The List

We'd like to extend our doubly-linked list to general list behaviour. We want to be able to:

- Inspect anywhere in the list by position
- Add anywhere in the list by position
- Replace an item anywhere in the list by position
- Remove an item anywhere in the list by position
- Iterate through the list
- Clear the list

We'll only look at pseudocode today as implementing this in Python will be part of the upcoming continuous assessment.

The List ADT

- `get(pos)` – return the item at position `pos`
- `replace(pos, item)` – replace the value at `pos` with `item`
- `add(pos, item)` – add the `item` after position `pos`
- `remove(pos)` – remove the node after position `pos`
- `clear()` – remove all elements

`get(pos)`

Linear search.

```
initialise a counter to 0
get the first node                          #node = getfirst()
while counter is not at pos
    get next node                           #node = node.next
    increment counter
return element at node
```

What Do We Do if `pos` is Not Valid?

Three options:

1. Do nothing. It's the responsibility of the code that calls our method to get it right.
    - This is the default approach in C.
2. Return special values (None, False, -1, …) – the calling code should check the return values to be safe.
    - We must be careful in our choice of return values. It must never be a value that could be validly returned from the method.

3. Throw an exception – the calling code must specify how to handle the exception.
    - Use exceptions only for truly exceptional events. (This viewpoint is debated.)

`replace(pos, item)`

Linear search until we find the item, replace it, and return it.

`add(pos, item)`

Linear search.

```
initialise a counter to 0
get the first node                       #node = getfirst()
while counter is not at pos
    get next node                        #node = node.next
    increment counter
add_after(item, node)
```

`remove(pos)`

```
initialise a counter to 0
get the first node                       #node = getfirst()
while counter is not at pos
    get next node                        #node = node.next
    increment counter
temp = remove_next(node)
```

`clear()`

Two ways:

- Update the pointers on head and tail, adjust the size to 0. Leaves a lot of data still in memory, that's hard for Python to realise should be deleted.
- Step from node to node, clearing everything as you go, then join head and tail together.

Here's the pseudocode for the second way:

```
initialise a counter to 0
get the first node                        #node = getfirst()
while counter is not at size
    node.prev = None
    node.element = None
    temp = node.next
    node.next = None
    node = temp
    counter += 1
tail.prev = head
head.next = tail
temp = remove_next(node)
```

When you're partway through this, you don't have a valid list.

Iterating Through the List

The `get(pos)` method is O(pos). If we use this to iterate through the list, our iteration is now O(n squared).

Instad, we're gonna maintain a `cursor` as an instance variable. We'll provide methods to allow the cursor to move one step forward or backward or jump to front or back. We'll also provide methods to get the element at the cursor, replace, add, or remove it.

But, there are no guarantees if nodes are moved by other methods.

List ADT (version 2)

- `get_current()` – return the element at the cursor
- `add_current(item)` – add a new element containing `item` after the cursor
- `replace_current(item)` – replace the element at the cursor with `item`
- `remove_current()` – remove the element at the cursor
- `next()` – move the cursor forward 1 place
- `prev()` – move the cursor backward 1 place
- `move_to_front()` – move the cursor to the first element
- `move_to_last()` – move the cursor to the last element
- `has_next()` – report whether or not there are more elements after the cursor

This now lets us have iteration at O(n).

List ADT (version 3)

In this version, we add extra helpful methods to version 2, as follows:

- `get_first()` – return the first element in the list
- `get_last()` – return the last element in the list
- `add_first(item)` – add a new element containing `item` to the start of the list

- `add_last(item)` – add a new element containing `item` to the end of the list
- `replace_first(item)` – replace the first element of the list with `item`
- `replace_last(item)` – replace the last element of the list with `item`
- `remove_first()` – remove the first element from the list
- `remove_last()` – remove the last element from the list
- `find(item)` – return the position of the first occurrence of `item`, if it is in the list

The methods here for the first and last elements are special cases, and are efficient in doubly-linked list implementations.

Notes

In some textbooks you will see a more general ADT called a Positional List. A position is then an abstract thing, which you can use to jump to a particular element in the list. The position stays with its element no matter where you move the element to in the list.

Iteration through the list is then done in a similar style to our cursor, but using Python iterable objects and an `iter()` method.

Array-Based Lists vs. Linked Lists

Array-Based Lists

A block of memory is reserved for the references. Direct access to any node using its index is O(1). (Because we know the size of each references and we know the location of the start of the list, so in RAM we can jump directly to any index in the list.)

This is how Python implements its list.

Linked Lists

We store each node in an arbitrary place in memory, but each node also has a reference to the next node in the list, and optionally a reference to the previous node in the list (SLLs and DLLs respectively).

We also defined head and tail nodes so that inserting nodes at the start and end is the same as inserting within the list. This avoids special cases.

Space is only reserved for size, head, and tail.

Access to individual items is by following links, so it's O(n) in the worst case.

Adding to Array-Based Lists (in Python)

If enough space is reserved:

- Adding at the end is immediate (O(1)).
- Adding in the middle requires shifting the remainder, so it's O(n).

If all the space is occupied:

- We need to reserve a larger space and copy the contents, so it's O(n).
- But if this is managed well, it is O(1) *on average*.

But, is O(1) on average good enough for real-time applications?

Adding to Linked Lists

- Adding an element at either end is O(1).
- Adding and element in the middle is O(1) if we have a reference to the position. Otherwise (e.g. if we want to add an item at the 300th position) it's O(n).

Deleting from Array-Based Lists (in Python)

- Deleting from the end is O(1) *on average*.
- Deleting from the middle is O(n).

Again, is O(1) on average good enough for real-time applications?

Deleting from Linked Lists

- Deleting from the start or the end is O(1).
- Deleting form the middle is O(1) if we have a reference to the position. Otherwise it's O(n).

Note

When the two types of list have the same complexity, operations on array-based lists are usually faster:

- Reserving space for each internal node (in a linked list) takes time.
- Updating linked list nodes takes 2 or 4 assignments.

Array-based lists may use less space (despite the extra space needed for growing the lists).

- With a list of 100 items, an array-based list may leave space for 200 references to leave space to grow, but a linked list will require space for 300 references to store a list of 100 items.
- Maintaining the list nodes requires 2 or 3 references per node.

Linked structures are useful because we may need them in other languages, which may not use references for everything as Python does.

Sorting an Array-Based List

- We'll look at sorting more in CS2516.

It's not very complicated with array-based lists.

Sorting in a Linked List

This is more complicated, as you have to manage the six references.

- Linked lists do not have direct access to arbitrary positions.
- This will be covered in more detail in CS2516.

Searching an Array-Based List

The simplest technique is a linear search.

- Worst case, with an unordered list, linear search is O(n).

If the list is ordered we can use binary search, which is O(log(n)). This is possible because we can jump to an arbitrary position in the list.

Searching a Linked List

If it's not ordered, we have to do a linear search, which is O(n).

If it is ordered, searching may still be O(n) because we can't jump to an arbitrary node. We have to instead cycle through the list.

Linked Structures for Efficient Searching?

A sorted (array-based) list with 24 elements can be searched in no more than 5 steps.

- Look at cell 12.
- Look at either cell 6 or 18.
- Look at either cell 3, 9, 15, or 21.
- …

If we represent this as a linked tree, where each item points to the next items you would check in a binary search. So 12 would be the root node, and its children would be nodes 6 and 18. Node 6's children would be nodes 3 and 9.

Now we can implement binary search with a linked structure.

A singly-linked tree only allows you to go from parent nodes to children nodes, while a doubly-linked tree allows you to go from children nodes back to parent nodes as well.

Adding nodes will require the tree to be rebalanced to preserve the searchable structure.