Data Structures & Algorithms

Linked Lists

NUST
SCHOOL OF ELECTRICAL ENGINEERING &
COMPUTER SCIENCE (SEECS)

Today‘s lecture

NUST SEECS

- Linked structures
  - Singly Linked Lists

Linked List Concepts

NUST SEECS

- Data is stored dynamically
  - Each node is created as necessary
- Nodes of linked lists are not necessarily stored contiguously in memory (as in an array)
- Although lists of data can be stored in arrays, linked lists provide several advantages

Advantages of Linked Lists

NUST SEECS

- The size of a "conventional" C++ array cannot be altered because the array size is fixed at compile time
- Also, arrays can become full (i.e., all elements of the array are occupied)

Advantage 1: **Dynamic**

- A linked list is appropriate when the number of data elements to be stored in the list is unknown
- Because linked lists are dynamic, their size can grow or shrink to accommodate the actual number of elements in the list
- A linked list is full only when the computer runs out of memory in which to store nodes

Advantages of Linked Lists

NUST SEECS

Advantage 2: **Easy Insertions and Deletions**

- Although arrays are easy to implement and use, they can be quite inefficient when sequenced data needs to be inserted or deleted.
- With arrays, it is difficult to rearrange data (copying to temporary variables, etc.)
- However, the linked list structure allows us to easily insert and delete items from a list

Disadvantages of Linked Lists

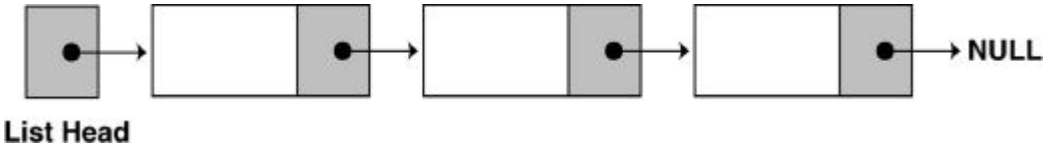
NUST SEECS

- Unfortunately, linked lists are also not without drawbacks:
  - For example, we can perform efficient searches on arrays (e.g., binary search) but this is not practical with a linked list.
  - No random access possible in linked-list like in arrays. The linked-list has to be traversed from node-to-node until the desired node has been accessed. Hence, they have linear time access O(n) in worst case.
    Arrays on the other hand have constant time access to its elements.

Linked List Composition

NUST SEECS

- A linked list is called "linked" because each node in the series has a pointer that points to the next node in the list



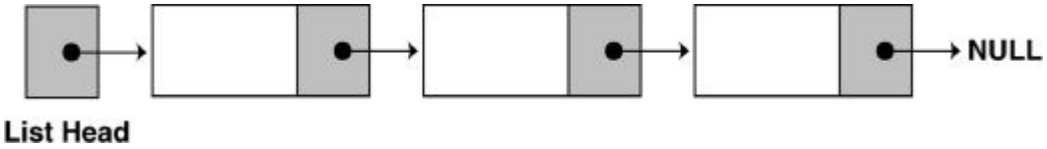
- I.e., every node contains a data member that is a pointer to another node allowing many nodes to be strung together and accessed using only one variable

- If a node has a link only to its successor in the sequence, the list is then called a ***singly linked list***

Declarations in Singly Linked List

NUST SEECS

- First you must declare a data structure that will be used for the nodes



  - E.g., the following **struct** could be used to create a list where each node holds a float:

```
struct ListNode {
   float value;
   struct ListNode *next;
};
```

Declarations in Singly Linked List

NUST SEECS

- The next step is to declare a pointer to serve as the list head, as shown below.

ListNode *head;

- Once you have declared a node data structure and have created a **NULL** head pointer, you have an empty linked list.
- The next step is to implement operations with the list.

Linked List Operations

NUST SEECS

- Creating the list
  - Initialize pointers to NULL;
- Inserting nodes
  - Insert at beginning
  - Insert at middle
  - Insert at last
- Deleting nodes
  - Delete from beginning, middle, last
- Traversing the list
- Searching a specified item in the list
- Destroying the list

```
//floatList.h
class FloatList {
private:
        // Declare a structure for the list
        struct ListNode {
                float value;
                struct ListNode *next;
         };
        ListNode *head; // List head pointer
public:
        FloatList(void) { // Constructor
                head = NULL;
        }
        ~FloatList(void) { }; // Destructor
         void appendNode(float);
        void displayList(void);
        void deleteNode(float);
};
```

```
//floatList.h
class FloatList {
private:
        // Declare a structure for the list
        struct ListNode {
                float value;
                struct ListNode *next;
         };
        ListNode *head; // List head pointer


public:


        FloatList(void) // Constructor {
                head = NULL;
        }
~FloatList(void) { }; // Destructor
 void appendNode(float);
void displayList(void);
void deleteNode(float);
};
```

Appending a node to the list

NUST SEECS

- To append a node to a linked list means to add the node to the end of the list.

- The pseudo code is shown below:

  - *Create a new node.*
  - *Store data in the new node.*
  - *If there are no nodes in the list*
    - *Make the new node the first node.*
  - *Else*
    - *Traverse the List to Find the last node.*
    - *Add the new node to the end of the list.*
  - *End If.*

Appending a node to the list

NUST SEECS

```
void FloatList::appendNode(float num)
 {
        ListNode *newNode, *nodePtr;

        // Allocate a new node & store num
        newNode = new ListNode;
        newNode->value = num;
        newNode->next = NULL;

        // If there are no nodes in the list make newNode the first node
         if (!head) // head == NULL
                head = newNode;
         else  // Otherwise, insert newNode at end
        {
              nodePtr = head;
              while (nodePtr->next) // Find the last node in the list
                        nodePtr = nodePtr->next;
              nodePtr->next = newNode; // Insert newNode as the last node
              }
```

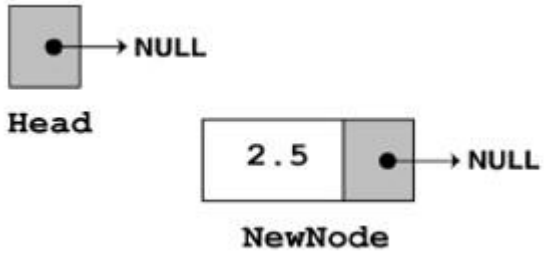
```
void main (void)
{
FloatList list;
list.appendNode(2.5);
list.appendNode(7.9);
list.appendNode(12.6)
;
}
```

Appending a node to the list

NUST SEECS

```
void FloatList::appendNode(float num)
 {
        ListNode *newNode, *nodePtr;

        // Allocate a new node & store num
        newNode = new ListNode;
        newNode->value = num;
         newNode->next = NULL;

        // If there are no nodes in the list make newNode the first node
         if (!head) // head == NULL
                 head = newNode;
         else  // Otherwise, insert newNode at enc
        { nodePtr = head;
              while (nodePtr->next) // Find the last
                        nodePtr = nodePtr->next;

              nodePtr->next = newNode; // Insert newNode as the last node
              }
```

```
void main (void)
{
FloatList list;
list.appendNode(2.5);
list.appendNode(7.9);
list.appendNode(12.6)
;
}
```

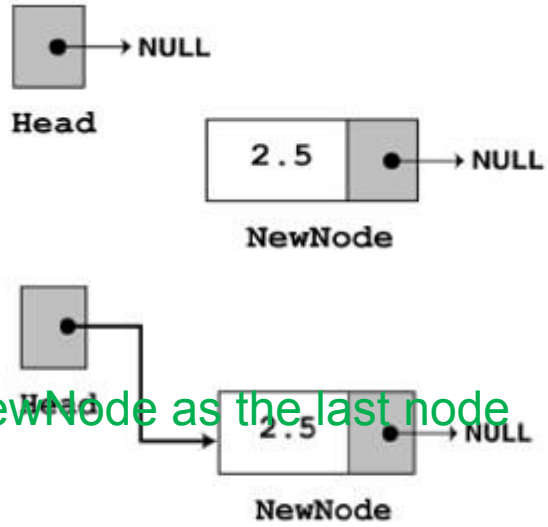


Appending a node to the list

NUST SEECS

```
void FloatList::appendNode(float num)
 {
        ListNode *newNode, *nodePtr;

        // Allocate a new node & store num
        newNode = new ListNode;
        newNode->value = num;
         newNode->next = NULL;

        // If there are no nodes in the list make newNode the first node
         if (!head) // head == NULL
                head = newNode;
         else  // Otherwise, insert newNode at enc
        { nodePtr = head;
              while (nodePtr->next) // Find the last
                        nodePtr = nodePtr->next;

              nodePtr->next = newNode; // Insert newNode as the last node
              }
```

```
void main (void)
{
FloatList list;
list.appendNode(2.5);
list.appendNode(7.9);
list.appendNode(12.6)
;
}
```

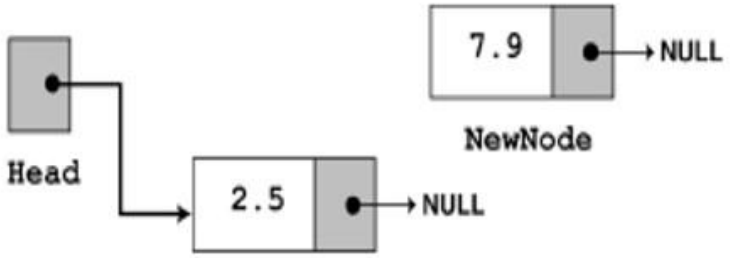


Appending a node to the list

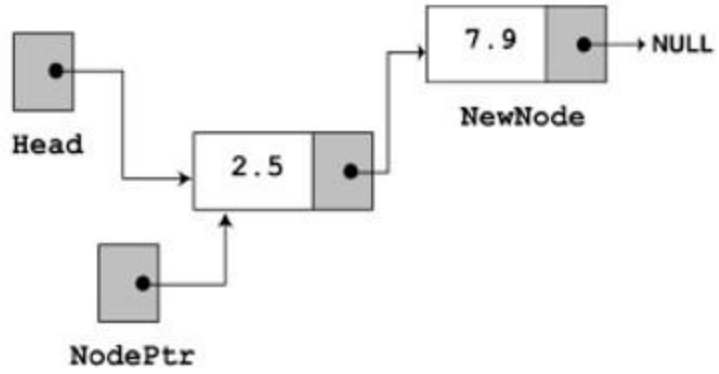
NUST SEECS

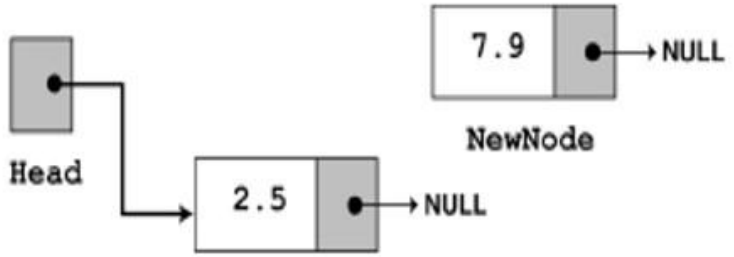
```
void FloatList::appendNode(float num)
 {
        ListNode *newNode, *nodePtr;

        // Allocate a new node & store num
        newNode = new ListNode;
        newNode->value = num;
         newNode->next = NULL;

        // If there are no nodes in the list make
         if (!head) // head == NULL
                 head = newNode;
         else  // Otherwise, insert newNode at
        { nodePtr = head;
              while (nodePtr->next) // Find the last node in the list
                        nodePtr = nodePtr->next;

              nodePtr->next = newNode; // Insert newNode as the last node
              }
```

```
void main (void)
{
FloatList list;
list.appendNode(2.5);
list.appendNode(7.9);
list.appendNode(12.6)
;
}
```



Appending a node to the list






```
else  // Otherwise, insert newNode at
{ nodePtr = head;
      while (nodePtr->next) // Find the last node in the list
            nodePtr = nodePtr->next;

      nodePtr->next = newNode; // Insert newNode as the last node
      }
```

Appending a node to the list

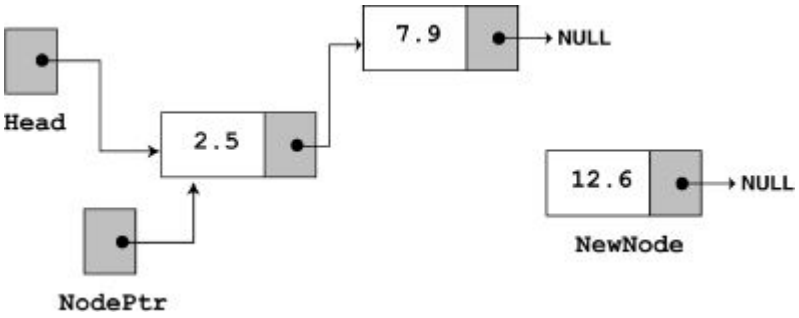
NUST SEECS

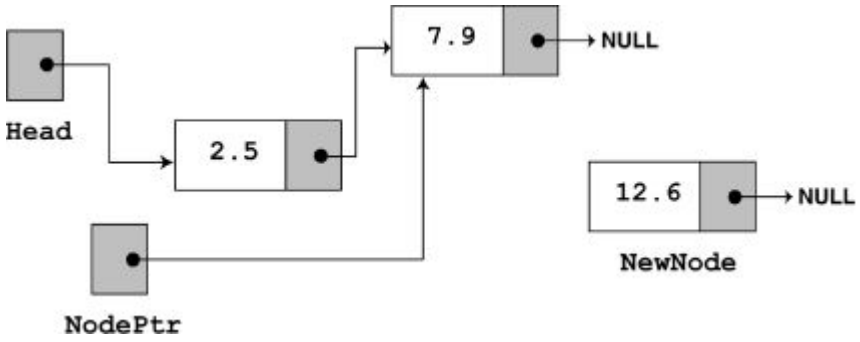
```
void FloatList::appendNode(float num)
 {
        ListNode *newNode, *nodePtr;

        // Allocate a new node & store num
        newNode = new ListNode;
        newNode->value = num;
         newNode->next = NULL;

        // If there are no nodes in the list make newNode the first node
         if (!head) // head == NULL
                head = newNode;
         else  // Otherwise, insert newNode at end
        { nodePtr = head;
              while (nodePtr->next) // Find the last node in the list
                        nodePtr = nodePtr->next;

              nodePtr->next = newNode; // Insert newNode as the last node
              }
```

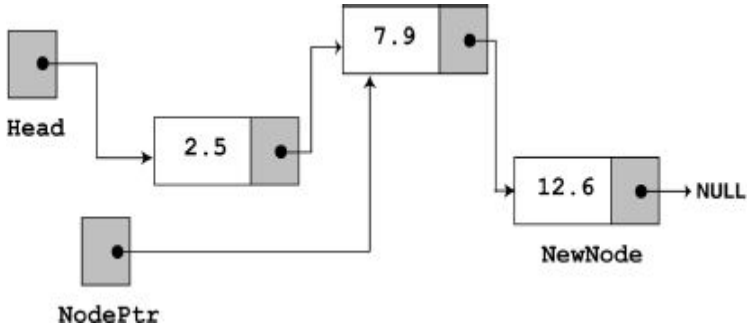
```
void main (void)
{
FloatList list;
list.appendNode(2.5);
list.appendNode(7.9);
list.appendNode(12.6)
;
 }
```

Appending a node to the list

NUST SEECS







Traversing the list

NUST SEECS

- To traverse the list we need a walking (navigator/traversal) pointer
  - This pointer is used to move from node to node as each element is processed
  - Example: displayList function….

Assign List head to node pointer.
While node pointer is not NULL
- *Display the value member of the node pointed to by node pointer.*
- *Assign node pointer to its own next member.*

End While.

Display list

NUST SEECS

```
void FloatList::displayList(void)
 {
   ListNode *nodePtr;
   nodePtr = head;

   while (nodePtr)
   {
            cout << nodePtr->value << endl;
           nodePtr = nodePtr->next;
   }
}
```

```
void main(void)
{
    FloatList List;
    list.appendNode(2.5);
    list.appendNode(7.9);
    list.appendNode(12.6);
    list.displayList();
}
```

OUTPUT

2.5

7.9

12.6

Deleting a node

NUST SEECS

- Deleting a node from a linked list requires two steps:
  - Remove the node from the list without breaking the links created by the next pointers
  - Deleting the node from memory

Deleting a node

NUST SEECS

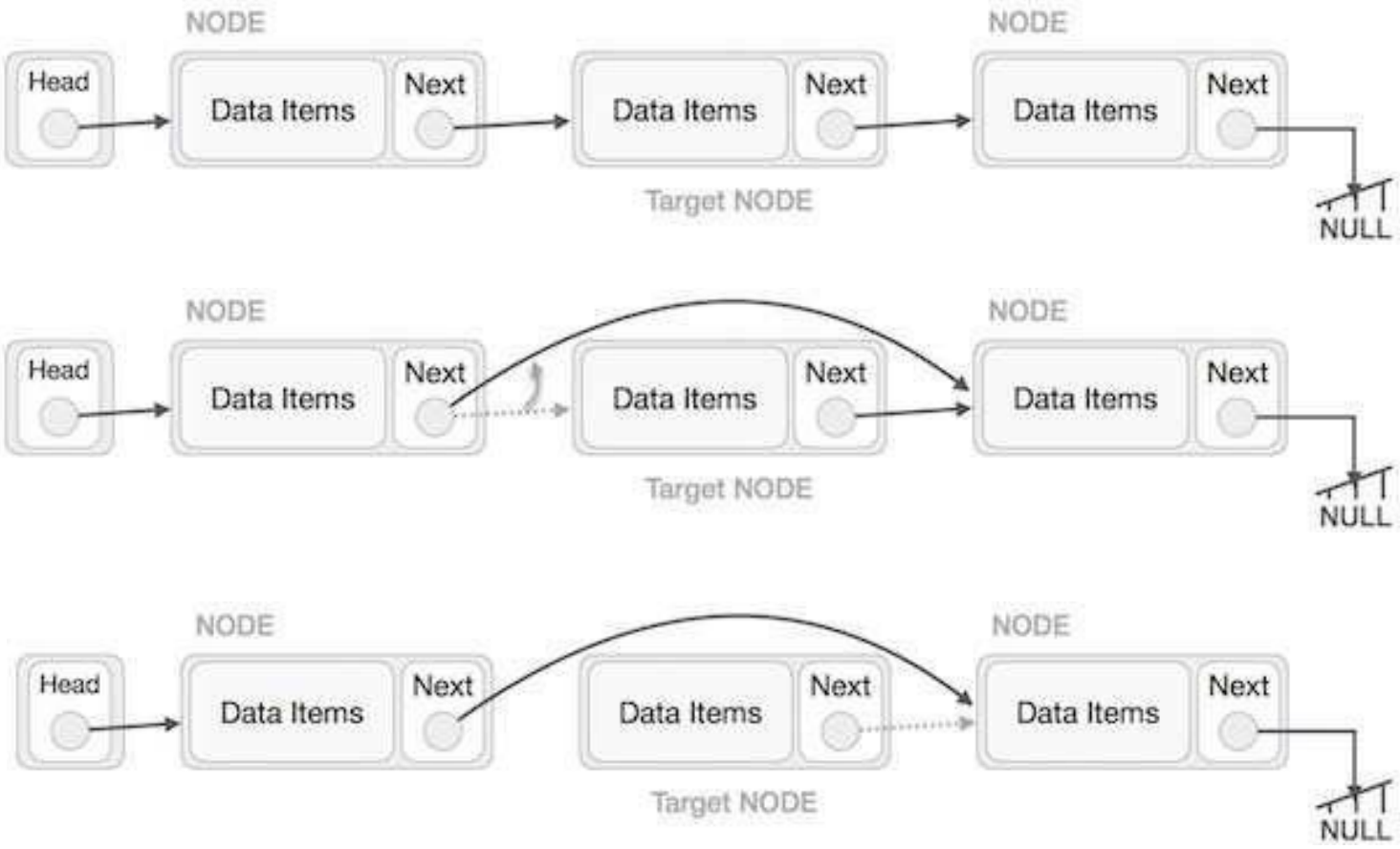
```
void FloatList::deleteNode(float num)
 {
   ListNode *currentNode, *previousNode;
   // If the list is empty, do nothing.
   if (head==nullptr)
            return;
   // Determine if the first node is the one.
   if (head->value == num) {
      currentNode = head->next;
      delete head;
      head = currentNode;
 }
```

Deleting a node

NUST SEECS

```
else {
// Initialize nodePtr to head of list
    currentNode =head;
// Skip all nodes whose value member is not equal to num.
   while (currentNode != NULL && currentNode->value != num)
   {
            previousNode = currentNode;
            currentNode = currentNode->next;
   }
// Link the previous node to the node after nodePtr, then delete  nodePtr.
            if (currentNode != nullptr) {
                 previousNode->next = currentNode->next;
                 delete currentNode;
            }
    }
}
```

Deleting a node

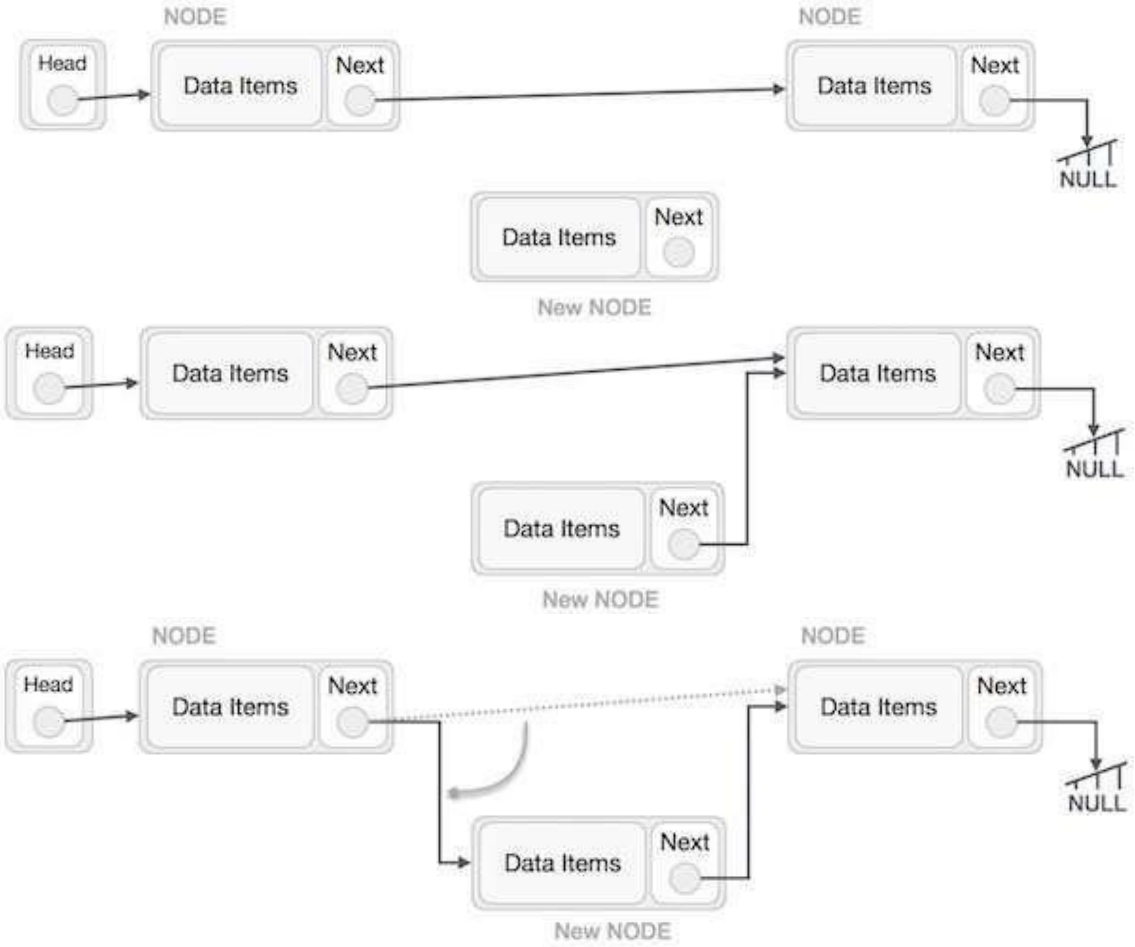


From:
http://www.tutorialspoint.com/data_structures_algorithms/data_structures_algorithms_tutorial.pdf

Insert in the middle

NUST SEECS

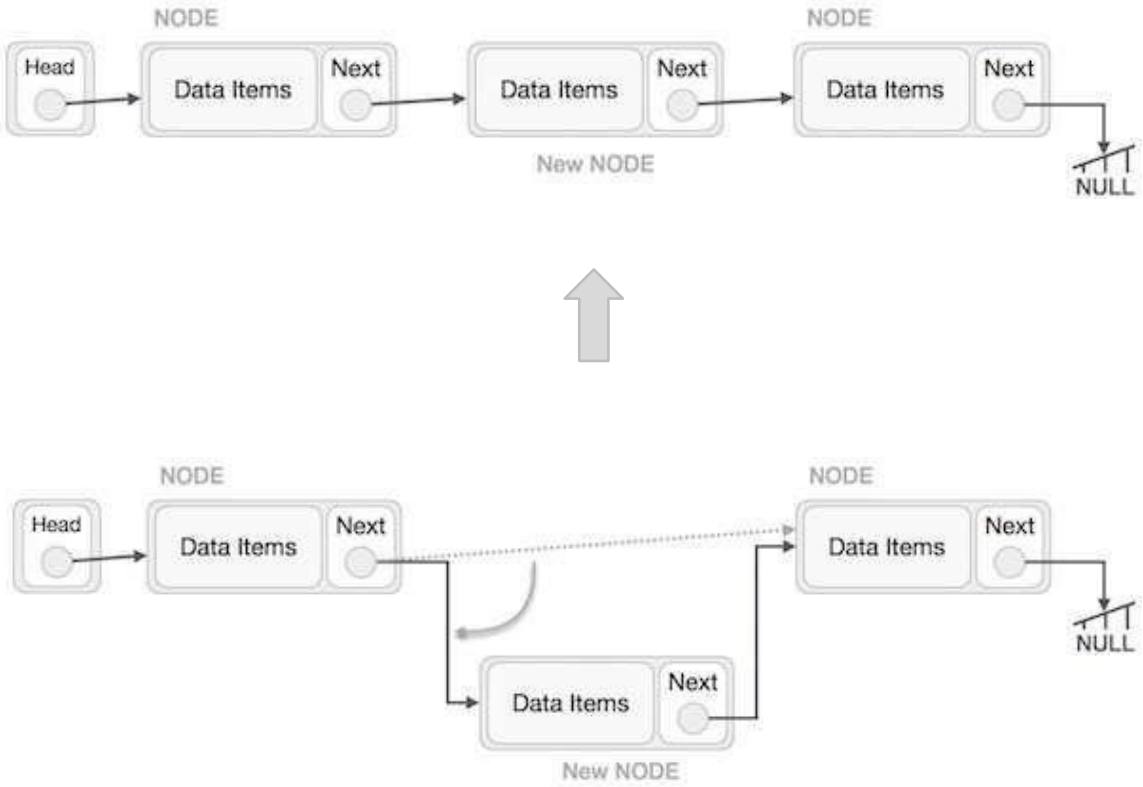
- Insert an item in the middle of the list
  - Diagram/Code?



Insert in the middle

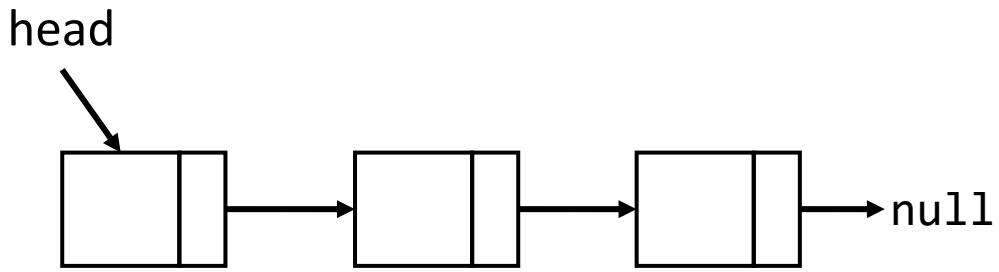
NUST SEECS

- Insert an item in the middle of the list
  - Diagram/Code?



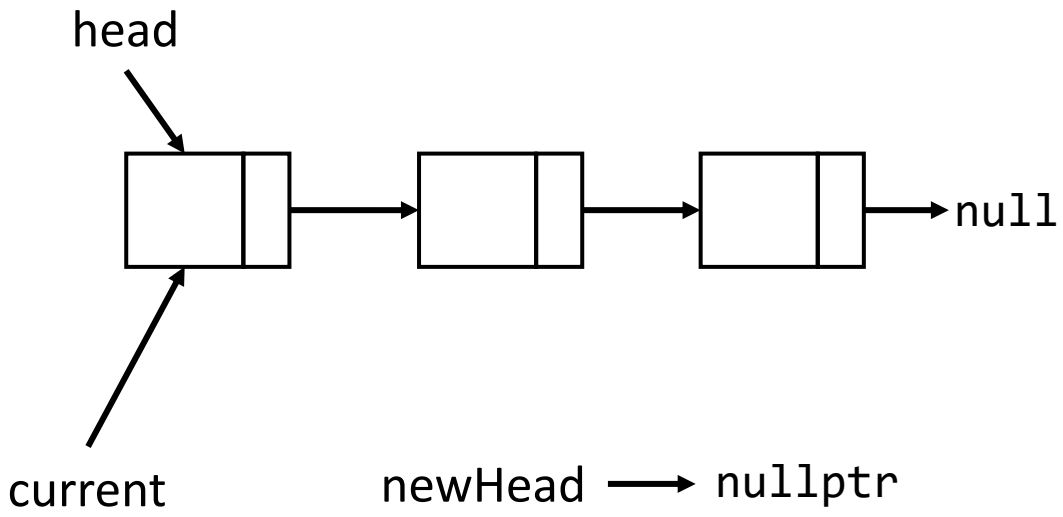
What is the following code doing?

```
struct Node* xxxxxxx(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



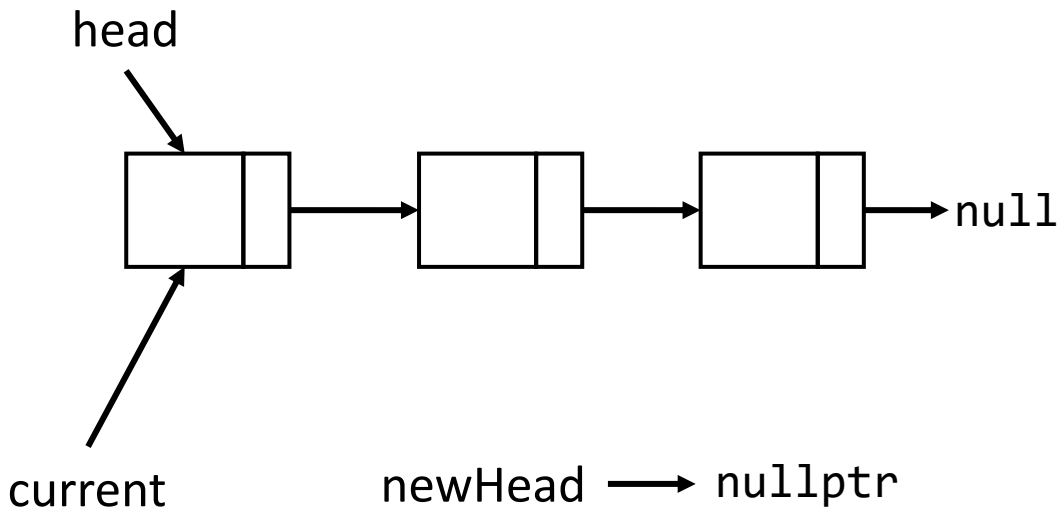
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



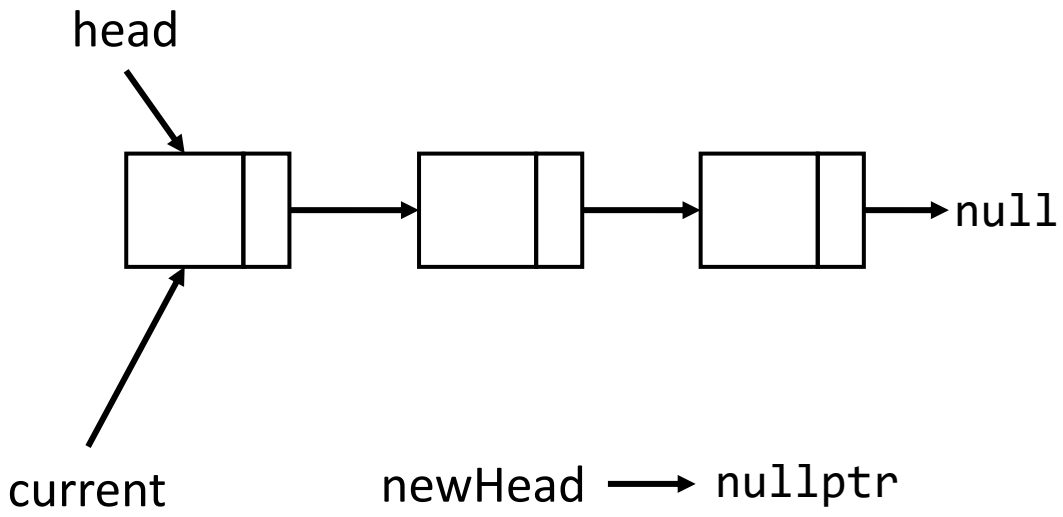
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



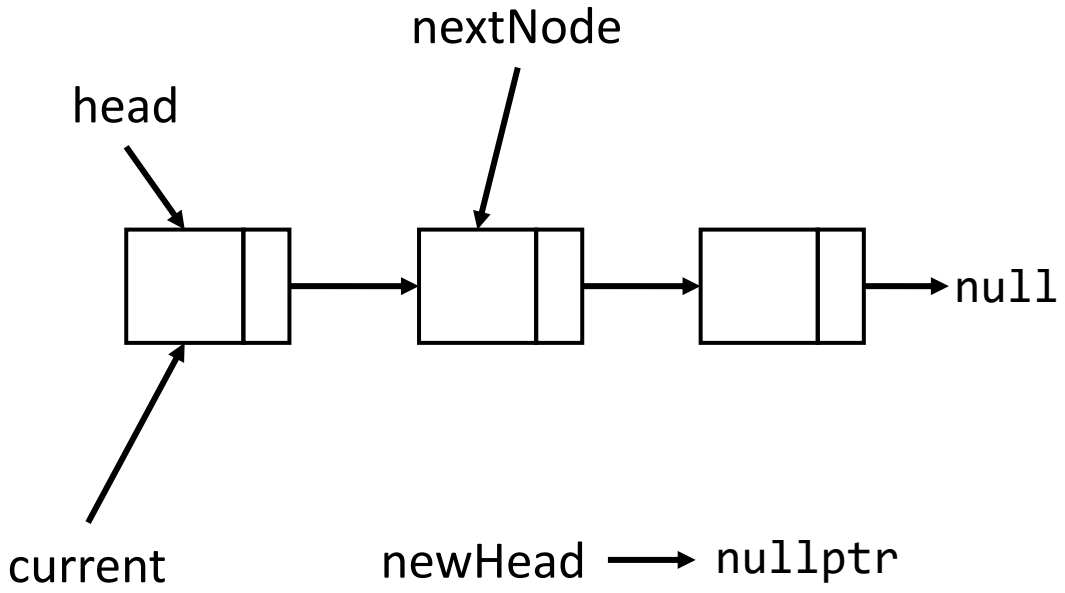
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



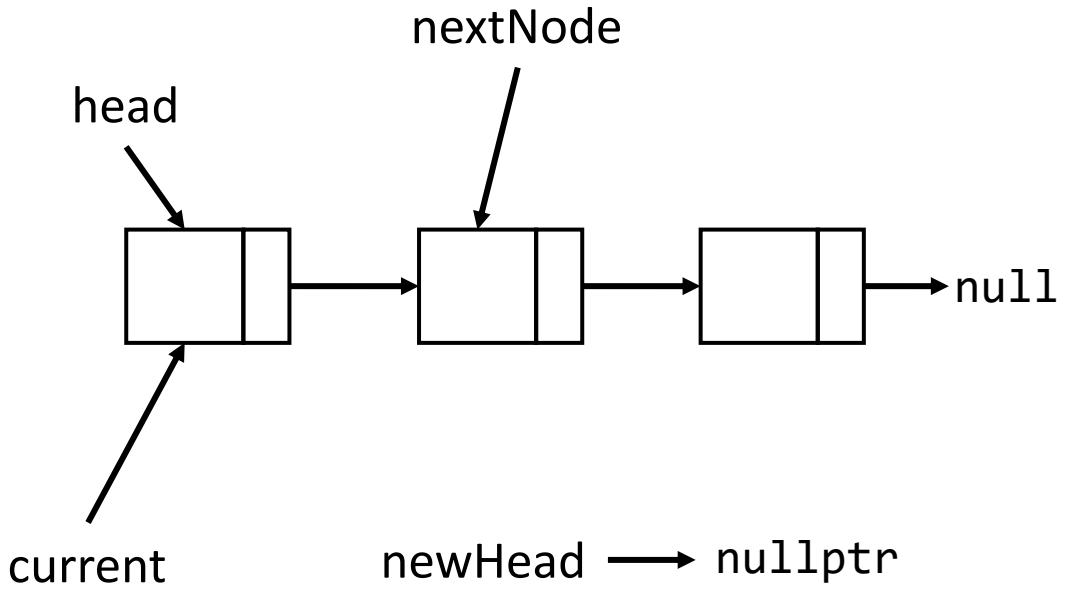
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



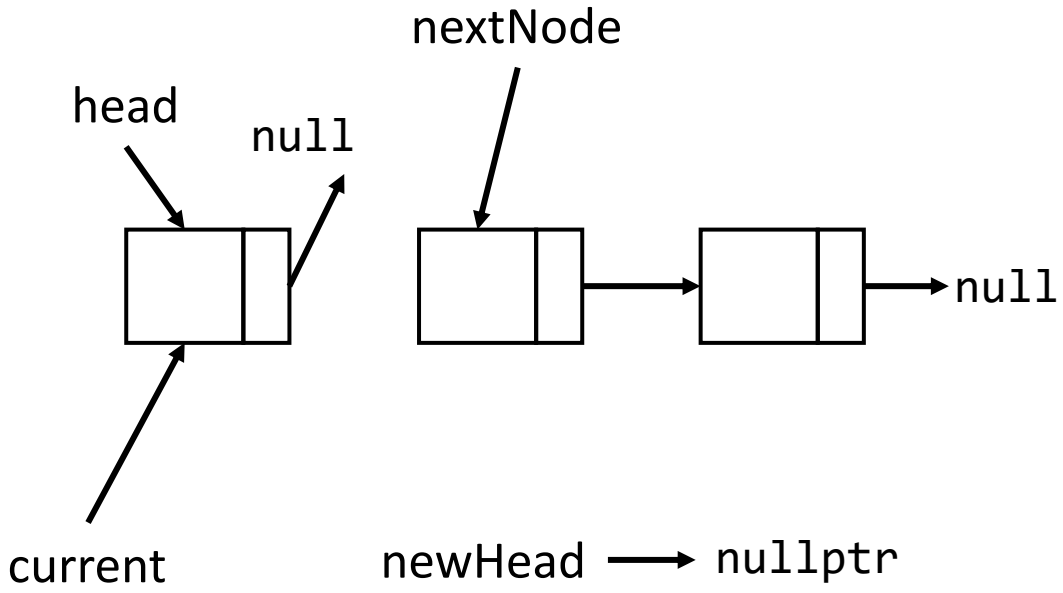
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



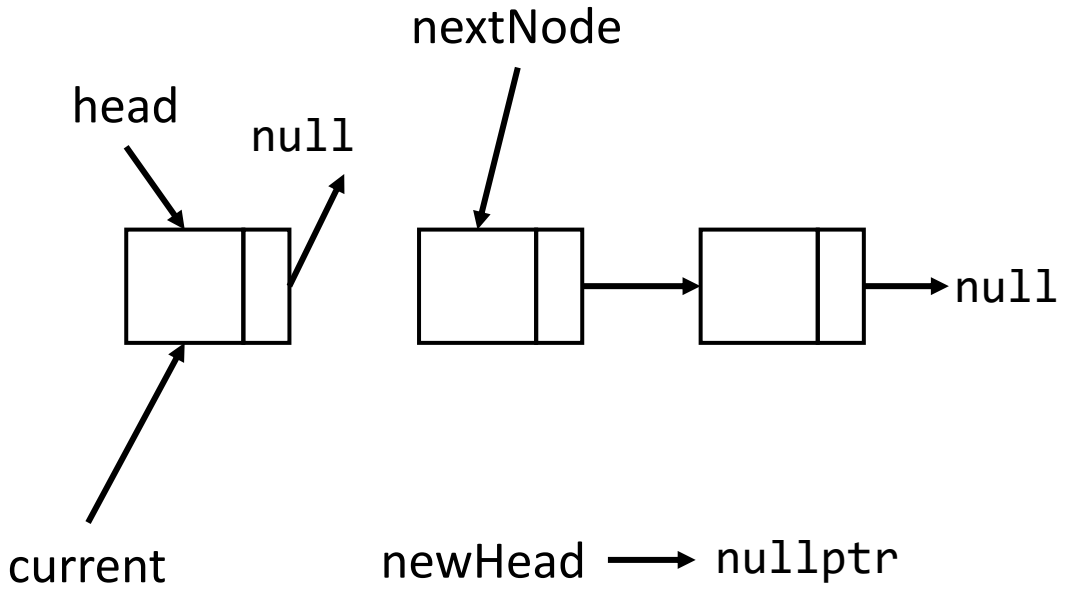
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



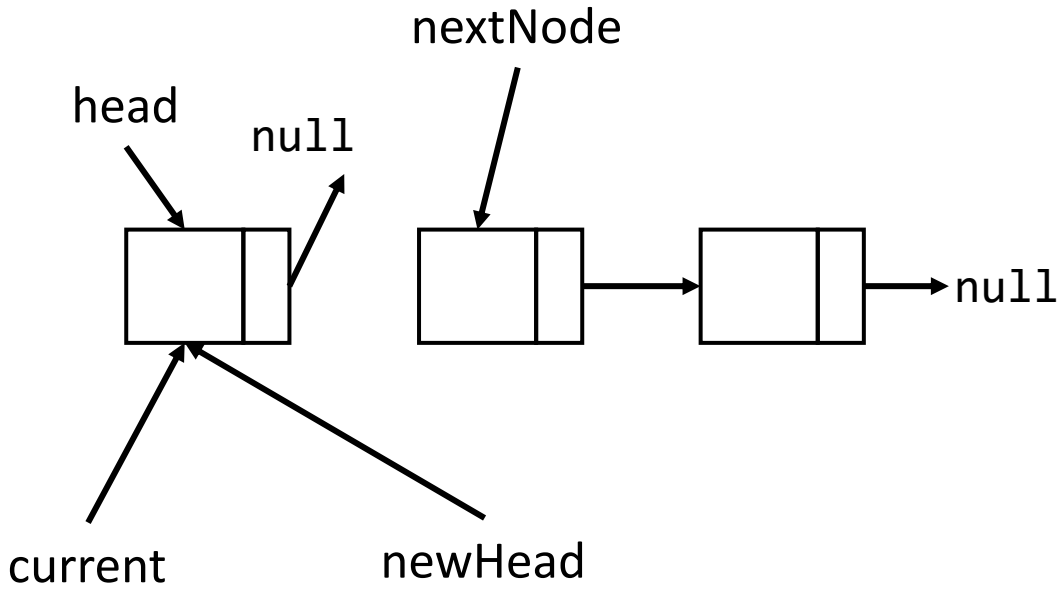
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



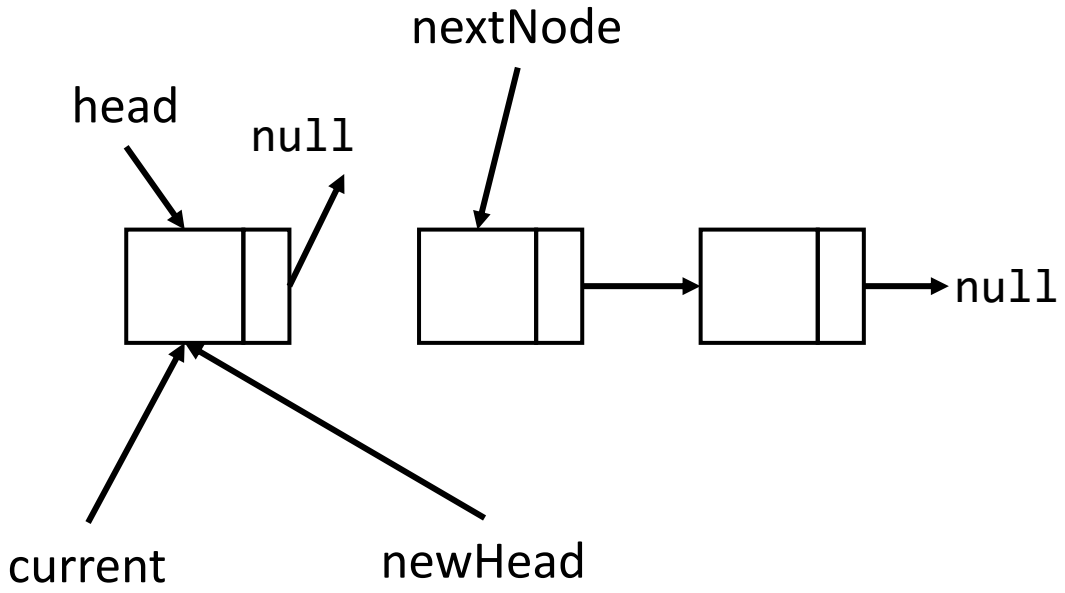
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



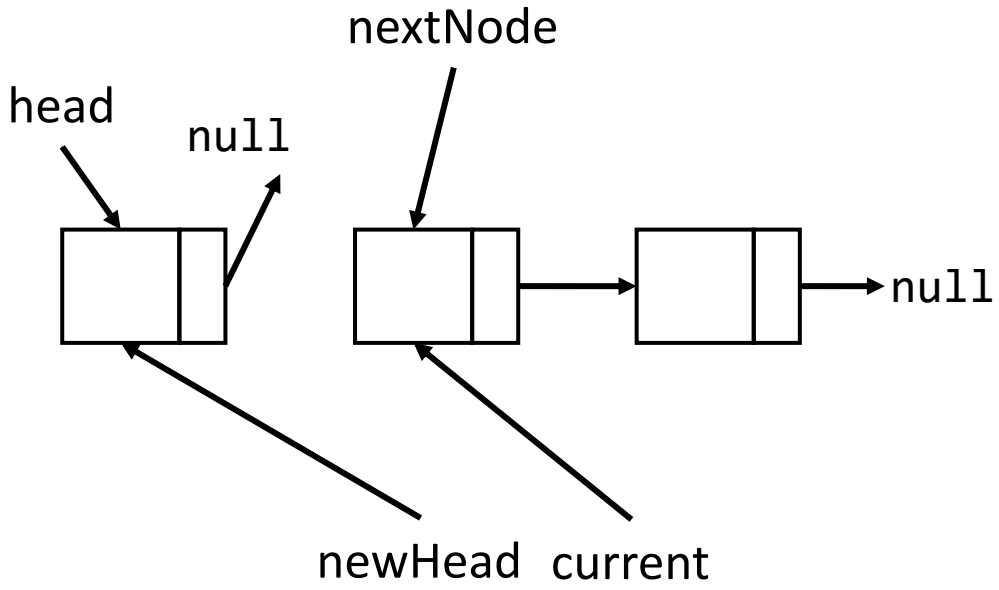
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



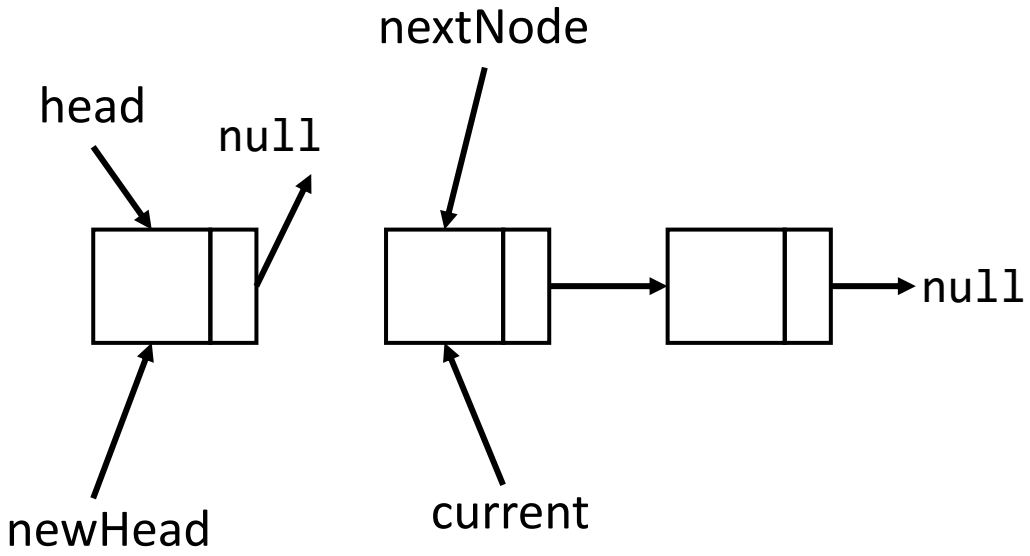
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



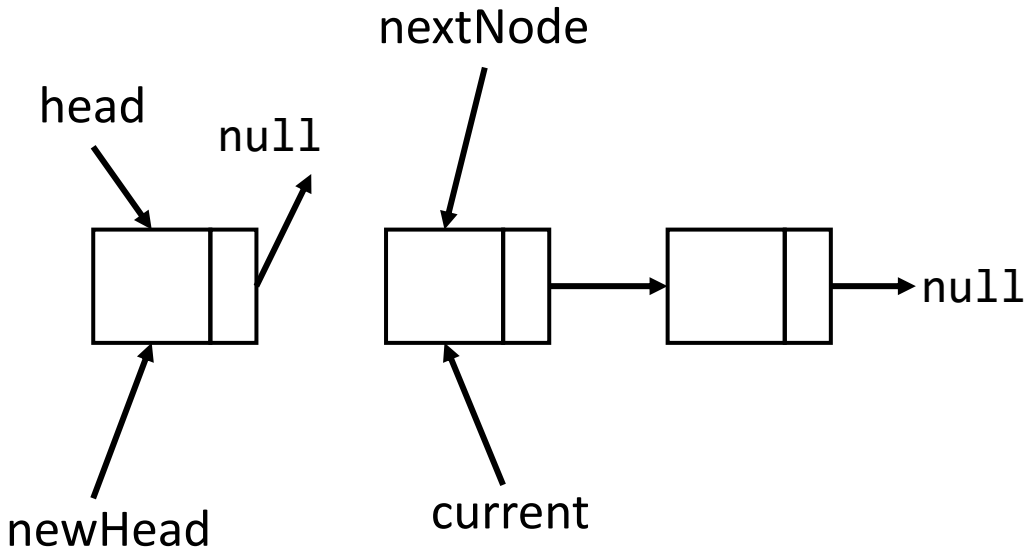
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



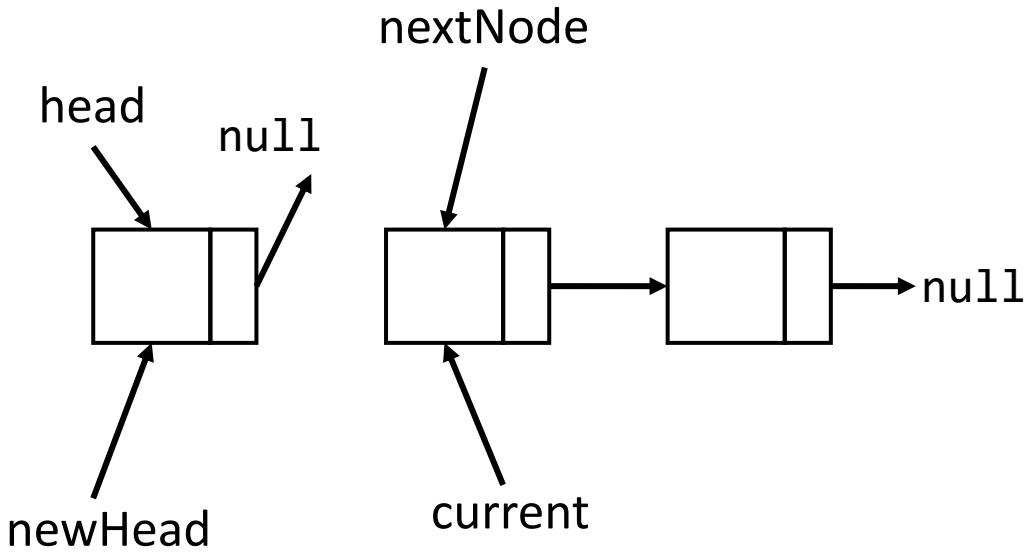
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



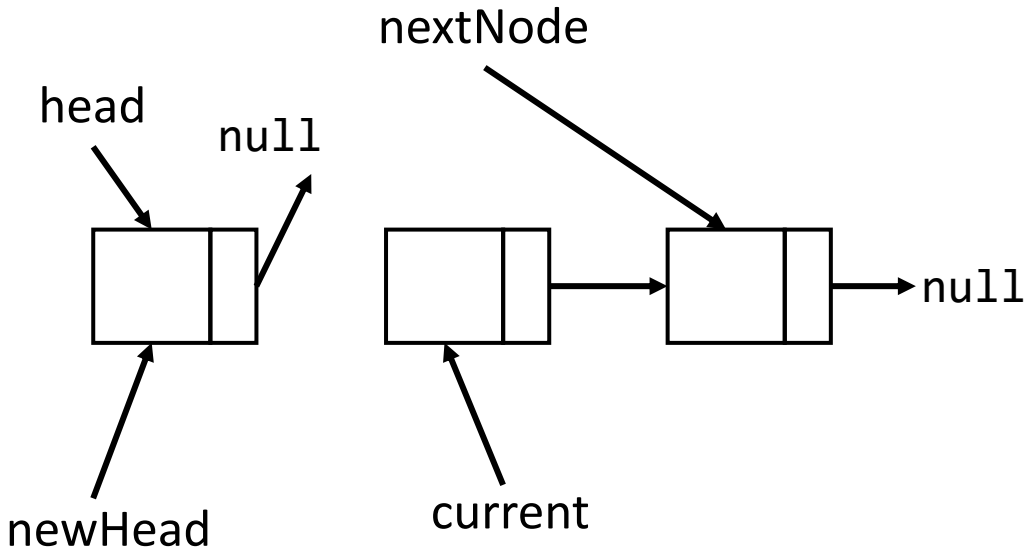
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



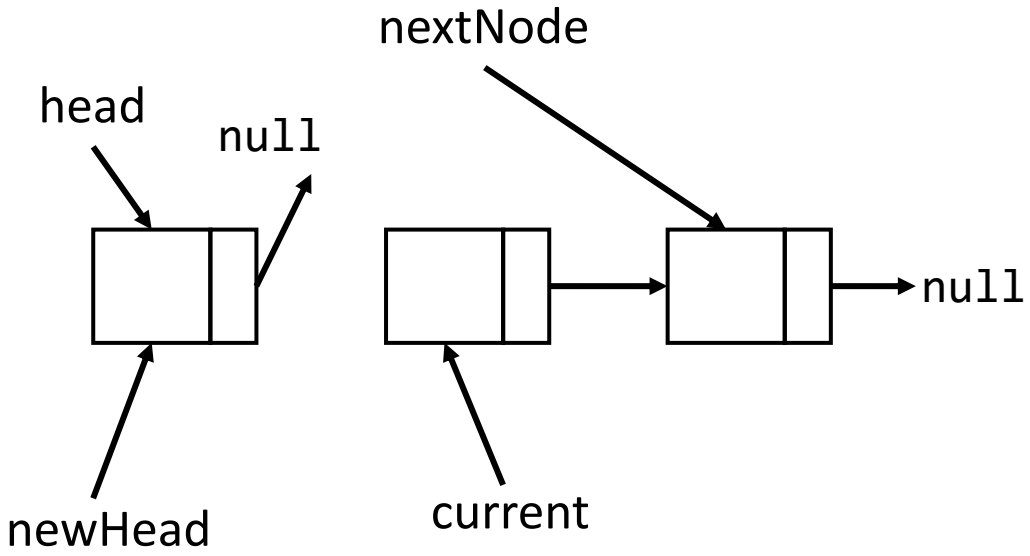
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



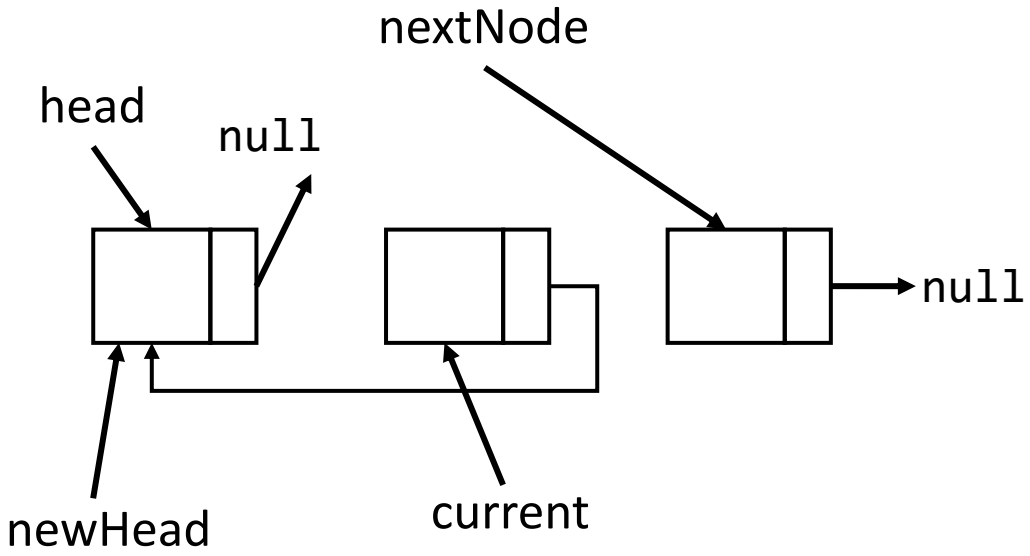
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



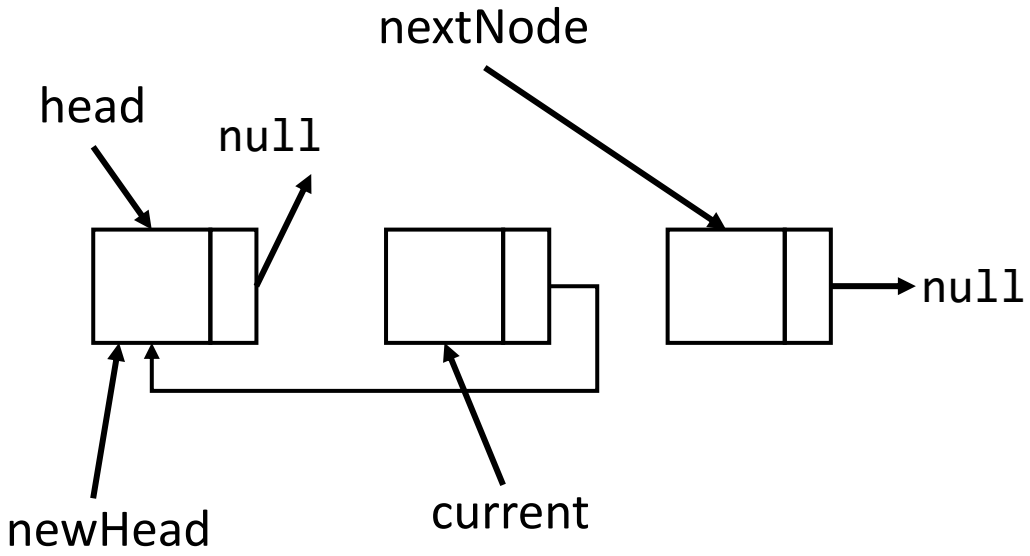
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



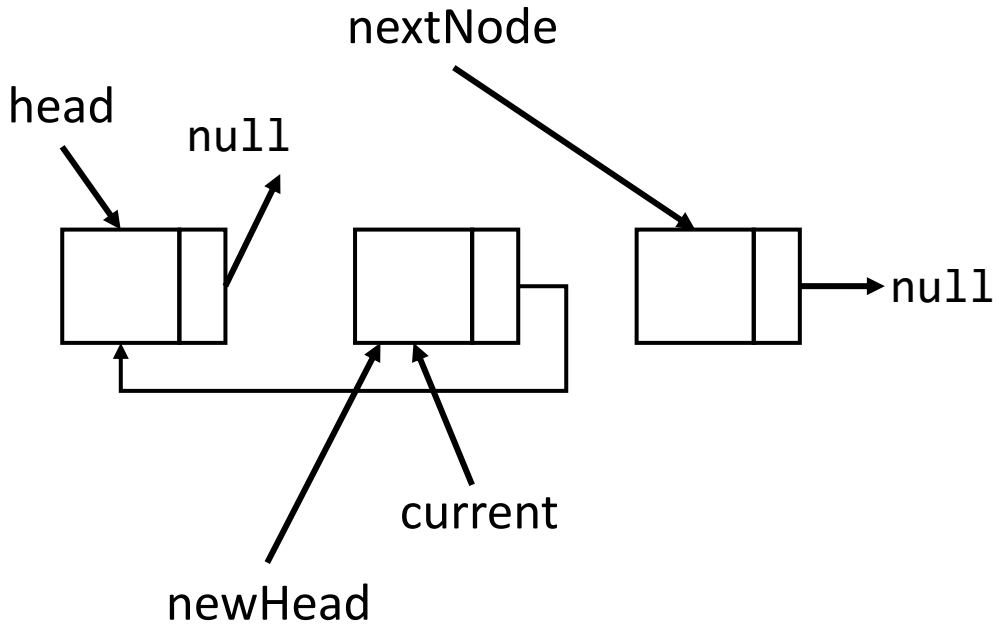
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



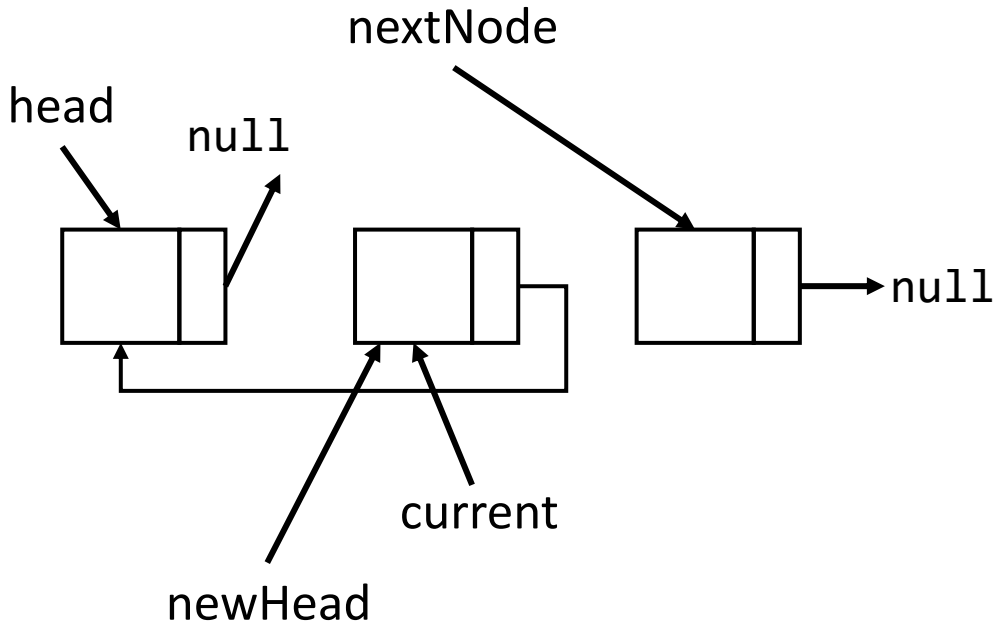
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



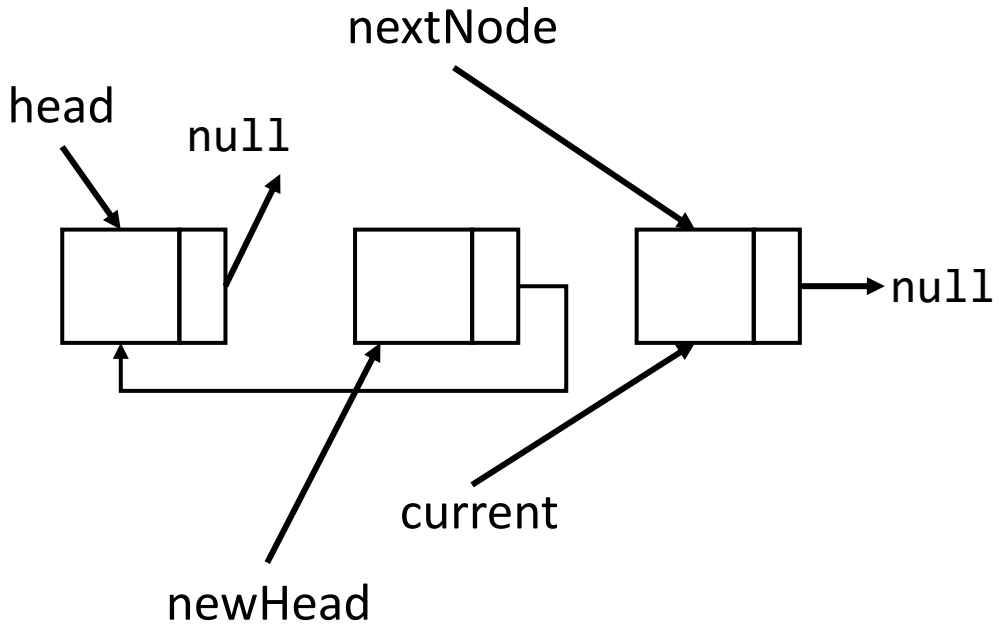
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



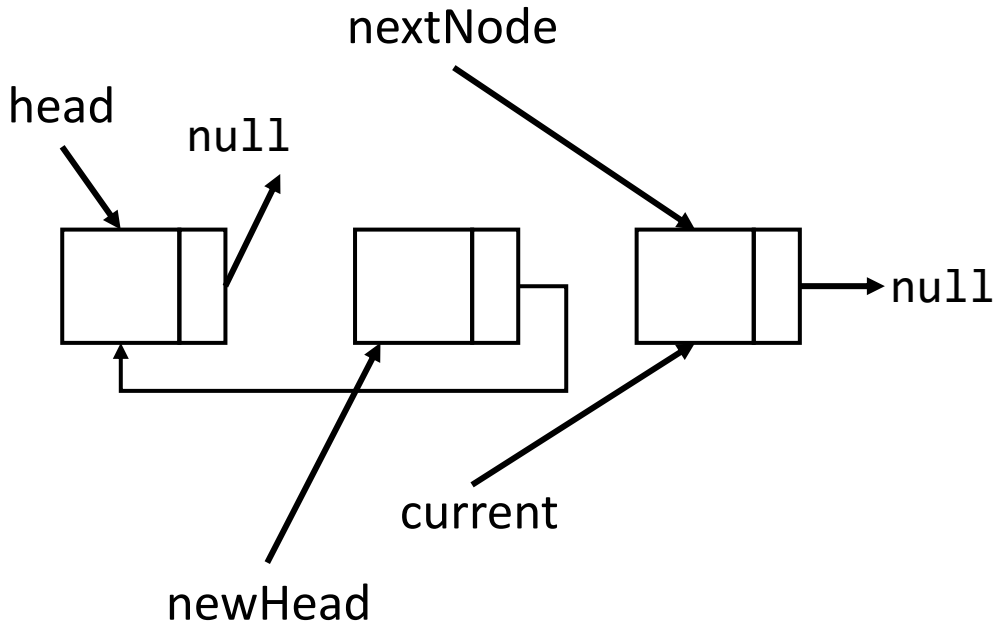
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



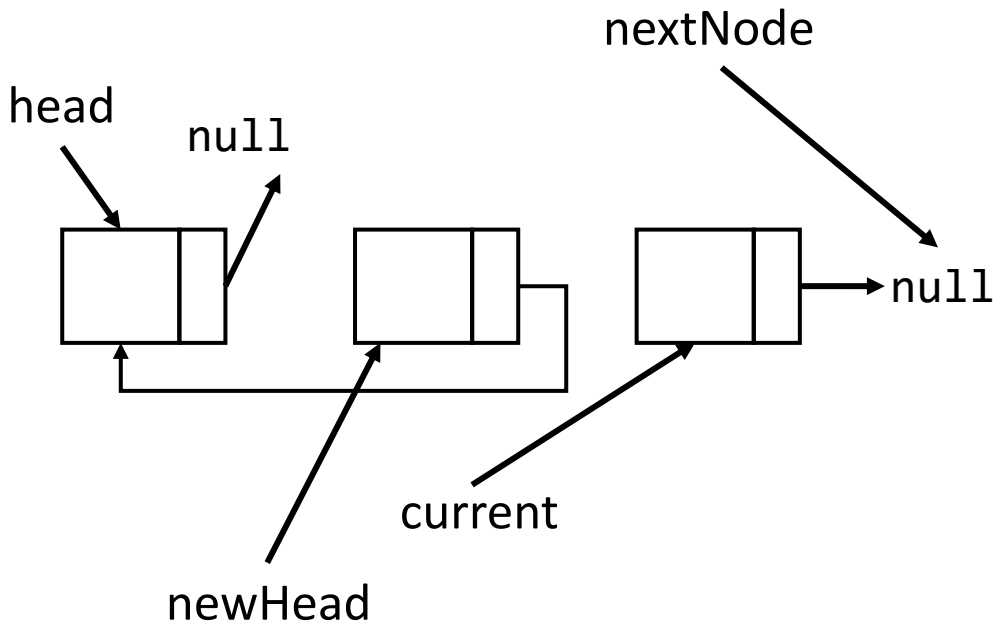
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



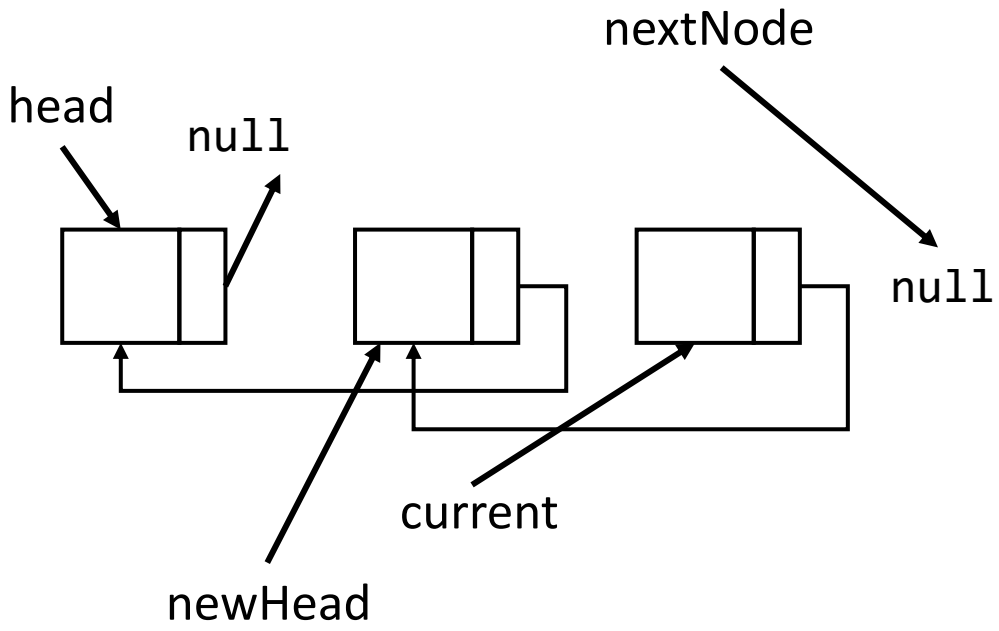
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



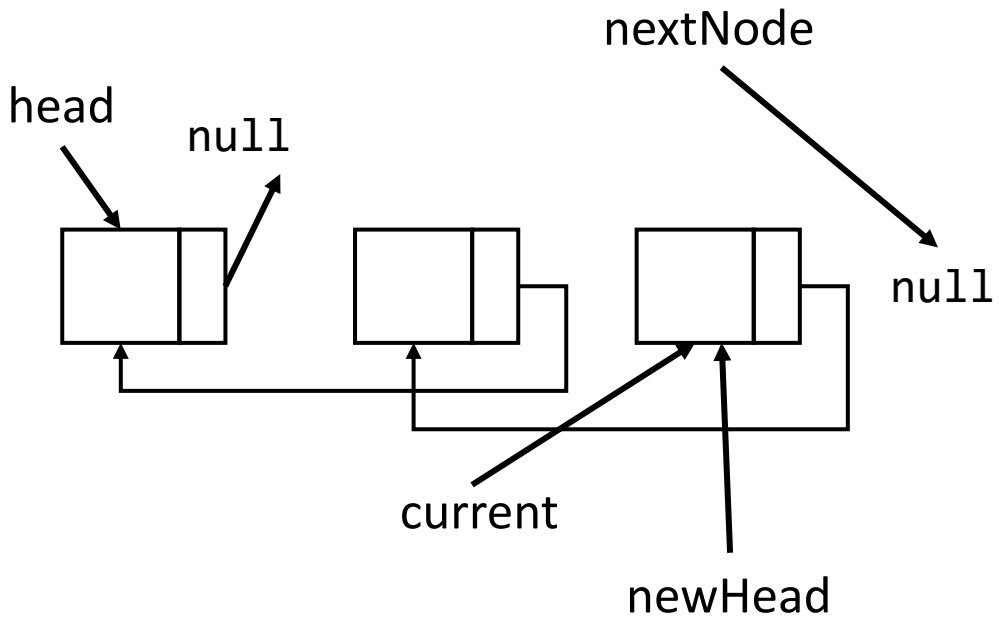
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



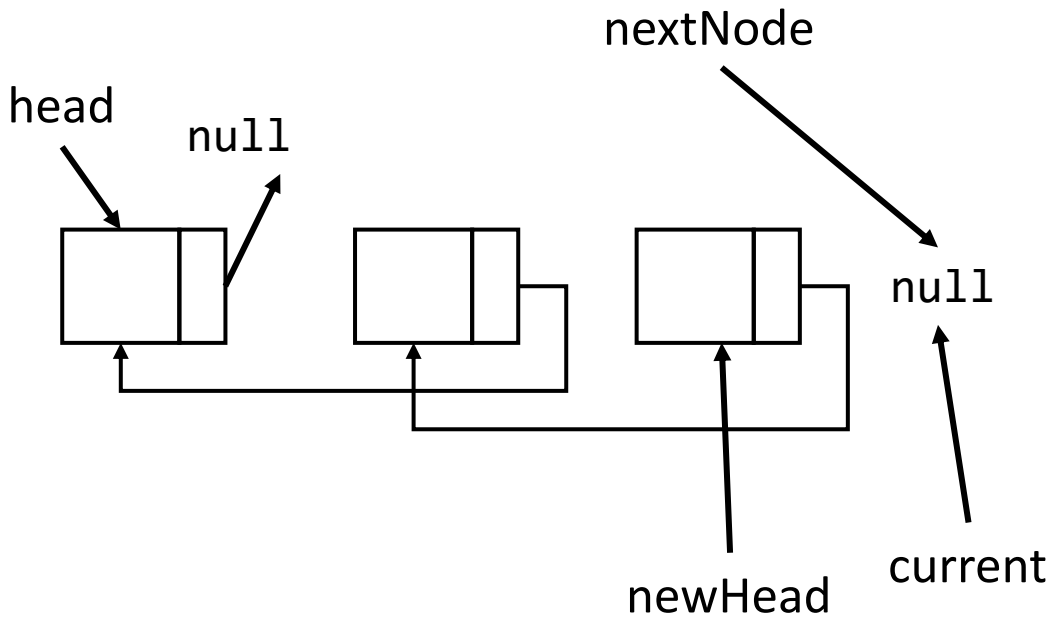
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



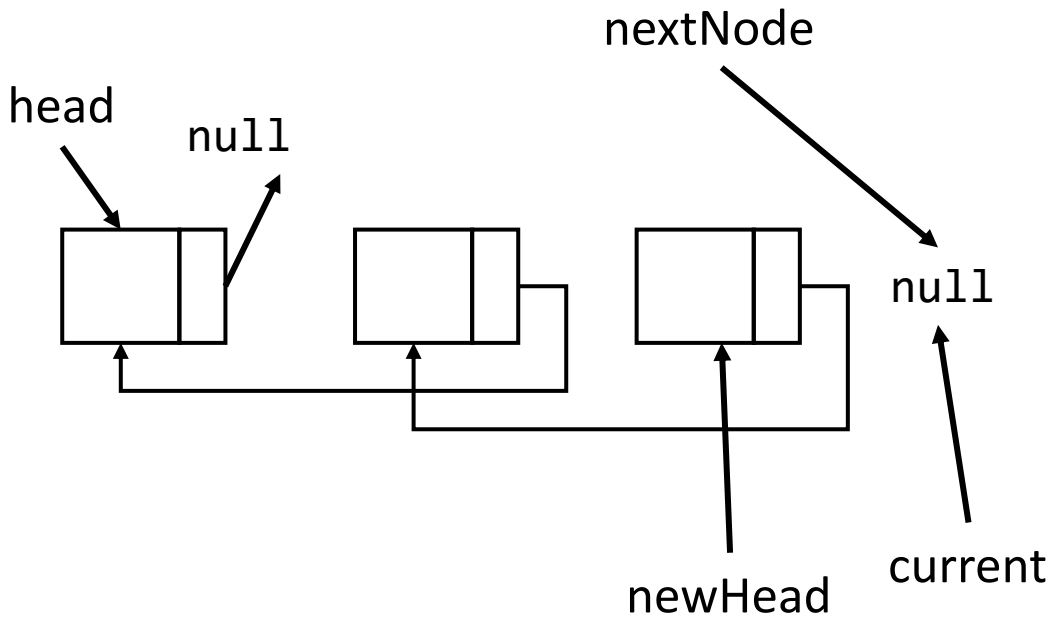
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



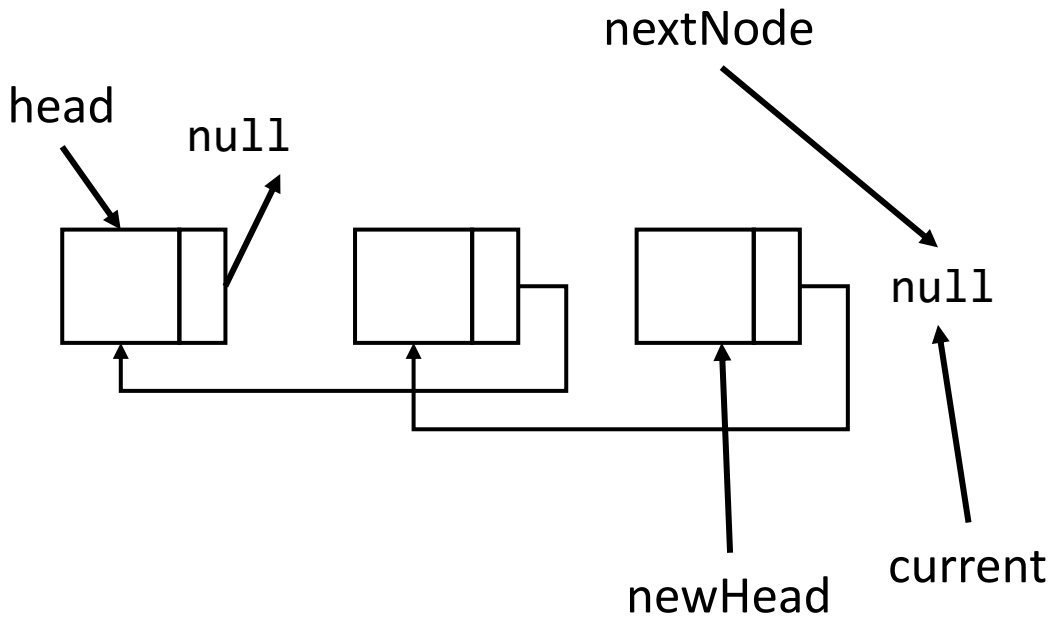
Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



Reversing a linked list

```
struct Node* reverse(struct Node* head)
{
    Node* newHead = nullptr;
    Node* current = head;
    while (current != nullptr)
    {
        Node* nextNode = current->next;
        current->next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```



Another way of linked list declaration

Declarations in Singly Linked Lists

NUST SEECS

- Declare two classes; one for nodes of the list while the other for access to the list

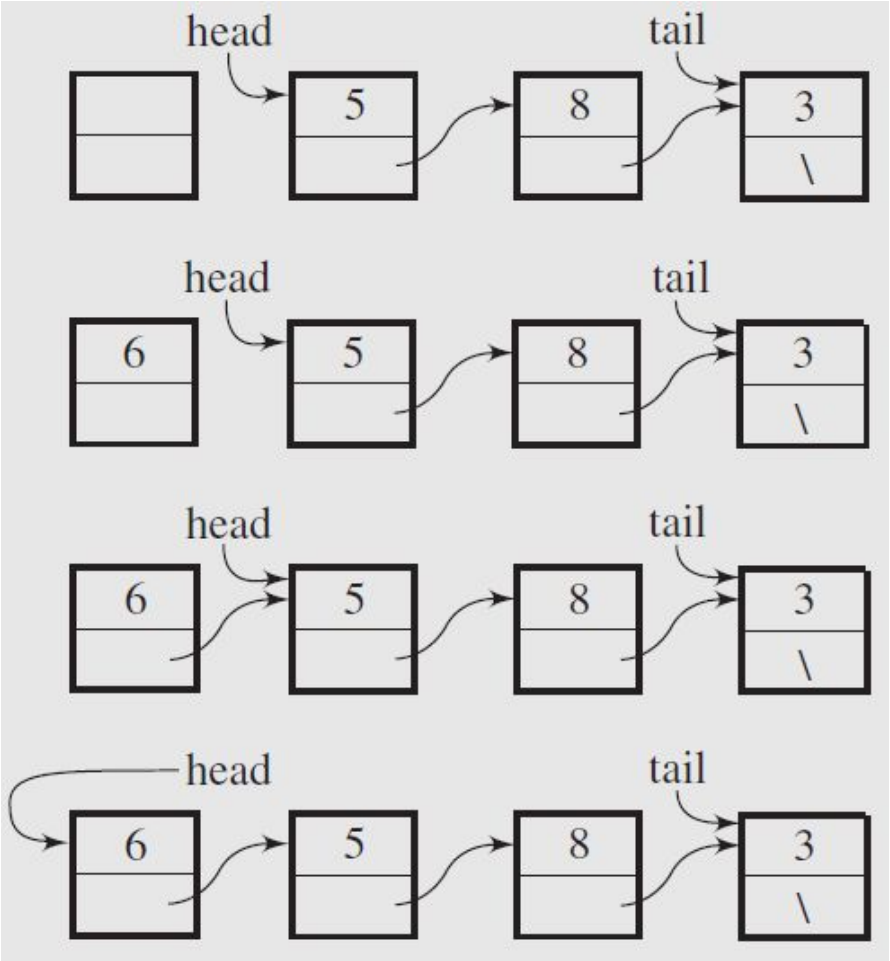
```
class Node {
public:
        Node() {
                next = 0;
        }
        Node(int i, Node *in = 0) {
                info = i; next = in;
        }
        int info;
        Node *next;
};
```

```
class AccessNode{
private:
        node *head, *tail;
Public:
        AccessNode() { head=tail=null;}
        int isEmpty() {return head==0;}
        void addToHead(int);
        void addToTail(int);
        int deleteFromHead();
        int deleteFromTail();
        void deleteNode(int);
        bool isInList(int) const;
        ~ AccessNode();}
```

Adding a node at the beginning

NUST SEECS

```
void AccessNode ::addToHead(int e1){
        head=new Node(e1, head);
        if(tail==0)
                tail=head;
}
```



Adding a node at tail

NUST SEECS

```
void AccessNode::addToTail(int el) {
if (tail != 0) { // if list not empty;
        tail->next = new Node(el);
        tail = tail->next;
}
else
        head = tail = new Node(el);
}
```