Linked Lists

Written by Rachel Singh, last updated Sep 19, 2018

About

At this point, we've worked with arrays to store a lot of the same type of data. There are other types of structures we can use to store data (thus, *data structures*), so let's get into our first non-array one: Linked Lists, which use dynamic variables and a chaining structure to get around some of the shortfalls of arrays.

Topics

1. Review: Pointers

2. Dynamic Array vs. Linked List

3. How a Linked List works

4. Stepping through Linked List functionality
   - `PushBack`
   - `PushFront`
   - `Insert`
   - `PopBack`
   - `PopFront`
   - `Remove`
   - `GetFront`
   - `GetBack`
   - `Get or []`

Review: Pointers

1. Review: Pointers

Before we jump straight into building Linked Lists, let's review how we can use pointers.

Our Linked Lists will use both **dynamic variables and arrays** (dynamically allocating memory for a *single variable or an array of variables*, as-needed),

as well as pointing to addresses of variables that already exist, with the address-of operator &.

Notes

Address-of operator: &

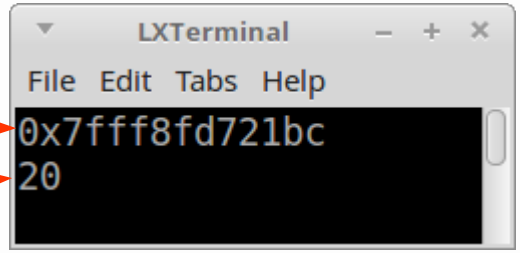
1. Review: Pointers

Pointers are just a type of variable that stores **addresses**. No matter how we're using a pointer, that's what it all boils down to.

```
int main()
{
    int someNumber = 20;

    int* ptrNum = &someNumber;

    Display address ptrNum is pointing to.
    cout << ptrNum << endl;
    cout << *ptrNum << endl;
    De-reference the pointer to get the value
    of the item it is pointing to.
```



Notes

Address-of operator: &

1. Review: Pointers

Every variable has an address, and we can look at what that address is with the **address-of operator, &**.

Even pointers have addresses – they are, after all, variables that store some data at some location in memory!

```
char var;
char * ptr;

// address of var
cout << &var;

// address of ptr
cout << &ptr;
```

Notes

Address-of operator: &

1. Review: Pointers

Notice the different ways we can use pointers...

```
// pointing to existing variable
int a;
int ptr = &a;
```

```
// dynamic variable
int * var = new int;
delete var;
```

```
// dynamic array
int * arr = new int[100];
delete [] arr;
```

Notes

Address-of operator: &

Ways to use ptrs:

Point to existing var
`ptr = &b;`

Dynamic variable
`TYPE* ptr = new TYPE;`

Dynamic array
`TYPE* ptr = new TYPE[size];`

1. Review: Pointers

If you need to further review pointers, I have three lecture videos from a previous course:

1. Pointers
https://www.youtube.com/watch?v=Jlr6nSzFdGo

2. Memory Management
https://www.youtube.com/watch?v=GxDETB16Clk

3. Dynamic Variables & Arrays
https://www.youtube.com/watch?v=oqnFZ9TfeDo

Notes

Address-of operator: &

Ways to use ptrs:

Point to existing var
```
ptr = &b;
```

Dynamic variable
```
TYPE* ptr = new TYPE;
```

Dynamic array
```
TYPE* ptr = new TYPE[size];
```

Dynamic Array vs. Linked List

2. Dynamic Array vs. Linked List

Previously, we implemented a class that used a dynamic array to store its data.

When the Push or Insert function was called, it would check to see if the array was full, and if it was, it would **Resize** the array.

Notes

2. Dynamic Array vs. Linked List

The resizing process includes the steps:

1) Create a new dynamic array of a larger size

2) Copy all the contents from the old (small) array to the new (large) array

3) Free up the memory from the old array

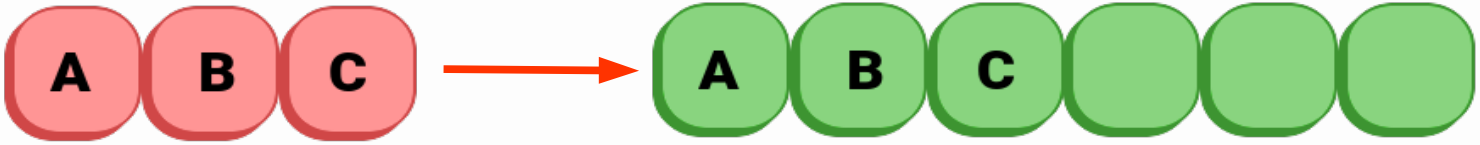
4) Update the pointer to point to the new (large) array.

Notes

2. Dynamic Array vs. Linked List

Every time the array had to be resized, the program would have to stop, allocate more memory, then *copy over* all the data.

If our structure were storing large objects (such as classes with lots of variables or arrays within them), this could be costly.



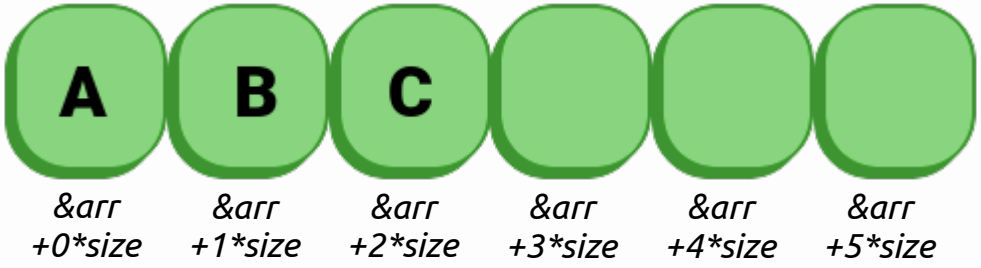
Notes

Dynamic Array
- Resize is costly

2. Dynamic Array vs. Linked List

On the other hand, *accessing* specific elements of the dynamic array at some given index was not expensive at all.

Since arrays are contiguous in memory, accessing an item at some index is almost instantaneous.



Notes

Dynamic Array
- Resize is costly
- Access is cheap

2. Dynamic Array vs. Linked List

So does the instantaneous **access time** make up for the costly **insert/resize time?**

It depends on what you're implementing!

If your program accesses data a lot more than it inserts new data, then this could be a good trade-off.

If your program inserts data more often than it accesses it, then this would be inefficient.

Notes

Dynamic Array

- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

2. Dynamic Array vs. Linked List

Dynamic Arrays

- Allocate a big chunk of memory ahead of time.
- Can run out of space, then need to resize.

Linked Lists

- Allocate one item at a time.
- Never “runs out of space”; creates more space each time a new item is created.

Notes

Dynamic Array

- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

How a Linked List works

3. How a Linked List works

With a Linked List, we start with a List object and no memory allocated for any elements.

LinkedList

```
Push( … )
Pop()
Get()
Clear()
IsEmpty()
Size()

Node* ptrFirst
Node* ptrLast
int itemCount
```

We have a second class – a **Node** – defined, which will actually store the data.

Node

```
Node* ptrNext

Node* ptrPrev

TYPE data
```

Notes

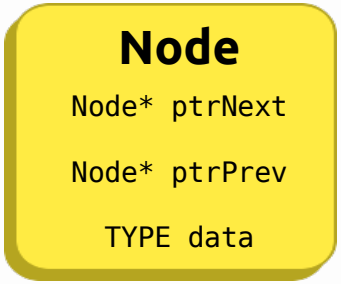
Dynamic Array

- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

Linked List

- Wraps element data in “Nodes”

3. How a Linked List works



The Node object wraps the data being stored in our linked list, as well as a pointer to the next item and maybe the previous item in the list.

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

    Node<T>* m_ptrNext;
    Node<T>* m_ptrPrev;

    T m_data;
};
```

For a **Singly-Linked List**, the Nodes only point to the next item.

For a **Doubly-Linked List**, the Nodes point to the next and previous items.

Notes

Dynamic Array
- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

Linked List
- Wraps element data in “Nodes”

3. How a Linked List works

LinkedList

Push(…)
Pop()
Get()
Clear()
IsEmpty()
Size()

Node* ptrFirst
Node* ptrLast
int itemCount

Then, the external structure is the LinkedList, which contains the standard functionality to **add items**, **remove items**, and **access items**.

```
template <typename T>
class LinkedList
{
    private:
    /* Member Variables */
    Node<T>* m_ptrFirst;
    Node<T>* m_ptrLast;
    int m_itemCount;

    public:
    /* Member Functions */
    LinkedList();
    ~LinkedList();

    void PushFront( T newData );
    void PushBack( T newData );
    void PopFront() noexcept;
    void PopBack() noexcept;
    T& GetFront();
    T& GetBack();
    T& operator[]( const int index );

    void Clear();
    bool IsEmpty();
    int Size();
    bool IsInvalidIndex( int index ) const;
};
```

Notes

Dynamic Array

- Resize is costly
- Access is cheap
- Whether to use is a *design decision*.

Linked List

- Wraps element data in “Nodes”

3. How a Linked List works

All the data structures we create will at least have:

- Add functions
- Remove functions
- Access functions

But different structures implement these in different ways, or restrict access to certain parts of the data.

```
template <typename T>
class LinkedList
{
    private:
    /* Member Variables */
    Node<T>* m_ptrFirst;
    Node<T>* m_ptrLast;
    int m_itemCount;

    public:
    /* Member Functions */
    LinkedList();
    ~LinkedList();

    void PushFront( T newData );
    void PushBack( T newData );
    void PopFront() noexcept;
    void PopBack() noexcept;
    T& GetFront();
    T& GetBack();
    T& operator[]( const int index );

    void Clear();
    bool IsEmpty();
    int Size();
    bool IsInvalidIndex( int index ) const;
};
```

Add, remove, and access

Notes

Dynamic Array

- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

Linked List

- Wraps element data in “Nodes”

3. How a Linked List works

When we start with a empty LinkedList, no Nodes exist, and the pointers to the first and last items are both pointing to **nullptr**.

This is an empty LinkedList.

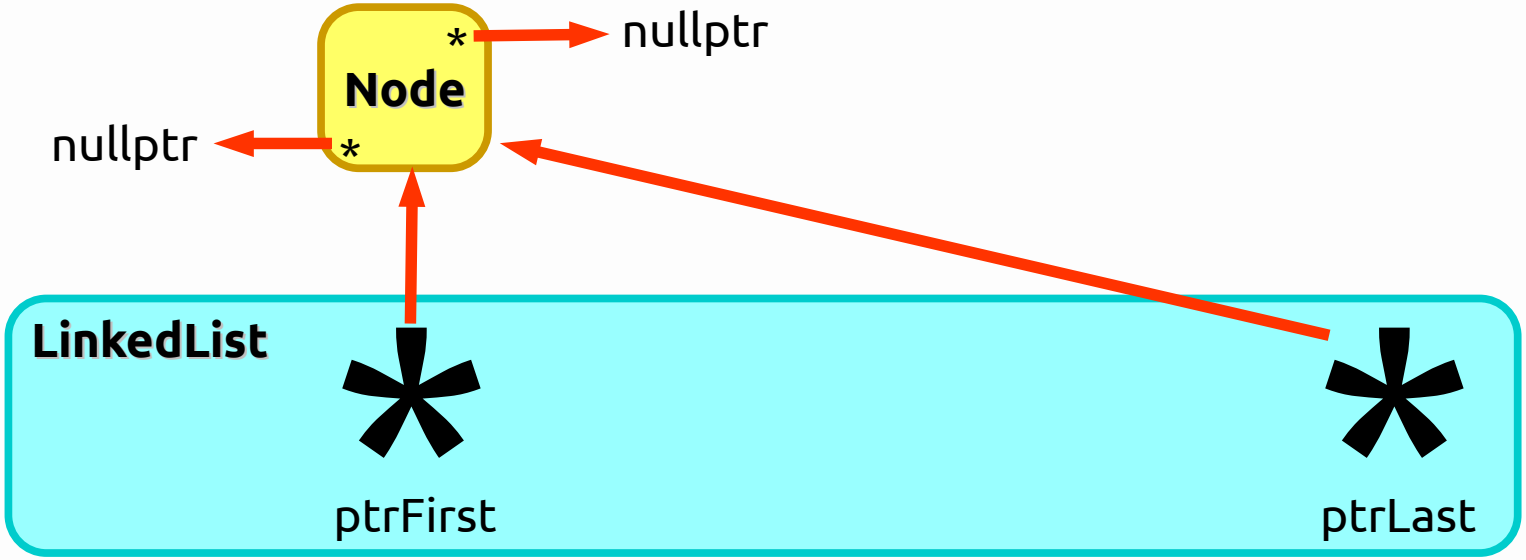
Notes

Dynamic Array
- Resize is costly
- Access is cheap
- Whether to use is a *design decision*.

Linked List
- Wraps element data in “Nodes”

3. How a Linked List works

Any time we add a new item to the Linked List, we allocate memory for that new **Node** at that time, then have our List point to it.



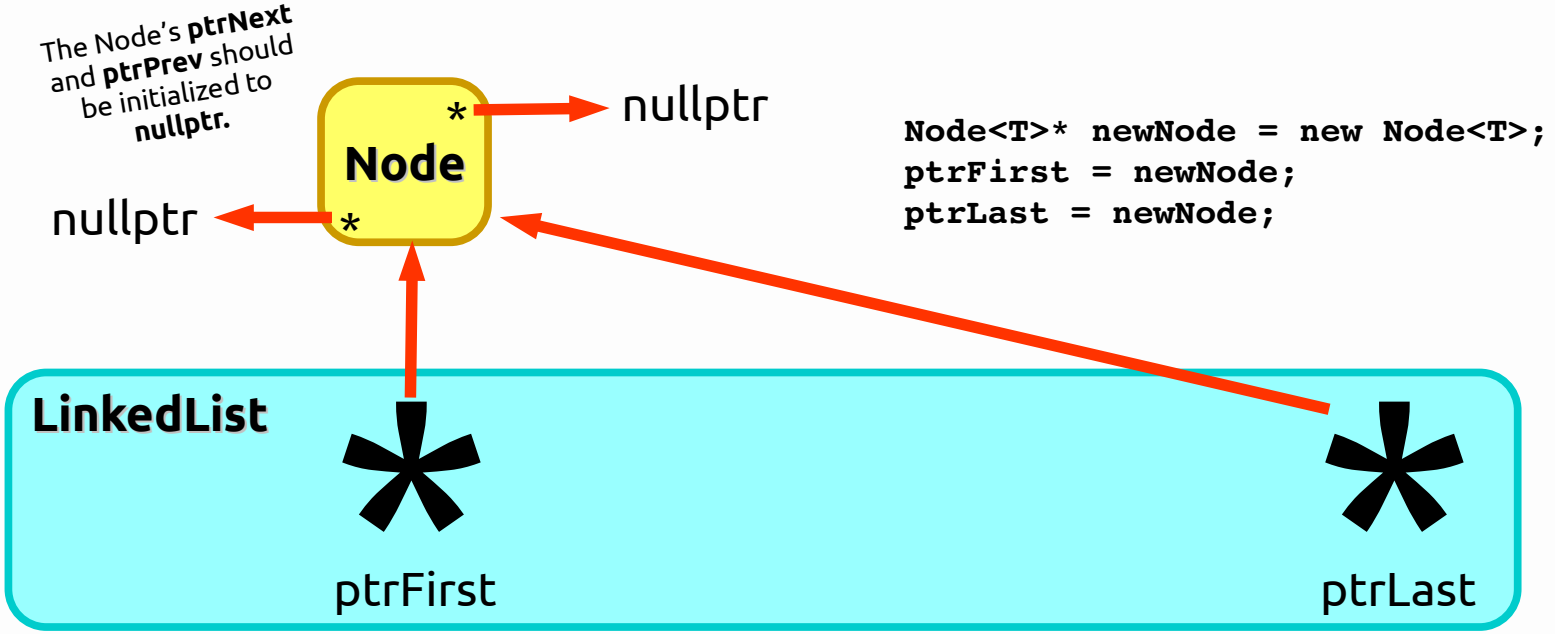
Notes

Dynamic Array
- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

Linked List
- Wraps element data in “Nodes”

3. How a Linked List works

Any time we add a new item to the Linked List, we allocate memory for that new **Node** at that time, then have our List point to it.



Notes

Dynamic Array

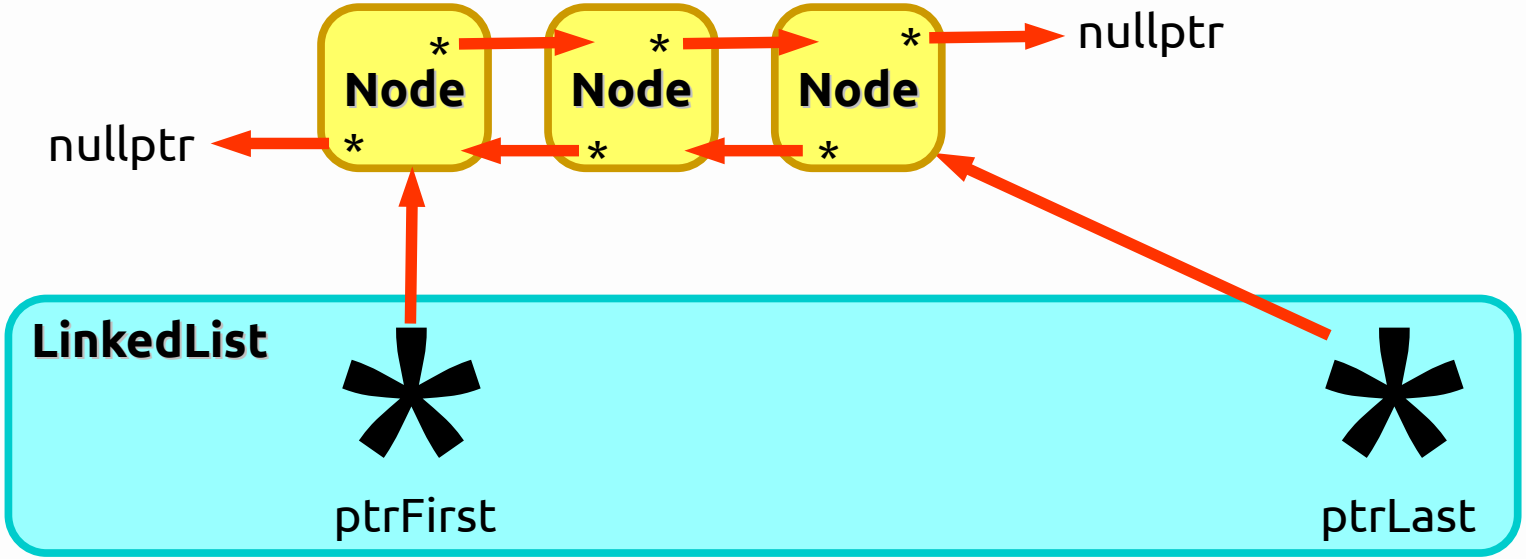
- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

Linked List

- Wraps element data in "Nodes"

3. How a Linked List works

As we keep pushing new data to the LinkedList, new **Nodes** are created each time.



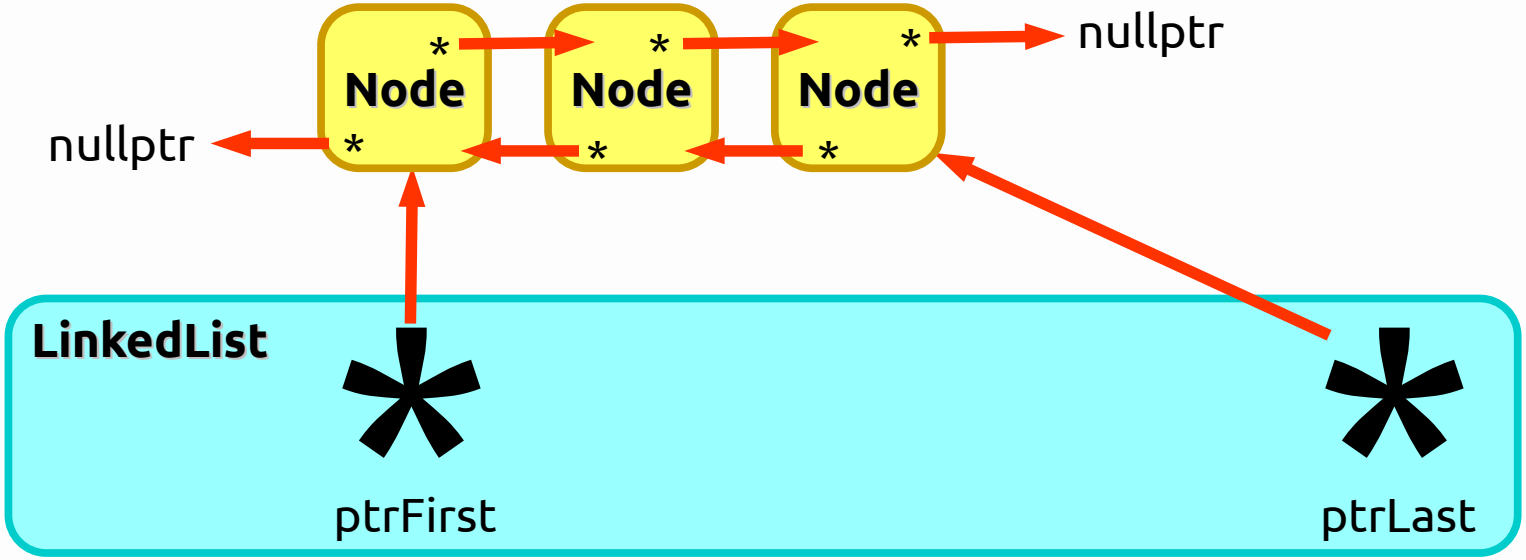
Notes

Dynamic Array
- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

Linked List
- Wraps element data in “Nodes”

3. How a Linked List works

A Doubly- Linked List keeps track of the **first** and **last** Nodes, and then each Node keeps track of its **next** and **previous** Nodes. This is all done with pointers.



Notes

Dynamic Array

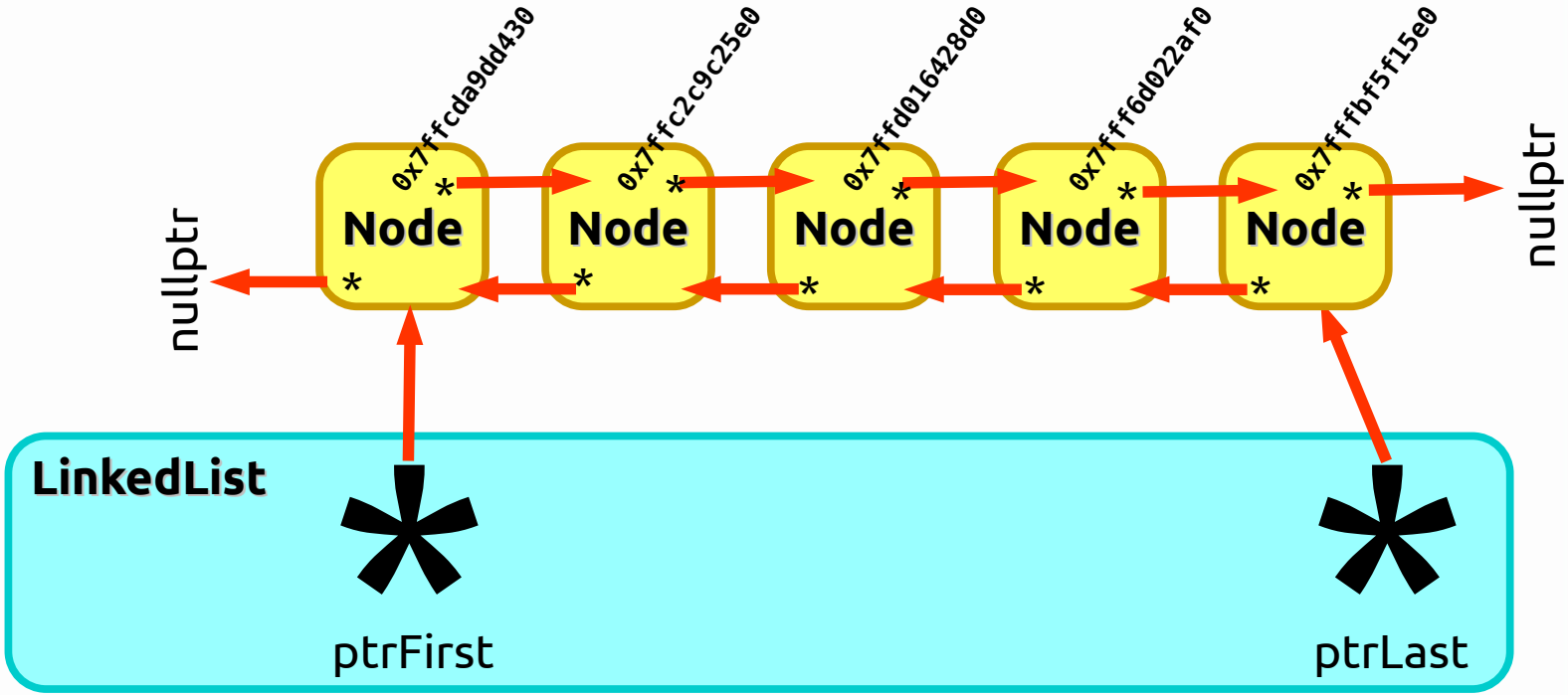
- Resize is costly
- Access is cheap
- Whether to use is a *design decision*.

Linked List

- Wraps element data in “Nodes”
- Pointers are used to build a “chain”

3. How a Linked List works

Because we only allocate memory for the list as-needed, then the **elements** of the list can be (and are!) **non-contiguous in memory**.



Notes

Dynamic Array

- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

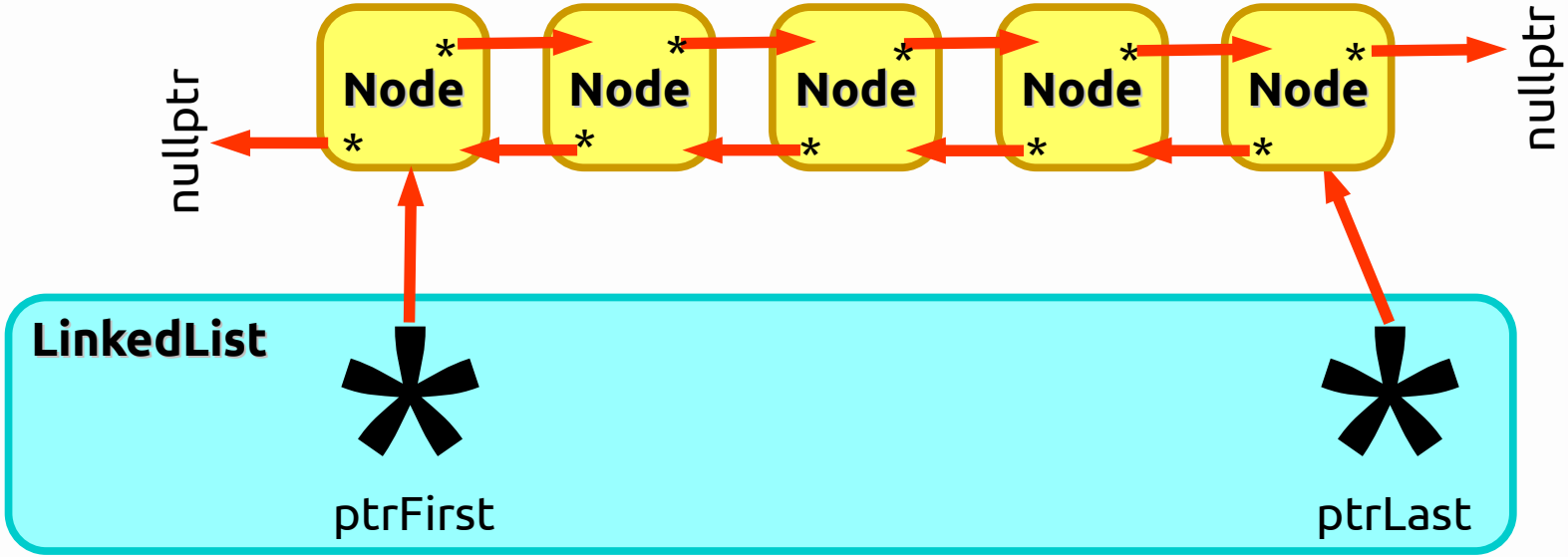
Linked List

- Wraps element data in “Nodes”
- Pointers are used to build a “chain”

3. How a Linked List works

This also means we don't have to pause and resize anything when a new item is added; we don't have to maintain the elements' contiguousness.

The efficiency of pushing on a new item is the same every time.



Notes

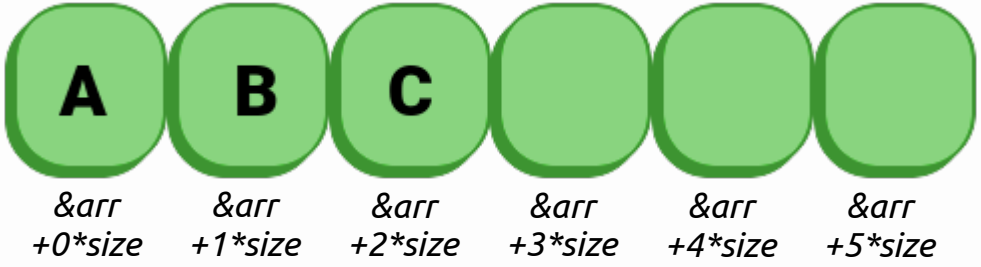
Dynamic Array

- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

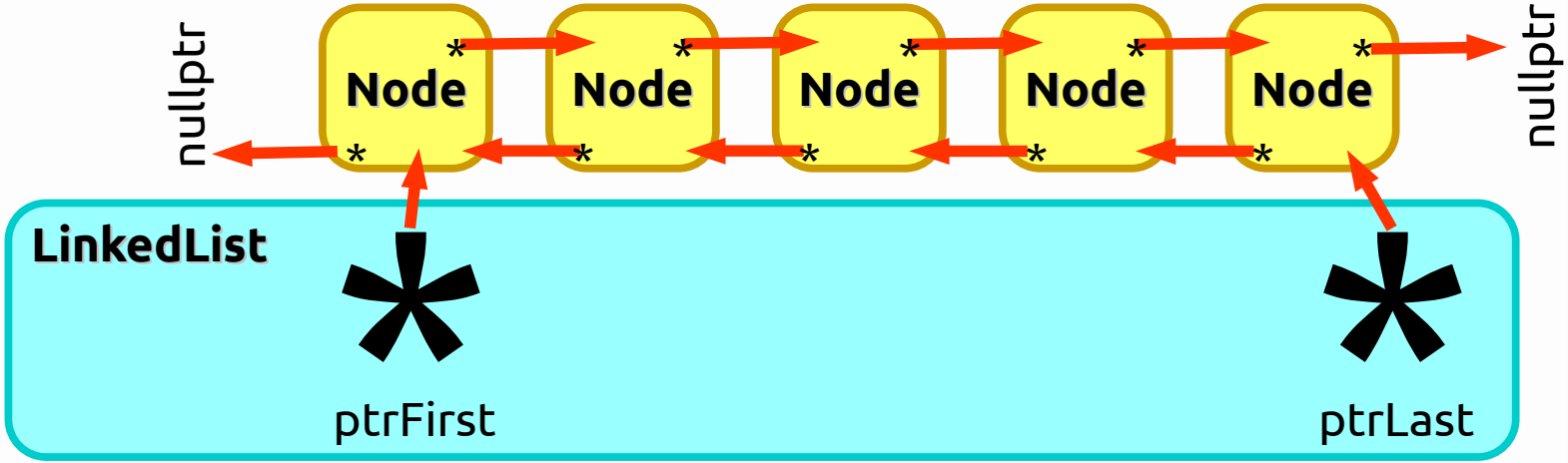
Linked List

- Wraps element data in "Nodes"
- Pointers are used to build a "chain"
- Elements are non-contiguous

3. How a Linked List works



However, because they are non-contiguous, we cannot access items as simply as with arrays.



Notes

Dynamic Array

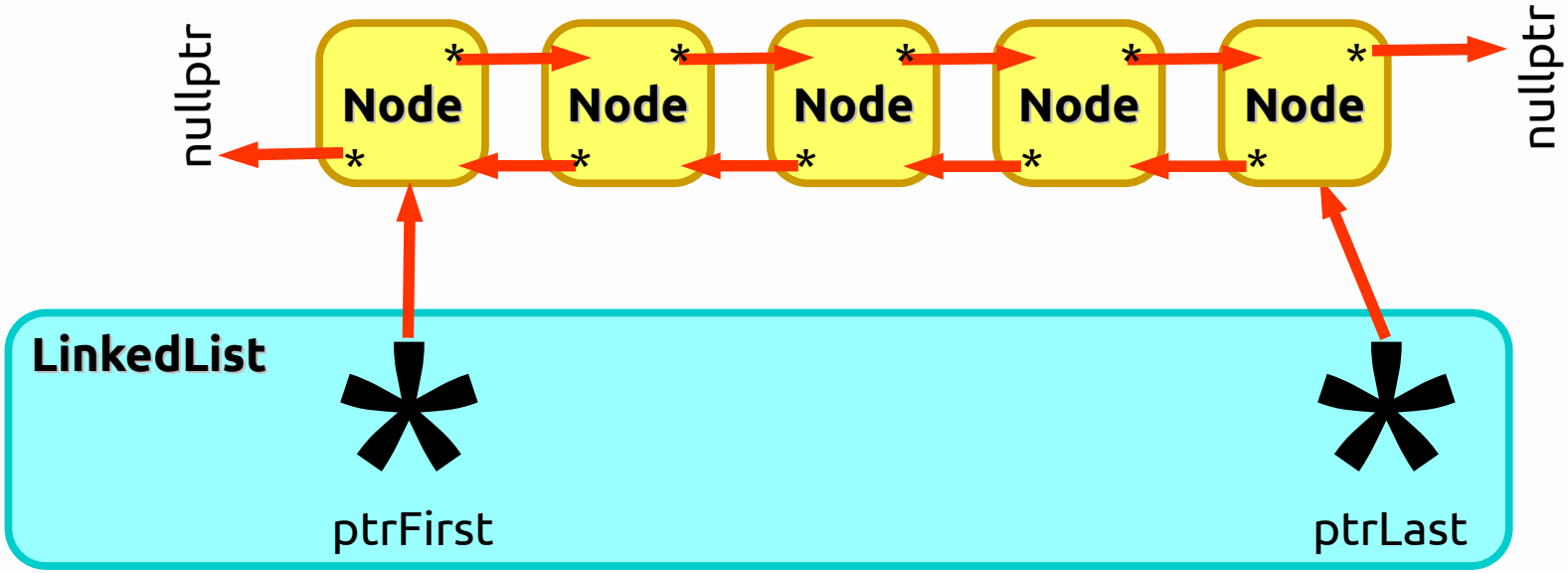
- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

Linked List

- Wraps element data in "Nodes"
- Pointers are used to build a "chain"
- Elements are non-contiguous

3. How a Linked List works

To find an item at some *position* in the list, we have to start at the beginning and “walk” over to it.



Notes

Dynamic Array

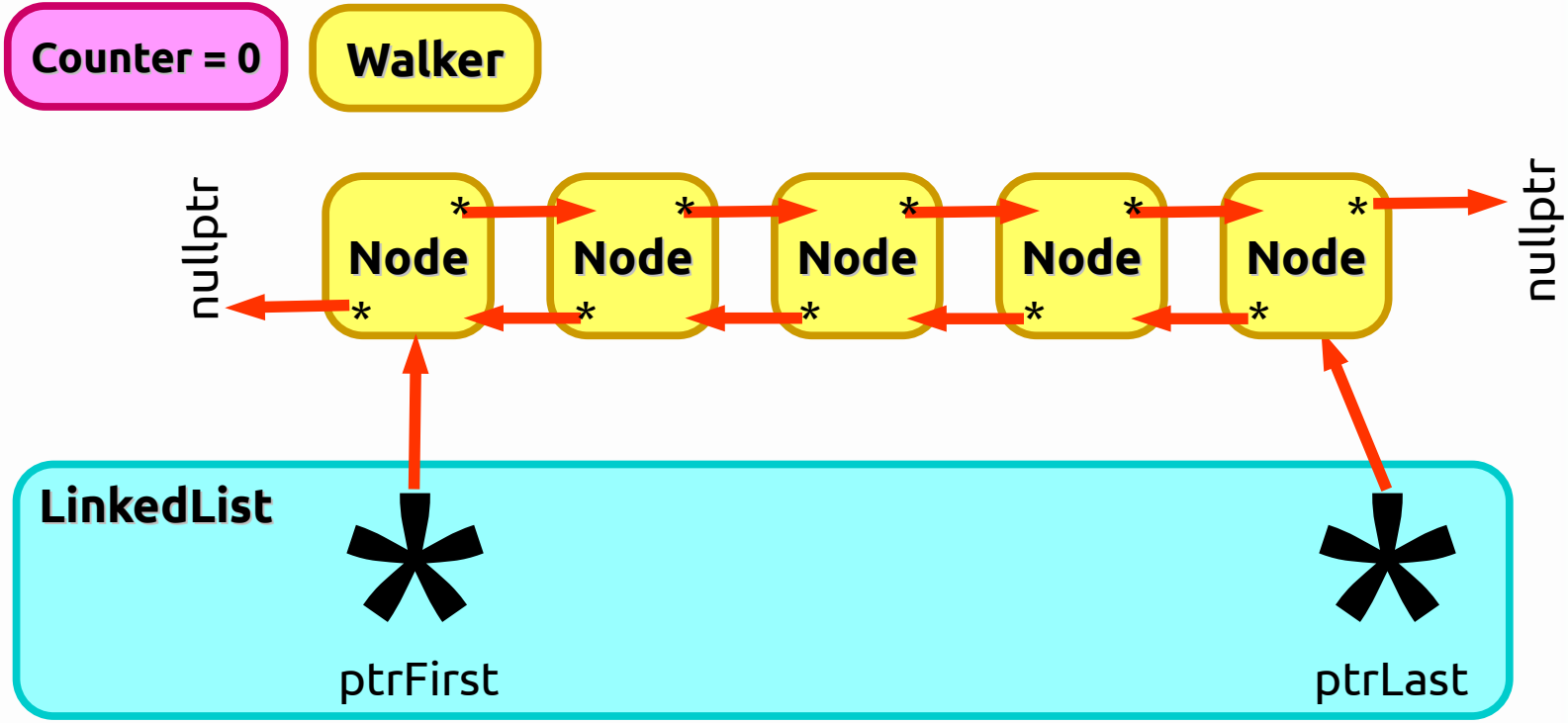
- Resize is costly
- Access is cheap
- Whether to use is a *design decision*.

Linked List

- Wraps element data in “Nodes”
- Pointers are used to build a “chain”
- Elements are non-contiguous
- Traversing the list is slower

3. How a Linked List works

We would create a pointer to keep track of the Node we're currently looking at, and an integer counter to keep track of how many Nodes we've looked at…



Notes

Dynamic Array

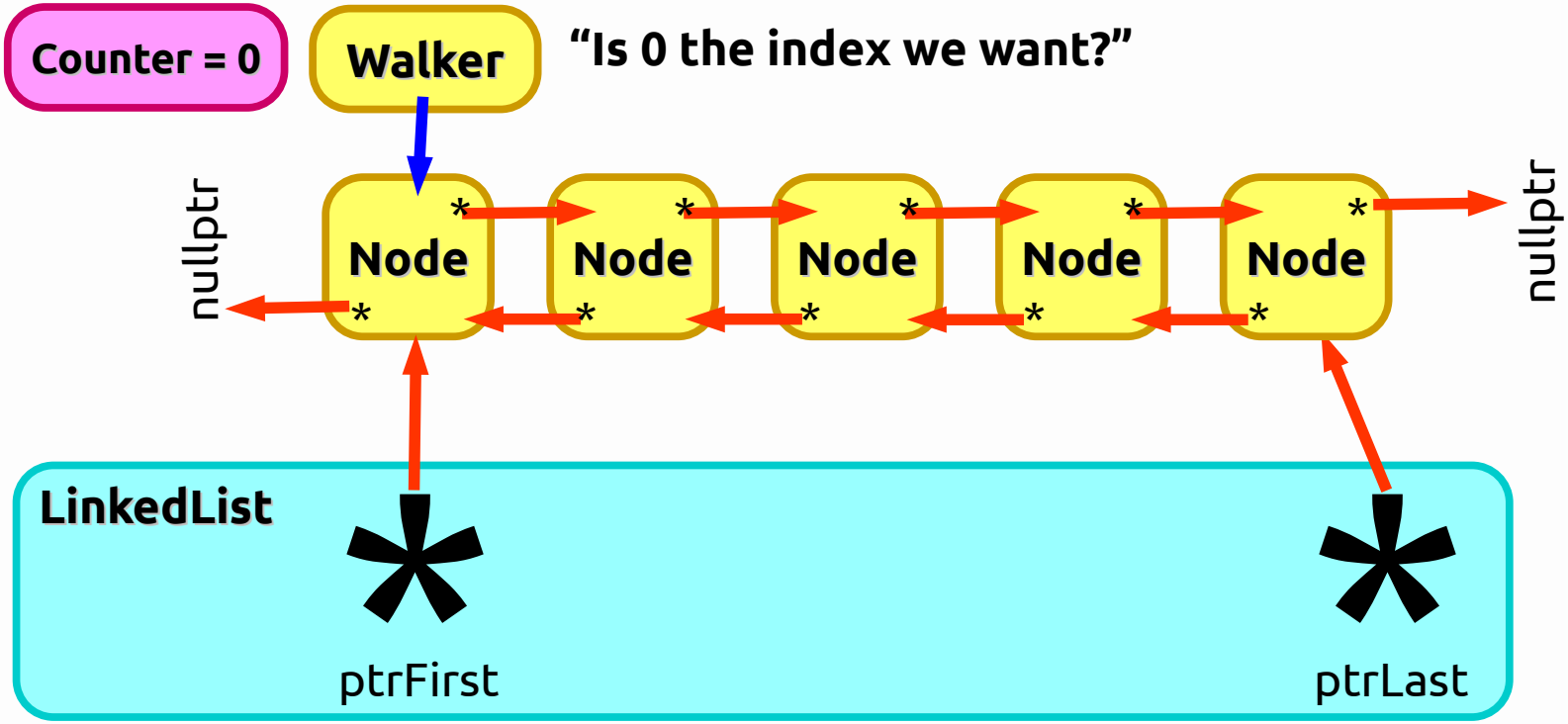
- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

Linked List

- Wraps element data in "Nodes"
- Pointers are used to build a "chain"
- Elements are non-contiguous
- Traversing the list is slower

3. How a Linked List works

We would begin by having our “Walker” pointer point to the first item, and our counter set to 0.



Notes

Dynamic Array

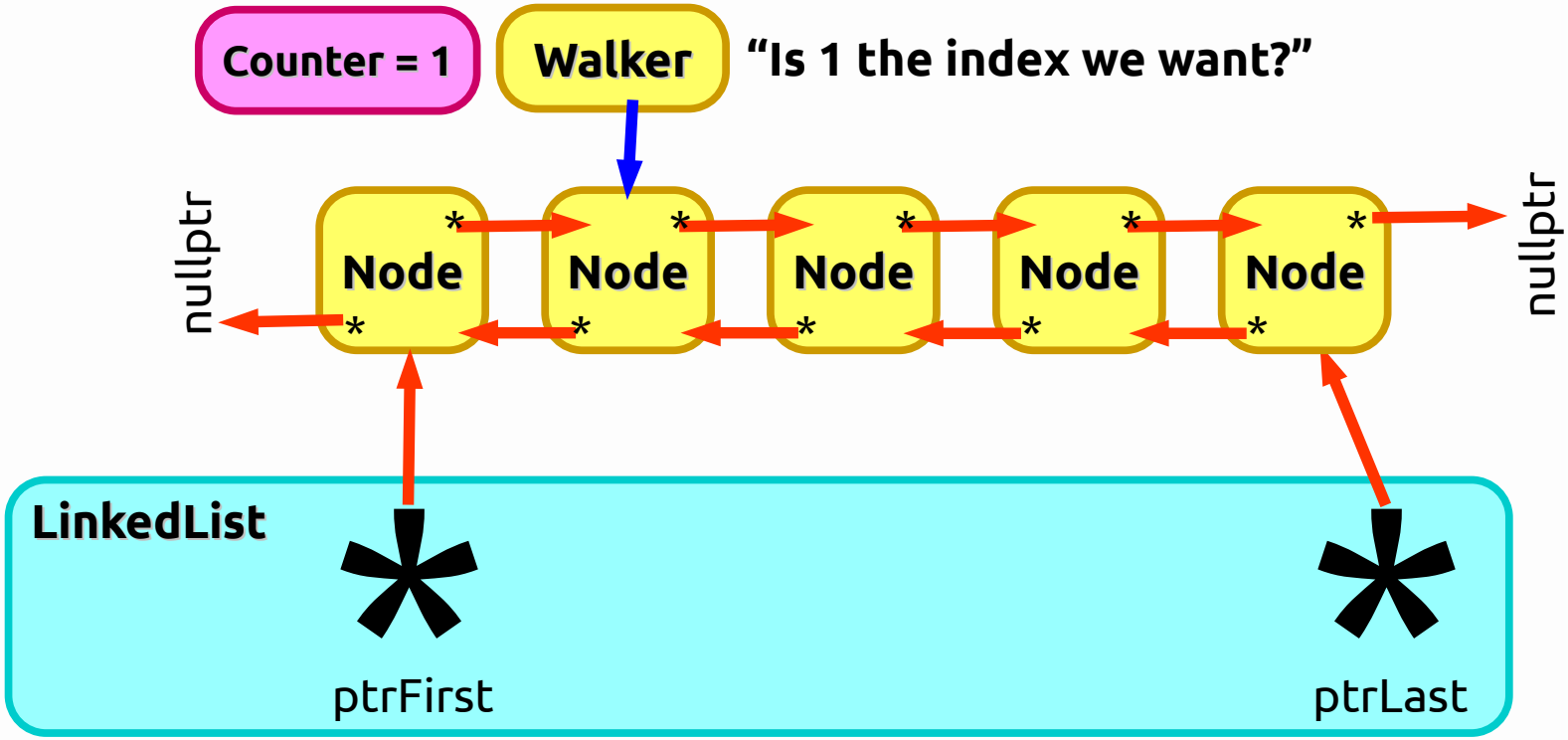
- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

Linked List

- Wraps element data in “Nodes”
- Pointers are used to build a “chain”
- Elements are non-contiguous
- Traversing the list is slower

3. How a Linked List works

Since each Node points to the next Node, we simply update the Walker pointer as we go…



Notes

Dynamic Array

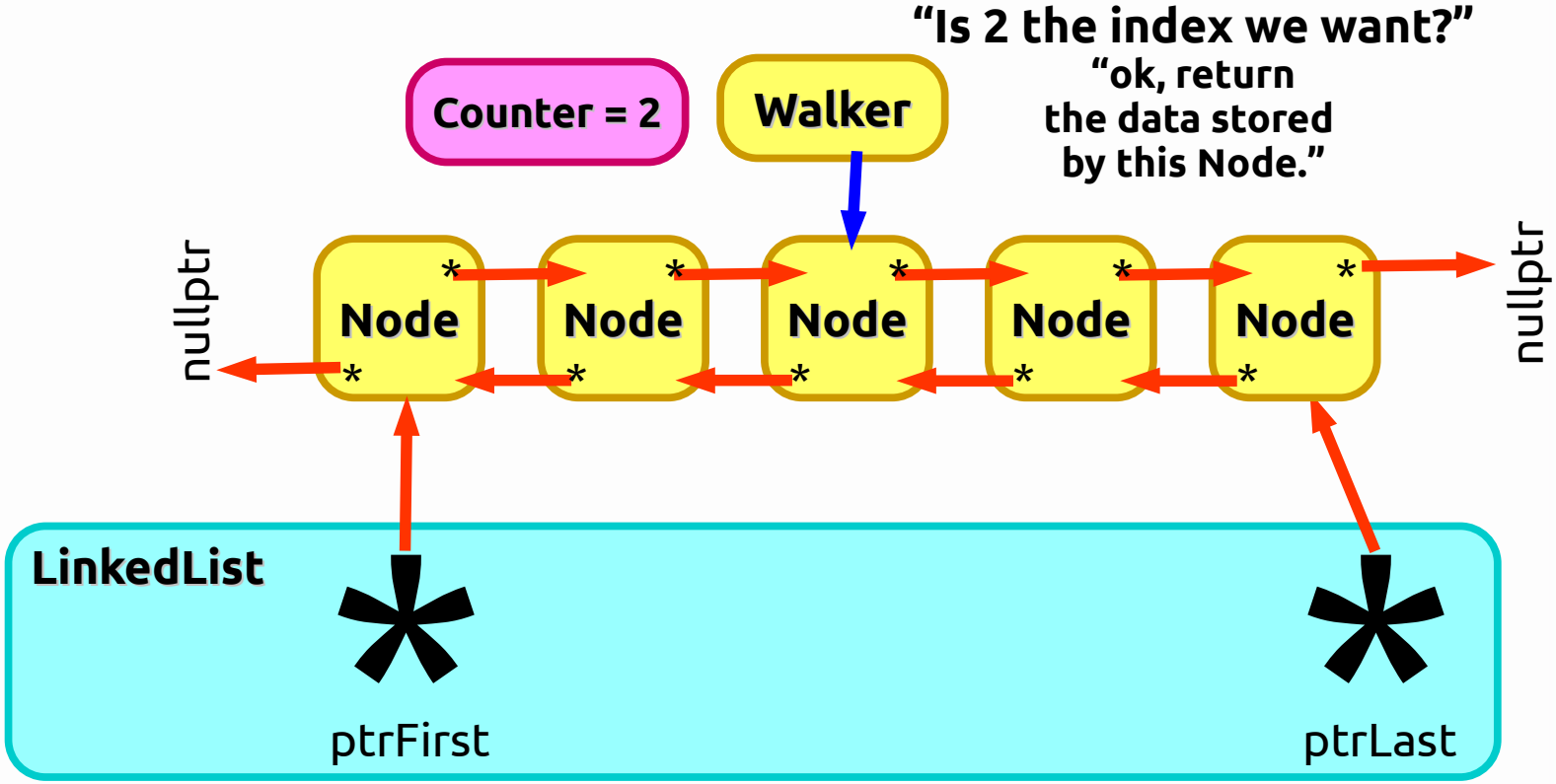
- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

Linked List

- Wraps element data in “Nodes”
- Pointers are used to build a “chain”
- Elements are non-contiguous
- Traversing the list is slower

3. How a Linked List works

Since each Node points to the next Node, we simply update the Walker pointer as we go…



Notes

Dynamic Array

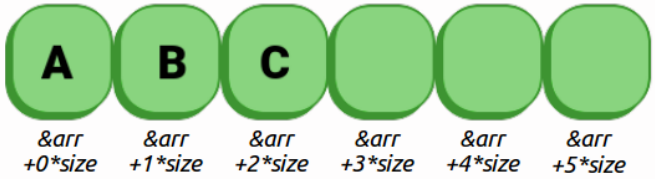
- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

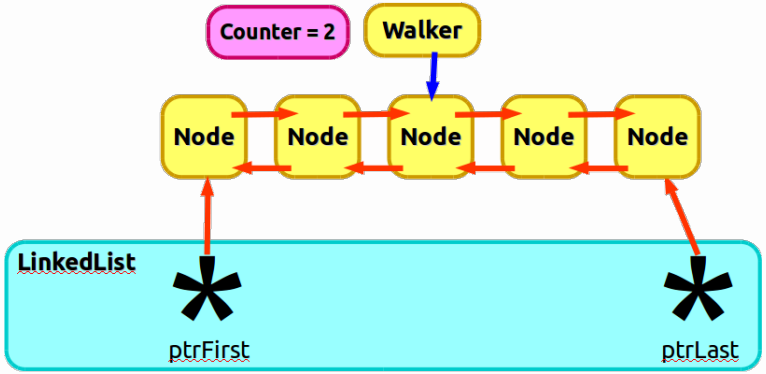
Linked List

- Wraps element data in "Nodes"
- Pointers are used to build a "chain"
- Elements are non-contiguous
- Traversing the list is slower

3. How a Linked List works

We will go over this functionality step-by-step in a moment, but because we have to start at the first Node, and keep moving forward by 1 until we get where we need to go, the **access time** of a Linked List is more costly than with an array.





Notes

Dynamic Array

- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

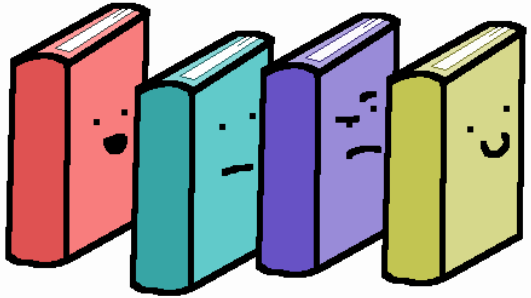
Linked List

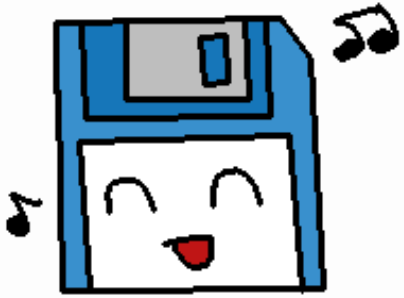
- Wraps element data in “Nodes”
- Pointers are used to build a “chain”
- Elements are non-contiguous
- Traversing the list is slower

3. How a Linked List works

Selecting one or the other means having to make the design decision: Are you going to be **accessing items** more often, or **inserting data** more often?

The idea of coming up with different ways to add, store, and access data, and the efficiency of each, is the central theme of data structures.





Notes

Dynamic Array

- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

Linked List

- Wraps element data in “Nodes”
- Pointers are used to build a “chain”
- Elements are non-contiguous
- Traversing the list is slower

3. How a Linked List works

One last thing relating to Linked Lists…
there are several different kinds!

We are going to implement a **Doubly Linked List**, where each **Node** points to its *next item* and *previous item*.

A **Singly Linked List** is where each **Node** only points forward to its next item.

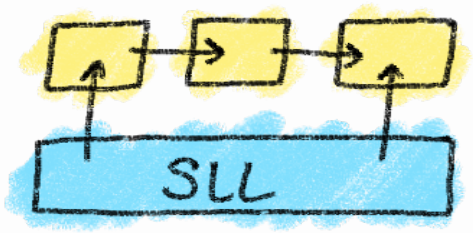
There are also **Circularly Linked Lists**, where the last Node of the list points to the first Node as its *next item*.

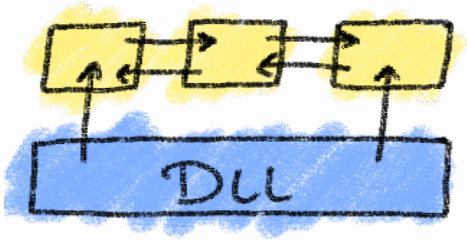
Notes

Dynamic Array
- Resize is costly
- Access is cheap
- Whether to use is a *design decision*.

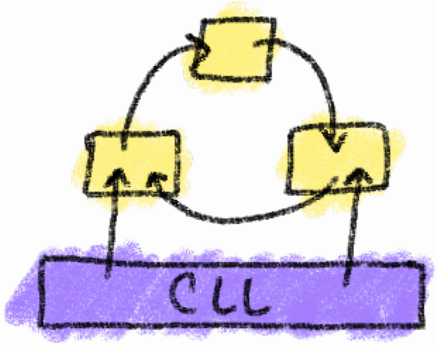
Linked List
- Wraps element data in “Nodes”
- Pointers are used to build a “chain”
- Elements are non-contiguous
- Traversing the list is slower

3. How a Linked List works





I think the Doubly Linked List is easiest to grasp, and if you are able to understand how it works, you can probably figure out how to write the same kinds of functions for Singly- and Circularly- Linked Lists.



Notes

Dynamic Array

- Resize is costly
- Access is cheap
- Whether to use is a *design decision*.

Linked List

- Wraps element data in “Nodes”
- Pointers are used to build a “chain”
- Elements are non-contiguous
- Traversing the list is slower

Stepping through Linked List functionality

4. Linked List Functionality

PushBack(“A”)
There are two scenarios we can run into with a function to add data: (1) We have an empty list, or (2) there is already something in the list.

We will have two different methods of dealing with each scenario.

Notes

4. Linked List Functionality

PushBack(“A”): List is empty

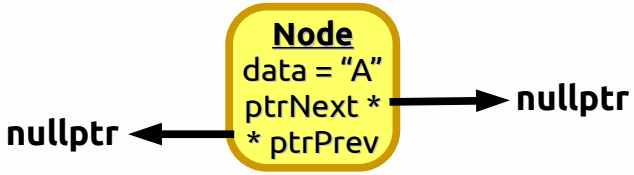
When a list is empty, we need to create a new node dynamically.

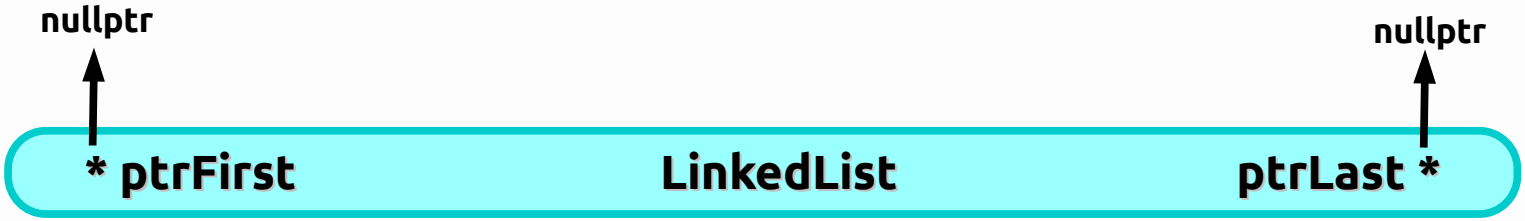
Notes

4. Linked List Functionality

PushBack(“A”): List is empty

When we create a new node, we set its **data** to whatever data was passed in that the user wants to store. ptrNext and ptrPrev should default to nullptr, but that can be handled in the Node’s constructor.



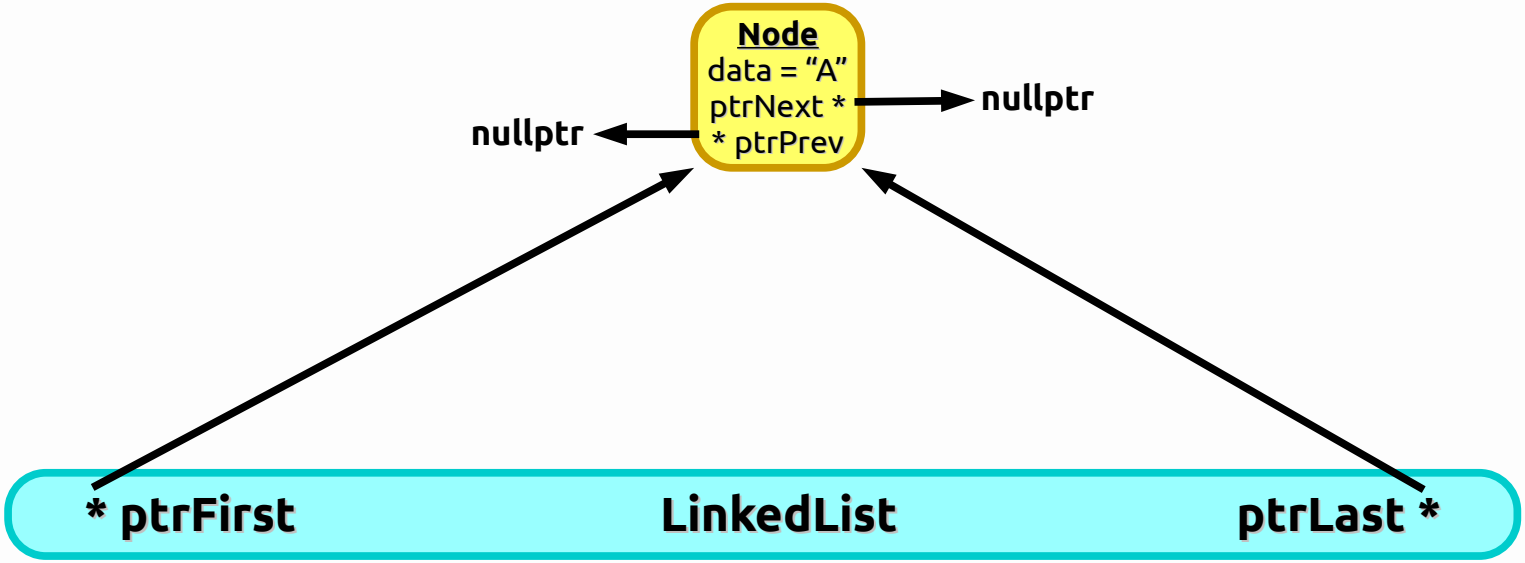


Notes

4. Linked List Functionality

PushBack(“A”): List is empty

After creating the Node, it’s just kind of floating around in memory. We need to associate it to the LinkedList. Since this is the first and only item in the list, you need to set the LinkedList’s ptrFirst and ptrLast to this new node.

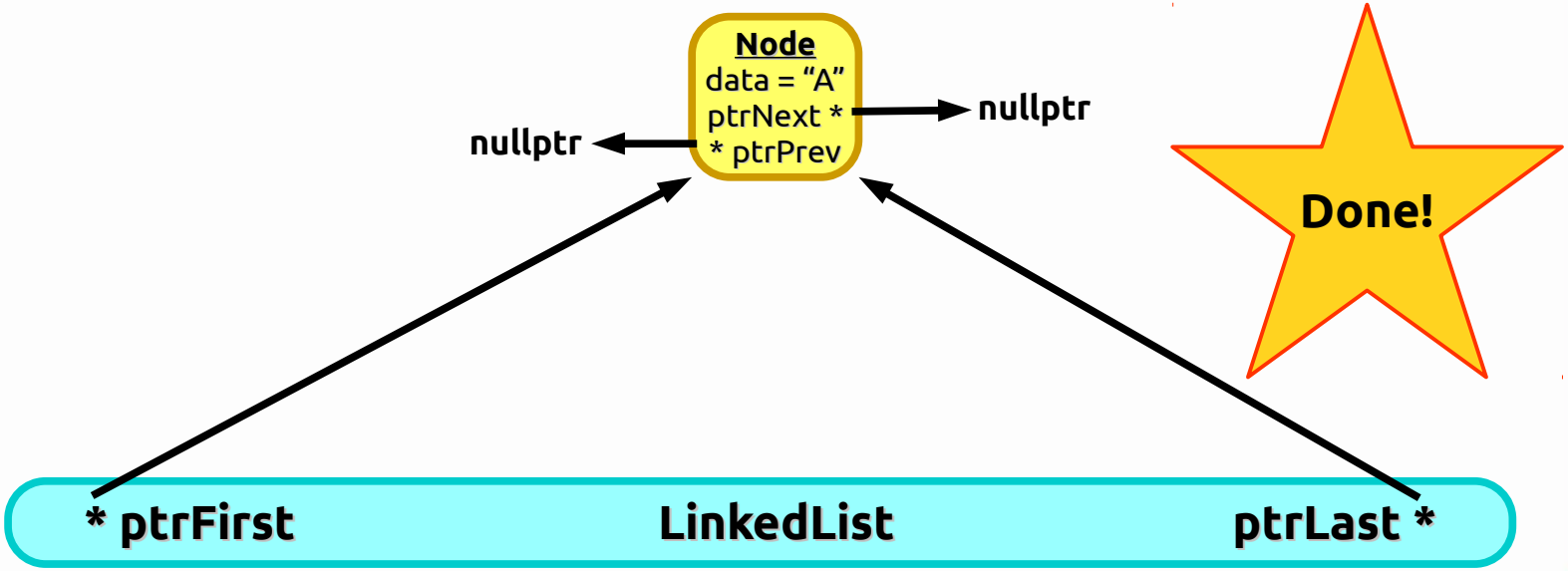


Notes

4. Linked List Functionality

PushBack(“A”): List is empty

After creating the Node, it’s just kind of floating around in memory. We need to associate it to the LinkedList. Since this is the first and only item in the list, you need to set the LinkedList’s ptrFirst and ptrLast to this new node. (Don’t forget to increment the itemCount of the Linked List)



Notes

4. Linked List Functionality

PushBack(“Z”): List is not empty

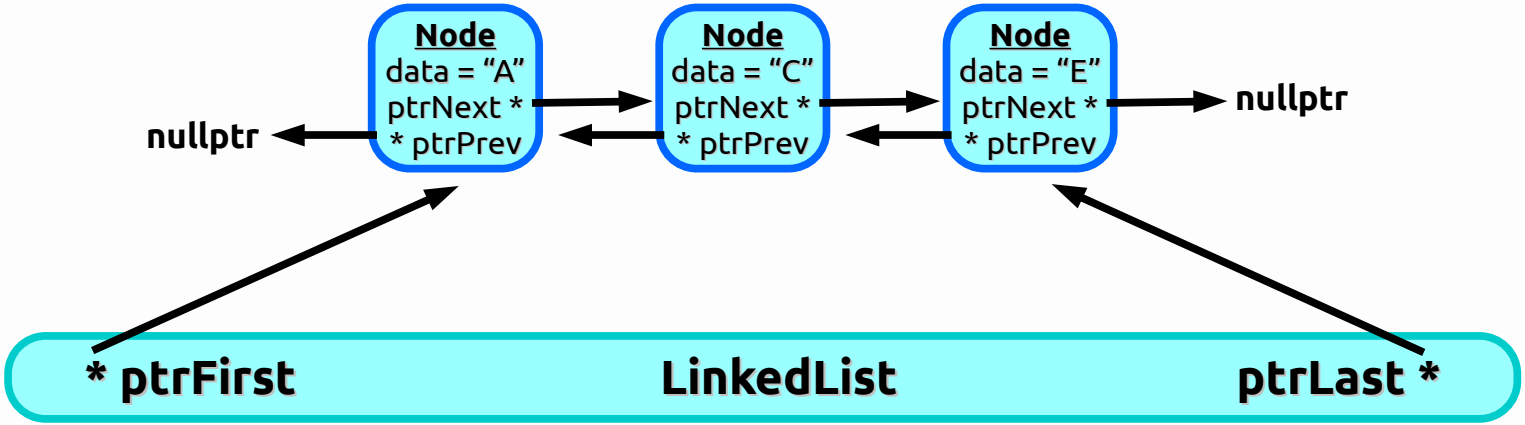
For the second scenario, there could be 1 or more items already in the List. Then the approach is a little different.

Notes

4. Linked List Functionality

PushBack("Z"): List is <u>not</u> empty

Here's a LinkedList that contains some items already. Now we're going to create a new node.

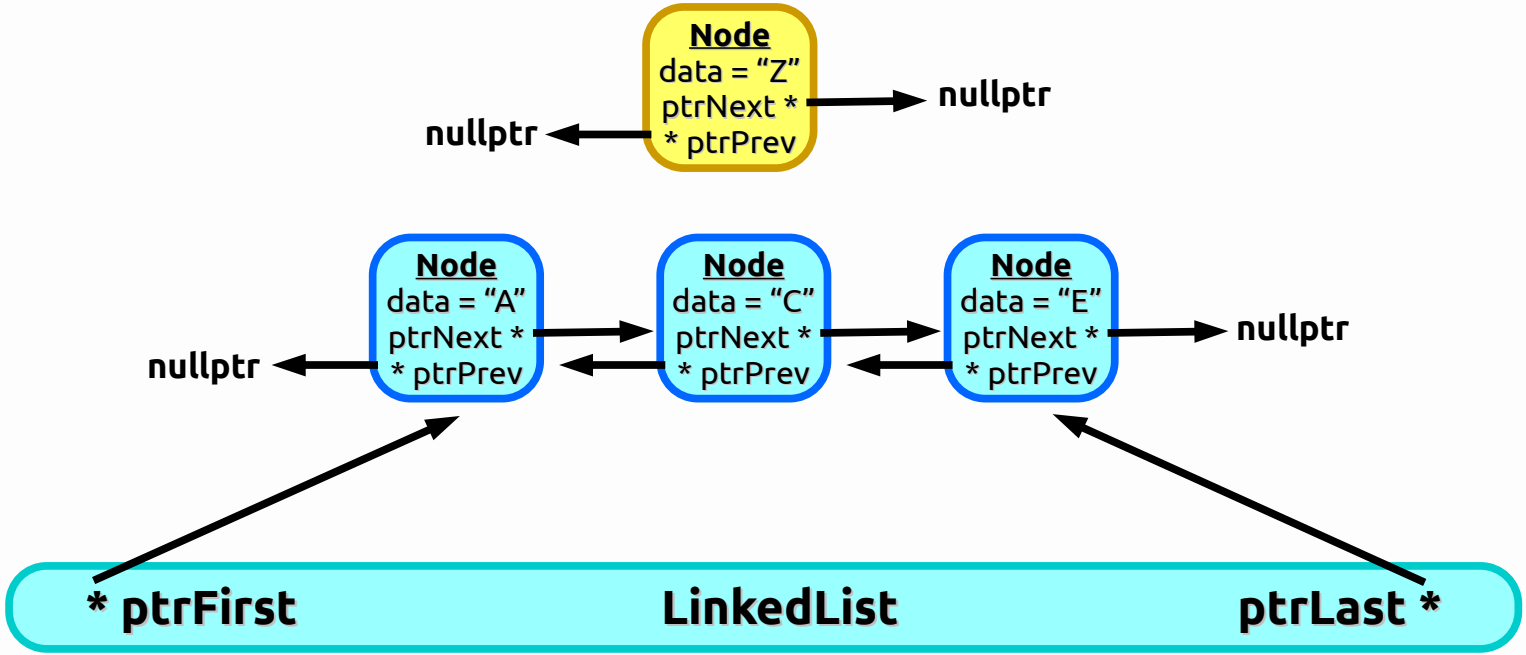


Notes

4. Linked List Functionality

PushBack(“Z”): List is <u>not</u> empty

With PushBack, we’re going to add this new element to the end of the list.

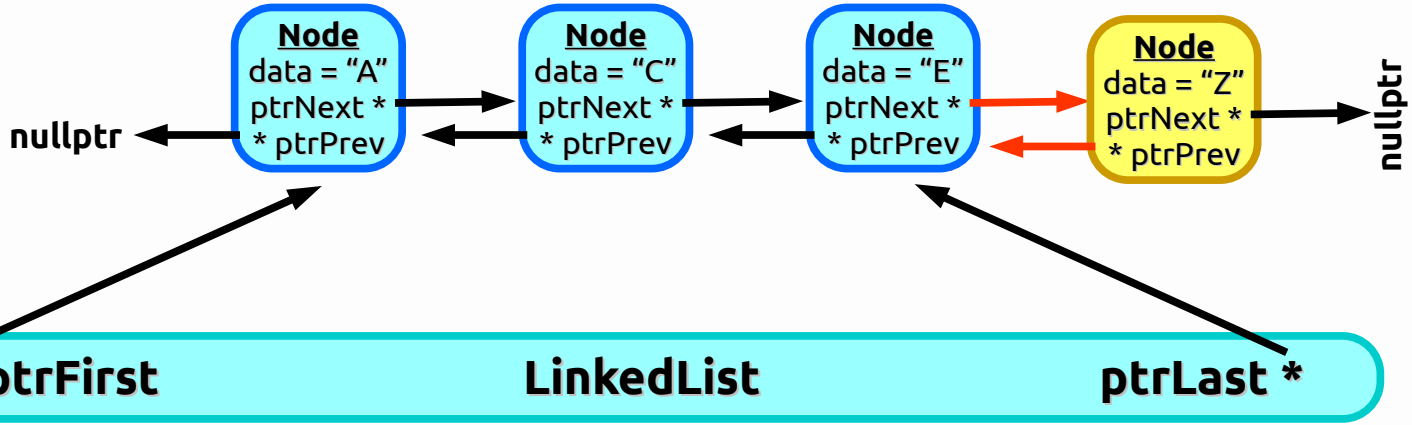


Notes

4. Linked List Functionality

PushBack("Z"): List is <u>not</u> empty

We have to set the current Last item's **ptrNext** to the new node, and set the new node's **ptrPrev** to the current Last item.

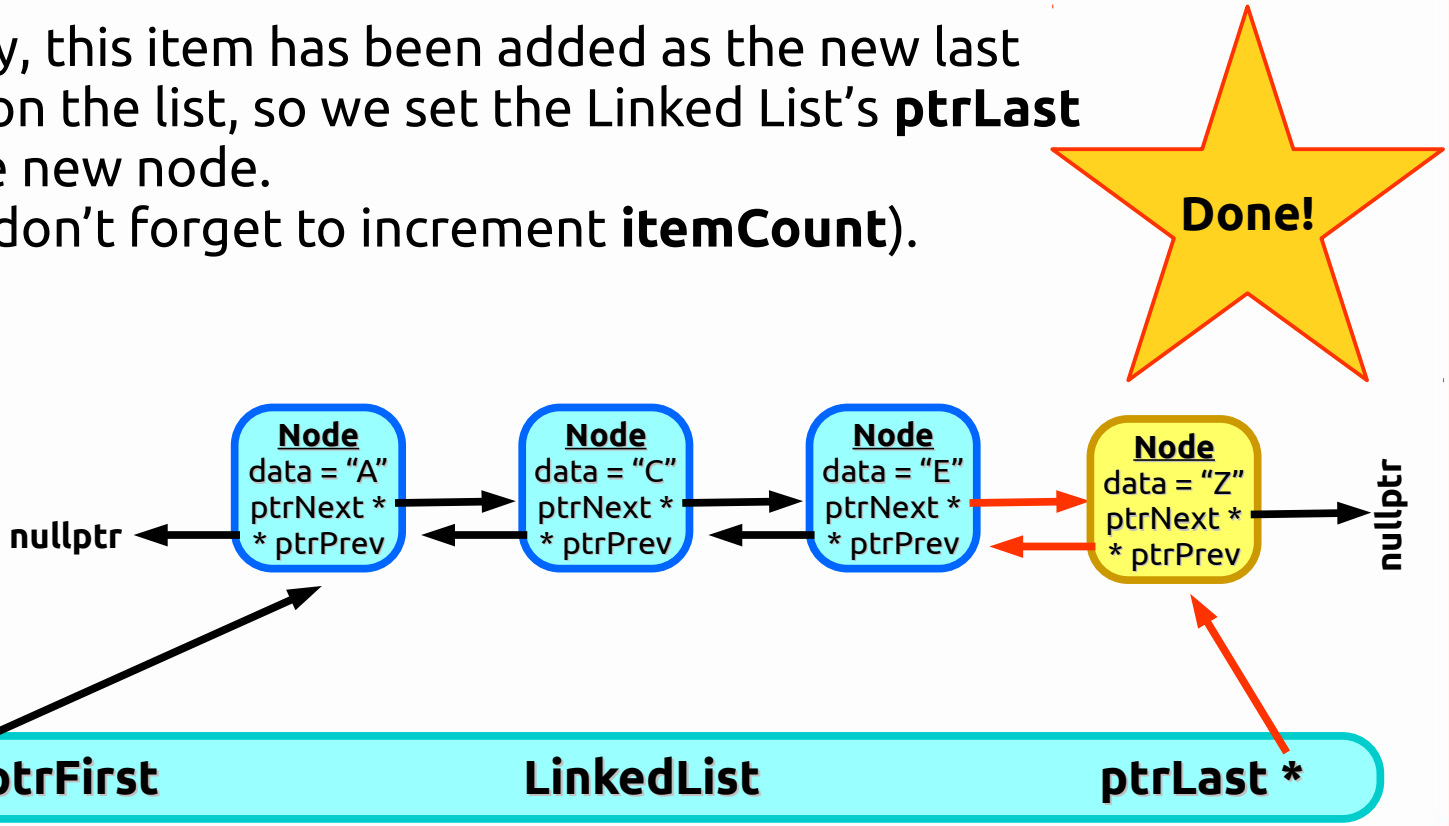


Notes

4. Linked List Functionality

PushBack("Z"): List is not empty

Finally, this item has been added as the new last item on the list, so we set the Linked List's **ptrLast** to the new node.
(And don't forget to increment **itemCount**).

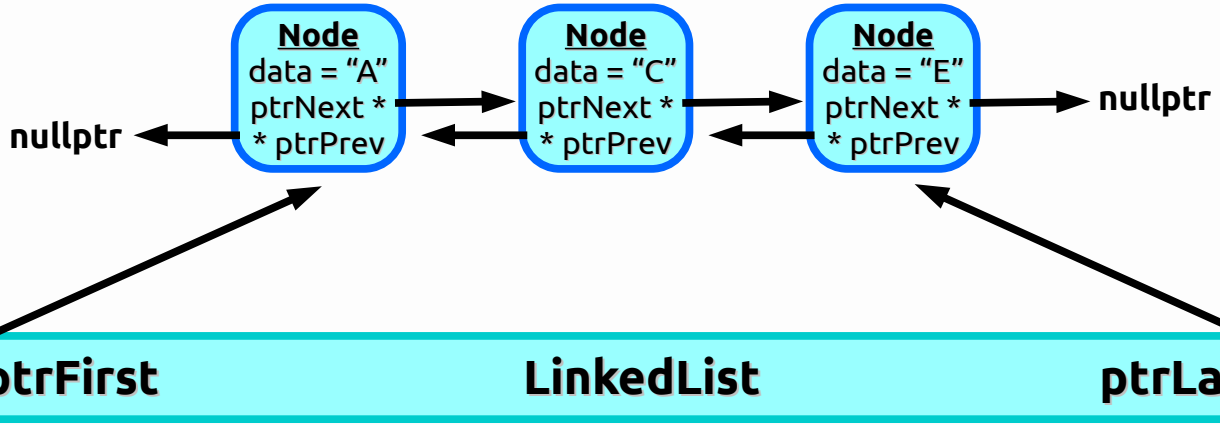


Notes

4. Linked List Functionality

GetBack()

The GetBack() function is just responsible for returning the item stored at the back of the list.

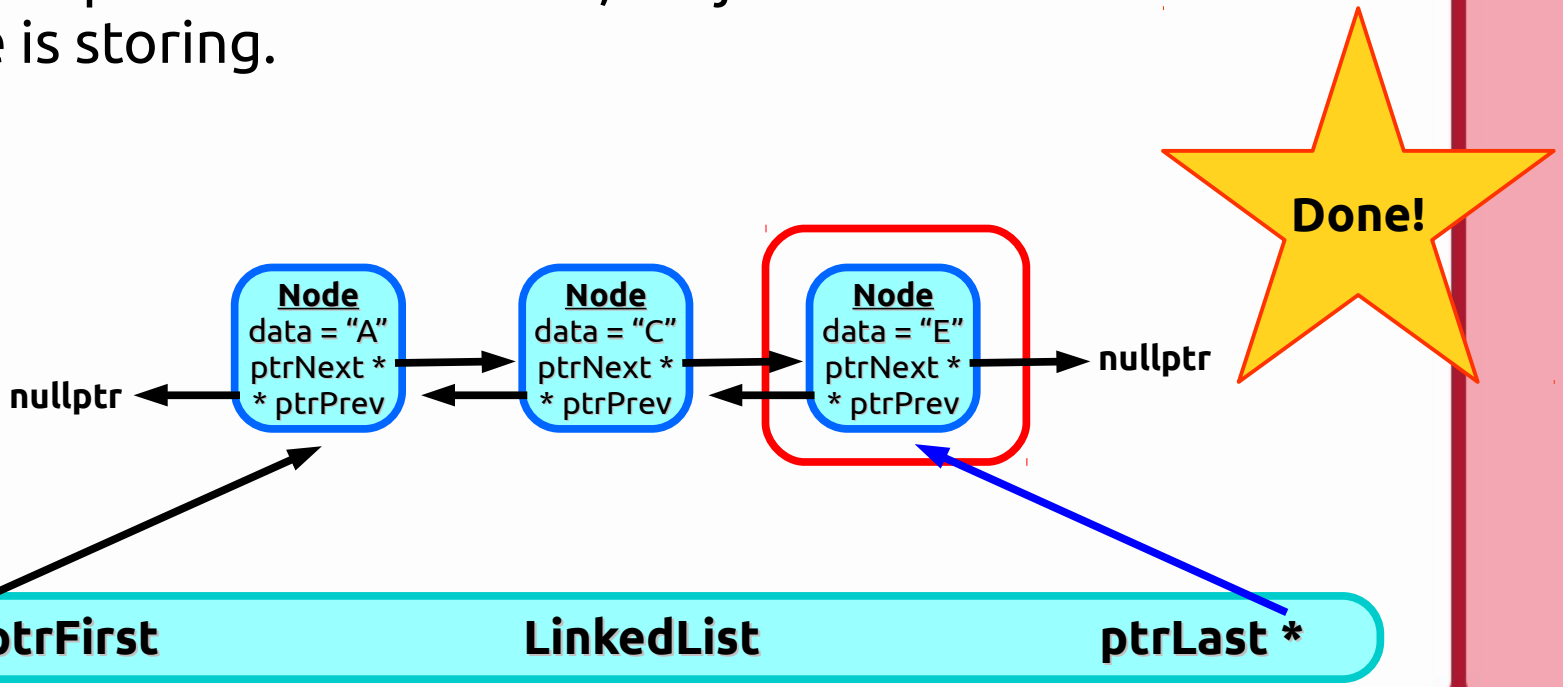


Notes

4. Linked List Functionality

GetBack()

We can access the Back-most item via the Linked List's **ptrLast** pointer. From there, we just return the data this Node is storing.



Notes

4. Linked List Functionality

PopBack()

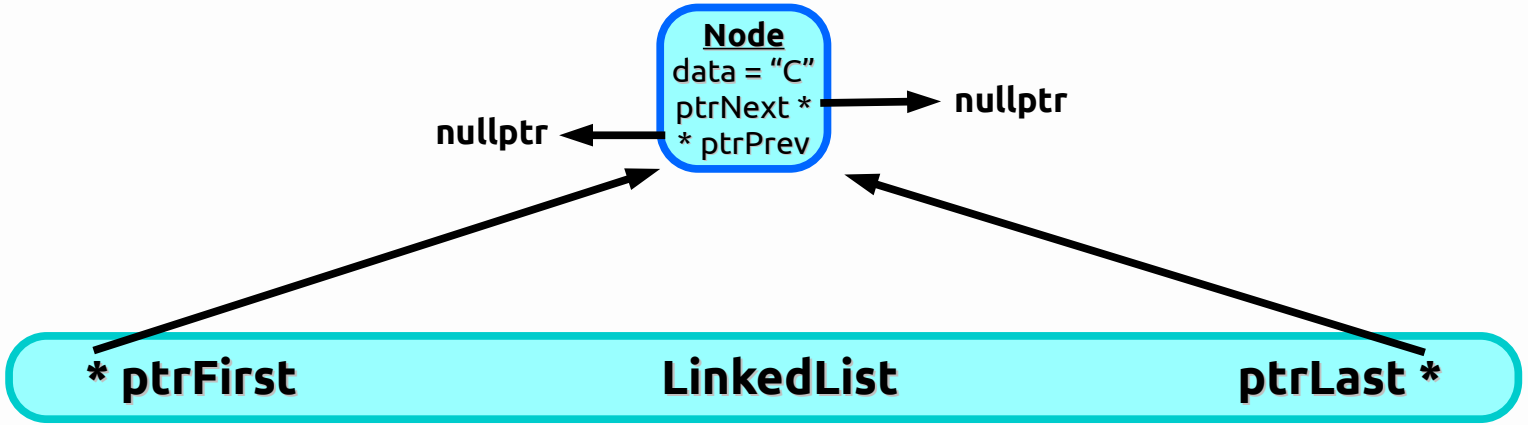
For PopBack, the two scenarios we care about is if we're removing the very last item in the list (only one item in the list) or if we're just removing an item in a list of more than 1.

Notes

4. Linked List Functionality

PopBack() **itemCount = 1**

If there's only one item in the list, then we delete it and update the LinkedList's **ptrFirst** and **ptrLast** to nullptr.



Notes

4. Linked List Functionality

PopBack() **itemCount = 1**

If there's only one item in the list, then we delete it and update the LinkedList's **ptrFirst** and **ptrLast** to nullptr.

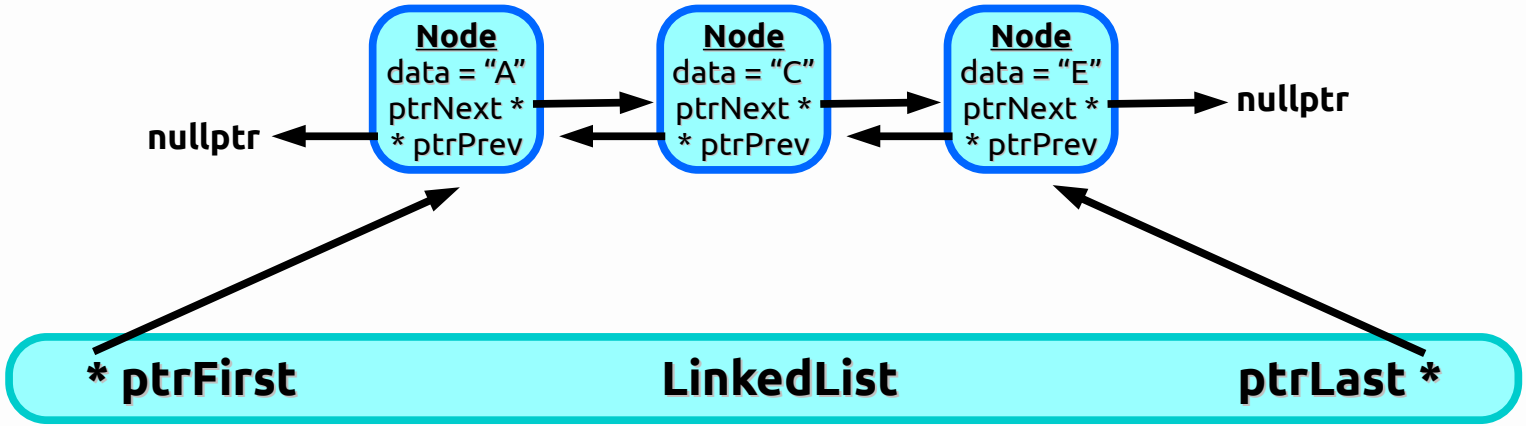
(Don't forget to decrement itemCount!)

Notes

4. Linked List Functionality

PopBack() **itemCount > 1**

For PopBack(), we need to remove the last-most item and set a new Last item via the LinkedList **ptrLast** pointer.

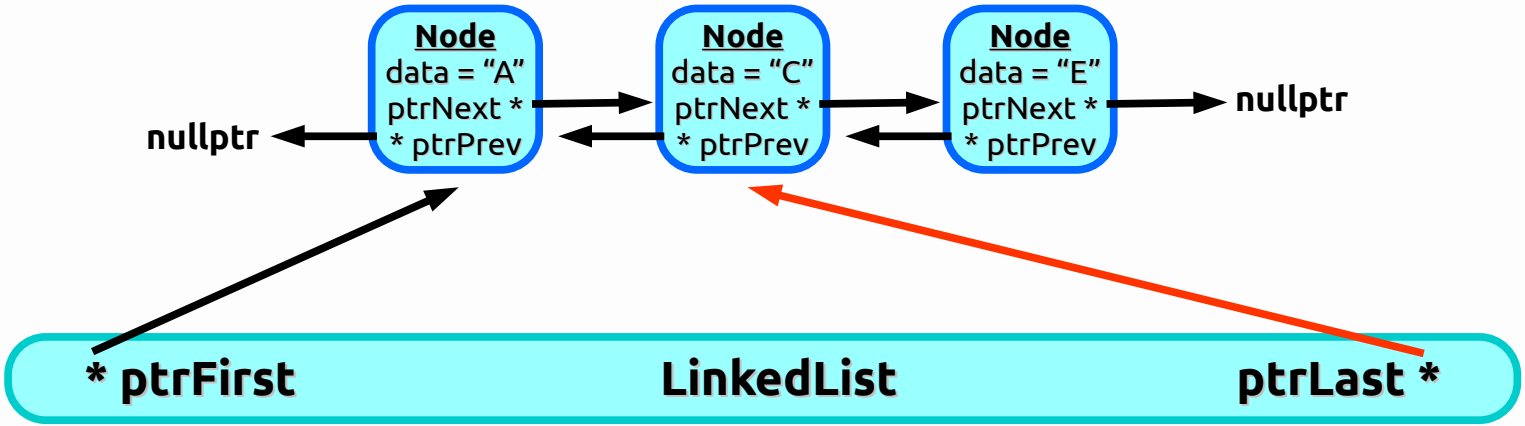


Notes

4. Linked List Functionality

PopBack() **itemCount > 1**

You can update **ptrLast** to point to its previous item first. Then, we can access the “old last” item via **ptrLast->ptrNext**

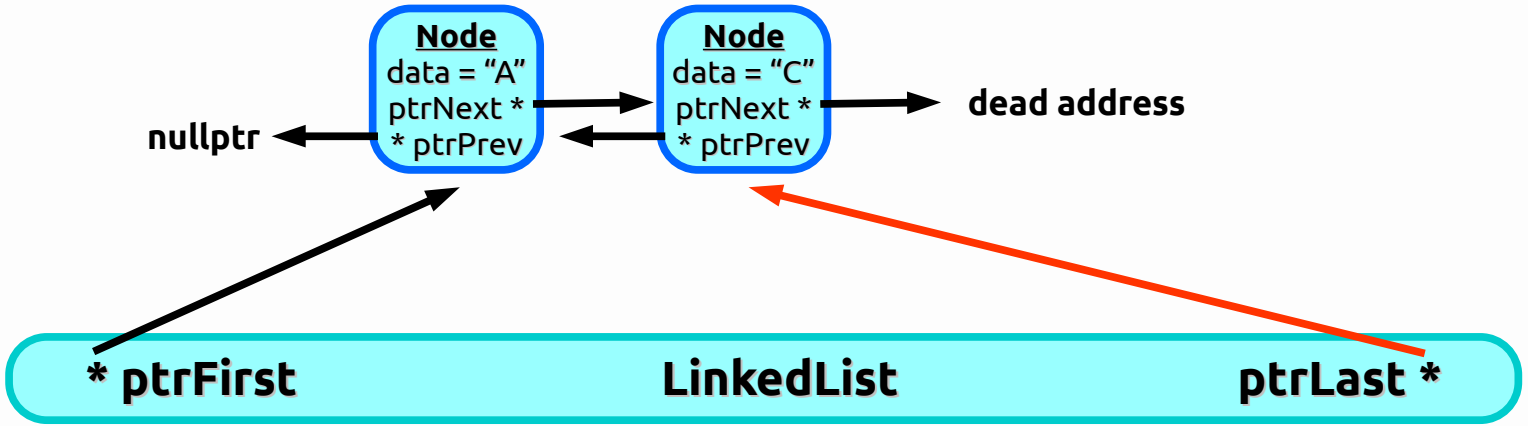


Notes

4. Linked List Functionality

PopBack() **itemCount > 1**

So then we **delete ptrLast->ptrNext**. That leaves us with a bad address being pointed to from **ptrLast->ptrNext**.



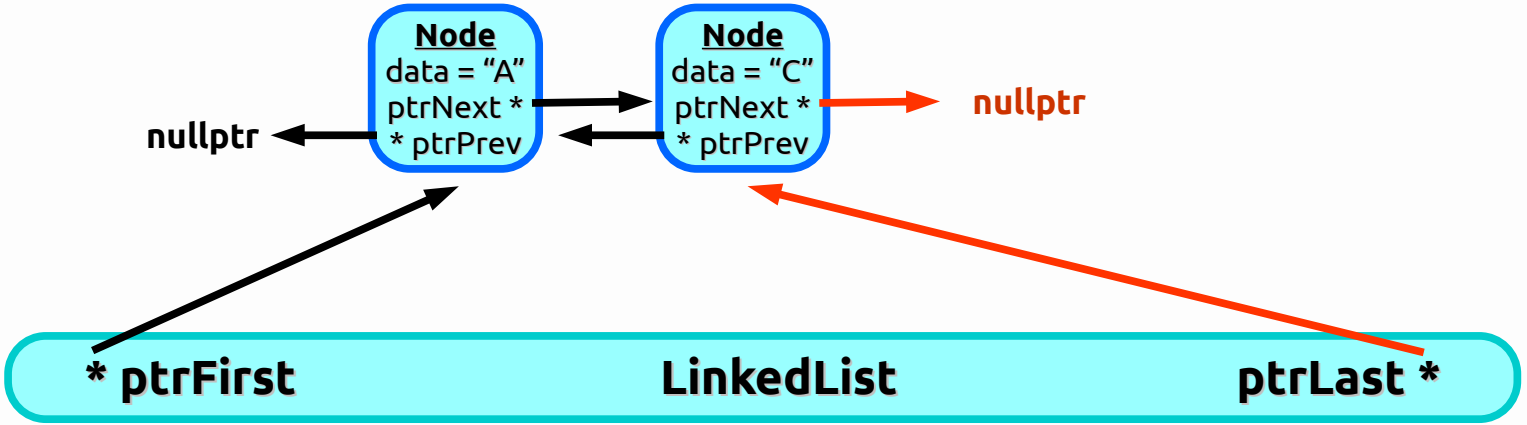
Notes

4. Linked List Functionality

PopBack() **itemCount > 1**

So finally, we set **ptrLast->ptrNext** to nullptr.

(And don't forget to decrement **itemCount**!)

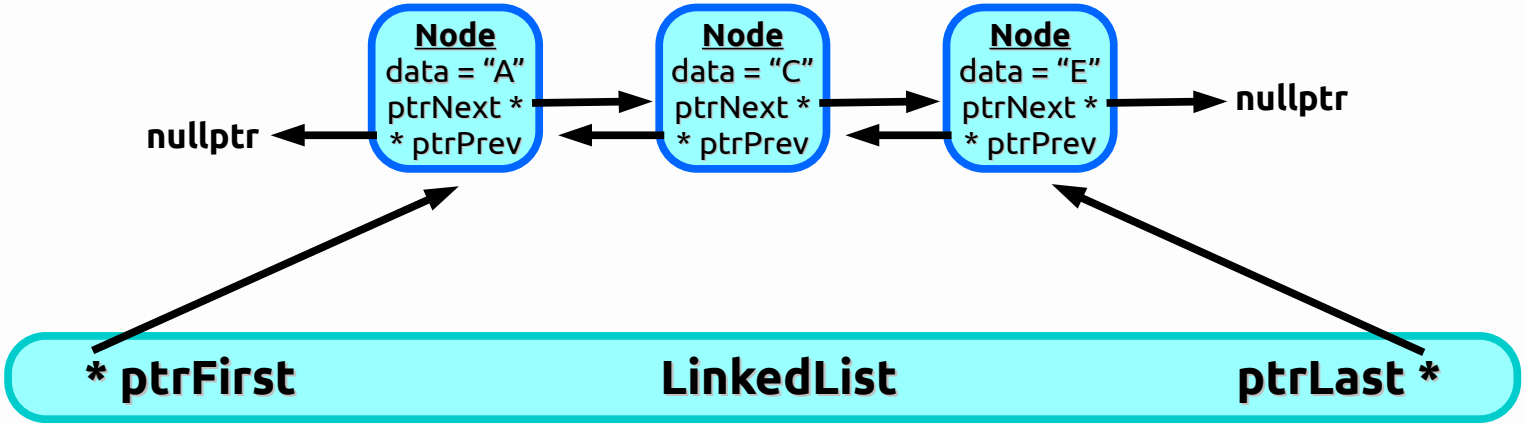


Notes

4. Linked List Functionality

PushFront(data)

Again, with PushFront, we might have an empty List or a non-empty List, and have two scenarios to deal with.



Notes

4. Linked List Functionality

PushFront(“A”): List is empty

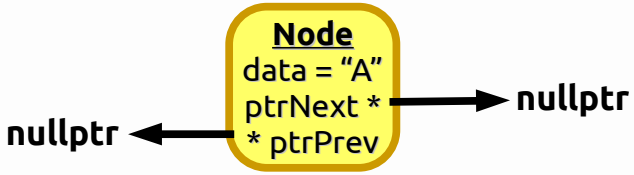
When a list is empty, we need to create a new node dynamically.

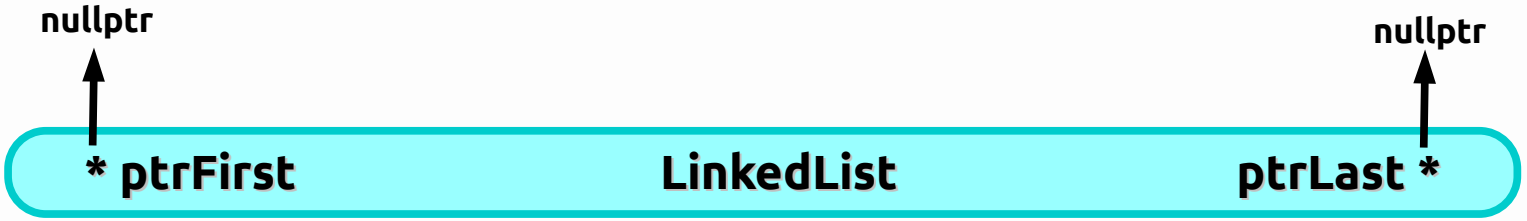
Notes

4. Linked List Functionality

PushFront(“A”): List is empty

When we create a new node, we set its **data** to whatever data was passed in that the user wants to store. ptrNext and ptrPrev should default to nullptr, but that can be handled in the Node’s constructor.



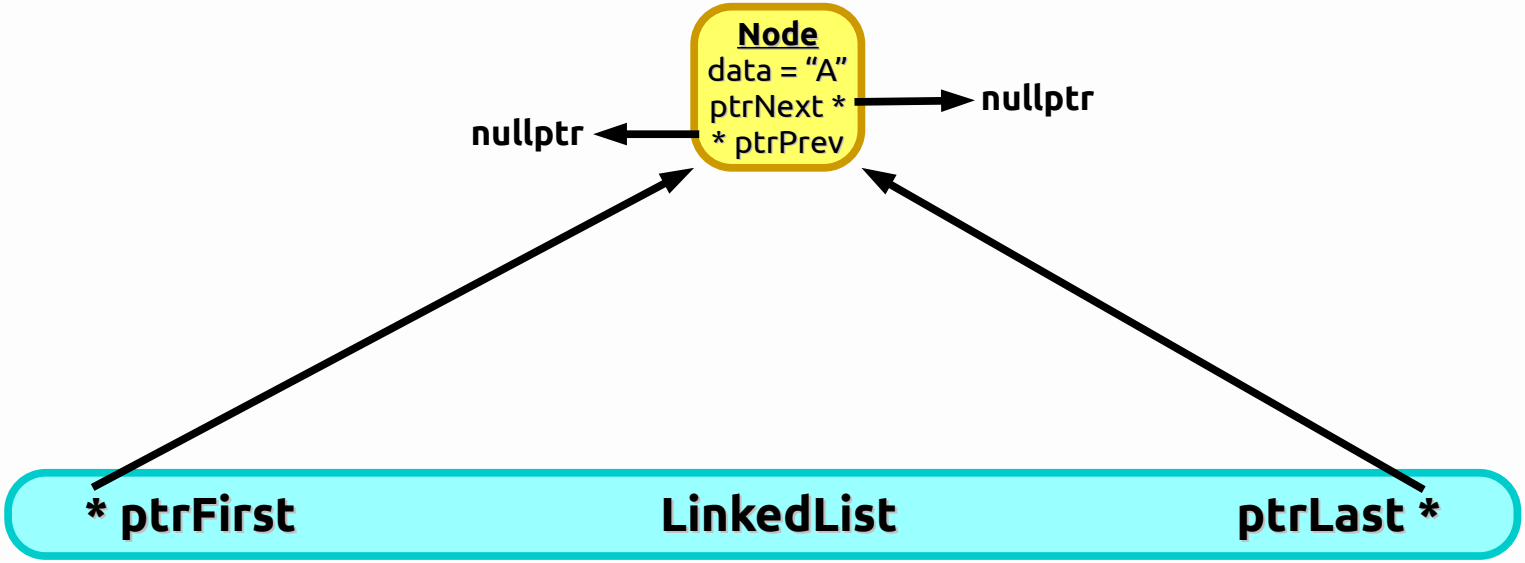


Notes

4. Linked List Functionality

PushFront(“A”): List is empty

After creating the Node, it’s just kind of floating around in memory. We need to associate it to the LinkedList. Since this is the first and only item in the list, you need to set the LinkedList’s ptrFirst and ptrLast to this new node.

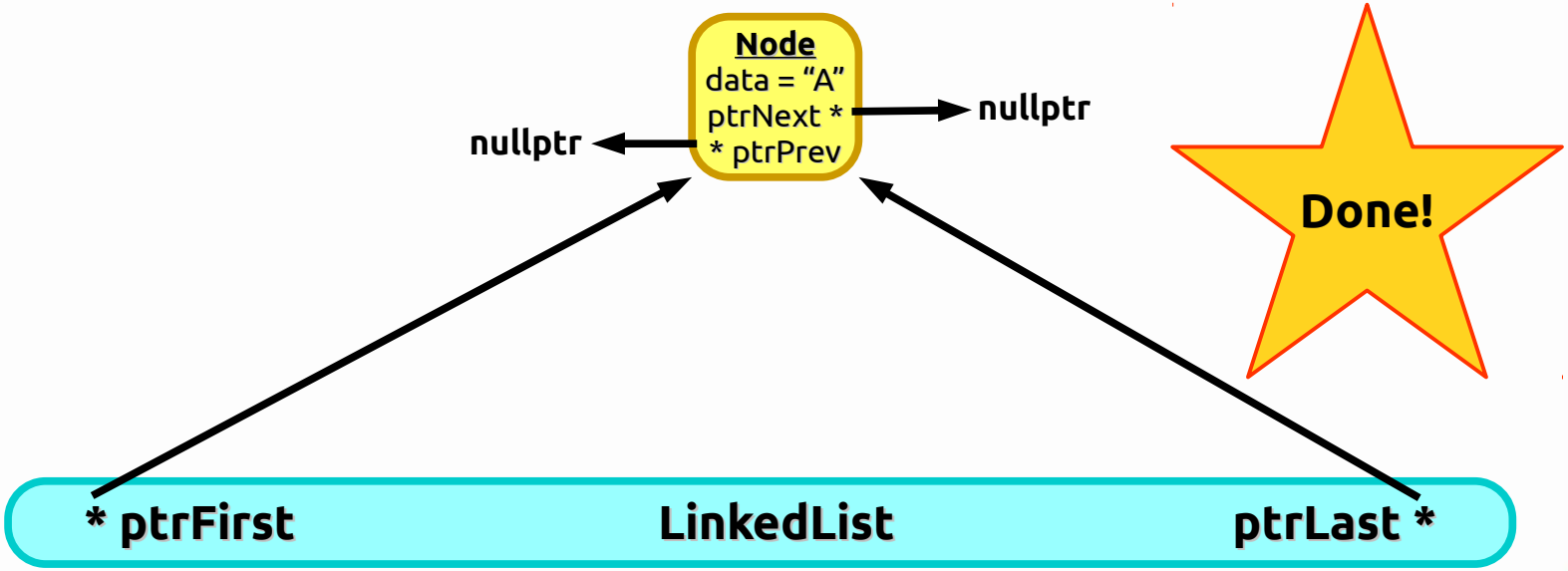


Notes

4. Linked List Functionality

PushFront(“A”): List is empty

After creating the Node, it’s just kind of floating around in memory. We need to associate it to the LinkedList. Since this is the first and only item in the list, you need to set the LinkedList’s ptrFirst and ptrLast to this new node. (Don’t forget to increment the itemCount of the Linked List)



Notes

4. Linked List Functionality

PushFront(“Z”): List is not empty

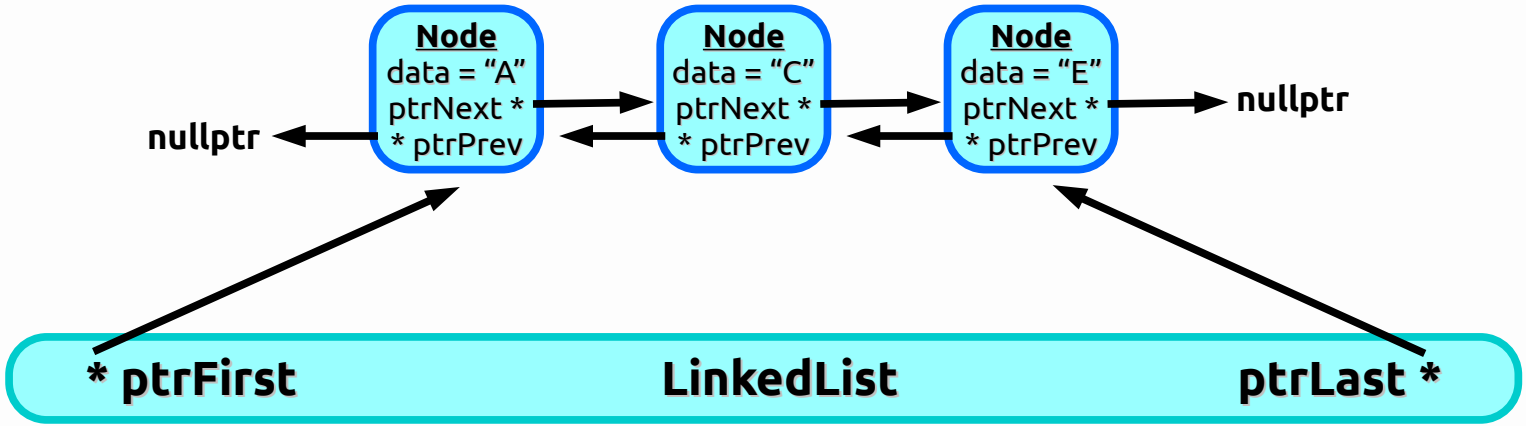
For the second scenario, there could be 1 or more items already in the List. Then the approach is a little different.

Notes

4. Linked List Functionality

PushFront(“Z”): List is <u>not</u> empty

Here’s a LinkedList that contains some items already. Now we’re going to create a new node.

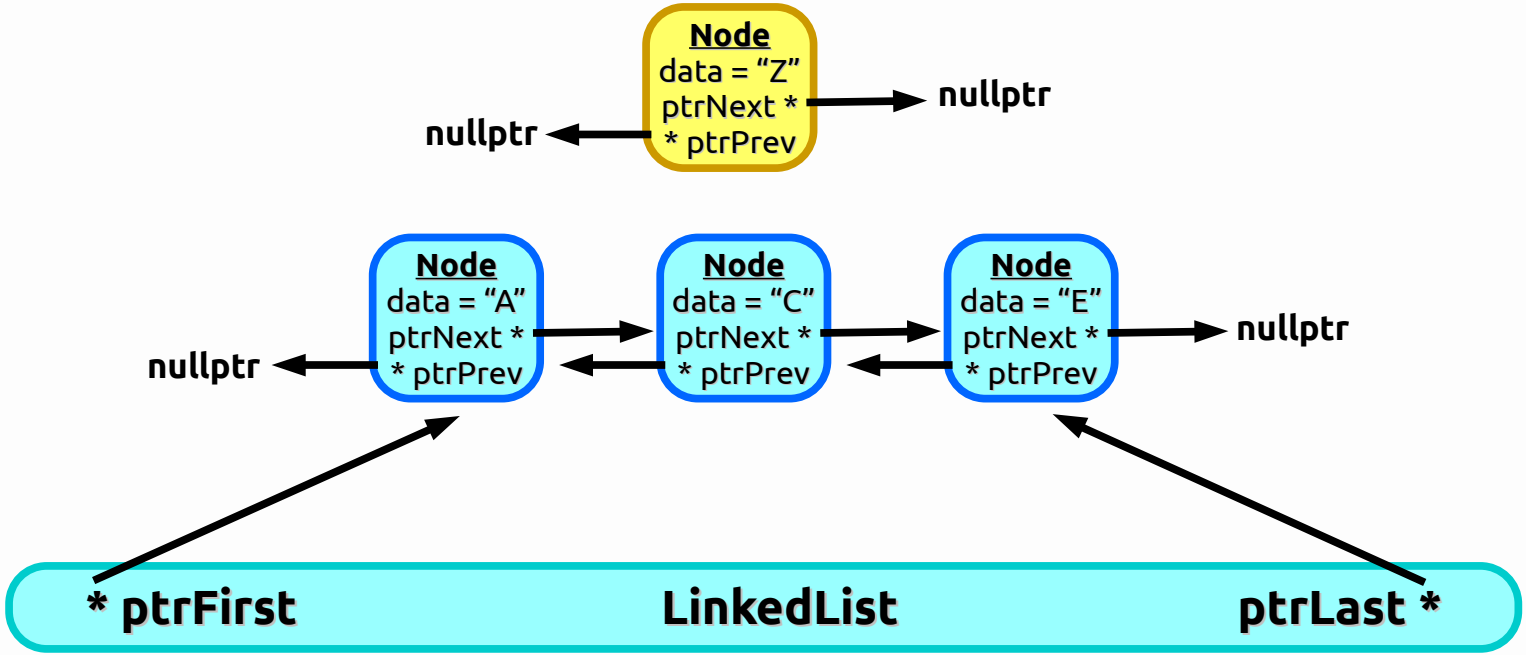


Notes

4. Linked List Functionality

PushFront(“Z”): List is <u>not</u> empty

With PushFront, we’re going to add this new element to the beginning of the list.

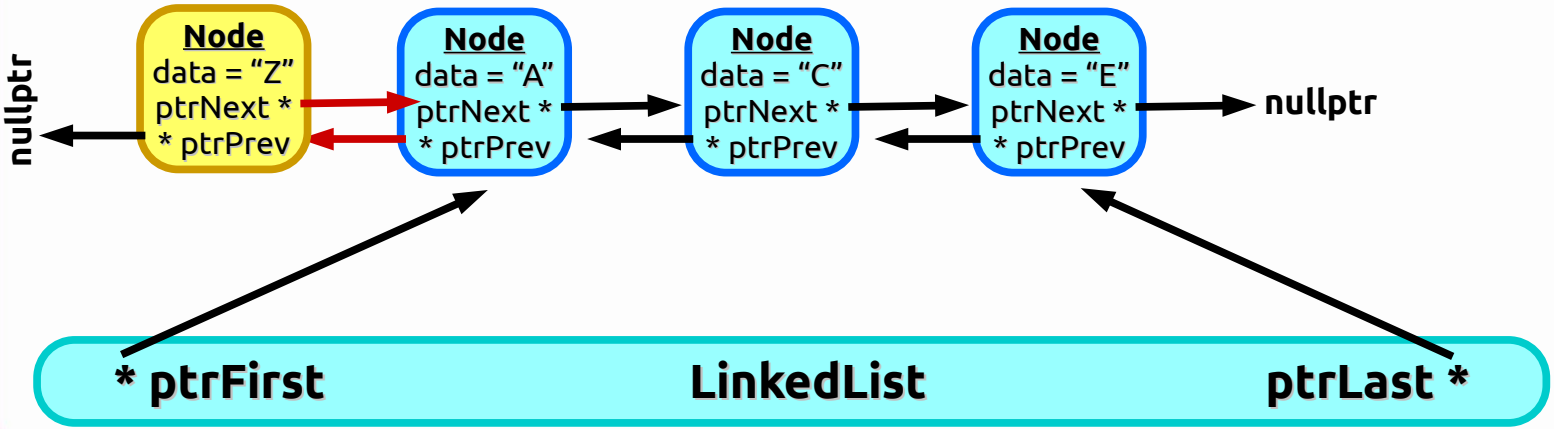


Notes

4. Linked List Functionality

PushFront(“Z”): List is <u>not</u> empty

After creating the node, we set the current **ptrFirst**’s **ptrPrev** pointer to the new node, and the new node’s **ptrNext** pointer to the list’s **ptrFirst** item.



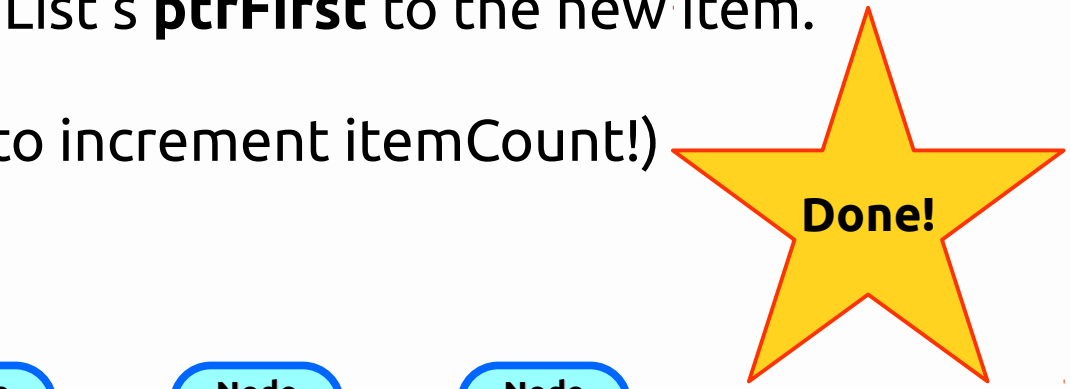
Notes

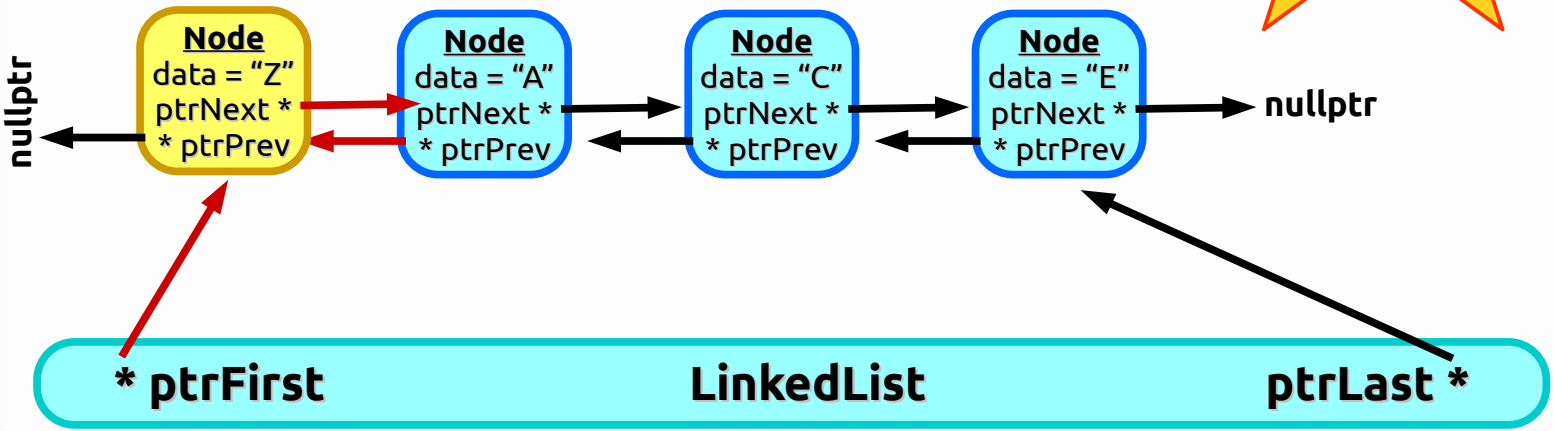
4. Linked List Functionality

PushFront("Z"): List is <u>not</u> empty

Finally, we set the List's **ptrFirst** to the new item.

(And don't forget to increment itemCount!)



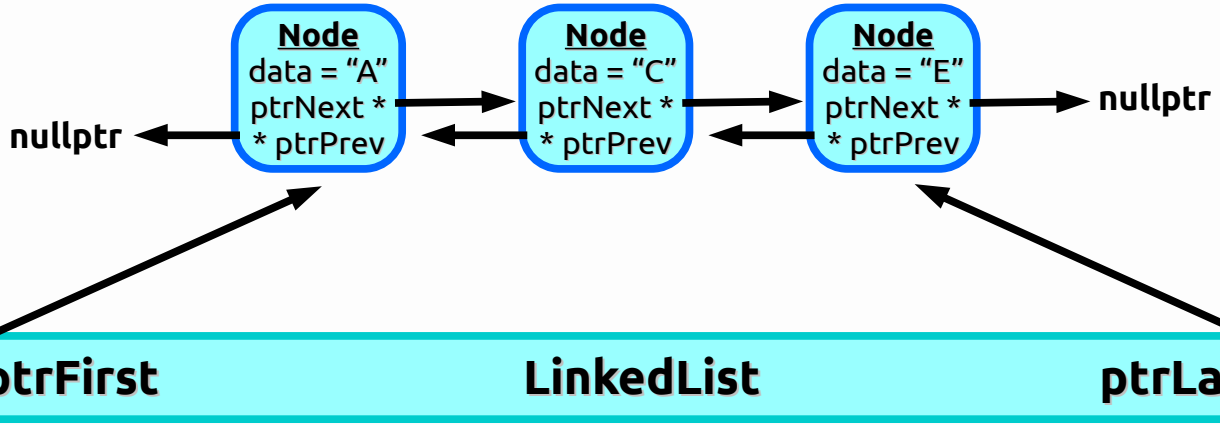


Notes

4. Linked List Functionality

GetFront()

For GetFront, we want the data stored at the front of the list.

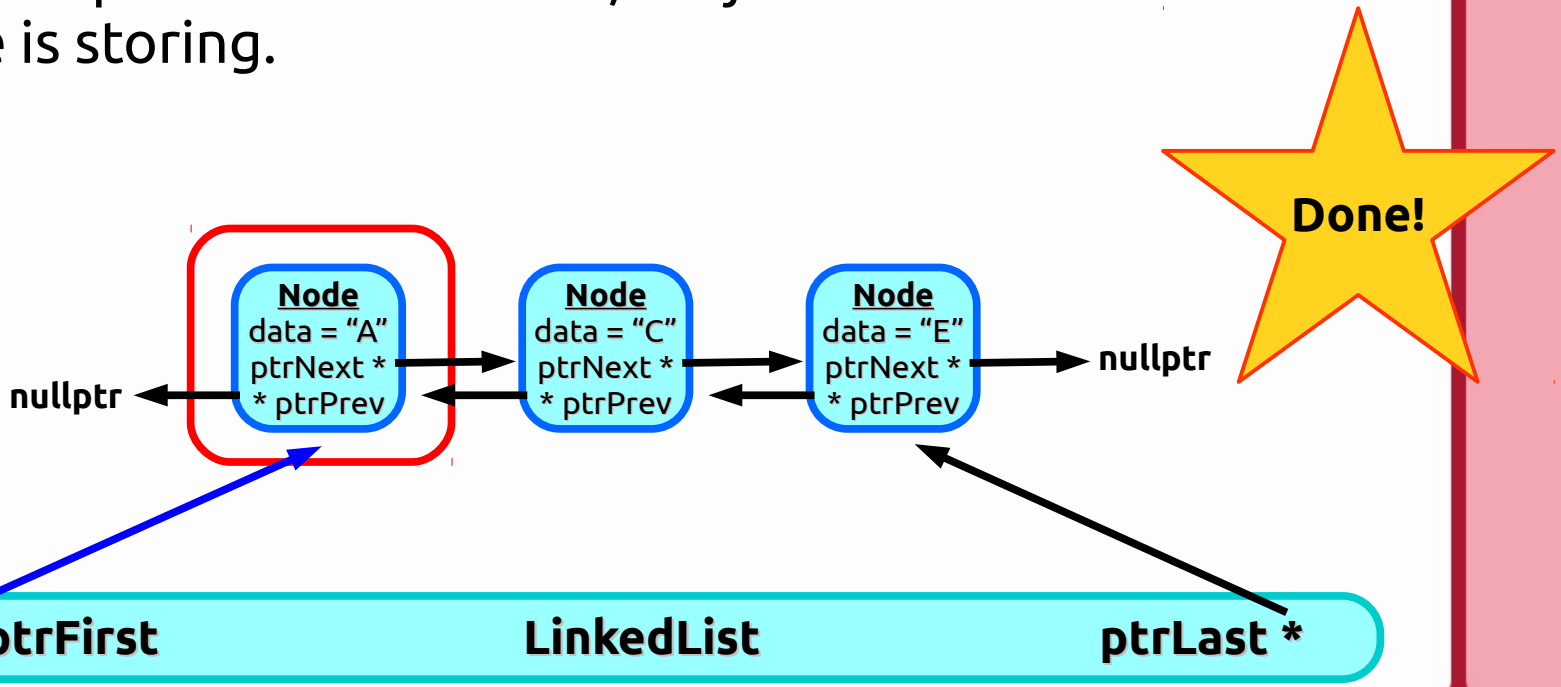


Notes

4. Linked List Functionality

GetFront()

We can access the Front-most item via the Linked List's **ptrFirst** pointer. From there, we just return the data this Node is storing.



Notes

4. Linked List Functionality

PopFront()

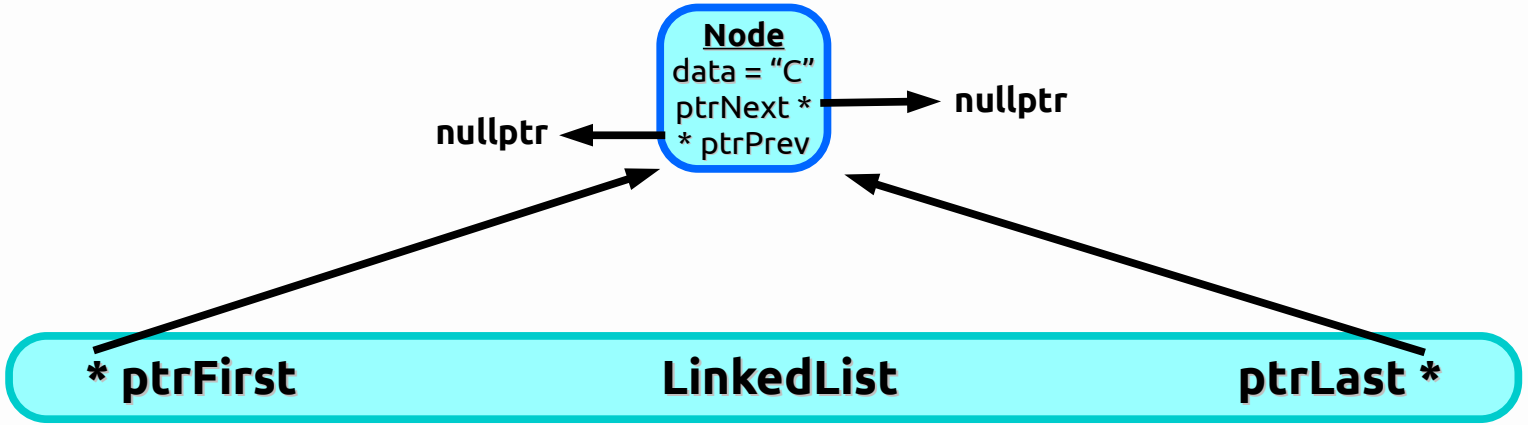
For PopFront, the two scenarios we care about is if we're removing the very last item in the list (only one item in the list) or if we're just removing an item in a list of more than 1.

Notes

4. Linked List Functionality

PopFront() **itemCount = 1**

If there's only one item in the list, then we delete it and update the LinkedList's **ptrFirst** and **ptrLast** to nullptr.



Notes

4. Linked List Functionality

PopFront() **itemCount = 1**

If there's only one item in the list, then we delete it and update the LinkedList's **ptrFirst** and **ptrLast** to nullptr.

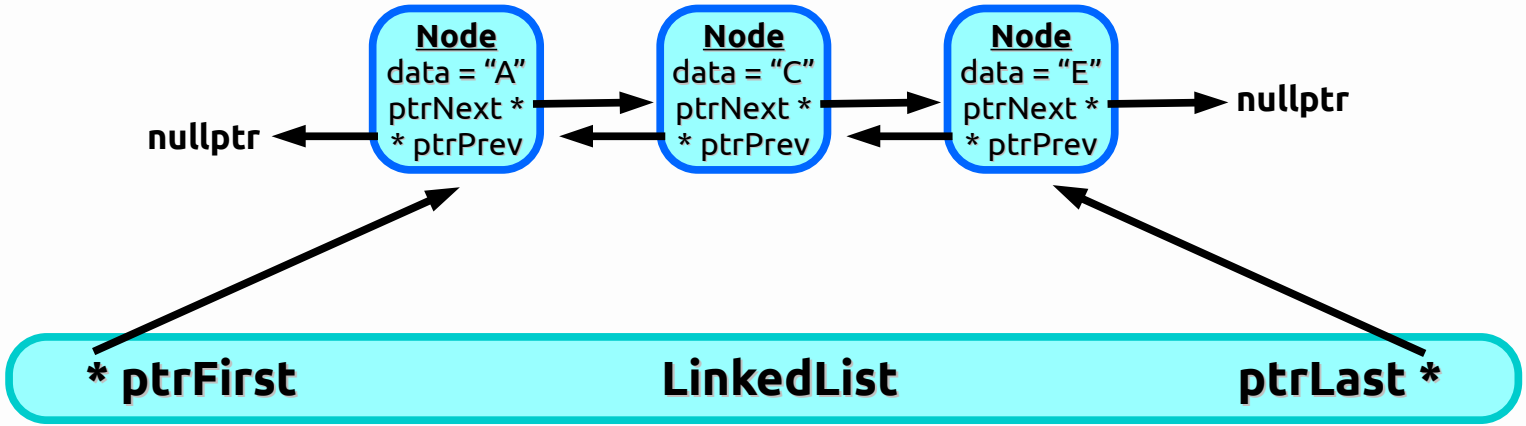
(Don't forget to decrement itemCount!)

Notes

4. Linked List Functionality

PopFront() **itemCount > 1**

For PopFront(), we need to remove the front-most item and set a new First item via the LinkedList **ptrFirst** pointer.

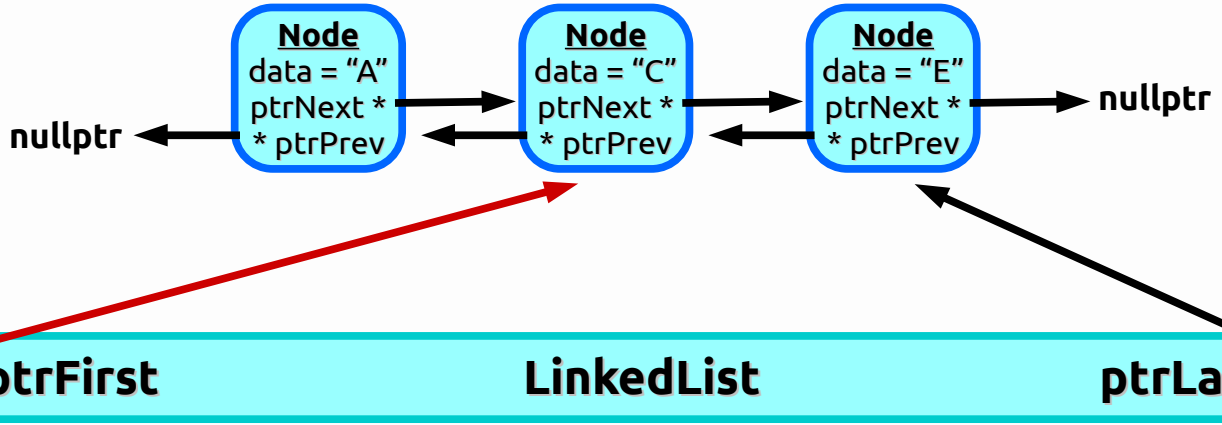


Notes

4. Linked List Functionality

PopFront() **itemCount > 1**

First update **ptrFirst** to the the current first item's **ptrNext**.

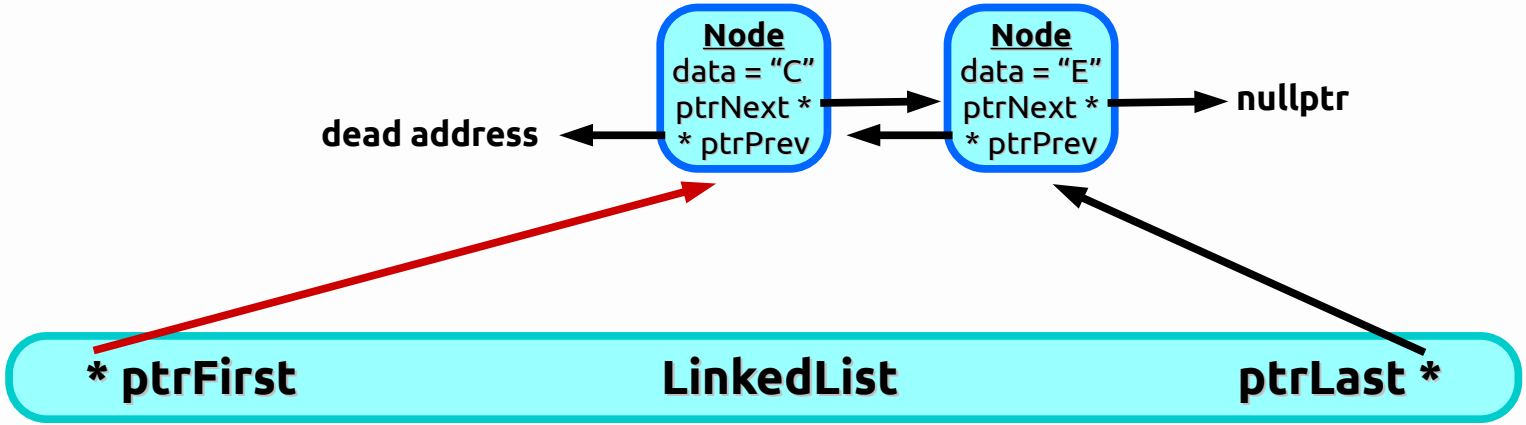


Notes

4. Linked List Functionality

PopFront() **itemCount > 1**

Then we can delete the old first item by using **delete ptrFirst->ptrPrev**.



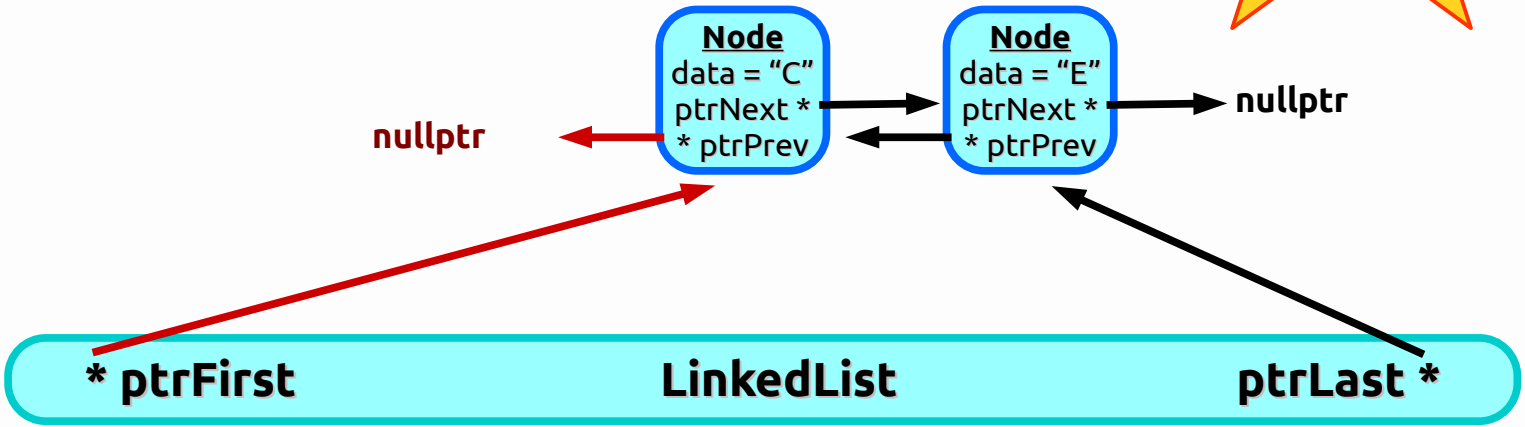
Notes

4. Linked List Functionality

PopFront() **itemCount > 1**

And you set the new **ptrFirst**'s **ptrPrev** to **nullptr**.

(Don't forget to decrement the itemCount!)

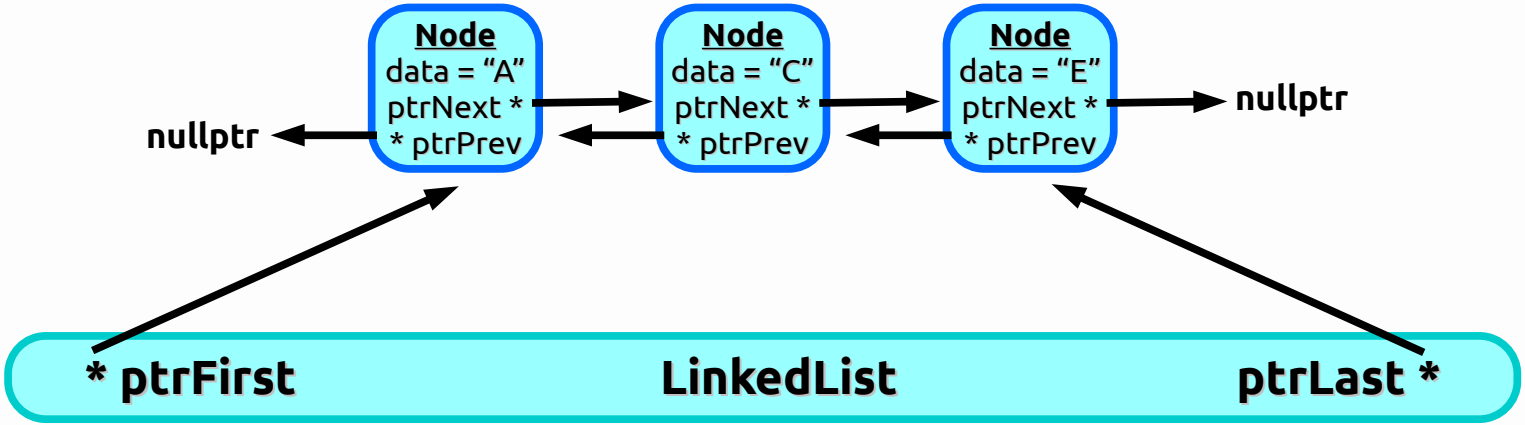


Notes

4. Linked List Functionality

Get(index)

For Get (or the subscript operator [ ]) we can access an item at any arbitrary index. However, we have to traverse through all the nodes to get there.

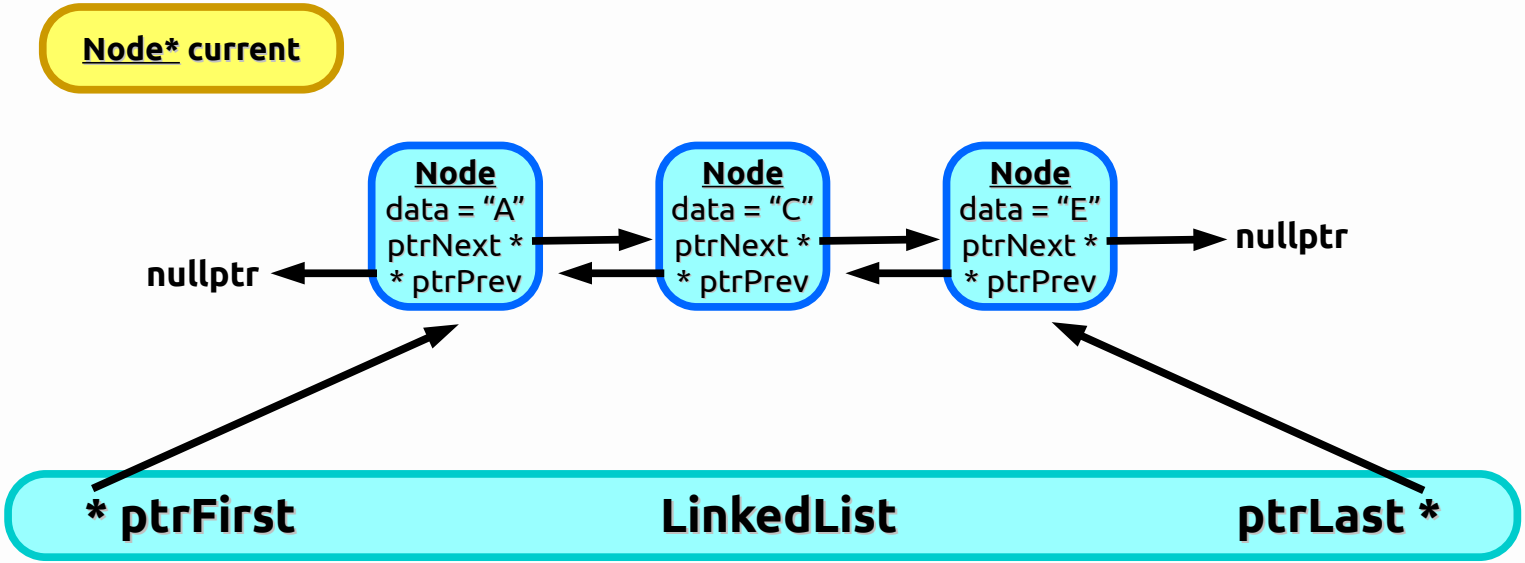


Notes

4. Linked List Functionality

Get(index)

We create a Node* pointer to start at the first Node. Note: we are NOT allocating new memory! We are just creating an node to point at existing memory!



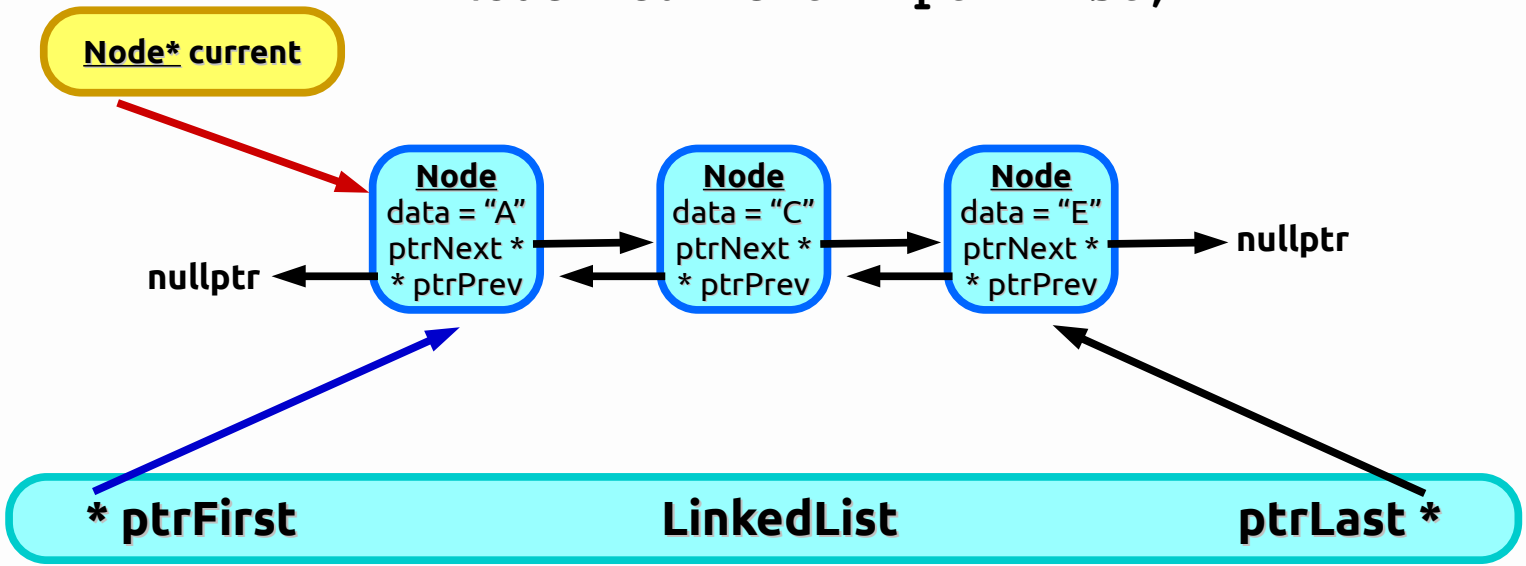
Notes

4. Linked List Functionality

Get(index)

We point the **current** pointer to the first item via the LinkedList.

```
Node* current = ptrFirst;
```

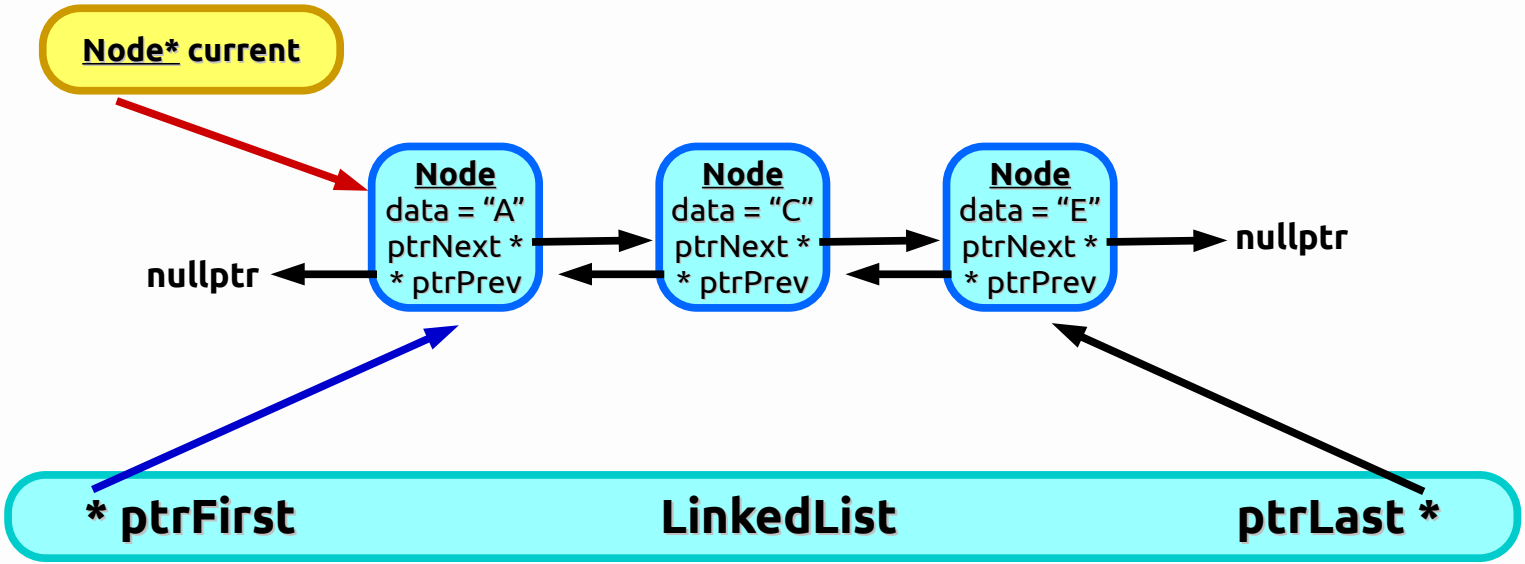


Notes

4. Linked List Functionality

Get(index)

Based on the **index** passed in, we know how many spots we want to go forward – so we can use a loop.



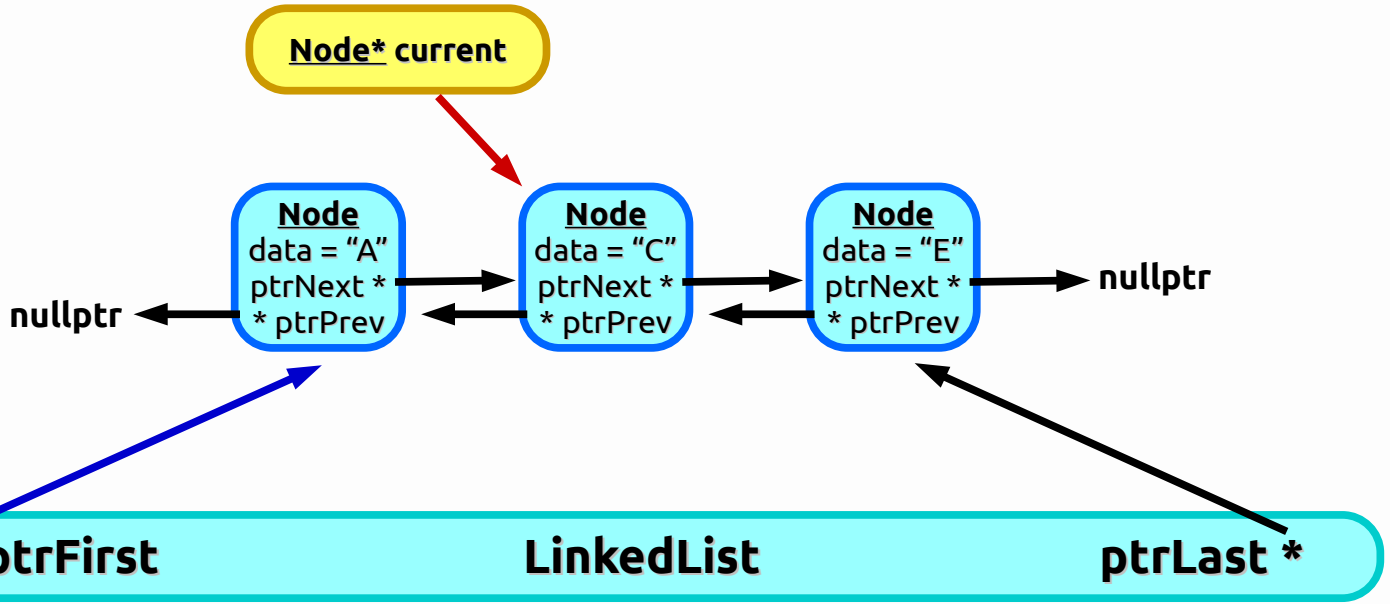
Notes

4. Linked List Functionality

Get(index)

We step the **current** pointer forward one by setting it to itself's **ptrNext**.

```
current = current->ptrNext;
```



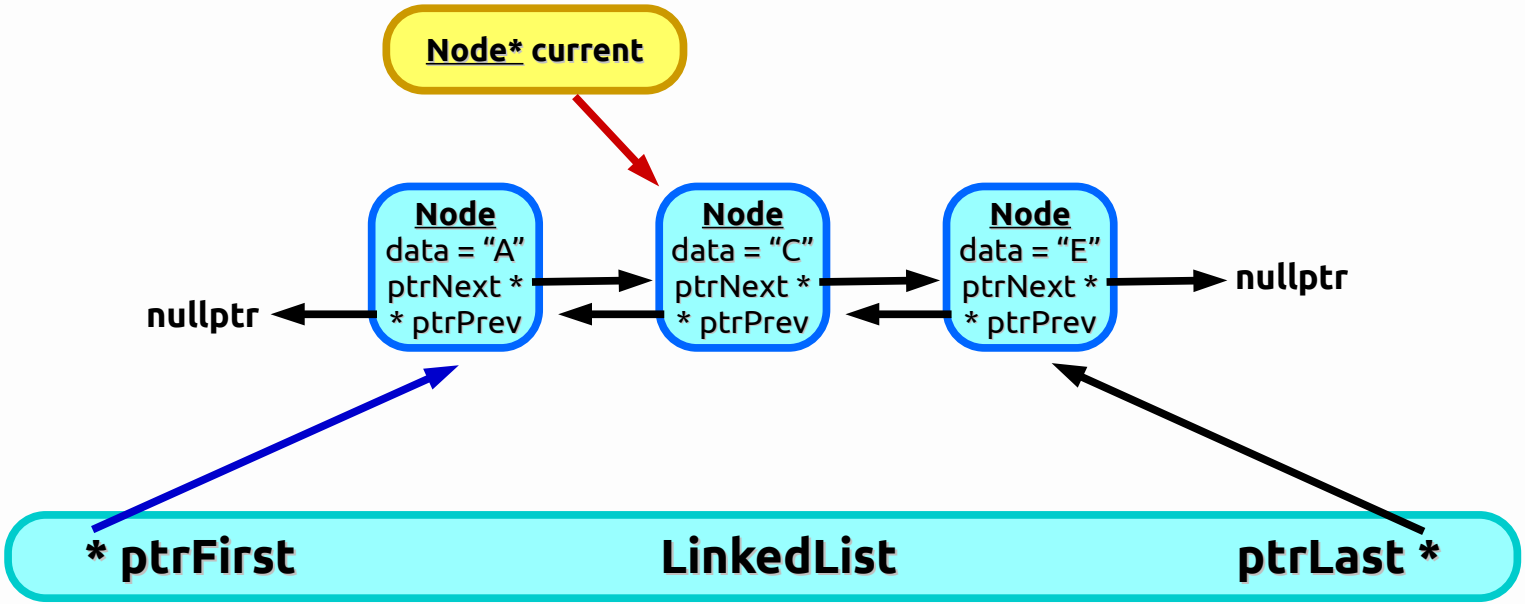
Notes

4. Linked List Functionality

Get(index)

Once we're at the index desired, we return the **data** at this node.


```
return current->data;
```

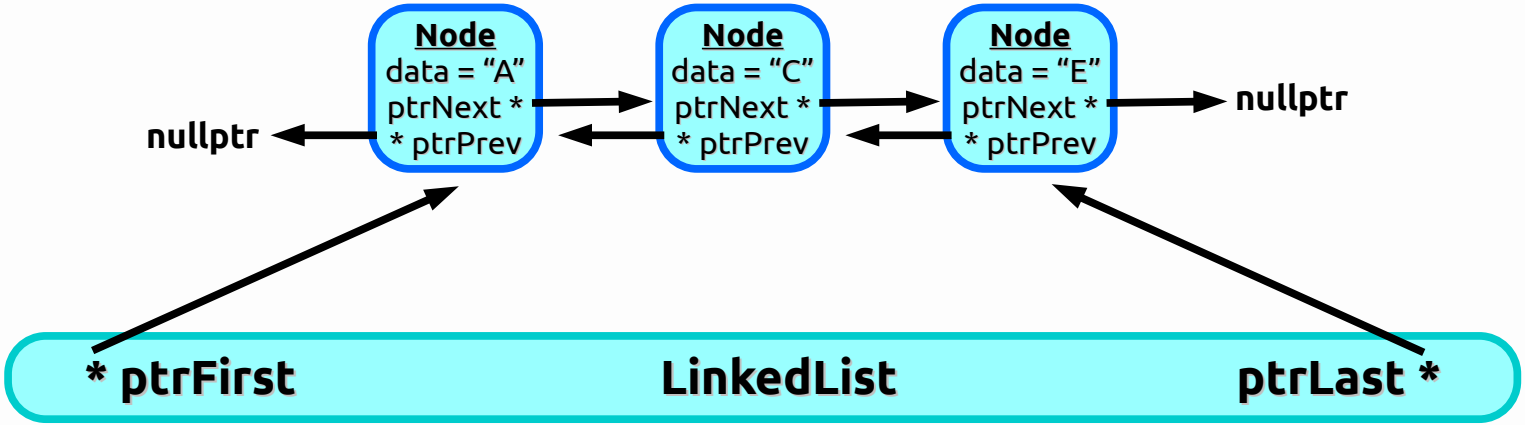


Notes

4. Linked List Functionality

Insert(position, data)

For Insert, we want to put a new item between two other items.

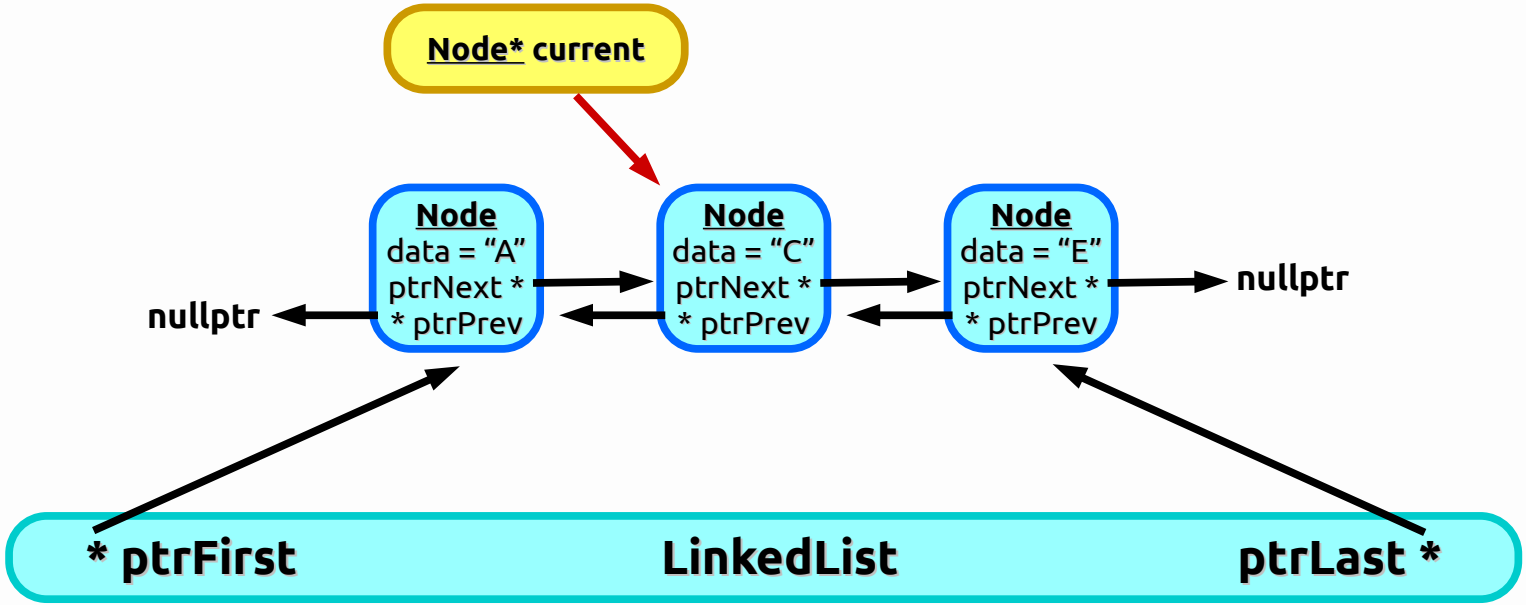


Notes

4. Linked List Functionality

Insert(position, data)

Before we can insert our item, we need to find the position we want it at. We can use the Get(index) function to get a location.

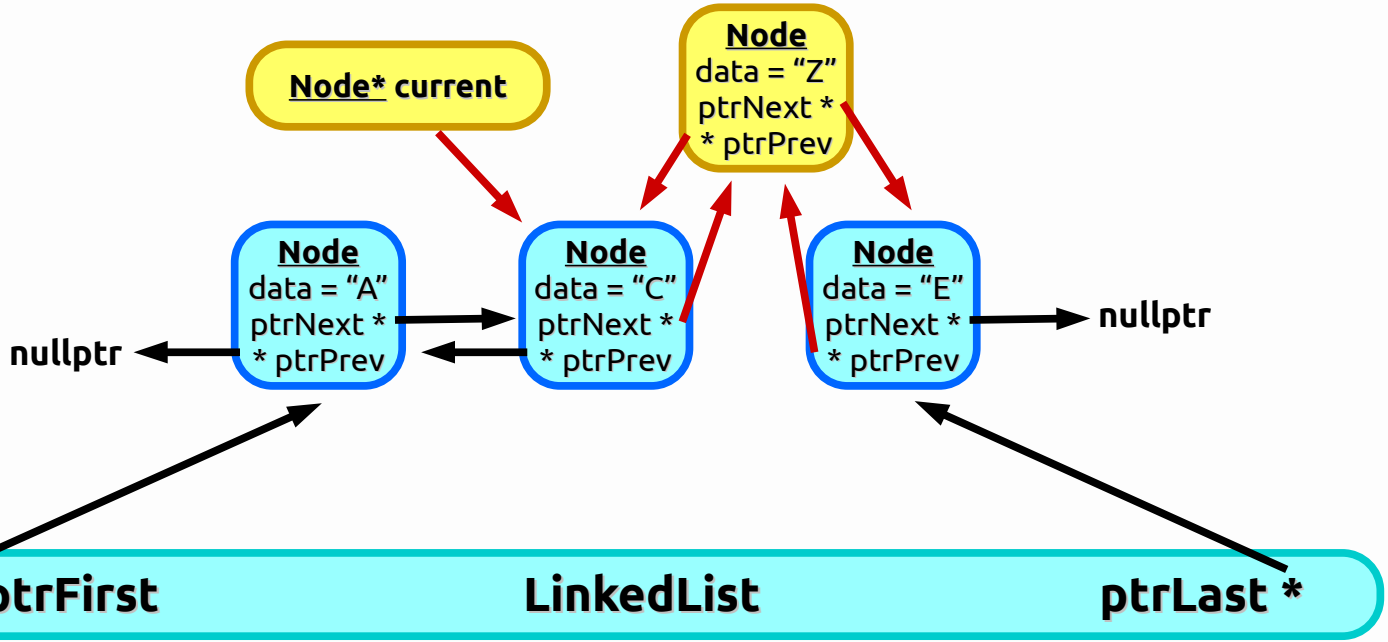


Notes

4. Linked List Functionality

Insert(position, data)

Create the new node, and then you're going to insert it between the other nodes. You can use the **current** ptr as a helper when you do this.



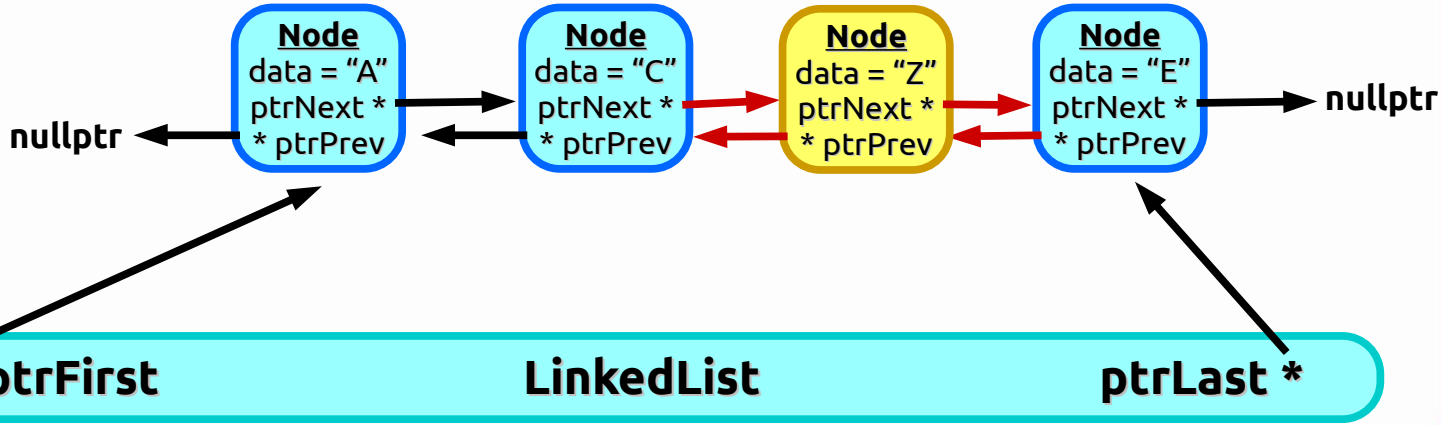
Notes

4. Linked List Functionality

Insert(position, data)

Then we have inserted our new item!

(Don't forget to increment the item count!)

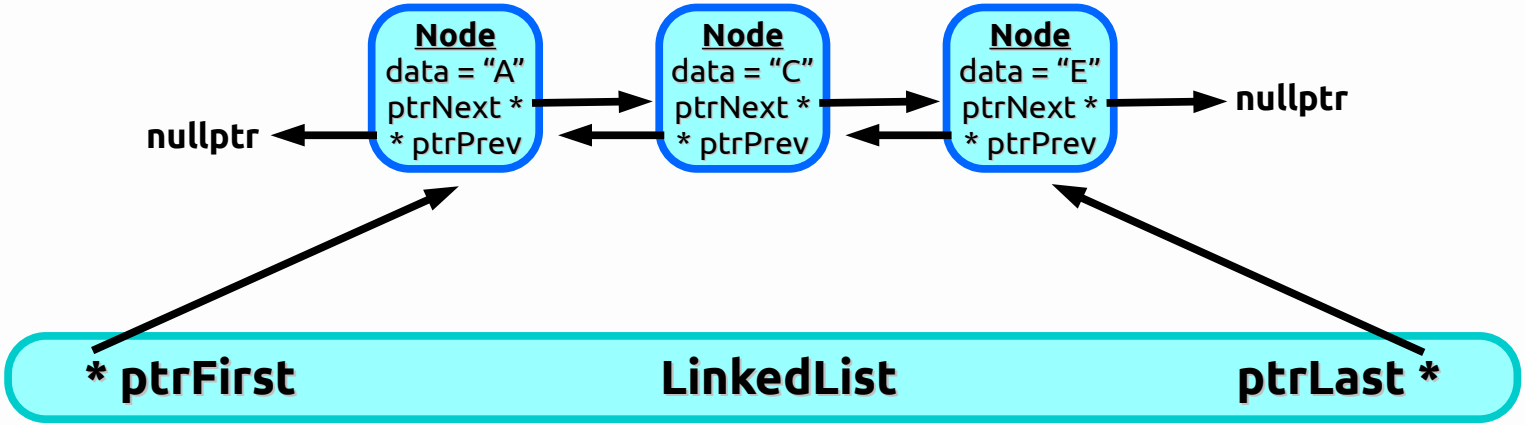


Notes

4. Linked List Functionality

Remove(index)

For Remove, we will have an index of a Node somewhere within the List to remove.

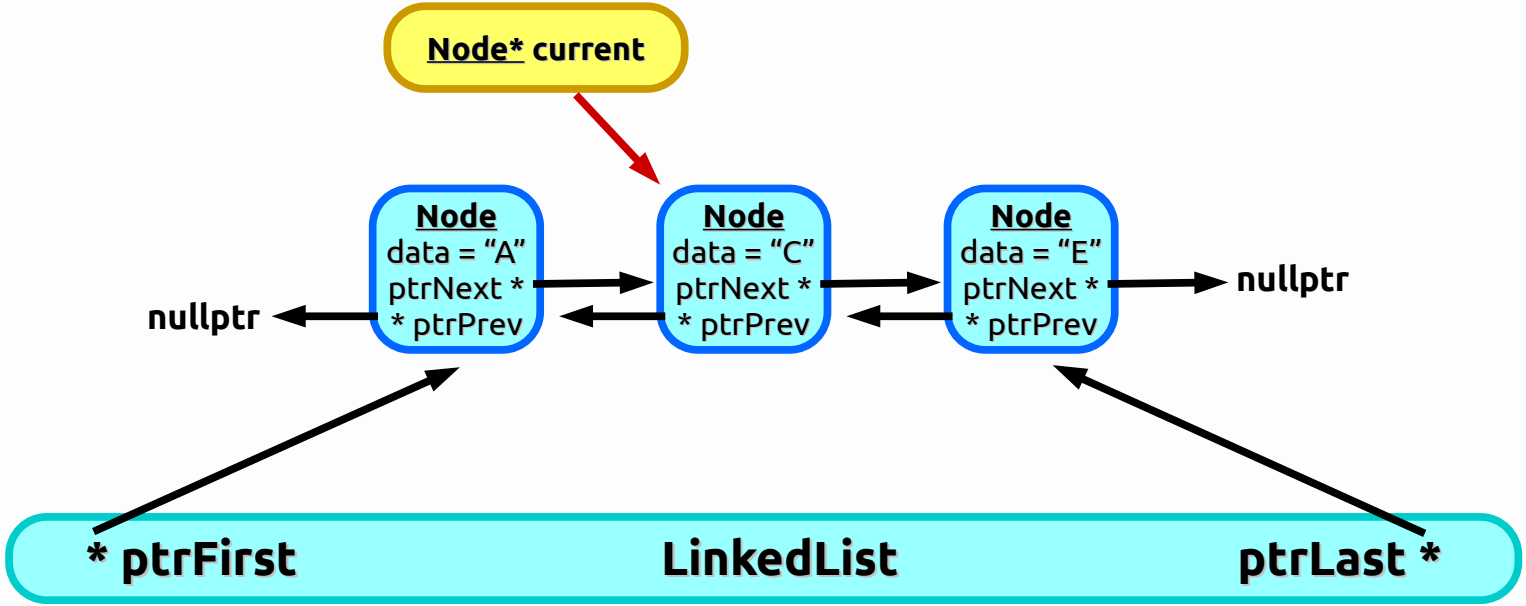


Notes

4. Linked List Functionality

Remove(index)

We can again use Get(index) to get the node we care about…

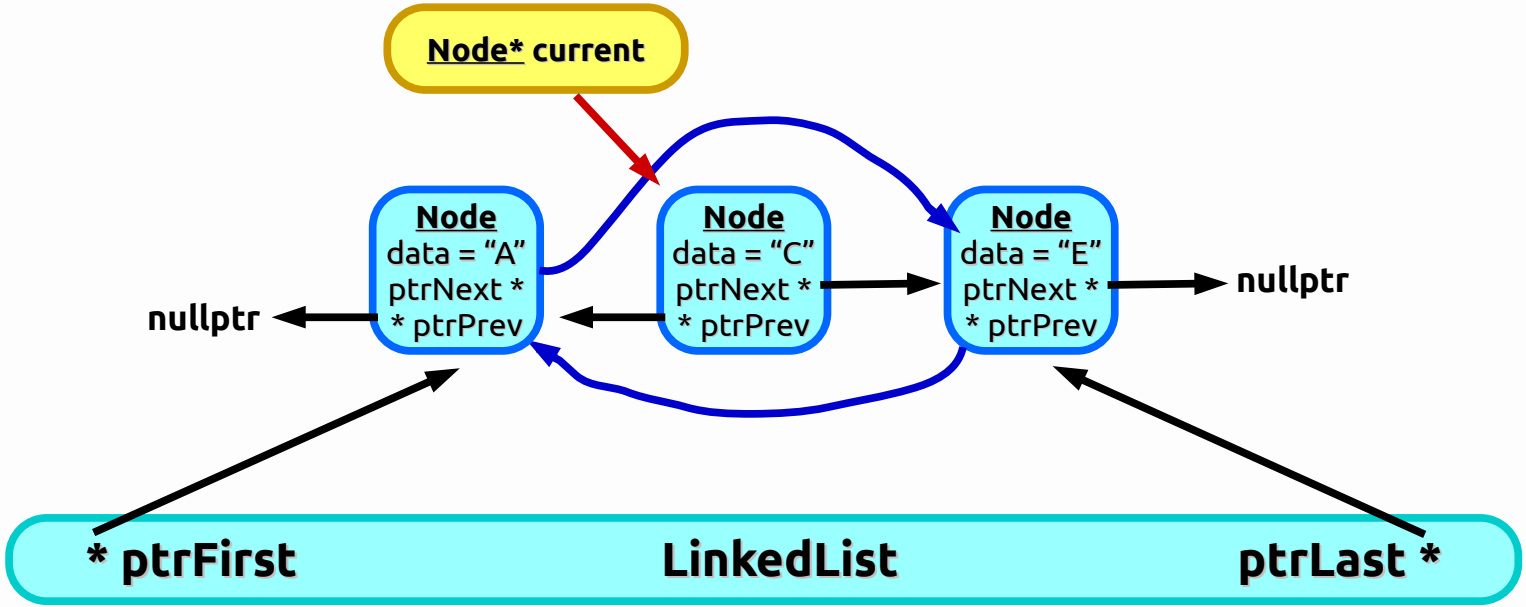


Notes

4. Linked List Functionality

Remove(index)

We update the pointer of the item **before** and **after** the Node we will delete, so we bypass the to-be-deleted Node.



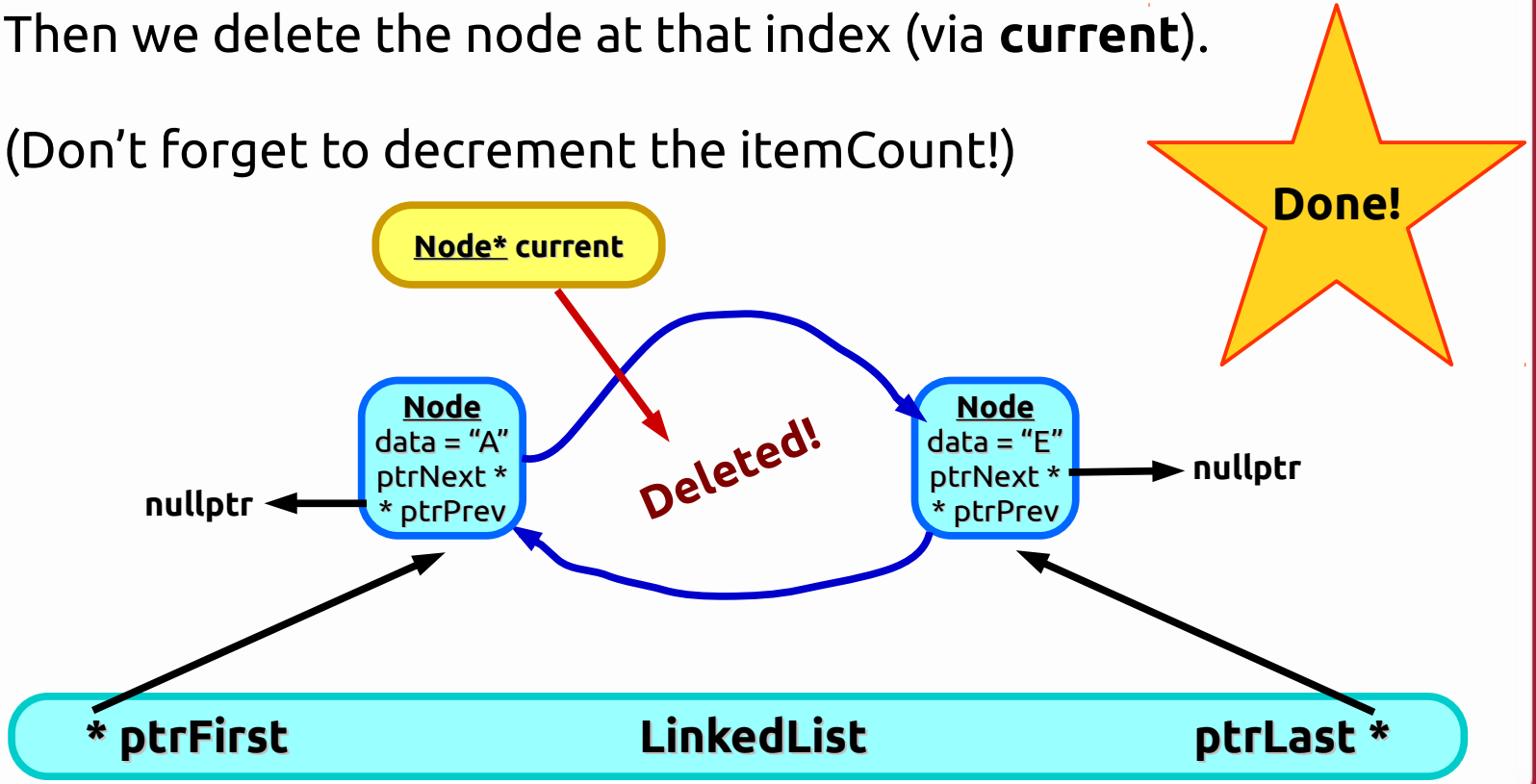
Notes

4. Linked List Functionality

Remove(index)

Then we delete the node at that index (via **current**).

(Don’t forget to decrement the itemCount!)



Notes

Conclusion

Conclusion

Linked Lists are only one type of new data structure that we'll be covering this semester. Make sure to review pointers if you need to, because other structures will also use 'em!