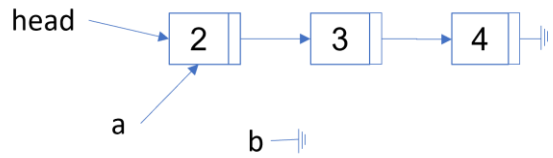
Drawing a Linked List

Consider this starting C++ data structure:



Draw the data structure that results after running this code fragment on the starting data structure:

```
Node* c = a->next;
a->next = b;
b = a;
a = c;
```

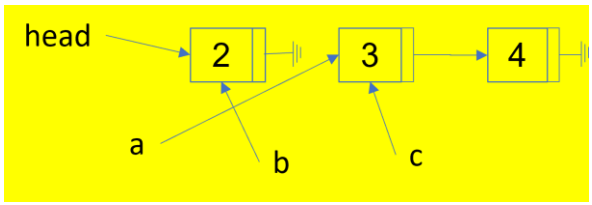
Make sure your drawing has everything: all variables, values, and pointers. Make it clear and easy to read.

Nodes have this type:

```
struct Node {
   int data;
   Node* next;
};
```

The variables `head`, `a`, and `b` are all of type `Node*`.

Solution



Linked Lists: Tail Deletion

Suppose you have a C++ program with a **singly-linked list** based on this:

```
struct Node {
   int data;
   Node* next;
};

Node* head = nullptr;  // head points to the first node on the list
```

`head` is a global variable that points to the first element of the list. If `head` is `nullptr`, then the list is empty.

Question

Using **detailed C++-like pseudocode**, write a function called `remove_last()` that:

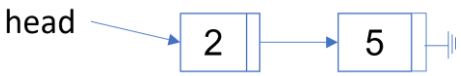
- Returns a copy of the data value of the last element in the list.
- Removes the last element, leaving a valid list with the other elements in the same order.
- Traverses the list at most one time.
- **Doesn't** use recursion.

You can assume that the list is **not** empty.

For example, suppose your program creates a list like this:

```
head = new Node{4, nullptr};
head = new Node{5, head};
head = new Node{2, head};
```

When you call `remove_last()` it returns 4, and the last node is deleted from the list:



Your `remove_last()` should work for *any* valid singly-linked list, not just this example. Your answer should be efficient, not use any unnecessary memory, and have **no memory leaks** or other errors.

Don't use arrays or vectors or other such data structures in your answer.

Sample Solution

The idea here is to separate consider two cases:

- Case 1: A list with a single element (a singleton list). This requires changing the head pointer.
- Case 2: A list with two or more elements. This does not require changing the head pointer.

In case 2, we first walk a pointer `p` from the start of the list to the node *before* the last node. With this pointer we can get the data value of the last node and correctly delete it. It's important that `p` points to the node before the last node because that node will become the new last point and so it's next value must be changed to `nullptr`.

```
// Pre-condition:
//    head is not nullptr (i.e. list is not empty)
//
// Post-condition:
//    Returns a copy of the last element of the list, and deletes
//    the final node.
//
int remove_last() {
    // single-node list is a special case since we need to set
    // head to nullptr
    if (head->next == nullptr) {
        int result = head->data;
        delete head;
        head = nullptr;
        return result;
    } else { // list has 2 or more elements
        // Use a loop to make p point to the node before
        // the last node (so we can change its next pointer).
        Node *p = head;
        while (p->next->next != nullptr) {
            p = p->next;
        }
        int result = p->next->data;
        delete p->next;
        p->next = nullptr;
        return result;
    }
}
```

Linked Lists: Copying

Suppose you have a C++ program with a **singly-linked list** based on this:

```
struct Node {
   int data;
   Node* next;
};

Node* head = nullptr;  // head points to the first node on this list
```

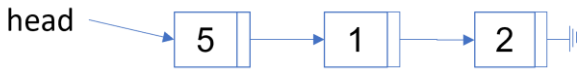
`head` is a global variable that points to the first element of the list. If `head` is `nullptr`, then the list is empty.

Question

Using **detailed C++-like pseudocode**, write a function called `copy_list()` that *returns* a pointer to the first node of a new singly-linked list that is a copy of the one head points to. The nodes in the copy are brand new nodes, and the list contains the same values, in the same order, as the original list. `head`, and the original list, are *not* modified in any way.

For example, suppose your program creates a list like this:

```
head = new Node{2, nullptr};
head = new Node{1, head};
head = new Node{5, head};
```



Then a copy is made after running this code:

```
Node* p = copy_list();
```

The original list is still there, unchanged.

Your `copy_list()` should work for any valid singly-linked list, not just this example! Make your answer efficient, and **don't** use arrays or vectors or other such tricks in your answer.

Sample Solution

The idea is to construct a new list using two pointers: the new head node, and also a tail node that always points to the last element of the list. The tail pointer lets us add the next node at the end of the list.

```
Node* copy_list() {
    Node *new_head = nullptr;
    Node *new_tail = nullptr;

    for (Node *p = head; p != nullptr;
                         p = p->next)
    {
        Node* new_node = new Node{p->data, nullptr};
        if (new_head == nullptr) {
            new_head = new_node;
            new_tail = new_node;
        } else {
            new_tail->next = new_node;
            new_tail = new_node;
        }
    }
    return new_head;
}
```

Depth of a Tree

The **depth of a node** `p` in a tree is defined to be the number of ancestors of `p`, excluding `p` itself. Using **detailed C++-like pseudocode**, write a function called `depth(T, p)` that **uses recursion** to calculate and return the depth of node `p` in tree `T`.

Assume the following:

- `T` is non-empty, and `p` points to a node in it
- `p->isRoot()` returns *true* if `p` points to the root of the tree, and *false* otherwise
- `p->parent` points to the parent of `p` (`nullptr` if `p` is the root)

Solution

This is a pseudocode solution that follows the recursion definition of a the depth of a node in a tree:

- Base case: the depth of the root is 0
- Recursive case: the depth of a non-root node is 1 plus the depth of its parent

```
depth(T, p)
   if p->isRoot()
      return 0
   else
     return 1 + depth(T, p->parent())
```

O-notation: Linear functions

a) State the mathematical definition of "$f(n)$ is $O(g(n))$", as given in the textbook.

Sample Solution

$f(n)$ is $O(g(n))$ if there exists constants $c > 0$ and $n_0 \geq 1$ such that this inequality holds:

$f(n) \leq cg(n)$ when $n \geq n_0$

b) Using the mathematical O-notation definition given in the textbook, prove that $an + b$ is $O(n)$, where $a > 0$ and b are both constant integers.

Sample Solution

According to the definition of O-notation given in the textbook, to prove $an + b$ is $O(n)$ we must find constants $c > 0$ and $n_0 \geq 1$ such that this inequality holds:

$an + b \leq cn$ when $n \geq n_0$

If we set c to a value greater than a, we can subtract an from both sides of the inequality to get a positive value on the right side. So setting $c = a + 1$ gives us:

$an + b \leq (a + 1)n$ when $n \geq n_0$

$an + b \leq an + n$ when $n \geq n_0$

$b \leq n$ when $n \geq n_0$

Since b is a fixed constant and n is a variable that can grow to be as big as we like, the inequality is true when n is b or greater. So setting $n_0 = b$ makes the inequality true.

We're done: we have shown that setting $c = a + 1$ and $n_0 = b$ make the inequality true.

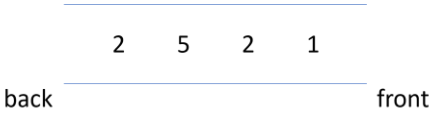
Summing a Queue

Suppose the `Queue` class contains `int`s, and has these methods:

- `Q.enqueue(x)` adds `x` to the back of `Q`
- `Q.dequeue()` removes the item at the front of `Q`; it's an error if `Q` is empty
- `Q.front()` returns a copy of the item at the front of `Q`; it's an error if `Q` is empty
- `Q.size()` returns the number of items in `Q`

Question
Write a function (**not** a method!) called `sum(Queue& Q)` that returns the sum of all the `int`s in `Q`. After calling `sum(Q)` the elements of `Q` should be the same, and in the same order, as before sum was called.

For example, suppose `Q` contains these elements:



Then `sum(Q)` returns 10, and `Q` looks exactly like it did before calling `sum`.

Important *Don't* use any extra data structures, e.g. no arrays, no vectors, no lists, no stacks, etc. You can use as many queues as you like in your answer (but try to use as few as possible). You cannot assume anything about how the queue is implement: only use methods defined above for it in your answer.

Important `sum` **can** modify `Q`, but when it's done `Q` should have the same elements in the same order as when it started.

Important `sum(Queue& Q)` is a function, not a method!

Sample Solution

The trick here is a standard one with queues: de-queue every item and then enqueue back into the queue. As you do this, keep a running total of the removed elements. Assuming enqueue and dequeue are both O(1) operations, this is an O(n) function (where n is the number of items in the queue).

```
sum(Q)
   int total = 0
   for(int i = 0; i < Q.size(); i++) {
      int n = Q.front()
      Q.dequeue()
      Q.enqueue(n)
      total += n
   }
return n
```

Algorithm Performance Estimation 1

If a **quadratic** algorithm takes 3 seconds to process 50 items, about how long would you expect it to take to process 100 items? Justify you answer.

Sample Solution

Suppose an algorithm does n^2 key operations when it processes an input of size n. If t is the time it takes to do one key operation, then the time it takes to process n items is $T(n) = t \cdot n^2$. If you double the input size, then the algorithm will take $T(2n)$ time, which we can write like this:

$$T(2n) = t \cdot (2n)^2 = t \cdot 4n^2 = 4\,T(n)\ .$$

So we have the formula $T(2n) = 4\,T(n)$.

If it takes 3 seconds for the algorithm to process an input of size $n = 50$, then $T(50) = 3$. To process $n = 100$ items, it will take $T(100)$ time, which can be calculated using the $T(2n)$ formula:

$$T(100) = T(2 \cdot 50) = 4T(50) = 4 \cdot 3 = 12 \text{ seconds}$$

This shows that when you *double* the input to an $O(n^2)$ algorithm, the running time *quadruples*, i.e. it increases by a factor of 4. In contrast, if you double the size of the input to an $O(n)$ algorithm, the running time only doubles.

Algorithm Performance Estimation 2

If an **exponential** algorithm takes 3 seconds to process 50 items, about how long would you expect it to take to process 100 items? Justify your answer.

Sample Solution

Suppose an algorithm does 2^n key operations in the worst case when it processes an input of size n. If t is the time it takes to do one key operation, then the time it takes to process n items is $T(n) = t \cdot 2^n$. If you double the input size, then the algorithm takes $T(2n)$ time, which we can write like this:

$$T(2n) = t \cdot 2^{2n} = t \cdot 2^n \cdot 2^n = 2^n \cdot T(n) .$$

So we have the formula $T(2n) = 2^n \cdot T(n)$.

If it takes 3 seconds for the algorithm to process an input of size $n = 50$, then $T(50) = 3$. To process $n = 100$ items, it will take $T(100)$ time, which can be calculated using the $T(2n)$ formula:

$$T(100) = T(2 \cdot 50) = 2^{50} \cdot T(50) = 3 \cdot 2^{50} \text{ seconds} \approx 107 \text{ million years}$$

This shows that when you *double* the input to an $O(2^n)$ algorithm, the running time increases by a factor of 2^n.