Data Structures - Arrays 02

Dr. TGI Fernando [1] [2]

[1] Email: tgi.fernando@gmail.com
[2] URL: http://tgifernando.wordpress.com/

Ordered Array

An array in which data items are arranged in a particular order.

E.g. ascending order - The smallest value at index 0, and each cell holding a value larger than the cell below.

Why ... ordered array?
To speed up the searching dramatically by using the binary search.

Insertion

- The correct location must be found for the insertion: **just above a smaller value** and **just below a larger one**.
- Then all the larger values must be moved up one cell to make room the new item.

Searching - two ways to search an ordered array: **linear** and **binary**

Linear Search

Operate in much the same way as the searches in an unordered array.
The difference is that in the ordered array, the search quits if an item with a larger key is found.

Non-existent item - search terminates when the first item larger than the search key is reached. There is no point of looking further.

Binary Search

The payoff for using an ordered array comes when we use a binary search. This kind of search is much faster than a linear search, especially for large arrays.

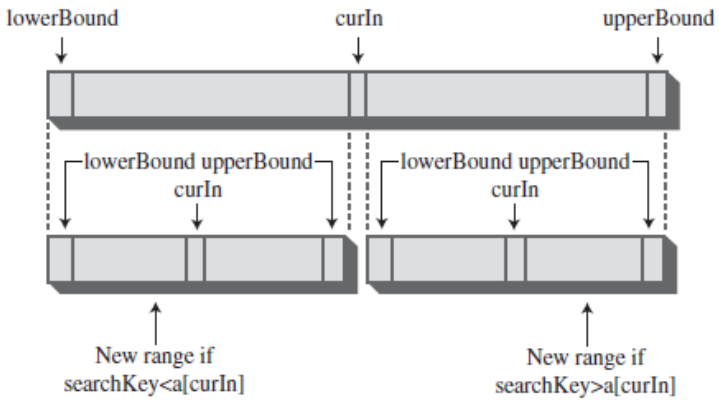
Guessing a number game - Let's say that number to be guessed is 33 (less than 100).

Step Number	Number Guessed	Result	Range of Possible Values
0			1–100
1	50	Too high	1–49
2	25	Too low	26–49
3	37	Too high	26–36
4	31	Too low	32–36
5	34	Too high	32–33
6	32	Too low	33–33
7	33	Correct	

The correct number is identified in only seven guesses. This is the maximum.

Binary Search (Contd.)

The method begins by setting **'lowerBound'** and **'upperBound'** variables to the first and the last occupied cells in the array.



Then the current index **'curIn'** is set to the middle of this range **[lowerBound, upperBound]**.

Binary Search (Contd.)

If **curIn** is pointing to the desired item, the method returns **curIn** as the output.

Otherwise, the method must find the desired item is in the first half of the array or the second half of the array.

- First half - Set upperBound = curIn - 1
- Second half - Set lowerBound = curIn + 1

Set **curIn** as the middle of the new range **[lowerBound, upperBound]**.

Repeat this process until the method found the desired item or **lowerBound** > **upperBound** (desired item is not found).

Output:

- Desired item found - **curIn**
- Desired item is not found - **nElems** (no of elements in the array) - this is not a valid index

Deletion

Deletion works much the same as it did in the unordered arrays, shifting items with higher index numbers down to fill in the hole left by the deletion.

In the ordered array, however, the deletion algorithm can quit partway through if it doesn't find the item, just as the search routine.

The **delete()** method could call **find()** method [search method] to figure out the location of the item to be deleted.

size() method - returns the number of data items (**nElems**) currently in the array.

This information is helpful for the class user (e.g. main()) when it calls find() method to search for an item that is not in the array.

Lab Work 03 - OrdArray

Description: The class 'OrdArray' given in 'OrdArrayApp.txt' is written to implement an ordered array.

Tasks: Complete the coding of classes 'OrdArray' and 'OrdArrayApp' given in 'OrdArrayApp.txt' text file.

Pros and cons of ordered arrays

The major advantage is that search times are much faster than in an unordered array.

The disadvantage is that the insertion takes longer because all the data items with a higher key value must be moved up to make a room for the new one.

Deletions are slow in both ordered and unordered arrays because items must be moved down to fill the hole left by the deleted item.

Useful in situations in which searches are frequent, but insertions and deletions are not.

Pros and cons of ordered arrays (Contd.)

E.g. 1: Database of company employees - Hiring new employees and laying off existing ones would probably be infrequent occurrences compared with accessing an existing employee's record for information, or updating it to reflect changes in salary, address, and so on.

E.g. 2: Retail store inventory - would not be a good candidate for an ordered array because the frequent insertions and deletions, as items arrived in the store and were sold, would run slowly.

Comparisons - Binary search Vs. Linear search

Range	Binary Search (Max)	Linear Search (Avg. $= N/2$)
10	4	5
100	7	50
1,000	10	500
10,000	14	5,000
100,000	17	50,000
1,000,000	20	500,000
100,000,000	24	5,000,000

Powers of 2

What is the exact size of the maximum range that can be searched in 5 steps?

Step s, same as $\log_2(r)$	Range r	Range Expressed as Power of 2 (2^s)
0	1	2^0
1	2	2^1
2	4	2^2
3	8	2^3
4	16	2^4
5	32	2^5
6	64	2^6
7	128	2^7
8	256	2^8
9	512	2^9
10	1024	2^{10}

If s represents no of steps and r represents the range, then the equation is

$$r = 2^s$$

The opposite of raising two to a power

Our original question was, given the range (r), how many comparisons are required to complete a search?

i.e. given r, we want an equation that gives us s.

$$r = 2^s$$
$$\Rightarrow s = \log_2(r)$$

E.g.
If $r = 100$,

$$\begin{aligned} s &= \log_2(100) \\ &= \log_{10}(100) \times 3.322 \\ &= 2 \times 3.322 \\ &= 6.644 \end{aligned}$$

When we rounded up this number, we get 7.

Big O Notation

It's useful to have a shorthand way to say how efficient a computer algorithm is. In computer science, this rough measure is called "Big O" notation.

We need a comparison that tells how an algorithm's speed is related to the number of items.

Insertion in an unordered array

Does not depend on how many items are in an array.

The new item always placed in the next available position, at **a[nElems]**, and **nElems** is then incremented.

We can say the time, T, to insert an item into an unsorted array is a constant K:

$$T = K$$

Big O Notation (Contd.)

In a real situation, the actual time required by the insertion is related to

- speed of the microprocessor,
- how efficient the compiler has generated the program code, and
- other factors.

The constant K in the preceding equation is used to take into account for all such factors.

Linear search: proportional to N

The number of comparisons that must be made to find a specified item is, on average, half of the total number of items.

$$\begin{aligned} \text{i.e. } T &= K' \times N/2 \\ &= K \times N \text{ (Letting } K = K'/2) \end{aligned}$$

Big O Notation (Contd.)

Binary Search: Proportional to log(N)
Similarly, we can derive a formula relating T and N for a binary search:

$$\begin{aligned} T &= K' \times \log_2(N) \\ &= K' \times 3.322 \times \log(N) \\ &= K \times \log(N) \text{ (Letting } K = K' \times 3.322) \end{aligned}$$

Don't need the constant

When comparing algorithms, we don't really care about the particular microprocessor chip or compiler; all we want to compare is **how T changes for different values of N**, not what the actual numbers are. Therefore, the constant isn't needed.

Big O Notation (Contd.)

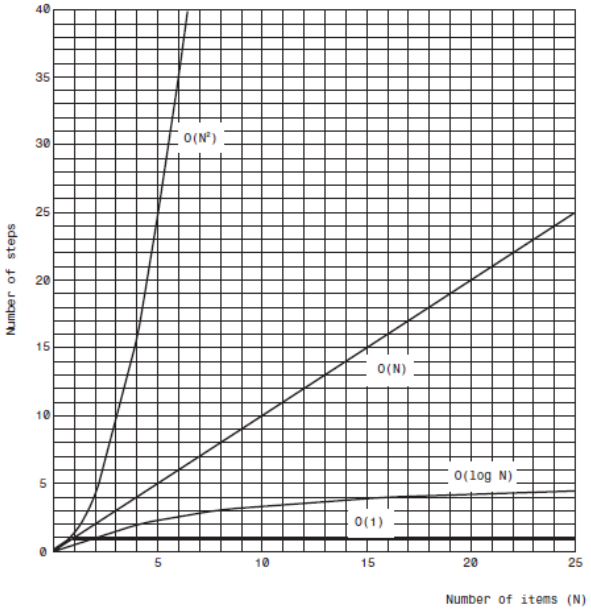
Big O notation uses uppercase letter O, which means "order of."

Running times in Big O notation

Algorithm	Running Time in Big O Notation
Linear search	O(N)
Binary search	O(log N)
Insertion in unordered array	O(1)
Insertion in ordered array	O(N)
Deletion in unordered array	O(N)
Deletion in ordered array	O(N)

We might rate Big O values like this:
$O(1)$ - excellent
$O(\log N)$ - good
$O(N)$ - fair
$O(N^2)$ - poor

Graph of Big O times



Why not arrays for everything

Some operations take $O(N)$ time - It would be nice if there were data structures that could do everything - insertion, deletion and searching - quickly, ideally $O(1)$ time, but if not that, then in $O(\log N)$ time.

	Unordered	Ordered
Insertion	$O(1)$	$O(N)$
Searching	$O(N)$	$O(\log N)$
Deletion	$O(N)$	$O(N)$

Array size is fixed - Usually when the program starts initially, we don't know exactly how many items will be needed in future, so we should guess this number. If this guess is too large, we will waste the memory. If your guess is too small, our program overflow the array.