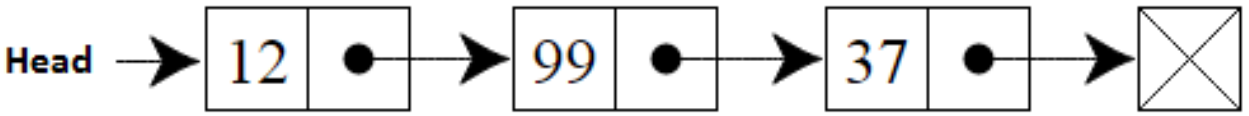
List

List

- Linked list is a commonly used data structure.
- You can think of it as a bunch of nodes in a chain.

- Each node contains a value (or some data), and knows how to get to its neighbor(s).
  - If a node knows to get to only one of its neighbors (next one), then it's a *singly* linked list.
  - If a node knows to get to both its neighbors (next and previous), then it's a *doubly* linked list.

- Here's a singly linked list.
  - Three nodes,
  - Each node has an integer value



https://en.wikipedia.org/wiki/Linked_list

Linked list uses nodes to

- Hold the data
- Hold a reference to the next node(s).
    - Singly linked list will have one reference to the next node.
    - Doubly linked list will need to hold two references, to the next node and previous node.

Memory address 0x100 is where MyClass object happened to be allocated

Node n = new Node();

n.value = new MyClass();

Singly linked list:

```
Node
{
   MyClass  value                  // data stored in the node. This example shows an integer, but can be any type
   Node  next     // reference to the next node, null for the last node
}
```

Doubly linked list:

```
Node
{
   integer  value                  // data stored in the node. This example shows an integer, but can be any type
   Node  next     // reference to the next node, null for the last node
   Node  previous                  // reference to the previous node, null for the first/head node
}
```

Singly linked list

- A singly linked list can traverse only in one direction, as you would expect, since a node only knows about its next neighbor.

- The *head* is the beginning of the list.
  - If it is null, it means the list is empty.

- So, if you start at the head node, and then keep going to the next node until you reach the last node, you would traverse the whole list.
  - Last node will contain *null* for its next node reference.
  - That's how you know you are at the last node of the list.

Traversing a list:

```
Node node = head;
while (node is not null)
    node = node.next;
```

← *Keep going until you find a node that is null. If head was null, then control will not go in here.*

Adding a Node

- To append a node to a list, you do the following:
    - Check if *head* node is null
    - If yes:
        - Allocate a new node and make it the head node.
            - Like: head = new Node();
    - If no:
        - Allocate a new node and add it to the end.

```
void  Append( int value)
{
   Node newNode = new Node;
   newNode.value = value;
   newNode.next = null;

   if (head is null)
            head = newNode;
  else
   {
            Node node = head;

            // Find the end of the list
            while (node.next is not null)
               node = node.next;

            // Now add to end of list
            node.next = newNode;          ← node.next was null, but now points to newNode.  newNode.next is null
   }
}
```

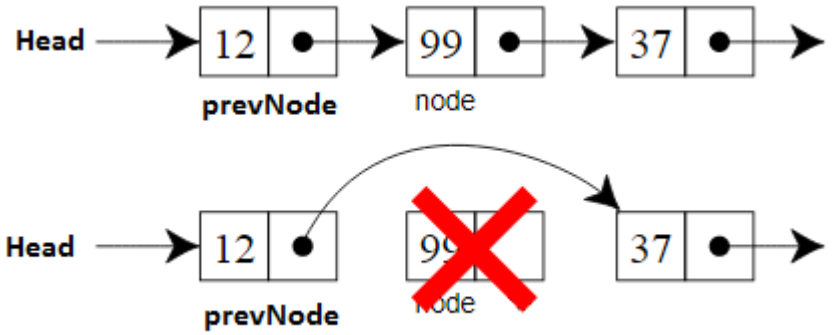
Adding a Node

- Now, to prevent the traversal to the end every time a node has to be added, we can keep track of the tail node, just like we kept track of the head.
  - We will do this as a lab after a few slides.

- What is the Big O time complexity for traversing to the end of the list?
  - Time complexity?
  - Space complexity?

Deleting a Node

- To remove a value from a list:
  - Traverse the list, looking for the value.
  - While traversing, keep track of the previous node.
    - This is needed because when you remove a node, you need to update that node's previous node's next reference to point to that node's next node reference ☺
      - The diagram below should help understand.



Circular list

- Linked list:
  - Last node’s next node points to a null, signifying it’s the last node.

- Circular list:
  - Last node’s next node points to first node.
  - This means none of the nodes point to a null, unless the list is empty.

Usage

- Circular lists can be used in applications where we want to keep rotating among the elements in a list without coming to an end.
  - Examples:
    - A game with N players:
      - you keep going from one player to next in a sequence.
    - Operating system scheduling:
      - Circulate among jobs that need scheduling on CPU cores.

Double linked list

- A double linked list is where each node has two references:
  - next node
  - previous node

- In a double linked list:
  - The previous node of the head node points to null.
  - The next node of the tail node (last node) points to null.

- In a circular double linked list:
  - Previous node of head points to tail node.
  - Next node of the tail points to head node.

- Linked lists can be used to implement the following data structures:
    - Stack
    - Queue

Big O for lists

Linked list operation	Time complexity
Access	O(n)
Search	O(n)
Insertion (known location)	O(1)
Deletion (known location)	O(1)
Insertion (needing traversal to find location)	O(n)
Deletion (needing traversal to find location)	O(n)

Some questions

- Lets say you need to find the length of two singly linked lists.
  - For the first list, you are given the head and tail pointers.
  - For the second list, you are given only the head pointer.
  - Is finding the length of one of these any easier than the other?
    - If yes, why?
    - If no, why?

- Is it possible to find the length of a singly linked list when given only the tail pointer?
- Is it possible to find the length of a double linked list when given only the tail pointer?

- Is it possible to find the length of a circular singly linked list when given only the tail pointer?

LAB

- Write a function to return the length of a linked list.
  - int Length(Node head)
- Write the length function for a circular linked list.
  - int LengthCircular(Node head)

- Write a function to append a node at the end of a list without having to traverse the whole list.

- Write a function to delete a node from a list.
  - void DeleteNode(Node head, int valueToDelete)

- Write a function that returns the N^{th} node from the end of the list.
  - Node GetNthNode(Node head, int N);

Qs on the LAB

- Write a function to return the length of a linked list.
  - int Length(Node head) ← Time and Space complexity ?
- Write a function to return the length of a circular linked list.
  - int LengthCircular(Node head) ← Time and Space complexity ?

- Write a function to append a node at the end of a list without having to traverse the whole list.
  - What would the function signature look like?
  - ← Time and Space complexity ?

- Write a function to delete first occurrence of a value from a list.
  - void DeleteNode(Node head, int valueToDelete) ← Time (*best* and *worst*) and Space complexity ?
  - What if this function had to delete ***all*** occurrences of a value from a list, what would the *best* and *worst* case time complexities be?

- Write a function that returns the N^{th} node from the end of the list.
  - Node GetNthNode(Node head, int N); ← Time and Space complexity ?

Array

- Array is typically a contiguous block of memory (unlike linked list).
- As a result of this, random access of arrays is possible.

  - What do we mean by random access?

  - It means that accessing any element in the array takes the same amount of time, which is unlike list.
  - Lets say we have an array with one million elements.
    - Accessing the first element of this array takes the same time as accessing the last.
    - This means Big O time complexity for array access is?
    - If you had to do the same two accesses in a linked list, then accessing the last element would mean traversing through the whole list.

Array Insertion

- Insertion in an array can be expensive.
- This is because an array is a contiguous block of memory, and inserting an element in a given location would mean that the elements to the right of that location would need to shift right by one. So,
  - If an element is inserted in the first position, all elements need to shift right by 1.
    - So, N elements move right by 1.
  - If an element is inserted in the middle, all elements in the 2nd half need to shift right by 1.
    - So, N / 2 elements move right by 1.
  - If an element is inserted at the end (appended), none of the elements need shifting.
    - Assuming there is space at the end.
    - This is the best case of insertion in an array, and it is O(1).

  - Average Big O for array insertion would be the case where half the elements shift right by 1.
    - So that is O (N/2), which really is O (N).

Array Deletion

- Array deletion can also be expensive, just like array insertion.

- You can apply the same logic as we talked about in case of array insertion, the only difference being that elements need to shift *left* by 1 in case of deletion.
- Everything else is the same as in array insertion, including the Big O complexity

Dynamic array

- Now, with arrays, you need to specify the size at compile time. This means you need to know your capacity / size needs when writing the program. This is not always the case.

- This is where dynamic arrays come in, and in these, the size can be specified at run time, and can vary depending on what input or scenario is being handled.

Dynamic array

- How do dynamic arrays work… how does their dynamic sizing work?
- Its quite simple:
- When an element is added to a dynamic array, one of following two scenarios happen:

    - Array is not full, so the element is added.

    - Array is full. In this case:
        - A bigger piece of memory is allocated (how big?)
        - The existing elements are copied from current memory to the new memory.
        - Old memory is released (not needed in managed runtimes like Java or C#).
        - The incoming element is added, since the array now has space.

Dynamic array

```
void Add( int value )
{
        if (currentSize >= array.Length)
                ResizeArray(currentSize + currentSize * 0.5);

        array[currentSize] = value;
        ++ currentSize;
}
void ResizeArray( int newSize )
{
        allocate new memory of size newSize
        Copy existing elements to this newly allocated chunk of memory
        Release existing memory (only for non-managed languages like C, C++)
}
```

List vs Array

Operation	Linked list time complexity	Array time complexity
Access	O(n)	O(1)
Search	O(n)	O(n)
Insertion (known location)	O(1)	O(n)
Deletion (known location)	O(1)	O(n)
Insertion (needing traversal to find location)	O(n)	O(n)
Deletion (needing traversal to find location)	O(n)	O(n)

Locality of reference

- One thing to keep in mind about arrays vs lists is the locality of reference.

- Since array is a contiguous block of memory, accessing one element will typically bring its neighboring elements also into the cache, thereby making access of those neighboring elements fast.
- This may not be the case with linked lists, because the nodes in a list are typically not next to each other in memory… when adding nodes to a list, the nodes are allocated at insertion time, and hence could be anywhere on the heap.
- So, arrays will have good spatial locality.