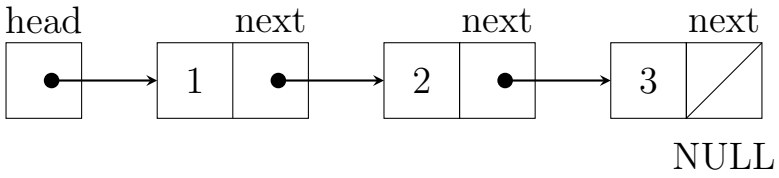
Linked Lists

Deletion

1 Linked Lists

What are *Linked Lists*?

Linked Lists, or simply *lists*, are a linear collection of self-referential objects (*nodes*), connected by pointer *links*. A linked list is often accessed via a pointer to the first node of the list. Subsequent nodes are accessed by the link-pointer member stored in each node.



Data are stored in a linked list dynamically—each node is created as necessary. A node can contain data of any type including objects of other classes (`structs`).

1.1 List Construction

Linked lists can be implemented in several different ways. Implementing lists as classes makes them very easy to use *once the code is functioning properly*. Creating list objects requires us to use some features of C++ that may possibly be new to you.

```
1 struct Node
2 {
3    int  info;
4    struct Node* next;     //  struct optional in C++
5 };
6
7 typedef struct Node*  NodePtr;
```

Basic list implementation (construction):

- Node as head of the list.
- Head pointer as the head of the list.

What is the difference?

Initially, it is easiest to implement lists as global variables using the technique that the *head* of the list is a head pointer. It is easy to convert this code into a class (as we shall see later).

1.2 List Operations

Common List Operations:

- Print
- Size/Length
- Insertion
- **Deletion**

The heart of all these operations is the ability to *traverse* (move through) a list.

1.2.1 Insertion

Where is the information to be inserted?

- Front
- Organized (sorted/middle)
- Back

Inserting an item on the front of a list is probably the easiest. All we have to do is attach the current list to the new node (which will become the front of the list).

Inserting an item on the end of a list is slightly more complicated. We have to move to the end of the list and attach the new node to the end.

Inserting an item in the middle of a list (in order) is the most complicated.

General operation

- Allocate new node
- Initialize new node
- Place new node in desired location

1.2.2 Deletion

Where is the node to be deleted?

- Front
- Back
- Match (or other condition)

Deleting an item from the front of a list is probably the easiest. All we have to do is move the head pointer to its next node (which will become the front of the list).

Deleting an item from the end of a list is a bit more complicated.

Deleting an item from the middle of a list is the most complicated.

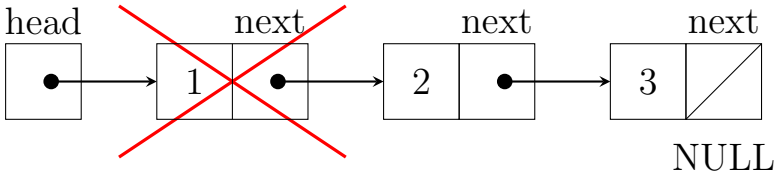
General operation

- Find node
- Update (break) links
- Release memory of deleted node

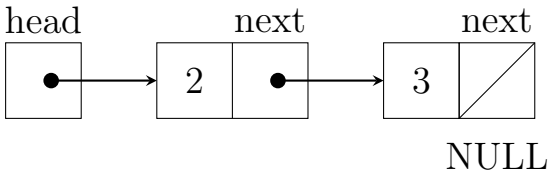
Remove item from front

Deleting the first node

Initial list:



After deleting first node:



Note: In some cases it is necessary to return the data stored in the node.

Naive[1] approach:

```
1 head = head->next;
```

What's wrong with this?

[1](of a person or action) showing a lack of experience, wisdom, or judgment.

Q: What's wrong with this?

A: Fails to:

- Handle empty list!
- Release memory.

1. Assign pointer to head of the list
2. Remove first node
   - `if` list is empty, do nothing
   - `else` assign head to next node in the list, then delete pointer to original head

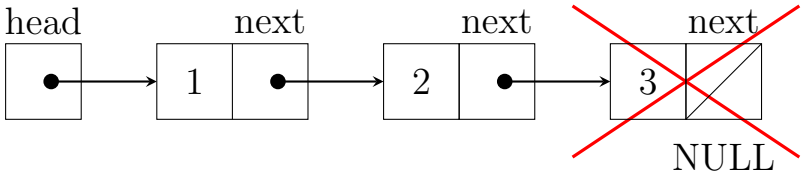
```
1  void RemoveFromFront()
2  {
3      NodePtr  p = head;
4
5      if( p == NULL )
6      {
7          //  do nothing
8      }
9      else
10     {
11         head = p->next;
12
13         p->next = NULL;
14
15         delete p;
16     }
17 }
```

Does it work if the list has only one node?

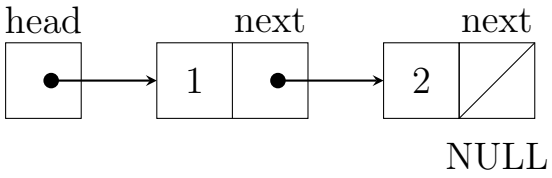
Remove node from end

Deleting the last node

Initial list:



After deleting last node:



Note: In some cases it is necessary to return the data stored in the node.

1. Assign pointer to head of the list
2. Remove last node
   - `if` list is empty, do nothing
   - `else` find last node in the list
     delete pointer to original end

```
1  void RemoveFromEnd()
2  {
3      if( head == NULL )
4      {
5          //  do nothing
6      }
7      else
8      {
9          NodePtr curr = head;
10         NodePtr prev = head;
11
12         while( curr->next != NULL )
13         {
14             prev = curr;
15             curr = curr->next;
16         }
17
18         prev->next = NULL;
19
20         delete curr;
21     }
22 }
```

Does it work if the list has only one node?

Shorter version—obscures behavior.

```
1  void RemoveFromEnd()
2  {
3      if( head != NULL )
4      {
5          NodePtr curr = head;
6          NodePtr prev = NULL;
7
8          while( curr->next != NULL )
9          {
10             prev = curr;
11             curr = curr->next;
12         }
13
14         if( prev == NULL )    // One node case!
15             head = NULL;
16         else
17             prev->next = NULL;
18
19         delete curr;
20     }
21 }
```

Switch to Objects/Classes...