Structured Data

Linked Lists

CIS 15 : Spring 2007

Functionalia

HW4 Part B due this SUNDAY April 15st: 11:59pm

Today:

- Structured Data
- Dynamic Structures
- Linked Lists

Anatomy of a `struct`

Here is a structure (called `Time`) that contains 3 integers (`hour`, `minutes`, and `seconds`).

```
struct Time
{
   int hour;
   int minutes;
   int seconds;
};



Time now;
```

Anatomy of a `struct`

Here is a structure (called **`Time`**) that contains 3 integers (**`hour`**, **`minutes`**, and **`seconds`**).

```
struct Time
{
   int hour;
   int minutes;
   int seconds;
};

Time now;
```

Keyword **`struct`** to indicate that what follows is a **`struct`**.

Anatomy of a `struct`

Here is a structure (called **`Time`**) that contains 3 integers (**`hour`**, **`minutes`**, and **`seconds`**).

```
struct Time
{
   int hour;
   int minutes;
   int seconds;
};



Time now;
```

The *tag* (i.e. name of the new structured data-type you are defining).

Anatomy of a `struct`

Here is a structure (called **`Time`**) that contains 3 integers (**`hour`**, **`minutes`**, and **`seconds`**).

```
struct Time
{
   int hour;
   int minutes;
   int seconds;
};


Time now;
```

It is customary to always Capitalize the first letter of the name of a structure.

Anatomy of a `struct`

Here is a structure (called `Time`) that contains 3 integers (`hour`, `minutes`, and `seconds`).

```
struct Time
{
   int hour;
   int minutes;
   int seconds;
};
```

The member data types are contained in curly braces (just like functions)

```
Time now;
```

Anatomy of a `struct`

Here is a structure (called `Time`) that contains 3 integers (`hour`, `minutes`, and `seconds`).

```
struct Time
{
   int hour;
   int minutes;
   int seconds;
};
```

NOTE! There is a semi-colon at the end of the `struct` (*unlike* functions)

```
Time now;
```

Anatomy of a `struct`

Here is a structure (called **`Time`**) that contains 3 integers (**`hour`**, **`minutes`**, and **`seconds`**).

```
struct Time
{
    int hour;
    int minutes;
    int seconds;
};
```

Variables are declared like in any function.

```
Time now;
```

Anatomy of a `struct`

Here is a structure (called `Time`) that contains 3 integers (`hour`, `minutes`, and `seconds`).

```
struct Time
{
   int hour;
   int minutes;
   int seconds;
};
```

```
Time now;
```

Your structure can be declared now as any other variable.

Anatomy of a `struct`

Here is a structure (called `Time`) that contains 3 integers (`hour`, `minutes`, and `seconds`).

```
struct Time
{
   int hour;
   int minutes;
   int seconds;
};
```

This is the **definition**. (Only one).

This is the **instantiation**. (Can be many).

```
Time now;
Time later;
```

Definition of `structs`

Typically `struct`'s are defined outside of any functions, and at the top of the program (i.e. near the prototypes). Why?

```
struct Date
{
   int day, month, year;
   char longName[255];
};

int main() {
   Date current;


}
```

Definition of `structs`

Note the mixture of data-types, and using one line for the integers.

```
struct Date
{
   int day, month, year;
   char longName[255];
};

int main() {
   Date current;


}
```

Accessing the members of a `struct`

You can access the members of a `struct` variable through **dot-notation**.

```
struct Date
{
   int day, month, year;
   char longName[255];
};

int main() {
   Date current;
   current.day = 26;
   current.month = 3;
   current.year = 2007;
   strcpy(current.longName, “Bangladesh - Independence Day”);
}
```

Arrays of Structs

```
struct BookInfo
{
   char title[50];
   char author[30];
   char publisher[25];
   double price;
};

BookInfo bookList[20]; // creates 20 BookInfo's


cout << bookList[10].title << endl;
cout << bookList[10].title[0] << endl;
```

What's the difference?

Structs used in Functions

House h is a **copy**

```
void showHouse(House h)
{
   cout << h.footprint.length << " by "
        << h.footprint.width << " feet and, "
        << h.footprint.height << "high" << endl;
}
```

House h is a ***reference to the original***

```
void showHouse(House &h)
{
   cout << h.footprint.length << " by "
        << h.footprint.width << " feet and, "
        << h.footprint.height << "high" << endl;
}
```

Structs used in Functions

a copy of the whole House is returned

```
House buildHouse(int length, int width, int height)
{
        House h;
        h.footprint.length = length;
        h.footprint.width = width;
        h.height = height;
        return h;
}
```

Allows you to return ***more than one value*** from your function!

Pointers to structs

```
struct Rectangle
{
   int length;
   int height;
};

Rectangle * rPtr;
Rectangle rect = {20, 40};

rPtr = &rect;
```

Using `rPtr` how does one access the length and the width of the Rectangle?

Pointers to structs

```
struct Rectangle
{
   int length;
   int height;
};

Rectangle * rPtr;
Rectangle rect = {20, 40};

rPtr = &rect;

cout << *rPtr.length << endl;
```

Dot Notation has higher precedence than the indirection operator (*)!

DOES NOT WORK!

Pointers to structs

```
struct Rectangle
{
   int length;
   int height;
};

Rectangle * rPtr;
Rectangle rect = {20, 40);

rPtr = &rect;

cout << (*rPtr).length << endl;
```

WORKS... but it is unwieldy!

Pointers to structs

```
struct Rectangle
{
   int length;
   int height;
};

Rectangle * rPtr;
Rectangle rect = {20, 40);

rPtr = &rect;

cout << rPtr->length << endl;
```

-> is the Structure Pointer Operator

Structures can be Dynamically Allocated

```
Rectangle * rect;
rect = new Rectangle;

rect->length = 20;
rect->height = 30;
```

```
Rectangle * manyRects;
manyRects = new Rectangle[5];

for(int i = 0; i < 5; i++)
{
   manyRects[i].length = 0;
   manyRects[i].height = 0;
}
```

Structs can Contain Dynamically Allocated Memory

```
struct Numbers
{
   int uno;
   char * dos;
   double * tres;
};
```

```
Numbers n1;
n1.dos = new char;
n1.tres = new double;

n1.uno = 1;
```

Access dos and tres?

Structs can Contain Dynamically Allocated Memory

```
struct Numbers
{
   int uno;
   char * dos;
   double * tres;
};
```

```
Numbers n1;
n1.dos = new char;
n1.tres = new double;

n1.uno = 1;
*n1.dos = '2';
*(n1.tres) = 3.33;
```

Dynamically Created Structs can Contain Dynamically Allocated Memory

```
struct Numbers
{
   int uno;
   char * dos;
   double * tres;
};
```

```
Numbers * nPtr = new Numbers;
```

Access uno, dos, and tres?

Dynamically Created Structs can Contain Dynamically Allocated Memory

```
struct Numbers
{
   int uno;
   char * dos;
   double * tres;
};
```

```
Numbers * nPtr = new Numbers;
nPtr->dos = new char;
nPtr->tres = new double;

(*nPtr).uno = 1;
```

Dynamically Created Structs can Contain Dynamically Allocated Memory

```
struct Numbers
{
   int uno;
   char * dos;
   double * tres;
};
```

```
Numbers * nPtr = new Numbers;
nPtr->dos = new char;
nPtr->tres = new double;

nPtr->uno = 1;
```

Same thing

Dynamically Created Structs can Contain Dynamically Allocated Memory

```
struct Numbers
{
   int uno;
   char * dos;
   double * tres;
};
```

```
Numbers * nPtr = new Numbers;
nPtr->dos = new char;
nPtr->tres = new double;

nPtr->uno = 1;
*nPtr->dos = '2';
```

Pointer to Struct is dereferenced, along with pointer (dos) to the char.

Dynamically Created Structs can Contain Dynamically Allocated Memory

```
struct Numbers
{
   int uno;
   char * dos;
   double * tres;
};
```

```
Numbers * nPtr = new Numbers;
nPtr->dos = new char;
nPtr->tres = new double;

nPtr->uno = 1;
*nPtr->dos = '2';
*(*nPtr).tres = 3.33;
```

Same thing (different way of writing it).

Dynamically Created Structs can Contain Dynamically Allocated Memory

```
struct Numbers
{
   int uno;
   char * dos;
   double * tres;
};
```

```
Numbers * nPtr = new Numbers;
nPtr->dos = new char;
nPtr->tres = new double;

nPtr->uno = 1;
*nPtr->dos = ‘2’;
*nPtr->tres = 3.33;
```

Dynamically Created Structs can Contain Dynamically Allocated Memory

```
struct Numbers
{
   int uno;
   char * dos;
   double * tres;
};
```

```
Numbers * nPtr = new Numbers;
nPtr->dos = new char;
nPtr->tres = new double;
```

How does one go about deleting all of this memory?

Dynamically Created Structs can Contain Dynamically Allocated Memory

```
struct Numbers
{
   int uno;
   char * dos;
   double * tres;
};
```

```
Numbers * nPtr = new Numbers;
nPtr->dos = new char;
nPtr->tres = new double;

delete nPtr;
```

?

Dynamically Created Structs can Contain Dynamically Allocated Memory

```
struct Numbers
{
   int uno;
   char * dos;
   double * tres;
};
```

```
Numbers * nPtr = new Numbers;
nPtr->dos = new char;
nPtr->tres = new double;

delete nPtr->dos;
delete nPtr->tres;
delete nPtr;
nPtr = NULL;
```

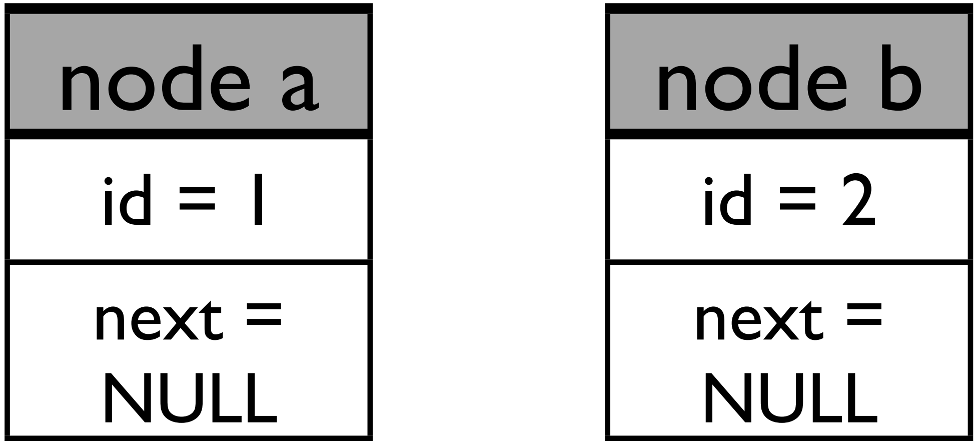
Need to delete everything that was allocated !
(in the correct order)

Set the unused Pointer to NULL!

Dynamically Linked Structs

From the Homework:

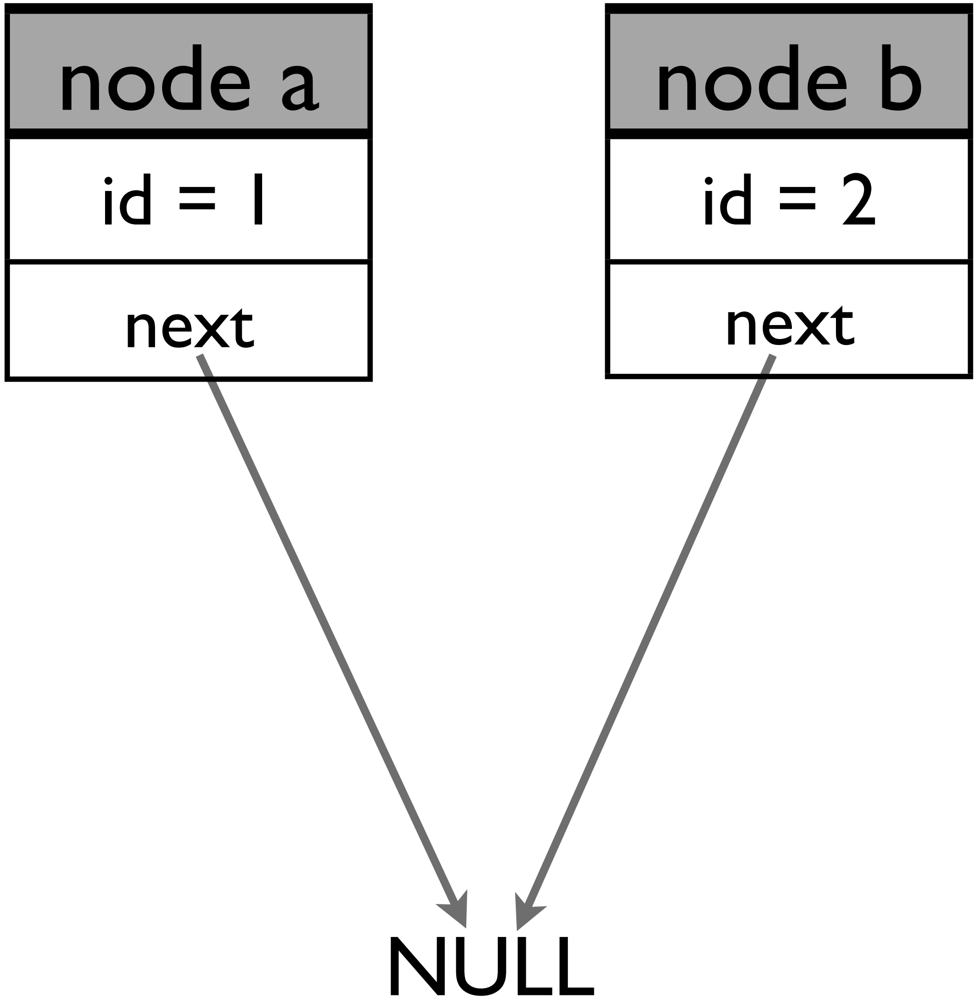
```
struct node {
   int id;
   node * next;
};

node a = {1, NULL};
node b = {2, NULL};
```



Dynamically Linked Structs

From the Homework:

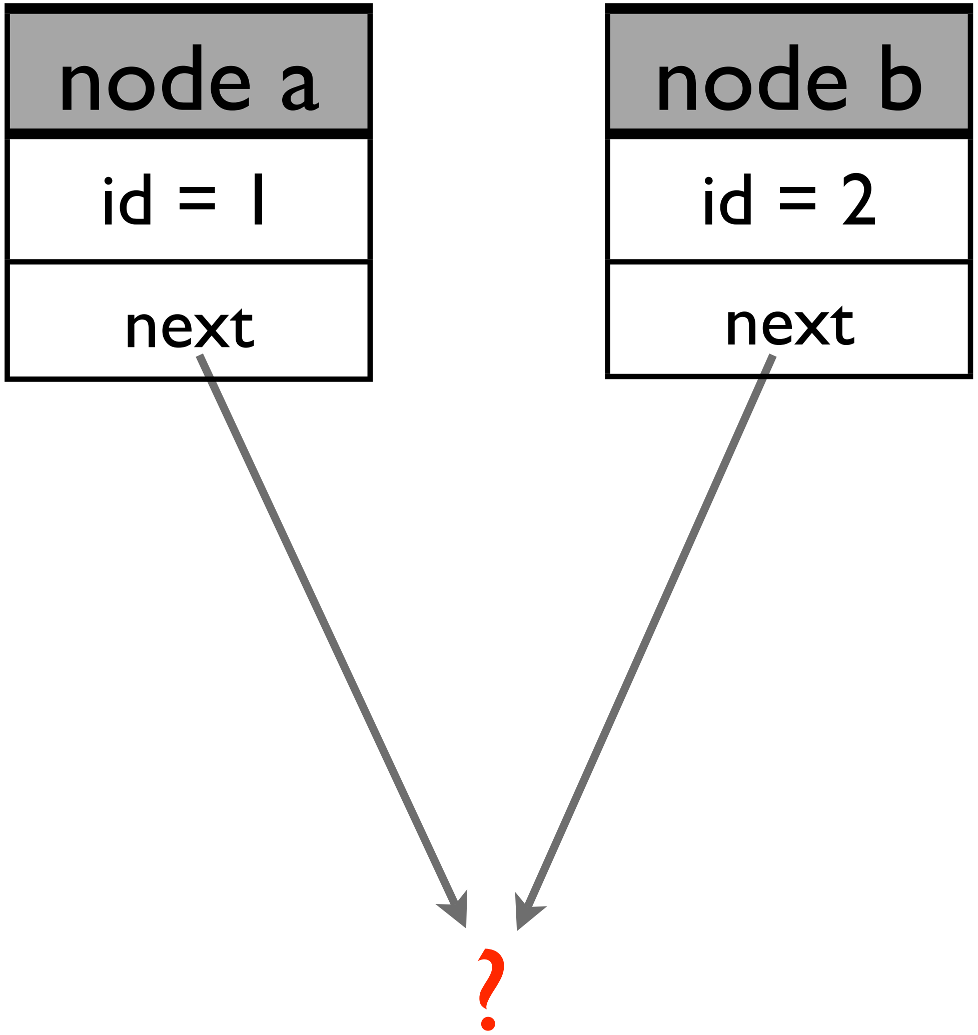
```
struct node {
   int id;
   node * next;
};

node a = {1, NULL};
node b = {2, NULL};
```



Dynamically Linked Structs

From the Homework:

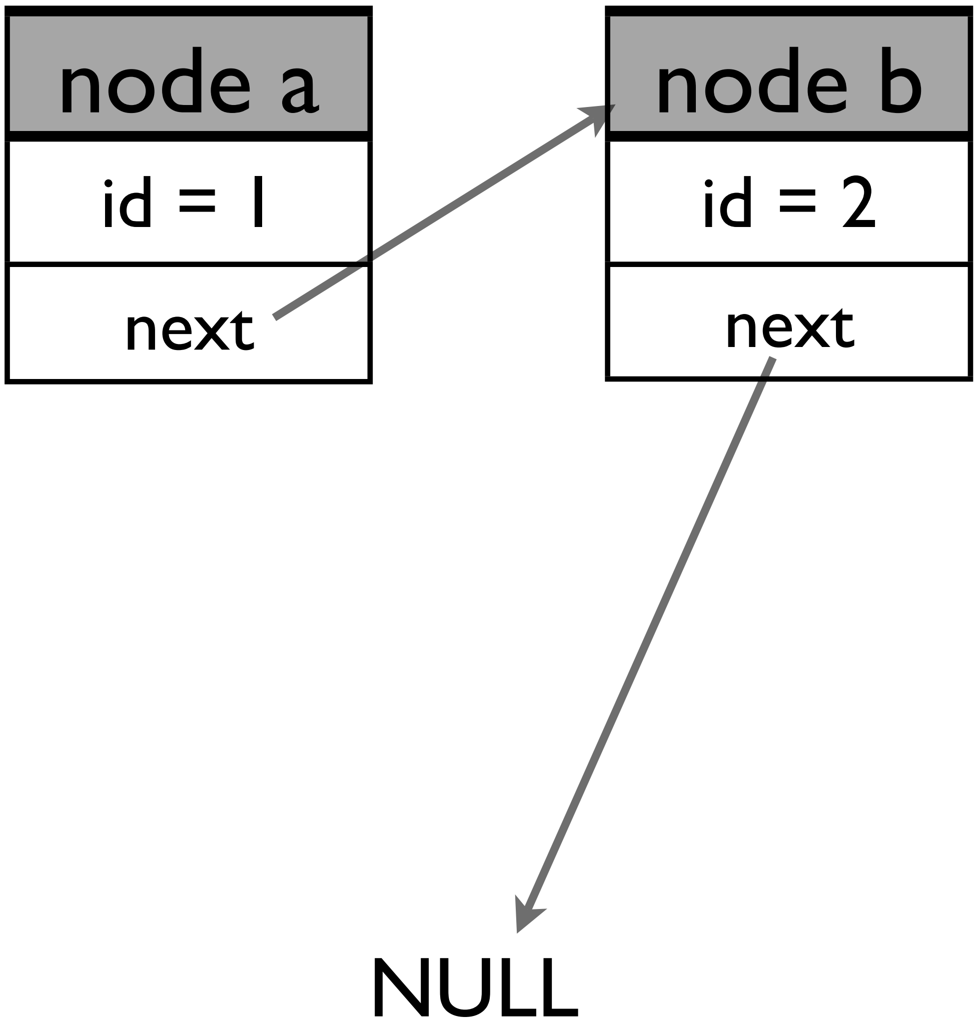
```
struct node {
   int id;
   node * next;
};

node a = {1, NULL};
node b = {2, NULL};
a.next = &b;
```



Dynamically Linked Structs

From the Homework:

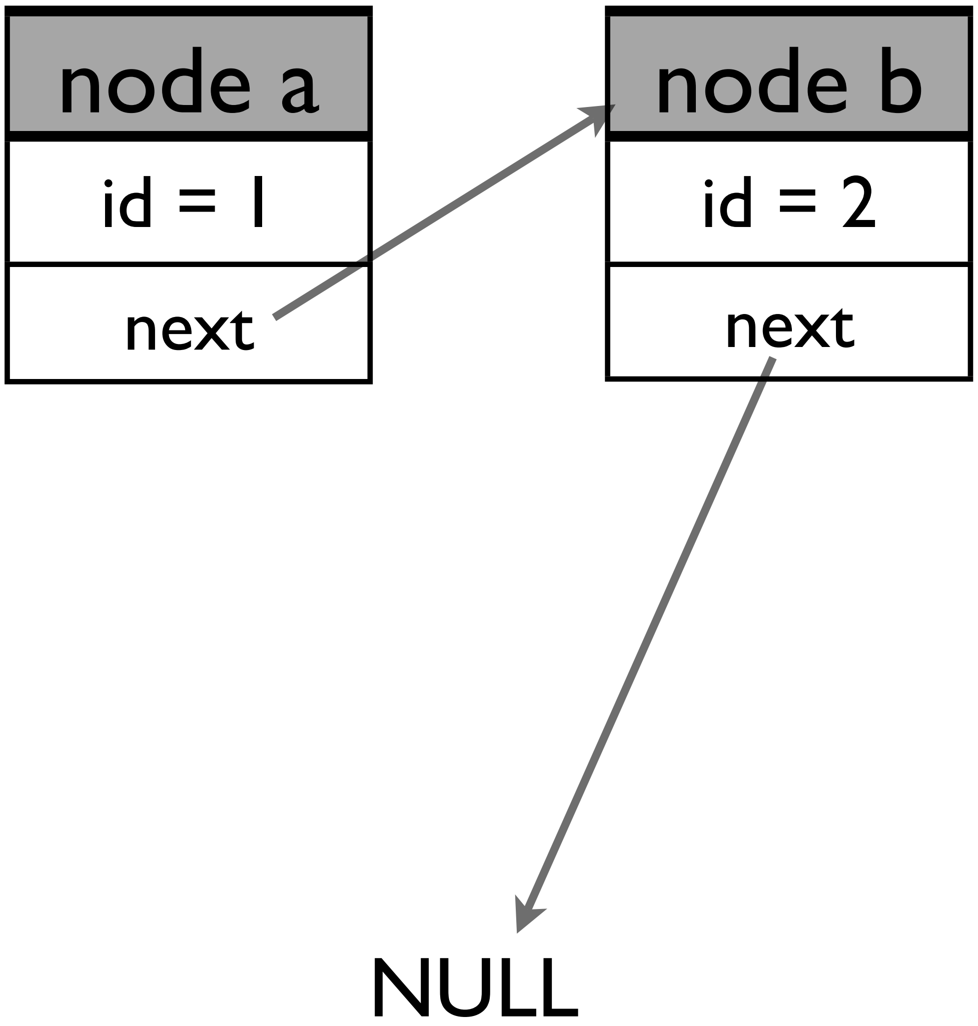
```
struct node {
   int id;
   node * next;
};

node a = {1, NULL};
node b = {2, NULL};
a.next = &b;
```



Dynamically Linked Structs

From the Homework:

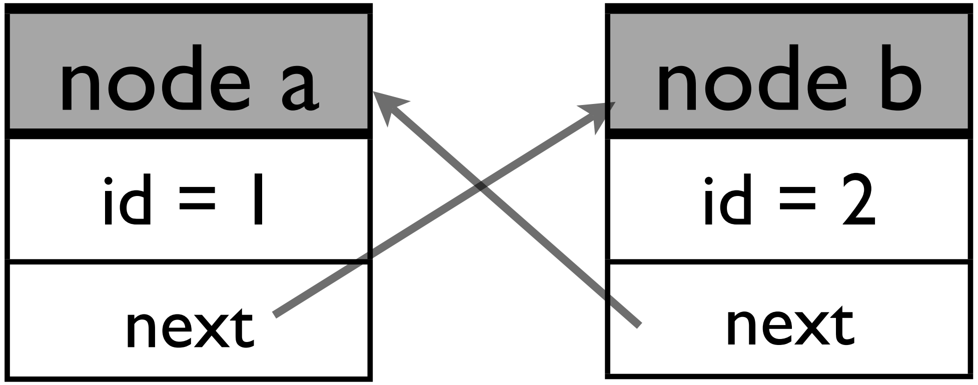
```
struct node {
   int id;
   node * next;
};

node a = {1, NULL};
node b = {2, NULL};
a.next = &b;
b.next = &a;
```



Dynamically Linked Structs

From the Homework:

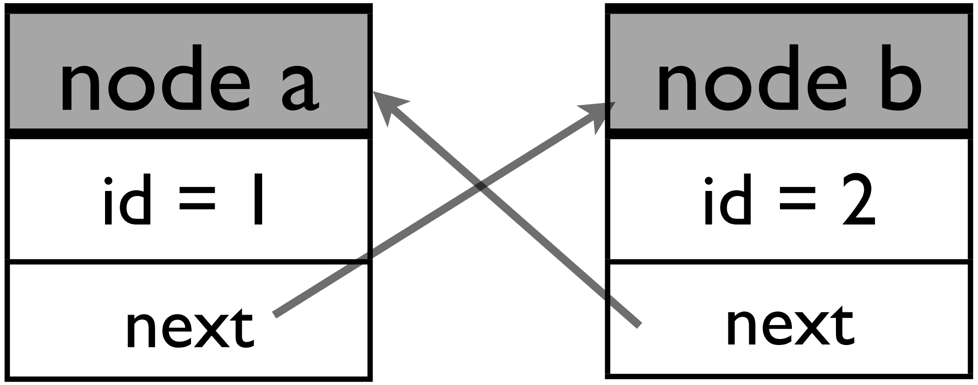
```
struct node {
   int id;
   node * next;
};

node a = {1, NULL};
node b = {2, NULL};
a.next = &b;
b.next = &a;
```



Dynamically Linked Structs

From the Homework:

```
struct node {
   int id;
   node * next;
};
```




```
node a = {1, NULL};
node b = {2, NULL};
a.next = &b;
b.next = &a;
cout << a.id << " " << a.next->id;
```

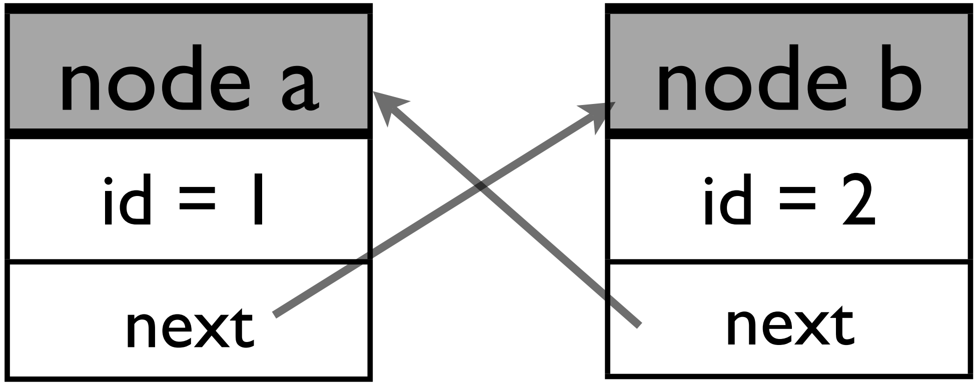
What gets printed out?

NULL

Dynamically Linked Structs

From the Homework:

```
struct node {
   int id;
   node * next;
};

node a = {1, NULL};
node b = {2, NULL};
a.next = &b;
b.next = &a;
cout << a.id << " " << a.next->id;
```



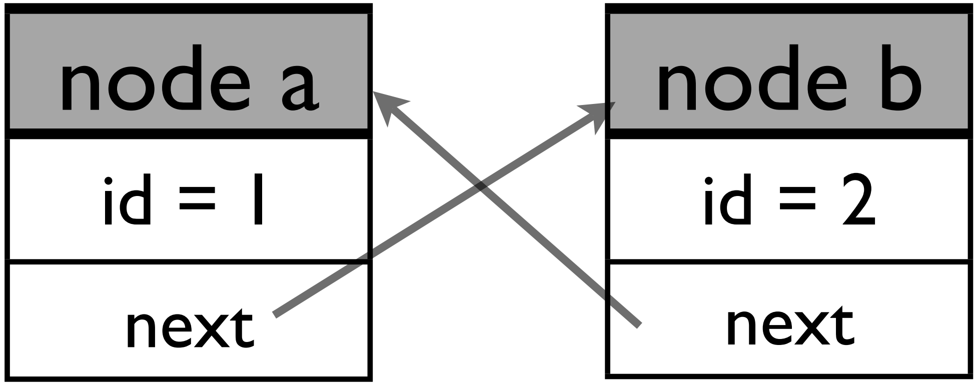
1 2

NULL

Dynamically Linked Structs

From the Homework:

```
struct node {
   int id;
   node * next;
};

node a = {1, NULL};
node b = {2, NULL};
a.next = &b;
b.next = &a;
cout << a.next->next->id;
```



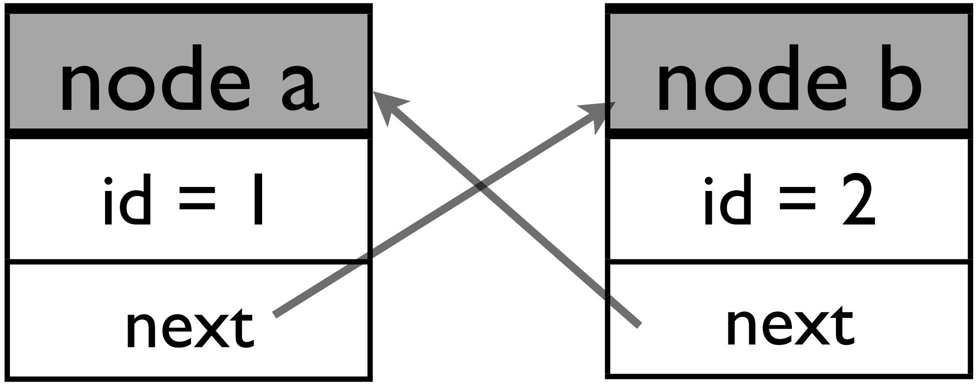
What about now?

NULL

Dynamically Linked Structs

From the Homework:

```
struct node {
   int id;
   node * next;
};

node a = {1, NULL};
node b = {2, NULL};
a.next = &b;
b.next = &a;
cout << a.next->next->id;
```



Back to 1

NULL

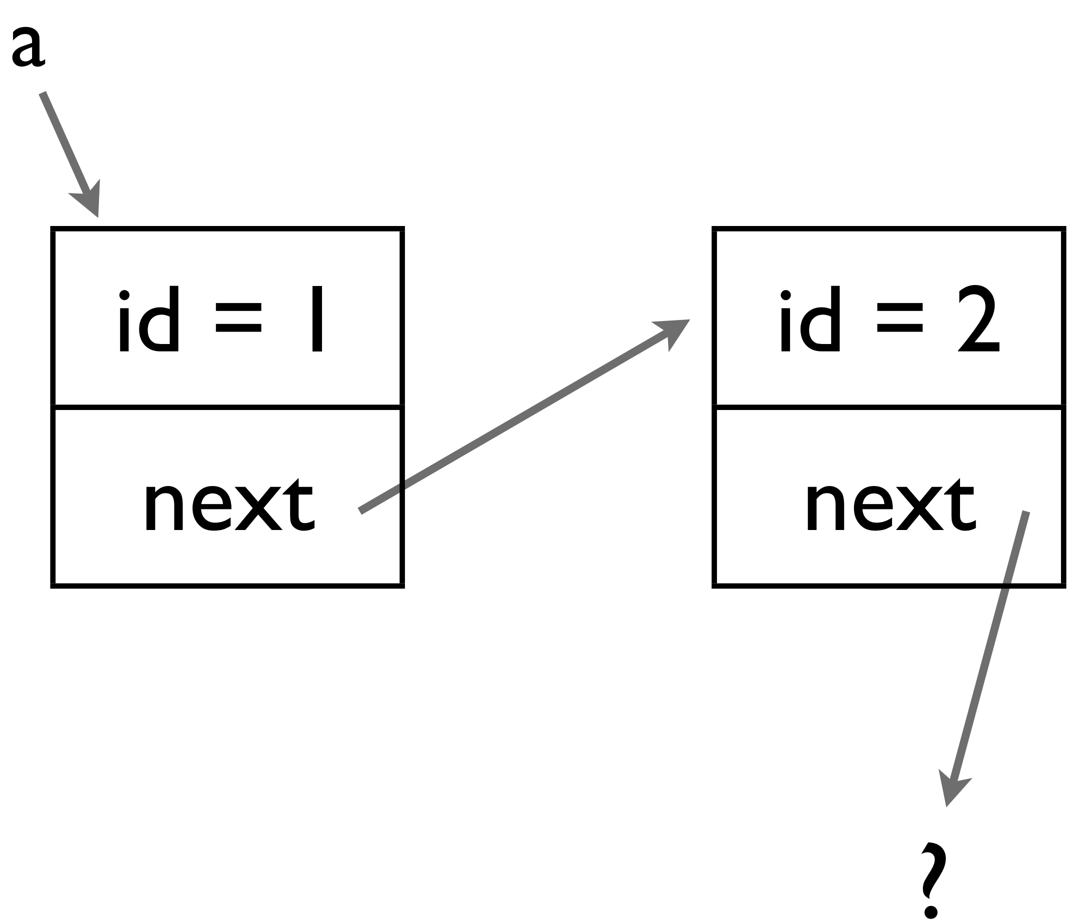
Dynamically Allocate the nodes

```
struct node {
   int id;
   node * next;
};

node * a;

a = new node;
a->id = 1;
a->next = new node;
a->next->id = 2;
```

What does this look like?

Dynamically Allocate the nodes

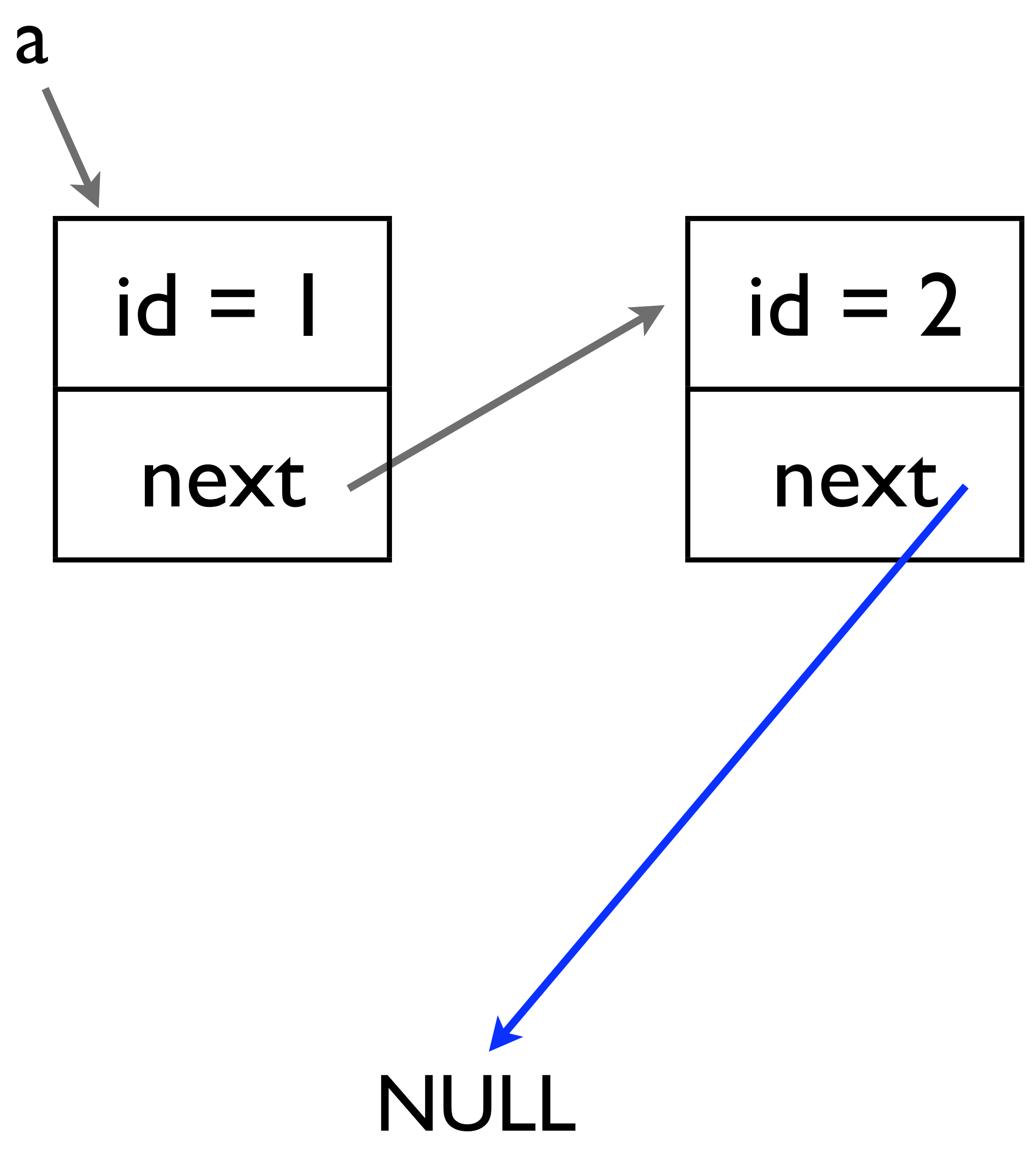
```
struct node {
   int id;
   node * next;
};

node * a;

a = new node;
a->id = 1;
a->next = new node;
a->next->id = 2;
```



NULL

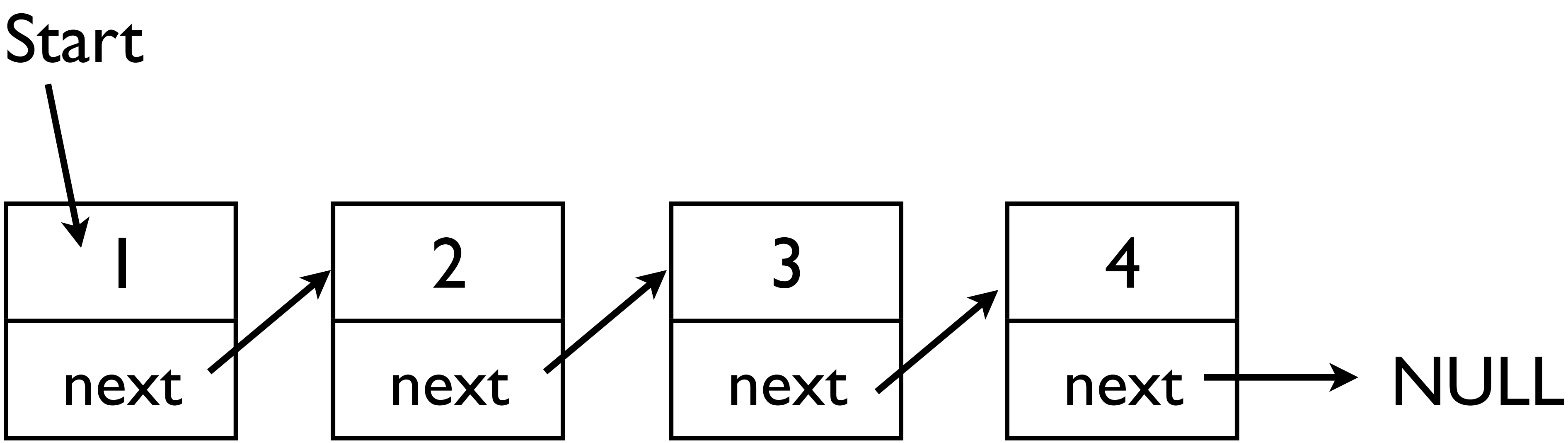
Dynamically Allocate the nodes

```
struct node {
   int id;
   node * next;
};

node * a;

a = new node;
a->id = 1;
a->next = new node;
a->next->id = 2;
a->next->next = NULL;
```



Linked List

A series of dynamically linked structs (i.e. nodes) is called a **Linked List.**

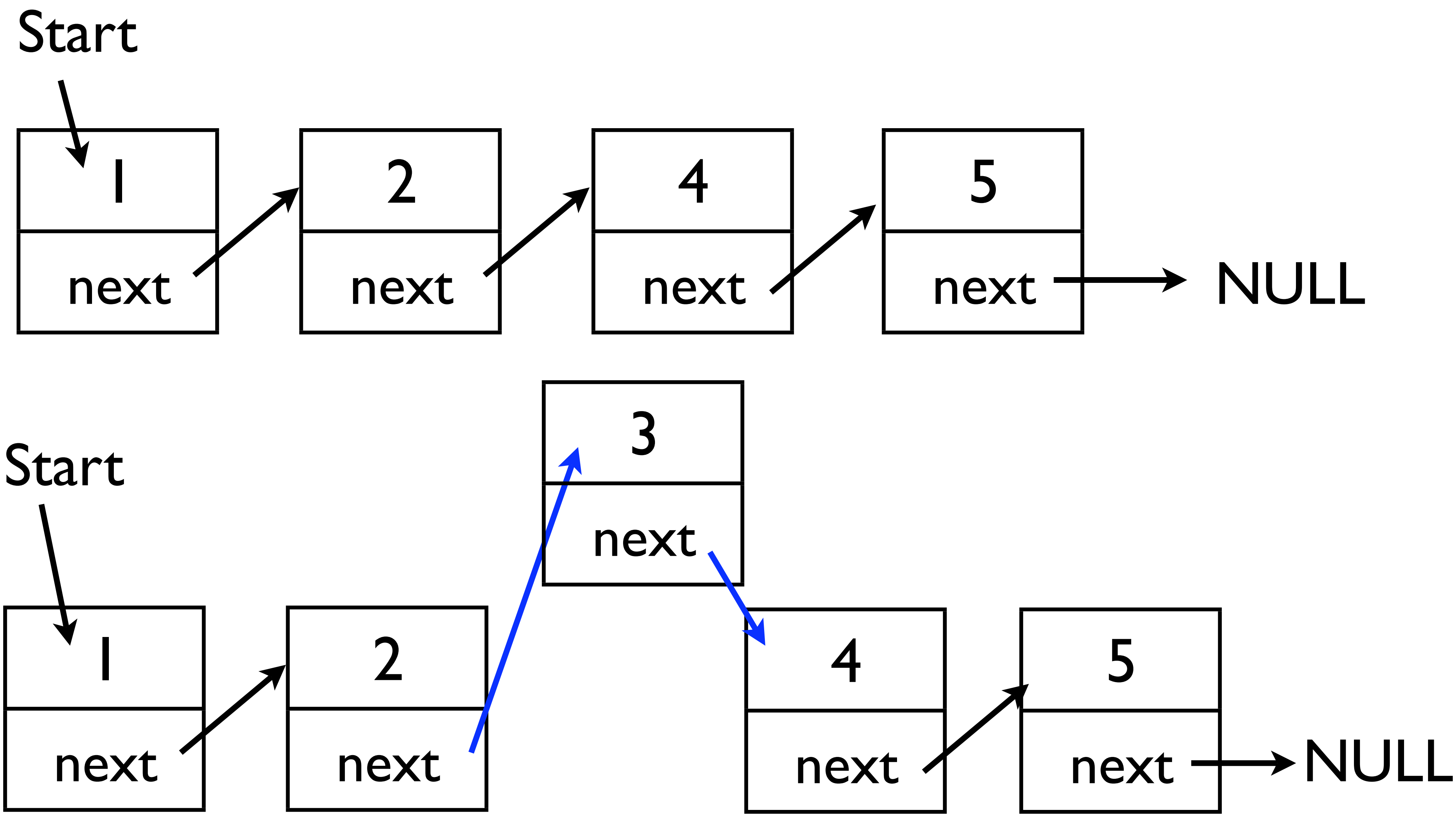


Dynamically Linked Structs (and in C++ Objects & Classes) is the basis for more Advanced Data Structures (i.e. **Linked Lists**)

Linked List

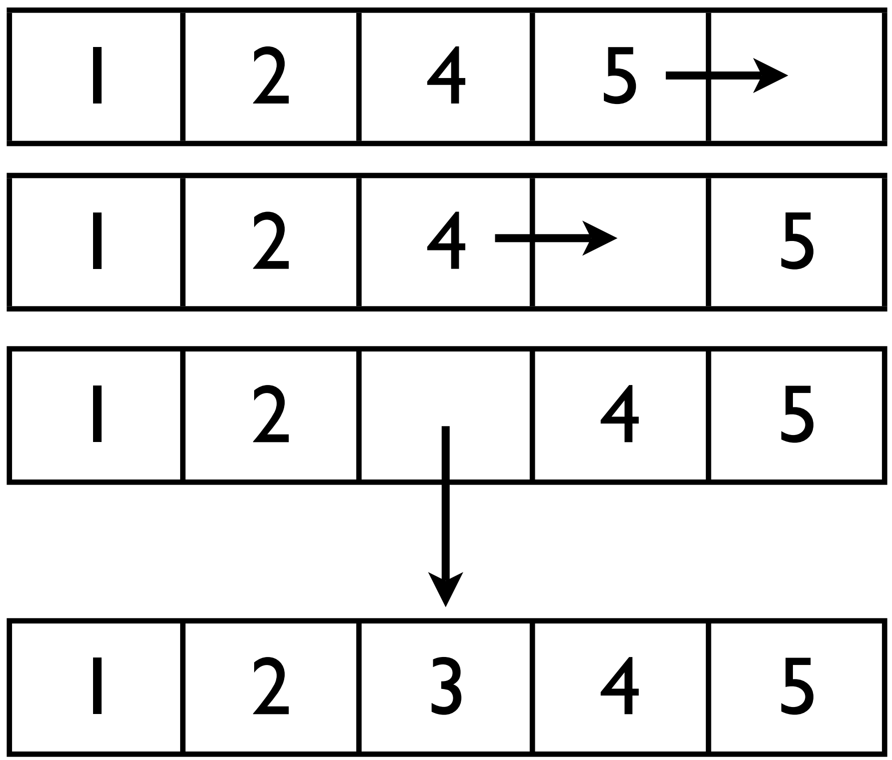
Linked Lists can be more dynamic than arrays.

Consider Adding a node to the middle of the list.



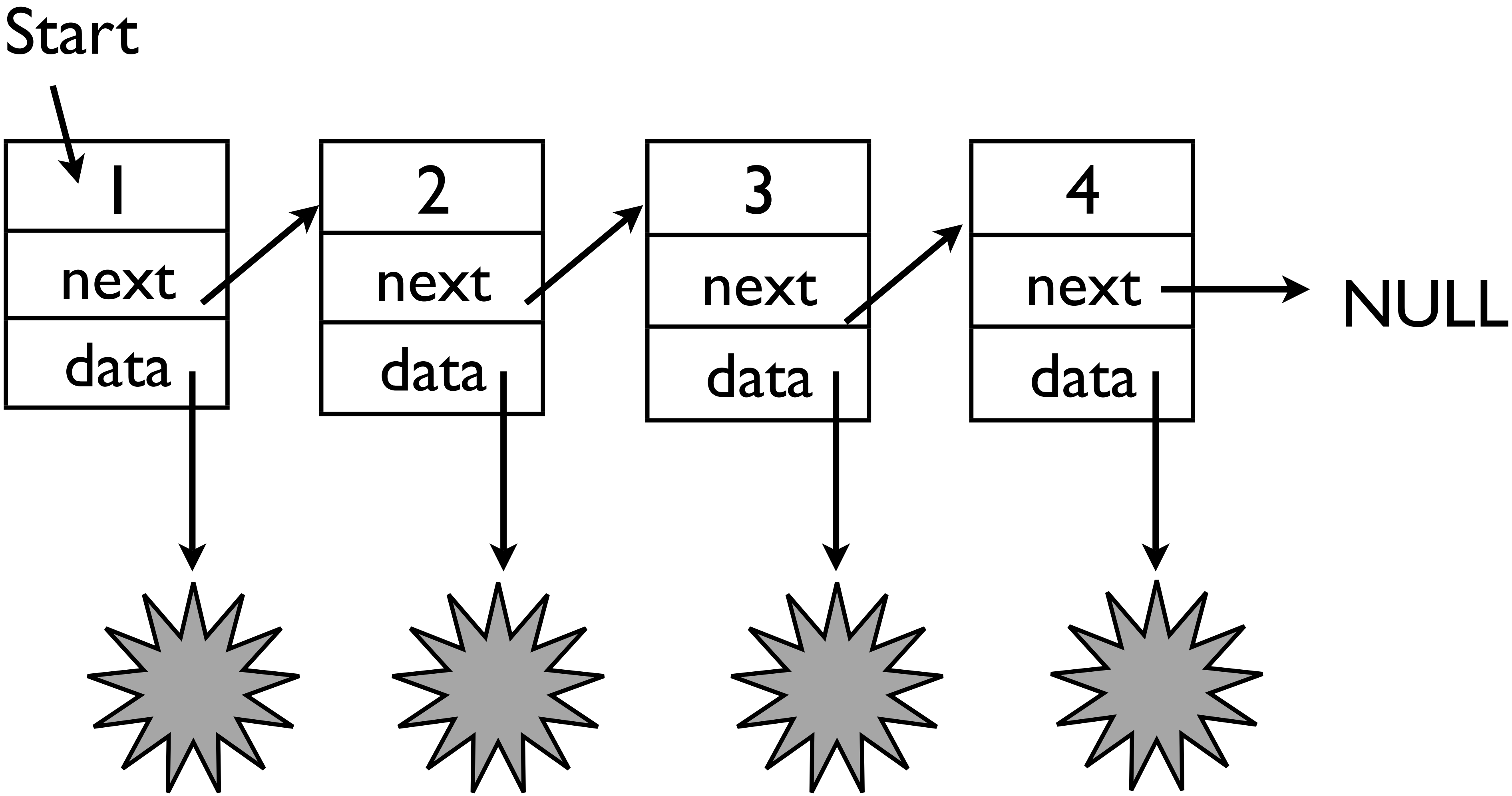
Linked List

Adding an element to an array requires moving all of the other elements to make space for the new element.



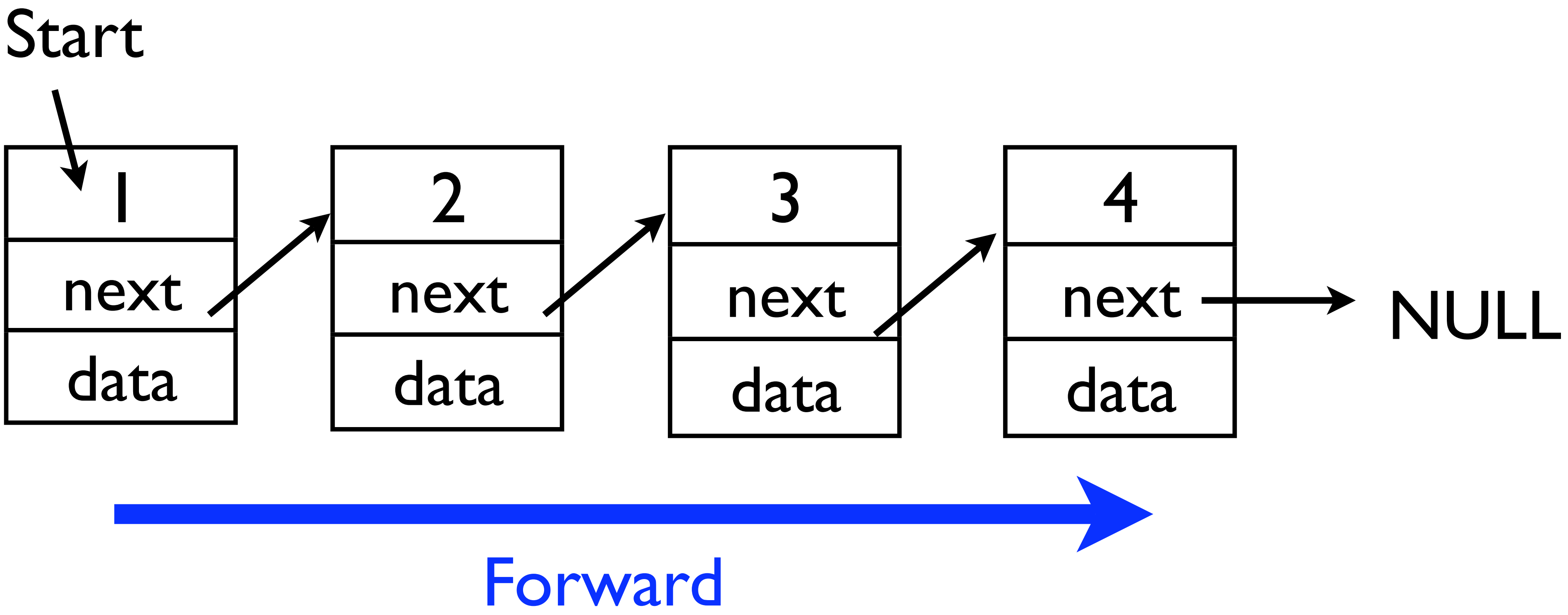
Linked List

Nodes in a Linked List can point to other dynamically allocated **data**.



Linked List

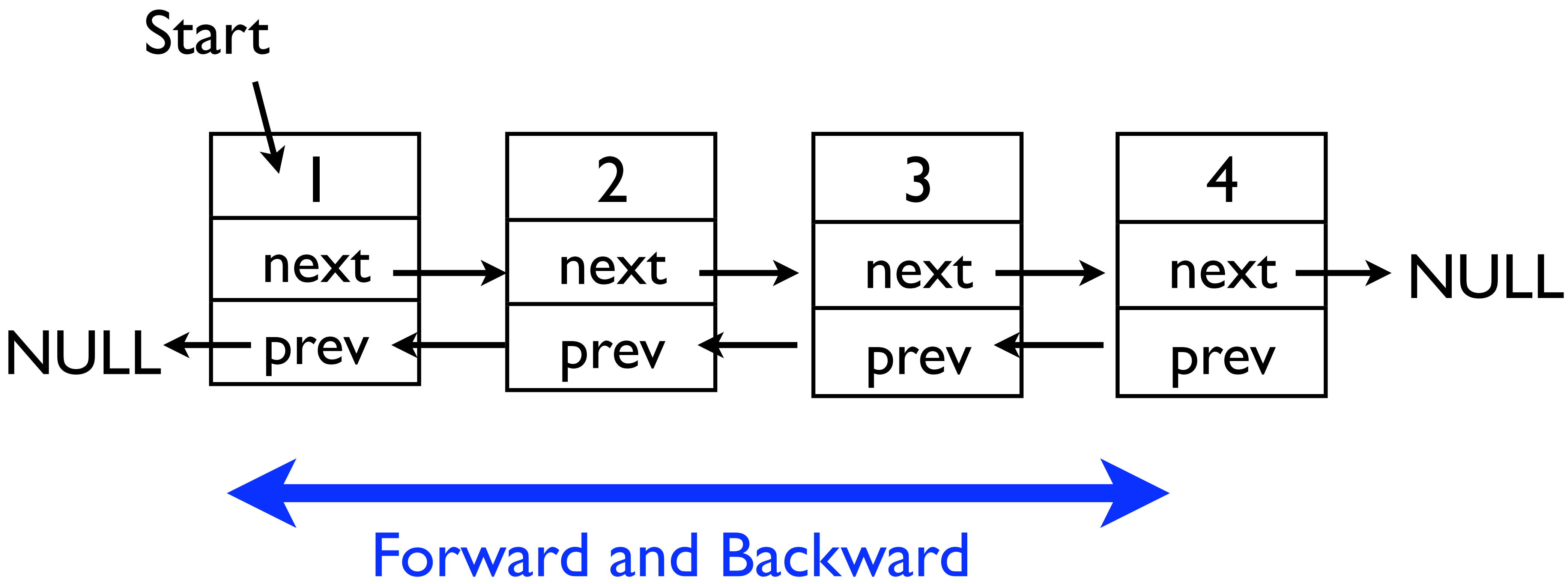
Having a single link (the node * next pointer) allows for forward traversing of the list.



How might one go backwards?

Linked List

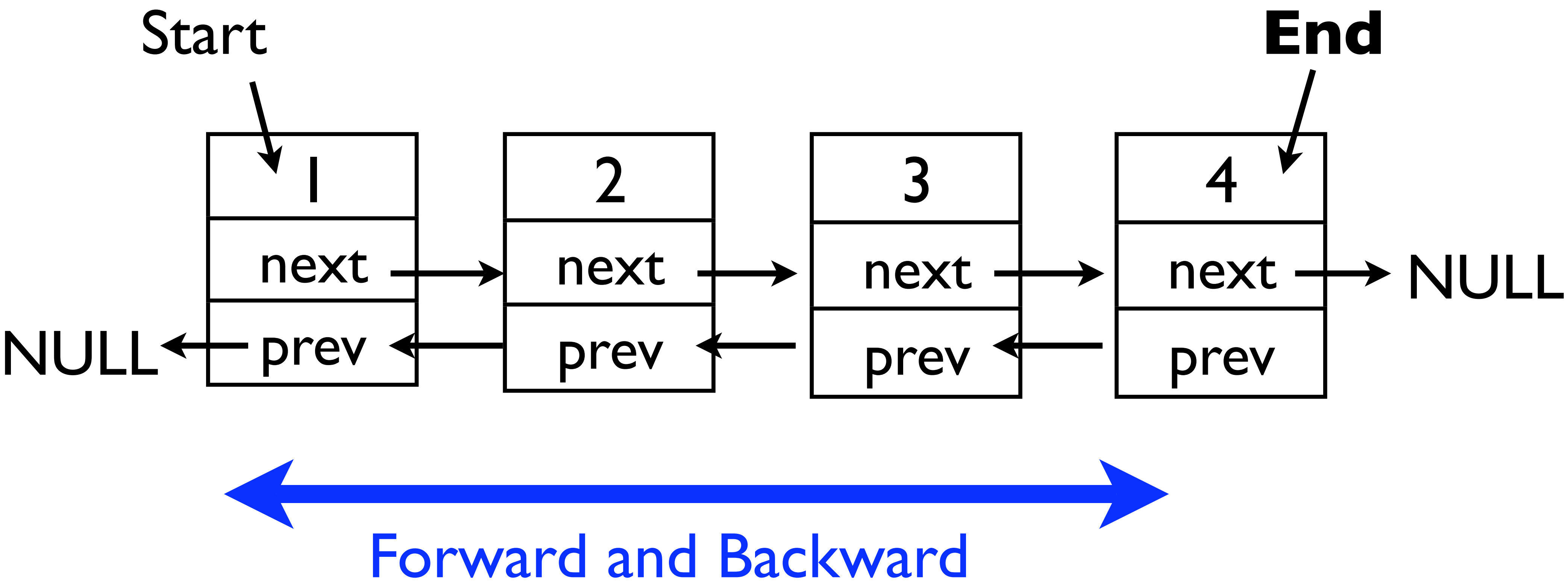
Add another link (a node *) to every node. A ***doubly-linked*** list.



Extra links allow for extra ways of ordering the list.

Linked List

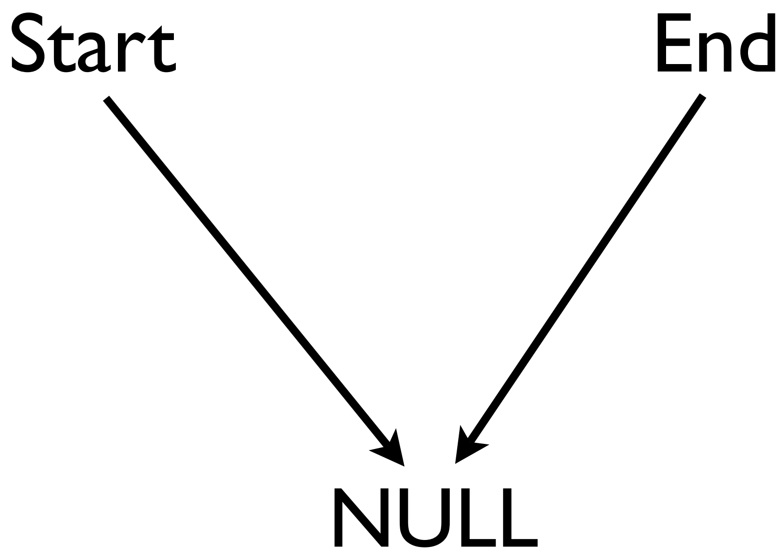
Sometimes an extra Pointer is used to keep track of the **End** of the list. (Helps in list maintenance)



Extra links allow for extra ways of ordering the list.

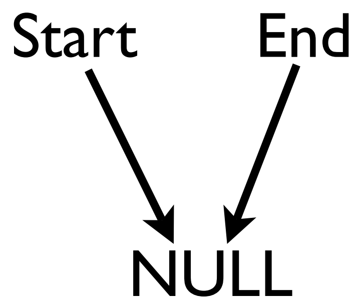
So how does one build a Linked List.

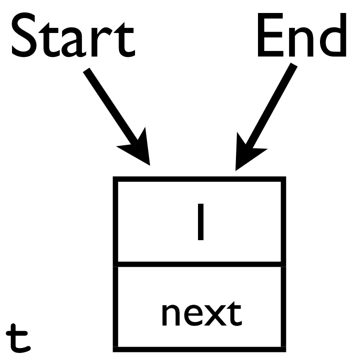
```
struct node {
  int id;
  node * next;
};

node * start = NULL;
node * end = NULL;
```

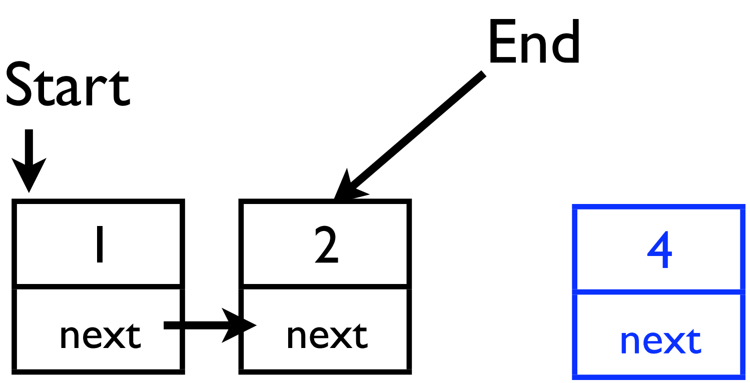


Adding a Node to the End of the List

```
// Create a New Node
node * n = new node;
n->id = id;  // some id

// 1. First, the Empty List
if((start == NULL) && (end == NULL)) {
   start = n;
   end = n;
} else { // 2. Second, already have a built list
   end->next = n; // (1)
   end = n; // (2)
   end->next = NULL; // (3)
}
```

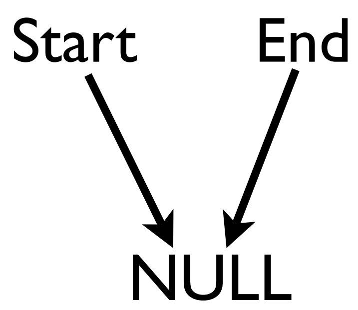


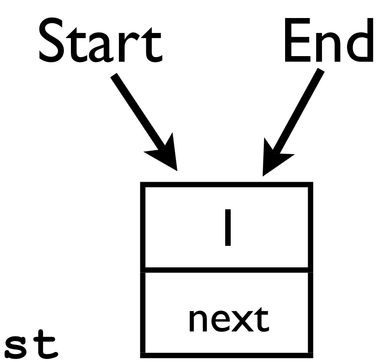


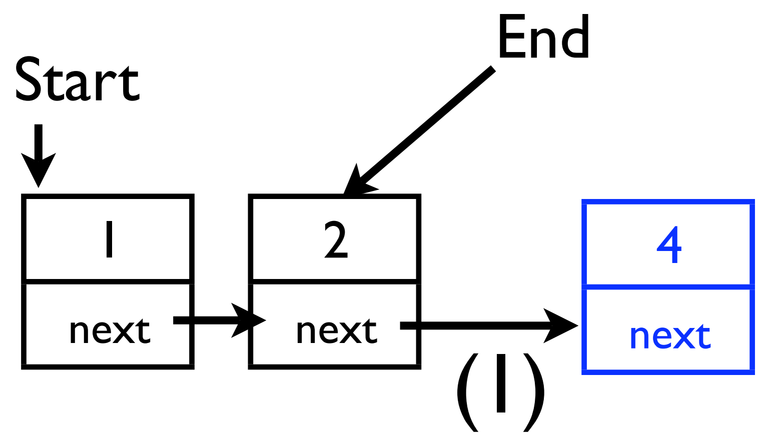


Adding a Node to the End of the List

```
// Create a New Node
node * n = new node;
n->id = id;  // some id

// 1. First, the Empty List
if((start == NULL) && (end == NULL)) {
   start = n;
   end = n;
} else { // 2. Second, already have a built list
   end->next = n; // (1)
   end = n; // (2)
   end->next = NULL; // (3)
}
```

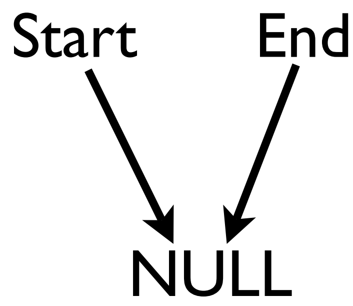




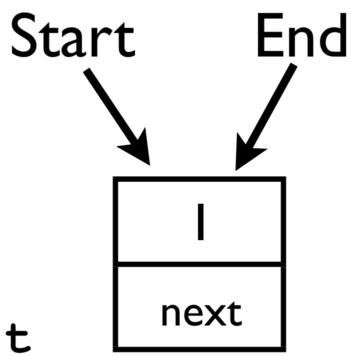


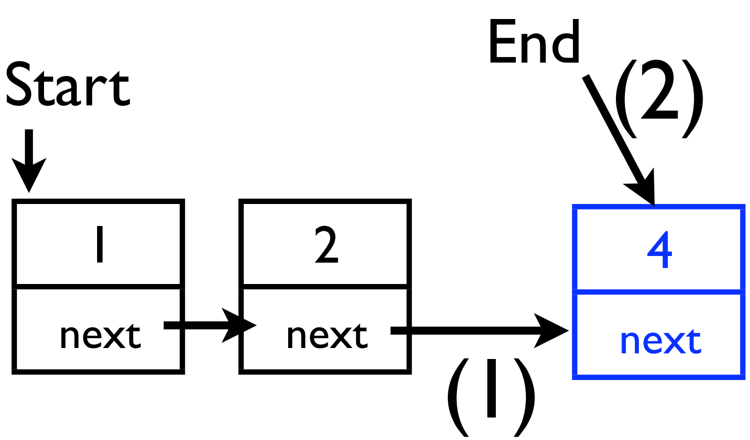
Adding a Node to the End of the List

```
// Create a New Node
node * n = new node;
n->id = id;  // some id
```



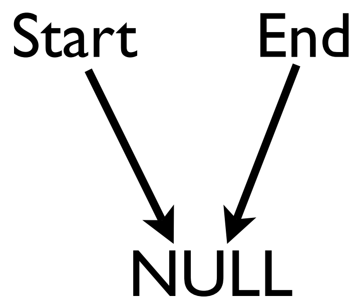

```
// 1. First, the Empty List
if((start == NULL) && (end == NULL)) {
   start = n;
   end = n;
} else { // 2. Second, already have a built list
   end->next = n; // (1)
   end = n; // (2)
   end->next = NULL; // (3)
}
```

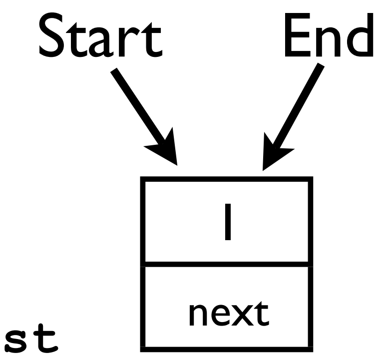


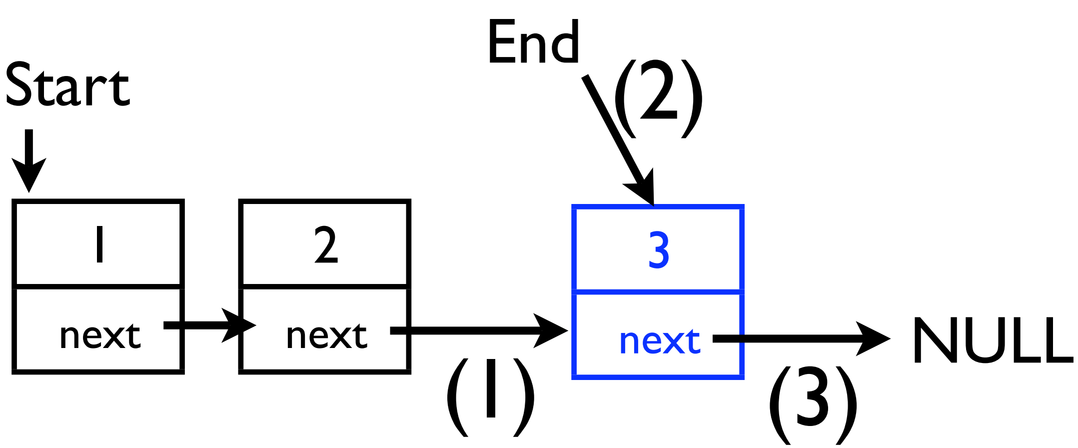


Adding a Node to the End of the List

```
// Create a New Node
node * n = new node;
n->id = id;  // some id

// 1. First, the Empty List
if((start == NULL) && (end == NULL)) {
   start = n;
   end = n;
} else { // 2. Second, already have a built list
   end->next = n; // (1)
   end = n; //(2)
   end->next = NULL; //(3)
}
```

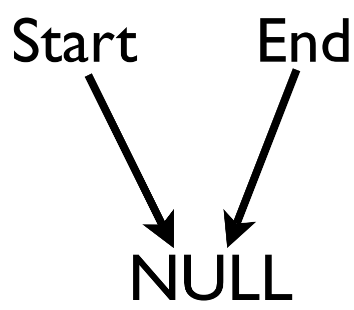


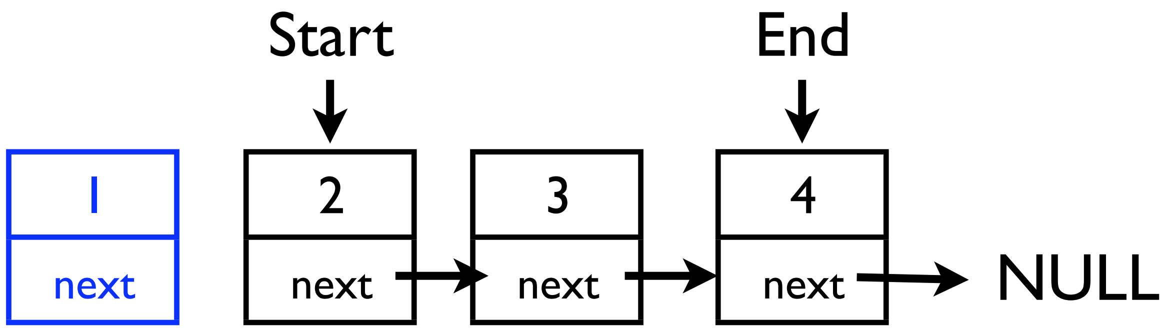




Adding a Node to the Beginning of the List

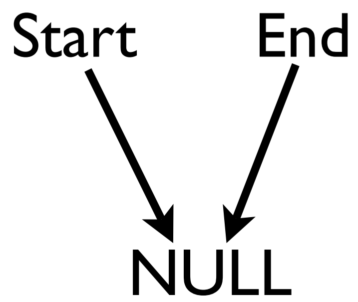
```
// Create a New Node

node * n = new node;

n->id = id;  // some id
```

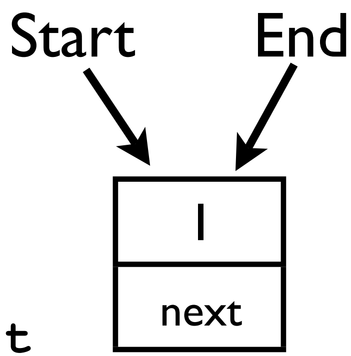


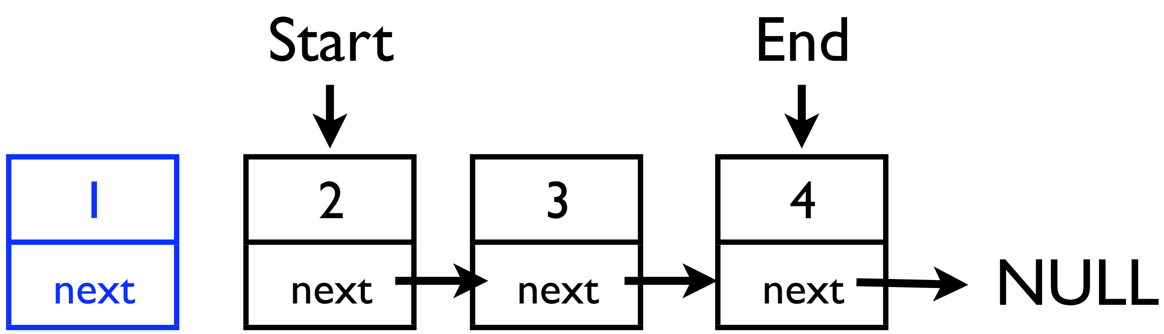


Adding a Node to the Beginning of the List

```
// Create a New Node
node * n = new node;
n->id = id;  // some id

// 1. First, the Empty List
if((start == NULL) && (end == NULL)) {
   start = n;
   end = n;
} else { // 2. Second, already have a built list
   n->next = start; // (1)
   start = n; // (2)
}
```

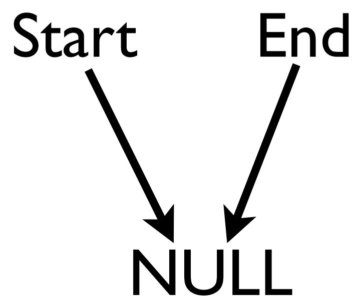


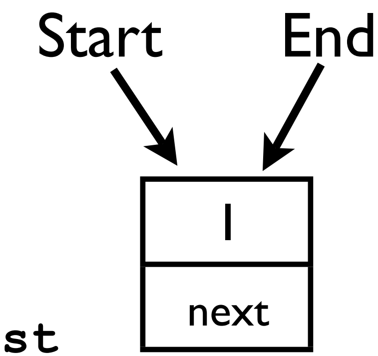


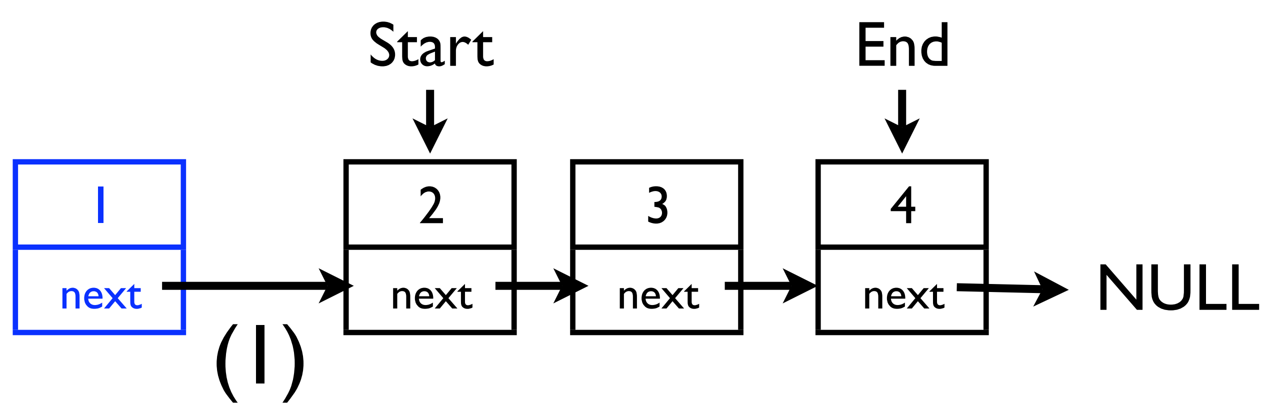


Adding a Node to the Beginning of the List

```
// Create a New Node
node * n = new node;
n->id = id;  // some id

// 1. First, the Empty List
if((start == NULL) && (end == NULL)) {
   start = n;
   end = n;
} else { // 2. Second, already have a built list
   n->next = start; // (1)
   start = n; // (2)
}
```

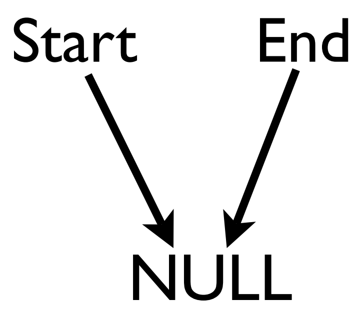




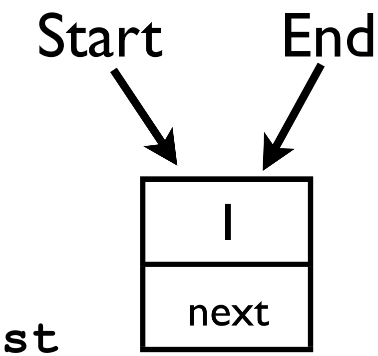


Adding a Node to the Beginning of the List

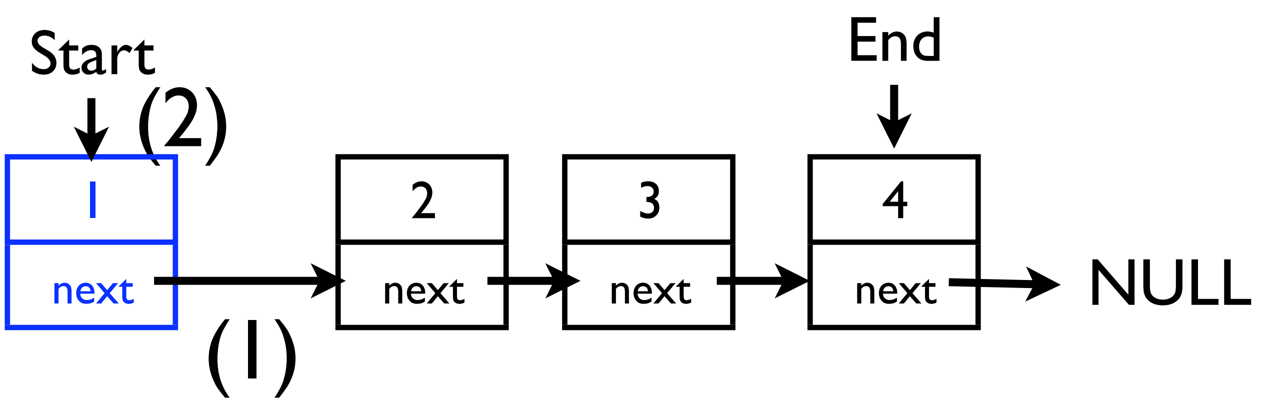
```
// Create a New Node
node * n = new node;
n->id = id;  // some id
```




```
// 1. First, the Empty List
if((start == NULL) && (end == NULL)) {
   start = n;
   end = n;
```




```
} else { // 2. Second, already have a built list
   n->next = start; // (1)
   start = n; // (2)
}
```



Adding a Node to the Middle of (an already built) List

```
// Create a New Node
node * n = new node;
n->id = id;  // some id

node * here;  // assume that we want to insert after this pointer
```