5SENG003W - Algorithms, Week 2

Dr. Klaus Draeger

RECAP

Last week...

- We introduced the idea of algorithms
- We saw how performance of algorithms can be compared
  - Big-O notation
  - Some important complexity classes (logarithmic, linear, quadratic, exponential, ...)
  - How to determine them empirically (doubling hypothesis)

Algorithms and data structures

- We have already seen how the **size** of the data affects the running time of algorithms
- The data **structure** can be just as important!
- Different data structures offer different performance for elementary operations like insertion or lookup
- Let's have a look at some examples
  - For simplicity, the data contained in the structures will be integers

What is a Data Structure?

- A **data structure** is used to organise and store bits of (usually related) **information**.
- Programs need this in order to process the information.
- Programming languages provide some **basic primitive data types** as part of the language, e.g. integers, characters, booleans, etc.
- More complicated structures are built from these using constructions like
  - **arrays** to have a number of values of the same type
  - **structs/classes** to group together data which are part of a bigger whole
    for example, a **student's record** containing: student number, personal details, course title, year of entry, list of modules and marks, etc.

What is a Data Structure?

- Objects often contain **pointers/references** to other objects in order to represent data structures.
- Example:

```java
public class Person{
    String name;
    int age;
    Person father, mother;
}
```

Each person has references to two others (which may be **null**), giving a (family) **tree**.

What is a Data Structure?

- The family tree from the previous slide is an example for a **linked** data structure as opposed to an **indexed** one (like arrays):
  - Indexed structures are accessed using the specific position: `a[17]` but also e.g. `M[2][3]` in a matrix (two-dimensional array)
  - Linked structures are explored by following the links (pointers/references)
- Other categories are sequential vs branching or static vs dynamic
  - Sequential structures have all data in a row, as opposed to trees/graphs which are branching
  - Static structures have a fixed sizewhile dynamic ones can grow or change in other ways

Sequential data structures: Arrays

- Arrays are one of the simplest data structures
- Array entries are laid out sequentially in memory:

i	0	1	2	3	4
a[i]	3	1	4	1	5

- Can access arbitrary entries in **constant time** ($O(1)$):
  - Size **s** of entries is fixed and known (e.g. 4 bytes for 32-bit ints)
  - Starting address **a** of the array is known at runtime
  - So entry number **i** starts at memory address **a + i * s**

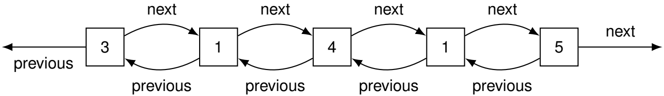
Sequential data structures: Arrays

- More versatile: Java ArrayLists and C++ vectors
  - These have flexible size, i.e. can grow to accomodate additional data
  - Internally use a regular array which gets replaced with a bigger one when needed.
- Some important basic operations on arrays:
  - Reading/writing at any given index: e.g. **`a[i] = a[i+1];`**
    Cost: still constant – independent of array size
  - Adding/removing at the end (in ArrayLists or vectors)
    Cost: constant – independent of array size
  - Adding/removing in the middle
    Cost: **linear** – must move the rest of the contents!

Sequential data structures: Arrays and linked lists

- Linked lists store data in **nodes**:

```
public class ListNode{
    private int data;          // contents of this node
    private ListNode next;
    private ListNode previous;
    // ...
}
```



- Some important basic operations on linked lists:
  - Reading/writing at any given index:
    Cost: **linear** – must traverse the list to find the position!
  - Adding/removing at the end
    Cost: constant – independent of list size
  - Adding/removing in the middle
    Cost: constant – assuming we already have an **iterator** pointing at the position

Algorithms and data structures

- Choice of data structure can influence what algorithms are suitable
- Typically there is a trade-off between complexities of operations
- We will be seeing other (non-sequential) structures later, like **trees** and **graphs**
- Besides structure, the **supported operations** are an important characteristic of data structures
  - A sequence could be a queue, a stack, ...
  - A tree could be a binary search tree, heap, ...

Searching

- Common problem:
  - Given: a data structure, a piece of data
  - Question: Is that piece of data in the structure?
    If yes, where?
- Case 1: array

```
public static int findLocation(int[] values,
                               int findMe){
    /* What goes here? */
}
```

Searching

- Common problem:
  - Given: a data structure, a piece of data
  - Question: Is that piece of data in the structure?
    If yes, where?
- Case 1: array – simplest solution:

```java
public static int findLocation(int[] values,
                               int findMe){
    for(int i=0; i<values.length; i++)
        if(values[i] == findMe)
            return i;
    return -1;  // indicates failure
}
```

Searching

- Given: a data structure, a piece of data
- Question: Is that piece of data in the structure?
  If yes, where?
- Case 2: Linked list

```
public class ListNode{
    private int data;
    private ListNode next;
    private ListNode previous;

    public ListNode findLocation(int findMe){
    /* What goes here? */
    }
}
```

Searching

- Given: a data structure, a piece of data
- Question: Is that piece of data in the structure?
  If yes, where?
- Case 2: Linked list – simplest solution

```
public class ListNode{
    private int data;
    private ListNode next;
    private ListNode previous;

    public ListNode findLocation(int findMe){
        ListNode n = this;
        while((n != null) && (n.data != findMe))
            n = n.next;
        return n;    //  null indicates failure
    }
}
```

Brute force algorithms

- Linear search is an example of a **Brute Force** algorithm:
  - In order to find a solution, look at all possible candidates
- On the one hand, this works for many different data structures
- On the other hand, it ignores the benefits that suitable data structures offer
- Often the **easiest** algorithm to implement
- Usually not optimal

Linear vs binary search

- We had to potentially look at each value in the array/list.
- Can we do better? **Yes**, if the container is:
  - indexable
  - sorted
- Idea:
  - Check the middle element
    (This is what we need indexability for)
  - If it is. . .
    - what we are looking for: success, return this index
    - less: repeat with just the right half
    - greater: repeat with just the left half
  - Repeat until no candidates left
    - Return -1 to indicate failure if this happens

Binary search: example

- Searching for 9 in this array:

0	1	2	3	4	5	6	7	8	9	10	11
first					mid						last

- 5 is less than 9, so focus on the right half:

0	1	2	3	4	5	6	7	8	9	10	11
						first		mid			last

- 8 is less than 9, so focus on the right half:

0	1	2	3	4	5	6	7	8	9	10	11
									first	mid	last

- 10 is greater than 9, so focus on the left half:

0	1	2	3	4	5	6	7	8	9	10	11
									first mid last		

Binary search: implementation

```java
public int binarySearch(int[] values, int findMe){
    // Everything between first and last is a candidate
    // Initially, that is the whole array
    int first = 0, last = values.length - 1;
    // As long as there are candidates...
    while(first <= last){
        // ...check the middle of the range.
        int middle = (first + last) / 2;
        // If it is what we are looking for: success
        if(values[middle] == findMe)
            return middle;
        // If it is less: repeat with the right half
        else if(values[middle] < findMe)
            first = middle + 1;
        // Else (i.e. it is greater): repeat with the left half
        else
            last = middle - 1;
    }
    return -1;  // Indicates failure: no more candidates left
}
```

Binary search

- Each iteration cuts the search range in half
- If the array has size n, and n is less than 2^k:
  - After 1 iteration there are less than 2^{k-1} candidates left
  - After 2 iterations there are less than 2^{k-2} candidates left
  - ...
  - After $k-1$ iterations there are less than 2 candidates left
- The search finishes after at most $k = \lceil \log n \rceil$ iterations
- So complexity of binary search is $O(\log n)$
- It relies on the data being indexable and sorted
  - Think about what happens if we try this with a linked list
  - We will be looking at sorting soon

Divide and conquer

- While linear search is a Brute Force algorithm, binary search uses a strategy called **Divide and Conquer**:
  In order to solve a problem on a data structure,
  1. Split the structure into smaller parts
  2. Solve the problem on each part
  3. Get an overall solution from the partial solutions
- How each step is done depends on the problem
- In binary search, the steps boil down to
  1. If not found yet, split the array into the low and high half
  2. Ignore the half where the value cannot be, use recursion on the other
  3. Overall solution is the solution from the relevant half
- Divide and Conquer is an important strategy, we will see more examples later!