# Linked Lists

---

- Representing sequences as elements linked together
- Implementing Stacks with Linked Lists

## Why Was the Queue Implementation So Complex?

- We were using the list provided by Python, which reserves a block of memory, and then manages the memory and provides access.
- To make it efficient for what we wanted to do, we had to take control of the memory management.

### Other Options

- We create our own array and manage the memory ourselves.
    - This will have the same problems. Anything we write would have to be just as complex as using Python's list structure.
- Free storage of list elements

## Free Storage of List Elements

We will store each reference (lists are normally sequences of references to items, where each item is stored individually by Python) individually. We won't require them to be stored in consecutive memory.

We'll let Python decide where to put them, but with each reference, we'll also store a reference to the next reference. The last reference will be stored with a reference to `None`, to mark it as the last reference.

With this implementation we now have no space management issues. As long as there is any memory left for Python to use anywhere, we can create a new element objects, and a new 'list node' object.

However, we have to do the work to read the element. We can't rely on efficient list lookup anymore.

## Singly-Linked Lists

## Class Representation

To represent the double blocks that hold the reference to an item and the reference to the next

reference, we create a class `SLLNode` (standing for "Singly-Linked List Node"), which will contain `item` and `next`, where `item` is any object, and `next` is another `SLLNode` object.

To represent the whole list, we create an `SLL` class. This has two pieces of data:

- `first` – an `SLLNode`
- `length` – the number of items

# Python Skeleton

```
class SLLNode:
    def __init__(self, item, nextnode):
        self.element = item
        self.next = nextnode

class SLinkedList:
    def __init__(self):
        self.first = None
        self.size = 0

    def add_first(self, item):

    def add_last(self, item):

    def get_first(self, item):

    def get_last(self, item):

    def remove_first(self):

    def remove_last(self):

    def is_empty(self):
```

It's good to work out what the methods look like (and create a test block) before you write the code for the methods. Then you can do the testing as you go, rather than debugging at the end.

# 3 Methods: `add_first`, `get_first`, and `remove_first`

### `add_first`

- Point the `nextnode` of the new element to the current first element of the list.
- Update `first` to point to the new first element.

### `get_first`

- Go to the first node and read the element from it.

`remove_first`

- [check]

### Code

```
def add_first(self, element):
    node = SSLNode(element, self.first)
    self.first = node
    self.size = self.size + 1

def get_first(self):
    if self.size == 0:
        return None
    return self.first.element

def remove_first(self):
    if self.size == 0:
        return None
    item = self.first.element
    self.first = self.first.next
    self.size = self.size - 1
    return item
```

### Complexity

There're no inbuilt functions used, so there's no hidden complexity. Also there are no loops. So these three methods are all O(1), which is good.

## 2 Methods: `add_last` and `get_last`

`add_last`

- Create a new SLLNode which points to the element and to None.
- Step through the list until the last element, and point it to the new last element.

`get_last`

- Step through the list to the last element and return its item.

### Code

```
def add_last(self, element):
    newnode = SLLNode(element, None)
    if self.first == None:
        self.first = newnode
    else:
        node = self.first
        while node.next:
            node = node.next
        node.next = newnode
    self.size += 1

def get_last(self):
    if self.size == 0:
        return None
    node = self.first
    while node.next:
        node = node.next
    return node.element
```

#### Complexity

Now that there's a loop, these are O(n), which is very high for what should be a basic operation.

## Exercise

- Implement a `remove_last()` method.

## Implementing a Stack with a LinkedList

We decide the front of the linked list should be the top of the stack, because the operations there are O(1).

| Stack method: | Linked List method |
|---|---|
| `push()` | `add_first()` |
| `pop()` | `get_first()` |
| [check] | [check] |

### Not Using a Linked List

The linked lists will get more complex as we try to implement other ADTs, so often the functionality will be done inside those ADTs without using a linked list class.

# Testing the LinkedList Method

Use all the test methods and other functions we wrote in the lab (I haven't done this yet! D:) to test this implementation. They should work without modification.

We are aiming to build a library of different implementations of different ADTs, so that we can call whichever is most appropriate for the situation at hand.

# Simplifying Linked List Operations

Other ADTs will want to access both ends of the list. We add a new pointer `last` that points to the last element, and now we no longer need to step through the list to access that side of the list. This solves the O(n) problem with the end of our linked list.

# Exercise

- Adapt the code to use a `last` reference.

# Implementing a Queue with a LinkedList

We want to match the efficiency of our previous queue implementation. We want to operate on both ends of the sequence.

We want to add an element to end of the queue, ideally without searching through the whole list to find the end (our previous `add_last` and `get_last` implementations were O(n), which is too slow).

We can do this by storing a reference to the last object in the list as well as to the first. This gives us O(1) for all our methods.

Our enqueue method now becomes:

```
def enqueue(self, element):
    self.body.add_last(element)
```

# Moving On

LinkedLists are a general data structure – we can use them for any sequential storage. What happens if we want to remove an item from the middle of the list?

To remove the node x, we need to link the node w to the node z, by setting w.next = z. We can find z from x.next, but how can we find w? We have to search through the list. Doubly-Linked Lists are a solution to this problem.

# Doubly-Linked Lists

Similar to singly-linked lists, but every node has a reference to the last node as well as the next one.

## Head & Tail

We add two dummy elements, one at the start and one at the end. These have no data, and serve to mark the start and end of the list. This way, every node that contains data has the same structure, and links to a next node and a previous node.

The head node points to the first node with its `next` reference, but points to `None` with its `previous` reference.

The tail node points to the last node with its `previous` reference, but points to `None` with its `next` reference.

## Code Skeleton

We will be asked to write this code for the first continuous assessment, which is upcoming. Here's a skeleton:

```
class DLLNode(object):
    def __init__(self, item, prevnode, nextnode)L
        self.element = item
        self.next = prevnode

class DLinkedList(object):
    def __init__(self):

    def add_after(self, item, before):

    def add_first(self, item):

    def add_last(self, item):

    def get_first(self):

    def get_last(self):

    def remove_node(self, node):

    def remove_first(self):

    def remove_last(self):

    [more here]
```

Implementing these functions is tricky – you need to be very careful that you don't create cycles, or dead ends in the middle of the list.

### `add_after()`

We add the new node, set its `next` and `previous` references to point to the right nodes, update the `next` and `previous` references to point to the new node. We also update the size.

- create a new node for item with `previous` and `next` referring to `None`
- `n1.next.prev = newnode`
- `newnode.next = n1.next`
- `newnode.prev = n1`
- `n1.next = newnode`
- `size += 1`

### `add_first()`

We use `add_after()` on the head node.

```
self.add_after(item, self.head)
```

### `add_last()`

We use `add_after()` on the node before the tail node.

```
self.add_after(item, self.tail.prev)
```

### `remove_node()`

Removing node y from a list where x is before it and z is after it.

- `y.prev.next = y.next`
- `y.next.prev = y.prev`
- `y.prev = None`
- `y.next = None`
- `size -= 1`

We may also want to return the element and then set it to be `None`, so that node y will be removed instead of continuing to take up memory. Python will know that it's no longer usable since nothing is pointing to it and it is pointing to nothing except `None`.

## Exercises

Implement these methods:

- a method to swap two adjacent nodes
- a method to swap two arbitrary nodes

The simpler way to do these is to just swap the elements of the nodes. There are reasons you may not want to do this (if you're maintaining more complex data structures and have something else pointing to these nodes because you know what's in them and you want to be able to jump straight to them), so there's a second way. This way is more standard, but is messier. It consists of changing the `next` and `previous` fields of the four relevant items.

With the arbitrary nodes, it's difficult because they may also be adjacent, or adjacent in reverse order. The simplest approach is to only use your pre-existing methods `swap_adjacent`, `remove()` and `insert_after()`. If it's not going to add much more effort, it's worth using previously written functions because you've already tested them. You reduce the chance of making mistakes and spend less time writing your code.