Linked Lists

Work in progress (Sept 10)
I need to finish editing together the video (+ animations?) for this lecture, here are the slides so far.

Written by Rachel J. Morris, last updated Sep 10, 2017

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. Stepping through Linked List functionality
   - `PushBack`
   - `PushFront`
   - `PopBack`
   - `PopFront`
   - `GetFront`
   - `GetBack`

1. 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 (dynamically allocating memory for a *single variable*, 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.

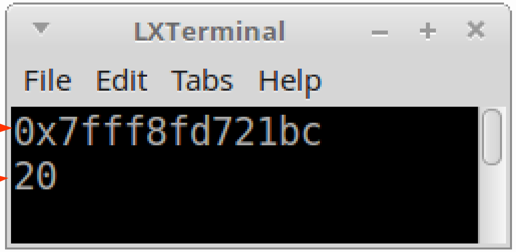
```
int main()
{
    int someNumber = 20;

    int* ptrNum = &someNumber;

    cout << ptrNum << endl;
    cout << *ptrNum << endl;
```

Display address ptrNum is pointing to.

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

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

With a linked list, our element **Nodes** will have pointers to other existing nodes, such as “ptrPrevious” and “ptrNext”.

```
struct Node
{
    Node* ptrNext;
    Node* ptrPrev;
    // ...
};
```

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

We will have an over-arching "LinkedList" class that will also contain pointers to the **first Node** (and possibly the last Node)...

```
class LinkedList
{
    public:
    // ... functions ...

    private:
    Node* ptrFirst;
    Node* ptrLast;
};
```

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

When we need a new item in our List, then we use the **new** keyword to allocate memory for a new item.

```
void LinkedList::Push( const DATA& newData )
{
    Node* ptrNew = new Node;
    ptrNew->data = newData;

    // ... place in list ...
}
```

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

And when we're **traversing** the list, we will use another pointer to walk through all the nodes, one at a time. As it moves forward, the pointer is updated to point to the next Node in the list.

```
void LinkedList::DisplayAll()
{
    Node* ptrCurrent = ptrFirst;

    while ( ptrCurrent != nullptr )
    {
        cout << ptrCurrent->data << endl;
        ptrCurrent = ptrCurrent->ptrNext;
    }
}
```

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 use an assignment statement between two pointers, the pointer on the LHS (left-hand-side) will now point to the same place as the pointer on the RHS (right-hand-side).

```
void LinkedList::DisplayAll()
{
    Node* ptrCurrent = ptrFirst;

    while ( ptrCurrent != nullptr )
    {
        cout << ptrCurrent->data << endl;
        ptrCurrent = ptrCurrent->ptrNext;
    }
}
```

```
int* ptr1 = &a;
int* ptr2;
ptr2 = ptr1;
// now they both
// point at a.
```

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

Also, if a pointer is poing to an object, and you want to access the original items' **member functions and variables**, you need to use the arrow operator ->.

```
void LinkedList::DisplayAll()
{
    Node* ptrCurrent = ptrFirst;

    while ( ptrCurrent != nullptr )
    {
        cout << ptrCurrent->data << endl;
        ptrCurrent = ptrCurrent->ptrNext;
    }
}
```

```
struct Node
{
    Node* ptrNext;
    Node* ptrPrev;
    // ...
};
```

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];`

2. 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

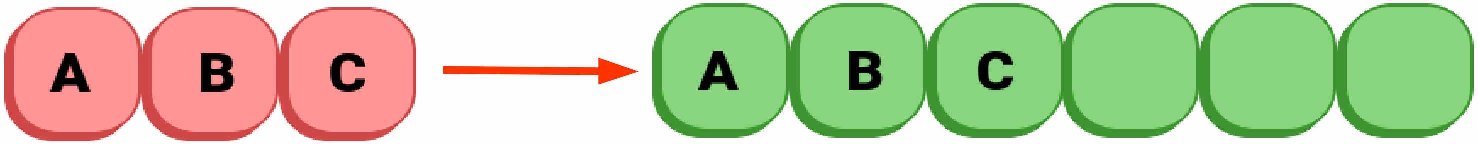
4) Update the pointer to point to the new (large) array.

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

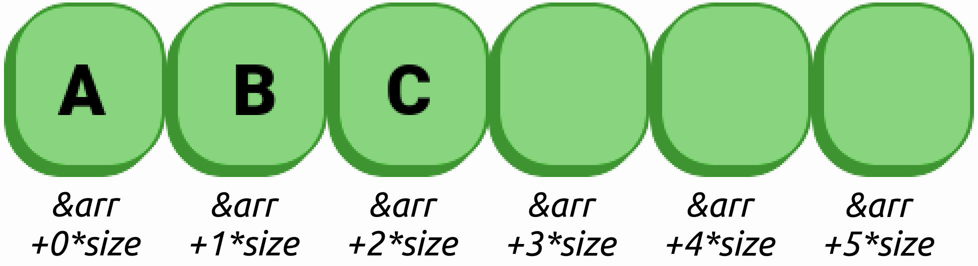
Dynamic Array

- 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

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

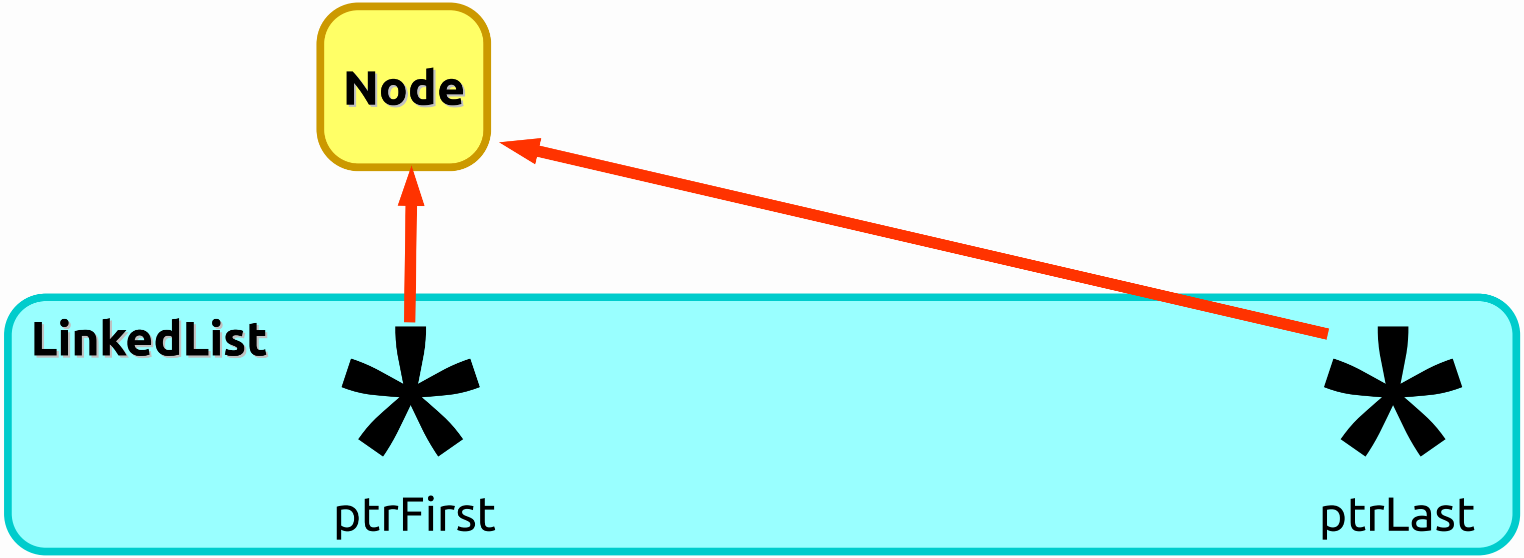
- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

Linked List

- Wraps element data in "Nodes"

2. Dynamic Array vs. Linked List

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



Notes

Dynamic Array

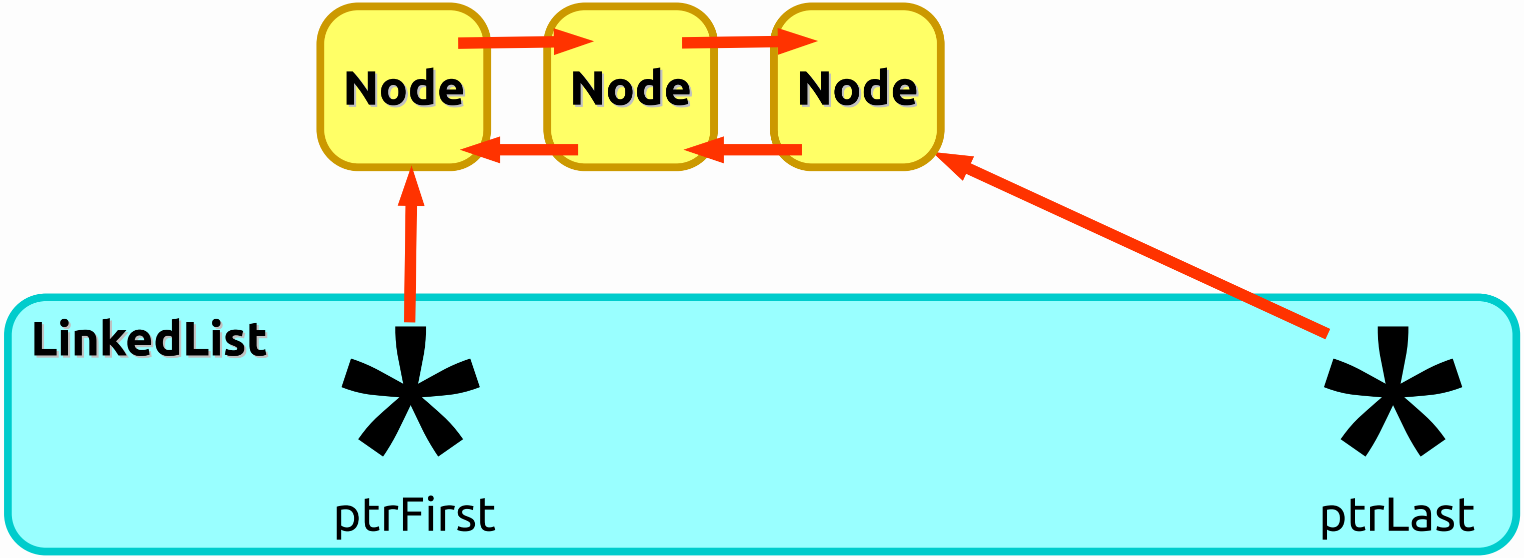
- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

Linked List

- Wraps element data in "Nodes"

2. Dynamic Array vs. Linked List

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



Notes

Dynamic Array

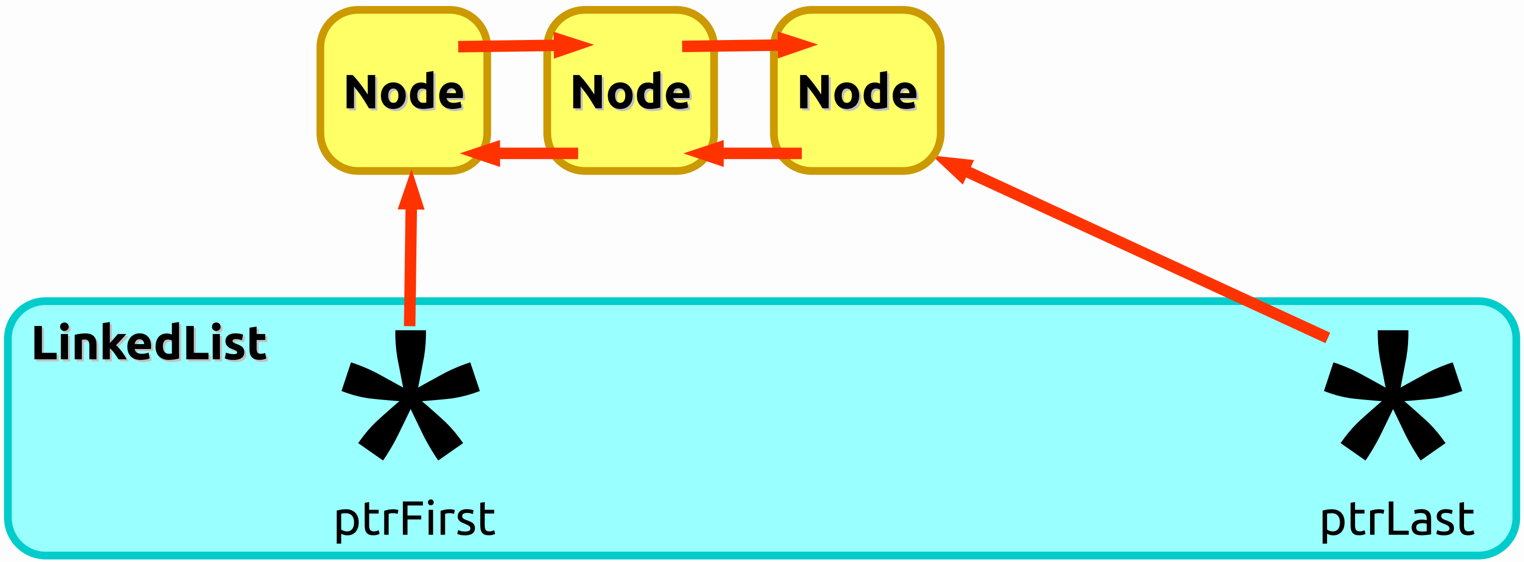
- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

Linked List

- Wraps element data in “Nodes”

2. Dynamic Array vs. Linked List

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

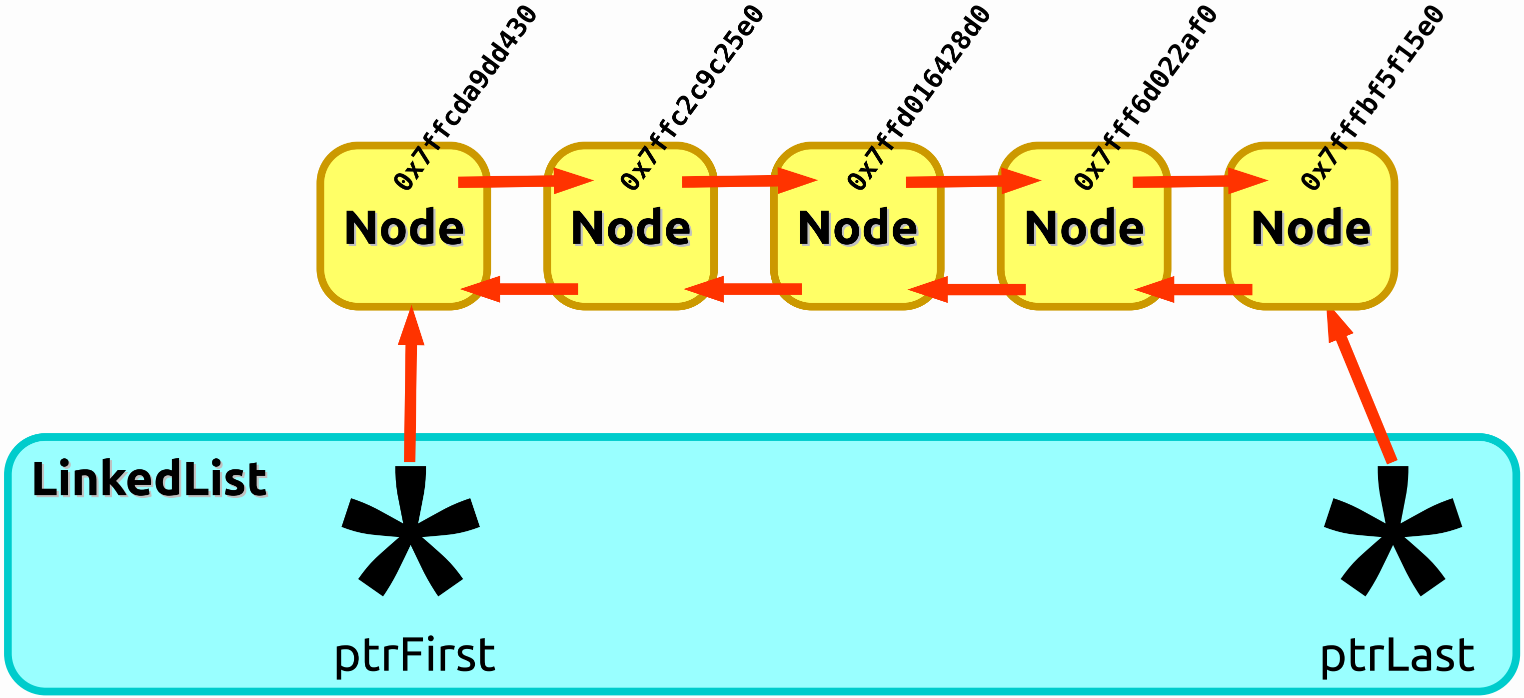
- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

Linked List

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

2. Dynamic Array vs. Linked List

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.*

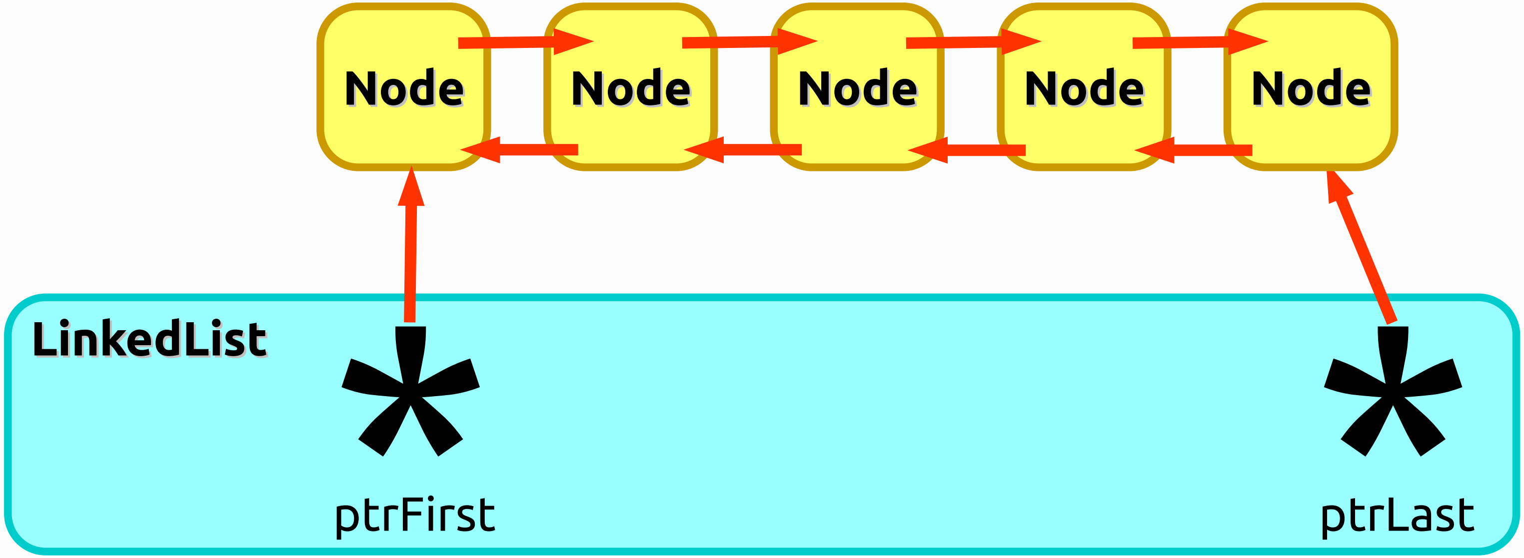
Linked List

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

2. Dynamic Array vs. Linked List

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

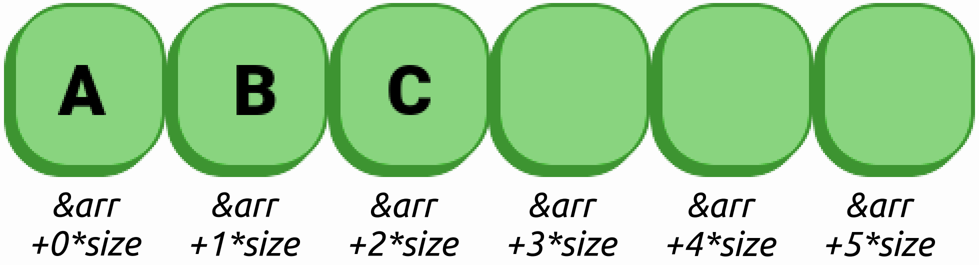
Dynamic Array

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

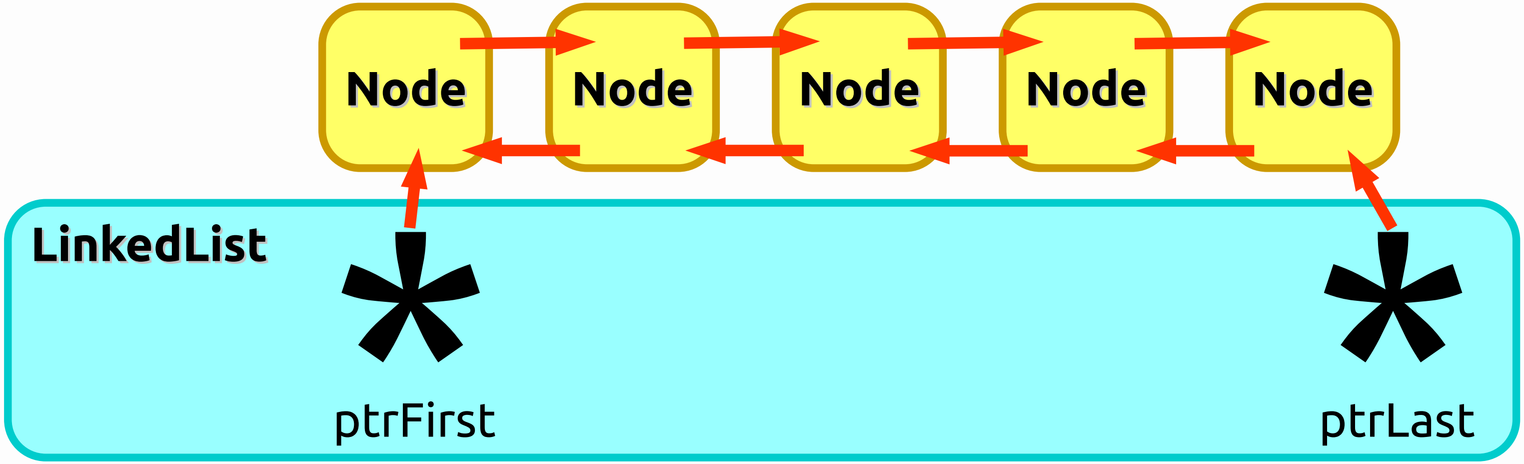
Linked List

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

2. Dynamic Array vs. Linked List



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



Notes

Dynamic Array

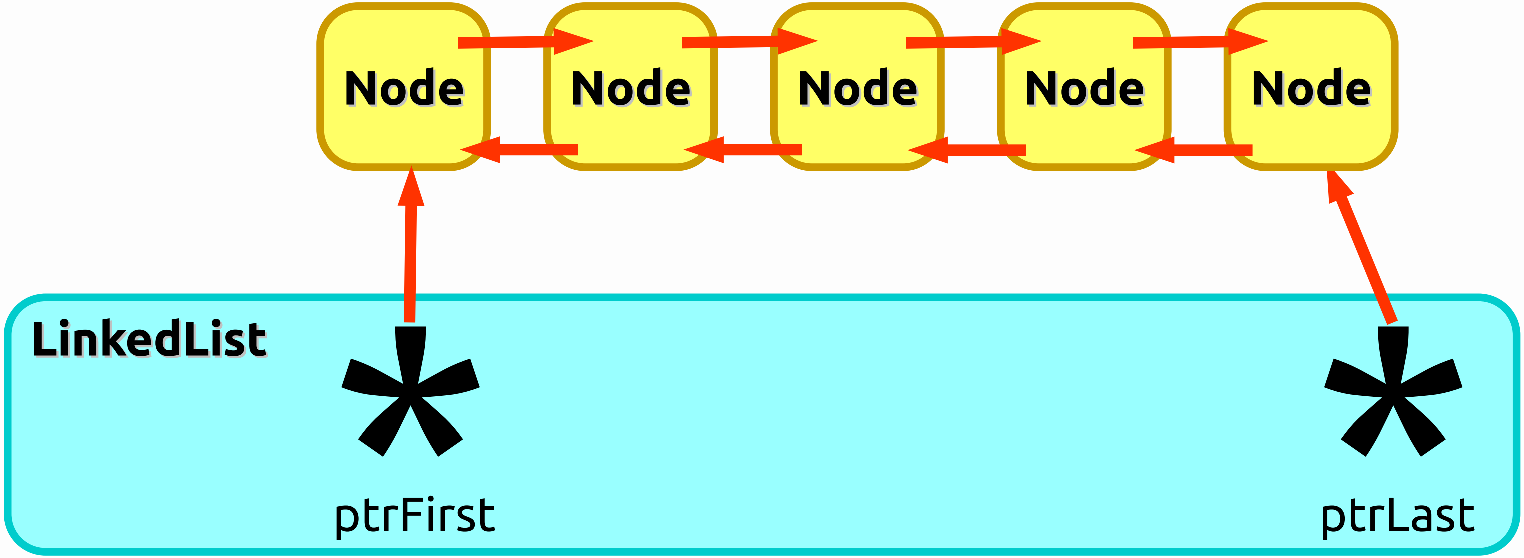
- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

Linked List

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

2. Dynamic Array vs. Linked List

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

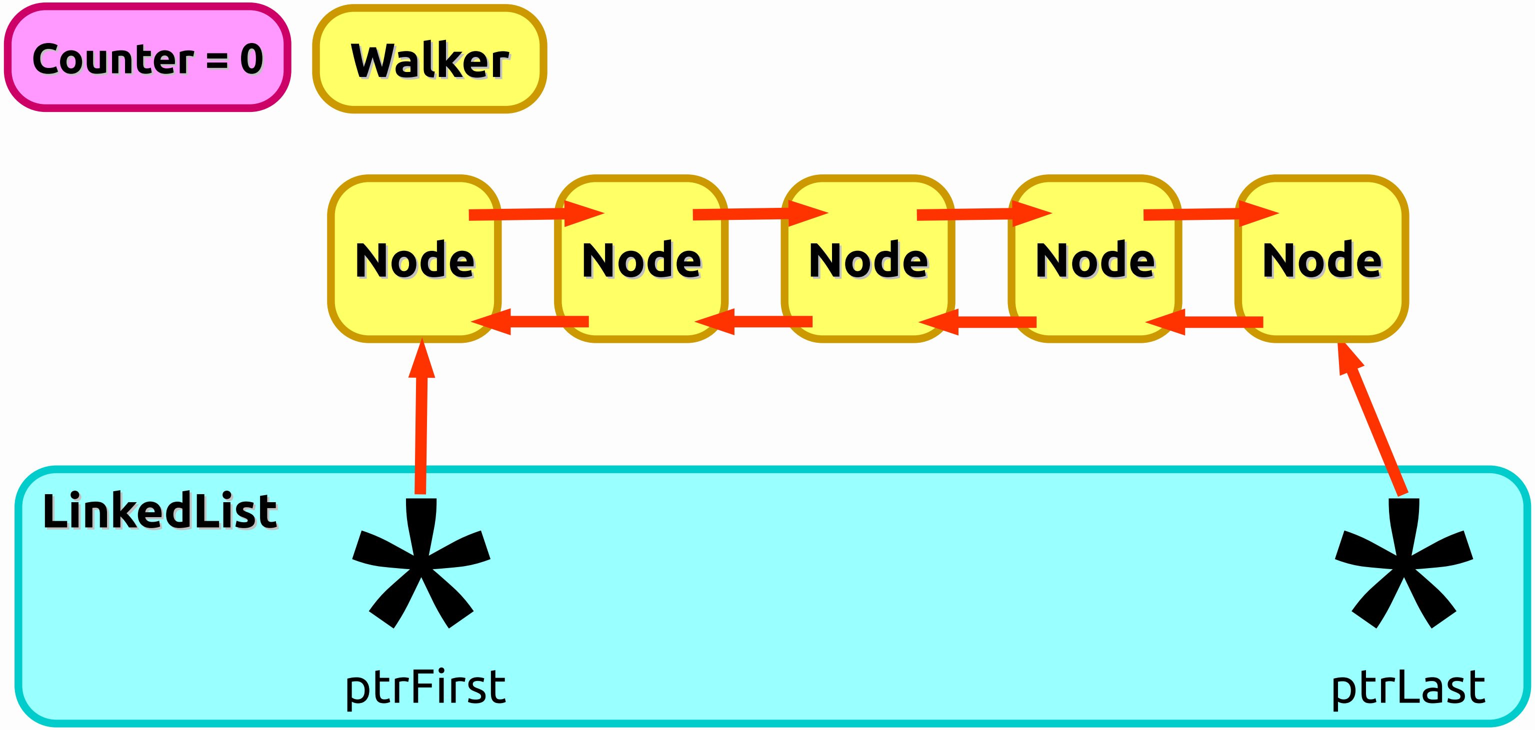
- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

Linked List

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

2. Dynamic Array vs. Linked List

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

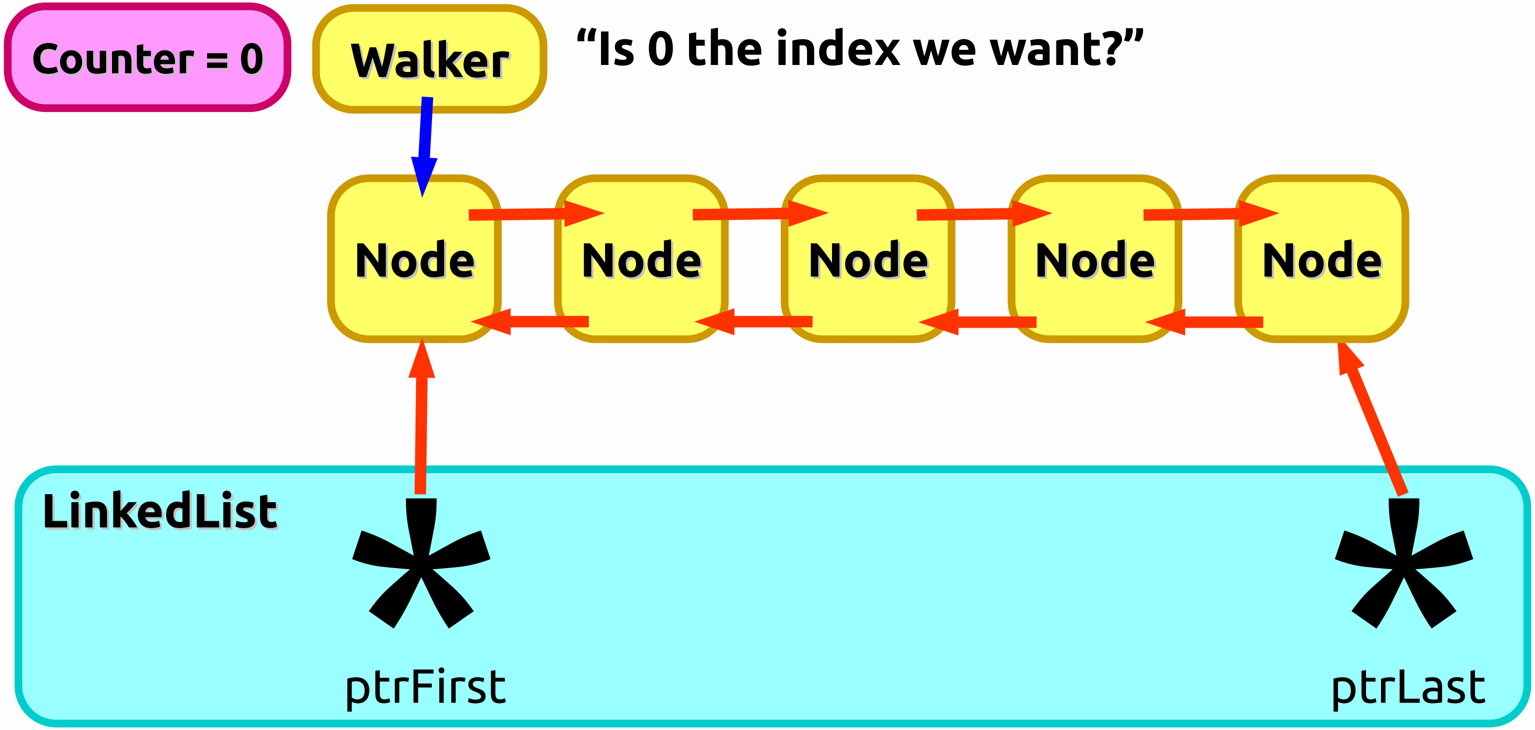
- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

Linked List

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

2. Dynamic Array vs. Linked List

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



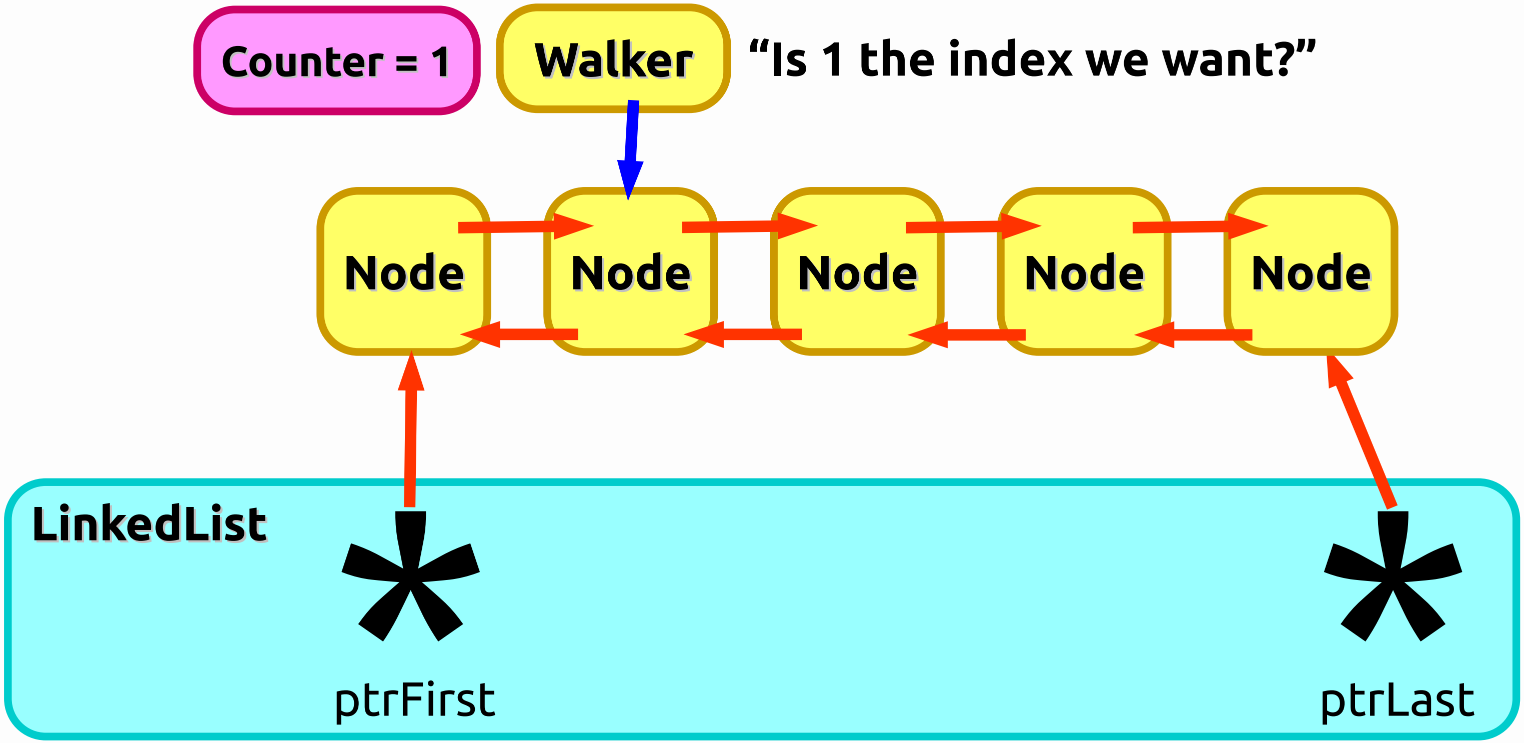
Notes

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

2. Dynamic Array vs. Linked List

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



Notes

Dynamic Array

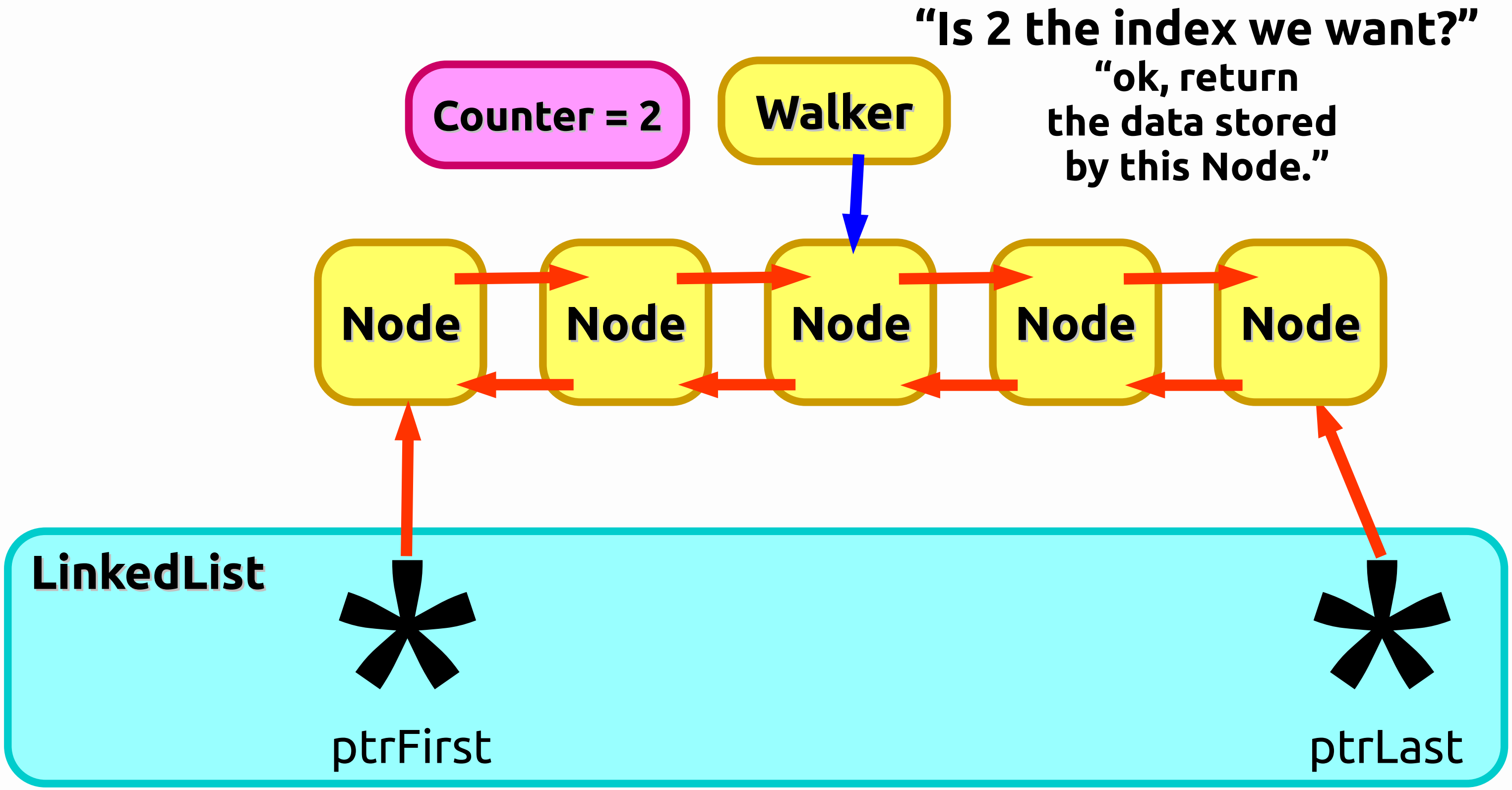
- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

Linked List

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

2. Dynamic Array vs. Linked List

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



Notes

Dynamic Array

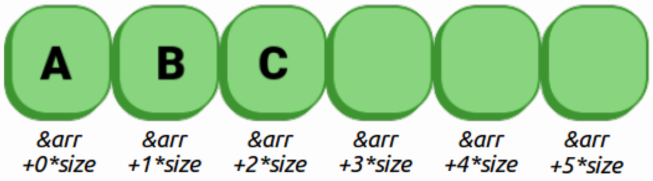
- Resize is costly
- Access is cheap
- Whether to use is a *design decision.*

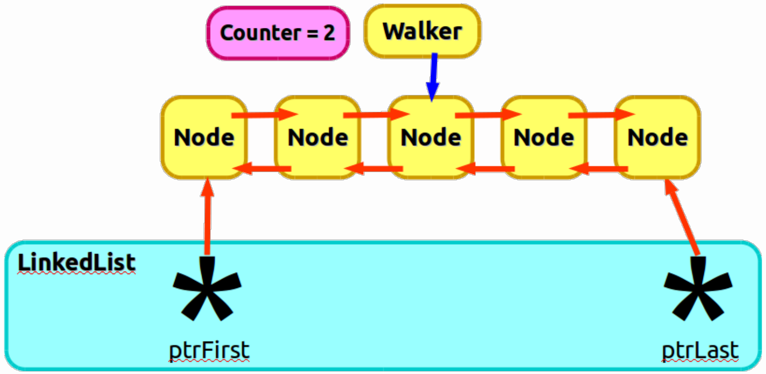
Linked List

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

2. Dynamic Array vs. Linked List

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.*

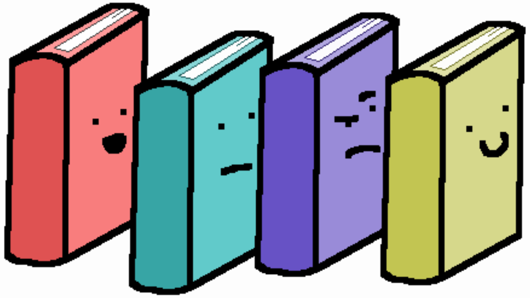
Linked List

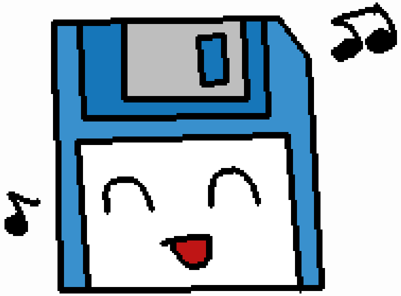
- Wraps element data in "Nodes"
- Pointers are used to build a "chain"
- Elements are non-contiguous
- Traversing the list is slower

2. Dynamic Array vs. Linked List

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

2. Dynamic Array vs. Linked List

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.

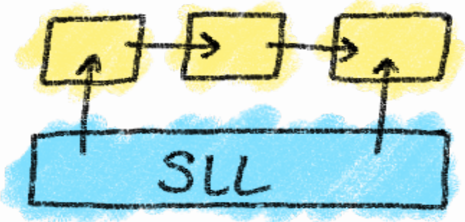
There are also **Circularly Linked Lists**, where the last Node of the list points to the first Node as its *next item*.

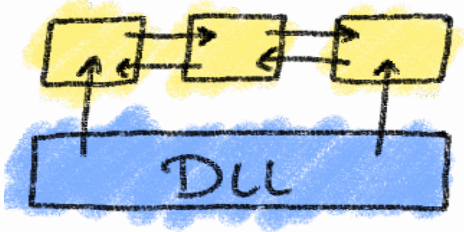
Notes

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

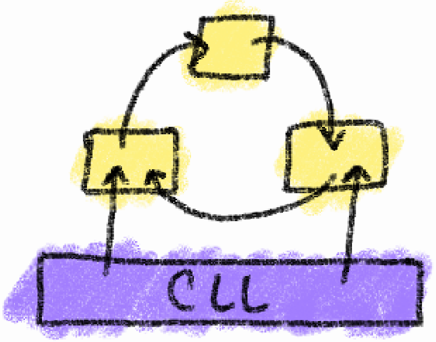
Linked List
- Wraps element data in "Nodes"
- Pointers are used to build a "chain"
- Elements are non-contiguous
- Traversing the list is slower

2. Dynamic Array vs. Linked List





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

3. Stepping through Linked List functionality

2. Functionality - Classes

The LinkedList

Notes

Conclusion

Conclusion

We did a thing.

Next time we do more things.