Data Structures and Algorithm

Moazzam Ali Sahi

Room # B1, B-Block

COMSATS University Islamabad, Lahore Campus

Agenda

- Introduction to the course
- What is CDF?
- C-Programing Language (Recap)

Course Description File (CDF)

1	Course Title	Data Structure and Algorithm
2	Course Code	CSC 211
3	Credit Hours	4 (3,1)
4	Prerequisites	CSC 103
5	Semester	4^{th}
6	Resource Person/Lab Engineer	Moazzam Ali Sahi
7	Contact Hours (Theory)	3 hours per week
8	Contact Hours (Lab)	3 hours per week
9	Office Hours	2:00 PM to 4:00 PM (weekdays)

Course Description

- This course includes the basic foundations in of data structures and algorithms. This course covers concepts of various data structures like stack, queue, list, tree and graph. Additionally, the course includes idea of sorting and searching

Course Objective

- To introduce data abstraction and data representation in memory
- To describe, design and use of elementary data structures such as
  - Stack
  - Queue
  - linked list
  - Tree
  - Graph
- To discuss decomposition of complex programming problems into manageable sub- problems
- To introduce algorithms and their complexity

Course Learning Outcomes (CLO)

Theory

- **CLO-1:** Employ linear data structures to solve computing problems. (**Applying**)
- **CLO-2:** Use non-linear data structures to solve computing problems. (**Applying**)
- **CLO-3:** Analyze the time complexity of various algorithms. (**Analyzing**)

Lab

- **CLO-4:** Implement data structures and algorithms. (**Applying**)
- **CLO-5:** Develop a project using appropriate data structures in a team environment. (**Creating**)

Syllabus / Theory Course Outline

- 1. Data Structure:
  - Overview,
  - Importance
  - Classification
  - Operations and
  - Abstract Data Types.
- 2. Static List; Dynamic List:
  - Single Linked Lists
  - Circular Linked List
  - Doubly Linked List
- 3. Stack:
  - Concept
  - Applications.
- 4. Queue:
  - Concept
  - Types
  - Applications.

- 5. Tree:
  - Concept & Terminology
  - Traversal Algorithms;
    - Binary Search Trees & its Various Operations;
    - AVL Tree and Min & Max Heaps.
- 6. Graph:
  - Basic Concepts & Terminology;
  - Representation,
  - Types;
  - Graph Traversal Techniques:
    - Breadth First Search,
    - Depth First Search;
    - Minimum Spanning Trees:
    - Kruskal Algorithm,
    - Prims Algorithm;
    - Shortest Path Problem:
    - Dijkstra's Algorithm.

- 7. Sorting algorithms:
  - Bubble,
  - Selection,
  - Insertion & Merge Sort,
- Searching Algorithms:
  - Linear & Binary Search;
  - Hashing: Hash Functions,
  - Hash Tables
  - Strategies for Avoiding & Resolving Collisions.
- 8. Complexity Analysis:
  - Growth Rate of Function;
  - Asymptotic Notation;
  - Time Complexity of Searching and Sorting Algorithms

Abstract Data Types and Data Structures

- “A data structure is a way of organizing, storing, and performing operations on data.”
- “An abstract data type (ADT) is a data type described by predefined user operations, such as "insert data at rear," without indicating how each operation is implemented.
- An ADT can be implemented using different underlying data structures.”

Key Goals For This Course

- Selecting appropriate ADT's
  - Based on the logical requirements of the problem: How will we need to interact with our data
- Selecting appropriate Data Structures
  - How can we organize our data so that we can interact with it efficiently in both time and space.
- Algorithm Analysis
  - How can we determine the performance characteristics of our code before we write it

Basic Definitions [Abstract Data Type]

- When an application requires a special kind of data which is not available as built-in data type then it is the responsibility of the programmer to implement his own kind of data.
- The programmer has to give more effort regarding how to store value for that data
- Amount of memory require to store for a variable
- The programmer has to decide all these things and accordingly implement it.
- Programmer own data type is termed as abstract data type aka user-defined data type

Basic Definitions [Abstract Data Type]

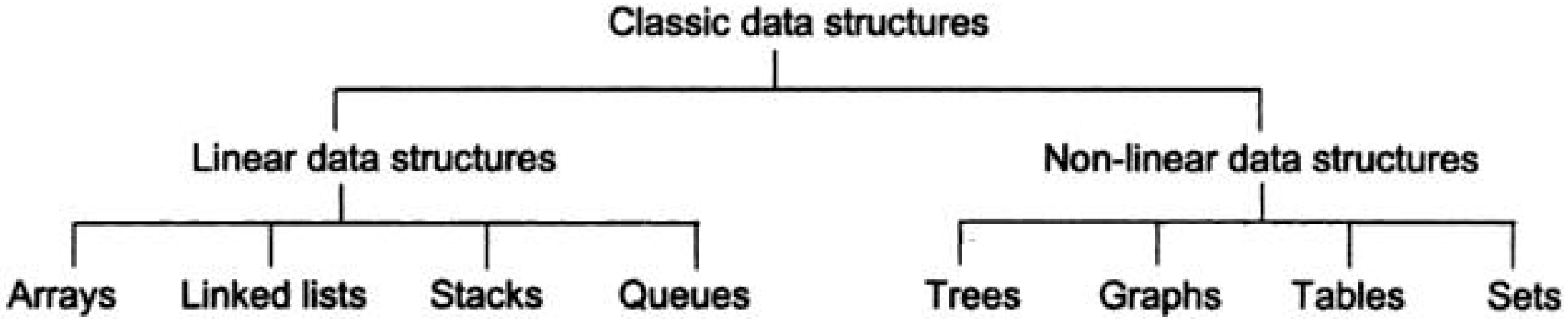
For example

- To process dates of the form dd/mm/yy (No built-in data types in C)
- Define an ADT, say Date with various operations like
  - Add few days to a date to obtain another date
  - Find days between two dates
  - Find day for a given date, etc.
- Besides this, programmer has to decide how to store the data
- What amount of memory will be needed to represent a value of Date, etc.
- An ADT can be built using built-in data types and other ADTs already built by the programmer. Like using struct/class in C/C++

Overview of Data Structures

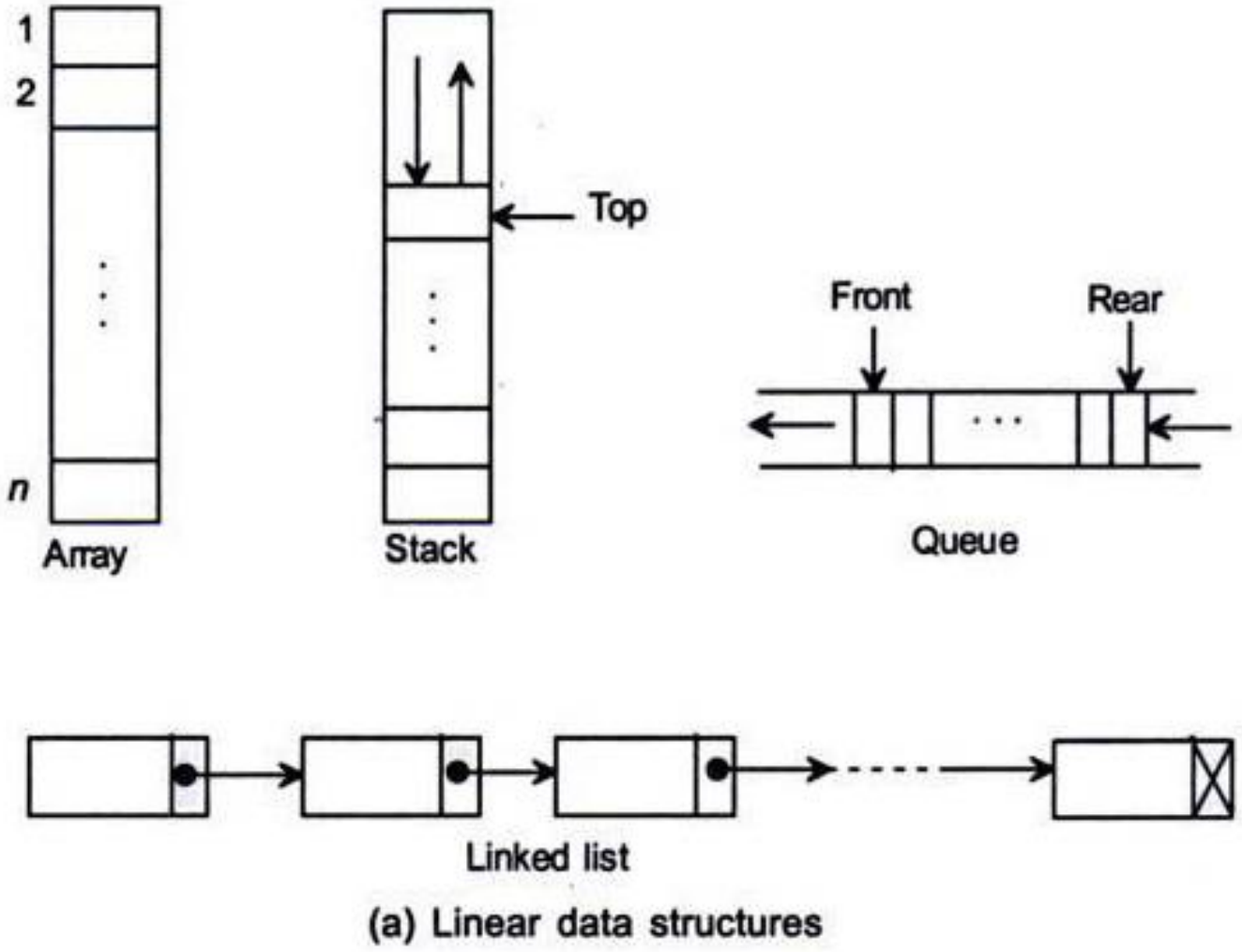
- Many data structures are known depending on area of application
- Few data structures are there which are frequently used almost in all application areas and with the help of which almost all complex data structures can be constructed -> fundamental data structures or classic data structures

Classification of all classic data structures



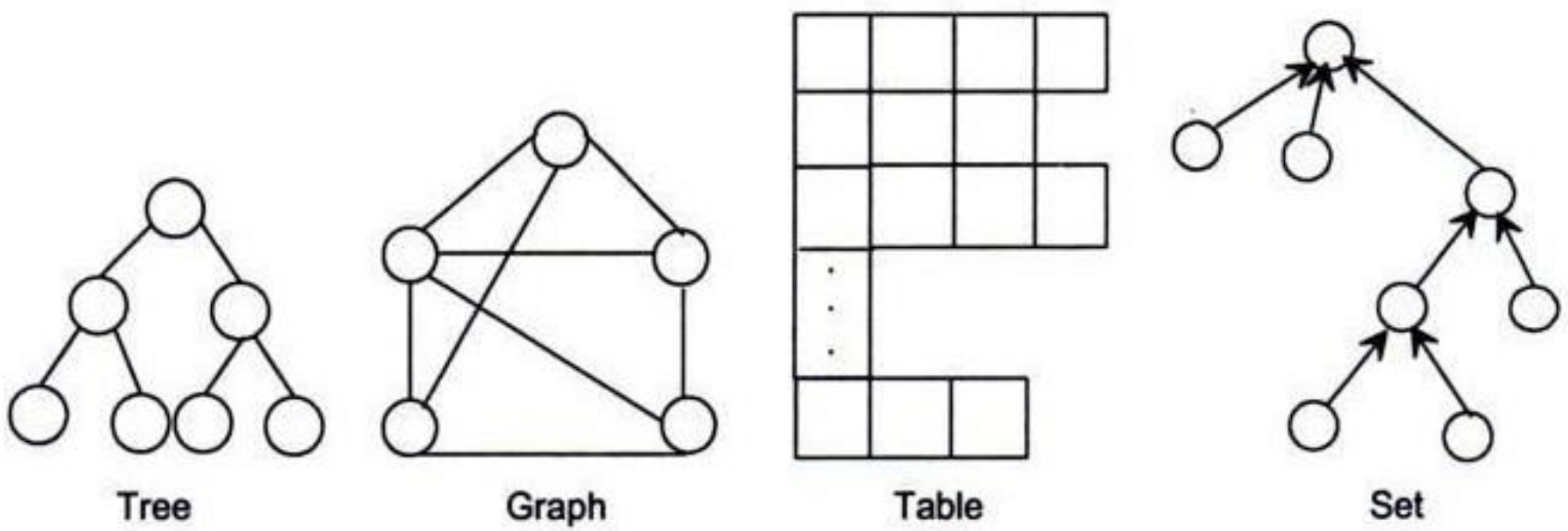
In addition to these classic data structures, other data structures such as lattice, Petri nets, neural nets, search graphs, semantic nets, etc., are known in various applications. These are known to be very complex data structures. (Discussion of all these complicated data structures is beyond the scope of this book.)

Classification of all classic data structures



(a) Linear data structures

Classification of all classic data structures



(b) Non-linear data structures

Why Data Structures Matter

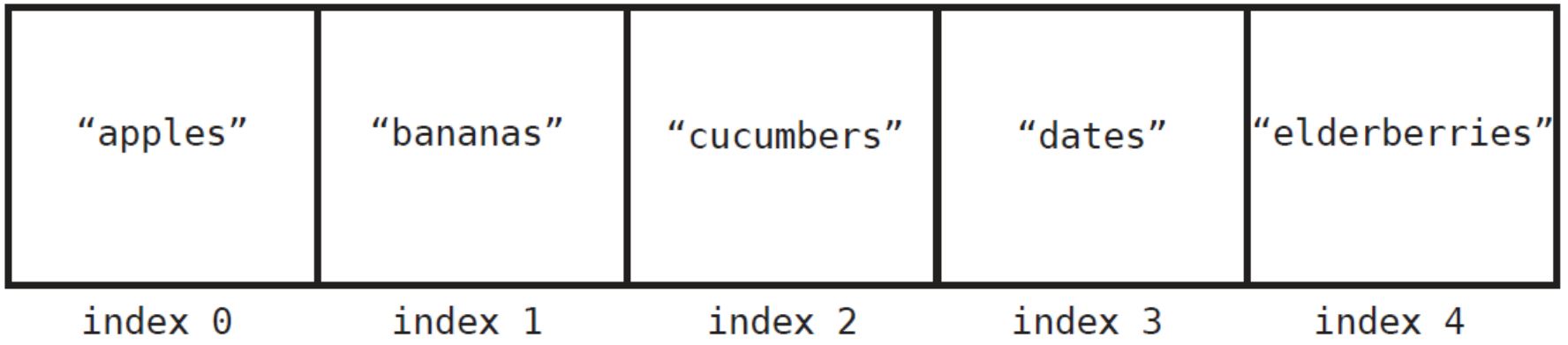
- Programming largely revolves around ***data***
- Computer programs are all about receiving, manipulating, and returning data.
- Data is a broad term that refers to all types of information, down to the most basic numbers and strings.
- In fact, even the most complex pieces of data usually break down into a bunch of numbers and strings.
- **The organization of data doesn't just matter for organization's sake but can significantly impact how fast your code runs.**

Analysis of Data Structure

- To begin our analysis of two data structures: **arrays** and **sets.**
- While the two data structures seem almost identical, you're going to learn the tools to analyze the performance implications of each choice.

Arrays

```
array = ["apples", "bananas", "cucumbers", "dates", "elderberries"]
```



Array Operations

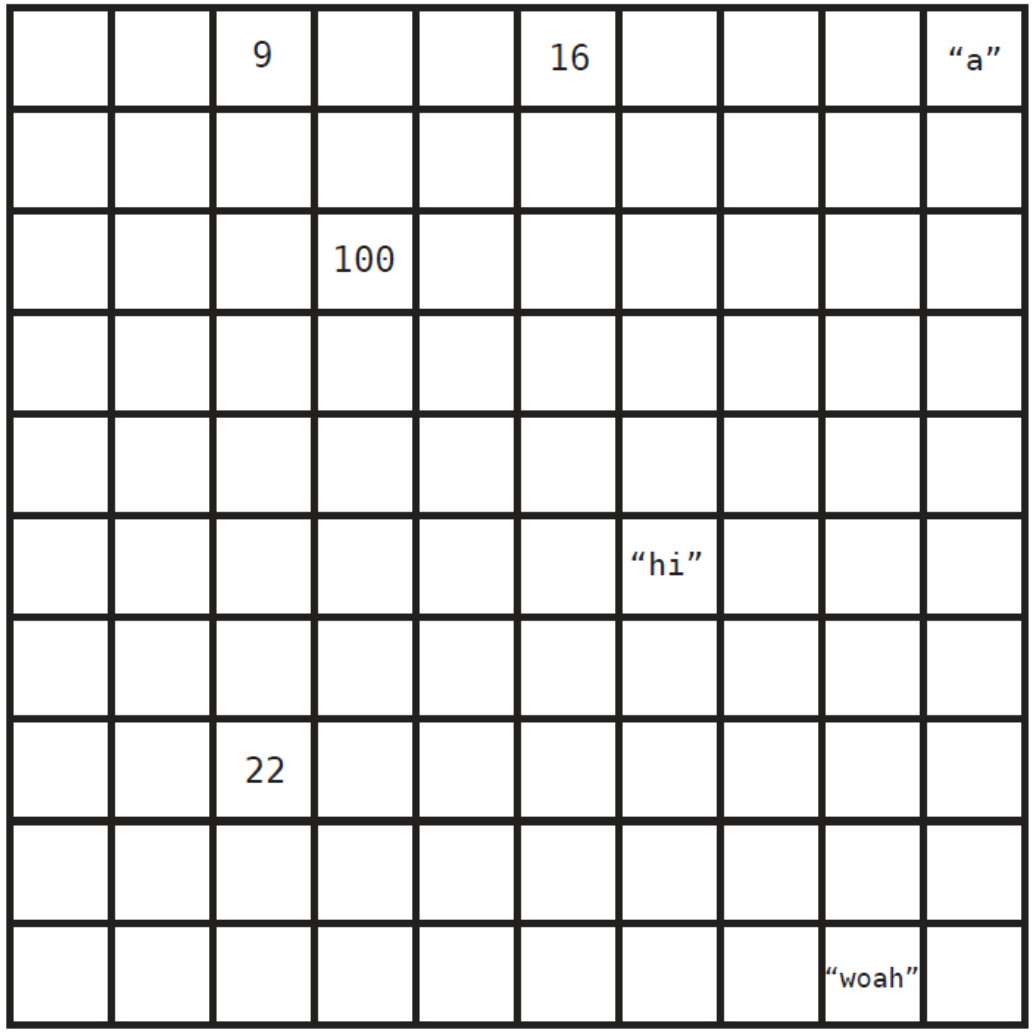
1) **Read**: Reading refers to looking something up from a particular spot within the data structure. With an array, this would mean looking up a value at a particular index. For example, looking up which grocery item is located at index 2 would be reading from the array.
2) **Search**: Searching refers to looking for a particular value within a data structure. With an array, this would mean looking to see if a particular value exists within the array, and if so, which index it's at. For example, looking to see if "dates" is in our grocery list, and which index it's located at would be searching the array.
3) **Insert**: Insertion refers to adding another value to our data structure. With an array, this would mean adding a new value to an additional slot within the array. If we were to add "figs" to our shopping list, we'd be inserting a new value into the array.
4) **Delete**: Deletion refers to removing a value from our data structure. With an array, this would mean removing one of the values from the array. For example, if we removed "bananas" from our grocery list, that would be deleting from the array.


why?
why?
why?
why?
why?

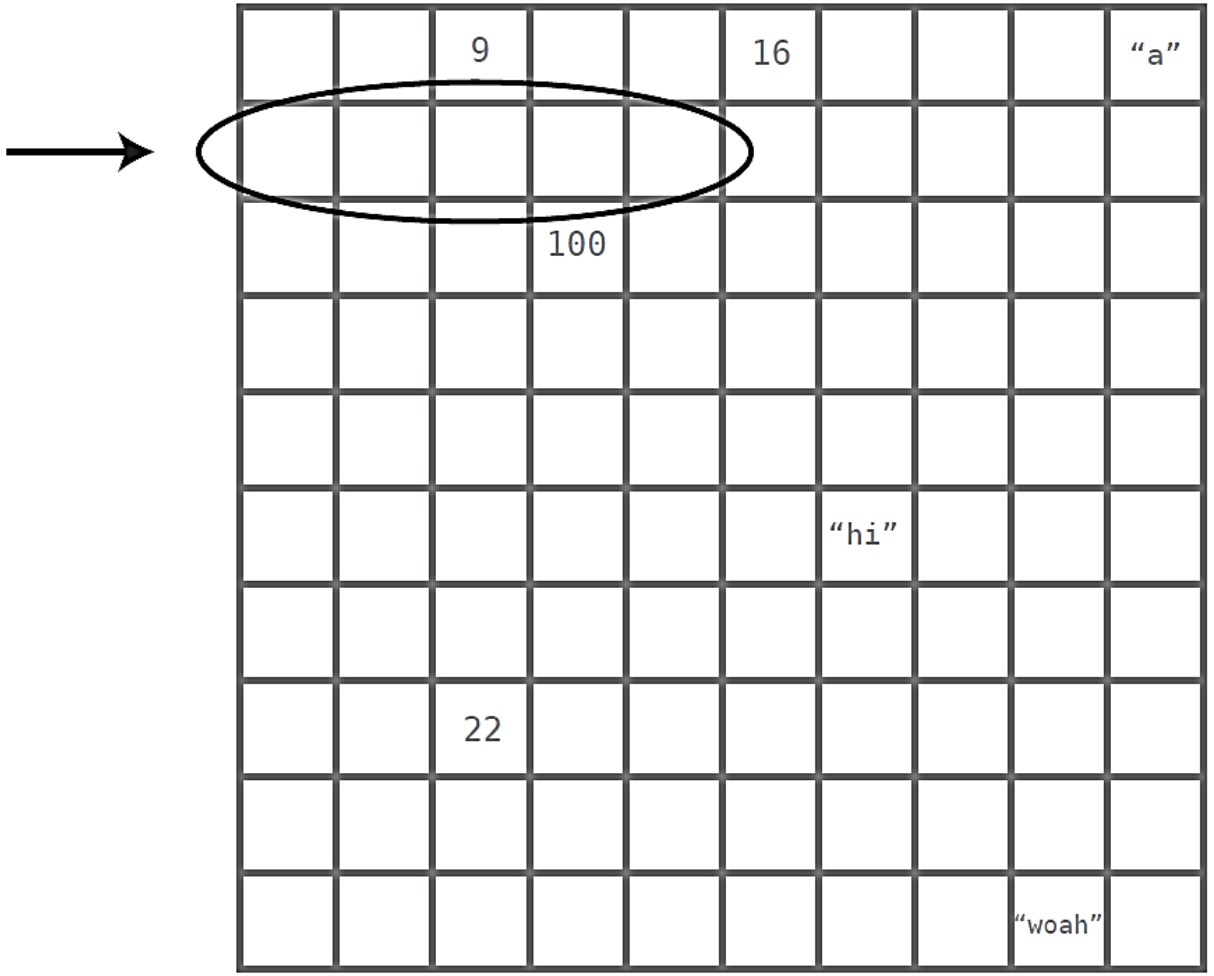
Measuring the Performance [Reading]

- Reading from an array actually takes just one step.
- This is because the computer has the ability to jump to any particular index in the array and peer inside.
- Why?

Computer Memory



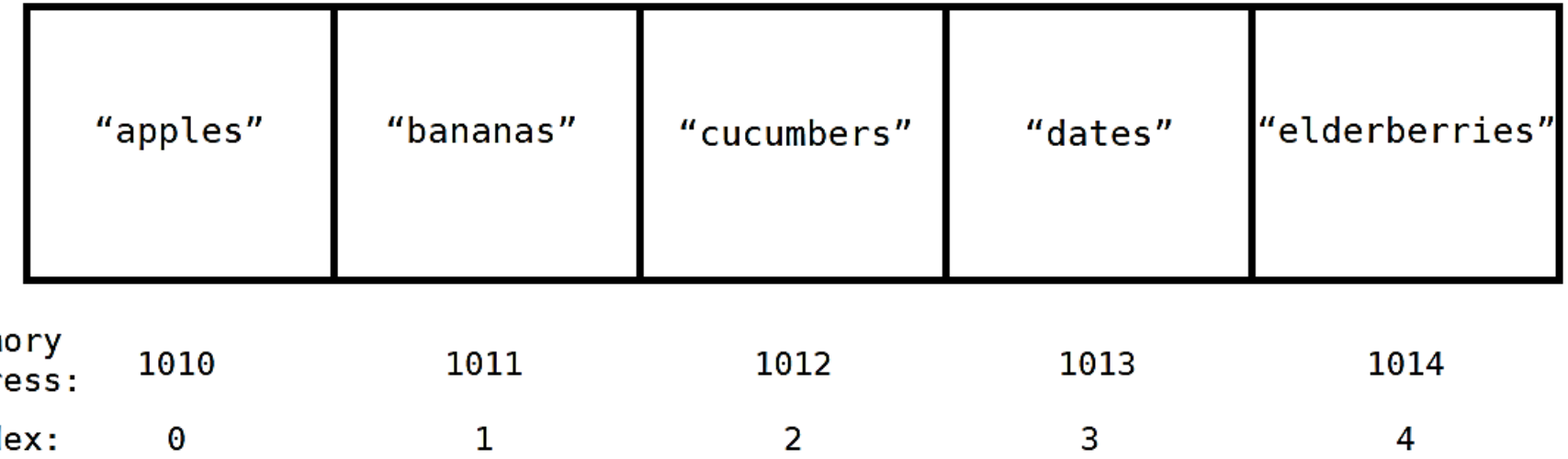
Computer Memory [Our array]



Memory Addresses

1000	1001	1002	1003	1004	1005	1006	1007	1008	1009
1010	1011	1012	1013	1014	1015	1016	1017	1018	1019
1020	1021	1022	1023	1024	1025	1026	1027	1028	1029
1030	1031	1032	1033	1034	1035	1036	1037	1038	1039
1040	1041	1042	1043	1044	1045	1046	1047	1048	1049
1050	1051	1052	1053	1054	1055	1056	1057	1058	1059
1060	1061	1062	1063	1064	1065	1066	1067	1068	1069
1070	1071	1072	1073	1074	1075	1076	1077	1078	1079
1080	1081	1082	1083	1084	1085	1086	1087	1088	1089
1090	1091	1092	1093	1094	1095	1096	1097	1098	1099

Reading



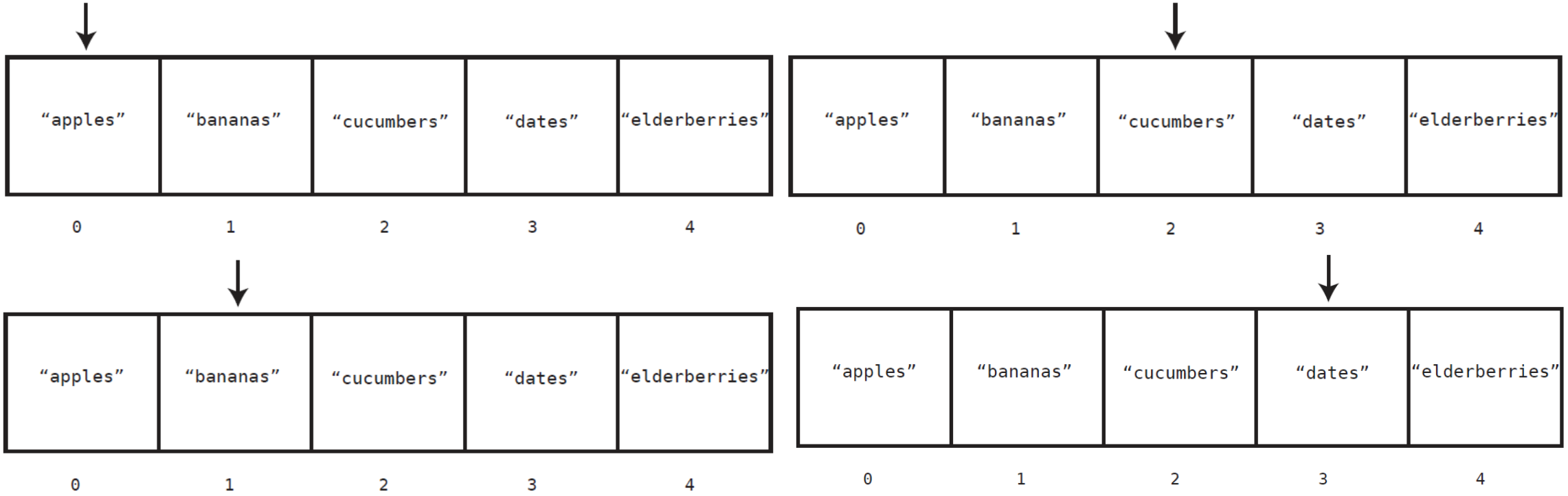
- When the computer reads a value at a particular index of an array, it can jump straight to that index in one step because of the combination.

Reading – Performance

- Reading from an array is, therefore, a very efficient operation, since it takes just one step.
- An operation with just one step is naturally the fastest type of operation.
- One of the reasons that the array is such a powerful data structure is that we can look up the value at any index with such speed.

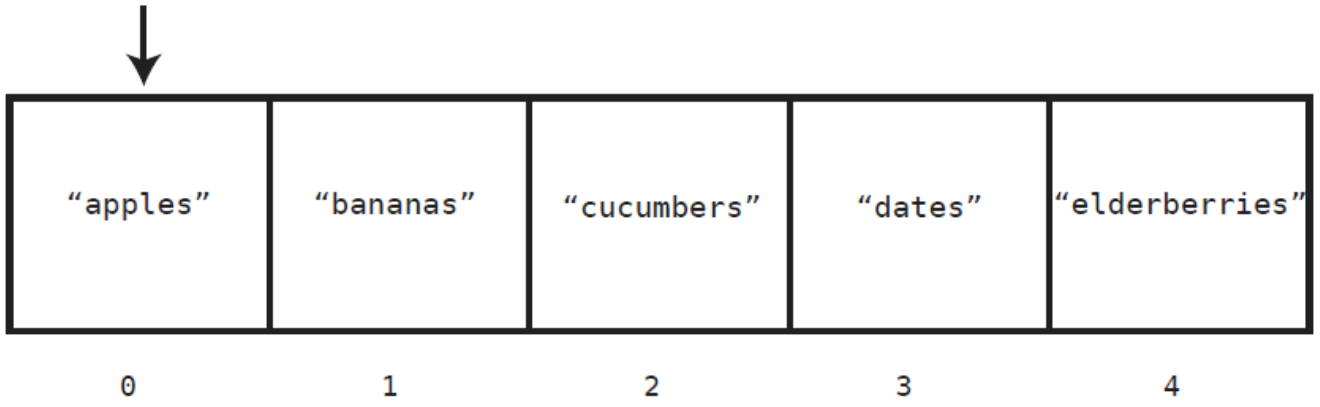
Searching

- “dates”



Searching- Performance

- Now, what is the maximum number of steps a computer would need to conduct a linear search on an array?
- Let's search for “elderberries”

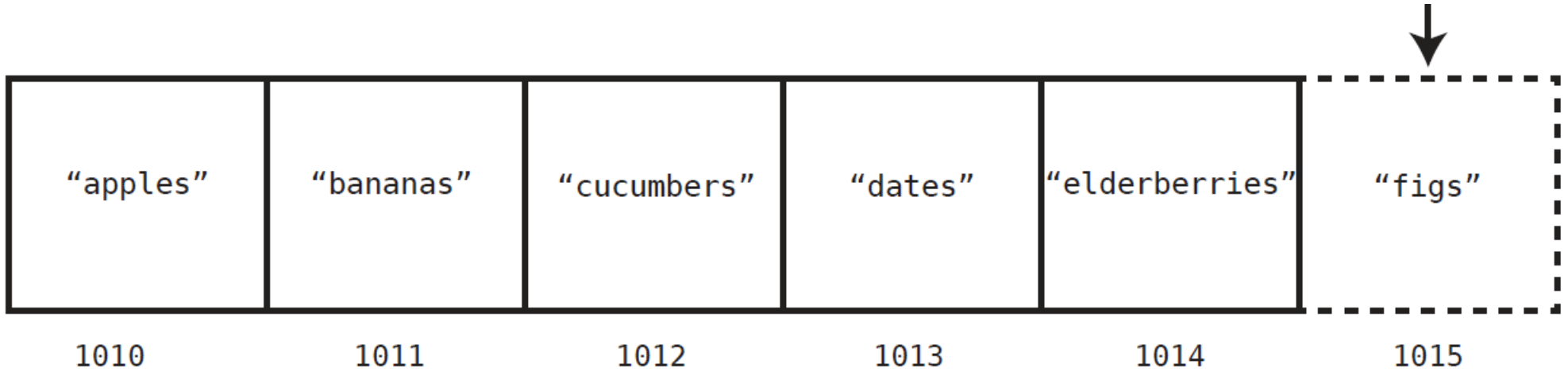


for N cells in an array, linear search will take a maximum of N steps

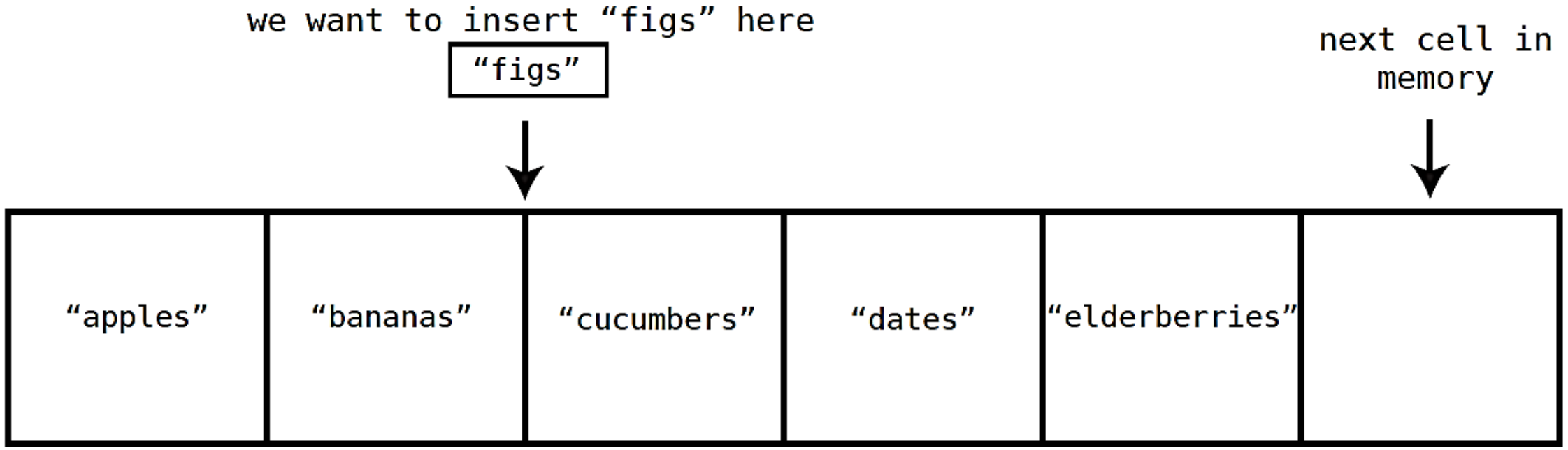
Insertion

- The efficiency of inserting a new piece of data inside an array depends on where inside the array you'd like to insert it.
  - End of the list
  - Start of the list
  - At a specific place

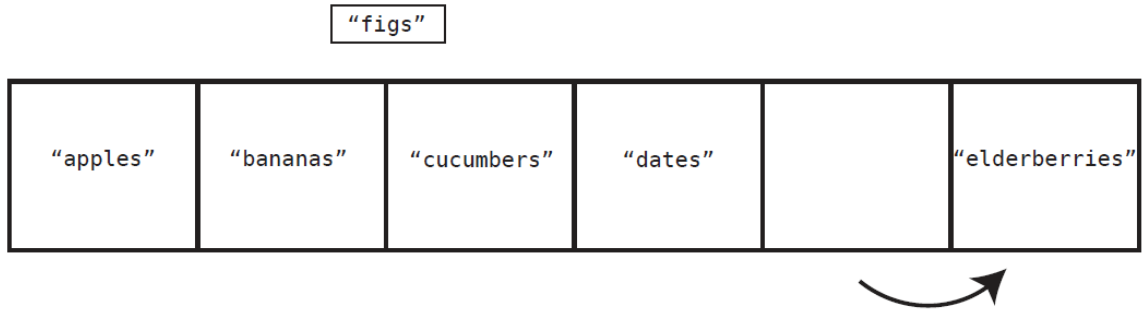
Insertion @ end

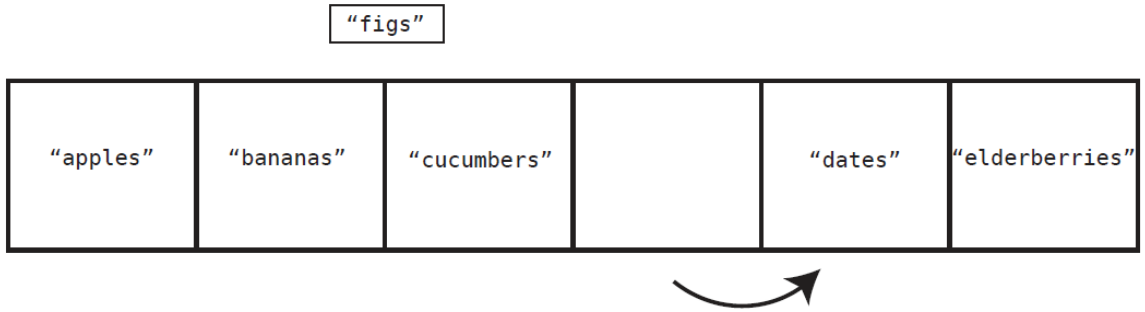


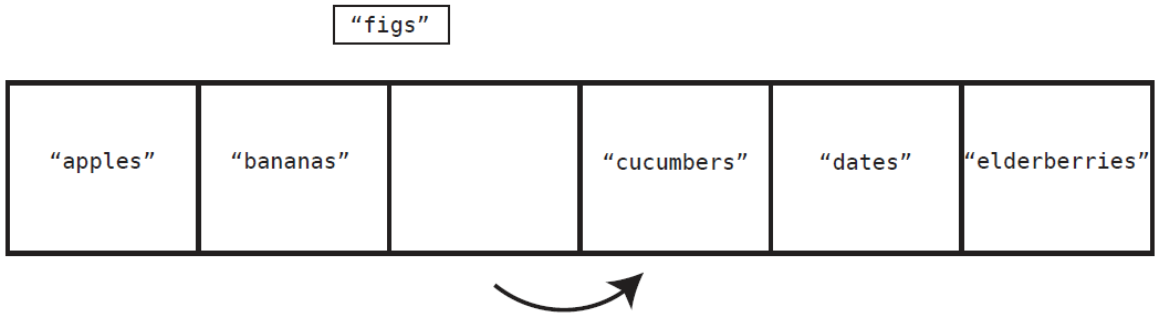
Insertion @ middle

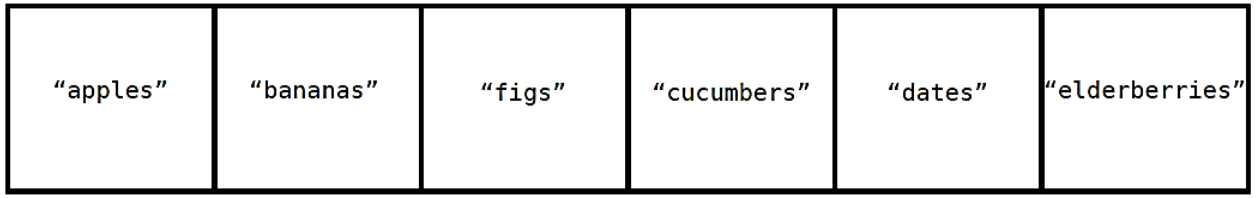


Insertion @ middle [demo]









There were four steps. Three of the steps were shifting data to the right, while one step was the actual insertion of the new value.

Insertion-Performance

- The worst-case scenario is insertion at the beginning of the array.

- This is because when inserting into the beginning of the array, we have to move all the other values one cell to the right.
- So we can say that insertion in a worst-case scenario can take up to N + 1 steps for an array containing N elements.

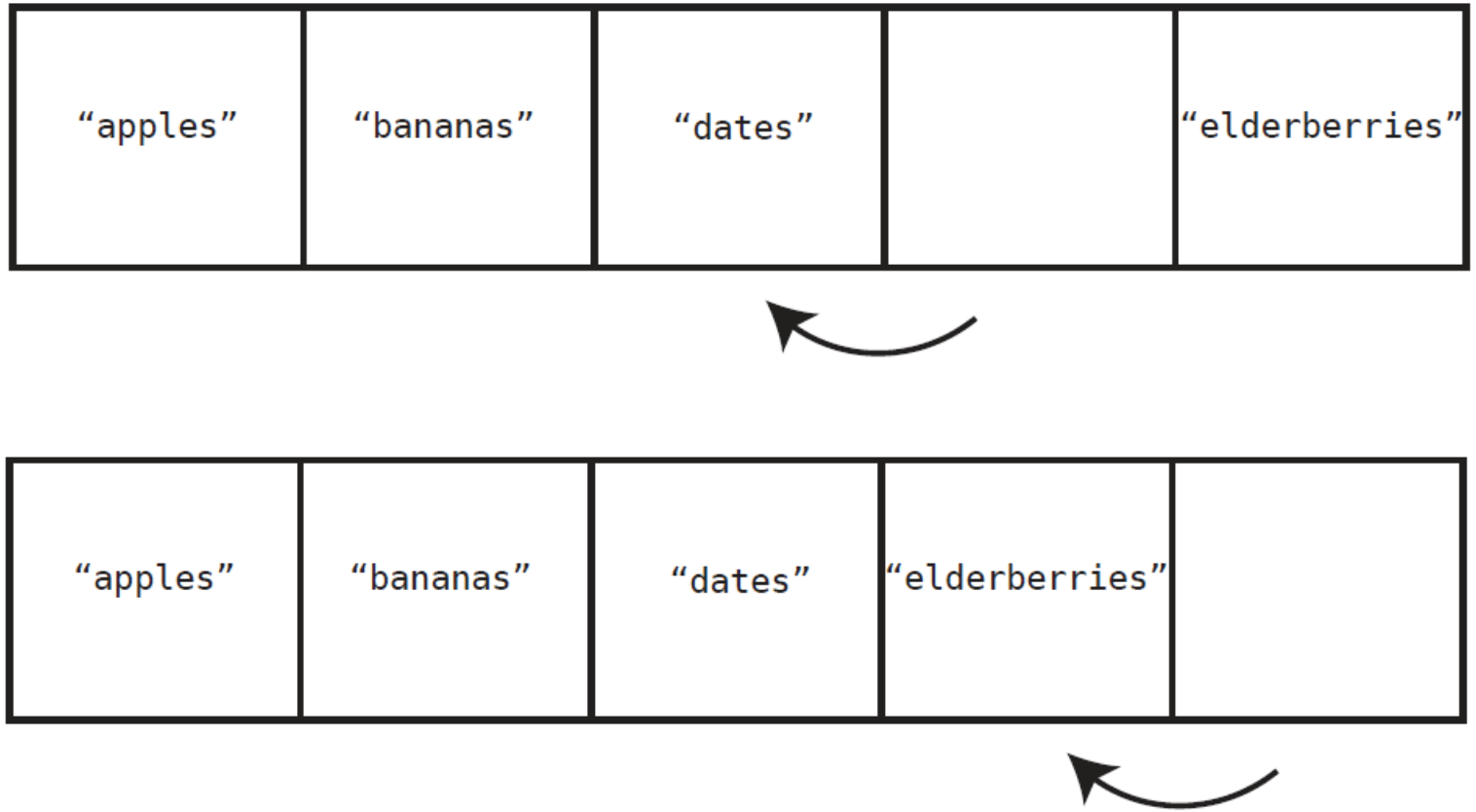
Deletion

- Deletion from an array is the process of eliminating the value at a particular index.
- Let’s return to our original example array and delete the value at index 2.
- In our example, this would be the "cucumbers".

Deletion

- While the actual deletion of "cucumbers" technically took just one step, we now have a problem: we have an empty cell sitting smack in the middle of our array.
- An array is not allowed to have gaps in the middle of it, so to resolve this issue, we need to shift "dates" and "elderberries" to the left.

Deletion


“apples”
“bananas”
“dates”
“elderberries”
“apples”
“bananas”
“dates”
“elderberries”

Deletion-Performance

- For an array of five elements, we'd spend one step deleting the first element, and four steps shifting the four remaining elements.
- For an array of 500 elements, we'd spend one step deleting the first element, and 499 steps shifting the remaining data.
- We can conclude, then, that for an array containing N elements, the maximum number of steps that deletion would take is N steps.

Conclusion

- Now that we've learned how to analyze the time complexity of a data structure, we can discover how different data structures have different efficiencies.
- This is extremely important, since choosing the correct data structure for your program can have serious ramifications as to how performant your code will be.

Sets

- Let's explore another data structure: the set.
- A set is a data structure that does not allow duplicate values to be contained within it.
- For this discussion, we'll talk about an array-based set.
- This set is just like an array—it is a simple list of values.
- The only difference between this set and a classic array is that the set never allows duplicate values to be inserted into it.
- Sets are useful when you need to ensure that you don't have duplicate data.

Reading-Performance

- Reading from a set is exactly the same as reading from an array—it takes just one step for the computer to look up what is contained within a particular index.
- As we described earlier, this is because the computer can jump to any index within the set since it knows the memory address that the set begins at.

Searching-Performance

- Searching a set also turns out to be no different than searching an array—it takes up to N steps to search to see if a value exists within a set.

Deletion-Performance

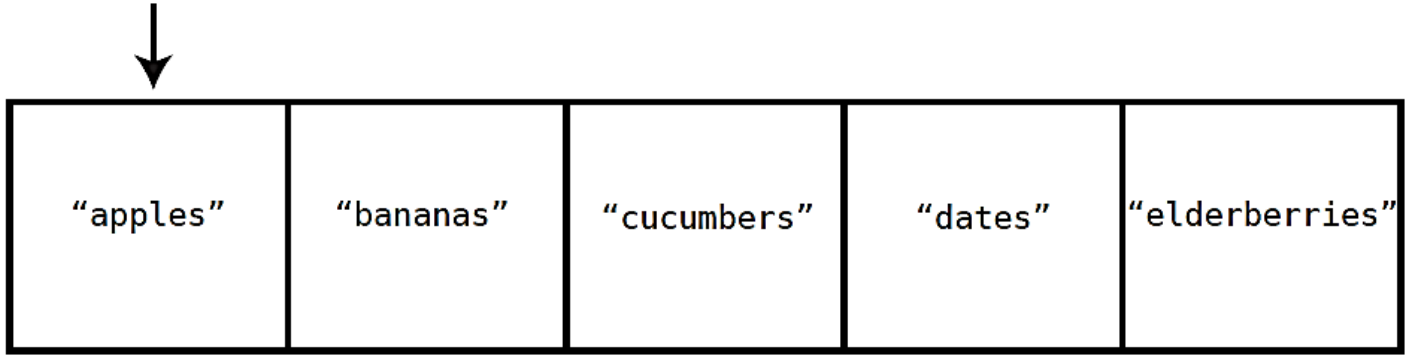
- Deletion is also identical between a set and an array—it takes up to N steps to delete a value and move data to the left to close the gap.

Insertion-Performance

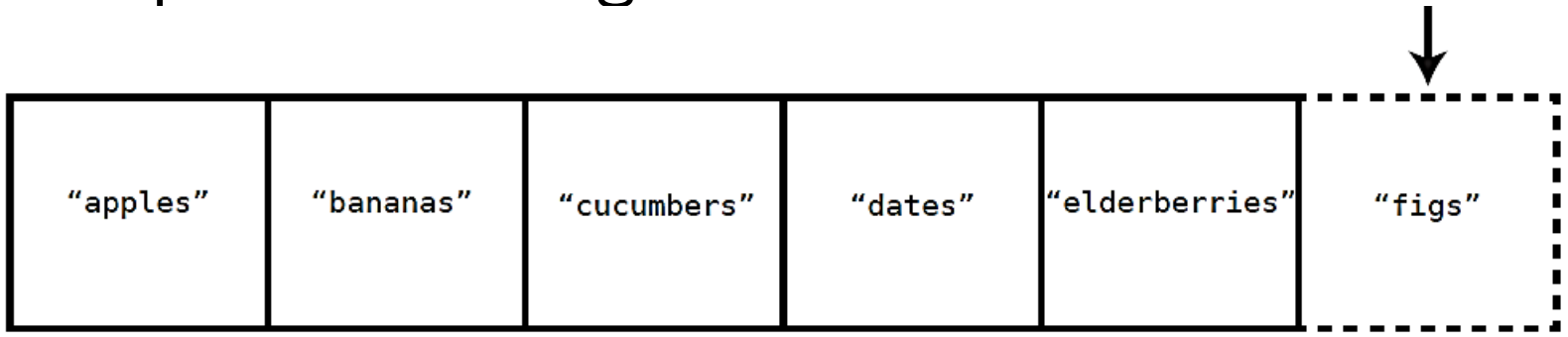
- Insertion, however, is where arrays and sets diverge.
- Let's first explore inserting a value at the end of a set, which was a best-case scenario for an array.
- With an array, the computer can insert a value at its end in a single step.
- With a set, however, the computer first needs to determine that this value doesn't already exist in this set—because that's what sets do: they prevent duplicate data.
- So every insert first requires a search.

Insertion-Demo *[best-case]*

- Insert figs

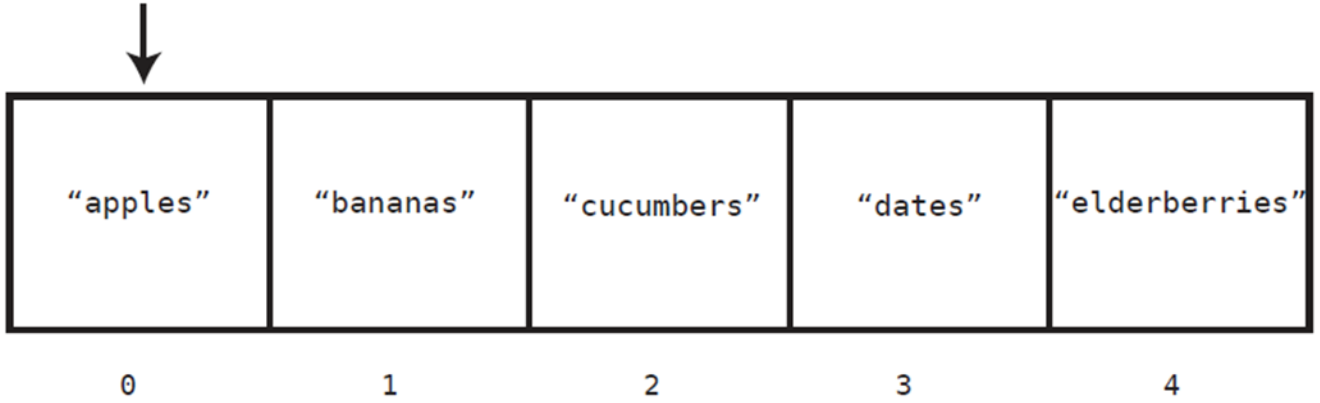


- Only now that we've searched the entire set do we know that it's safe to insert "figs". And that brings us to our final step.
- Step #6: Insert "figs" at the end of the set:



best-case scenario will take N + 1 steps for N elements

Insertion-Demo [worst-case]



- In a worst-case scenario, where we're inserting a value at the beginning of a set, the computer needs to search N cells to ensure that the set doesn't already contain that value, and then another N steps to shift all the data to the right, and another final step to insert the new value.
- That's a total of 2N + 1 steps.

Wrapping up

- Analyzing the number of steps that an operation takes is the heart of understanding the performance of data structures.
- Choosing the right data structure for your program can spell the difference between bearing a heavy load vs collapsing under it.
- You've learned to use this analysis to weigh whether an array or a set might be the appropriate choice for a given application.
- Now that we've begun to learn how to think about the time complexity of data structures, we can also use the same analysis to compare competing algorithms (even within the same data structure) to ensure the ultimate speed and performance of our code.

Why Algorithms Matter

Data Structures and Algorithms

- **Data organizations** are ways data is arranged in the computer using its various storage media (such as random-access memory, or RAM, and disk) and how that data is interpreted to represent something.
- **Algorithms** are the procedures used to manipulate the data in these structures.
- The way data is arranged can simplify the algorithms and make algorithms run faster or slower.
- Together, the data organization and the algorithm form a data structure.
- The data structures act like building blocks, with more complex data structures using other data structures as components with appropriate algorithms.

Data Structures and Algorithms

- Does the way data is arranged and the algorithm used to manipulate it make a difference?

- From the perspective of nonprogrammers, it often seems as though computers can do anything and do it very fast.
- That's not really true.
- To see why, let's look at a nonprogramming example.

Example – The Scenario

Let's assume that you have a written recipe

Method A

1) Read through the full recipe noting all the ingredients it mentions, their quantities, and any equipment needed to process them.
2) Find the ingredients, measure out the quantity needed, and store them.
3) Get out all the equipment needed to complete the steps in the recipe.
4) Go through the steps of the recipe in the order specified.

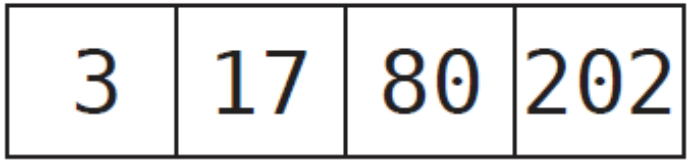
Method B

1) Read the recipe until you identify the first set of ingredients or equipment that is needed to complete the first step.
2) Find the identified ingredients or equipment.
3) Measure any ingredients found.
4) Perform the first step with the equipment and ingredients already found.
5) Return to the beginning of this method and repeat the instructions replacing the word first with next. If there is no next step, then quit.

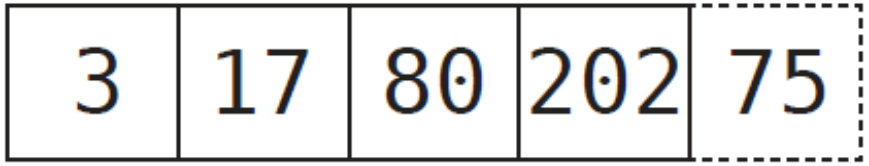
Analysis

- Both methods are complete in that that they should finish the complete recipe if all the ingredients and equipment are available.
- For simple recipes, they should take about the same amount of time too.
- The methods differ as the recipes get more complex. For example, what if you can't find the fifth ingredient?
- In method A, that issue is identified at the beginning before any other ingredients are combined.
- While neither method really explains what to do about exceptions like a missing ingredient, you can still compare them under the assumption that you handle the exceptions the same way.

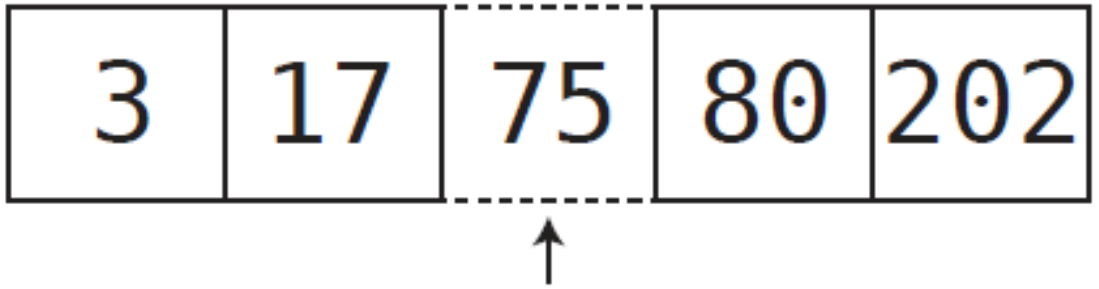
Ordered Arrays



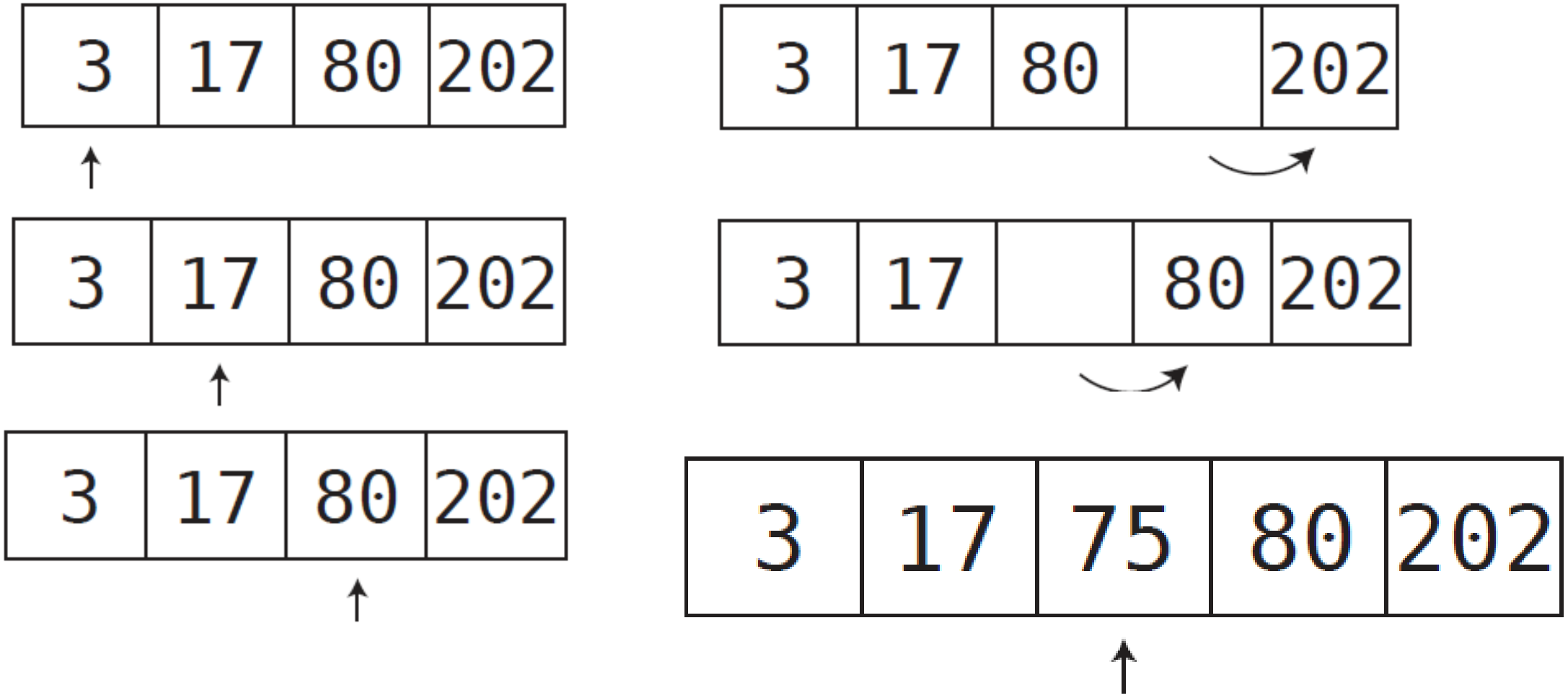
- Inserting “75” in typical array



- Inserting “75” in Ordered array



Insertion (75) in Ordered Array



Searching in Ordered Array

- Recall searching in simple arrays → Linear Search ***(only)***
- Search 22

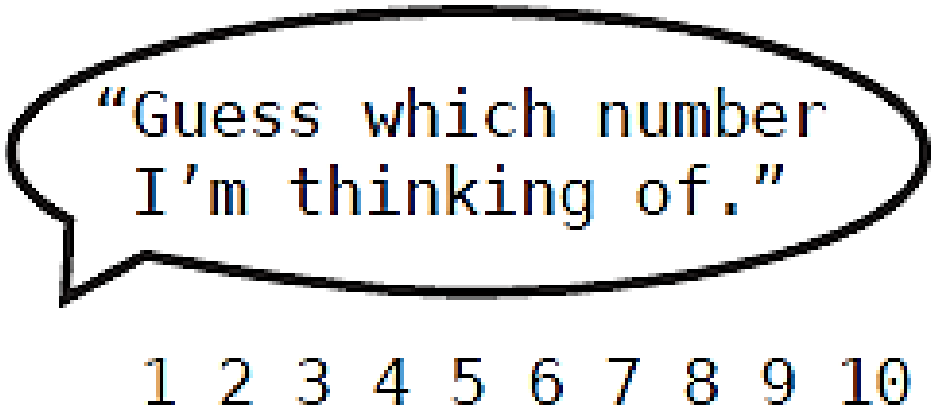
[17, 3, 75, 202, 80]

- Now in Ordered Array

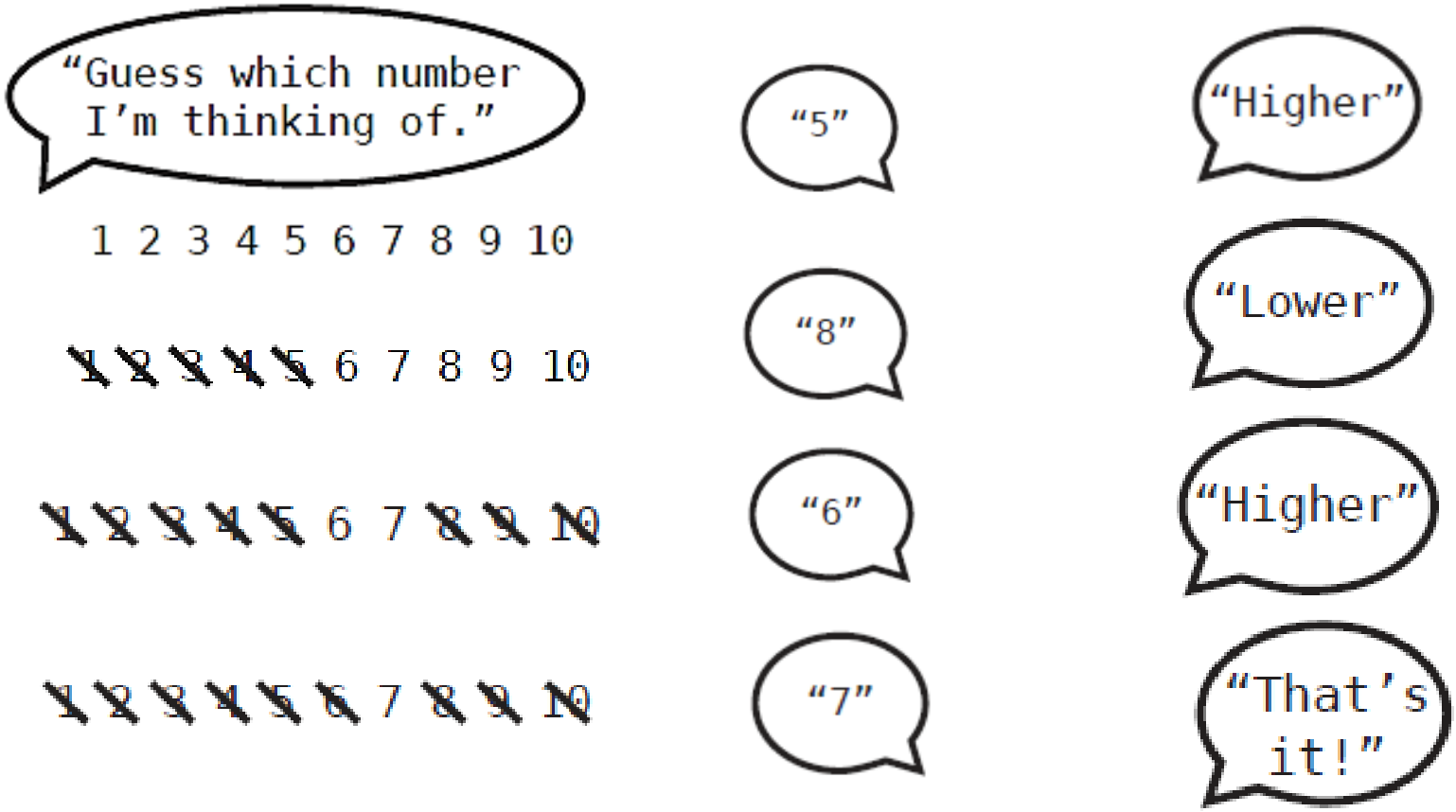
[3, 17, 75, 80, 202]

- Do you find any performance difference?

Binary Search (Ordered Array)



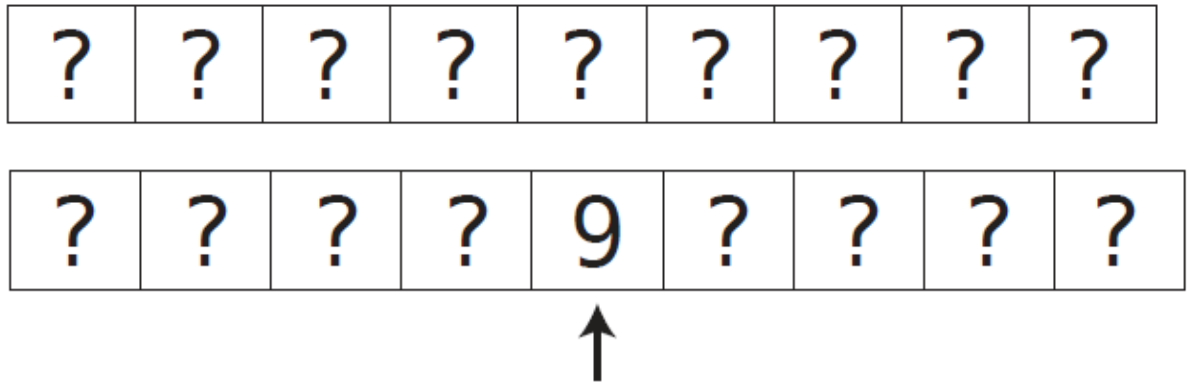
Binary Search (Guessed 7)



Analysis

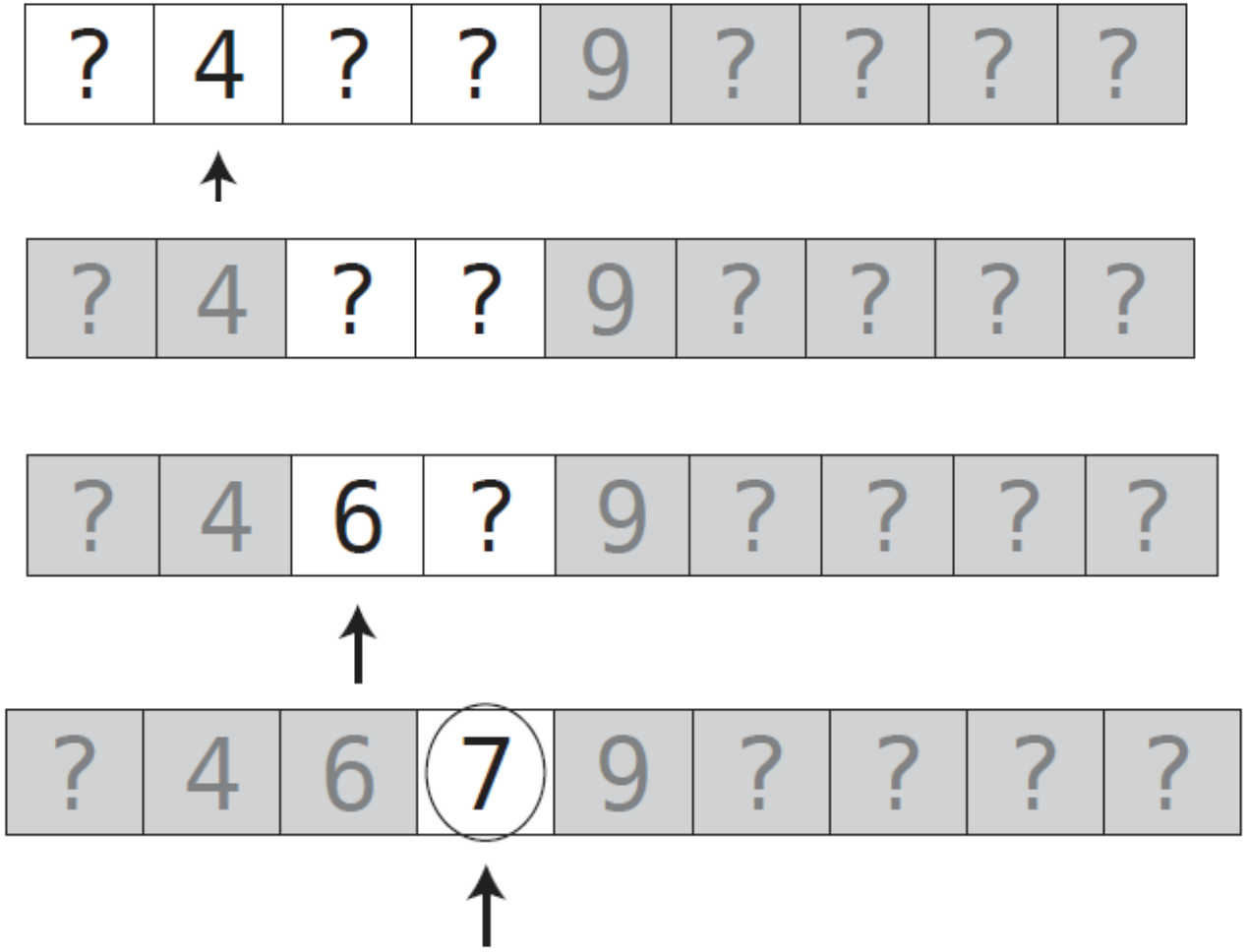
- The major advantage of an ordered array over a standard array is that we have the option of performing a binary search rather than a linear search.
- Binary search is impossible with a standard array because the values can be in any order.

Working of Binary Search (search 7)



- Since the value uncovered is a 9, we can conclude that the 7 is somewhere to its left. We've just successfully eliminated half of the array's cells—that is, all the cells to the right of the 9 (and the 9 itself):

Working of Binary Search (search 7)

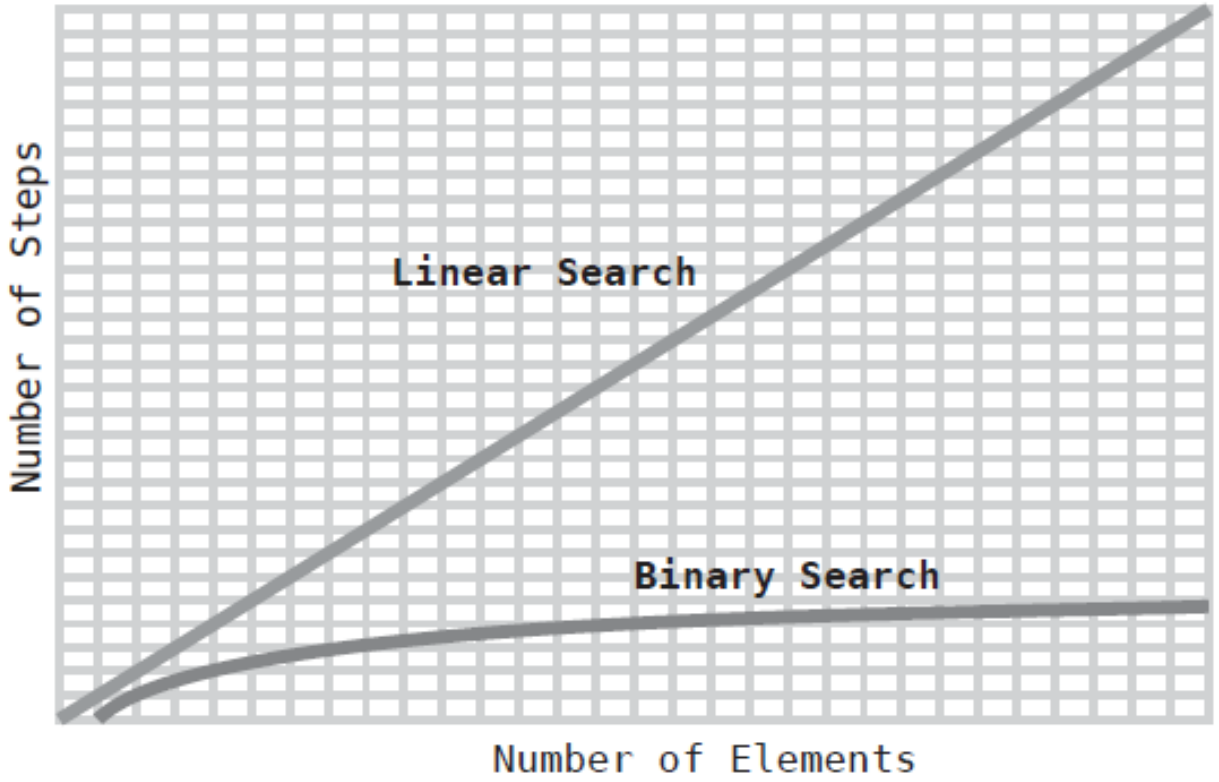


Binary Search vs. Linear Search

- With an array containing one hundred values, here are the maximum numbers of steps it would take for each type of search:
  - Linear search:

    one hundred steps
  - Binary search:

    seven steps



Wrapping Up

- Often, there is more than one way to achieve a particular computing goal, and the algorithm you choose can seriously affect the speed of your code.
- It's also important to realize that there usually isn't a single data structure or algorithm that is perfect for every situation.
  - For example, just because ordered arrays allow for binary search doesn't mean you should always use ordered arrays.
  - In situations where you don't anticipate searching the data much, but only adding data, standard arrays may be a better choice because their insertion is faster.
- As we've seen, the way to analyze competing algorithms is to count the number of steps each one takes.