

Arrays & Searching Algorithms

Data Structures

□ Data structure

- A particular way of storing and organising data in a computer so that it can be used efficiently

□ Types of data structures

- Based on memory allocation
 - Static (or fixed sized) data structures (Arrays)
 - Dynamic data structures (Linked lists)
- Based on representation
 - Linear (Arrays/linked lists)
 - Non-linear (Trees/graphs)

Array: motivation

- ❑ You want to store 5 numbers in a computer
 - Define 5 variables, e.g. num1, num2, ..., num5
- ❑ What, if you want to store 1000 numbers?
 - Defining 1000 variables is a pity!
 - Requires much programming effort
- ❑ Any better solution?
 - Yes, some structured data type
 - Array is one of the most common structured data types
 - Saves a lot of programming effort (cf. 1000 variable names)

What is an Array?

- A collection of data elements in which
 - all elements are of the same data type, hence **homogeneous** data
 - An array of students' marks
 - An array of students' names
 - An array of objects (OOP perspective!)
 - elements (or their references) are stored at contiguous/ consecutive memory locations
- Array is a static data structure
 - An array cannot **grow** or **shrink** during program execution – its size is fixed

Basic concepts

- ❑ Array name (data)
- ❑ Index/subscript (0...9)
- ❑ The slots are numbered sequentially starting at zero (Java, C++)
- ❑ If there are N slots in an array, the index will be 0 through N-1
 - Array length = $N = 10$
 - Array size = $N \times \text{Size of an element} = 40$
- ❑ Direct access to an element

data	
0	23
1	38
2	14
3	-3
4	0
5	14
6	9
7	103
8	0
9	-56

Homogeneity

- ❑ All elements in the array must have the same data type

Index:	0	1	2	3	4	5	6	7	8	9
Value:	5	10	18	30	45	50	60	65	70	80

Index:	0	1	2	3	4
Value:	5.5	10.2	18.5	45.6	60.5

Index:	0	1	2	3	4
Value:	'A'	10.2	55	'X'	60.5

Not an array

Contiguous Memory

- ❑ Array elements are stored at contiguous memory locations

Index:	0	1	2	3	4	5	6	7	8	9
Value:	5	10	18	30	45	50	60	65	70	80

- ❑ No empty segment in between values (3 & 5 are empty – not allowed)

Index:	0	1	2	3	4	5	6	7	8	9
Value:	5	10	18		45		60	65	70	80

Using Arrays

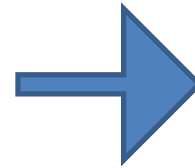
□ `Array_name[index]`

□ For example, in Java

- `System.out.println(data[4])` will display 0

- `data[3] = 99` will replace -3 with 99

data	
0	23
1	38
2	14
3	-3
4	0
5	14
6	9
7	103
8	0
9	-56



data	
0	23
1	38
2	14
3	99
4	0
5	14
6	9
7	103
8	0
9	-56

Some more concepts

`data[-1]` always illegal

`data[10]` illegal ($10 >$ upper bound)

`data[1.5]` always illegal

`data[0]` always OK

`data[9]` OK

Q. What will be the output of?

1. `data[5] + 10`

2. `data[3] = data[3] + 10`

data	
0	23
1	38
2	14
3	-3
4	0
5	14
6	9
7	103
8	0
9	-56

Array's Dimensionality

□ One dimensional (just a linear list)

- e.g.,

5	10	18	30	45	50	60	65	70	80
---	----	----	----	----	----	----	----	----	----
- Only one subscript is required to access an individual element

□ Two dimensional (matrix/table)

- e.g., 2 x 4 matrix (2 rows, 4 columns)

	Col 0	Col 1	Col 2	Col 3
Row 0	20	25	60	40
Row 1	30	15	70	90

Two dimensional Arrays

□ Let, the name of the two dimensional array is M

20	25	60	40
30	15	70	90

□ Two indices/subscripts are required (row, column)

□ First element is at row 0, column 0

▪ $M_{0,0}$ or $M(0, 0)$ or $M[0][0]$ (more common)

□ What is: $M[1][2]$? $M[3][4]$?

Array Operations (1)

- ❑ Indexing: inspect or update an element using its index. Performance is very fast $O(1)$

```
randomNumber = numbers[5];
```

```
numbers[20000] = 100;
```

- ❑ Insertion: add an element at certain index
 - Start: very slow $O(n)$ because of shift
 - End : very fast $O(1)$ because no need to shift
- ❑ Removal: remove an element at certain index
 - Start: **very slow $O(n)$ because of shift**
 - End : very fast $O(1)$ because no need to shift

Array Operations (2)

❑ Search: performance depends on algorithm

1) Linear: slow $O(n)$ 2) binary : $O(\log n)$

❑ Sort: performance depends on algorithm

1) Bubble: slow $O(n^2)$ 2) Selection: slow $O(n^2)$

3) Insertion: slow $O(n^2)$ 4) Merge : $O(n \log n)$

One Dimensional Arrays in Java

- ❑ To declare an array follow the type with (empty) []s
 - `int[] grade; //or`
 - `int grade[]; //both declare an int array`
- ❑ In Java arrays are objects so must be created with the **new** keyword
 - To create an array of ten integers:
 - `int[] grade = new int[10];`
 - Note** that the array size has to be specified, although it can be specified with a variable at run-time

Arrays in Java

- ❑ When the array is created memory is reserved for its contents
- ❑ Initialization lists can be used to specify the initial values of an array, in which case the **new** operator is not used

```
int[] grade = {87, 93, 35}; //array of 3 ints
```

- ❑ To find the length of an array use its **.length** property

```
int numGrades = grade.length; //note: not .length()!!
```

Searching Algorithms (1)

- ❑ Search for a **target (key)** in the search space
- ❑ Search space examples are:
 - All students in the class
 - All numbers in a given list
- ❑ One of the two possible outcomes
 - Target is found (success)
 - Target is not found (failure)

Searching Algorithms (2)

Index:	0	1	2	3	4
Value:	20	40	10	30	60

Target = 30 (success or failure?)

Target = 45 (success or failure?)

Search strategy?

List Size = $N = 5$

Min index = 0

Max index = 4 ($N - 1$)

Sequential Search (1)

- ❑ Search in a sequential order
- ❑ Termination condition
 - Target is found (success)
 - List of elements is exhausted (failure)

Sequential Search (2)

Index:	0	1	2	3	4
Value:	20	40	10	30	60

Target = 30

Step 1: Compare 30 with value at index 0

Step 2: Compare 30 with value at index 1

Step 3: Compare 30 with value at index 2

Step 4: Compare 30 with value at index 3 (success)

Sequential Search (3)

Index:	0	1	2	3	4
Value:	20	40	10	30	60

Target = 45

Step 1: Compare 45 with value at index 0

Step 2: Compare 45 with value at index 1

Step 3: Compare 45 with value at index 2

Step 4: Compare 45 with value at index 3

Step 5: Compare 45 with value at index 4

Failure

Sequential Search Algorithm (4)

Given: A list of N elements, and the target



1. $index \leftarrow 0$
2. Repeat steps 3 to 5
3. Compare target with $list[index]$
4. *if* target = $list[index]$ *then*
 return index // success
 else if $index \geq N - 1$
 return -1 // failure
5. $index \leftarrow index + 1$

Binary Search (1)

- ❑ Search through **a sorted list of items**
 - Sorted list is a **pre-condition** for Binary Search!
- ❑ Repeatedly divides the search space (list) into two
- ❑ Divide-and-conquer approach

Binary Search: An Example (Key \in List) (2)

Index:	0	1	2	3	4	5	6	7	8	9
Value:	5	10	18	30	45	50	60	65	70	80

Target (Key) = 30

First iteration: whole list (search space), compare with mid value

Low Index (LI) = 0; High Index (HI) = 9



Choose element with index $(0 + 9) / 2 = 4$

Compare value at index 4 (**45**) with the key (**30**)

30 is less than 45, so the target must be in the lower half of the list

Binary Search: An Example (Key \in List) (3)

Second Iteration: Lookup in the reduced search space

Index:	0	1	2	3	4	5	6	7	8	9
Value:	5	10	18	30	45	50	60	65	70	80
	 LI			 HI						

Low Index (LI) = 0; High Index (HI) = $(4 - 1) = 3$

Choose element with index $(0 + 3) / 2 = 1$



Compare value at index 1 (**10**) with the key (**30**)

30 is greater than 10, so the target must be in the higher half of the (reduced) list

Binary Search: An Example (Key \in List) (4)

Third Iteration: Lookup in the further reduced search space

Index:	0	1	2	3	4	5	6	7	8	9
Value:	5	10	18	30	45	50	60	65	70	80

Low Index (LI) = $1 + 1 = 2$; High Index (HI) = 3

Choose element with index $(2 + 3) / 2 = 2$



Compare value at index 2 (**18**) with the key (**30**)

30 is greater than 18, so the target must be in the higher half of the (reduced) list

Binary Search: An Example (Key \in List) (5)

Fourth Iteration: Lookup in the further reduced search space

Index:	0	1	2	3	4	5	6	7	8	9
Value:	5	10	18	30	45	50	60	65	70	80

Low Index (LI) = $2 + 1 = 3$; High Index (HI) = 3



Choose element with index $(3 + 3) / 2 = 3$

Compare value at index 3 (**30**) with the key (**30**)

Key is found at index 3

Binary Search: An Example (Key \notin List) (6)

Index:	0	1	2	3	4	5	6	7	8	9
Value:	5	10	18	30	45	50	60	65	70	80

 LI  HI

Target (Key) = 40

First iteration: Lookup in the whole list (search space)

Low Index (LI) = 0; High Index (HI) = 9



Choose element with index $(0 + 9) / 2 = 4$

Compare value at index 4 (**45**) with the key (**40**)

40 is less than 45, so the target must be in the lower half of the list

Binary Search: An Example (Key \notin List) (7)

Second Iteration: Lookup in the reduced search space

Index:	0	1	2	3	4	5	6	7	8	9
Value:	5	10	18	30	45	50	60	65	70	80
	 LI			 HI						

Low Index (LI) = 0; High Index (HI) = $(4 - 1) = 3$

Choose element with index $(0 + 3) / 2 = 1$



Compare value at index 1 (**10**) with the key (**40**)

40 is greater than 10, so the target must be in the higher half of the (reduced) list

Binary Search: An Example (Key \notin List) (8)

Third Iteration: Lookup in the further reduced search space

Index:	0	1	2	3	4	5	6	7	8	9
Value:	5	10	18	30	45	50	60	65	70	80

Low Index (LI) = $1 + 1 = 2$; High Index (HI) = 3

Choose element with index $(2 + 3) / 2 = 2$



Compare value at index 2 (**18**) with the key (**40**)

40 is greater than 18, so the target must be in the higher half of the (reduced) list

Binary Search: An Example (Key \notin List) (9)

Fourth Iteration: Lookup in the further reduced search space

Index:	0	1	2	3	4	5	6	7	8	9
Value:	5	10	18	30	45	50	60	65	70	80

Low Index (LI) = $2 + 1 = 3$; High Index (HI) = 3

Choose element with index $(3 + 3) / 2 = 3$



Compare value at index 3 (**30**) with the key (**40**)

40 is greater than 30, so the target must be in the higher half of the (reduced) list

Binary Search: An Example (Key \notin List) (10)

Fifth Iteration: Lookup in the further reduced search space

Index:	0	1	2	3	4	5	6	7	8	9
Value:	5	10	18	30	45	50	60	65	70	80

Low Index (LI) = $3 + 1 = 4$; High Index (HI) = 3

Since $LI > HI$, Key does not exist in the list

Stop; Key is not found

Binary Search Algorithm: Informal (11)

- Middle □ $(LI + HI) / 2$
- One of the three possibilities
 - Key is equal to List[Middle]
 - success and stop
 - Key is less than List[Middle]
 - Key should be in the left half of List, or it does not exist
 - Key is greater than List[Middle]
 - Key should be in the right half of List, or it does not exist
- Termination Condition
 - List[Middle] is equal to Key (success) OR $LI > HI$ (Failure)

Binary Search Algorithm (12)

- Input: Key, List
- Initialisation: $LI \square 0$, $HI \square \text{SizeOf(List)} - 1$
- Repeat steps 1 and 2 until $LI > HI$
 1. $Mid \square (LI + HI) / 2$
 2. If $List[Mid] = Key$ then
 - Return Mid // *success*
 - Else If $Key < List[Mid]$ then
 - $HI \square Mid - 1$
 - Else
 - $LI \square Mid + 1$
- Return -1 // *failure*

Search Algorithms: Time Complexity

□ Time complexity of Sequential Search algorithm:

- Best-case : $O(1)$ comparison
 - target is found immediately at the first location
- Worst-case: $O(n)$ comparisons
 - Target is not found
- Average-case: $O(n)$ comparisons
 - Target is found somewhere in the middle

□ Time complexity of Binary Search algorithm:

$O(\log(n))$ □ This is worst-case