3. List

If your program needs to store a few things—numbers, payroll records, or job descriptions for example—the simplest and most effective approach might be to put them in a list. Only when you have to organize and search through a large number of things do more sophisticated data structures like [**search trees**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-search-tree) become necessary.

 Many applications don't require any form of search, and they do not require that an ordering be placed on the objects being stored. Some applications require that actions be performed in a strict chronological order, processing objects in the order that they arrived, or perhaps processing objects in the reverse of the order that they arrived. For all these situations, a simple list structure is appropriate.

This chapter describes representations both for lists and for two important list-like structures called the [**stack**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-stack) and the [**queue**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-queue).

We begin by defining an [**ADT for lists**](https://traky.cs.hut.fi/Books/CSE-A1141/html/ListADT.html#listadt). Two implementations for the list ADT—the [**array-based list**](https://traky.cs.hut.fi/Books/CSE-A1141/html/ListArray.html#listarray) and the [**linked list**](https://traky.cs.hut.fi/Books/CSE-A1141/html/ListLinked.html#listlinked)

## 3.2 The list ADT

define a [**list**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-list) to be a finite, ordered sequence of data items known as [**elements**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-element). "Ordered" in this definition means that each element has a position in the list. The operations defined as part of the list [**ADT**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-adt) do not depend on the elemental [**data type**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-data-type).

A list is said to be [**empty**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-empty) when it contains no elements. The number of elements currently stored is called the [**length**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-length) of the list. The beginning of the list is called the [**head**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-head), the end of the list is called the [**tail**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-tail).

Now we can define the ADT for a list object in terms of a set of operations on that object. We will use an interface to formally define the list ADT. List defines the member functions that any list implementation inheriting from it must support, along with their parameters and return types.

**container class**

A [***data structure***](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-data-structure) that stores a collection of [***records***](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-record). Typical examples are arrays, [***search trees***](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-search-tree), and [***hash tables***](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-hash-table).

The current position is where any action such as insertion or deletion will take place.

**iterator**

In a [***container***](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-container) such as a List, a separate class that indicates position within the container, with support for [***traversing***](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-traversal) through all [***elements***](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-element) in the container.

**for** (L.moveToStart(); !L.isAtEnd(); L.next()) {

it = L.getValue();

doSomething(it);

}

*// Return true if k is in list L, false otherwise*

**bool** find(List& L, **int** k) {

**for** (L.moveToStart(); !L.isAtEnd(); L.next())

**if** (k == L.getValue()) **return** true; *// Found k*

**return** false; *// k not found*

}

## 3.8 Stacks

The [**stack**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-stack) is a list-like structure in which elements may be inserted or removed from only one end. While this restriction makes stacks less flexible than lists, it also makes stacks both efficient (for those operations they can do) and easy to implement. Many applications require only the limited form of insert and remove operations that stacks provide. In such cases, it is more efficient to use the simpler stack data structure rather than the generic list. For example, the **[freelist](https://traky.cs.hut.fi/Books/CSE-A1141/html/Freelist.html" \l "freelist)** is really a stack.

Note that one implication of the LIFO policy is that stacks remove elements in reverse order of their arrival.

The accessible element of the stack is called the top element. Elements are not said to be inserted, they are [**pushed**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-push) onto the stack. When removed, an element is said to be [**popped**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-pop) from the stack. Here is a simple stack [**ADT**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-adt).

The two approaches presented here are the [**array-based stack**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-array-based-stack) and the [**linked stack**](https://traky.cs.hut.fi/Books/CSE-A1141/html/StackLinked.html#stacklinked), which are analogous to array-based and linked lists, respectively.

## 3.8.2. Array-Based Stacks[¶](https://traky.cs.hut.fi/Books/CSE-A1141/html/StackArray.html#array-based-stacks)

StackArray must be declared of fixed size when the stack is created.

Variavble top: act like a current position value, also indicates the number of element currently

The array-based stack implementation is essentially a simplified version of the array-based list. The only important design decision to be made is which end of the array should represent the top of the stack.

1 pop: position 0; inefficient, cost theta(n)

2. pop: positon: n-1 theta(1)

For an empty array: pop 0 (first free position) or -1

**boolean** push(Object it) {

**if** (top >= maxSize) **return** **false**;

stackArray[top++] = it;

**return** **true**;

}

Object pop() {

**if** (top == 0) **return** **null**;

**return** stackArray[--top];

}

## 3.9 Linked Stacks

The linked stack implementation is quite simple. Elements are inserted and removed only from the head of the list. A header node is not used because no special-case code is required for lists of zero or one elements.

*// Put "it" on stack*

**boolean** push(Object it) {

top = **new** Link(it, top);

size++;

**return** **true**;

}

*// Remove "it" from stack*

Object pop() {

**if** (top == **null**) **return** **null**;

Object it = top.element();

top = top.next();

size--;

**return** it;

}

### 3.9.2. Comparison of Array-Based and Linked Stacks

All operations for the array-based and linked stack implementations take constant time, so from a time efficiency perspective, neither has a significant advantage.

The array-based stack must declare a fixed-size array initially, and some of that space is wasted whenever the stack is not full. The linked stack can shrink and grow but requires the overhead of a link field for every element.

When implementing multiple stacks, sometimes you can take advantage of the one-way growth of the array-based stack by using a single array to store two stacks.

One stack grows inward from each end hopefully leading to less wasted space. However, this only works well when the space requirements of the two stacks are inversely correlated. In other words, ideally when one stack grows, the other will shrink. This is particularly effective when elements are taken from one stack and given to the other. If instead both stacks grow at the same time, then the free space in the middle of the array will be exhausted quickly.

## 3.10. Freelists