Linked Lists

Step 1

Recall from the previous section that **Array Lists** are excellent if we know exactly how many elements we want to store, but if we don't, they can be problematic either in terms of time complexity (rebuilding the backing array as we need to grow it) or in terms of space complexity (allocating more space than we need just in case).

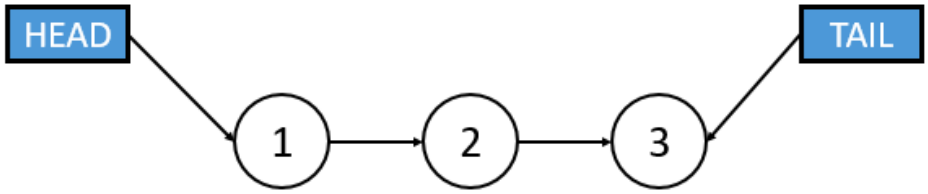
We will now introduce a data structure called a **Linked List**, which was developed in 1955 by Allen Newell, Cliff Shaw, and Herbert A. Simon at RAND Corporation. The **Linked List** is a dynamically-allocated data structures, meaning it grows dynamically in memory on its own very time-efficiently (as opposed to an **Array List**, for which we needed to explicitly reallocate memory as the backing array fills, which is a very time-costly operation). When analyzing various aspects of the **Linked List**, try to keep the **Array List** in mind because both are "list" structures, so they can effectively do the same things, but they each have their respective pros and cons.

Figure: Allen Newell (right) and Herbert Simon (left) (courtesy of Carnegie Mellon University)

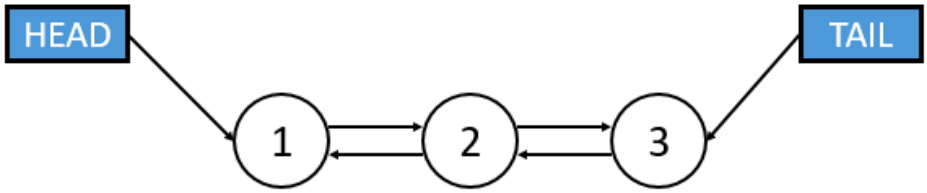
Step 2

A **Linked List** is a data structure composed of **nodes**: containers that each hold a single element. These nodes are "linked" to one another via pointers. Typically, we maintain one global ***head* pointer** (i.e., a pointer to the first node in the Linked List) and one global ***tail* pointer** (i.e., a pointer to the last node in a Linked List). These are the only two nodes we have direct access too, and all other nodes can only be accessed by traversing pointers starting at either the head or the tail node.

In a **Singly-Linked List**, each node only has a pointer pointing to the node directly after it (for internal nodes) or a null pointer (for the tail node). As a result, we can only traverse a Singly-Linked List in one direction.



In a **Doubly-Linked List**, each node has two pointers: one pointing to the node directly after it and one pointing to the node directly before it (of course, the head and the tail nodes only have a single pointer each). As a result, we can traverse a Doubly-Linked List in both directions.



Step 3

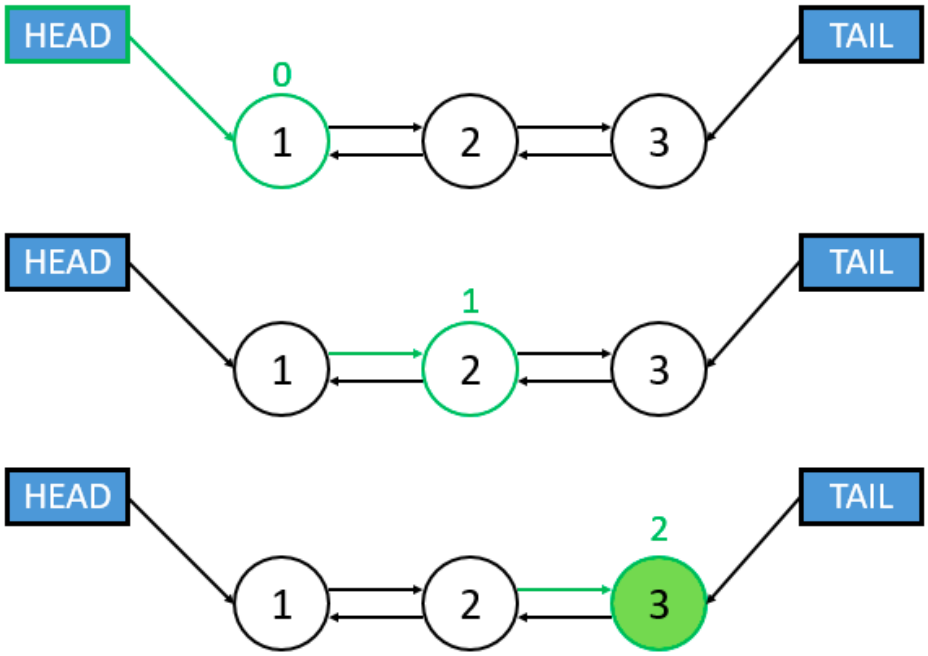
EXERCISE BREAK: If I only have direct access to the *head* or *tail* pointer in a **Linked List**, and I can only access the other elements by following the nodes' pointers, what is worst-case time complexity of finding an element in a **Linked List** with *n* elements?

To solve this problem please visit https://stepik.org/lesson/28867/step/3

Step 4

As you may have inferred in the previous step, to find an arbitrary element in a **Linked List**, because we don't have direct access to the internal nodes (we only have direct access to *head* and *tail*), we have to iterate through all *n* elements in the worst case.

Below is an example **Linked List**, in which we will find the element at index *i* = 2 (where indexing starts at *i* = 0):



STOP and Think: Notice that when we look for an element in the middle of the **Linked List**, we start at the *head* and step forward. This works fine if the index is towards the beginning of the list, but what about when the index of interest is large (i.e., closer to *n*)? Can we speed things up?

Step 5

As mentioned previously, to find an element at a given index in a **Linked List**, we simply start at the *head* node and follow forward pointers until we have reached the desired node. Note that, if we are looking for the *i*-th element of a **Linked List**, this is a O(*i*) operation, whereas in an array, accessing the *i*-th element was a O(1) operation (because of "random access"). This is one main drawback of a **Linked List**: even if we know exactly what index we want to access, because the data is not stored contiguously in memory, we need to slowly iterate through the elements one-by-one until we reach the node we want.

Below is pseudocode for the "find" operation of a **Linked List**:

```
find(element): // returns True if element exists in Linked List, otherwise returns False
    current = head                  // start at the "head" node
    while current is not NULL:
        if current.data == element:
            return True
        current = current.next      // follow the forward pointer of current
    return False                    // we iterated over all nodes but did not find "element"
```

If we want to find what element is at a given "index" of the **Linked List**, we can perform a similar algorithm, where we start at the *head* node and iterate through the nodes via their forward pointers until we find the element we desire. Note that in the pseudocode below, we use 0-based indexing.

```
find(index): // returns element at position "index" of Linked List (or NULL if invalid)
    if index < 0 or index >= n: // check for invalid indices
        return NULL             // we will use NULL to denote an invalid node

    curr = head                 // looking for an element in the middle of the Linked List
    repeat index-1 times:       // move forward index-1 times
        curr = curr.next
    return curr
```

Step 6

CODE CHALLENGE: Finding an Element in a Linked List

We have defined the following **Singly Linked List Node** C++ class for you:

```
class Node {
    public:
        int value;
        Node* next = NULL;
};
```

Write a function `find(Node* node, int element)` that starts at the given node and either returns true if the element exists somewhere in the **Linked List**, otherwise false if the element does not exist in the **Linked List**. You may choose to implement it either iteratively or recursively: we will pass in the *head* node when we call your `find` function, so both approaches have equally valid solutions.

Sample Input:

```
2
0 -> 1 -> 2 -> 3 -> 4 -> 5
```

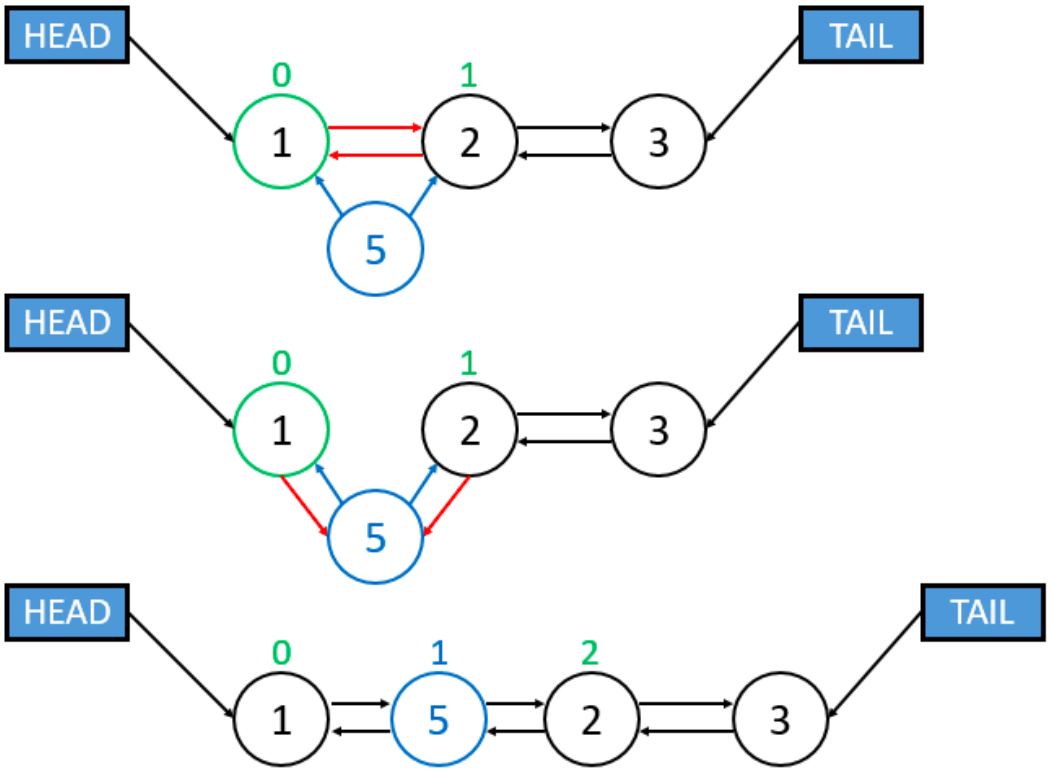
Sample Output:

```
true
```

To solve this problem please visit https://stepik.org/lesson/28867/step/6

Step 7

The "insert" algorithm is almost identical to the "find" algorithm: you first execute the "find" algorithm just like before, but once you find the insertion site, you rearrange pointers to fit the new node in its rightful spot. Below is an example of inserting the number 5 to index 1 of the **Linked List** (using 0-based counting):



Notice how we do a regular "find" operation to the index directly *before* index *i*, then we point the new node's "next" pointer to the node that was previously at index *i* (and the new node's "prev" pointer to the node that is directly *before* index *i* in the case of a **Doubly-Linked List**, as above), and then we point the "next" pointer of the node before the insertion site to point to the new node (and the "prev" pointer of the node previously at index *i* to point to the new node in the case of a **Doubly-Linked List**).

Because the structure of a **Linked List** is only based on pointers, the insertion algorithm is complete simply after changing those pointers.

Step 8

Below is pseudocode for the "insert" operation of a **Linked List** (with added corner cases for the first and last indices):

```
insert(newnode, index): // inserts newnode at position "index" of Linked List
    if index == 0:                  // special case for insertion at beginning of list
        newnode.next = head
        head.prev = newnode
        head = newnode

    else if index == size:          // special case for insertion at end of list
        newnode.prev = tail
        tail.next = newnode
        tail = newnode

    else:                           // general case for insertion in middle of list
        curr = head
        repeat index-1 times:       // move curr to directly before insertion site
            curr = curr.next
        newnode.next = curr.next    // update the pointers
        newnode.prev = curr
        curr.next = newnode
        newnode.next.prev = newnode

    size = size + 1                 // increment size
```

Step 9

CODE CHALLENGE: Inserting an Element into a Linked List

We have defined the following **Singly Linked List Node** C++ class for you:

```
class Node {
    public:
        int value;
        Node* next = NULL;
};
```

Write a function `insert(Node* head, Node* newnode, int index)` that inserts `newnode` into index `index` of the **Linked List**. We guarantee that we will not have you insert at the very beginning nor the very end of the **Linked List** (so you won't need to worry about updating the *head* or *tail* pointers of the **Linked List**).

Sample Input:

```
6 2
0 -> 1 -> 2 -> 3 -> 4 -> 5
```

Sample Output:

```
0 -> 1 -> 6 -> 2 -> 3 -> 4 -> 5
```

To solve this problem please visit https://stepik.org/lesson/28867/step/9

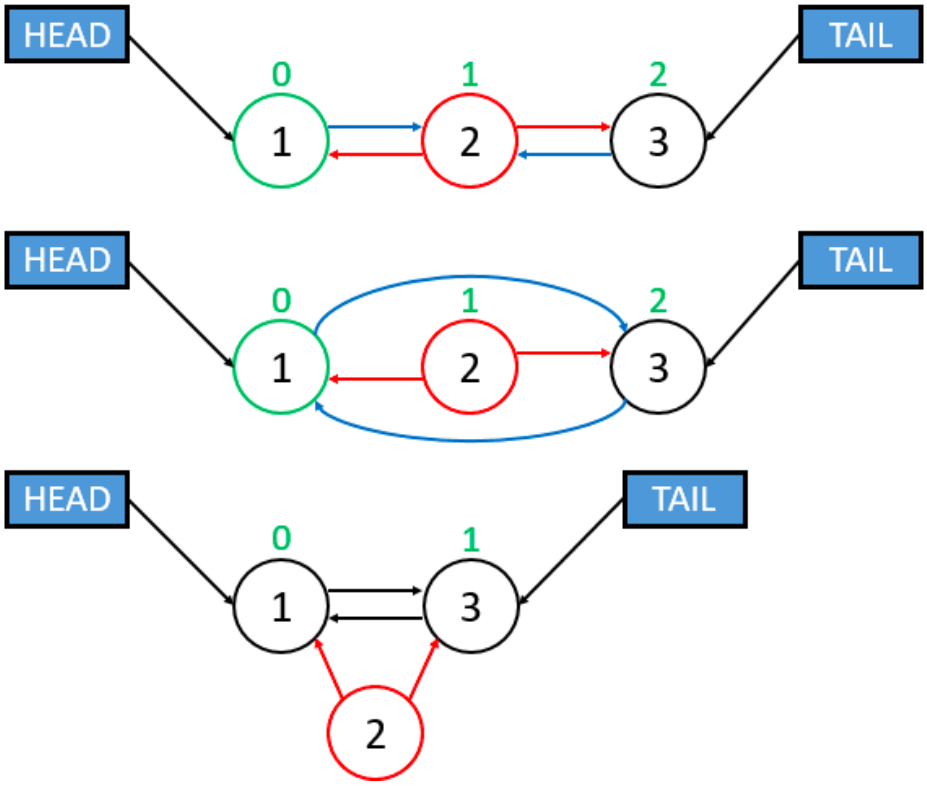
Step 10

The "remove" algorithm is also almost identical to the "find" algorithm: you first execute the "find" algorithm just like before, but once you find the insertion site, you rearrange pointers to remove the node of interest. Below is pseudocode for the "remove" operation of a **Linked List** (with added corner cases for the first and last indices):

```
remove(index): // removes the element at position "index" of Linked List
    if index == 0:                  // special case for removing from beginning of list
        head = head.next
        head.prev = NULL

    else if index == n:             // special case for removing from end of list
        tail = tail.prev
        tail.next = NULL

    else:                           // general case for removing from middle of list
        curr = head
        repeat index-1 times:       // move curr to directly before removal site
            curr = curr.next
        curr.next = curr.next.next // update the pointers
        curr.next.prev = curr

    n = n - 1                       // decrement n
```

Step 11

Below is an example of removing the element at index 1 of the **Linked List** (using 0-based counting):



STOP and Think: Notice that the node we removed still exists in this diagram. Not considering memory management (i.e., only thinking in terms of data structure functionality), is this an issue?

Step 12

CODE CHALLENGE: Removing an Element from a Linked List

We have defined the following **Singly-Linked List Node** C++ class for you:

```
class Node {
    public:
        int value;
        Node* next = NULL;
};
```

Write a function `remove(Node* head, int index)` that removes the element at index `index` of the **Linked List**. We guarantee that we will not have you remove from the very beginning nor the very end of the **Linked List** (so you won't need to worry about updating the *head* or *tail* pointers of the **Linked List**).

Sample Input:

```
2
0 -> 1 -> 2 -> 3 -> 4 -> 5
```

Sample Output:

```
0 -> 1 -> 3 -> 4 -> 5
```

To solve this problem please visit https://stepik.org/lesson/28867/step/12

Step 13

In summation, **Linked Lists** are great (constant-time) when we add or remove elements from the beginning or the end of the list, but finding elements in a **Linked List** (even one in which elements are sorted) cannot be optimized like it can in an **Array List**, so we are stuck with O(*n*) "find" operations. Also, recall that, with **Array Lists**, we needed to allocate extra space to avoid having to recreate the backing array repeatedly, but because of the dynamic allocation of memory for new nodes in a **Linked List**, we have no wasted memory here.

Array-based and Linked data structures are two basic starting points for many more complicated data structures. Because one strategy does not entirely dominate the other (i.e., they both have their pros and cons), you must analyze each situation to see which approach would be better.