Abstract Data Types

Florestan Brunck

February 7, 2019

This is a transcription of a lecture given by Luc Devroye on the 28th of February 2019 for the undergraduate class on Data Structures and Algorithms at McGill University (COMP 251). This lecture introduces abstract data types.

1 *Data Structures*

Definition 1. A **abstract data type (ADT)** consists of an *object*, containing the data (e.g., a set, list, graph or a grid), together with a set of *operations* acting on the data object (e.g. SORT, INSERT, DELETE, COPY).

A **data structure** refers to an implementation of an abstract data type. Examples of ADTs include **lists**, **stacks**, **queues** and **deques**, for which the data object consists of an ordered list of elements and the associated operations are suggested in Figure. 1, they will be presented in later sections.

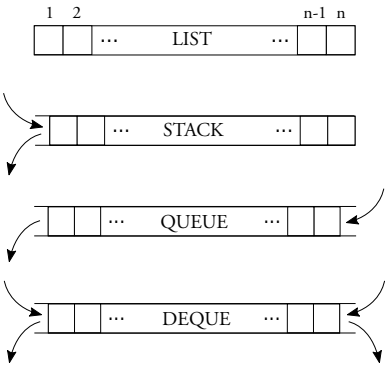

Figure 1: Operations on stacks, queues and deques.

2 *Lists*

A **List** is a linearly ordered set $L = [x_1, x_2, \ldots, x_i, \ldots x_n]$, where the index i indicates the position in the list. The **size** or number of elements in the list L is denoted by $|L|$. The ADT "list" associates two **atomic operations**[1] with such an ordered set:

1. SUBLIST$(L, [i \ldots])$ returns $[x_i, \ldots, x_n]$
   SUBLIST$(L, [\ldots i])$ returns $[x_1, \ldots, x_i]$
   SUBLIST$(L, [i \ldots j])$ returns $[x_i, \ldots, x_j]$
2. CONCATENATION(L_1, L_2) or L_1 & $L1$ returns $[L_1\ L_2]$

The following additional operations (which can be defined using only atomic operations) are usually implemented by default for convenience:

PUSH$(x, L) = L'$ & L where $L' = [x]$

POP(L) = SUBLIST$(L, [2 \ldots])$

ACCESS(i, L) = SUBLIST$(L, [i \ldots i])$

DELETE(i, L) = SUBLIST$(L, [\ldots i-1])$ & SUBLIST$(L, [i+1 \ldots])$

[1] *Atomic*, in this context, signifies that all other operations can be expressed using only atomic operations.

$\text{INSERT}(x, j, L)) = \text{SUBLIST}(L, [\ldots j]) \;\&\; [x] \;\&\; \text{SUBLIST}(L, [j+1\ldots])$

Note: In contrast, the SEARCH operation is usually *not* in the set of defining operations.

2.1 Implementations of Lists

Different data structures can be used to implement the *List* abstract data type, each with their own advantages and drawbacks.

Data Structure	SUBLIST	CONCATENATE
ARRAY	$O(1)$	$\Theta(n)$
LINKED LIST	$\Theta(n)$	$O(1)$
BINARY SEARCH TREE	$O(\log n)$	$O(\log n)$

In the following section we provide more details on the linked-list data structure. The binary search tree implementation will be discussed in a future chapter.

2.2 Linked Lists

Definition 2. A **Singly Linked List** consists of a collection of *cells*, each holding a *data object* and a unique *pointer* to another cell called its *successor*.

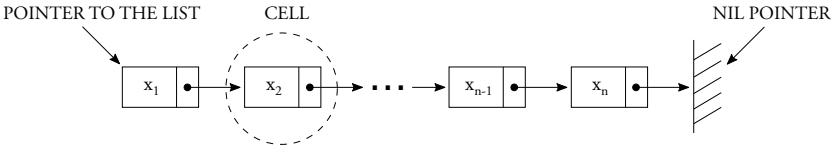

Figure 2: Singly linked list with *endogenous* data storage

Note: The data storage in lists may be either *endogenous*, i.e., each cell contains its data object internally, or *exogenous*, i.e., each cell has a pointer to its data object, stored elsewhere externally (Figure. 3).

Definition 3. A **Doubly Linked List** consists of a collection of *cells*, each holding a *data object* and a set of two *pointers* to other cells, called its *successor* and its *predecessor*.

Note: Both singly linked lists and doubly linked lists can form **circular lists** (see Figure. 5).

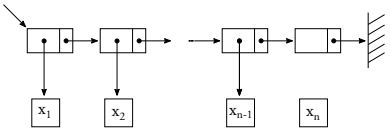

Figure 3: Singly linked list with *exogenous* data storage

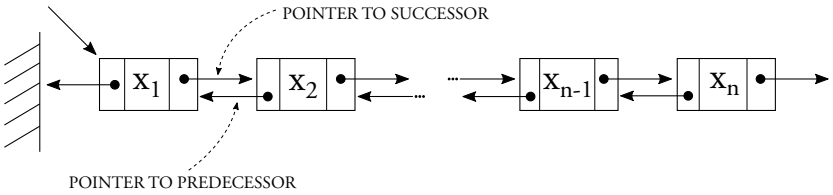

Figure 4: Doubly linked list

There is an additional trick which is often used to ensure that a list is never empty. Namely, one can pad the list with a *header* cell in the front of the list, receiving the initial pointer; and a *trailer* (or *sentinel*) cell receiving the last reference (see Figure. 6). Header cells often hold statistical information about the linked list, such as its size, the maximal value in the list, the average value, etc.

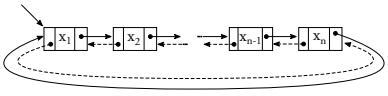

Figure 5: Singly or doubly circular linked lists

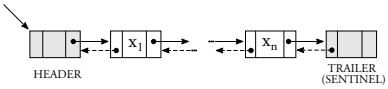

Figure 6: Singly linked list with *exogenous* data storage

2.3 *List-Related ADTs*

Lists are very useful to represent and manipulate a variety of abstract objects.

Example 1: Lists can be used for the *symbolic manipulation of polynomials*. Namely, to each polynomial $P(x) = \sum_{k=0}^{n} a_k \cdot x^k$ we can associate the singly linked list whose data objects consists of the *coefficients* a_i of P. As an example of symbolic computation, consider taking the derivative of P. With regards to our list this amounts to replacing the data object of the i-th cell (i.e., the coefficient a_i) with the new data object $(i \cdot a_i)$ for all i.

Example 2: A related example is the representation of arbitrarily large *unbounded integers* $\sum_{k=0}^{n} a_k \cdot b^k$ in basis b. We can recycle the same idea by replacing the literal x^i associated with our polynomial by the number b^i.

Example 3: A *window manager* can be represented by a linked list as well. Each cell is now associated with a given program (File explorer, Browser, etc.) and its successor cell/program is displayed *behind it*. That way the linear order now keeps track of which program is to be displayed in front of which other programs. The user's operations are reflected by operations on the list, e.g., clicking on a window reroutes the initial pointer to the associated cell and modifies the successor of that cell to be the former initial cell, while changing the successor of its predecessor to its successor. Closing/opening a windows corresponds to the DELETE/CREATE operations.

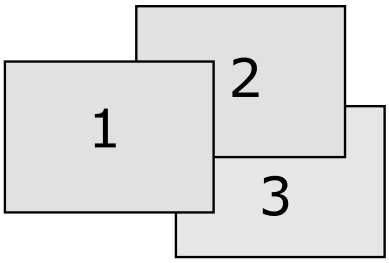

Figure 7: A window manager manipulates a linked list of programs

Example 4: Lists can also be used for *memory management* where the available spaces are linked through the list (see Figure. 8).

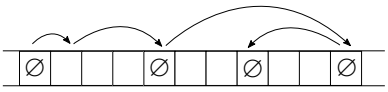

Figure 8: Memory management using linked lists

3 Stacks

Definition 4. The ADT **stack** associates the operations PUSH and POP with a list. Stacks treat the list as a single-ended entity and observe a *first-in, last-out* policy.

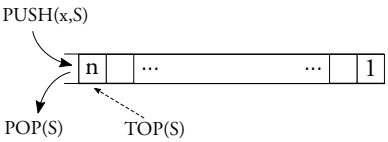

Figure 9: The stack ADT

Additionally, the following operations are usually defined:

TOP reports (but does not remove) the top element

MAKENULL(S) creates an empty stack

EMPTY(S) which returns TRUE if $|S| = 0$

Stacks are easily implemented with *linked lists*, for which all the described operations take *constant time*.

4 Queues

Figure 10: The queue ADT

Definition 5. The ADT **queue** associates the operations ENQUEUE and DEQUEUE with a list. The goal of these operations is to now treat the list as a double-ended object, where ENQUEUE$(x, L) = L \,\&\, [x]$ appends elements at the back of the list (notice the reversal of the order in the concatenation compared to PUSH in a stack), and DEQUEUE assumes the same role as POP does in a stack. Queues are therefore said to observe a *first-in, first-out* policy.

Similarly, the MAKENULL(S) and EMPTY(S) operations can be defined. And likewise, they can easily implemented with *linked lists*, for which all the described operations take *constant time*.

Two different implementations allow us to differentiation the front-end and the rear-end of a linked-list to implement queues. Both are equivalent and allow for constant time operations. In the *single pointer implementation*, we keep one sole pointer to the list and ask the for the last cell in the list (the rear-end element) to have point to the first cell. In the *double-pointer implementation* we simply keep two pointers to the front and the end of the list.

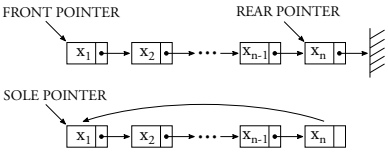

Figure 11: The single pointer and double-pointer implementations of a queue.

5 Deques

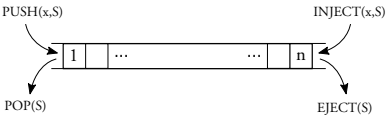

Figure 12: The deque ADT

Definition 6. The ADT **deque**[2] associates two additional operations to the stack operations (PUSH and POP) with a list: namely the operations INJECT and EJECT which assume symmetrical roles with regard to the tail of the list[3].

[2] The word *deque* is a shortening of "doubly ended queue"

[3] Try to write them out using atomic operations

Deques are usually implemented with *doubly linked circular lists*, for which all the described operations take *constant time*.

6 Stacks and Queues in Use

Example 1: To *evaluate/parse expressions*, stacks are particularly useful. A good example is parenthesis parsing, where left parenthesis are pushed onto a stack and popped as we read right parenthesis while reading an expression from the left to the right. To check whether the expression is correctly parsed or not we can simply have a program return an error if we pop an empty stack over as we scan our string or if EMPTY (S) returns FALSE once we reach the end of the expression.

Example 2: Another use of stacks is the *computation of arithmetic expressions*. While the regular way to present a computation requires parenthesis to parse the order in which operators need to be applied (*infix* notation), we may also use *postfix* notation, for which the operands and operators are written in the order they need to be applied and such that each operator unambiguously operates on the last two operands which have have not yet been operated on (read the side-note to the right for an example). Assuming an expression is given in *postfix*[4] notation, there is a simple algorithm to evaluate it which relies on a stack collecting operands.

[4] As an exercise, provide an algorithm to convert an expression given in infix notation into postfix notation, e.g. $5 \times (6+2) - 12/4$ becomes $5\,6\,2\,+\,\times\,1\,2\,4\,/\,-$

```
EVALUATE(Q)
 1  // The input Q is a queue holding the postfix expression
 2  MAKENULL(S)
 3  while |Q| > 0
 4      x ⟵ DEQUEUE(Q)
 5      if x ∈ {operator} then
 6          PUSH(x, S)
 7      else // If x ∈ {operands}
 8          a ⟵ POP(S)
 9          b ⟵ POP(S)
10          c ⟵ b x a
11          PUSH(c, S)
12  RESULT ⟵ POP(S)
```

Example 3: For graph and more specifically *tree traversal*, we will see in subsequent lectures that one can implement the Depth-First-Search (DFS) algorithm non recursively using a stack.

Example 4: Stacks also turn out to be the right object to manipulate function calls and nested objects, and in particular *recursion*. The

call stack handles the execution of programs and functions and is arguably the most fundamental use of stacks in computer science. Each time a function makes a call to another function, we PUSH it on the stack. If that function calls other sub-functions they become subsequent items to be PUSHed on the stack, which - given the well-suited first-in-last-out policy of stacks - will need to be handled first before they can return their values (as they are POPed from the stack) to the original caller/function, which can then be POPed itself. As an example, in a recursive problem, a large initial problem is pushed on the stack and will trigger calls to smaller subsequent sub-problems. The smaller the sub-problem, the later the call, the last items added to the stack being the sub-problems that can directly be handled.

Example 5: Stacks also yield easy, elegant solutions to a variety of problems. For example, if one were to escape a maze, keeping a stack to record the successive turns while marking the explored alleys gives a successful strategy to reach the exit. Figure 13 shows the first few steps of a player and the corresponding stack. We also give a simple algorithm to record the escape path in a maze.

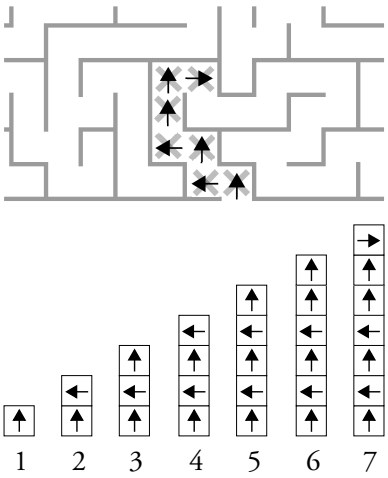

Figure 13: Using a stack to find the exit in a maze.

```
ESCAPE-PATH(Maze, Start)
1  // Start is a Start Node corresponding to the given entrance
2  MAKENULL(S), PUSH(Start, S), MARK(Start)
3  while TOP(S) ≠ Exit
4      if TOP(S) has an unmarked neighbour v
5          PUSH(v, S), MARK(v)
6      else // If there are no unmarked neighbours
7          POP(S)
8  // Upon halting, S holds the escape path from the Start to the Exit.
```