

Arrays and Lists

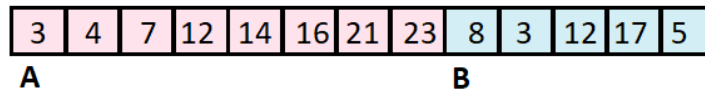
Arrays and Linked Lists both implement the **List** Abstract Data Type (**ADT**).

A List is any collection of things that have an ordering (i.e. one item comes after the other).

Lists and arrays can also represent items that do not have an ordering, this can improve the runtime of insert and delete for Arrays.

Arrays are stored contiguously in memory

```
int A[8] = {3, 4, 7, 12, 14, 16, 21, 23};  
int B[5] = {8, 3, 12, 17, 5};
```



Note: A is a sorted array B is unsorted

Linked Lists use pointers describing how one element connects to the next.

There are several varieties of Linked Lists:

Singly Linked Lists have a pointer to the next element and a NULL trailing

Doubly Linked Lists have a pointer to the next and previous element and a NULL leading and trailing pointer

Singly Circular Linked Lists have a pointer to the next element with trailing pointer wrapping around to the first element

Singly Linked Lists have a pointer to the next element and previous elements with leading and trailing pointers wrapping around to the last and first elements respectively.

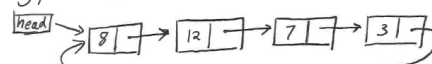
Singly Linked List



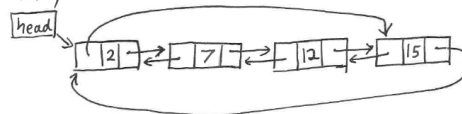
Doubly Linked List



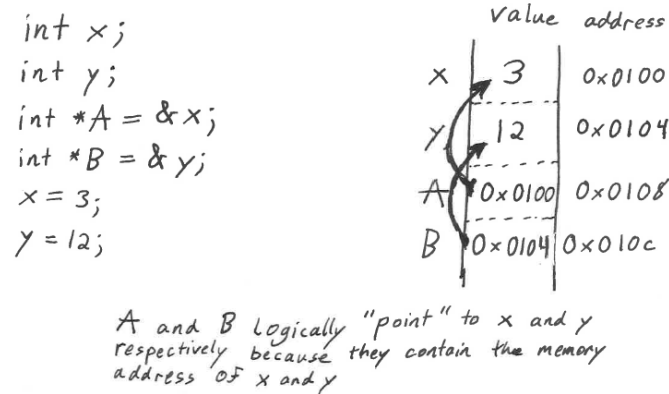
Singly Circular Linked List



Doubly Circular Linked List

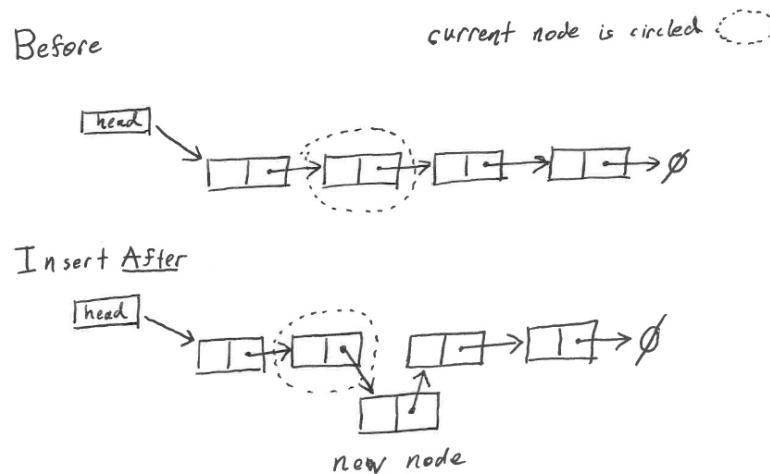


A **pointer** is actually a **memory address** every variable in address in **RAM** is in a separate address.

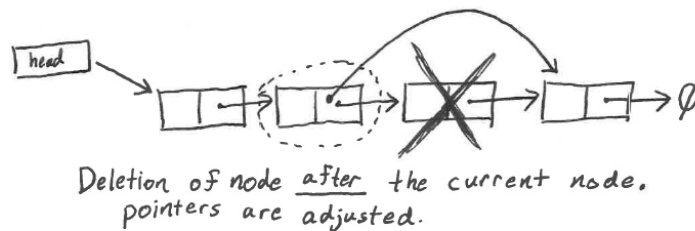


Linked Lists

Insertion into a linked list requires creating a new pointer and adjusting the pointers for insertion. This can be done in **constant time** or $O(1)$ given a pointer to the **previous** node.



Deletion from a linked list is also done in **constant time** $O(1)$ given a pointer to the **previous** node.



To **Find** an element in a Linked List requires traversal of all of the elements before that element occurs. Given K elements before the first element, the number of operations is $O(K)$.

In the **best case** we're lucky and it's the first element so **best case** $K=1$

In the **worst case** it's the last element so **worst case** $K=N$

In the **average case** it's in the middle, so **average case** $K = N/2$

Find

best case $O(1)$

worst case $O(N)$

average case $O(N)$



Best Case Find (7)

Worst Case Find (13) OR Find (666)

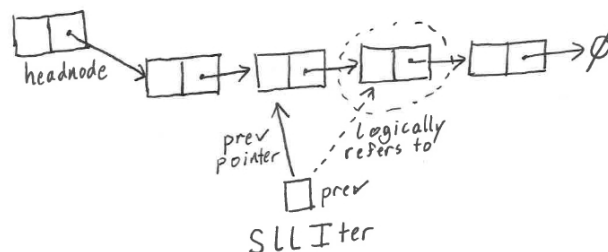
Average Case Find (5)

We may traverse a linked list using an **iterator**. In some simple cases, and iterator may be just a simple pointer, but in some cases it may be more complicated.

Many operations we looked at require a pointer to the previous node, so in the sample code, where the iterator has a physical pointer to the previous node but logically refers to the current node. However, the details of how to implement the iterator vary from implementation to implementation as well as whether the linked list is singly or doubly linked, circular or non-circular.

This implementation allows SinglyLinkedList class to delete the current node (by using a pointer to the previous), or insert values before the current node.

Singly Linked List class



To **Access** node at index K is also $O(K)$. This could be as fast as $O(1)$ in the **best case** if we are extraordinarily lucky and we're accessing the first element ($K=1$).

In the **worst case** it could be as slow as $O(N)$ if we Access the element in the last position ($K=N$). On average we must traverse $N/2$ elements to access an element in the linked list thus the **average case** is $O(N)$.

Access

best case $O(1)$

worst case $O(N)$

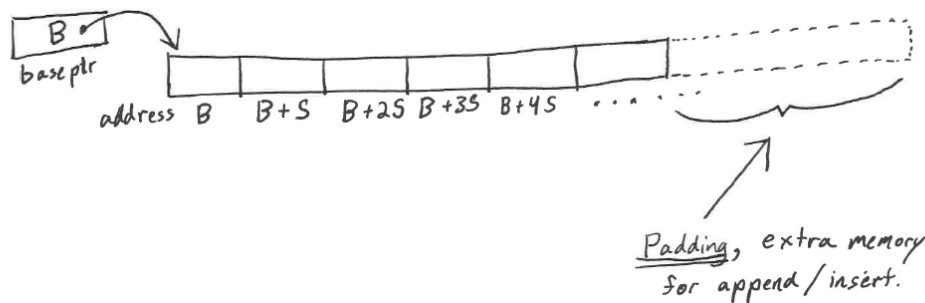
average case $O(N)$

Arrays

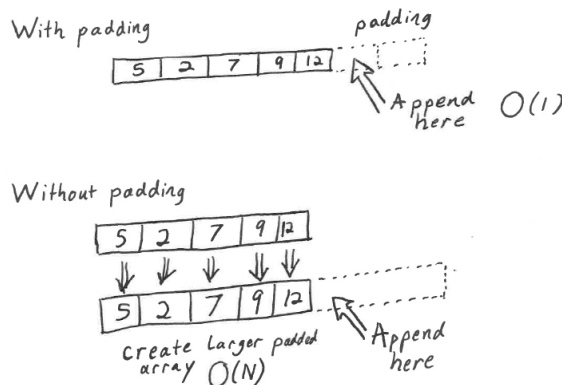
For Arrays, the time it takes to **Access** a node in an array is always $O(1)$.

Arrays utilize a pointer to the first element in memory, and access in RAM is performed by multiplication of the base address with the size of the element.

Arrays often additionally contain additional **padding** at the end of the array, this can greatly improve the amount of time it takes to **Append** an element.



To **Append** an element in an array if padding is available can be done in constant time $O(1)$. However, if no padding is available, **Append** requires us to create a larger buffer and copy all of the elements to a larger buffer. This takes $O(1)$ time.



Given **Padding**, If the data **has an ordering**, to **Insert** and **Remove** takes $O(K)$ time where K is the number of elements after the index we are inserting. This is because Insert and Remove must move all of the elements down one position.

Insert / Remove for **ordered** arrays

best case $O(1)$

worst case $O(N)$

average case $O(N)$

If the data is **unordered** then removal is constant time $O(1)$, just overwrite the element to be removed with the last element and decrease size.

Given **padding** and **unordered** data, **Insert** is also $O(1)$, it is sufficient to append to end because we do not care of the ordering of unordered data, so we can insert at the end.

