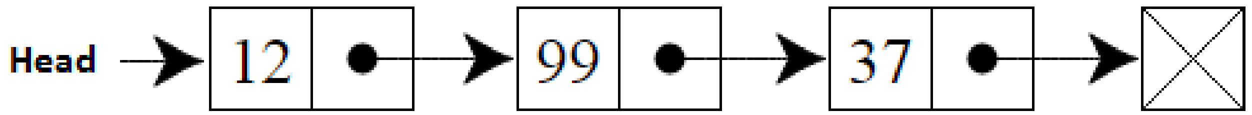
List

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, one to the *next* node and one to the *previous* node.

Singly 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
}
```

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.

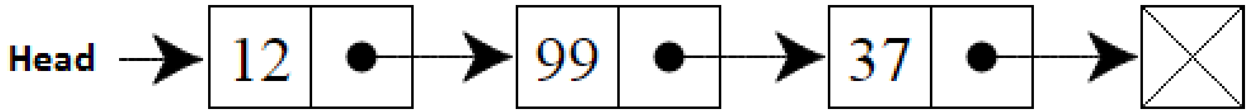
- So, if you start at the head node, and then keep going to the next node until you reach the last node, you would traverse the whole list.
  - Last node will contain *null* for its next node reference.
  - That’s how you know you are at the last node of the list.

Traversing a list:

```
Node node = head;

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

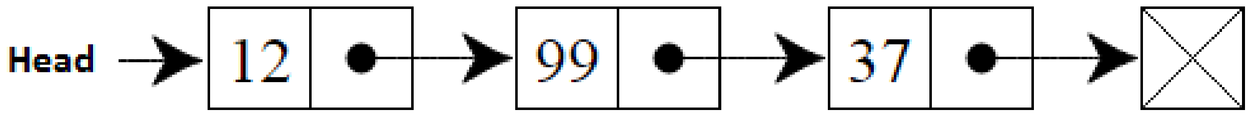
← Keep going until you find a node that is null. If head is null, then loop is not entered.



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 of the list.

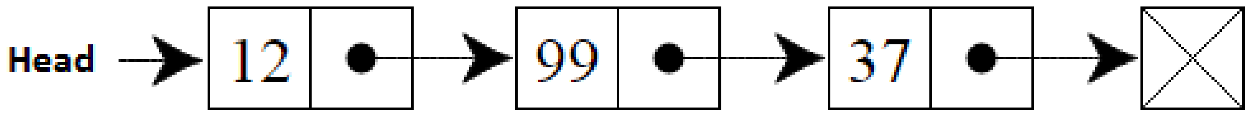



```
void  Append( int value)
{
   Node newNode = new Node;                                   //  First, allocate a node
   newNode.value = value;                    //  put the value in it.
   newNode.next = null;                      //  set its next field to null (since this new node will go to end of the list)

   if (head is null)                         // empty list
             head = newNode;                 //  this new dude is the first one in this list
  else
   {                                         // list has some pre-existing nodes, so we have some work to do

            Node lastNode = GetLastNode();        ← We will fill this on the next slide

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




```
void  Append( int value)
{
   Node newNode = new Node;                    //  First, allocate a node
   newNode.value = value;                      //  put the value in it.
   newNode.next = null;                        //  set its next field to null (since this new node will go to end of the list)

   if (head is null)                           // empty list
             head = newNode;                   //  this new dude is the first one in this list
  else
   {                                           // list has some pre-existing nodes, so we have some work to do
             Node lastNode = head;

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

             // lastNode now points to the end of the list.
             lastNode.next = newNode;           ← lastNode.next was null, but now points to newNode.
   }                                               newNode is the new last node, and  newNode.next is null
}
```

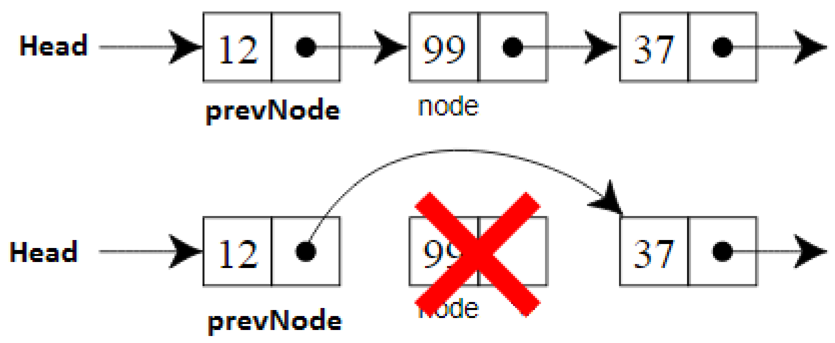
Adding a Node

- 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 complexity for traversing to the end of the list?
  - Time complexity?
  - Space complexity?

Deleting a Node

- To delete a value from a list:
  - Traverse the list, looking for the value.
  - While traversing, keep track of the previous node.
    - Previous node is needed because when you delete a node, you need to update the deleted node's previous node's next reference to point to deleted node's next node.
      - The diagram below should help understand.
    - 99 is the node to be deleted.
    - 12 is the previous node.
    - 37 is the deleted node's next node.



- deletedNode.previous.next = deletedNode.next

Circular list

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

- In a **circular** list:
    - Last node’s next node points to first node.
    - This means none of the nodes’ next field points to a null.
        - What if a circular list contains only one node? Where does its next point to?

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 (TBD: Add space complexity)

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?
  - Can compare the address of the nodes (reference). Don't need to look at value at all.

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 Kth node from the end of a singly linked list.
  - Node GetKthNode(Node head, int K);
  - Example: 1 2 3 4 5 6 7 8 9 10
  - For this list, 2nd from end is 9, 4th from the end is 7.

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

List access vs Array Access

- Lets say we write a function *GetElement,* that takes a linked list (can be singly linked or doubly linked), and also takes an integer index, and returns the element at that index.

- So, something like:
  - Node n = GetElement(list, ii);

- Now, we can write:
  - Node n1 = GetElement(list, 1); ← returns 1st element
    - OR
  - Node n10 = GetElement(list, 10); ← returns 10th element
    - OR
  - for (int ii = 0; ii < totalElements ; ++ ii)
    print GetElement(list, ii);

- Can we say that we implemented random access on a list?

  A. Yes
  B. No

Dynamic array

- Now, with built-in 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,
  - 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, this is what happens:
  - accessing one element typically brings neighboring elements also into cache,
  - access of those neighboring elements is now fast (since it’s a cache hit).

- This may not be the case with linked lists, because
  - nodes in a list are typically not next to each other in memory
    - when inserting in a list, nodes are allocated at insertion time (run time),
    - hence nodes could be anywhere on the heap (and will not be in contiguous memory).
    - Non – CS analogy: At a restaurant:
      - Group of 4 people arrive, are seated together (contiguous locations).
      - 4 people arrive (separately), are not going to be seated together, but wherever there are tables appropriate for a single person.

- So, arrays will have good **spatial** locality.
  - Binary searches in an array sort of work against the locality of reference advantage.
  - Look up **temporal** locality if interested.