LinkedLists

Mason Vail
Boise State University Computer Science

Interface - Implementation Separation

A list interface defines the operations and expected behavior of a list, but does not specify how a list implementation will manage list elements internally.

The user of a list does not need to know anything about the implementation to interact with a list through its interface. However, there is no single, “best” implementation of a list and the internal management will affect the efficiency of interface methods and memory use.

Avoiding shifting - more organic lists

Maintaining list elements in their corresponding indexed positions in arrays makes sense and allows fast lookups by index, but imposes O(n) shifting costs for insertions and deletions.

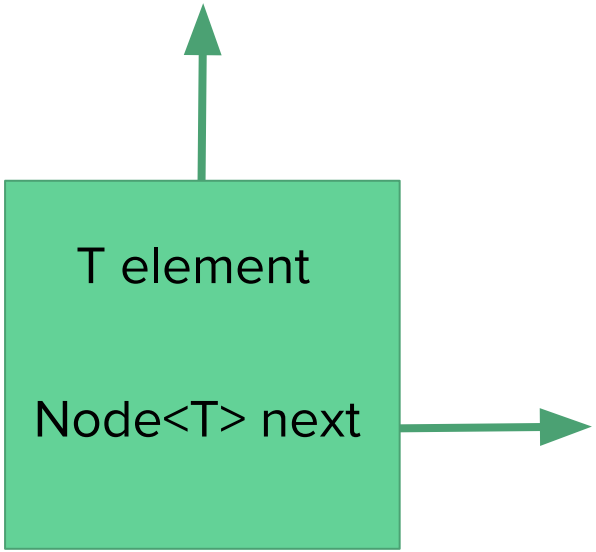
An alternative is to create an implementation that more closely matches the fluid, organic mental model of the interface, where a list is simply a sequence of elements and “space” between consecutive elements automatically expands or shrinks to accommodate list insertion or deletion modifications.

Nodes - building blocks of flexible structures

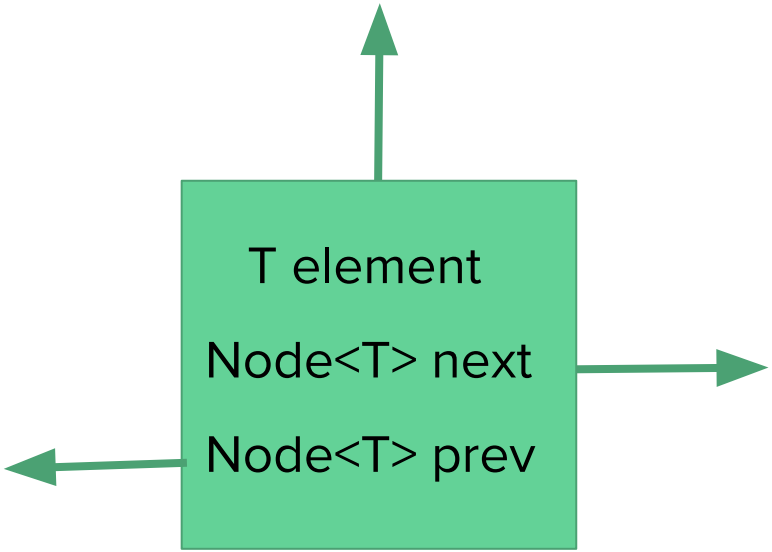
A simple, lightweight object, called a **node**, can be defined to hold the position of one element in a data structure by managing an element reference and at least one reference to another node in the structure.

For a list, a node would need a reference to the list element and a reference to the next node in the list, allowing traversal of all list elements from start to finish. Backward navigation would require an additional reference to the previous node in the list.

Node<T>



Single-linked

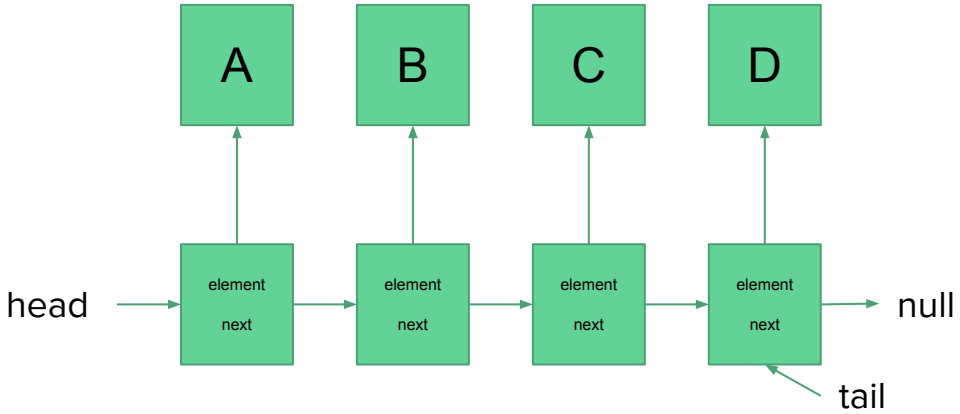


Double-linked

Interface vs Internal Perspective

User's abstract, interface perspective

Programmer's internal, implementation perspective

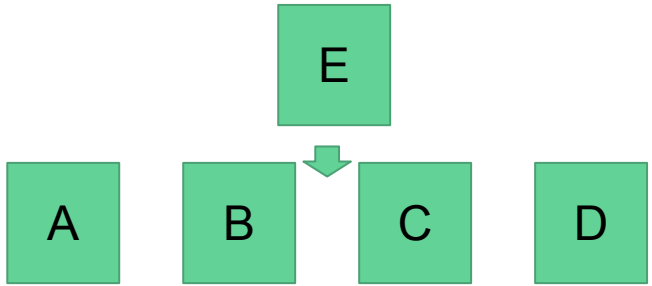


Code - Getting Started

- Write the Node<T> class (stand-alone class or private inner class of the list)
- Implement list interface
- Declare instance variables for head and tail nodes, and size
- Initialize instance variables in a constructor
- Identify useful first utility methods to implement

Adding an element - Interface perspective

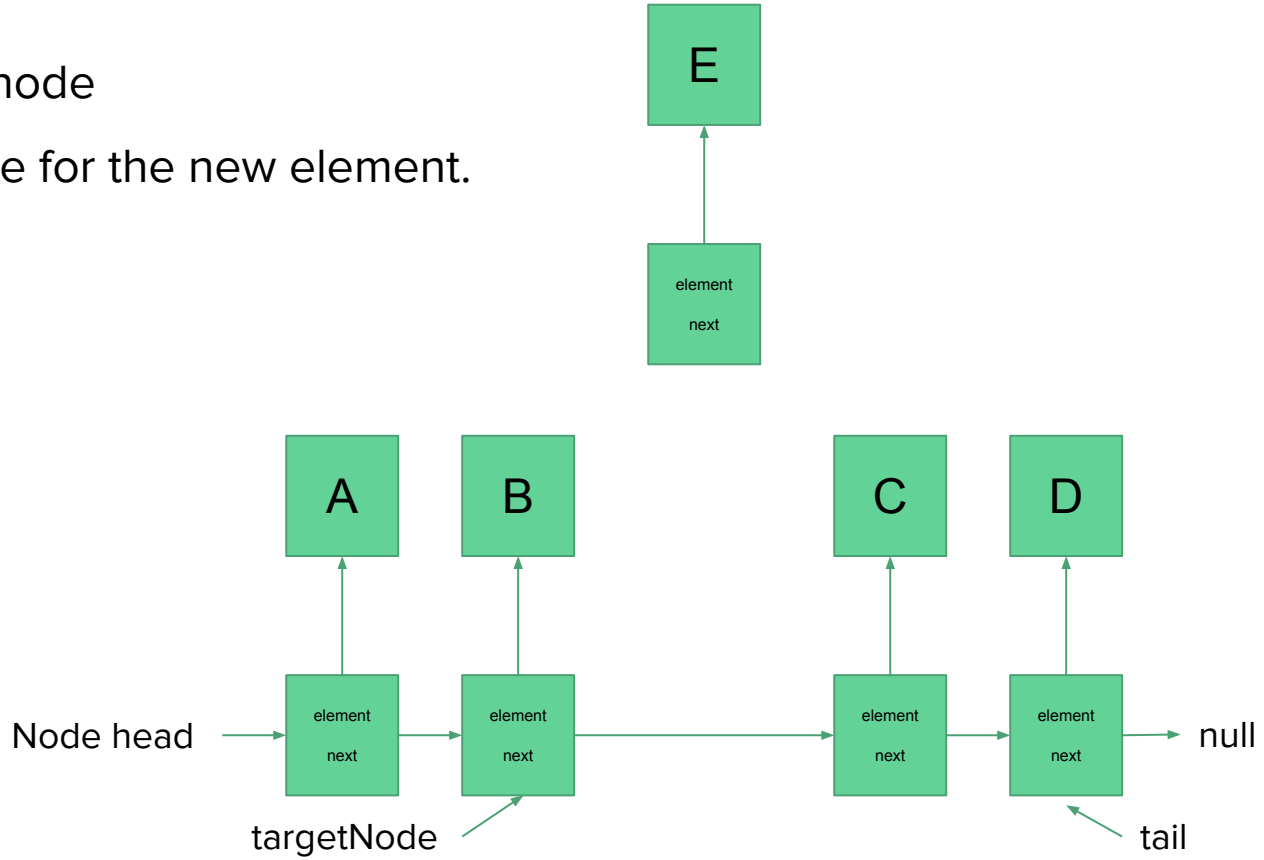
Add E after B



Done

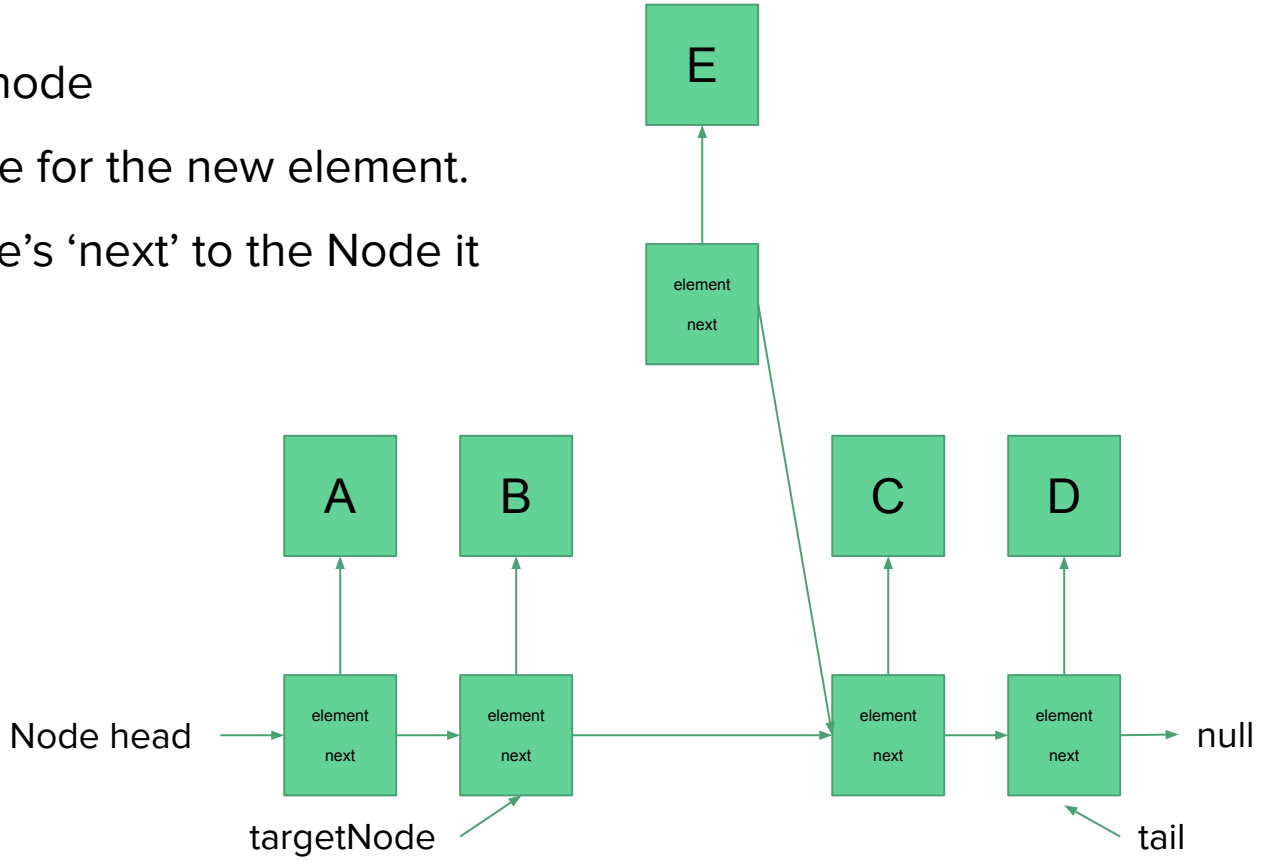
Adding an element - Internal perspective

1. Locate the target node
2. Create a new Node for the new element.



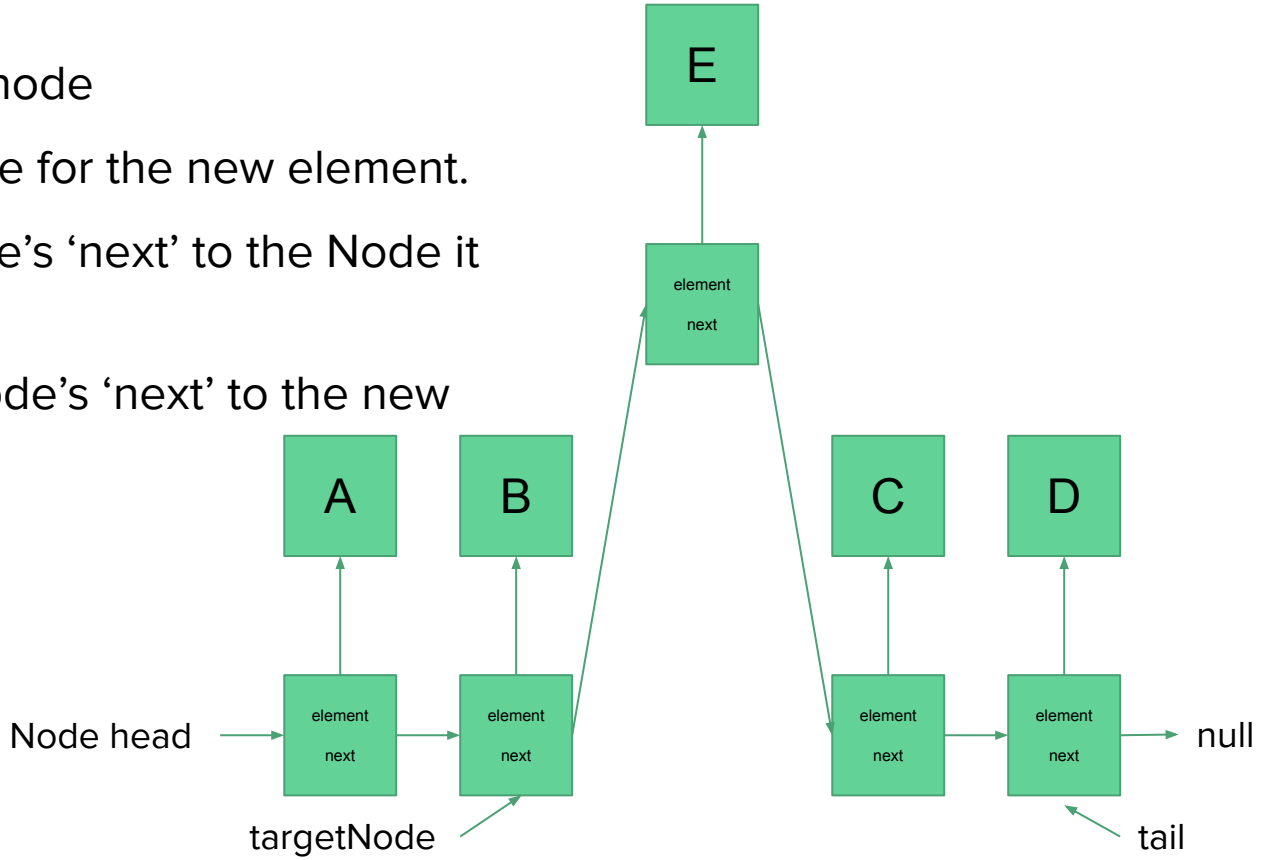
Adding an element - Internal perspective

1. Locate the target node
2. Create a new Node for the new element.
3. Connect new Node's 'next' to the Node it will precede.



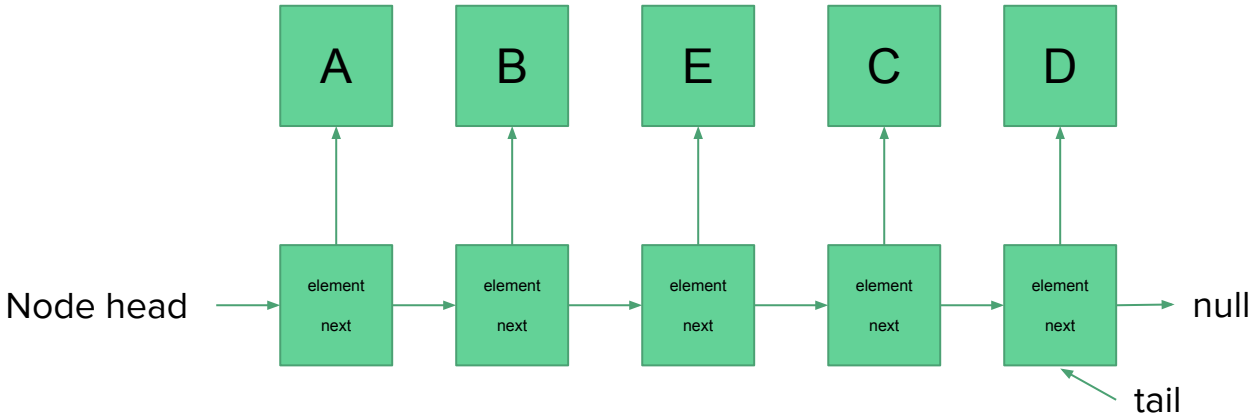
Adding an element - Internal perspective

1. Locate the target node
2. Create a new Node for the new element.
3. Connect new Node's 'next' to the Node it will precede.
4. Connect target Node's 'next' to the new Node.



Adding an element - Internal perspective

1. Locate the target node
2. Create a new Node for the new element.
3. Connect new Node's 'next' to the Node it will precede.
4. Connect target Node's 'next' to the new Node.
5. Increment size

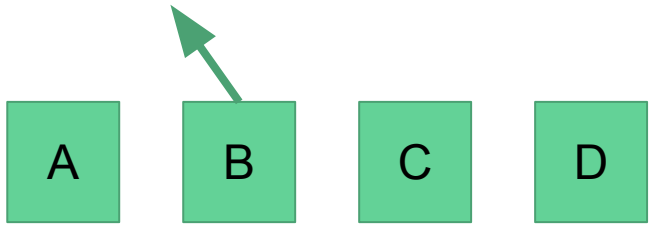


Code - addAfter(newElement, targetElement)

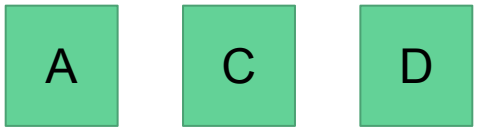
- Locate Node containing targetElement
  - Throw NoSuchElementException if not found
- Create a new Node with the newElement
- Connect new Node’s next reference to target Node’s next
- Connect target Node’s next to new Node
- Update tail reference if necessary
- Increment size

Removing an element - Interface perspective

Remove B

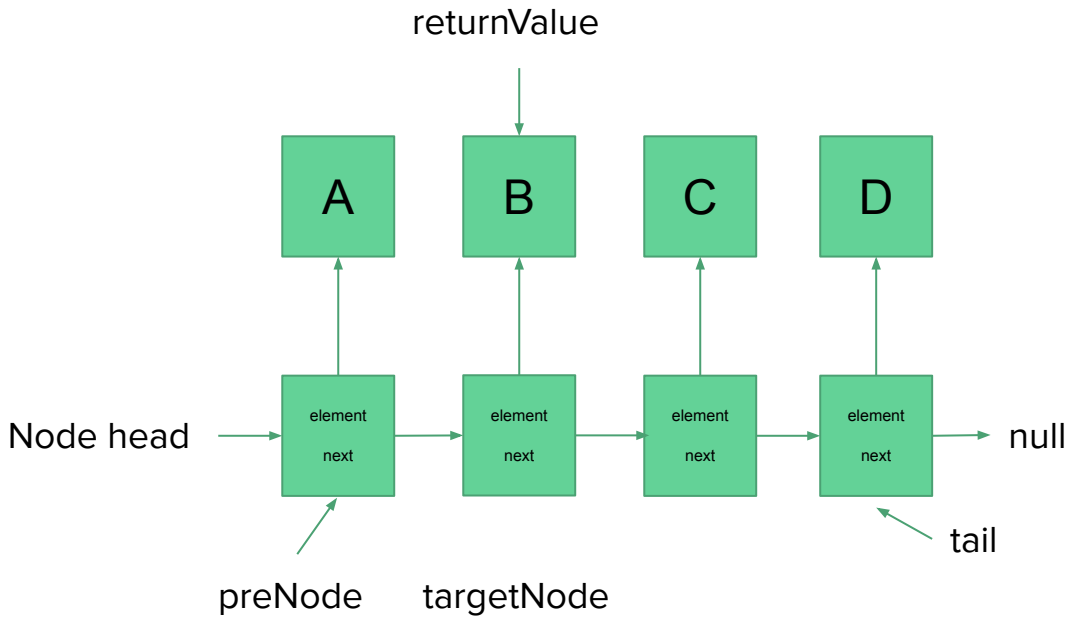


Done



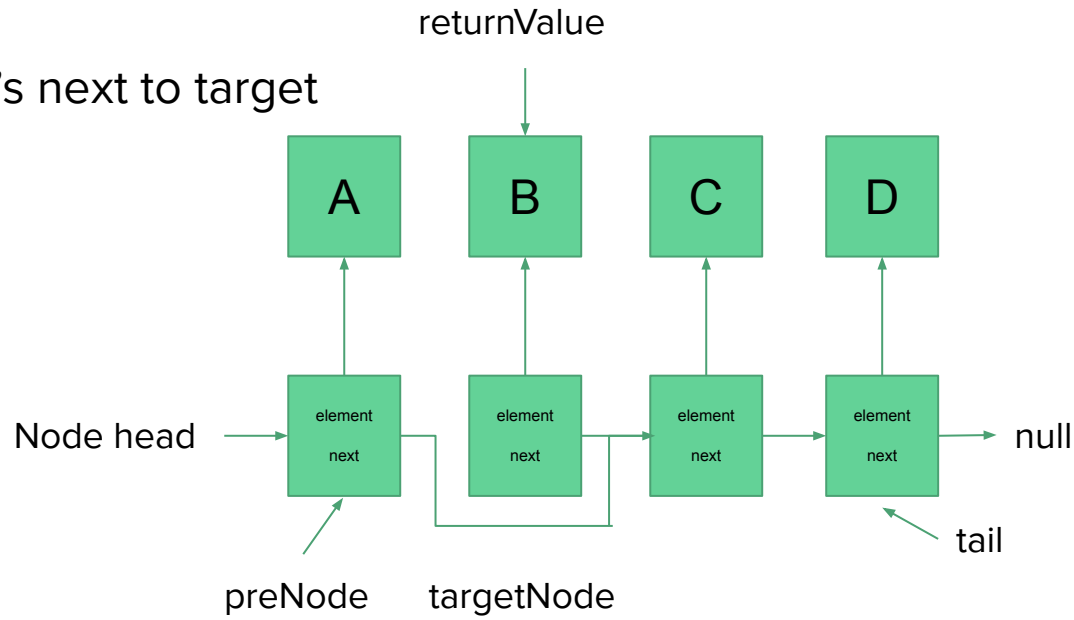
Removing an element - Internal perspective

1. Locate the node *before* the target node and remember the element in the target node for later return



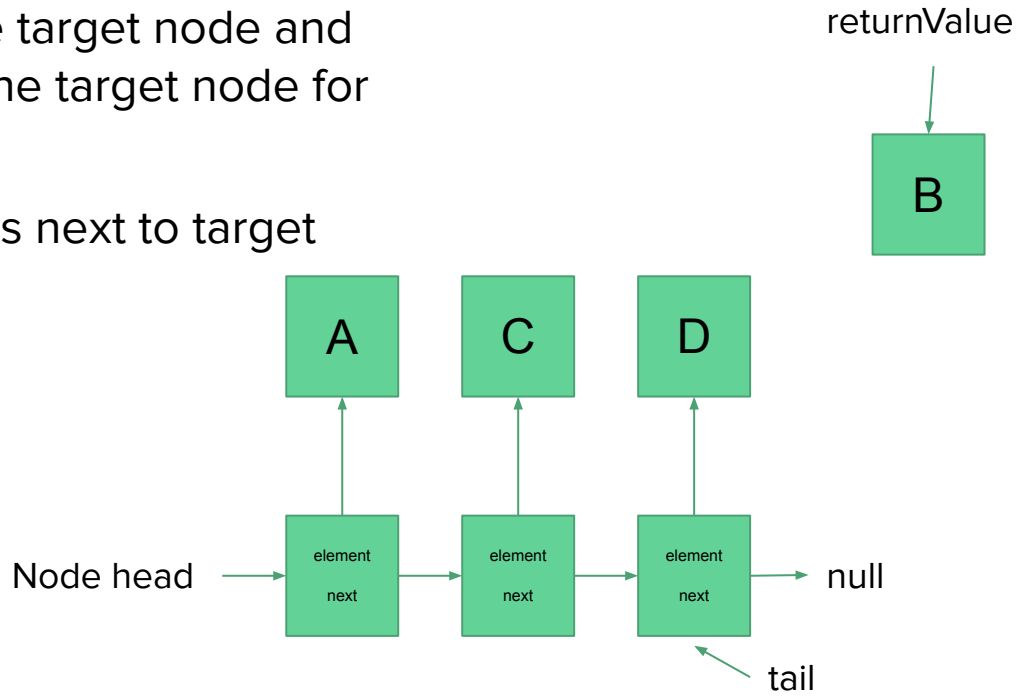
Removing an element - Internal perspective

1. Locate the node *before* the target node and remember the element in the target node for later return
2. Update predecessor Node's next to target Node's next



Removing an element - Internal perspective

1. Locate the node *before* the target node and remember the element in the target node for later return
2. Update predecessor Node's next to target Node's next
3. Decrement size
4. Return removed element



Code - remove(element)

- Handle special cases: one element or first element
- Else, locate the Node *before* the Node containing the element
  - Store a reference to the element being removed
  - Update predecessor Node’s next to target Node’s next
  - Update tail if necessary
- Decrement size
- Return removed element

LinkedLists

Mason Vail
Boise State University Computer Science