**Linked List: Deleting & Destroying**

- Linked list contains 0 or more nodes that are linked to one another:

```
list
head
```

- Linked list operations:
  - Append a node
  - Insert a node
  - Traverse the list
  - Delete a node
  - Delete/destroy the list
DELETING A NODE

- Every node is dynamically allocated:
  - `ListNode * newNode = new ListNode;`
  - `newNode->value = num; // set value to num`
  - `newNode->next = NULL; // set next to NULL`

- When a node is **deleted** from the list, it should be **deallocated** from memory

- “next” pointer that used to point to the deleted node must be updated to point to the deleted node’s “next”
DELETING A NODE

- Algorithm to delete a node requires **two pointers**:
  - One to locate the node to be deleted
  - One to point to the node before the node to be deleted

![Diagram of a linked list with nodes and pointers](image)
DELETING A NODE: num = 13

Initialization:
nodePtr points to 1st element; previousNode points to null.

ListNode * nodePtr = head;
ListNode * previousNode = 0;
DELETING A NODE:  
num = 13

1st iteration:  
Is nodePtr == NULL?  
Is nodePtr->value == 13?  
No. Thus, update the pointers.

previousNode

null

nodePtr

list

head

5

13

19

null
DELETING A NODE: num = 13

1st iteration: Update nodePtr and previousNode

previousNode = nodePtr;
nodePtr = nodePtr->next;
DELETING A NODE: num = 13

2nd iteration:
Is nodePtr == NULL?
Is nodePtr->value == 13?
Yes. We have arrived at the node that must be deleted.
DELETING A NODE: num = 13

previousNode

nodePtr

list head

5 -> 13 -> 19 -> null

Delete node: What pointers must change? previousNode->next
DELETING A NODE:
NUM = 13

previousNode  nodePtr

5 13 19

Update
previousNode->next

previousNode->next = nodePtr->next;
DELETING A NODE:
num = 13

previousNode | nodePtr
---|---

list
head

5 | 19

null

delete nodePtr;

Delete the node!
DELETING A NODE: SPECIAL CASES

- What if the list is empty?
- `deleteNode(3);`
- Just exit the function: nothing to delete!
DELETING A NODE: SPECIAL CASES

- What if the node is not there?
- `deleteNode(3);`
- If `nodePtr` == `NULL`, just exit the function: nothing to delete!
DELETING A NODE: SPECIAL CASES

- What if the first node must be deleted?
- `deleteNode(5);`
- Must update the head pointer

```cpp
nodePtr = head->next;
delete head;
head = nodePtr;
```
DELETING A NODE: SPECIAL CASES

- What if the first node must be deleted?
- `deleteNode(5);`
- Must update the head pointer

```c
nodePtr = head->next;
delete head;
head = nodePtr;
```

```
prevNode  nodePtr

list head

5    13    19
null
```
Deleting a Node: Special Cases

- What if the first node must be deleted?
- `deleteNode(5);`
- Must update the head pointer

```
nodePtr = head->next;
delete head;
head = nodePtr;
```
DELETING A NODE: SPECIAL CASES

- What if the first node must be deleted?
- `deleteNode(5);`
- Must update the head pointer

```c
nodePtr = head->next;
delete head;
head = nodePtr;
```
void NumberList::deleteNode(double num) {
    ListNode *nodePtr;    // To traverse the list
    ListNode *previousNode; // To point to the previous node

    // If the list is empty, do nothing.
    if (!head)
        return;

    // Determine if the first node is the one.
    if (head->value == num)
    {
        nodePtr = head->next;
        delete head;
        head = nodePtr;
    }
    else
    {
        // Initialize nodePtr to head of list
        nodePtr = head;

        // Skip all nodes whose value member is
        // not equal to num.
        while (nodePtr != nullptr && nodePtr->value != num)
        {
            previousNode = nodePtr;
            nodePtr = nodePtr->next;
        }
    }
}
151    // If nodePtr is not at the end of the list,
152    // link the previous node to the node after
153    // nodePtr, then delete nodePtr.
154    if (nodePtr)
155    {
156        previousNode->next = nodePtr->next;
157        delete nodePtr;
158    }
159
160    }
Can you delete a node using nodePtr only?

You can’t!
A temporary pointer is necessary, otherwise address of node 13 is lost.
DESTROYING A LINKED LIST

- Linked List destructor must remove all nodes used in the list
- To do this, use list traversal to visit each node
- For each node,
  - Unlink the node from the list
  - Free the node’s memory
- Set the list head to NULL
DESTROYING THE LIST ONE NODE AT A TIME

At every iteration:
- Move nodePtr forward;
- Delete head;
- Set head = nodePtr;
At every iteration:

- Move nodePtr forward;
- Delete head;
- Set head = nodePtr;
At every iteration:
- Move nodePtr forward;
- Delete head;
- Set head = nodePtr;
DESTROYING THE LIST ONE NODE AT A TIME

At every iteration:

- Move nodePtr forward;
- Delete head;
- Set head = nodePtr;
DESTROYING THE LIST ONE NODE AT A TIME

- At every iteration:
  - Move nodePtr forward;
  - Delete head;
  - Set head = nodePtr;
DESTROYING THE LIST ONE NODE AT A TIME

At every iteration:

- **Move nodePtr forward**;
- **Delete head**;
- **Set head = nodePtr**;

![Diagram showing list destruction](image)
DESTROYING THE LIST ONE NODE AT A TIME

- At every iteration:
  - Move nodePtr forward;
  - Delete head;
  - Set head = nodePtr;
  - Done!