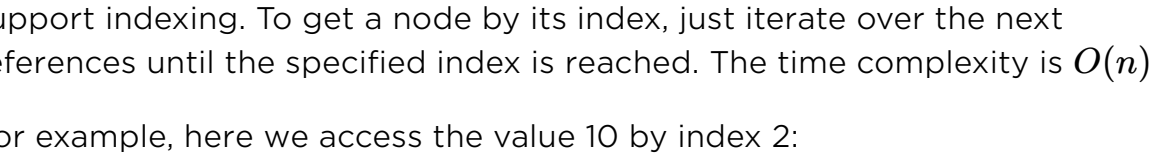
Theory: Singly linked list

7 minutes 0 / 3 problems solved

§1. Essentials

A **singly linked list** is a linear data structure represented as a sequence of connected nodes. Each node stores data and a reference to the next node in a list, but the last one does not refer to any other node. The first node of a list is usually called a **head**.

The following picture demonstrates a list with four nodes that store numbers 18, 15, 10, 21:



Here are some important points about a linked list:

- it keeps a precedence order that is important in some applications;
- nodes can be stored anywhere in the memory unlike arrays where items are located sequentially;
- it is dynamic, so the length can be increased or decreased during its existence.

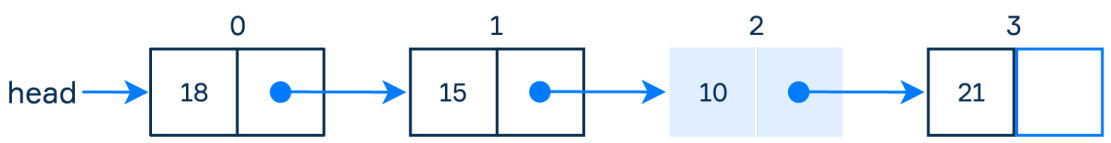
Every linked list supports adding and removing elements as well as getting the size. Some operations require a list traversal which can be performed using a loop starting from the head.

Since lists are widely used in programming practice, it is important to understand their essentials. In a way, a real-life analogy of a linked list is a train that has a locomotive (head) and a series of connected cars (nodes) with people on board (data).

§2. Supporting indexes

Linked lists do not allow random access to the data, but they can still support indexing. To get a node by its index, just iterate over the next references until the specified index is reached. The time complexity is $O(n)$.

For example, here we access the value 10 by index 2:



We start from the head at index 0 and follow the links exactly two times (0→1, 1→2) since we need to access the node at index 2. The result is 10.

§3. Adding elements

There are three possible ways to add a new element to a linked list:

- at the top of the list
- at the end of the list
- at any other place specified by an index.

Many programming sources call these operations *prepend*, *append* and *insert*. Let's take a closer look at them.

Top of the list. All we need is to create a node for the new element, set its *next* to the old head of the list, and make the new element the head (updating reference to the head of the list).

The following example shows how item 30 is added to the list.

Current topic:

Singly linked list

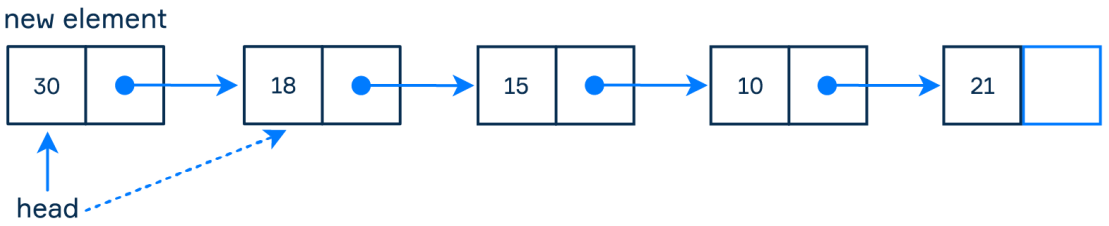
Topic depends on:

✓ Data structures

Topic is required for:

Doubly linked list

Table of contents:

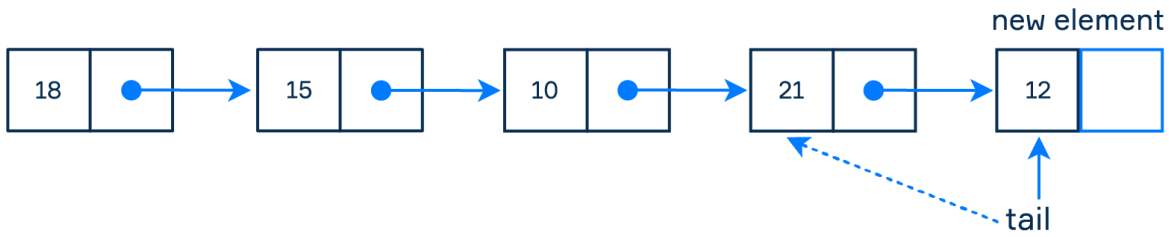


The time complexity of adding a new first element is $O(1)$ because we just update references.

End of the list. This operation is similar to the previous one, but from the other side – create a node for the new element and set *next* reference of the current last node to the new node.

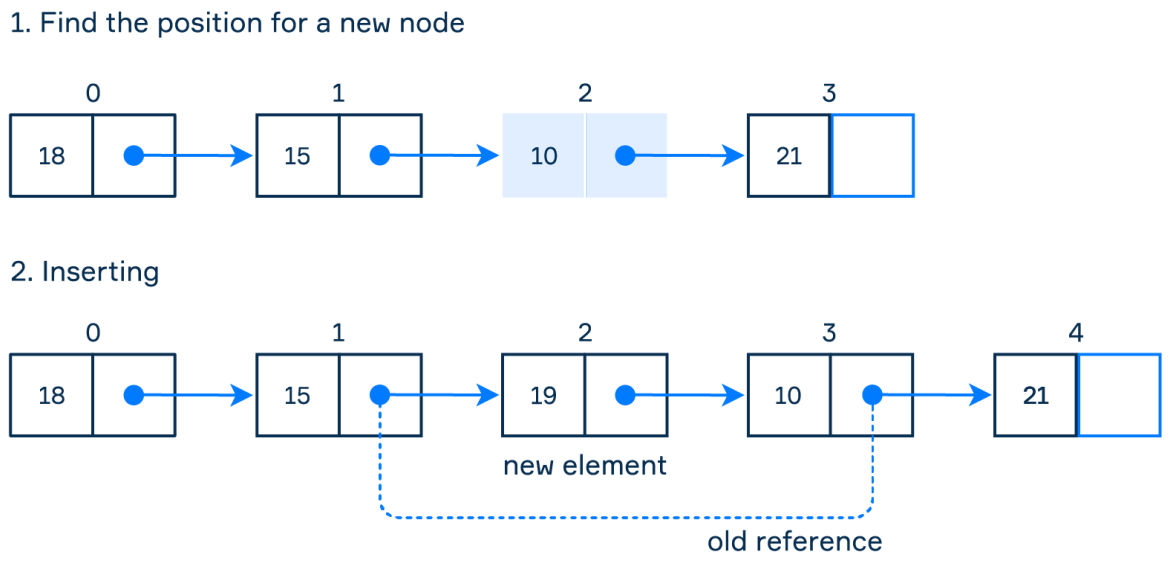
If we store the reference to the actual last element (**tail**), we do not need to iterate over the entire list. In this case, the time complexity is $O(1)$. But we also should not forget to update the tail reference. If we don't keep a reference to the tail, first we must find the last node starting from the head, and then do the same as above. In such a case, the time complexity is $O(n)$ where n is the number of elements in the list.

Here is an example where the value 12 is added at the end of a list:



By index. First, we need to find the node with the required index: we go through the *next* references starting from the head. Once we have reached it, we create a new node and update the references: the previous node should refer to the new node, and the new node should refer to the one we were looking for. Now, the new node is specified by the required index.

The following picture demonstrates how to add a new element with the value 19 by index 2:



This operation takes $O(n)$ time since we first need to find the position to insert.

§4. Removing elements

There are several possibilities to delete an element. You can remove: (1) the head, (2) the tail, (3) an element by a specified index, (4) an element by value, and so on. Like adding operations, removing operations are performed by manipulating references between list nodes. We will not consider it in this topic.

Report a typo

59 users liked this theory. **3** didn't like it. **What about you?**

Start practicing

Comments (3) Hints (0) Useful links (0) Show discussion