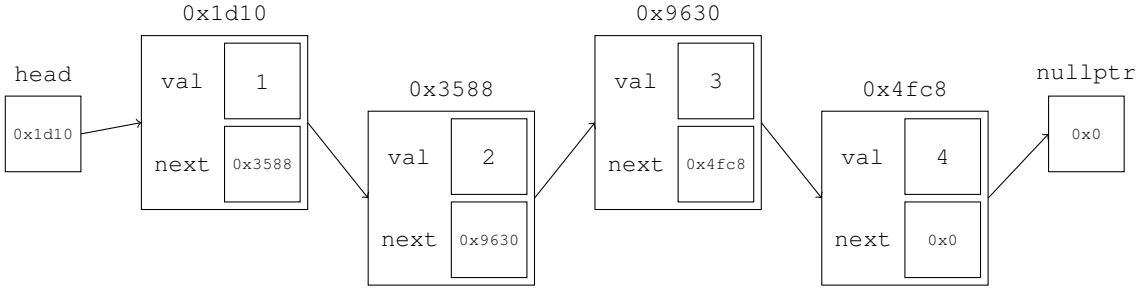
Chapter 8
Linked Lists

8.1 Singly- and Doubly-Linked Lists

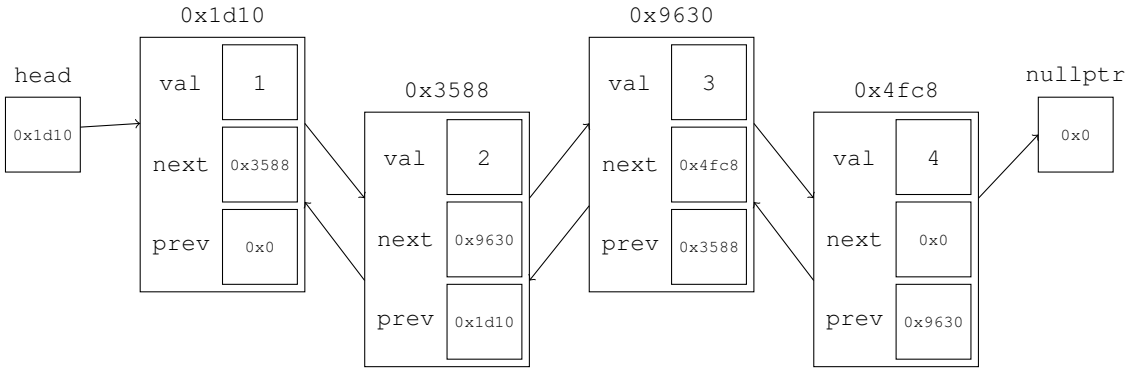
In the previous chapter, we introduced the vector data structure, which stores its data contiguously in memory. The contiguity of elements makes it possible to access any element in a vector in constant time using pointer arithmetic. Vectors also tend to provide fast access to its data due to a phenomenon known as *caching*, which makes it faster to sequentially access elements that are closer together in memory. However, the need to keep elements contiguous in memory causes inefficiencies for certain operations. For instance, inserting or erasing elements from anywhere other than the back of the vector would require elements to be shifted.

In this chapter, we will discuss the concept of a **linked list**, which does not require its elements to be stored contiguously in memory. A linked list is made up of a sequence of *nodes*. Each node not only stores a data element of the list, but also pointers that can be used to determine the relative ordering of nodes within the entire list.

In a **singly-linked list**, each node in the list stores a pointer to the next node in the sequence. The last node in the list points to `nullptr`. A depiction of a singly-linked list is shown below:



In a **doubly-linked list**, each node in the list stores a pointer to both the previous and next nodes in the sequence. The last node's `next` pointer and the first node's `prev` pointer both point to `nullptr`. Doubly-linked lists make it easier to traverse the list in both directions, but they require more memory due to the extra pointer. A depiction of a doubly-linked list is shown below:



Because the nodes in a linked list are not contiguous in memory, pointer arithmetic cannot be used to determine the memory address of any node in the list. If we wanted to access the n^{th} element in the list, for some arbitrary value n, we would have to start at the head and iterate through n elements of the list (which takes $\Theta(n)$ time).

In what ways is a linked list more efficient than a vector? As mentioned in the previous chapter, vectors store their data in a heap-allocated array, and reallocation is necessary if the size of the data exceeds the array's capacity. Because excess capacity is allocated ahead of time in a vector, memory may be wasted. Furthermore, reallocation takes time and invalidates pointers and iterators to a vector's data. Another issue with vectors is that insertions and removals require elements after the modification point to be shifted, which could take up to $\Theta(n)$ time.

A linked list does not face these issues. Memory is allocated as needed, so you do not need to worry about allocated memory going unused. In addition, iterators and pointers to an element in a list are never invalidated until the element is destroyed; this is because, unlike in a vector, an element in a linked list will never have to be reallocated in memory. Adding or removing elements at the beginning or middle of a container is also asymptotically more efficient in a list, since none of the elements afterward need to be shifted.

That being said, linked lists also have their downsides. Each node in a linked list not only needs to store its data value, but also pointers to the next element (and previous element if doubly-linked). Hence, linked lists often require much more memory than a vector to store the exact same data (assuming the vector is properly resized or reserved to the correct size). Linked lists are often slower than vectors as well; to access an arbitrary element, you will have to traverse through all the elements before it. Additionally, lists cannot take advantage of memory contiguity and caching as easily as a vector. Data access in a vector is often faster than data access in a list — so much faster that vectors tend to outperform lists in many situations, even in cases where lists asymptotically have an advantage!

8.2 Node Insertion

Much like the `Array` container class that we began implementing in chapter 6, we will start by discussing a linked list's implementation as a container class. Each node in a linked list stores a value, as well as a pointer that points to the next node (and a pointer to the previous node if it is doubly-linked). The list also maintains a `head` pointer that points to the first element in the list, as well as an optional `tail` pointer that points to the last element in the list. This is shown in the class definition below:

```
1   template <typename T>
2   class LinkedList {
3     struct Node {
4       T data;
5       Node* prev;
6       Node* next;
7       Node() : prev{nullptr}, next{nullptr} {}
8       Node(T x) : data{x}, prev{nullptr}, next{nullptr} {}
9     };
10
11    Node* head;
12    Node* tail;
13  public:
14    LinkedList() {
15      head = tail = nullptr;
16    } // LinkedList()
17
18    ~LinkedList() {
19      Node* temp;
20      while (head != nullptr) {
21        temp = head->next;
22        delete head;
23        head = temp;
24      } // while
25    } // ~LinkedList()
26    ...
27  };
```

Given a linked list, how would you insert a new node into the list? It turns out that this implementation is slightly different depending on where the node is inserted. There are four cases that you have to consider:

1. The insertion is done on an empty list.
2. The insertion happens at the front of the list.
3. The insertion happens at the back of the list.
4. The insertion happens anywhere in the middle of the list (not beginning or end).

1. The insertion is done on an empty list.

If a list is empty, its `head` (and `tail` if doubly-linked) will point to `nullptr`. To insert an element into an empty list (with initial value `val`), simply allocate a new node and set `head` and `tail` to point to that new element.

```
if (head == nullptr) {
  Node* new_node = new Node{val};
  head = new_node;
  tail = new_node;
} // if
```

Before you use the `->` operator to access the data of a node, you should always check to make sure that the node is not `nullptr`! Using `->` on a `nullptr` will cause a segmentation fault.

2. The insertion happens at the front of the list.

If the insertion happens at the front of the list, the following steps should be completed:

1. Allocate a new node.
2. Set the `next` pointer of this node to the head of the linked list.
3. If doubly-linked, set the `prev` pointer of the new node to `nullptr` and the `prev` pointer of `head` to the new node.
4. Set `head` to point to the new node.

The following code inserts a node at the very front of a doubly-linked list, initialized to a value of `val`:

```
// allocate a new node, initialized to val
Node* new_node = new Node{val};
// set the next of this node to the current head
new_node->next = head;
// if list is not empty, set prev of current head to new node
// otherwise, set tail to the new node
if (head != nullptr) {
  head->prev = new_node;
} // if
else {
  tail = new_node;
} // else
// set head to the new node
head = new_node;
```

3. The insertion happens at the back of the list.

If the insertion happens at the very back of the list, the following steps should be completed:

1. Allocate a new node.
2. If doubly-linked, set the `prev` pointer to the last node in the list (you can retrieve this last element by either using the `tail` pointer if there is one, or iterating to the end of the list if no `tail` pointer exists).
3. It there is a `tail` pointer, set it to the new node.

The following code inserts a node at the very back of a doubly-linked list, initialized to a value of `val`:

```
// allocate a new node, initialized to val
Node* new_node = new Node{val};
// set prev to tail
new_node->prev = tail;
// if list is not empty, set next of tail to new node
// otherwise, set head to the new node
if (tail != nullptr) {
  tail->next = new_node;
} // if
else {
  head = new_node;
} // else
// set tail to the new node
tail = new_node;
```

4. The insertion happens in the middle of the list.

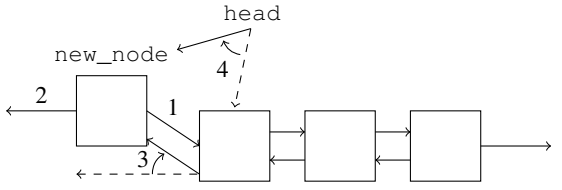
If the insertion happens in the middle of the list, you will have to allocate the new node and update the connections of the two nodes adjacent to the insertion point (or just the node before it if singly-linked). The following code takes a pointer to a node in the list (named `prev_node`) and inserts a node initialized to `val` directly after it:

```
// allocate a new node
Node* new_node = new Node{val};
// update prev and next of the new node
new_node->prev = prev_node;
new_node->next = prev_node->next;
// update next of prev_node
prev_node->next = new_node;
// update prev of the node directly after insertion point
if (new_node->next != nullptr) {
  new_node->next->prev = new_node;
} // if
else {
  tail = new_node; // this runs if prev_node is the last node
} // else
```

The insertion process for a doubly-linked list is summarized below. The process for a singly-linked list is similar, just without the `prev` pointers.

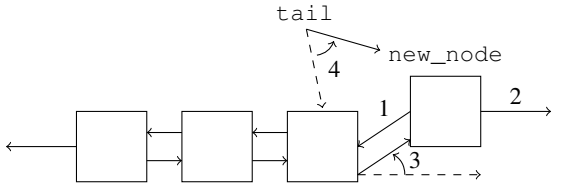
Insert at beginning:

1. Set `next` of `new_node` to `head`.
2. Set `prev` of `new_node` to `nullptr`.
3. Set `prev` of `head` to `new_node`.
4. Update `head` to point to `new_node`.



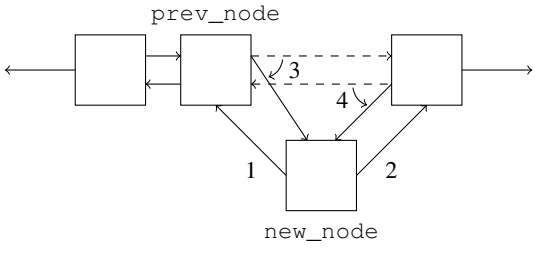
Insert at end:

1. Set `prev` of `new_node` to `tail`.
2. Set `next` of `new_node` to `nullptr`.
3. Set `next` of `tail` to `new_node`.
4. Update `tail` to point to `new_node`.



Insert in middle, given `prev_node`:

1. Set `prev` of `new_node` to `prev_node`.
2. Set `next` of `new_node` to `prev_node->next`.
3. Set `next` of `prev_node` to `new_node`.
4. Set `prev` of `new_node->next` to `new_node`.



Example 8.1 Consider the following definition of a `Node`:

```
struct Node {
  int32_t data;
  Node* prev;
  Node* next;
  Node() : prev{nullptr}, next{nullptr} {}
  Node(int x) : data{x}, prev{nullptr}, next{nullptr} {}
};
```

Write a function that inserts a value `val` into a *sorted* doubly-linked list. In other words, you have to insert a given value in a position to keep the list sorted. The list class keeps track of a `head` and `tail` pointer, which point to the first and last elements in the list respectively. If the list is empty, both `head` and `tail` point to `nullptr`. The `Node` constructor sets `prev` and `next` to `nullptr`.

Since this question involves inserting elements into a list, we have to take into account the four conditions covered previously. In addition, we need to take duplicates into account. For the sake of consistency, we will insert any duplicate value directly after the duplicates that are already in the list. By doing this, we will only need to worry about the equality case when adding to the middle or end, and not the beginning.

1. The list is empty.

If the list is empty, we must create a new node and set both `head` and `tail` to the new node. This happens when both the `head` and `tail` pointers are `nullptr`. The code for this first condition is shown below:

```
1  // Case #1: the list is empty
2  if (head == nullptr) {
3    // set both head and tail to the new node
4    Node* new_node = new Node{val};
5    head = new_node;
6    tail = new_node;
7  } // if
```

2. `val` is smaller than all the other elements in the list.

If `val` is the smallest element in the entire list, it would have to be inserted at the very beginning. Since the list is sorted, we can check if `val` is smaller than all the other elements by just comparing the first element in the list with `val`. If the first element in the list is larger than `val`, we can follow the procedure for inserting the node at the beginning:

```
1  // Case #2: insert at beginning of list
2  if (val < head->data) {
3    Node* new_node = new Node{val};
4    new_node->next = head;
5    head->prev = new_node;
6    head = new_node;
7  } // if
```

3. `val` is larger than all the other elements in the list.

If `val` is the largest element in the entire list, it would have to be inserted at the very end. Since the list is sorted, we can check if `val` is larger than all the other elements by just comparing the last element in the list with `val`. If the last element in the list is smaller than `val`, we can follow the procedure for inserting the node at the end:

```
1  // Case #3: insert at end of list
2  if (val >= tail->data) {
3    Node* new_node = new Node{val};
4    new_node->prev = tail;
5    tail->next = new_node;
6    tail = new_node;
7  } // if
```

Note that this process was simple because we had access to the last element through a `tail` pointer. If the list did not have a `tail` pointer, we would need to iterate through the entire list to reach the point of insertion if `val` were larger than all other values in the list.

4. `val` is neither the smallest nor largest value in the list.

If `val` is neither the smallest nor largest value in the list, it should be inserted somewhere in the middle. As a result, we will need to find the position `val` should be inserted at; because the list is sorted, `val` should be inserted after the largest value that less than or equal to it.

To find the correct position of insertion, a good technique is to use a `while` loop to iterate through the elements of the list until the largest value that is less than or equal to `val` is found. In our case, we want to find the node directly before the position of insertion, which stores the largest value that is less than or equal to `val` (this allows us to follow the template of inserting in the middle of the list, which we covered earlier). This can be done by iterating through the list until we reach the first node whose `next` is larger than `val`, as shown:

```
1  Node* prev_node = head;
2  while (prev_node->next && prev_node->next->data < val) {
3    prev_node = prev_node->next;
4  } // while
```

At the end of this loop, `prev_node` should be the node directly before the position of the new node. We can now use the insertion procedure to add `val` to the correct position. There is no need to check if `prev_node` is `nullptr` here, since the `while` loop guarantees that `prev_node` is valid since `prev_node` is only assigned to `prev_node->next` if `prev_node->next` is not `nullptr`.

```
1  // Case #4: insert in middle of list
2  Node *new_node = new Node{val};
3  new_node->prev = prev_node;
4  new_node->next = prev_node->next;
5  prev_node->next = new_node;
6  new_node->next->prev = new_node;
```

Putting this all together, we get the following solution (written as a member function of the list class). Note that it is also possible to combine the third and fourth cases (which would have to be done if there is no `tail` pointer), but this will require you to iterate through the entire list if you want to insert an element at the end.

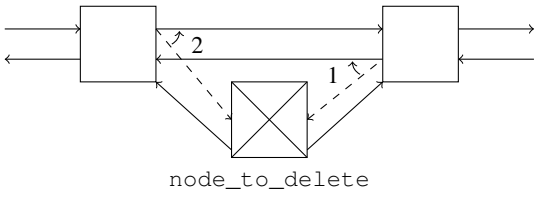
```
1  void insert(int32_t val) {
2    Node* new_node = new Node{val};
3    if (head == nullptr) {
4      head = new_node;
5      tail = new_node;
6      return;
7    } // if
8    if (val < head->data) {
9      new_node->next = head;
10     head->prev = new_node;
11     head = new_node;
12     return;
13   } // if
14   if (val >= tail->data) {
15     new_node->prev = tail;
16     tail->next = new_node;
17     tail = new_node;
18     return;
19   } // if
20   Node* prev_node = head;
21   while (prev_node->next && prev_node->next->data < val) {
22     prev_node = prev_node->next;
23   } // while
24   new_node->prev = prev_node;
25   new_node->next = prev_node->next;
26   prev_node->next = new_node;
27   new_node->next->prev = new_node;
28 } // insert()
```

8.3 Node Deletion

What is the time complexity of deleting a node in a linked list, given its *index*? Since linked lists do not offer random access, this process is $\Theta(n)$ on average for both singly- and doubly-linked lists: to delete an element at a specified index, you would have to iterate through the list from the beginning until you find the node that you want to delete.

What if you were given a *pointer* to the node you wanted to delete, instead of its index? Does the time complexity of deleting that element change? For a *doubly*-linked list, the time complexity would become $\Theta(1)$, since we can just update the nodes that are adjacent to the deleted node (the following code assumes deletion occurs in the middle; additional checks should be made if the node is at the beginning or end):

```
1  node_to_delete->next->prev = node_to_delete->prev;
2  node_to_delete->prev->next = node_to_delete->next;
3  delete node_to_delete;
```



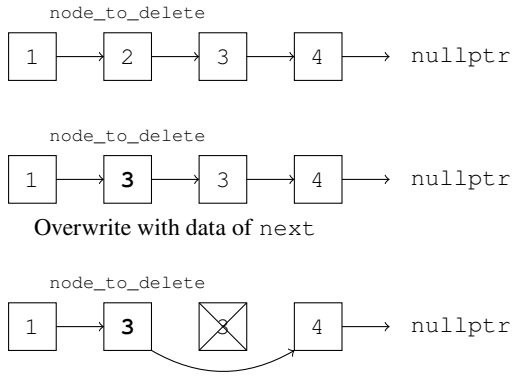
However, the time complexity of deleting an element given a pointer to a *singly*-linked list is still $\Theta(n)$. This is because we need to update the previous node's `next` so that it points to the deleted node's `next`. We do not have direct access to the previous node in a singly-linked list, so we would have to traverse through the list to find it.

Is it somehow possible for a singly-linked list to support $\Theta(1)$ deletion in certain situations? It is possible if the pointer passed in is a pointer to the node directly *before* the node to be deleted (i.e., `deleteNode(Node* prev)`). However, this approach is rather counter-intuitive and, depending on the list implementation, may not support deleting the head node of a list.

What if the pointer passed into the deletion function *must* point to the node to be deleted? If we are given nothing else, is $\Theta(1)$ deletion still possible for a singly-linked list? The following shows one potential approach:

1. Overwrite the data in the node to be deleted with the data of the next node's data.
2. Delete the next node.

```
1  Node* next_node = node_to_delete->next;
2  node_to_delete->data = next_node->data;
3  node_to_delete->next = next_node->next;
4  delete next_node;
```



After this deletion, the contents of the list are exactly what we wanted. Since we only swapped a few pointers around and changed a value, this was all done in constant time. However, this approach does not always work! There are three reasons why:

1. If the node we wanted to delete were the last node, trying to overwrite and delete `next` would fail.
2. This implementation assumes that the data can be copied, which cannot be guaranteed.
3. This method is not safe, since we end up deleting `node_to_delete->next` rather than `next`. Even though we swapped the data so that the list still *looks* valid, the memory addresses are no longer the same. If we had any pointers, iterators, or any data that depended on the address of `node_to_delete->next`, they would be invalidated.

As a result, the above approach is not perfect, even if it can delete a given node of a singly-linked list in constant time. In general, if you do not have a reference to the node directly *before* the one you want to delete, the worst-case time complexity of deleting an element from a singly-linked list cannot be better than $\Theta(n)$.

The deletion process is very similar to the insertion process. Make sure you consider all possible edge cases and update the surrounding nodes after the deletion to ensure that the entire list remains valid.

Example 8.2 Consider the following definition of a node:

```
struct Node {
  int32_t data;
  Node* next;
  Node() : next{nullptr} {}
  Node(int x) : data{x}, next{nullptr} {}
};
```

Write a function that deletes a node from a *singly*-linked list, given the index of this node. 0-indexing is used. The node constructor sets `next` to `nullptr` by default. The list class only supports a `head` pointer, which points to the first element in the list.

To solve this problem, we want to first find the node that we want to delete. This requires us to iterate through the list. However, since this is a deletion problem on a singly-linked list, we also need to find the node directly before the one we want to delete to implement a $\Theta(1)$ solution.

We will also need to consider several edge cases:

1. If the linked list is empty, we shouldn't delete anything at all.
2. If the index we want to delete is 0, we should update the `head` pointer before deleting.
3. If the index we get is larger than the largest index possible, we should not delete anything at all.
4. If there were a `tail` pointer, deleting the last element in the list would require an update to the `tail` pointer (this does not apply here).

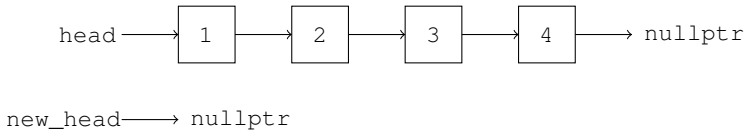
The solution to this problem is shown below. It checks the cases and iterates up to the node directly before the one that needs to be deleted. It then deletes the correct node and resets the pointers so that the entire list remains valid after the deletion.

```
void delete_at_index(size_t index) {
  if (head == nullptr) {   // check for empty list
    return;
  } // if
  if (index == 0) {        // check if index is 0
    Node* temp = head;
    head = temp->next;
    delete temp;
    return;
  } // if
  // find the node before one to delete by looping through list - make sure not to iterate off end
  Node* prev_node = head;
  for (size_t i = 0; prev_node != nullptr && i < index - 1; ++i) {
    prev_node = prev_node->next;
  } // for i
  // make sure the index is not larger than the largest index possible
  if (prev_node == nullptr || prev_node->next == nullptr) {
    return;
  } // if
  // delete the node and update the pointers
  Node* node_to_delete = prev_node->next;
  prev_node->next = node_to_delete->next;
  delete node_to_delete;
} // delete_at_index()
```

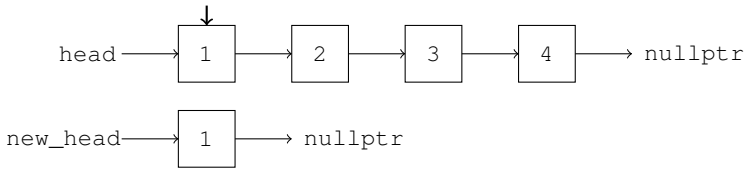
8.4 Reversing a Linked List

※ 8.4.1 *A Naïve Solution For Reversing a Linked List*

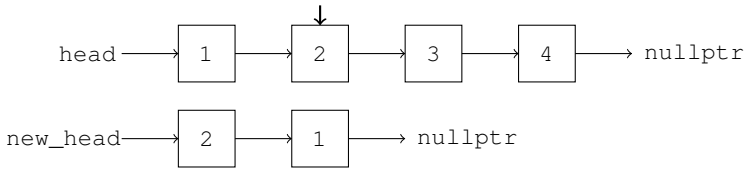
In this section, we will discuss a common interview problem: reversing a linked list. To start off, we will introduce a naïve approach for solving this problem: simply iterate through the linked list and copy the elements to the front of a new list. An illustration of this process is shown below:



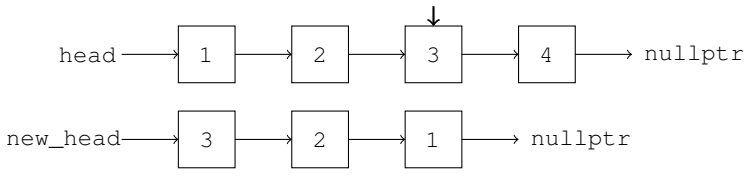
Begin iterating through the original list, and copy `1` to the beginning of the new list:



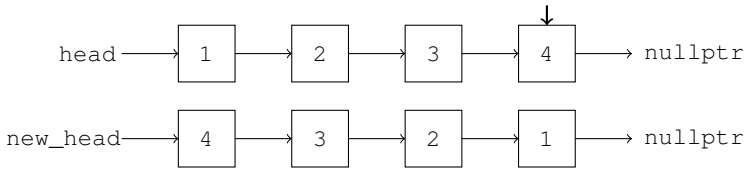
Iterate to `2` and copy it to the beginning of the new list:



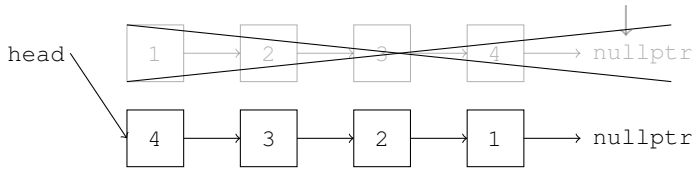
Iterate to `3` and copy it to the beginning of the new list:



Iterate to `4` and copy it to the beginning of the new list:



Once you iterate to a `nullptr`, deallocate the old list and update the `head` pointer to point to the new list:



The code for this solution is shown below (here, the deallocation occurs during the traversal, but the idea is the same):

```
1   void reverse_list(Node*& head) {
2     Node* new_head = nullptr;
3     while (head != nullptr) {
4       Node* new_node = new Node{head->data};
5       new_node->next = new_head;
6       new_head = new_node;
7       Node* old_node = head;
8       head = head->next;
9       delete old_node;
10    } // while
11    head = new_head;
12  } // reverse_list()
```

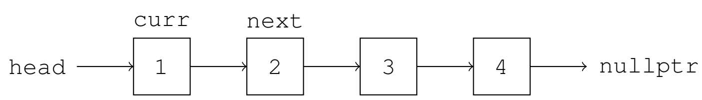
This solution takes $\Theta(n)$ time and $\Theta(n)$ auxiliary space, since it iterates through all n elements of the list and makes an additional copy of the list to write the reversed elements to. Although this solution works, it is *not* the most efficient solution. Not only are we making a separate copy of the list (which takes up additional memory), we are also wasting time constructing new nodes (and if the data we store in each node were large, trying to duplicate each element could be costly).

※ 8.4.2 *An Optimized Iterative Approach*

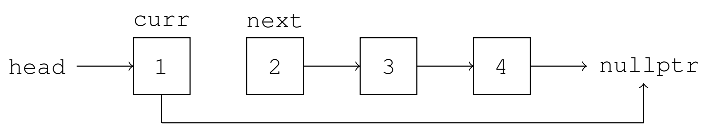
A better solution does not involve making a duplicate copy of the list at all. Instead, we only need to make one pass through the list and modify pointers along the way. The algorithm is as follows:

1. Initialize three pointers: `curr`, `prev`, and `next`. `curr` should be set to `head` upon initialization.
2. While `curr` is not `nullptr`, repeat the following:
   a. Set `next` to `curr->next`.
   b. Reverse `curr`'s `next` pointer by setting `curr->next` to `prev`.
   c. Move all pointers one position forward by setting `prev` to `curr` and `curr` to `next`.
3. After the loop ends (`curr` hits `nullptr`), set `head` to `prev`.

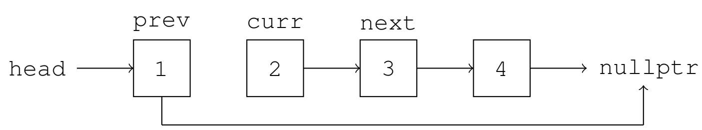
An illustration of this algorithm is shown below:



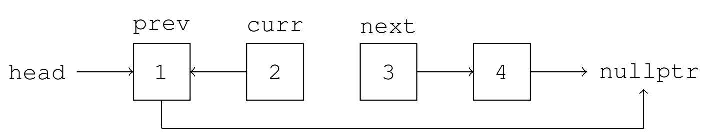
Set `curr->next` to `prev` (in this case, `prev` is `nullptr`):



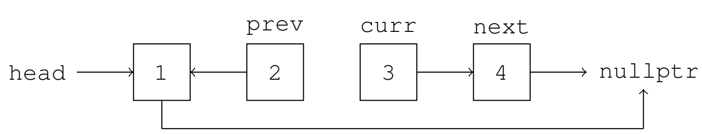
Move each pointer forward by one:



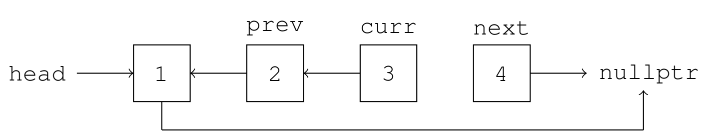
Set `curr->next` to `prev`:



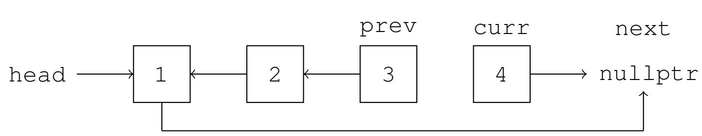
Move each pointer forward by one:



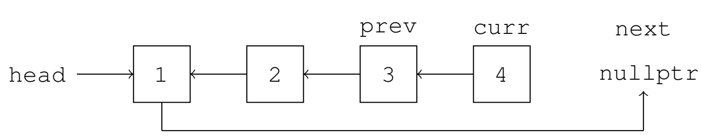
Set `curr->next` to `prev`:



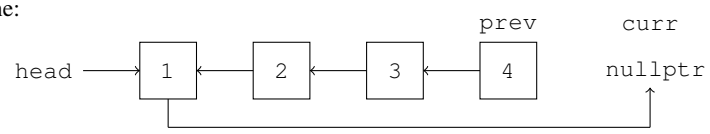
Move each pointer forward by one:



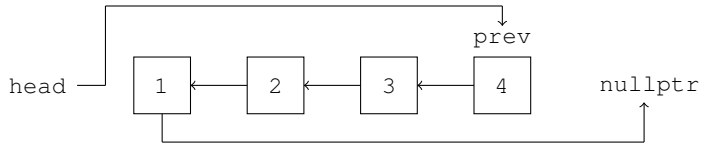
Set `curr->next` to `prev`:



Move each pointer forward by one:



Once `curr == nullptr`, set `head` to `prev`:



The list has been successfully reversed. The benefit of this approach over the first method is that an additional copy of the list is not needed; the reversing is done in-place! In addition, we do not have to deallocate an entire list after we are done, since we are modifying the original list itself rather than making an copy. The following code implements the above approach and reverses a singly-linked list using $\Theta(1)$ auxiliary space:

```
1   void reverse_list(Node*& head) {
2     // initialize curr, prev, and next
3     Node* curr = head;
4     Node* prev = nullptr;
5     Node* next = nullptr;
6     // loop until curr is nullptr
7     while (curr != nullptr) {
8       // update next to curr->next
9       next = curr->next;
10      // reverse curr's next pointer
11      curr->next = prev;
12      // move all pointers forward by one, next gets updated during next iteration
13      // of while loop (this is done because we want to guarantee that the updated
14      // curr is not nullptr before we try to assign curr->next to next)
15      prev = curr;
16      curr = next;
17    } // while
18    // after the loop terminates, set head to prev
19    head = prev;
20  } // reverse_list()
```

The time complexity of this function is $\Theta(n)$ because the entire list is traversed. The auxiliary space used by this function is $\Theta(1)$ since the additional space needed to complete the reversal does not depend on the size of the list n.

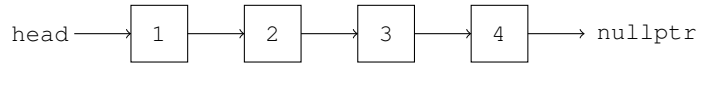
The above approach can be used to reverse a *doubly*-linked list using $\Theta(1)$ auxiliary space as well. The only difference is that two directional pointers have to be adjusted at each step rather than just one.

※ 8.4.3 *An Optimized Recursive Solution*

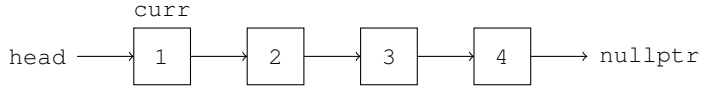
A linked list can also be reversed recursively. The recursive algorithm is as follows:

1. Initialize a pointer `curr` that points to the head of the list.
2. Check for the base cases:
   a. If `curr` is `nullptr`, return.
   b. If `curr->next` is `nullptr`, it must be the last node in the list, so make this node the new head and return.
3. Make a recursive call on `curr->next`.
4. Set `curr->next->next` to `curr`.
5. Set `curr->next` to `nullptr`.

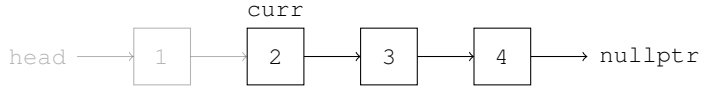
Let's look at how this works visually. Suppose we are given the following linked list, which we want to reverse recursively:



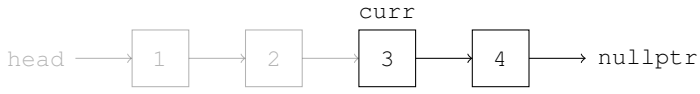
First, we initialize `curr` to point to the head of the list:



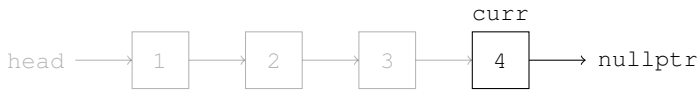
We then check for the base cases. Since neither `curr` nor `curr->next` are `nullptr`, the base case does not run. As a result, we make a recursive call on `curr->next`.



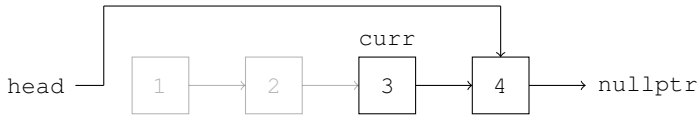
We check the base cases again. Neither `curr` nor `curr->next` are `nullptr`, so we make a recursive call on `curr->next`.



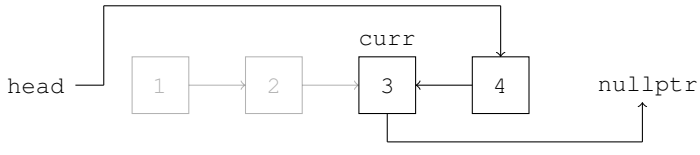
Again, neither `curr` nor `curr->next` are `nullptr`, so we make another recursive call on `curr->next`.



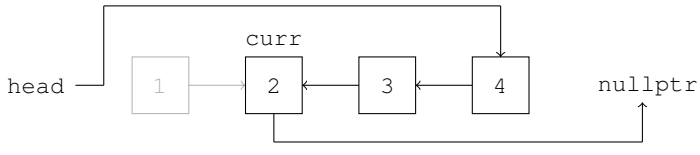
At this point, `curr->next` is `nullptr`, so the base case runs. We know that `4` must be the last element in the list, so we make this node the new head and return. The recursion unrolls, and we end up back at node `3`.



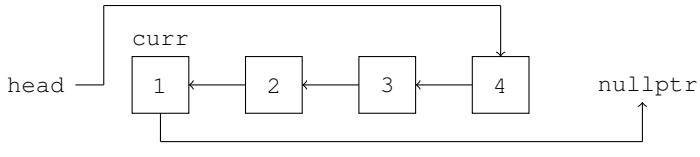
The recursive call that was made at node `3` is now complete, so we can move on to steps 4 and 5 of the algorithm. We set `curr->next->next` to `curr`, and `curr->next` to `nullptr`.



The recursion unrolls, and we end up back at node `2`. Since the recursive call that was made at node `2` is complete, we will set `curr->next->next` to `curr`, and `curr->next` to `nullptr`.



The recursion unrolls, and we end up back at node `1`. Since the recursive call that was made at node `1` is complete, we will set `curr->next->next` to `curr`, and `curr->next` to `nullptr`. The linked list has now been successfully reversed.



The code for recursively reversing a linked list is shown below:

```
1   void helper(Node*& head, Node*& curr) {
2     if (curr == nullptr) {
3       return;
4     } // if
5     if (curr->next == nullptr) {
6       head = curr;
7       return;
8     } // if
9     helper(head, curr->next);
10    curr->next->next = curr;
11    curr->next = nullptr;
12  } // helper()
13
14  void reverse_list(Node *&head) {
15    Node* curr = head;
16    helper(head, curr);
17  } // reverse_list()
```

Remark: In all of these functions, a *pointer by reference* was passed in (e.g., `*&head`). Passing in the pointer by reference allows us to modify where the pointer is pointing to inside the function (in these examples, we were able to change where `head` was pointing to). Without the pointer by reference, the function would take in the pointer by *value* (e.g., the `head` variable in the function would be a local *copy* of the actual `head` pointer, and reassigning `head` in the function would reassign the function's local copy, and not the original `head` of the linked list). Consider the following two functions:

```
1  void foo(Node* head) {
2    head = nullptr;
3  } // foo()
4
5  void bar(Node*& head) {
6    head = nullptr;
7  } // bar()
8
9  int main() {
10   Node* head = new Node{281};
11   foo(head);
12   std::cout << head << std::endl;  // prints address 0xd20c20 (arbitrary)
13   bar(head);
14   std::cout << head << std::endl;  // prints address 0x0
15 } // main()
```

Notice that line 12 did not print out address `0x0` (`nullptr`). This is because the `foo()` function took in a pointer by *value*, so the function made a local copy of `head` and set that local copy to `nullptr`. Thus, the original `head` variable created on line 10 did not change after it was passed into `foo()`. On the other hand, the `bar()` function took in a pointer by *reference*, so it set the original `head` value to `nullptr`. This is why `0x0` was printed after `head` was passed into `bar()`. In summary, if you pass a pointer into a function `f()` by reference, any changes to the pointer in `f()` will also be reflected in the function that invoked `f()`.

8.5 Techniques for Solving Linked List Problems

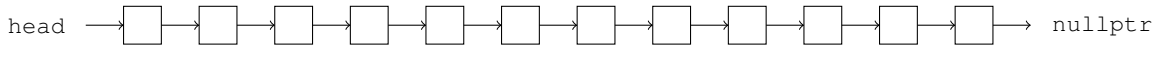
One common technique that can be used to solve linked list problems involves using two pointers to iterate through a linked list, with one either at a fixed distance from the other, or one that moves faster than the other. Let's look at a few examples that can be solved using this approach.

Example 8.3 You are given an integer k and the head pointer of a singly-linked list. Write a function to find the k^{th} to last element in the list. You may assume that the list is not empty and that k is a valid number that makes sense for the problem. Return the value of this element (i.e., the value stored in `data`). The definition of a node is shown below:

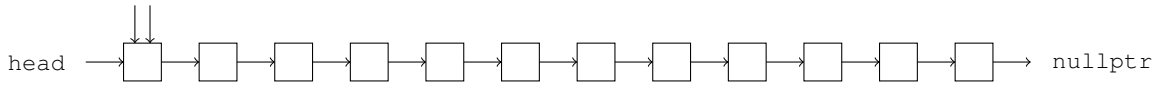
```
1  struct Node {
2    int32_t data;
3    Node* next;
4    Node(int32_t x) : data{x}, next{nullptr} {}
5  };
```

The trick to notice here is that the k^{th} to last element is k from the end of the list. Thus, we can use two pointers that are a distance of k nodes apart to find the k^{th} to last element. In our algorithm, we can start from the beginning of the list and increment both pointers until the front pointer reaches the end of the list. Since the back pointer is k nodes behind the front pointer, once the front pointer reaches the end, the back pointer must point to the k^{th} to last element — the element we want!

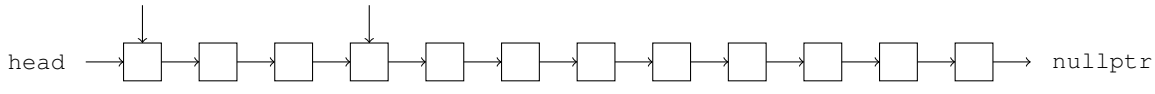
The following illustrates the process of finding the 3rd to last element in a list:



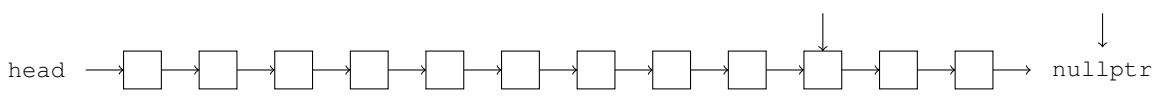
Initialize two pointers that point to the head of the list:



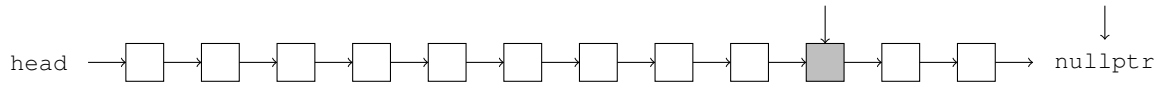
Move one pointer ahead by k positions (in this case 3 positions, since $k = 3$):



Now, iterate both pointers forward at the same time, at the same speed. This ensures that the pointers are always k nodes apart. Stop iterating as soon as the first pointer reaches the `nullptr` at the end of the list:



Return the value pointed to by the second pointer, which must be pointing to the k^{th} to last node:



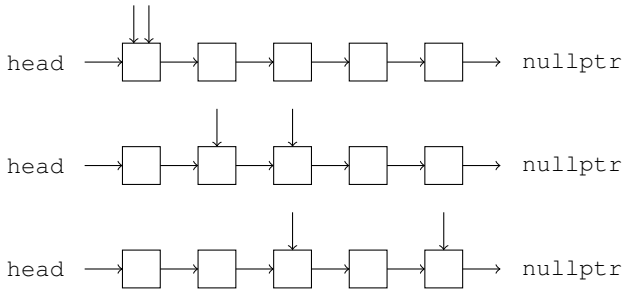
The above algorithm can be implemented as follows:

```
int32_t kth_to_last(Node* head, int32_t k) {
  // initializes two pointers
  Node* fast = head;
  Node* slow = head;
  // move the fast pointer forward by k positions
  // per instructions, k is always valid (else you need to check for nullptr)
  for (int32_t i = 0; i < k; ++i) {
    fast = fast->next;
  } // for i
  // move both fast and slow forward until fast reaches the end
  while (fast != nullptr) {
    fast = fast->next;
    slow = slow->next;
  } // while
  // once fast reaches the end, slow must point to the kth to last element
  return slow->data;
} // kth_to_last()
```

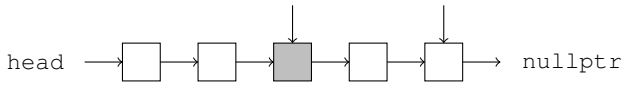
Example 8.4 You are given the head of a singly-linked list. Write a function that returns the value of the middle node. If there are two middle nodes, return the value of the second middle node. A node is defined as follows:

```
struct Node {
  int32_t data;
  Node* next;
  Node(int32_t x) : data{x}, next{nullptr} {}
};
```

Similar to the previous problem, we can iterate both pointers through the list in such a way that one pointer points to the node we want once the other reaches the end. This can be done by having one pointer move twice as fast as the other. We first initialize two pointers, `fast` and `slow`, that both start iterating at the beginning. However, with each iteration, we would increment `fast` by two and `slow` by one. When `fast` reaches the end, `slow` must point to the middle node.



Incrementing the `fast` pointer here would cause it to point to `nullptr`, so return the value pointed to by `slow`:



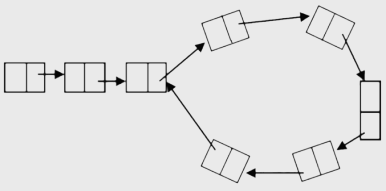
This solution is implemented in the code below:

```
int32_t find_middle_node(Node* head) {
  // initialize two pointers
  Node* fast = head;
  Node* slow = head;
  // move fast forward 2 steps for each step slow moves forward
  // continue iterating until fast reaches the end; we only need
  // to check validity of fast since slow will always be valid
  // if fast is valid (since fast hits the end of the list first)
  while (fast != nullptr && fast->next != nullptr) {
    slow = slow->next;
    fast = fast->next->next;
  } // while
  // once fast reaches the end, slow must hold the value we want
  return slow->data;
} // find_middle_node()
```

Example 8.5 You are given the head of a singly-linked list. Write a function to determine if the list contains a cycle (or loop), where there exists a node that points to a previous node in the list. A node is defined as follows:

```
struct Node {
  int32_t data;
  Node* next;
  Node(int32_t x) : data{x}, next{nullptr} {}
};
```

The following is an example of a linked list that has a cycle:



This is a slightly more challenging question, but it can be solved using the same approach as the previous two problems. How can we move two pointers through the list in a way that can help us determine whether there exists a cycle?

Let's start off with a similar problem that can lead us to the solution: suppose we are given a linked list, and we want to know whether there exists *a cycle of length 6* (such as in the illustration shown on the previous page). How can we use two pointers to determine if a cycle of length 6 exists? Well, if we had two pointers that were a distance of 6 nodes apart, then eventually the two pointers would always point to the same node if we increment them at the exact same rate. This is because, if a cycle of length 6 exists, the node that is 6 nodes away from one pointer must be itself, since everything loops around.

Now, what if we were instead asked if the list had a cycle of length 7? We could solve this problem in a similar manner: instead of iterating two pointers 6 nodes apart, we would iterate two pointers 7 nodes apart. If a cycle of length 7 exists, then two pointers that are 7 nodes apart in the cycle must point to the exact same node.

Even though keeping two pointers a constant distance apart does not solve the problem (since we do not know the length of the cycle we want to find), we can use this idea to construct a solution that can find the existence of any cycle in the list, regardless of its length. The trick to notice here is that, even though a constant distance does not work, we can cover all possible cycle lengths if we iterate one pointer at twice the speed of the other! Why is this the case? Suppose that there exists a loop in the list. With each iteration, the distance between the fast and slow pointer increases by 1. Thus, regardless of what the length of the cycle is, the distance between the pointers will eventually reach the length of that cycle. Because of this, if the fast and slow pointer ever meet and point to the same node, then there must exist a cycle.

What if the distance between the two pointers already exceeds the length of the cycle when the slow pointer enters the cycle? For instance, suppose a cycle of length 20 exists in our list, but that cycle is more than 20 nodes away from the head. As a result, when the two pointers are 20 nodes apart, the slow pointer has not made it into the cycle yet. Would our algorithm fail in this case, since the two pointers would not be pointing to the same node, even though a cycle of length 20 exists and the two pointers are 20 nodes apart?

It turns out that this case is still handled by our algorithm. Even if the distance between the two pointers already exceeds the size of the loop when the slow pointer enters the cycle, the nature of the loop ensures that multiples of the cycle size would also allow us to detect the cycle. For instance, if there existed a cycle of length 20, and the fast and slow pointers were already more than 20 nodes apart when the slow pointer entered the cycle, both pointers would eventually point to the same node again when they are 40, 60, 80, … nodes apart. Think about it: in a cycle of size 20, the node 40 steps way is also the exact same node! No matter how far the cycle is from the head of the list, eventually both the fast and slow pointers will be in the cycle, and the first multiple of the cycle length will allow us to detect the cycle's existence.

This algorithm is officially known as *Floyd's Cycle-Finding Algorithm*, and it is implemented below:

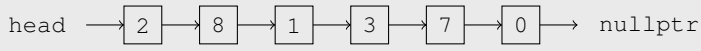
```
bool has_cycle(Node* head) {
  // use two pointers, where one moves faster than the other;
  // if the fast pointer catches up to the slow pointer.
  // then there exists a cycle
  Node* fast = head;
  Node* slow = head;
  while (fast && fast->next) {
    slow = slow->next;
    fast = fast->next->next;
    if (fast == slow) {
      return true;
    } // if
  } // while
  return false;
} // has_cycle()
```

In the following examples, we will cover some additional interview-style list problems that you may encounter.

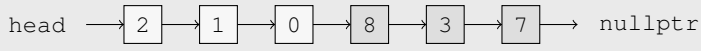
Example 8.6 You are given the head of a singly-linked list and an integer `p`. Write a function that partitions the list in a way such that all nodes less than `p` come before nodes that are greater than or equal to `p`. The relative order of nodes should be preserved in each of the two partitions. This partitioned list should be returned by your function. A node is defined as follows:

```
1  struct Node {
2    int32_t data;
3    Node* next;
4    Node(int32_t x) : data{x}, next{nullptr} {}
5  };
```

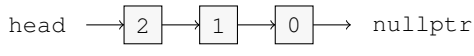
For example, given the following list and a partition value of `p = 3`:

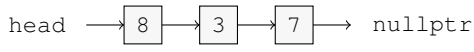


you should return a list with the values rearranged so that all values less than `3` (in this case, `2`, `1`, and `0`) come before all values greater than or equal to `3` (in this case, `8`, `3`, and `7`). The relative order of these elements is maintained (so `2` still comes before `1`, and `1` still comes before `0`; same for the values greater than or equal to `3`).

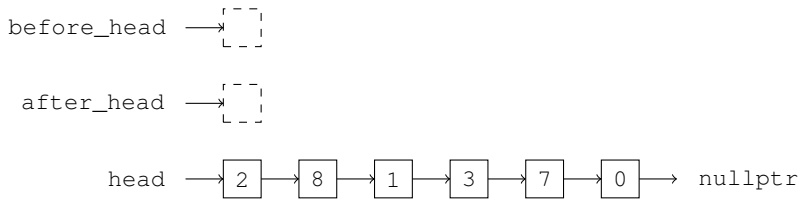


To approach this problem, a key insight is to notice that the list you want to return is actually a combination of two lists that are joined together: one with values less than `p`, and one with values greater than or equal to `p`. In our example, these two lists are as follows:

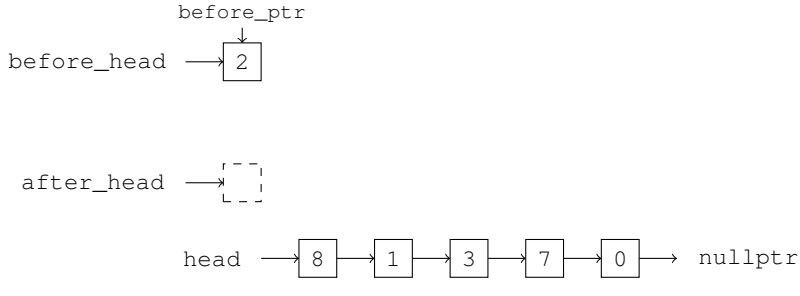




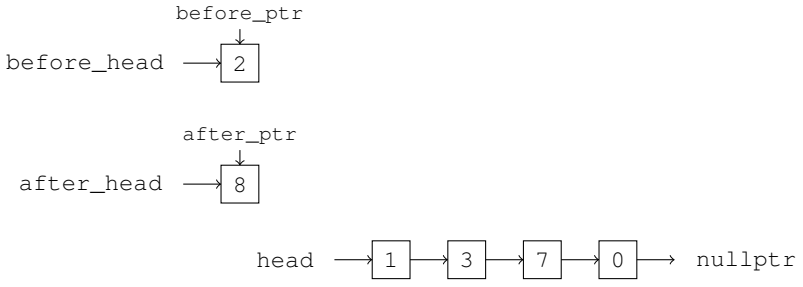
If we can recreate these two smaller lists, then we simply have to join them to get our solution. This can be done by iterating over the original list and moving each node to one of two sublists, one that stores values less than `p`, and one that stores values greater than or equal to `p`. This is shown below, where `before_head` is a pointer to the sublist with values $<$ `p`, and `after_head` is a pointer to the sublist with values $\geq$ `p`. We will also keep track of two pointers, `before_ptr` and `after_ptr`, to indicate the position of insertion for each of the two sublists.



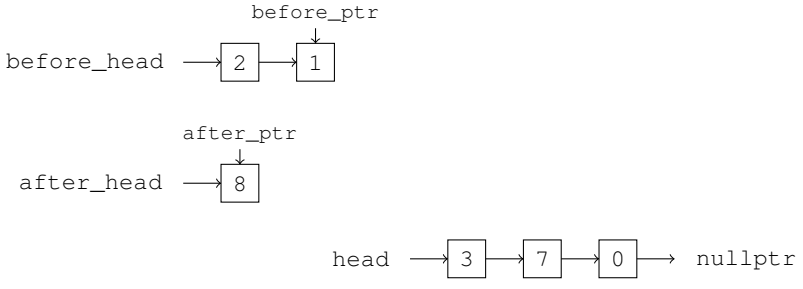
We iterate over the original list using its `head` pointer. If the value pointed to by `head` is less than `x`, we move it to the `before_head` list; otherwise, we move it to the `after_head` list. In the example, the first value of `2` is less than the value of `p = 3`, so it is moved to the `before_head` list, as shown:



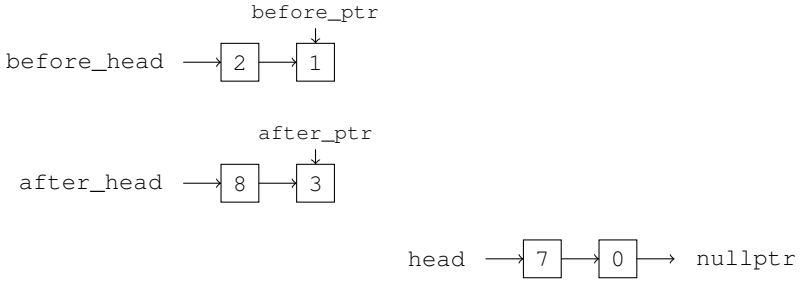
The next value, `8`, is greater than `p`, so `8` is moved to the `after_head` list:



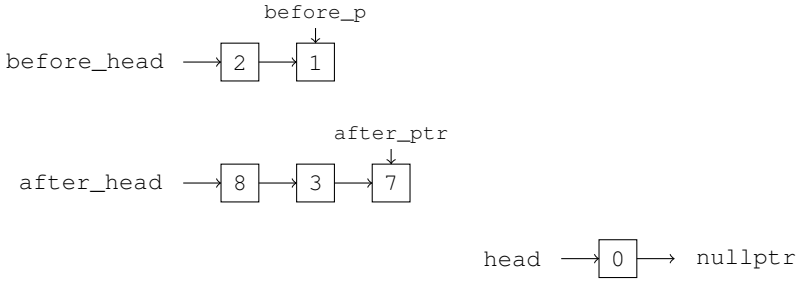
The next value, `1`, is less than `p`, so `1` is moved to the `before_head` list. We also increment the `before_ptr` pointer to indicate where the next value in the list should be added (which facilitates insertion of additional elements).



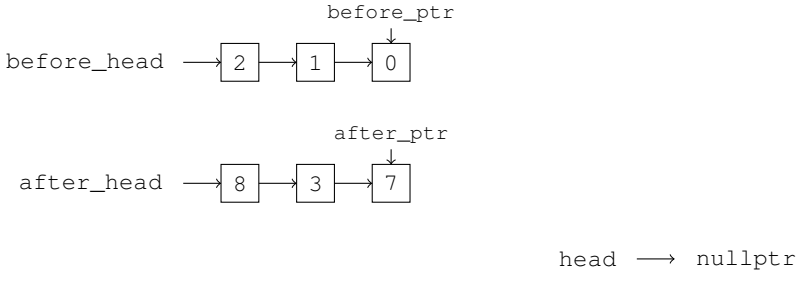
The next value, `3`, is equal to `p`, so it is moved to the `after_head` list (per the rules for elements equal to the given `p`). The `after_ptr` pointer is also incremented to facilitate insertion of additional elements.



The next value, `7`, is greater than `p`, so it is moved to the `after_head` list, and `after_ptr` is incremented.



The next value, `0`, is less than `p`, so it is moved to the `before_head` list, and `before_ptr` is incremented.



The value of `head` is now `nullptr`, so we have successfully completed our traversal of the original list. The two smaller lists are then combined (i.e., `before_head->next = after_head`) to obtain our solution list. An implementation of this solution is provided below:

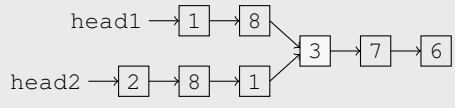
```
Node* partition_list(Node* head, int32_t p) {
  // this creates a dummy node at the beginning of the
  // before_head and after_head sublists to make implementation easier
  // (also why before_head.next and after_head.next are used for return value)
  Node before_head{0}, *before_ptr = &before_head;
  Node after_head{0}, *after_ptr = &after_head;
  while (head != nullptr) {
    if (head->val < p) {
      before_ptr->next = head;
      before_ptr = before_ptr->next;
    } // if
    else {
      after_ptr->next = head;
      after_ptr = after_ptr->next;
    } // else
    head = head->next;
  } // while

  // combine the two lists at before_head and after_head
  after_ptr->next = nullptr;
  before_ptr->next = after_head.next;
  return before_head.next;
} // partition_list()
```

The time complexity of this solution is $\Theta(n)$, where n is the number of nodes in the original list. This is because our implementation iterates over all the nodes of this list. The auxiliary space is $\Theta(1)$ because we only reorganized the nodes in the original list instead of duplicating them; thus, the additional memory allocated by this solution is a constant that does not depend on the size of the original list.

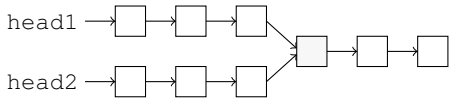
Example 8.7 You are given the head of two singly-linked lists, `head1` and `head2`. Write a function that returns the node at which these two lists intersect. If the two lists do not intersect at all, return `nullptr`. You may assume that there are no cycles in the list structure you are given. A node is defined as follows:

```
struct Node {
  int32_t data;
  Node* next;
  Node(int32_t x) : data{x}, next{nullptr} {}
};
```

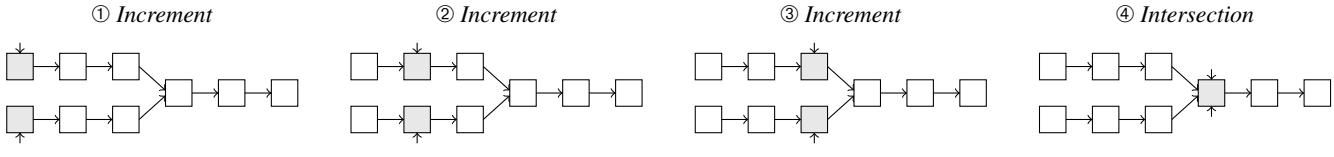
For example, given the following head pointers, you would return the node with the value `3`:



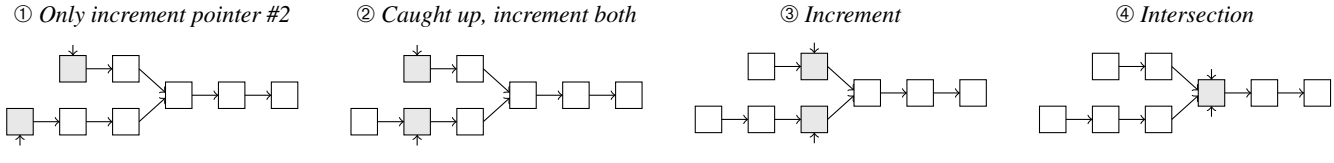
For this problem, we will go over two solutions: a relatively straightforward solution, and one that requires a bit more ingenuity. To start, let's consider the case where both lists have the same number of nodes before the point of intersection, as shown:



To find the node of intersection here, we can simply initialize two pointers, one that points to `head1` and one that points to `head2`. We then increment each pointer in tandem; if the two pointers ever end up pointing to the same node, then that node must be the point of intersection.



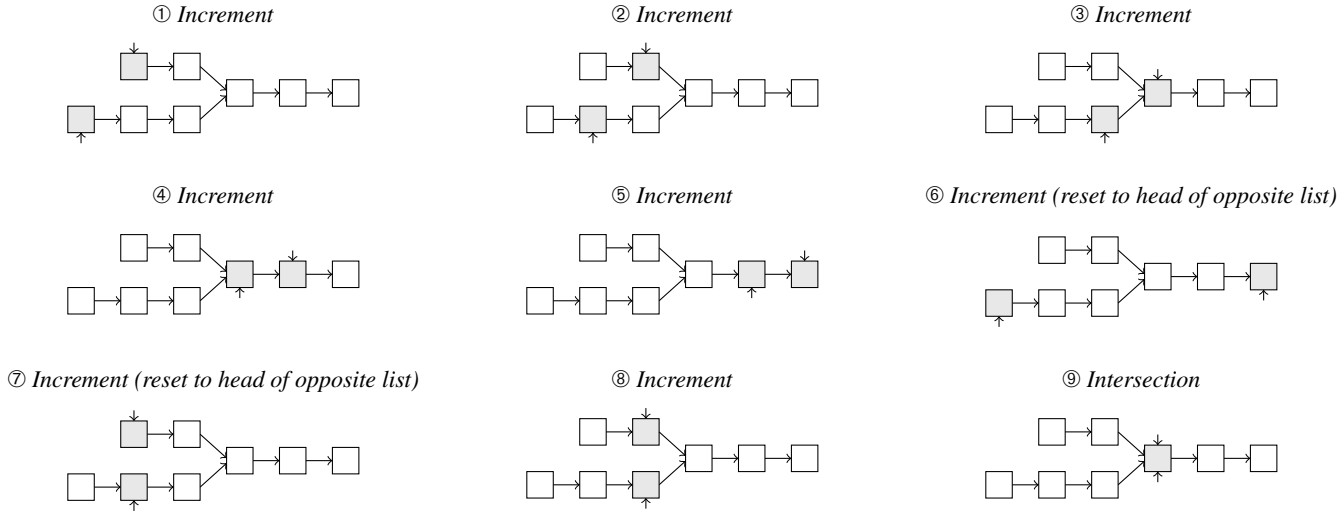
This approach does *not* work if the two lists have a different number of nodes before the point of intersection, as with our initial example. However, we can use this idea to come up with a solution; instead of incrementing both pointers immediately, we first allow the pointer in the longer chain to catch up with the pointer in the shorter chain. Then, we begin incrementing the two pointers in tandem, similar to before.



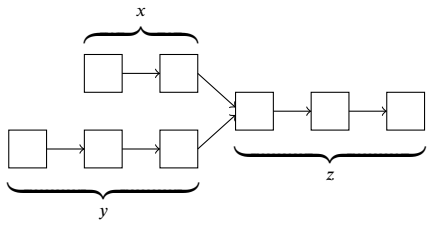
To determine when the pointer on the longer end has caught up with the pointer on the shorter end, we would need to know the difference in length between the linked list at `head1` and the one at `head2`. This requires us to do some preprocessing beforehand to compute the lengths of the two lists. Once we have the lengths, we find the difference and increment the pointer on the longer end by this difference to line it up with the pointer on the shorter end. The code for this is shown below:

```
1   // helper function that finds length of list
2   int32_t get_list_length(Node* head) {
3     int32_t length = 0;
4     while (head != nullptr) {
5       ++length;
6       head = head->next;
7     } // while
8     return length;
9   } // get_list_length()
10
11  Node* get_list_intersection(Node* head1, Node* head2) {
12    int32_t len1 = get_list_length(head1);
13    int32_t len2 = get_list_length(head2);
14
15    // increment pointer in longer list to catch up with pointer in shorter list
16    if (len1 < len2) {
17      int32_t diff = len2 - len1;
18      for (int32_t i = 0; i < diff; ++i) {
19        head2 = head2->next;
20      } // for i
21    } // if
22    else if (len2 < len1) {
23      int32_t diff = len1 - len2;
24      for (int32_t j = 0; j < diff; ++j) {
25        head1 = head1->next;
26      } // for j
27    } // else if
28
29    while (head1 != nullptr && head2 != nullptr && head1 != head2) {
30      head1 = head1->next;
31      head2 = head2->next;
32      if (head1 == head2) {
33        return head1;
34      } // if
35    } // while
36
37    // ternary handles the case where head1 and head2 initially point to the same node
38    return head1 == head2 ? head1 : nullptr;
39  } // get_list_intersection()
```

However, it is possible to write a solution using two pointers that does not need to precompute the lengths at all! This solution requires a key insight: we do not need to know the list lengths to align our two pointers if we simply allow each pointer to iterate over *both* lists. In other words, we are going to increment both pointers in tandem, regardless of how far away they are. However, once a pointer reaches the end, we restart it at the beginning of the opposite list (i.e., once the pointer of the shorter list reaches the end, reset it to the head of the longer one, and vice versa). By doing this, both pointers are guaranteed to meet at the same node if an intersection exists.



Why does this work? Recall from earlier that the two pointers are guaranteed to meet at the intersection point if they traverse the same number of nodes before this point of intersection. By forcing each pointer to restart back at the head of the opposite list once it reaches the end of its initial traversal, we ensure that both pointers traverse the same number of nodes before the point of intersection, regardless of the lengths of the lists.



Pointer starting at shorter head traverses x nodes in the shorter list, z nodes to the end, and then y nodes in the longer list.
Pointer starting at longer head traverses y nodes in the longer list, z nodes to the end, and then x nodes in the shorter list.
After traversing $x + y + z$ nodes, both pointers are guaranteed to meet at the intersection node, if there is one!

An implementation of this solution is shown below. If both pointers make it past the second iteration without meeting at the exact same node, then there is no intersection and `nullptr` is returned.

```
1   Node* get_list_intersection(Node* head1, Node* head2) {
2     Node *p1 = head1, *p2 = head2;
3     if (p1 == nullptr || p2 == nullptr) {
4       return nullptr;
5     } // if
6
7     while (p1 != nullptr && p2 != nullptr && p1 != p2) {
8       p1 = p1->next;
9       p2 = p2->next;
10
11      // return if both pointers meet at the same node
12      if (p1 == p2) {
13        return p1;
14      } // if
15
16      // if p1 reaches the end, restart it at the head of head2
17      if (p1 == nullptr) {
18        p1 = head2;
19      } // if
20
21      // if p2 reaches the end, restart it at the head of head1
22      if (p2 == nullptr) {
23        p2 = head1;
24      } // if
25    } // while()
26
27    return p1;
28  } // get_list_intersection()
```

The following is a concise version of the same solution. The ternary operators on lines 4 and 5 reset the pointers once they reach the end, and the condition of the `while` loop on line 3 handles the case when there is no intersection (since both would eventually point to `nullptr`):

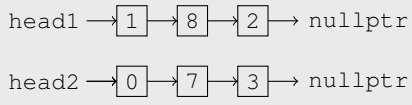
```
1   Node* get_list_intersection(Node* head1, Node* head2) {
2     Node *p1 = head1, *p2 = head2;
3     while (p1 != p2) {
4       p1 = p1 ? p1->next : head2;
5       p2 = p2 ? p2->next : head1;
6     } // while()
7     return p1;
8   } // get_list_intersection()
```

The time complexity of this solution is $\Theta(n)$, where n is the total number of nodes in the combined lists, since this is the number of nodes that each pointer may have to traverse if there is an intersection. If there is no intersection, the pointers may need to visit more than n nodes, but the total number of nodes visited cannot be greater than $2n$ (since the algorithm terminates once both pointers hit a `nullptr` on the second traversal), so the time complexity in this scenario would still be $\Theta(n)$. The auxiliary space is $\Theta(1)$ since the amount of additional memory allocated by the function is a constant that does not depend on the size of the lists that are passed in.

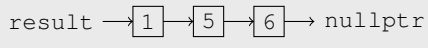
Example 8.8 You are given two non-empty singly-linked lists that represent two non-negative integers. The digits in the linked list are stored in reverse order, such that the least-significant digit is stored at the head, and the most-significant digit is stored at the tail. Each node only stores a single digit, and the two lists do not contain any leading zeros. Implement a function that adds the two numbers represented by the two given lists and returns the sum as a linked list in the same format. A node is defined as follows:

```
struct Node {
  int8_t digit;
  Node* next;
  Node(int8_t x) : digit{x}, next{nullptr} {}
};
```

For example, the following two lists represent the numbers 281 and 370:



If given these two lists, your function should return the sum of these two numbers, 651, as a list with the same reverse-digit format:

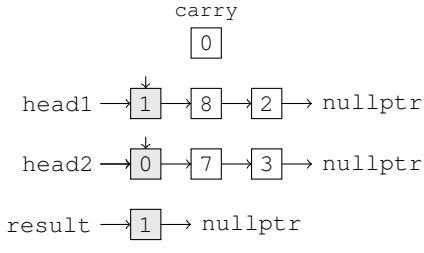


This problem may seem complicated at first, but the algorithm for solving this problem is the same as the one you learned in grade school: add the digits from right to left, and carry any additional significant digits for values over 9 to the next column.

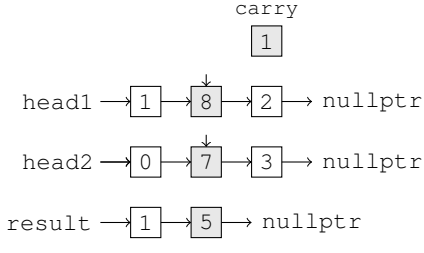
```
  2 8 1      2 8 1        1          1
+ 3 7 0    + 3 7 0      2 8 1      2 8 1
-------    -------    + 3 7 0    + 3 7 0
           -------      -----      -----
    1          1          5 1        6 5 1
```

Because the digits in the list are stored in reverse order, we can follow this process by iterating over both of the lists from front to back and adding the digits together. Much like our initial algorithm, we will also keep a carry variable that keeps track of any value we need to carry. A depiction of this process is shown below:

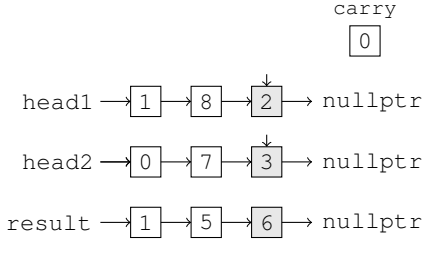
1 + 0 = 1, so we append a node with value 1 to the back of the result list.



8 + 7 = 15, so we append a node with value 5 to the back of the result list and set the carry value to 1.

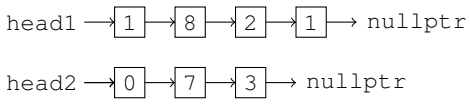


1 + 2 + 3 = 6, so we append a node with value 6 to the back of the result list.

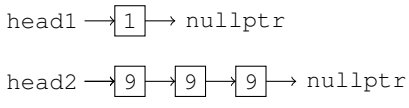


A non-trivial portion of this problem's difficulty comes from identifying the edge cases that may be encountered. In our example, we used two three-digit numbers that also summed up to a three-digit number. This made list construction a fairly straightforward process. However, this is not guaranteed, and your algorithm will need to be able to handle additional edge cases. A few of these edge cases are highlighted below.

One list is longer than the other (make sure you do not iterate off the end).



There is an additional carry value at the end (which should be appended as an additional node at the end of the list).



An implementation of this solution is provided below.

```
1  Node* add_two_lists(Node* head1, Node* head2) {
2    // same as previous example, the dummy node makes implementation easier
3    // pre_head.next is the head of the actual list we want to return
4    Node pre_head{0}, *result = &pre_head;
5    Node *iter1 = head1, *iter2 = head2;
6    int8_t carry = 0;
7    // continue iterating as long as one list still has nodes to visit
8    while (iter1 != nullptr || iter2 != nullptr) {
9      int8_t sum = (iter1 ? iter1->val : 0) + (iter2 ? iter2->val : 0) + carry;
10     // the value of the next digit is the remainder of sum when divided by 10
11     result->next = new Node{sum % 10};
12     result = result->next;
13     // value of carry is the number of times sum evenly divides into 10 (here, either 0 or 1)
14     carry = sum / 10;
15     // iterate as long as there are more nodes to visit
16     if (iter1 != nullptr) {
17       iter1 = iter1->next;
18     } // if
19     if (iter2 != nullptr) {
20       iter2 = iter2->next;
21     } // if
22   } // while
23   // if carry left over, append it as the final node
24   if (carry != 0) {
25     result->next = new Node{carry};
26   } // if
27   return pre_head.next;
28 } // add_two_lists()
```

Assuming that the two lists have lengths of m and n, the time complexity of this algorithm is $\Theta(\max(m,n))$, since the algorithm iterates over both lists (and the total number of iterations is dependent on whichever list is longer). The auxiliary space used by this algorithm is also $\Theta(\max(m,n))$, since the length of the return list (i.e., the number of digits in the sum) is at least $\max(m,n)$ and at most $\max(m,n)+1$.

Remark: What if we are given the same problem, but with the list direction reversed? That is, what if the most significant digits were stored near the head of the list rather than near the tail? If such a list were singly-linked, we would not be able to iterate over the list as easily as when the digits were stored in reverse order, like in the previous example. However, there are several approaches that can be used to address this limitation. One approach is to reverse the linked list (using the algorithm covered in section 8.4) and then repeat the above process once the digits are in reverse order. Another potential approach is to use a last-in, first-out container like a `std::stack<>` to help you access the digits in the correct order — we will cover stacks in greater detail in the next chapter.

8.6 The STL List Container (⁎)

Much like the `std::vector<>`, the C++ standard template library provides its own implementations of linked lists. A brief overview of these containers is provided below. You likely will not ever use any of these containers for EECS 281, but these are included for your reference since a few upper-level classes do use them.

※ 8.6.1 *std::list*

First, the STL implementation of a *doubly*-linked list is defined as a `std::list<>`, and it can be used if you `#include <list>`. A few `std::list<>` operations are shown in the table on the next page. Note that this table is not comprehensive — for full coverage on list operations, you are encouraged to read the STL `<list>` documentation.

The following methods can be used to initialize a `std::list<>`:

```
template <typename T>
std::list<T>();
```
Default constructor for list that holds elements of type `T`. Creates an empty list without any elements; size is initially 0.

```
template <typename T>
std::list<T>(const std::list<T>& other);
```
Copy constructor, copies the contents of `other` into the constructed list.

```
template <typename T>
std::list<T>(size_t sz);
```
Creates a list of `sz` elements, where each element is value initialized; size equal to `sz`.

```
template <typename T>
std::list<T>(size_t sz, T& val);
```
Creates a list of `sz` elements, where each element is initialized to a value of `val`; size equal to `sz`.

```
template <typename T, typename InputIterator>
std::list<T>(InputIterator begin_iter, InputIterator end_iter);
```
Creates a list with all elements in the iterator range `[begin_iter, end_iter)` — inclusive begin but exclusive end. Both `begin_iter` and `end_iter` are input iterators.

```
template <typename T>
std::list<T>(std::initializer_list<T> init);
```
Initializes the list with the contents of the initializer list.

The following methods can be used to insert and remove elements from the front or back of a `std::list<>`.

```
template <typename T>
void std::list<T>::push_back(const T& val);
```
Appends an element `val` to the back of the list.

```
template <typename T, typename... Args>
T& std::list<T>::emplace_back(Args&&... args);
```
Constructs a new element in place at the back of the list using the constructor arguments that are passed in. Returns reference since C++17.

```
template <typename T>
void std::list<T>::pop_back();
```
Removes last element in list; references/iterators to the erased element are invalidated. Causes undefined behavior if used on an empty list.

```
template <typename T>
void std::list<T>::push_front(const T& val);
```
Appends an element `val` to the front of the list.

```
template <typename T, typename... Args>
T& std::list<T>::emplace_front(Args&&... args);
```
Constructs a new element in place at the front of the list using the constructor arguments that are passed in. Returns reference since C++17.

```
template <typename T>
void std::list<T>::pop_front();
```
Removes first element in list; references/iterators to the erased element are invalidated. Causes undefined behavior if used on an empty list.

```
template <typename T>
void std::list<T>::resize(size_t sz, const T& val);
```
Resizes the container to hold `sz` elements, each initialized to `val`. The second argument is optional, and if omitted, new elements are value-initialized and added to the list until its size is `sz`. If current size is greater than `sz`, the container is reduced to the first `sz` elements.

The `.insert()` and `.erase()` member functions of a list behave similarly to those of a vector:

```
template <typename T>
iterator std::list<T>::insert(const_iterator pos, const T& val);
```
Inserts `val` directly before the element pointed to by the iterator `pos` and returns an iterator the newly inserted element.

```
template <typename T>
iterator std::list<T>::insert(const_iterator pos, size_t n, const T& val);
```
Inserts `n` copies of `val` directly before the element pointed to by the iterator `pos` and returns an iterator the first element added.

```
template <typename T, typename InputIterator>
iterator std::list<T>::insert(const_iterator pos, InputIterator first, InputIterator last);
```
Inserts all elements in the iterator range `[first, last)` directly before `pos` and returns an iterator the first new element added.

```
template <typename T>
iterator std::list<T>::insert(const_iterator pos, std::initializer_list<T> init);
```
Inserts the elements in the initializer list directly before the iterator `pos` and returns an iterator to the first new element added.

```
template <typename T, typename... Args>
iterator std::list<T>::emplace(const_iterator pos, Args&&... args);
```
Inserts a new element directly before the element at position `pos`. The new element is constructed in place using arguments for its constructor (`args`). An iterator pointing to the emplaced object is returned.

```
template <typename T>
iterator std::list<T>::erase(iterator pos);
```
Erases the element pointed to by the iterator `pos` and returns an iterator to the element following the one that was erased.

```
template <typename T>
iterator std::list<T>::erase(iterator first, iterator last);
```
Erases all elements in the range `[first, last)` and returns an iterator to the element following the last element erased.

The `std::list<>` container also provides a `.splice()` method that can be used to transfer nodes around, either to another list or to a different position in the same list. Because of how a list stores its data, this method simply repoints the internal pointers of the lists, and no iterators or references are invalidated. Thus, splicing a single item in a list takes constant time.

```
template <typename T>
void std::list<T>::splice(const_iterator pos, list& other);
```
Transfers all elements from `other` into the list on which `.splice()` is called (i.e., `*this`). The elements are inserted before the element pointed to by `pos`. The list `other` becomes empty after this operation. If `other` is the same as `*this` (the list that `.splice()` is called on), this method produces undefined behavior.

```
template <typename T>
void std::list<T>::splice(const_iterator pos, list& other, const_iterator it);
```
Transfers the element pointed to by `it` from `other` into the list on which `.splice()` is called (i.e., `*this`). The element is inserted before the element pointed to by `pos`.

```
template <typename T>
void std::list<T>::splice(const_iterator pos, list& other, const_iterator first, const_iterator last);
```
Transfers the elements in the range `[first, last)` from `other` into the list on which `.splice()` is called (i.e., `*this`). The elements are inserted before the element pointed to by `pos`. Behavior is undefined if `pos` is an iterator in the range `[first, last)`.

Some additional `std::list<>` member functions are summarized below. Note that lists have their own `.sort()` method (unlike most other containers); this is because the algorithm library's generic `std::sort()` function cannot be used on a list (for reasons we will discuss later).

Function	Behavior
`.front()`	Returns a reference to the first element in the list
`.back()`	Returns a reference to the last element in the list
`.empty()`	Returns whether the list is empty
`.size()`	Returns the number of elements in the list
`.clear()`	Clears the contents of the list
`.begin()`	Returns a bidirectional iterator to the first element in the list
`.end()`	Returns a bidirectional iterator to the position one past the last element in the list
`.cbegin()`	Returns a *constant* bidirectional iterator to the first element in the list
`.cend()`	Returns a *constant* bidirectional iterator to the position one past the last element in the list
`.rbegin()`	Returns a *reverse* iterator to the last element in the list
`.rend()`	Returns a *reverse* iterator to the position one before the first element in the list
`.crbegin()`	Returns a constant reverse iterator to the last element in the list
`.crend()`	Returns a constant reverse iterator to the position one before the first element in the list
`.sort()`	Sorts the contents of the list in ascending order (or using an optional comparator that is passed in)

Examples of list operations are shown in the code below:

```
// initializes the list lst1 with zero size
std::list<int32_t> lst1;
// initializes lst2 to have contents {1, 2, 3}
std::list<int32_t> lst2 = {1, 2, 3};
// push 0 to the front of lst2
lst2.push_front(0);
// push 4 to the back of lst2
lst2.push_back(4);
// initializes lst3 with the contents of lst2, or {0, 1, 2, 3, 4}
std::list<int32_t> lst3{lst2};
// pops 0 off the front of lst3
lst3.pop_front();
// pops 4 off the back of lst3
lst3.pop_back();
// resizes lst3 to size 5
lst3.resize(5);  // contents of lst3 are now {1, 2, 3, 0, 0}
// sorts lst3
lst3.sort();  // contents of lst3 are now {0, 0, 1, 2, 3}
```

※ 8.6.2 *std::forward_list* (✻)

The STL also provides an implementation for a *singly*-linked list, defined as a `std::forward_list<>`. A forward list can be used if you `#include <forward_list>`. This container is typically preferable to a `std::list<>` in cases where bidirectional iteration is not needed, as forward lists do not keep track of a previous pointer (thereby saving memory). However, forward lists are also more limited in terms of functionality; they do not support `.push_back()` or `.pop_back()` operations, nor do they support reverse iterators. They also do not support a `.size()` member function for efficiency purposes.

Forward lists have several practical uses and are optimized for containers that are typically empty or have very small sizes. A few member functions of the `std::forward_list<>` container are summarized below. To explore this container's full functionality, you are encouraged to read the STL documentation for the `<forward_list>` container class.

`template <typename T>` `std::forward_list<T>();`
Default constructor for forward list that holds elements of type `T`. Creates an empty list without any elements.
`template <typename T>` `std::forward_list<T>(const std::forward_list<T>& other);`
Copy constructor, copies the contents of `other` into the constructed forward list.
`template <typename T>` `std::forward_list<T>(size_t sz);`
Creates a list of `sz` elements, where each element is value initialized.
`template <typename T>` `std::forward_list<T>(size_t sz, T& val);`
Creates a list of `sz` elements, where each element is initialized to a value of `val`.
`template <typename T, typename InputIterator>` `std::forward_list<T>(InputIterator begin_iter, InputIterator end_iter);`
Creates a list with all elements in the iterator range `[begin_iter, end_iter)` — inclusive begin but exclusive end. Both `begin_iter` and `end_iter` are input iterators.
`template <typename T>` `std::forward_list<T>(std::initializer_list<T> init);`
Initializes the forward list with the contents of the initializer list.
`template <typename T>` `void std::forward_list<T>::clear();`
Clears out the contents of the forward list.
`template <typename T>` `bool std::forward_list<T>::empty();`
Returns whether the forward list is empty.
`template <typename T>` `iterator std::forward_list<T>::insert_after(iterator pos, const T& val);`
Inserts `val` after the element pointed to by `pos` and returns an iterator to the inserted element.
`template <typename T>` `iterator std::forward_list<T>::insert_after(const_iterator pos, const T& val);`
Inserts `val` after the element pointed to by `pos` and returns an iterator to the inserted element.
`template <typename T>` `iterator std::forward_list<T>::insert_after(const_iterator pos, size_t n, const T& val);`
Inserts `n` copies of `val` after the element pointed to by `pos` and returns an iterator to the last element inserted.
`template <typename T, typename InputIterator>` `iterator std::forward_list<T>::insert_after(const_iterator pos, InputIterator first, InputIterator last);`
Inserts all elements in the iterator range `[first, last)` after the element at `pos` and returns an iterator to the last element inserted.

```
template <typename T>
iterator std::forward_list<T>::insert_after(const_iterator pos, std::initializer_list<T> init);
```
Inserts all elements in the initializer list after the element at `pos` and returns an iterator to the last element inserted, or `pos` if the given initializer list is empty.

```
template <typename T, typename... Args>
iterator std::forward_list<T>::emplace_after(const_iterator pos, Args&&... args);
```
Constructs a new element in place after the element at `pos` using the constructor arguments that are passed in.

```
template <typename T>
iterator std::forward_list<T>::erase_after(const_iterator pos);
```
Erases the element after the one pointed to by the iterator `pos` and returns an iterator to the element following the one that was erased.

```
template <typename T>
iterator std::forward_list<T>::erase_after(const_iterator first, const_iterator last);
```
Erases all elements after the one pointed to by the iterator `first`, up until the element pointed to by `last`, and returns an iterator to the element following the last element erased.

```
template <typename T>
void std::forward_list<T>::push_front(const T& val);
```
Prepends `val` to the beginning of the forward list.

```
template <typename T, typename... Args>
T& std::forward_list<T>::emplace_front(Args&&... args);
```
Constructs a new element in place at the front of the forward list, using the given constructor arguments. Returns reference since C++17.

```
template <typename T>
void std::forward_list<T>::pop_front();
```
Removes the first element of the container (undefined behavior if list is empty).

8.7 Summary of List Complexities

In this section, we will summarize of time complexities of several list operations. Note that a `head` pointer is required, but a `tail` pointer is optional depending on implementation. The complexities of both implementation variations are provided.

Operation	Type	Average-Case Time	Worst-Case Time
Finding an element	Singly-linked	$\Theta(n)$	$\Theta(n)$
	Doubly-linked	$\Theta(n)$	$\Theta(n)$

Regardless of whether the list is singly- or doubly-linked, the worst-case of finding an element in a list of size n occurs when the element is the last one in the list, or if the element cannot be found at all. This requires the search to look at all n elements in the list. In the average case, the element is somewhere in the middle of the list, which would require the algorithm to look at approximately $n/2$ elements. This is still $\Theta(n)$ since we can drop the coefficient term of $1/2$.

In the STL, this can be done using `std::find()` in the `<algorithm>` library, which will be covered in chapter 11.

Operation	Type	Average-Case Time	Worst-Case Time
Accessing first element	Singly-linked	$\Theta(1)$	$\Theta(1)$
	Doubly-linked	$\Theta(1)$	$\Theta(1)$

Since the `head` pointer points to the first element of the list, the first element of a list can be accessed in constant time regardless of whether the list is singly- or doubly-linked. In the STL, this can be done using the `.front()` member of both lists and forward lists.

Operation	Type	Average-Case Time	Worst-Case Time
Accessing last element	Singly-linked	$\Theta(n)$ if no tail pointer	$\Theta(n)$ if no tail pointer
		$\Theta(1)$ if tail pointer	$\Theta(1)$ if tail pointer
	Doubly-linked	$\Theta(n)$ if no tail pointer	$\Theta(n)$ if no tail pointer
		$\Theta(1)$ if tail pointer	$\Theta(1)$ if tail pointer

The efficiency of accessing the last element depends on whether a `tail` pointer exists. If it exists, you can just refer to it to retrieve the last element. Otherwise, you will have to iterate through all n elements to get to the last element. For the STL list (doubly-linked), this can be done using the `.back()` member method. On the other hand, STL forward lists (singly-linked) do not support `.back()`, so you would need to iterate until the `.end()` iterator if you wanted to find the last element.

Operation	Type	Average-Case Time	Worst-Case Time
Accessing arbitrary element	Singly-linked	$\Theta(n)$	$\Theta(n)$
	Doubly-linked	$\Theta(n)$	$\Theta(n)$

Unlike a vector, elements in a list are not contiguous in memory. As a result, you cannot use arithmetic to access arbitrary elements in constant time. You would have to iterate through all the nodes of the list in the worst-case, which is $\Theta(n)$ for a list size of n. In the average-case, the element you are trying to access is in the middle, which would still require you to iterate through $n/2$ elements — this is also a $\Theta(n)$ operation.

Operation	Type	Average-Case Time	Worst-Case Time
Inserting element at front	Singly-linked	$\Theta(1)$	$\Theta(1)$
	Doubly-linked	$\Theta(1)$	$\Theta(1)$

To insert an element at the front of the list, all you have to do is change some pointers around. Since you can access the front of the list directly using the `head` pointer, this is a constant time operation regardless of the type of list you are trying to insert into. For STL lists and forward lists, this can be done using the `.push_front()` and `.emplace_front()` members.

Operation	Type	Average-Case Time	Worst-Case Time
Inserting element at back	Singly-linked	$\Theta(n)$ if no tail pointer	$\Theta(n)$ if no tail pointer
		$\Theta(1)$ if tail pointer	$\Theta(1)$ if tail pointer
	Doubly-linked	$\Theta(n)$ if no tail pointer	$\Theta(n)$ if no tail pointer
		$\Theta(1)$ if tail pointer	$\Theta(1)$ if tail pointer

If you have access to a `tail` pointer, this operation can be done in constant time, since you just need to modify a few pointers to insert the node. However, if there is no tail pointer, you will have to traverse the list before you can insert the element. Note that this complexity assumes you are not given the node to insert after. If you are passed in the actual node to insert after, then this insertion becomes a constant time operation (much like with the `tail` pointer). For the STL list, this can be done using `.push_back()` or `.emplace_back()`. For singly-linked STL forward lists, `.push_back()` and `.emplace_back()` are not supported, and insertion at the back can be done by using `.insert_after()` or `.emplace_after()` on the last element in the list.

Operation	Type	Average-Case Time	Worst-Case Time
Inserting element at arbitrary index	Singly-linked	$\Theta(n)$	$\Theta(n)$
	Doubly-linked	$\Theta(n)$	$\Theta(n)$

Given just an index of insertion, you would have to iterate through the list first to get the position you want to insert at. This is why the time complexities here are $\Theta(n)$. Again, if you were given a pointer to the node to insert after rather than an index, insertion would be a constant time operation. In the STL, insertion can be done using `.insert()` and `.emplace()` for a doubly-linked list, or `.insert_after()` and `.emplace_after()` for a singly-linked forward list (these STL methods take constant time, but you must pass in an iterator to the insertion position, which could take linear time to obtain).

Operation	Type	Average-Case Time	Worst-Case Time
Erasing element at front	Singly-linked	$\Theta(1)$	$\Theta(1)$
	Doubly-linked	$\Theta(1)$	$\Theta(1)$

Since you have access to the first element through the `head` pointer, you can erase the first element my shifting a few pointers around. This takes constant time, regardless of the type of list. In the STL, this can be done using the `.pop_front()` method of both lists and forward lists.

Operation	Type	Average-Case Time	Worst-Case Time
Erasing element at back	Singly-linked	$\Theta(n)$	$\Theta(n)$
	Doubly-linked	$\Theta(n)$ if no tail pointer	$\Theta(n)$ if no tail pointer
		$\Theta(1)$ if tail pointer	$\Theta(1)$ if tail pointer

Regardless of whether you are given a pointer to the last node of the list, the complexity of deleting the last element in a singly-linked list is $\Theta(n)$. This is because you will need to change the `next` value of the node directly before the last node, which you do not have access to without iterating through the list. For a doubly-linked list, erasing at the back is $\Theta(1)$ if you have a `tail` pointer, because you can access the node before the last node using the last node's `prev` pointer (which isn't possible in a singly-linked list). For the STL's doubly-linked list, you can erase the last element using `.pop_back()`. For the STL's singly-linked forward lists, the `.pop_back()` operation is not supported, so you will have to find the element directly before the one at the back and pass its iterator to the `.erase_after()` member function.

Operation	Type	Average-Case Time	Worst-Case Time
Erasing element at arbitrary index	Singly-linked	$\Theta(n)$	$\Theta(n)$
	Doubly-linked	$\Theta(n)$	$\Theta(n)$

If you are given the index of the node to delete, you must first iterate to the correct position of the list (since you can't use constant-time pointer arithmetic to get to the node you want to delete). This is why erasing a node given only its index is a linear time operation.

Operation	Type	Average-Case Time	Worst-Case Time
Erasing element given its pointer	Singly-linked	$\Theta(n)$	$\Theta(n)$
	Doubly-linked	$\Theta(1)$	$\Theta(1)$

To delete a node, you must update the `next` pointer of the node directly before the one you want to delete, and (if doubly-linked) the `prev` pointer of the node directly after the one you want to delete. With a doubly-linked list, you can access these two nodes in constant time, so the entire deletion process is also constant time (you just need to modify some pointers). However, singly-linked lists do not support a `prev` pointer, so you cannot access the node directly before the one you want to delete in constant time. As a result, you must iterate through the list to access this previous node, which is why erasing from a singly-linked list is still $\Theta(n)$ even if you are given a pointer to the node you want to delete. In the STL, erasure can be done using `.erase()` for a doubly-linked list, or `.erase_after()` for a singly-linked forward list.

Operation	Type	Average-Case Time	Worst-Case Time
Checking if list is empty	Singly-linked	$\Theta(1)$	$\Theta(1)$
	Doubly-linked	$\Theta(1)$	$\Theta(1)$

This can be done in constant time by just checking if `head` is `nullptr`. For the STL's list and forward list containers, you can use the `.empty()` method to determine if the list is empty.