Linked lists

Node implementation

Python3

```
class ListNode:
    def __init__(self, val):
        self.val = val
        self.next = None

one = ListNode(1)
two = ListNode(2)
three = ListNode(3)
one.next = two
two.next = three
head = one

print(head.val)
print(head.next.val)
print(head.next.next.val)
```

JavaScript

```
class ListNode {
    constructor(val) {
        this.val = val;
        this.next = null;
    }
}

(function main() {
    let one = new ListNode(1);
    let two = new ListNode(2);
    let three = new ListNode(3);
    one.next = two;
    two.next = three;
```

```
    let head = one;

    console.log(head.val);
    console.log(head.next.val);
    console.log(head.next.next.val);
}());
```

Java

```
public class Example {
    public static class ListNode {
        int val;
        ListNode next;
        ListNode (int val) {
            this.val = val;
        }
    }

    public static void main(String[] args) {
        ListNode one = new ListNode(1);
        ListNode two = new ListNode(2);
        ListNode three = new ListNode(3);
        one.next = two;
        two.next = three;
        ListNode head = one;

        System.out.println(head.val);
        System.out.println(head.next.val);
        System.out.println(head.next.next.val);
    }
}
```

C++

```
struct LinkedListNode {
    int val;
    LinkedListNode *next;
    LinkedListNode(int val): val (val), next(nullptr) {}
};
```

```
int main() {
    LinkedListNode* one = new LinkedListNode(1);
    LinkedListNode* two = new LinkedListNode(2);
    LinkedListNode* three = new LinkedListNode(3);
    one->next = two;
    two->next = three;
    LinkedListNode* head = one;

    cout << head->val << endl;
    cout << head->next->val << endl;
    cout << head->next->next->val << endl;
}
```

Advantages and disadvantages compared to arrays

The main advantage of a linked list is that you can add and remove elements at any position in $O(1)$. The caveat is that you need to have a reference to a node at the position in which you want to perform addition/removal, otherwise the operation is $O(n)$, because you will need to iterate starting from the `head` until you get to the desired position.

The main disadvantage of a linked list is that there is no random access. If you have a large linked list and want to access the 150,000th element, then there usually isn't a better way than to start at the head and iterate 150,000 times. So while an array has $O(1)$ indexing, a linked list could require $O(n)$ to access an element at a given position.

Linked lists have the advantage of not having fixed sizes. While dynamic arrays can be resized, under the hood they still are allocated at a fixed size - it;s just that when this size is exceeded, the array is resized, which is expensive. Linked lists don't suffer from this. However, linked lists have more overhead than arrays - every element needs to have extra storage for the pointers. If you are only storing small items like booleans or characters, then you may be more than doubling the space needed.

Mechanics of a linked list

When you assign a pointer to an existing list node, the pointer refers to the object in memory. Let's say you have a node `head`:

Python3

```
ptr = head
head = head.next
head = None
```

JavaScript

```
let ptr = head;
head = head.next;
head = null;
```

Java

```
ListNode ptr = head;
head = head.next;
head = null;
```

C++

```
ListNode* ptr = head;
head = head->next;
head = nullptr;
```

A language like C++ has explicit pointers, indicated by the asterisk `*`. In languages without explicit pointers, all non-primitive variables (like custom class objects) are treated as pointers.

After these lines of code, `ptr` still refers to the original `head` node, even though the `head` variable changed. `ptr = something` is the only way to modify `ptr`.

Iterating forward through a linked list can be done with a simple loop. To sum the values of a linked list:

Python3

```
def get_sum(head):
    ans = 0
    while head:
        ans += head.val
        head = head.next
```

```
    return ans
```

JavaScript

```
let getSum = head => {
    let ans = 0;
    while (head) {
        ans += head.val;
        head = head.next;
    }

    return ans;
}
```

Java

```
int getSum(ListNode head) {
    int ans = 0;
    while (head != null) {
        ans += head.val;
        head = head.next;
    }

    return ans;
}
```

C++

```
int getSum(ListNode* head) {
    int ans = 0;
    while (head != nullptr) {
        ans += head->val;
        head = head->next;
    }

    return ans;
}
```

Moving to `head.next` is the equivalent of iterating to the next element in an array. Traversal can also be done recursively:

Python3

```
def get_sum(head):
    if not head:
        return 0

    return head.val + get_sum(head.next)
```

JavaScript

```
let getSum = head => {
    if (!head) {
        return 0;
    }

    return head.val + getSum(head.next);
}
```

Java

```
int getSum(ListNode head) {
    if (head == null) {
        return 0;
    }

    return head.val + getSum(head.next);
}
```

C++

```
int getSum(ListNode* head) {
    if (head == nullptr) {
        return 0;
    }
```

```
    return head->val + getSum(head->next);
}
```

Types of linked lists

Singly linked list

Let's say you want to add an element to a linked list so that it becomes the element at position `i`. To do this, you need to have a pointer to the element currently at position `i - 1`. The next element (currently at position `i`), call it `x` will be pushed to be the element at position `i + 1` after the insertion. This means that `x` should become the `next` node to the one being added, and the node being added should become the `next` node to the one currently at `i - 1`:

Python3

```
class ListNode:
    def __init__(self, val):
        self.val = val
        self.next = None

# Let prev_node be the node at position i - 1
def add_node(prev_node, node_to_add):
    node_to_add.next = prev_node.next
    prev_node.next = node_to_add
```

JavaScript

```
class ListNode {
    constructor(val) {
        this.val = val;
        this.next = null;
    }
}

// Let prevNode be the node at position i - 1
let addNode = (prevNode, nodeToAdd) => {
    nodeToAdd.next = prevNode.next;
```

```
    prevNode.next = nodeToAdd;
}
```

Java

```
class ListNode {
    int val;
    ListNode next;
    ListNode (int val) {
        this.val = val;
    }
}

// Let prevNode be the node at position i - 1
void addNode(ListNode prevNode, ListNode nodeToAdd) {
    nodeToAdd.next = prevNode.next;
    prevNode.next = nodeToAdd;
}
```

C++

```
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int val) : val(val), next(nullptr) {}
};

// Let prevNode be the node at position i - 1
void addNode(ListNode* prevNode, ListNode* nodeToAdd) {
    nodeToAdd->next = prevNode->next;
    prevNode->next = nodeToAdd;
}
```

Note: it is unusual that you will have a pointer to the node at the position before where you want to perform an operation; we wrot these functions as a demonstration. Typically you will be doing these operations on the fly, as you iterate through the list. If you don't have a pointer to the desired position at all, you will need to iterate from the `head` until you are at the desired position, which means the operation would be $O(n)$. If you have a pointer already, it's $O(1)$.

Let's say you want to delete the element at position `i`. Again, you need to have a pointer to the element currently at position `i - 1`. The element at position `i + 1` , call it `x`, will be shifted over to be at position `i` after the deletion. Therefore, you should set `x` as the `next` node to the element currently at position `i - 1`:

Python3

```
class ListNode:
    def __init__(self, val):
        self.val = val
        self.next = None

# Let prev_node be the node at position i - 1
def delete_node(prev_node):
    prev_node.next = prev_node.next.next
```

JavaScript

```
class ListNode {
    constructor(val) {
        this.val = val;
        this.next = null;
    }
}

// Let prevNode be the node at position i - 1
let deleteNode = prevNode => {
    prevNode.next = prevNode.next.next;
}
```

Java

```
class ListNode {
    int val;
    ListNode next;
    ListNode (int val) {
        this.val = val;
    }
}
```

```
// Let prevNode be the node at position i - 1
void deleteNode(ListNode prevNode) {
    prevNode.next = prevNode.next.next;
}
```

C++

```
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int val) : val(val), next(nullptr) {}
};

// Let prevNode be the node at position i - 1
void deleteNode(ListNode* prevNode) {
    prevNode->next = prevNode->next->next;
}
```

`prevNode.next` is the node being deleted. `prevNode.next.next` is the node that should be kept. We change the `next` pointer of `prevNode` to point at that node instead of the one being deleted.

Because the node being deleted could only have been reached from `prevNode` and we have now severed that connection, it is no longer part of the list.

Note we are ignoring freeing the memory in C++ which is important if each node is dynamically allocated.

Deleting has the same runtime as insertion.

Doubly linked list

In a singly linked list, we needed a reference to the node at `i - 1` if we wanted to add or remove at `i`. This is because we needed to perform operations on the `prevNode`. With a doubly linked list, we only need a reference to the node at `i`. This is because we can simply reference the `prev` pointer of that node to get the node at `i - 1`, and then do the exact same operations as above.

Implementation:

Python3

```
class ListNode:
    def __init__(self, val):
        self.val = val
        self.next = None
        self.prev = None

# Let node be the node at position i
def add_node(node, node_to_add):
    prev_node = node.prev
    node_to_add.next = node
    node_to_add.prev = prev_node
    prev_node.next = node_to_add
    node.prev = node_to_add

# Let node be the node at position i
def delete_node(node):
    prev_node = node.prev
    next_node = node.next
    prev_node.next = next_node
    next_node.prev = prev_node
```

JavaScript

```
class ListNode {
    constructor(val) {
        this.val = val;
        this.next = null;
        this.prev = null;
    }
}

let addNode = (node, nodeToAdd) => {
    let prevNode = node.prev;
    nodeToAdd.next = node;
    nodeToAdd.prev = prevNode;
    prevNode.next = nodeToAdd;
```

```
    node.prev = nodeToAdd;
}

let deleteNode = node => {
    let prevNode = node.prev;
    let nextNode = node.next;
    prevNode.next = nextNode;
    nextNode.prev = prevNode;
}
```

Java

```
class ListNode {
    int val;
    ListNode next;
    ListNode prev;
    ListNode (int val) {
        this.val = val;
    }
}

void addNode(ListNode node, ListNode nodeToAdd) {
    ListNode prevNode = node.prev;
    nodeToAdd.next = node;
    nodeToAdd.prev = prevNode;
    prevNode.next = nodeToAdd;
    node.prev = nodeToAdd;
}

void deleteNode(ListNode node) {
    ListNode prevNode = node.prev;
    ListNode nextNode = node.next;
    prevNode.next = nextNode;
    nextNode.prev = prevNode;
}
```

C++

```
struct ListNode {
    int val;
    ListNode *next;
    ListNode *prev;
    ListNode(int val) : val(val), next(nullptr), prev(nullptr) {}
};

// Let node be the node at position i
void addNode(ListNode* node, ListNode* nodeToAdd) {
    ListNode* prevNode = node->prev;
    nodeToAdd->next = node;
    nodeToAdd->prev = prevNode;
    prevNode->next = nodeToAdd;
    node->prev = nodeToAdd;
}

void deleteNode(ListNode* node) {
    ListNode* prevNode = node->prev;
    ListNode* nextNode = node->next;
    prevNode->next = nextNode;
    nextNode->prev = prevNode;
}
```

Linked lists with sentinel nodes

We call the start of a linked list the `head` and the end of a linked list the `tail`.

Sentinel nodes sit at the start and end of linked lists and are used to make operations and the code needed to execute those operations cleaner. Even when there are no nodes in the linked list, you still keep pointers to a `head` and `tail`. The real head of the linked list is `head.next` and the real tail is `tail.prev`. The sentinel nodes themselves are not part of our linked list.

The previous code we looked at is prone to errors. For example, if we are trying to delete the last node in the list, then `nextNode` will be `null`, and trying to access `nextNode.next` would result in an error. With sentinel nodes, we don't need to worry about this scenario because the last node's `next` points to the sentinel tail.

Sentinel nodes also allow us to easily add and remove from the front or back of the linked list. Recall that addition and removal is only $O(1)$ if we have a reference to the node at the position we are performing the operation on. With a sentinel tail node, we can perform operations at the end of the list in $O(1)$.

Python3

```
class ListNode:
    def __init__(self, val):
        self.val = val
        self.next = None
        self.prev = None

def add_to_end(node_to_add):
    node_to_add.next = tail
    node_to_add.prev = tail.prev
    tail.prev.next = node_to_add
    tail.prev = node_to_add

def remove_from_end():
    if head.next == tail:
        return

    node_to_remove = tail.prev
    node_to_remove.prev.next = tail
    tail.prev = node_to_remove.prev

def add_to_start(node_to_add):
    node_to_add.prev = head
    node_to_add.next = head.next
    head.next.prev = node_to_add
    head.next = node_to_add

def remove_from_start():
    if head.next == tail:
        return

    node_to_remove = head.next
    node_to_remove.next.prev = head
    head.next = node_to_remove.next

head = ListNode(None)
tail = ListNode(None)
```

```
head.next = tail
tail.prev = head
```

JavaScript

```
class ListNode {
    constructor(val) {
        this.val = val;
        this.next = null;
        this.prev = null;
    }
}

let addToEnd = nodeToAdd => {
    nodeToAdd.next = tail;
    nodeToAdd.prev = tail.prev;
    tail.prev.next = nodeToAdd;
    tail.prev = nodeToAdd;
}

let removeFromEnd = () => {
    if (head.next == tail) {
        return;
    }

    let nodeToRemove = tail.prev;
    nodeToRemove.prev.next = tail;
    tail.prev = nodeToRemove.prev;
}

let addToStart = nodeToAdd => {
    nodeToAdd.prev = head;
    nodeToAdd.next = head.next;
    head.next.prev = nodeToAdd;
    head.next = nodeToAdd;
}

let removeFromStart = () => {
    if (head.next == tail) {
        return;
```

```
    }

    let nodeToRemove = head.next;
    nodeToRemove.next.prev = head;
    head.next = nodeToRemove.next;
}

let head = new ListNode(-1);
let tail = new ListNode(-1);
head.next = tail;
tail.prev = head;
```

Java

```
class ListNode {
    int val;
    ListNode next;
    ListNode prev;
    ListNode (int val) {
        this.val = val;
    }
}

void addToEnd(ListNode nodeToAdd) {
    nodeToAdd.next = tail;
    nodeToAdd.prev = tail.prev;
    tail.prev.next = nodeToAdd;
    tail.prev = nodeToAdd;
}

void removeFromEnd() {
    if (head.next == tail) {
        return;
    }

    ListNode nodeToRemove = tail.prev;
    nodeToRemove.prev.next = tail;
    tail.prev = nodeToRemove.prev;
}
```

```
void addToStart(ListNode nodeToAdd) {
    nodeToAdd.prev = head;
    nodeToAdd.next = head.next;
    head.next.prev = nodeToAdd;
    head.next = nodeToAdd;
}

void removeFromStart() {
    if (head.next == tail) {
        return;
    }

    ListNode nodeToRemove = head.next;
    nodeToRemove.next.prev = head;
    head.next = nodeToRemove.next;
}

ListNode head = new ListNode(-1);
ListNode tail = new ListNode(-1);
head.next = tail;
tail.prev = head;
```

C++

```
struct ListNode {
    int val;
    ListNode *next;
    ListNode *prev;
    ListNode(int val) : val(val), next(nullptr), prev(nullptr) {}
};

void addToEnd(ListNode* nodeToAdd) {
    nodeToAdd->next = tail;
    nodeToAdd->prev = tail->prev;
    tail->prev->next = nodeToAdd;
    tail->prev = nodeToAdd;
}

void removeFromEnd() {
    if (head->next == tail) {
```

```
        return;
    }

    ListNode* nodeToRemove = tail->prev;
    nodeToRemove->prev->next = tail;
    tail->prev = nodeToRemove->prev;
}

void addToStart(ListNode* nodeToAdd) {
    nodeToAdd->prev = head;
    nodeToAdd->next = head->next;
    head->next->prev = nodeToAdd;
    head->next = nodeToAdd;
}

void removeFromStart() {
    if (head->next == tail) {
        return;
    }

    ListNode* nodeToRemove = head->next;
    nodeToRemove->next->prev = head;
    head->next = nodeToRemove->next;
}

ListNode* head = new ListNode(-1);
ListNode* tail = new ListNode(-1);
head->next = tail;
tail->prev = head;
```

Dummy pointers

We usually want to keep a reference to the `head` to ensure we can always access any element. Sometimes, it's better to traverse using a "dummy" pointer and to keep `head` at the head.

Python3

```
def get_sum(head):
    ans = 0
    dummy = head
    while dummy:
        ans += dummy.val
        dummy = dummy.next

    # same as before, but we still have a pointer at the head
    return ans
```

JavaScript

```
let getSum = head => {
    let ans = 0;
    let dummy = head;
    while (dummy) {
        ans += dummy.val;
        dummy = dummy.next;
    }
    // same as before, but we still have a pointer at the head
    return ans;
}
```

Java

```
int getSum(ListNode head) {
    int ans = 0;
    ListNode dummy = head;
    while (dummy != null) {
        ans += dummy.val;
        dummy = dummy.next;
    }

    // same as before, but we still have a pointer at the head
    return ans;
}
```

C++

```
int getSum(ListNode* head) {
    int ans = 0;
    ListNode* dummy = head;
    while (dummy != nullptr) {
        ans += dummy->val;
        dummy = dummy->next;
    }

    // same as before, but we still have a pointer at the head
    return ans;
}
```

Using the `dummy` pointer allows us to traverse the linkeed list without losing a reference to the `head`.

It's important to build an intuition behind how to correctly re-assign pointers and the order in which the re-assignments should happen.