Array List

Memory Requirements

All elements are stored in contiguous memory. The array May allocate extra space to amortize the cost of resizing. The memory for each element isjust the size of the element plus a small constant overhead.

An array list will have fast random access (O(1) for get and erase). This would be the way to go if you need fast random access. It wil however have slow insertions/deletions in the middle (O(n) due to shifting). The worst case for resizing is O(n), but the average case over many operations is O(1) per insertion. The arrays will often allocate more memory than is needed, making this a bad choice if memory overhead is a concern.

Efficiency Analysis

- insert(item, position) - O(n): In the worst case, all elements after position must be shifted right.
- get(position) - O(1): The position is stored in the array giving direct access.
- find(item) - O(n): May need to check every element in worst case.
- remove(item) - O(n): Finding the item is O(n), and removing it may require shifting elements.
- print() - O(n): Must access each element.
- length() - O(1): The size is stored as a member variable.
- count(item) - O(n): Must scan the entire list to count all occurrences of item. Only a counter variable is needed.
- remove_duplicates() - $O(n^2)$: In the worst case, every element is compared to every other element, which means the nested loops will run n*n times.
- reverse() - O(n): Each element is read once. A stack size n is used to temporarily store list values.
- append(&list) - O(n): Because the size is stored, this will run n times where n is the length of the added lists. If the current array isn't big enough for both lists, resizing may be required (O(n) to copy elements to a new array).

Singly Linked List

Memory Requirements

Each element is stored in a node, which contains the data and a pointer to the next node. The linked list does not use contiguous memory. Each node requires extra memory for the pointer(s), but no extra memory is needed for future elements. Each node requires extra memory for a pointer.

The Linked List will have slow random access (O(n) for get and insert/remove in the middle) but fast insertions/deletions at the head (O(1)). This would be a good option if continuous memory or resizing is too expensive. It is also good if you don't need fast random access.

Efficiency Analysis

- insert(item, position) - O(n): Must traverse from the head to position to insert, worst case would be the entire list.
- get(position) - O(n): Must traverse from the head to position.
- find(item) - O(n): Must traverse from the head to position.
- remove(item) - O(n): Must traverse to find the item, then update pointers.
- print() - O(n): Must access each node.
- length() - O(1): Size is stored.
- count(item)- O(n): Must traverse the entire list to count all occurrences of item. Only a counter variable is needed.
- remove_duplicates() - $O(n^2)$: For each node, traverse all previous nodes to check for duplicates.
- reverse() - O(n): Each node is visited once to reverse the pointers. A stack is used to store data in reverse order.
- append(&list) - O(n): Traverse to the end of the current list, then append each node of list.