Searching Data

Searching techniques

- · Searching algorithms in unordered "lists"
 - "Lists" may be arrays or linked lists
 - The algorithms are adaptable to either
- · Searching algorithms in ordered "lists"
 - Primarily arrays with values sorted into order
 - We can exploit the order to search more efficiently
- · We will only consider arrays
 - The data held in the array might be

Simple (e.g. numbers, strings) or more complex objects: the search will probably be based on a chosen *key field* in the objects (eg, search student records by registration number)

- We will only consider simple data - the searching techniques are the same

Sequential search in an unordered list

- In these algorithms we will assume:
 - The data is integers
 - Held in an array variable numbers,
 - Is in random order
 - The number of data values is indicated by a variable size
 - The data is in elements indexed 0 to (size-1)
 - We are seeking the integer held in variable val
- The basic technique to be used is "sequential search"
 - Compare val with the value in numbers [0], then with that in numbers [1], etc
- · We will look at two versions of an algorithm encoding this
 - Other adaptations are possible

Algorithm 1: Standard sequential search

 Here is a basic search algorithm. It leaves its result in a variable called position:

```
int position = 0;
while (position < size)
{
  if (numbers[position] == val)
    break; // Exit loop if found
  position++;
}</pre>
```

- If val is not present:
 - The entire array will be scanned taking size steps
 - position will have a final value of size
- But if **val** is present:
 - break; => the while loop terminates immediately
 - The average number of scanning steps expected is size/2
- Easy to adapt to return a boolean, or throw an exception

• If we are careful, we can combine the loop test and the array element check:

```
int position = 0;
while (position < size && numbers[position] != val)
    position++;
- The && test checks position < size first,
- and if it is false does not check
    numbers[position] != val
- otherwise would get
    ArrayIndexOutOfBoundsException if val is not
    present!
- This is called "conditional" or "short-circuit" behaviour:
    it applies to && and | |</pre>
```

Algorithm 2: Sequential search with a "sentinel"

- We can improve the basic search algorithm if the array numbers has one extra element, numbers [size], that is never used for actual data
 - Instead we place a copy of the sought value there, so the search *always succeeds*. This means that the loop does not need to carry out the "end of array" test less work, so quicker.

```
int[] numbers = new int[size+1];
...
int position = 0;
numbers[size] = val; // Insert "sentinel"
while (numbers[position] != val)
    position++;
```

- As before, position has the final value size if val is not present

Complexity Analysis (Reminder)

- We try to identify how the time taken to execute an algorithm depends on the quantity of data to be processed
 - True "time" is tricky we work in terms of abstract steps
 - One abstract step should correspond to a consistent "unit of time"
- If the number of data items is N then:
 - If the number of steps is independent of the quantity of data (eg, 6 steps)
 then we have a constant time algorithm, and we state that the "order of complexity" is O(1) (pronounced "order one")
 - If the number of steps is proportional to the quantity of data (eg, N+2, N/2, 3N) then we have a linear algorithm, and we state that the order of complexity is O(N) (pronounced "order N")

7

Complexity Analysis (Reminder)

- The order of complexity does not usually tell us exactly how many steps are taken
 - But it does indicate how the number of steps taken varies with the quantity of data
 - It tells us how the overall time will vary with the quantity of data
- · Examples:
 - If an algorithm is constant time,O(1), and it takes 1 second to process 10 items of data, then it will take 1 second to process 1000 items of data
 - If an algorithm is linear, O(N), and it takes 1 second to process 10 items of data, then it will take 100 seconds to process 1000 items of data

Complexity Analysis (Reminder)

- We may be interested in an algorithm's best case, worst case or average execution time:
- For the sequential search algorithm (with or without sentinel):
 - Best case is 1 step: O(1)
 - Worst case is N steps: O(N)
 - The actual average number of steps depends on ratio of successful/unsuccessful searches:

The average of successful searches is N/2 steps, and so is O(N)

All unsuccessful searches take N steps, which is O(N), So overall the average complexity is O(N)

9

Searching an Ordered List

- · Again we will assume:
 - The data is integers, held in an array sequence,
 - So the data is in elements indexed 0 to (length-1)
 - But this time we assume that the values are held in ascending numerical order
 - We are seeking the integer held in val
- We could use the sequential search algorithm, but this does not take advantage of the knowledge that the data is ordered. (The complexity remains O(N).)
- Instead, we will take advantage of the ordering to improve search efficiency (ie, to reduce the complexity)

Binary Search

- If the data is already ordered, we can do much better than a linear time algorithm. Here is the scheme:
 - Pick the middle element in the array
 - If it is equal to val, stop the search
 - If it is greater than val, search the lower half of the remaining array
 - If it is less than val, search the remaining upper half
- · At each iteration:
 - We are searching in a remaining partition of the array
 - We cut the remaining partition in half, rather than just removing one element
- · Example: Searching for 11 in

```
1, 3, 5, 7, 9, 11, 13
```

- First compare with 7, so search in 9, 11, 13
- Now compare with 11 found it in two steps

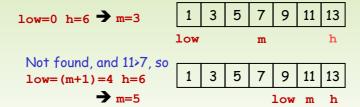
1

Binary Search

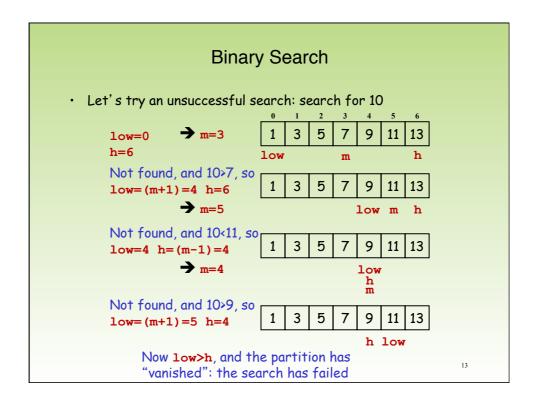
- · Concretely:
 - Let variable **low** indicate the lowest element of the partition (index **0** initially)

high (h) indicate the highest element (size-1 initially)
middle (m) indicate the next element being tested

- The search for 11 proceeds like this:



Found it, at index 5



```
Binary Search
· Algorithm binarySearch:
    INPUT: val - value of interest, sequence - sorted data
    OUTPUT: object or value of interest if exists, null otherwise
        int low = 0, middle = 0, high = seq.length;
        while (high >= low) {
          middle = (high + low) / 2;
          if (sequence[middle] == val)
  return sequence[middle];
                                               // Found it
          else if (sequence[middle] < val)
  low = middle + 1;</pre>
                                                 // Search upper half
            high = middle - 1;
                                                // Search lower half
        return null;
· The outcomes:
    - Ordinary loop exit when the indexes "cross" → not found (i.e. high < low)
    - Loop exit on return → found (detect this by testing high >= low)
                                                                                14
```

The Complexity of Binary Search Best case: val is exactly sequence [middle] at the first step - The search stops after first step, so complexity O(1) Worst case:

- This will be when we continue dividing until the "partition" contains only one value: then it is either equal to val or not
- For 250 elements this turns out to be about 8 iterations
- For 500 it is about 9
- For 1000 it is about 10
- Double the amount of data → Add one step!
- In general: the size is approximately 2^{steps}
- So the number of steps is approximately log_2 size
- Complexity is $O(\log_2 N)$
- For emphasis: double the amount of data -> Add one step!
- · Average case:
 - Don't need to consider this: the worst case is very good!

15

End of Lecture