Lecture Notes Datastructures, Lecture 1

Sebastian Altmeyer

2018/19, Period 2

1 Data Types, Data Structures, Abstract Data Types

Each programming language features a set of basic, built-in data types, such as those listed in Table 1. Nearly all programming language provide native support for these data types, the programming language C features additional basic data types such as *void*, and *signed* and *unsigned* variations of ints and floating point values, and floats with varying precision. Derived data types then represent data types based on the primitive types, such as arrays, structs, enumerations, but also pointers are considered derived data types. Programming languages still provide built-in support for these data types, but derived data types can not exist on their own.

Table 1: Common Basic Data Types

Type	Values	Operators (selection)
Int	$\approx \mathbb{Z}$	$=, -, /, *$
Bool	$\{0, 1\}$	$\wedge, \vee, \neg$
Float	$\approx \mathbb{R}$	$=, -, /, *$
Char	$\{a, \ldots, z, A, \ldots, Z, 0, \ldots, 9, =, -, \&, @ \ldots\}$	

Remark. *The definition of data types, data structures, signatures etc. are not written in stone, but the terms are used very differently depending on the contexts, and sometimes even interchangeably. Similarly, the classification into primitive and composite data types is not absolute, as well as the notation itself.*

Data structures, on the other hand, refer to a particular way of organizing data of any type. Data structures describe collections of other data types including operations on these collections, and are typically provided via libraries. In this regard, an array can be considered both a data type and a data structure. Due to the importance of arrays both in programming languages and as a basis for other data structures, we first provide explain arrays in more detail.

1.1 Array

An array refers to a contiguous part of memory reserved to store data of a given type. For instance, the declaration

```
char A[5];
```

refers to a memory of size $5 *$ size(char). In C, an array is simply implemented by a pointer to the very first element of the array. The operator $[i]$ can be used to access the ith element of an array (write and read access), unless used within a declaration, of course. The access `A[i]` is simply translated to a memory access to memory address A $+\ i *$ size(char). The direct and immediate access to an arbitrary element of a data structure is rather uncommon. This uncommon benefit of a direct access comes at the cost of fixed sized: The size of an array is statically determined and cannot be changed at run-time.

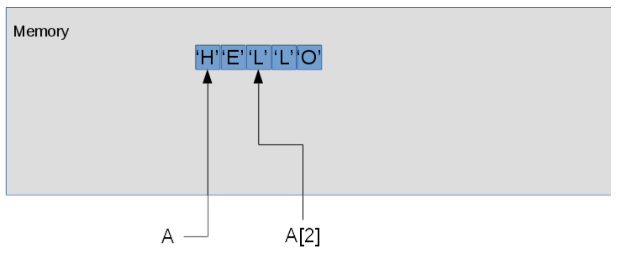

Figure 1: Example of an Array A of size 5. The data of an array is stored contiguously in memory.

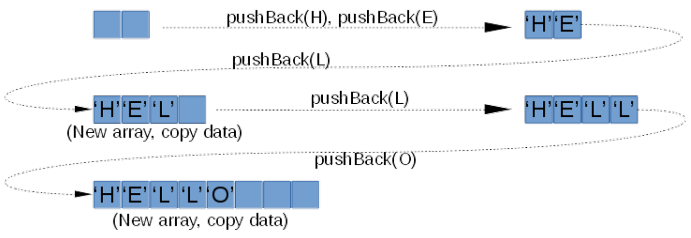

Figure 2: Example of an Array A of size 5

Dynamic Array/Vector Arrays of dynamic size are not natively supported in C or $C++$, but are provided by a library. To avoid confusion, we will refer to dynamic arrays only as vectors. In addition to operators on static arrays, a vector provides two additional operations, *pushBack* and *popBack*. Yet, vectors are implemented using (static) arrays. When the size of the current array is sufficient to store all data, the behavior of array and vector are the same. The operation *pushBack* may require more space than is currently allocated. In this case, the vector is dynamically resized to accommodate the new data, as shown in Figure 2. Such a resize operation is implemented in three steps, (i) a new static array of twice the required size is allocated, (ii) the data is copied from the old array to the new array, (iii) the pointer is reassigned to point to the new array. See Figure 3 for a graphical representation of these three steps. We assume for the sake of simplicity that vectors

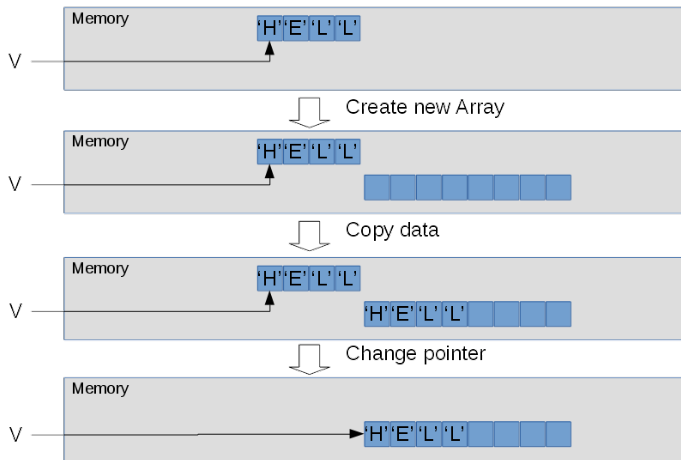

Figure 3: Resizing a vector by creating a new static array and copying data

are not automatically resized once they shrink below a threshold. Such an extension is, however, trivial to implement and to analyze.

1.2 Linked Lists

Linked lists provide a simple data structure with variable size. The elementary component of a list is a tuple consisting of an element of the data type of the list, and a pointer to the next list element. A list of n chars therefore requires a memory of $n * (\text{size(char)} + \text{size(pointer)})$ blocks. Lists do not require a contiguous chunk of memory, but can be scattered around in memory, see Figure 4. The list is represented by a pointer to the

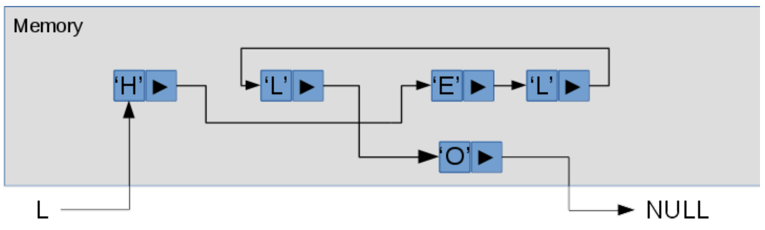

Figure 4: Example of a list storing the chars 'HELLO'.

very first element, with a pointer to NULL representing an empty list. The pointer of the last list element also points to NULL. Appending an element to the list is implemented by reserving memory for a new element, changing the pointer of the currently last list element to the newly created element. The steps to remove an element from in-between the list are shown in Figure 5. Other common list operations include prepending an element, removing an element, searching for an element or reversing the list. The direct access of the ith element, as provided by an array, is not available in such an implementation of a list; it is always necessary to traverse the list first. An important variant of a linked list is a doubly-linked list as shown in Figure 6. The struct

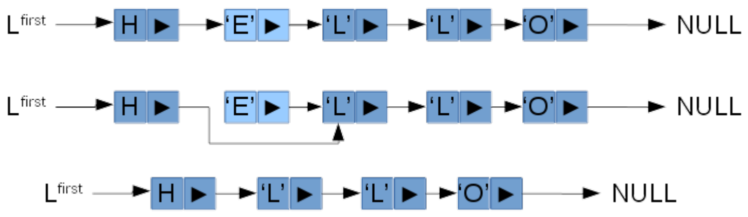

Figure 5: Removal of the second element of the list containing 'HELLO'.

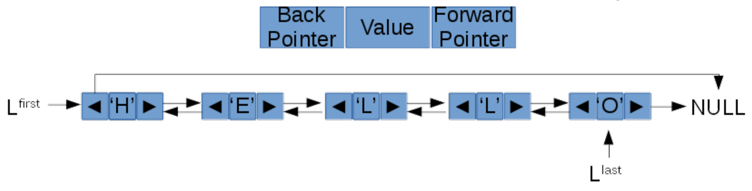

Figure 6: Example of a doubly-linked list storing the chars 'HELLO'.

constituting an element of a doubly-linked list contains a value and two pointers, one pointing to the preceding and one pointing to the next list element. Furthermore, doubly-linked lists feature an additional pointer to point to the very last list element. A pointer to the last element may also be used in singly-linked lists, although it is less common.

1.3 Abstract Data Types

Abstraction is fundamental theme of computer science, which allows to argue about entities without having to define all of their properties. In our previous examples, lists and arrays, we have introduced the data structure based on their implementation. An abstract data type now separates the behavior of a data type from its implementation. Examples of abstract data types are listed in Table 2. This separation also allows us to provide several implementations of an abstract data type, for instance with different memory requirements or runtime performances. Linked list and doubly-linked lists, for instance, can implement the same operations, but the

Table 2: Selection of Abstract Data Types and the defining Operations

Type	Operators (selection)
List	append, prepend, remove
Stack	top, pop/pull, push
Queue	enqueue, dequeue
Array	[]
Vector	pushBack, popBack, []

access to the predecessor of a list element requires traversal through all list element in case of singly-linked list, and a single access in case of a doubly-linked list. Abstract data types are therefore purely defined on their semantics/behavior manifested in the operations that data structure must provide. A selection of abstract data structures are listed in Table 2. Certain operations, such as *isEmpty*, *size*, or *create* are realized by nearly all data structures and therefore not listed. We note that the concept of an abstract data type is again not well defined.

1.4 Stack

Next, we study the abstract data type *stack*. A stack is a common concept – think of a stack of books – and a fundamental data structure in computer science. Elements of stack are accessed in LIFO order, last-in, first-out. A stack features two main operations, *push*, i.e., putting new elements on the stack, and *pull*, also called *pop*, i.e., removing the top element from the stack, see Figure 7. Most implementations also feature an *isEmpty* and

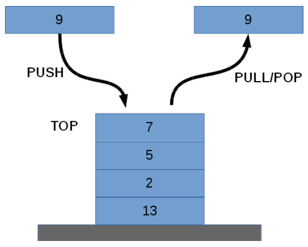

Figure 7: Example of a stack of integers and the operations on it.

top/peek operation, the first one is self-explanatory, the second one provides access to (peeks at) the top element without removing it from the stack.

Stacks are widely used, for instance to implement undo/redo feature in editors, function call mechanism in C, or to evaluate expressions. Algorithm 1 provide a simplified version of such an expression evaluation, which has been common in older table calculators.

Algorithm 1 Summation using Stacks

```
1: function SUM(Stack s)
2:     if s.isEmpty() then
3:         return 0
4:     else
5:         int value = s.peek()
6:         s.pull()
7:         return value + sum(s)
```

Stacks are abstract data types and do not define an implementation directly. Stacks can be implemented using arrays in case of bounded stack size, or using vectors in case of unbounded stack size. The vector implementation requires an additional pointer that points to the next free element, i.e., the element after the top of the stack. An example of such an implementation is shown in Figure 8, including two operations, *push* and *pop*. Alternatively, stacks can be implemented via linked lists, as shown in Figure 9. The top element of the stack is stored as first element within the list, to enable quick access to the element.

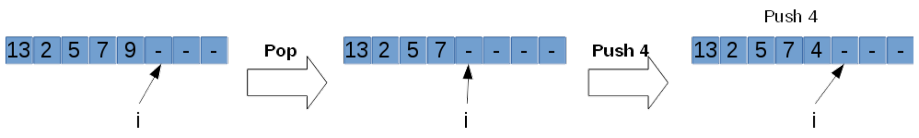

Figure 8: Stack implementations using Arrays.

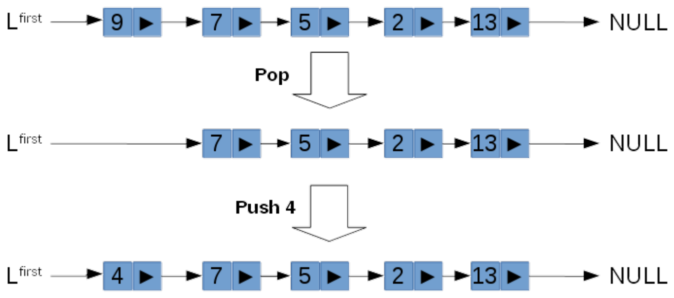

Figure 9: Stack implementations using Linked Lists.

1.5 Queue

Queues are another common data structure, which implement a FIFO (first-in first-out) order, see Figure 10. The defining operations of a queue are *enqueue* and *dequeue*. Most implementations also feature an *isEmpty*

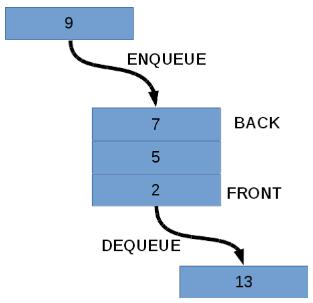

Figure 10: Example of a queue of integers and the operations on it.

and *top* operation, the first one is self-explanatory, the second one provides access to the top element without removing it from the queue.

Queues can be implemented using arrays or vectors. The array implementation requires two additional pointer, one to the first element of the queue, and one to the last element. The index of the last element may be smaller than the index of the first element. This does not indicate that the array is full, but rather that the queue wraps around the end of the array and starts again at the beginning. Such a case is shown in Figure 11, including the two operations, *enqueue* and *dequeue*. Alternatively, queues can be implemented via linked lists, as shown in Figure 12.

An efficient implementation requires either doubly-linked lists as shown in the figure, or a singly-linked list with additional pointer to the last element. In this case, however, the list elements must be in reverse order and not in order depicted in Figure 12

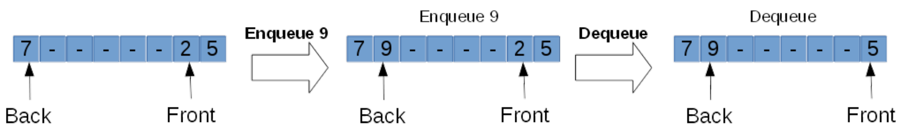

Figure 11: Queue implementations using Arrays.

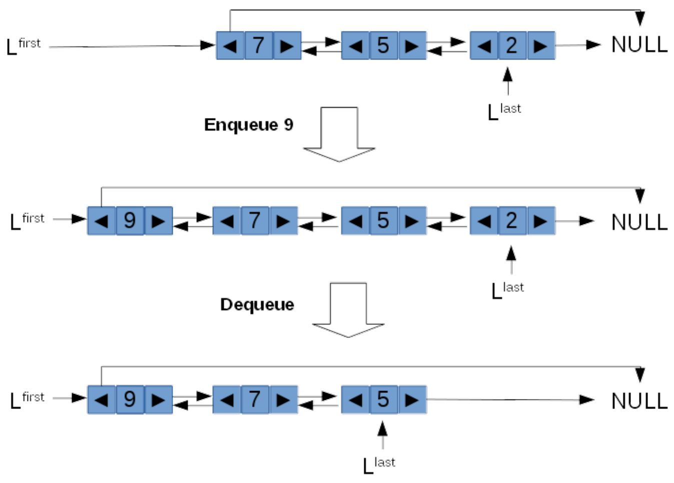

Figure 12: Queue implementations using Doubly-Linked Lists.