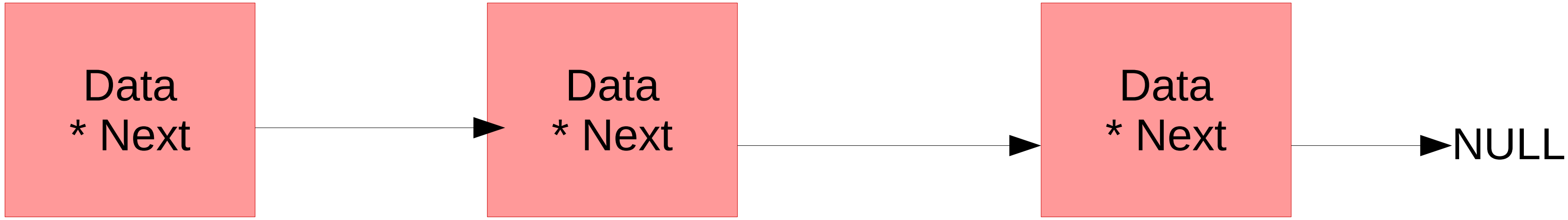
# Linked Lists



Written by Rachel J. Morris, last updated 2016-09-17

# Linked Lists

Linked Lists are a type of data structure that is implemented differently than the static and dynamic arrays that we've worked with so far.




But they still allow us to work with a *sequence* of data.
(Not all data structures keep its data sequential!)

# A Con of Dynamic Arrays...

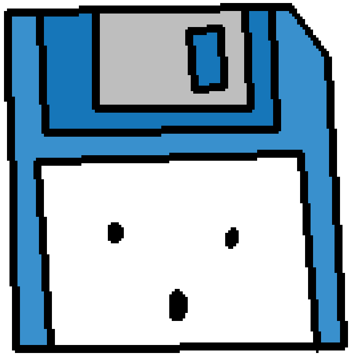
One of the problems of dynamic arrays is that, once you run out of memory, you have to spend processing time creating a new array, copying over the old array, and freeing the memory.

If your array is storing a lot of large data, or is pretty big, this could mean a noticeable performance hit.

```
string* newArray = new string[ size*2 ];

for ( int i = 0; i < size; i++ )
{
    newArray[i] = names[i];
}

delete [] names;

names = newArray;
newArray = NULL;
size *= 2;
```

# Add & remove memory *as-needed?*

Wouldn't it be nice to have a structure that adds memory as-needed, a little bit at a time, so that we don't have this bottleneck?



# The two parts of a Linked List

When programming a Linked List, there are two parts to it:

```
class LinkedList
{
    public:
    LinkedList();
    ~LinkedList();

    Node<T>* Begin();
    Node<T>* End();
    int Size();
    void Add( T data );
    void Remove( T data );

    private:
    Node<T>* head;
    Node<T>* tail;
    int m_size;
};
```

The first part is the List class itself, which contains functions to add, remove, and other useful helper-functions.

but rather than it storing an array of items internally, it only stores pointers - usually to the very first and very last thing in the list.

# The <u>two parts</u> of a Linked List

When programming a Linked List, there are two parts to it:

```
template <typename T>
class Node
{
    public:
    Node();

    T data;
    Node<T>* next;
};
```

The second part of a Linked List is the <u>data wrapper</u>; a class that wraps the data.

```
template <typename T>
class Node
{
    public:
    Node();

    T data;
    Node<T>* next;
    Node<T>* prev;
};
```

We create a Node object, which stores the **data**, but also stores a pointer to the **next item** in the list (and sometimes the previous item.)

# The two parts of a Linked List

When programming a Linked List, there are two parts to it:

```
template <typename T>
class Node
{
    public:
    Node();

    T data;
    Node<T>* next;
};
```

A Node could be a class or a struct, and can contain more information than this

but this is a pretty common node and similar to what you would see in many implementations.

# Pros and Cons of a Linked List

## Pros

- Avoid the stop-and-resize processing time like with a dynamic array
- Insertion and Deletion is cheap and easy - just moving pointers around!
- Stacks and Queues can be implemented easily with a Linked List

## Cons

- No random-access like with arrays: Have to traverse the list to get a specific element (Sequential access)
- More memory is needed due to next/prev pointers (vs. no pointers in an array)

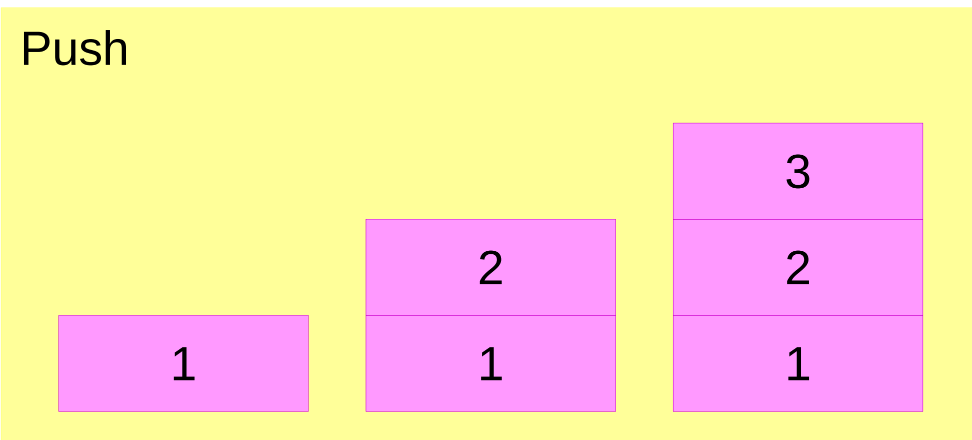
## Other

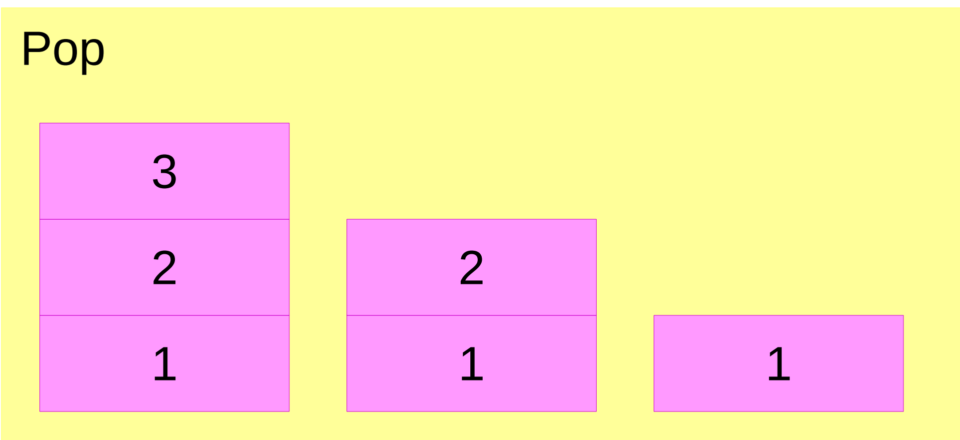
- Elements of the list don't have to be in contiguous memory spaces like with an array.

# Why else is a Linked List useful?

We can use a Linked List class to implement a **Stack** and **Queue** data structure...

**Stack**



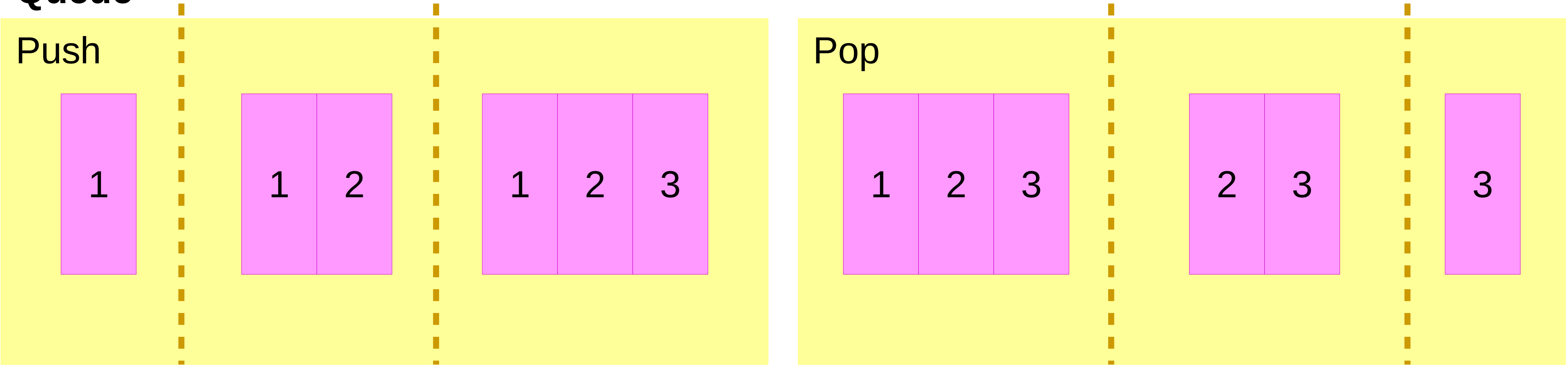

A **Stack** is a first-in, last-out structure.

You're only able to access the top-most item of the Stack.

# Why else is a Linked List useful?

We can use a Linked List class to implement a **Stack** and **Queue** data structure...

**Queue**




A **Queue** is a first-in, first-out structure.

You’re only able to access the front-most item of the Queue.

# Implementation

# Allocating memory as-needed

Let's assume we have the following List class written:

```
class List
{
    public:
    List();
    ~List();

    void AddItem( const string& newItem );
    string GetItem( int index );

    private:
    Node* m_ptrFirst;
    int m_nodeCount;
};
```

# Allocating memory as-needed

## Adding the **First Element:**

```
class List
{
    public:
    List();
    ~List();

    void AddItem( const string& newItem );
    string GetItem( int index );

    private:
    Node* m_ptrFirst;
    int m_nodeCount;
};
```

```
m_ptrFirst = new Node;
m_ptrFirst->data = newItem;
m_ptrFirst->ptrNext = nullptr;
```

When we go to add our first element of the list:

1. initialize **m_ptrFirst** as a new Node instance

2. Set **m_ptrFirst**’s data value to the new item being passed in.

3. If it isn’t **nullptr** already, set its **ptrNext** to nullptr.

(And add 1 to **m_nodeCount**)

# Allocating memory as-needed

## Adding the **Second Element:**

```
class List
{
    public:
    List();
    ~List();

    void AddItem( const string& newItem );
    string GetItem( int index );

    private:
    Node* m_ptrFirst;
    int m_nodeCount;
};
```

Then, the next time we go to add something, it will now be the second element in the list.

We can't just overwrite **m_ptrFirst,** but we can create a new node - which is the one it is pointing to with **ptrNext**:

```
Node* newNode = new Node;
newNode->data = newItem;
newNode->ptrNext = nullptr;
m_ptrFirst->ptrNext = newNode;
```

Create a new node

Set its data and next ptr

Point the **first** node's **next** ptr to this new node.

But we can't hard-code this logic; we have to generalize.

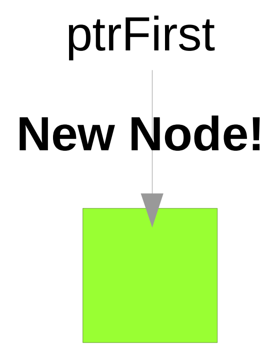
# Pseudocode: Adding an item

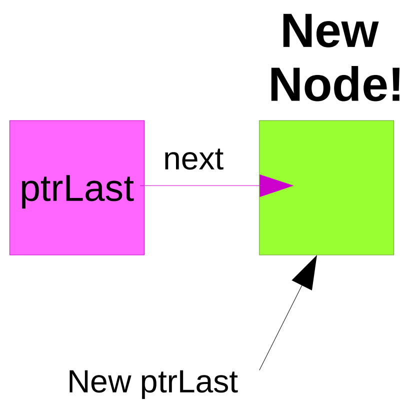
Logic behind adding a new item:

1)Add new data "d" to the list. (Function begin)

2)Is this the first item to be added to the list?
   i) YES:
      a)Instantiate our first-element Node pointer as a new item.
      b)Set its data to "d"
      c)Add 1 to the Node Count

   ii) NO:
      a)Get the end of the list (ptrTail*?)
      b)Create and instantiate a new Node* item.
      c)Set the new item's data to "d".
      d)For the last element of the list, set its new "ptrNext" pointer to the new Node that we made.
      e)Add 1 to the Node Count




(This assumes that Node's constructor sets ptrNext to nullptr!)

# Removing an item

When removing an item from a Linked List,

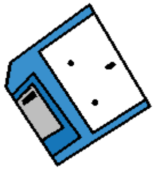
you could have a version that passes in an **index**,

or a version that passes in the **data** that is stored.

```
template <typename T>
class Node
{
    public:
    Node();

    T data;
    Node<T>* next;
};
```

If an index is passed in, you'll have to **traverse** the list while counting what position you're at.

If a value is passed in, you'll still have to **traverse** the list, but don't have to count the position.

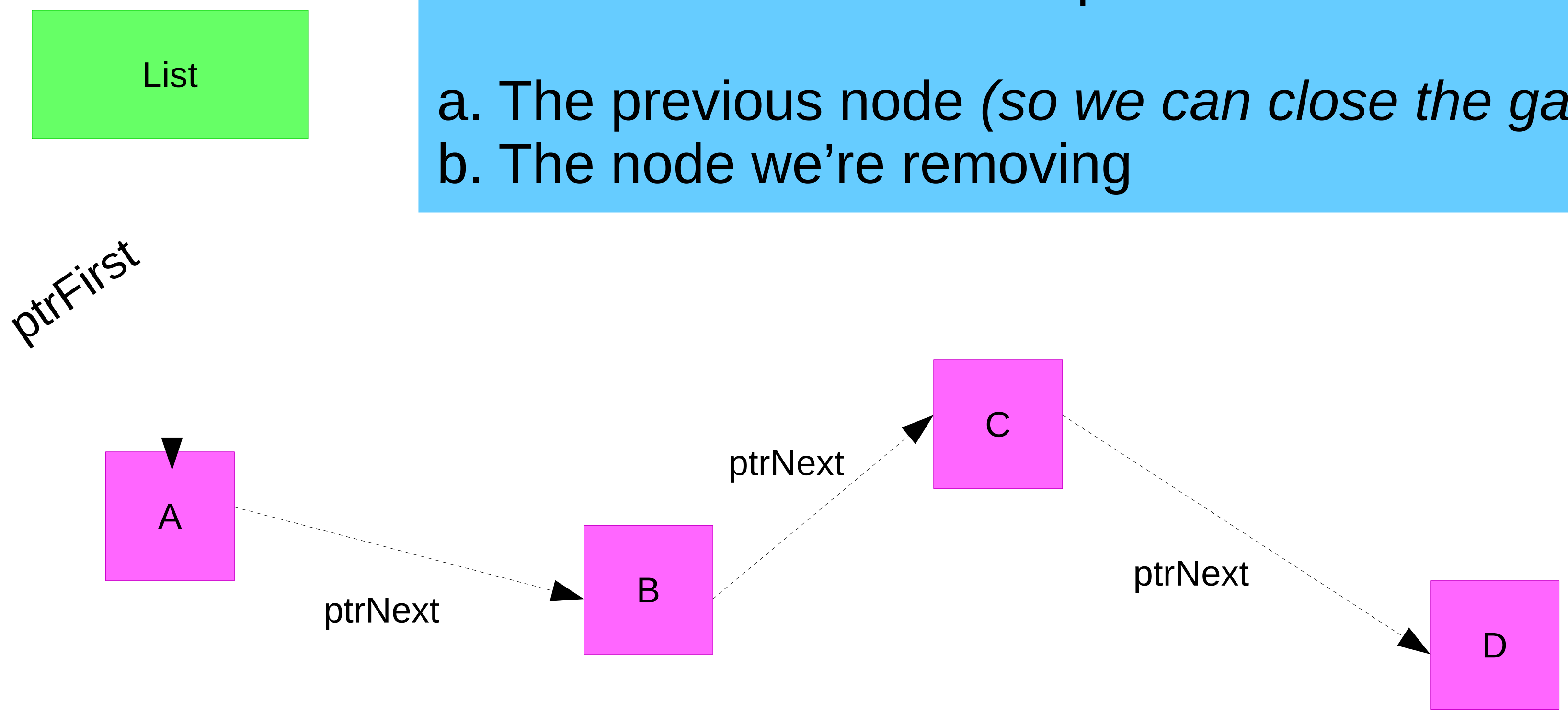


Depending on if your Node stores only the **next** ptr or both **next and prev** ptrs, the implementation might change...

# Removing an item

When we're removing an item from the list, we need to keep track of:

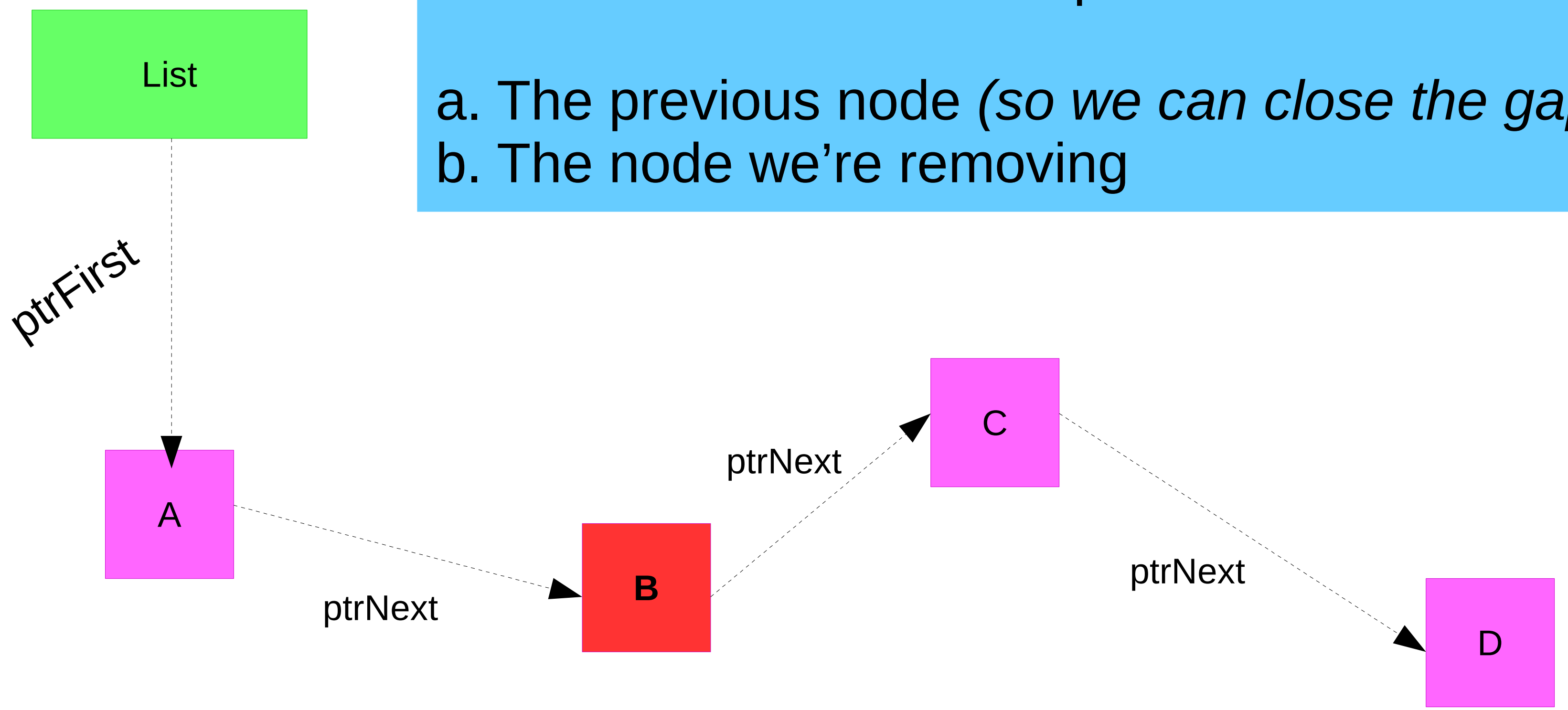
a. The previous node *(so we can close the gap)*
b. The node we're removing



# Removing an item

When we're removing an item from the list, we need to keep track of:

a. The previous node *(so we can close the gap)*
b. The node we're removing

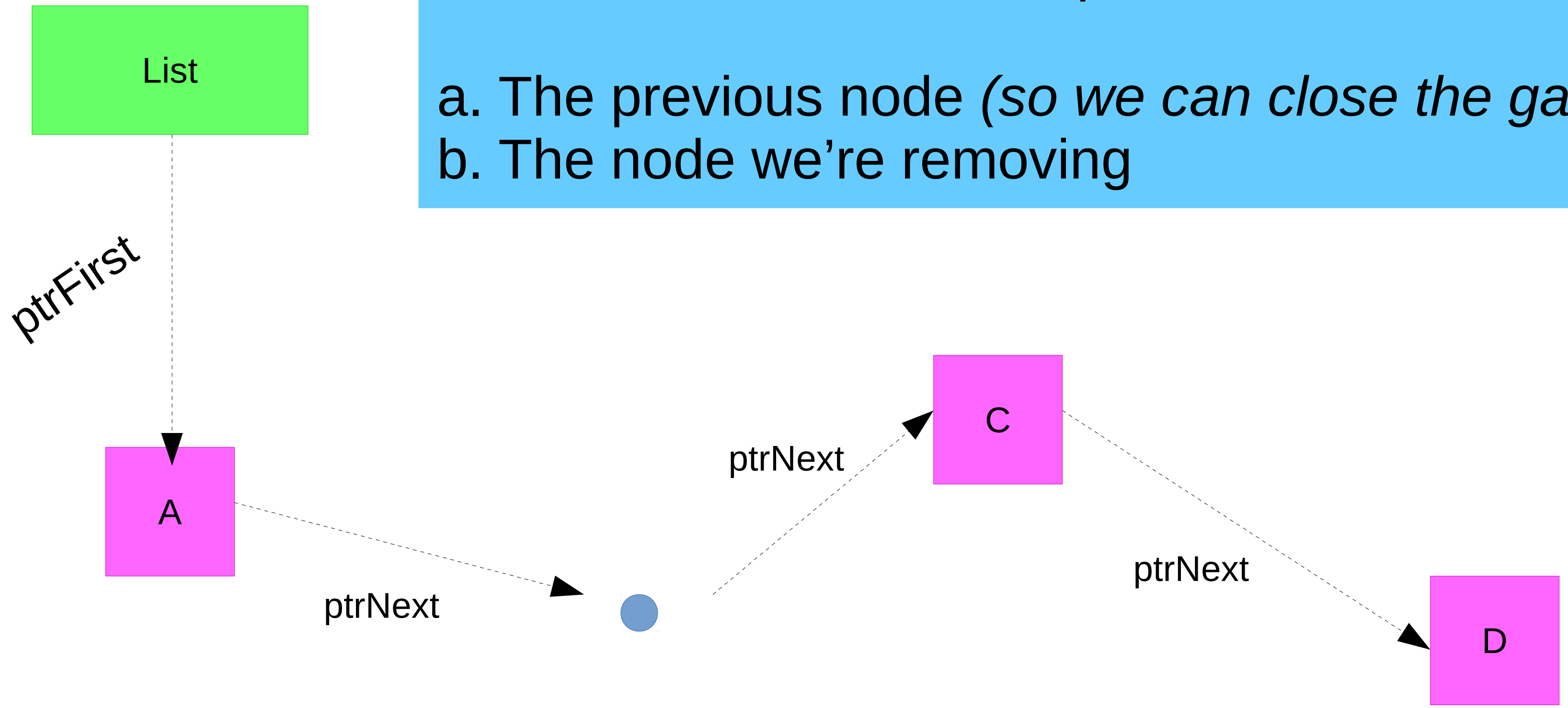



**Save what B's ptrNext is pointing at**
**&**
**Remove B**

# Removing an item

When we're removing an item from the list, we need to keep track of:

a. The previous node *(so we can close the gap)*
b. The node we're removing

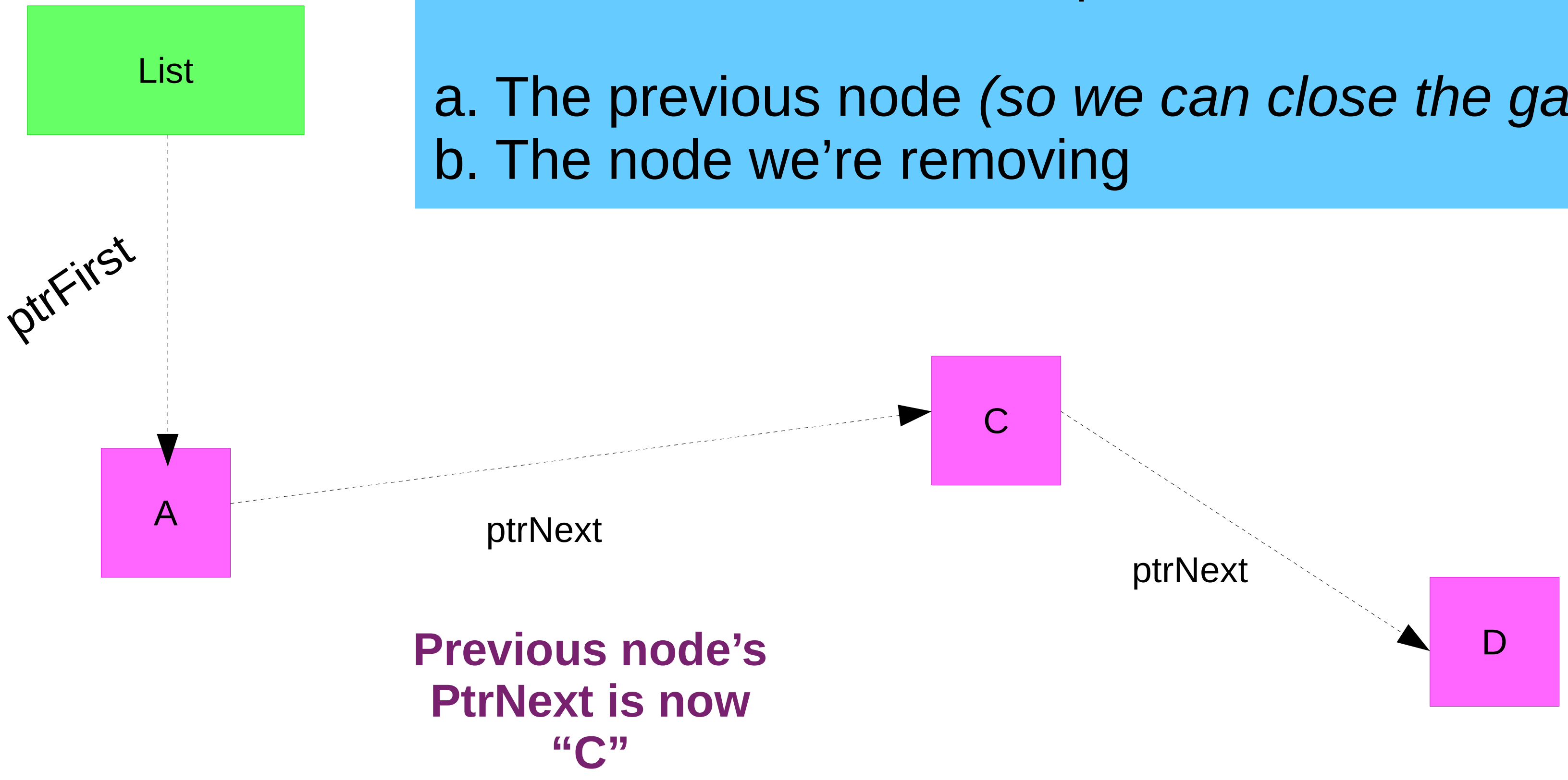



**Need to update the pointers!**

# Removing an item

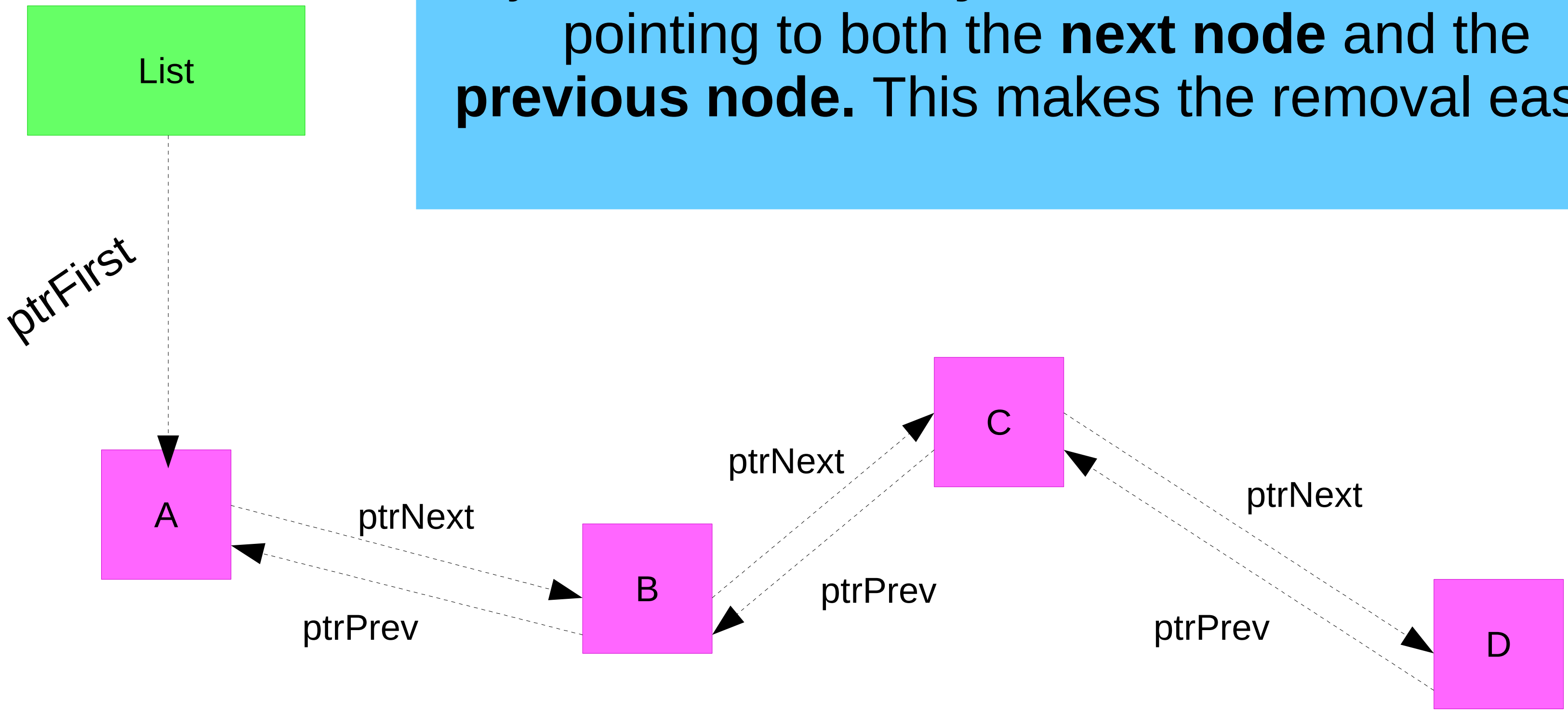
When we're removing an item from the list, we need to keep track of:

a. The previous node *(so we can close the gap)*
b. The node we're removing



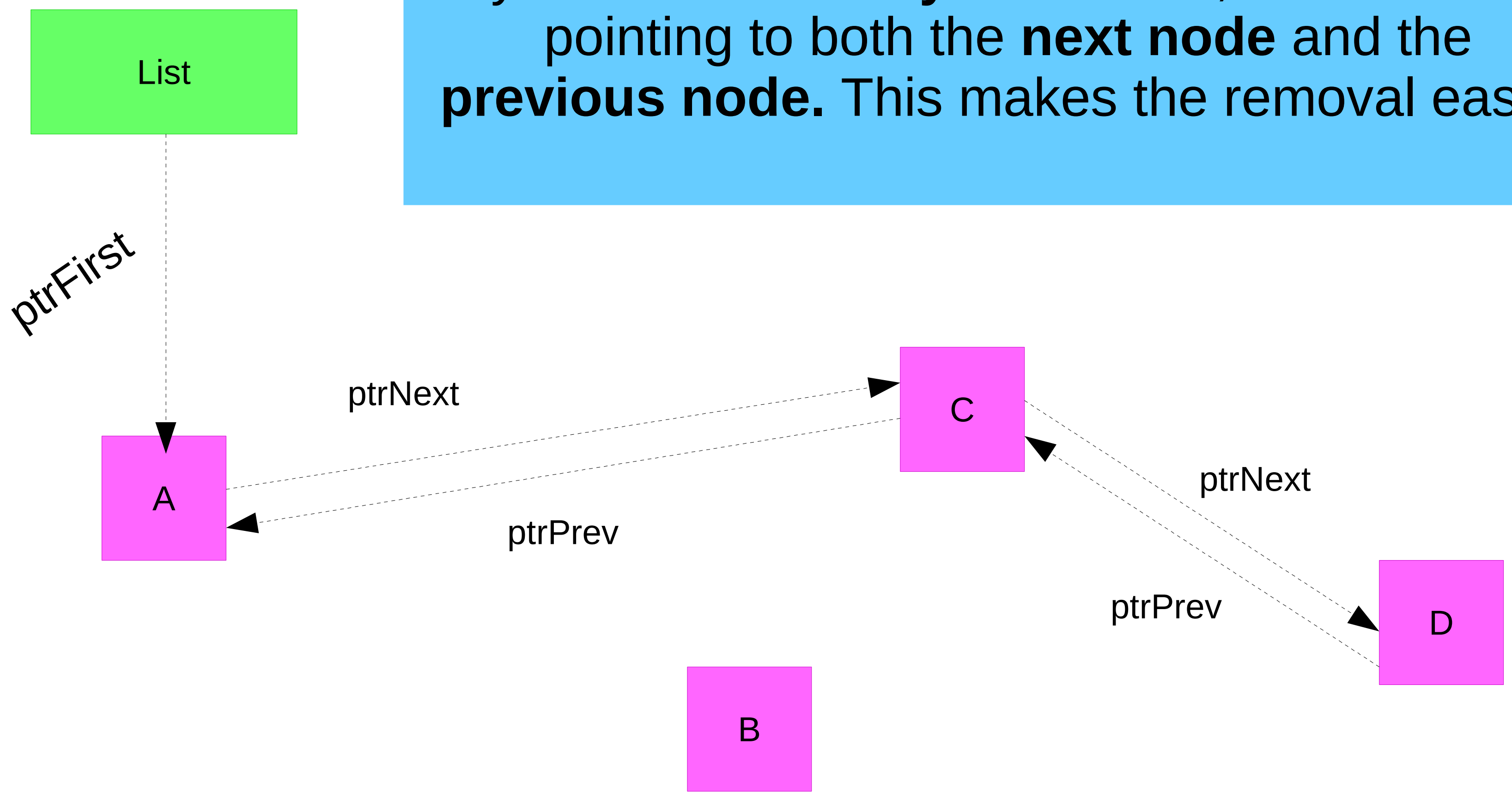
# Removing an item

If you have a **doubly linked list**, the Nodes are pointing to both the **next node** and the **previous node.** This makes the removal easier.



# Removing an item

If you have a **doubly linked list**, the Nodes are pointing to both the **next node** and the **previous node.** This makes the removal easier.



Implement in code...