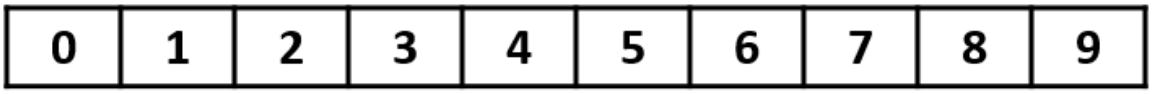
Array Lists

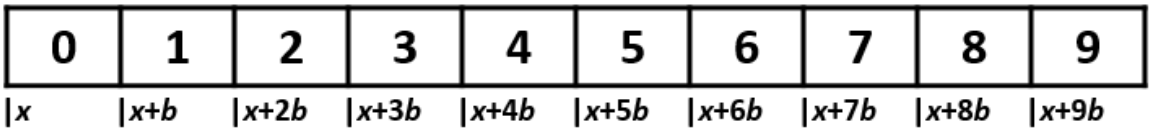
Step 1

The first data structure that will be covered in this text is one of the fundamental data structures of computer science: the **array**.
An **array** is a homogeneous data structure: all elements are of the same type (int, string, etc). Also, the elements of an **array** are stored in adjacent memory locations. Below is an example of an **array**, where the number inside each cell is the index of that cell, using 0-based indexing (i.e., the first element is at index 0, the second element is at index 1, etc.).



Because each cell has the same type (and thus the same size), and because the cells are adjacent in memory, it is possible to quickly calculate the address of any array cell, given the address of the first cell.

Say we allocate memory for an **array** of n elements (the total number of cells of the array must be defined beforehand), where the elements of the **array** are of a type that has a size of b bytes (e.g. a C++ `int` has a size of 4 bytes), and the resulting **array** is allocated starting at memory address x. Using 0-based indexing, the element at index $i = 0$ is at memory address x, the element at index $i = 1$ is at memory address $x + b$, and the element at index i is at memory address $x + bi$. Below is the same example **array**, where the number inside each cell is the index of that cell, and the number below each cell is the memory address at which that cell begins.



Because of this phenomenon of being able to find the memory address of any i-th element in constant time (and thus being able to access any i-th element in constant time), we say that **arrays** have **random access**. In other words, we can access any specific element we want very quickly: in O(1) time.

Step 2

EXERCISE BREAK: You create an **array** of integers (assume each integer is exactly 4 bytes) in memory, and the beginning of the **array** (i.e., the start of the very first cell of the array) happens to be at memory address 1000. What is the memory address of the start of cell 6 of the **array**, assuming 0-based indexing (i.e., cell 0 is the first cell of the **array**)?

To solve this problem please visit https://stepik.org/lesson/28868/step/2

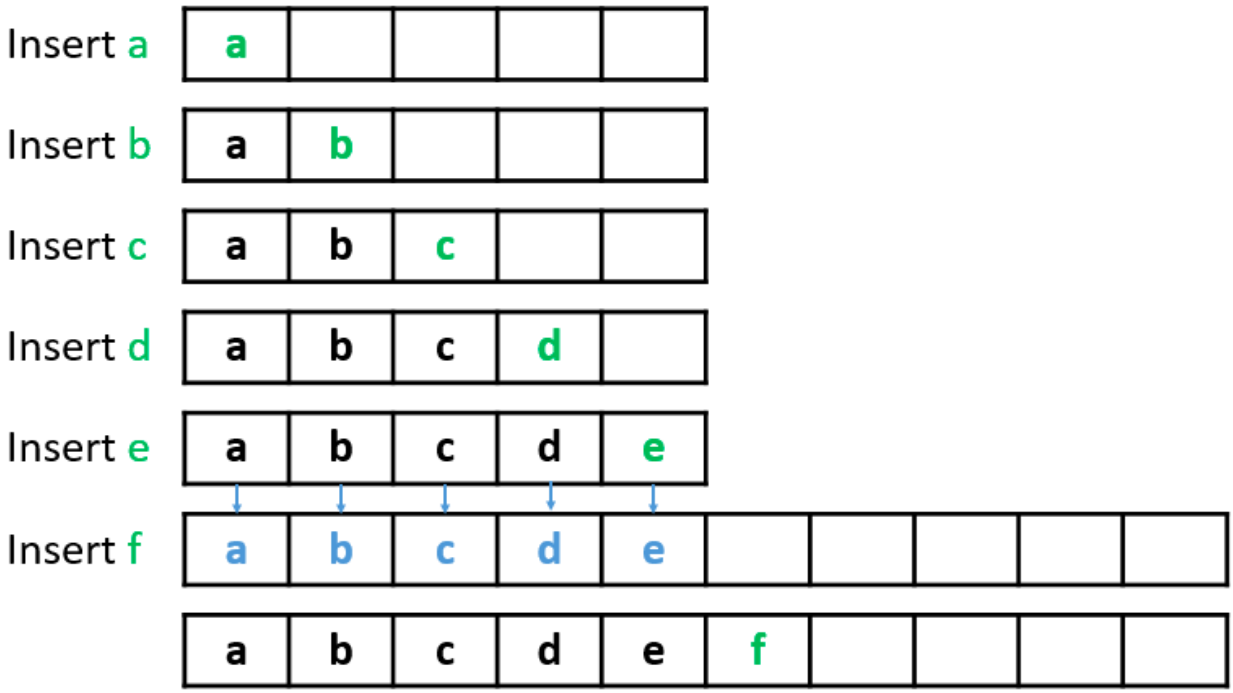
Step 3

For our purposes from this point on, we will think of arrays specifically in the context of using them to store contiguous lists of elements: an **Array List**. In other words, we'll assume that there are no empty cells between elements in the array (even though an array in general has no such restriction). As a result, we will assume that a user can only add elements to indices between 0 and n (inclusive), where n is the number of total elements that exist in the list prior to the new insertion.

As can be inferred, we as programmers don't always know exactly how many elements we want to insert into an **Array List** beforehand. To combat this, many programming languages implement **Array Lists** as "dynamic": they allocate some default "large" amount of memory initially, insert elements into this initial array, and once the array is full, they create a new larger array (typically twice as large as

the old array), copy all elements from the old array into the new array, and then replace any references to the old array with references to the new array. In C++, the "dynamic array" is the `vector` data structure.

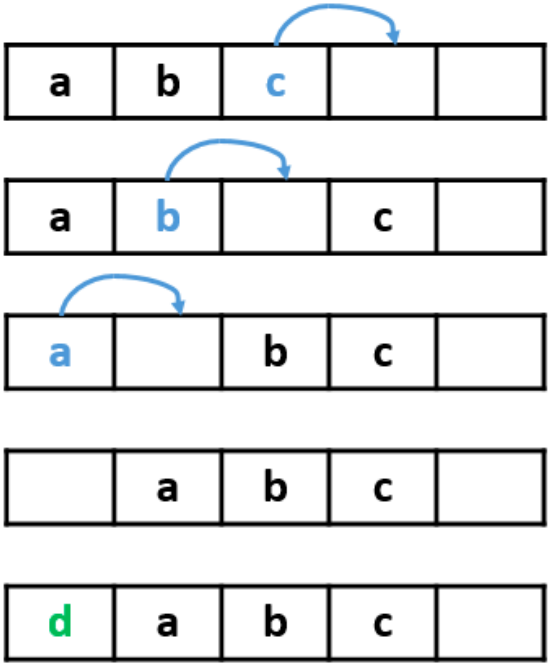
Below is an example of this in action, where we add the letters a-f to an **Array List** backed by an array initialized with 5 cells, where each insertion occurs at the end of the list (notice how we allocate a new array of twice the old array's length and copy all elements from the old array into the new array when we want to insert f):



STOP and Think: Array structures (e.g. the array, or the Java `ArrayList`, or the C++ `vector`, etc.) require that all elements be the same size. However, array structures can contain strings, which can be different lengths (and thus different sizes in memory). How is this possible?

Step 4

Based on the previous example, it should be clear that inserting at the end of an **Array List** for which the backing array is not full, assuming I know how many elements are currently in it, is constant time. However, what if I want to insert at the beginning of an **Array List**? Unfortunately, because of the rigid structure of an array (which was vital to give us random access), I need to move a potentially large number of elements out of the way to make room for the element I want to insert. Below is an example of inserting d to the beginning of an **Array List** with 3 elements previously in it:



As can be seen, even though the best-case time complexity for insertion into an array is O(1) (which we saw in the previous step), the worst-case time complexity for insertion into an **Array List** is **O(*n*)** because we could potentially have to move all *n* element in the array (or as in the case of the previous step, we may have to allocate an entirely new array and copy all *n* elements into this new array).

Step 5

Formally, below is the pseudocode of insertion into an **Array List**. Recall that we assume that all elements in the array must be contiguous. Also, note that the pseudocode uses 0-based indexing.

```
insert(element, index): // inserts element into array and returns True on success (or False on failure)
    // perform the required safety checks before insertion
    if index < 0 or index > n:   // invalid indices
        return False

    if n == array.length:        // if array is full
        newArray = empty array of length 2*array.length
        for i from 0 to n-1:     // copy all elements of old array to new array
            newArray[i] = array[i]
        array = newArray         // replace old array with new array

    // perform the insertion algorithm
    if index == n:               // insertion at end of array
        array[index] = element   // perform insertion

    else:                        // general insertion
        for i from n-1 to index: // make space for the new element (if insertion not at the end)
            array[i+1] = array[i]
        array[index] = element   // perform insertion

    n = n+1                      // increment number of elements
    return True
```

Also, below is the pseudocode of finding an element in an arbitrary array (i.e., the elements are in no particular order). Again, the pseudocode is using 0-based indexing.

```
find(element): // returns True if element exists in array, otherwise returns False
    for i from 0 to n-1:        // iterate through all n elements
        if array[i] == element: // if we have a match, return True
            return True
    return False                // if we reach this, we checked all n elements, so return False
```

Step 6

EXERCISE BREAK: What is the worst-case time complexity for an "insert" operation in an arbitrary **Array List** (i.e., we know nothing about the elements it contains)?

To solve this problem please visit https://stepik.org/lesson/28868/step/6

Step 7

EXERCISE BREAK: What is the worst-case time complexity for a "find" operation in an arbitrary **Array List** (i.e., we know nothing about the elements it contains)?

To solve this problem please visit https://stepik.org/lesson/28868/step/7

Step 8

CODE CHALLENGE: Finding an Element in an Arbitrary Array List

Given an arbitrary **Array List** of integers *arr* and an integer *n*, return true if *n* appears in *arr*, or false if it does not.

Sample Input:

```
5 9 4 1 0 2 3
0
```

Sample Output:

```
true
```

To solve this problem please visit https://stepik.org/lesson/28868/step/8

Step 9

As you may have inferred, the worst-case time complexity to find an element in an arbitrary **Array List** is **O(*n*)** because there is no known order to the elements, so we have to individually check each of the *n* elements.

However, what if our **Array List** happened to be sorted? Could we exploit that, along with the feature of random access, to speed things up? We will introduce an algorithm called **Binary Search**, which allows us to exploit random access in order to obtain a worst-case time complexity of **O(log *n*)** for searching in a **sorted Array List**. The basic idea is as follows: because we have random access, compare the element we're searching for against the middle element of the array. If our element is less than the middle element, our element must

exist on the left half of the array (if at all), so repeat the search on the left half of the array since there is no longer any point to search in the right half of the array. If our element is larger than the middle element, our element must exist on the right half of the array (if at all), so repeat on the right half of the array. If our element is equal to the middle element, we have successfully found our element.

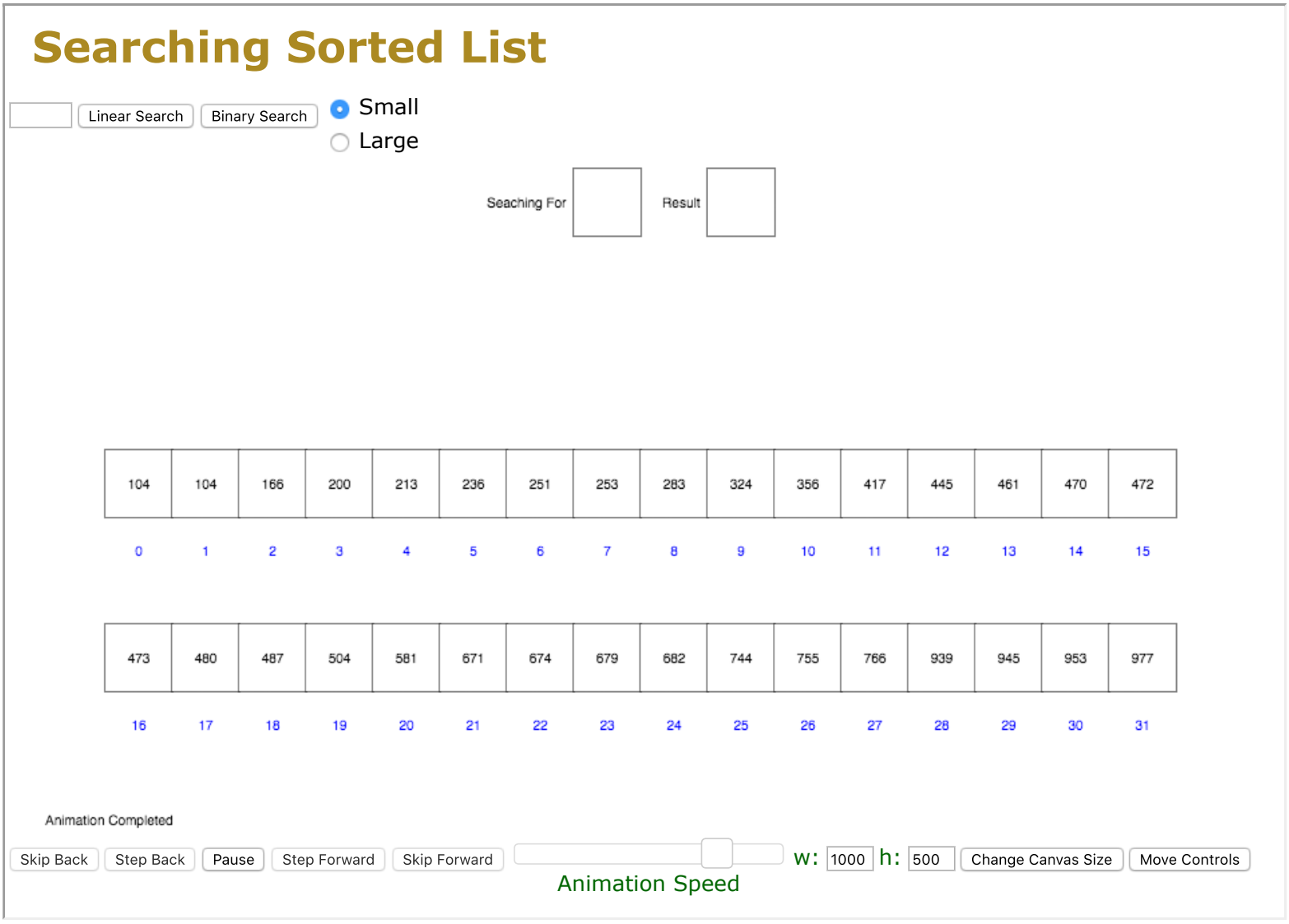
```
BinarySearch(array, element): // perform binary search to find element in array
    L = 0 and R = n-1             // initialize "left" and "right" indices

    loop infinitely:
        if L > R:                 // "left" index is larger than "right" index, so we failed
            return False
        M = the floor of (L+R)/2 // compute middle index

        if element == array[M]:  // if element equals middle element, we found our element
            return True
        if element > array[M]:   // if element is larger than middle element, "recurse" right
            L = M+1
        if element < array[M]:   // if element is smaller than middle element, "recurse" left
            R = M - 1
```

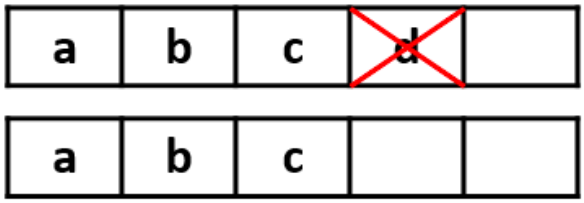
Step 10

The Binary Search algorithm is much easier to follow when shown visually, so below is a visualization created by David Galles at the University of San Francisco.

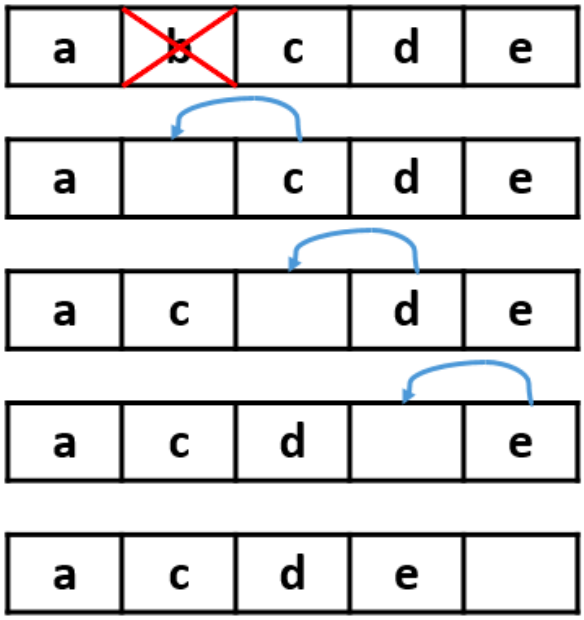


Step 11

Just like with insertion, removal at the end of our list of elements is very fast: we simply remove the element at the *n*-th index of our backing array, which is a constant-time operation. Below is an example of removing the last element from an **Array List**:



However, also just like with insertion, removal at the beginning of our list of elements is very slow: we remove the element at index 0 of our backing array, which is a constant-time operation, but we then need to move all of the elements left one slot, which is an O(*n*) operation overall. In the example below, instead of removing from the very beginning of the array, we remove the second element, and we see that we still have a worst-case O(*n*) operation when we have to move all of the elements left. Note that this "leftward move" requirement comes from our restriction that all elements in our array must be contiguous (which is necessary for us to have random access to any of the elements).



STOP and Think: When we remove from the very beginning of the backing array of an **Array List**, even before we move the remaining elements to the left, the remaining elements are all still contiguous, so our restriction is satisfied. Can we do something clever with our implementation to avoid having to perform this move operation when we remove from the very beginning of our list?

Step 12

EXERCISE BREAK: You are given an **Array List** with 100 elements, and you want to remove the element at index 7 (0-based indexing). How many "left-shift" operations will you have to perform on the backing array?

Step 13

Formally, below is the pseudocode of removal from an **Array List**. Recall that we assume that all elements in the array must be contiguous. Also, note that the pseudocode uses 0-based indexing.

```
remove(index): // removes element at position "index" in the array
    // check for valid index
    if index < 0 or index >= n:  // invalid indices
        return False

    // perform the removal algorithm
    clear array[index]
    if index < n-1:              // if we didn't remove from the very end of the list
        for i from index to n-2: // shift all elements left one slot
            array[index] = array[index+1]
        clear array[n-1]         // not technically necessary since it'll be overwritten next
insert

    n = n-1                      // increment number of elements
    return True
```

Step 14

In summation, **Array Lists** are great if we know exactly how many elements we want and if the data is already sorted, as finding an element in a sorted **Array List** is O(log *n*) in the worst case and accessing a specific element is O(1). However, inserting into an **Array List** is O(*n*) in the worst case and finding an element in a non-sorted **Array List** is O(*n*). Also, if we don't know exactly how many elements we want to store, we would need to allocate extra space in order to avoid having to rebuild the array over and over again, which would waste some space.

In general, all data structures have applications in which they excel as well as cases in which they fail, and it is up to the programmer to keep in mind these trade-offs when choosing what data structures to use.