movement. We can define functions ‘cursor-left’ and ‘cursor-right’ $\text{CRL}, \text{CRR} : Text \rightarrow Text$ which have the effect of the back-arrow and forward-arrow keys. Note that if the cursor is at the end of the file, then CRR has no effect, while if the cursor is at the beginning of the file then CRL has no effect; thus the two operations are exactly mirror images of one another. The relations are

$$\begin{aligned} \text{CRL}(\langle L, R \rangle) &= \begin{cases} \langle \text{tail}(L), \text{cons}(\text{head}(L), R) \rangle & (\text{if } L \neq \Lambda) \\ \langle L, R \rangle & (\text{otherwise}) \end{cases} \\ \text{CRR}(\langle L, R \rangle) &= \begin{cases} \langle \text{cons}(\text{head}(R), L), \text{tail}(R) \rangle & (\text{if } R \neq \Lambda) \\ \langle L, R \rangle & (\text{otherwise}) \end{cases} \end{aligned}$$

As usual, we can use these definitions to prove theorems which tell us useful or interesting properties of the operations on the data-type. For example, how are the two ‘delete’ operations related to insertion and overwrite? The answers are given by the following theorems. You can verify these empirically using your favourite text editor, but they can also be straightforwardly proved using the relations by which the operations are defined—a more watertight procedure.

3.1 Some word-processing theorems

1. $\text{INS}(a, \text{DEL}(X)) = \text{OVR}(a, X)$. This says that if you delete the character at the cursor and insert a character a , then this is equivalent to overwriting the character at the cursor with a , e.g.:

Commuter Science \rightarrow Computer Science \rightarrow Computer Science

2. $\text{DEL}(\text{INS}(a, X)) = \text{OVR}(X)$. If you insert the new character first and then delete, again this is equivalent to overwriting:

Commuter Science \rightarrow Compmuter Science \rightarrow Computer Science

3. $\text{BKD}(\text{INS}(a, X)) = X$. If you insert a character and then back-delete, this gets you back to where you started:

Commuter Science \rightarrow Compmuter Science \rightarrow Commuter Science

4. $\text{INS}(\text{head}(\text{left}(X)), \text{BKD}(X)) = X$. Here *left* selects the first element of an ordered pair (so $\text{left}(\langle L, R \rangle) = L$). So this theorem says that if you back-delete, and then insert the the character that was initially immediately to the left of the cursor, you get back to where you started:

Commuter Science \rightarrow Computer Science \rightarrow Commuter Science

The proof of the first of these theorems is as follows:

$$\text{INS}(a, \text{DEL}(\langle L, R \rangle)) = \text{INS}(a, \langle L, \text{tail}(R) \rangle) = \langle \text{cons}(a, L), \text{tail}(R) \rangle = \text{OVR}(a, \langle L, R \rangle)$$

The others are left as an exercise.