

Stack Overflow is a community of 4.7 million programmers, just like you, helping each other. Join them, it only takes a minute:

Sign up ✕

What are the time complexities of various data structures?



I am trying to list time complexities of operations of common data structures like Arrays, Binary Search Tree, Heap, Linked List, etc. and especially I am referring to Java. They are very common, but I guess some of us are not 100% confident about the exact answer. Any help, especially references, is greatly appreciated.

E.g. For singly linked list: Changing an internal element is $O(1)$. How can you do it? You *HAVE* to search the element before changing it. Also, for the Vector, adding an internal element is given as $O(n)$. But why can't we do it in amortized constant time using the index? Please correct me if I am missing something.

I am posting my findings/guesses as the first answer.

java data-structures time-complexity

edited Sep 9 '13 at 16:57

asked Sep 3 '11 at 17:19



Bhushan

5,001 9 43 87

1 When you vote a question to close, at least add a comment. — Bhushan Sep 3 '11 at 18:29

Downvoted for the reason given in my answer. — EJP Sep 4 '11 at 9:54

1 Time and Space Complexities for all data structures [Big O cheat sheet](#) — vbp Feb 26 '14 at 16:25

In case someone else steps into this, take a minute to also check this link: infotechgems.blogspot.gr/2011/11/ — vefthym Apr 7 '14 at 8:12

2 Answers

Arrays

- **Set, Check** element at a particular index: $O(1)$
- **Searching:** $O(n)$ if array is unsorted and $O(\log n)$ if array is sorted and something like a binary search is used,
- As pointed out by Aivean, there is no `Delete` operation available on Arrays. We can symbolically delete an element by setting it to some specific value, e.g. -1, 0, etc. depending on our requirements
- Similarly, `Insert` for arrays is basically `Set` as mentioned in the beginning

ArrayList:

- **Add:** Amortized $O(1)$
- **Remove:** $O(n)$
- **Contains:** $O(n)$
- **Size:** $O(1)$

Linked List:

- **Inserting:** $O(1)$, if done at the head, $O(n)$ if anywhere else since we have to reach that position by traversing the linkedlist linearly.
- **Deleting:** $O(1)$, if done at the head, $O(n)$ if anywhere else since we have to reach that position by traversing the linkedlist linearly.
- **Searching:** $O(n)$

Doubly-Linked List:

- **Inserting:** $O(1)$, if done at the head or tail, $O(n)$ if anywhere else since we have to reach that position by traversing the linkedlist linearly.
- **Deleting:** $O(1)$, if done at the head or tail, $O(n)$ if anywhere else since we have to reach that position by traversing the linkedlist linearly.
- **Searching:** $O(n)$

Stack:

- **Push:** $O(1)$
- **Pop:** $O(1)$
- **Top:** $O(1)$
- **Search** (Something like lookup, as a special operation): $O(n)$ (I guess so)

Queue/Deque/Circular Queue:

- **Insert:** $O(1)$
- **Remove:** $O(1)$
- **Size:** $O(1)$

Binary Search Tree:

- **Insert, delete and search:** Average case: $O(\log n)$, Worst Case: $O(n)$

Red-Black Tree:

- **Insert, delete and search:** Average case: $O(\log n)$, Worst Case: $O(\log n)$

Heap/PriorityQueue (min/max):

- **Find Min/Find Max:** $O(1)$
- **Insert:** $O(\log n)$
- **Delete Min/Delete Max:** $O(\log n)$
- **Extract Min/Extract Max:** $O(\log n)$
- **Lookup, Delete** (if at all provided): $O(n)$, we will have to scan all the elements as they are not ordered like BST

HashMap/Hashtable/HashSet:

- **Insert/Delete:** $O(1)$ amortized
- **Re-size/hash:** $O(n)$
- **Contains:** $O(1)$

edited Jul 21 at 19:10

answered Sep 3 '11 at 17:19



Bhushan

5,001 9 43 87

2 awesome answer. Thanks a lot +1 for you. – [Nikhil Agrawal](#) Apr 24 '13 at 8:58

2 Good reference, to recall things... – [Dreamer](#) Jul 6 '14 at 19:09

Inserting an element into Array (and by *insert* I mean adding new element into position, shifting all elements to the right) will take $O(n)$. Same for deletion. Only replacing existent element will take $O(n)$. Also it's possible that you mixed it with adding new element to resizable array (it has amortized $O(1)$ time). – [Aivean](#) Sep 23 '14 at 17:23

Also please note, that for Doubly-linked list inserting and deleting to both head and tail will take $O(1)$ (you mentioned only head). – [Aivean](#) Sep 23 '14 at 17:26

And final note, balanced search trees (for example, Red-black tree that is actually used for TreeMap in Java) has guaranteed worst-case time of $O(\ln n)$ for all operations. – [Aivean](#) Sep 23 '14 at 17:28



Did you find this question interesting? Try our newsletter

Sign up for our newsletter and get our top new questions delivered to your inbox ([see an example](#)).

Arrays

Set, Check element at a particular index: $O(1)$
Searching: $O(n)$ **if** array is unsorted and $O(\log n)$ **if** array is sorted and something like a binary search is used,
As pointed out by **Aivean**, there is no **Delete** operation available on **Arrays**. We can symbolically **delete** an element by setting it to some specific value, e.g. **-1**, **0**, etc. depending on our requirements
Similarly, **Insert** **for** arrays is basically **Set** as mentioned in the beginning

ArrayList:

Add: Amortized $O(1)$
Remove: $O(n)$
Contains: $O(n)$
Size: $O(1)$

Linked List:

Inserting: $O(1)$, **if** done at the head, $O(n)$ **if** anywhere **else** since we have to reach that position by traversing the linkedlist linearly.
Deleting: $O(1)$, **if** done at the head or tail, $O(n)$ **if** anywhere **else** since we have to reach that position by traversing the linkedlist linearly.
Searching: $O(n)$

Doubly-Linked List:

Inserting: $O(1)$, **if** done at the head or tail, $O(n)$ **if** anywhere **else** since we have to reach that position by traversing the linkedlist linearly.
Deleting: $O(1)$, **if** done at the head or tail, $O(n)$ **if** anywhere **else** since we have to reach that position by traversing the linkedlist linearly.
Searching: $O(n)$

answered Nov 10 '14 at 7:08



space complexity
21 1

1 I think that's exactly what the already posted answer says? – **Bhushan** Nov 10 '14 at 22:23
