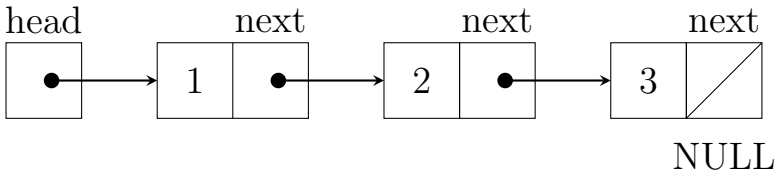
Linked Lists

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 Other Dynamic data structures

- Stacks
- Queues
- Trees (non-linear data structure)

These data structures are examined in detail in later lectures.

Saw a tweet (post?) awhile back by someone that said they were asked to list *all* data structures during a job interview.

1.2 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;
```

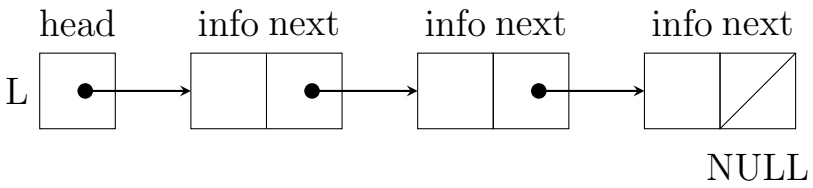
Basic list implementation (construction):

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

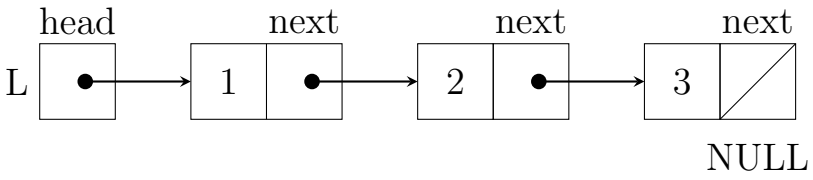
What is the difference?

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).

Singly-linked linear list



Singly-linked linear list (with values)



1.3 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.

1.3.1 Print

Let's consider how to print the contents of a list (defer explaining how the list was constructed for now). The key to printing the list is *visiting* (and printing the information) in each node in the list.

```
1  void PrintList()
2  {
3      NodePtr  p = head;
4
5      while( p != NULL )
6      {
7          cout << p->info << endl;
8          p = p->next;
9      }
10 }
```

1.3.2 Length/Size

Determining the length (or size) of a list is very similar to printing a list. We must count the number of nodes in the list. To do this, we must visit each node in the list. Do you see a *pattern*?

```
1  int ListLength()
2  {
3      int n = 0;
4      NodePtr p = head;
5
6      while( p != NULL )
7      {
8          n++;
9          p = p->next;
10     }
11
12     return n;
13 }
```

1.3.3 Insertion

Where is the information to be inserted?

- Front
- Organized (sorted/middle)
- Back

Inserting an item onto 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

Add item to front

1. Allocate new node
2. Initialize new node
3. Put node at the front
    - `if` list is empty, head is the new node
    - `else` attach head to the new node

```
1  void AddToFront( int x )
2  {
3      NodePtr  n = new Node;      //  Allocate
4
5      n->info = x;                //  Initialize
6      n->next = NULL;
7
8      if( head == NULL )          //  Place
9      {
10         head = n;
11     }
12     else
13     {
14         n->next = head;
15         head = n;
16     }
17 }
```

Shorter version—obscures behavior.

```
1  void AddToFront( int x )
2  {
3      NodePtr  n = new Node;      //  Allocate
4
5      n->info = x;                //  Initialize
6      n->next = NULL;
7
8      if( head != NULL )          //  Place
9      {
10         n->next = head;
11     }
12
13     head = n;
14 }
```

Even shorter version—behavior very obscure.

```
1 void AddToFront( int x )
2 {
3     NodePtr  n = new Node;
4
5     n->info = x;
6     n->next = head;
7
8     head = n;
9 }
```

Add item to end

1. Allocate new node
2. Initialize new node
3. Put node at the end
    - `if` list is empty, head is the new node
    - `else` attach new node to the end of the list

```
1  void AddToEnd( int x )
2  {
3      NodePtr  n = new Node;          //  Allocate
4
5      n->info = x;                    //  Initialize
6      n->next = NULL;
7
8      if( head == NULL )              //  Place
9      {
10         head = n;
11     }
12     else
13     {
14         NodePtr p = head;
15         while( p->next != NULL )
16             p = p->next;
17
18         p->next = n;
19     }
20 }
```