

Lecture 6

Linked Lists and Iterators



EECS 281: Data Structures & Algorithms

Linked List



- Each node is represented by one person
- Each node points to the next node
- Last node's next pointer points to nullptr (nobody)

Doubly-linked List



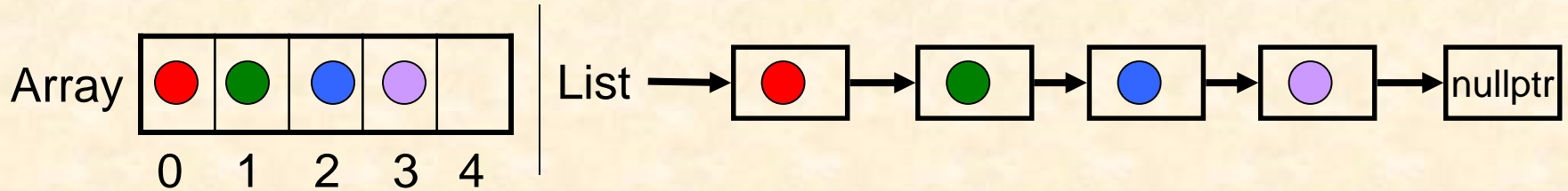
- Each node is represented by one person
- Each node points to next and previous nodes
- Last node's next pointer points to nullptr


Circularly-linked List



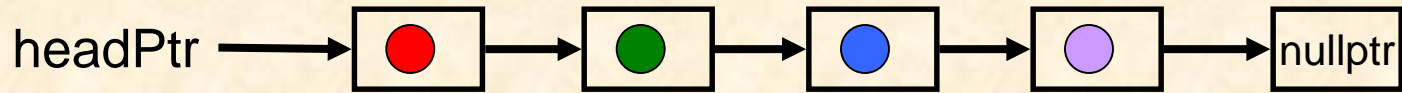
- First and last nodes point to each other
- No node designated as beginning or end
- Can be singly-linked or doubly-linked

Arrays versus Linked Lists



	Arrays		Linked Lists	
Access	Random in $O(1)$ time Sequential in $O(1)$ time	Worst Case	Random in $O(n)$ time Sequential in $O(1)$ time	Worst Case
Insert Append	Inserts in $O(n)$ time Appends in $O(n)$ time		Inserts in $O(n)$ time  Appends in $O(n)$ time	
Bookkeeping	ptr to beginning currentSize (or ptr to end of space used) maxSize (or ptr to end of allocated space – needed if dynamic sizing occurs)		head pointer to first node size (optional) tail ptr to last node (optional) In each node, ptr to next node Wasteful for small data items	
Memory	Wastes memory if size is too large Requires reallocation if too small		Allocates memory as needed Requires memory for pointers	

Linked Lists



```
1 class LinkedList {
2     struct Node {
3         double item;
4         Node* next;
5         Node() {next = nullptr;}
6     }; // Node
7
8     Node *headPtr;
9
10 public:
11     LinkedList();
12     ~LinkedList();
13     // insert methods here
14 }; // LinkedList
```

Methods

```
1 int size() const;
2
3 bool addItem(double item);
4 bool appendNode(Node *n);
5
6 bool deleteItem(double item);
7 bool deleteNode(Node *n);
```

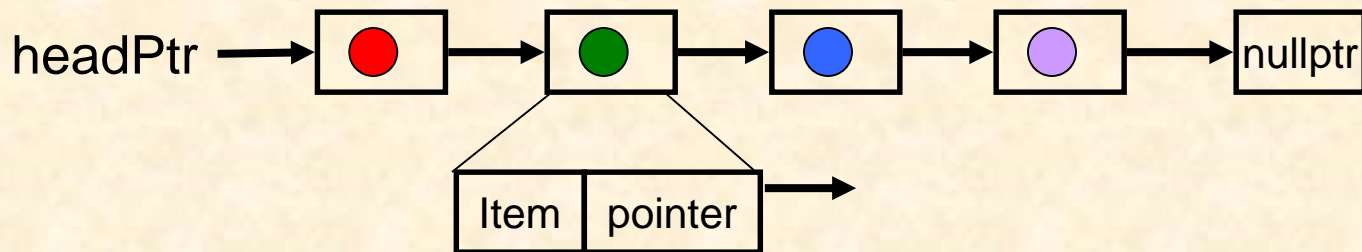
Why no `const` in `appendNode`?

What if we wanted to store objects?

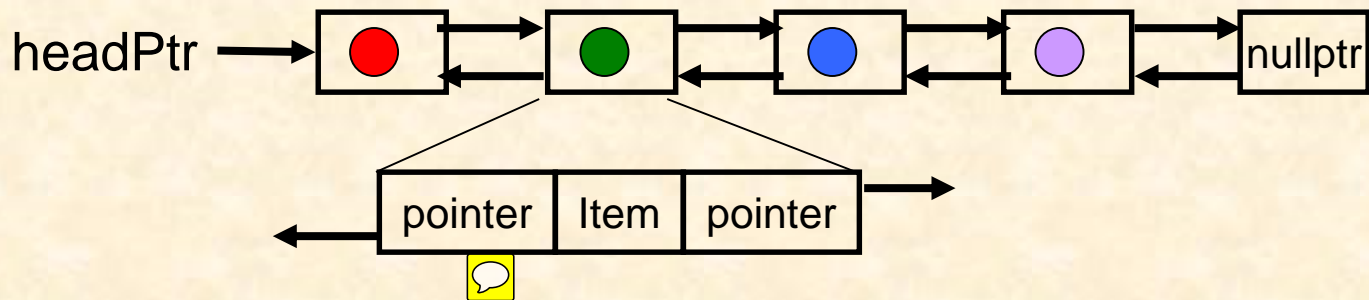
What other methods would be useful?

Singly-linked and Doubly-linked Lists

Singly-linked




Doubly-linked



Determine fraction of memory used for pointers in each version:

- If item is a **double**
- If item is an **int**
- If using a 32-bit system
- If using a 64-bit system

Complexity of Deleting a Node?

- For singly-linked list: $O(n)$ 
- For doubly-linked list: $O(1)$

Why is deleting an element from a doubly-linked easier?

In a singly-linked list, $O(n)$ suffices, but can we do better?

Idea 1: $O(1)$ time

- Pass a pointer to the “previous” node
 - Must use a dummy prev node when deleting first node
 - `deleteNode(node *prev)`
-

Idea 2: $O(1)$ time

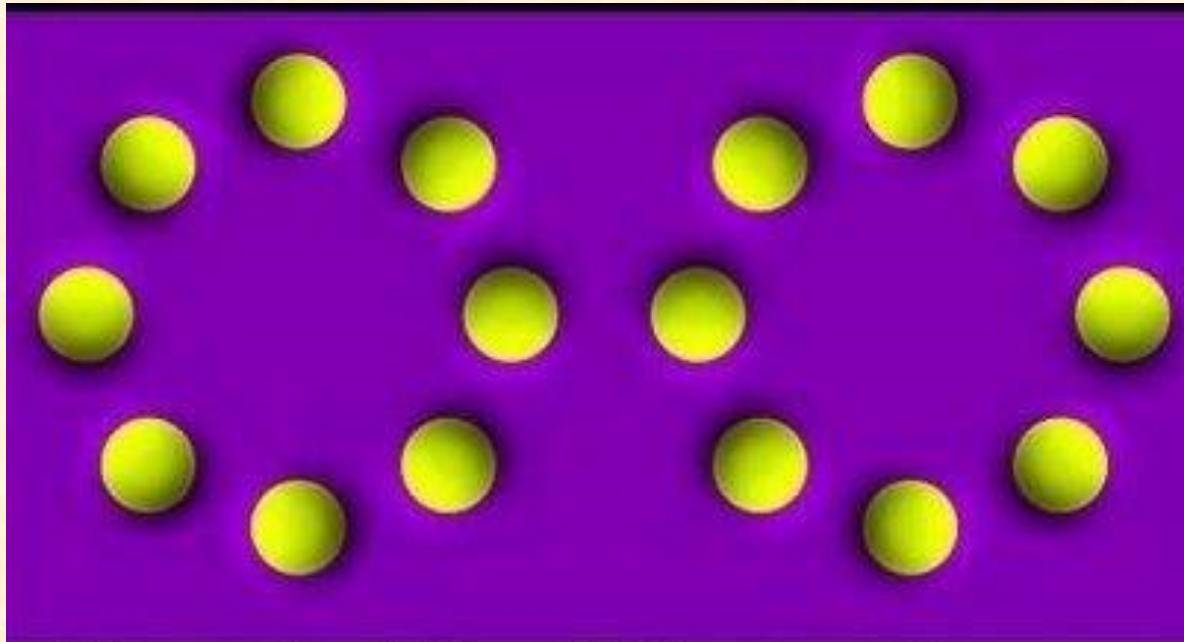
- Overwrite data in node to delete w/ next node data
- Delete next node
- Assumes data can be copied
 - Some data such as references can not be copied

Maintaining Consistency

- Arrays:
 - Stored size matches number of elements at all times
 - Be sure that `startPtr + size < endPtr`
 - `start_ptr`: pointer to start of array
 - `end_ptr`: pointer to one slot past last element
 - `size`: stored size of the array
- Linked Lists
 - Stored size matches number of elements
 - The last node points to `nullptr`
 - If the list is a circular list, last node next points to head
 - In a doubly-linked list, next/prev ptrs are also consistent
(`p == p->next->prev`) and (`p == p->prev->next`)

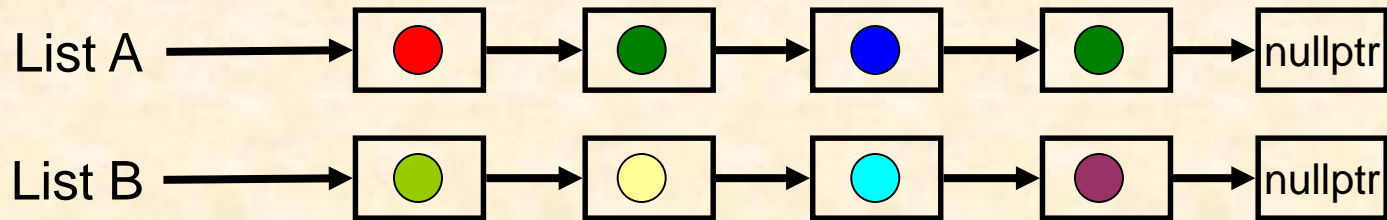


Interview Questions: dealing with Loopy Lists

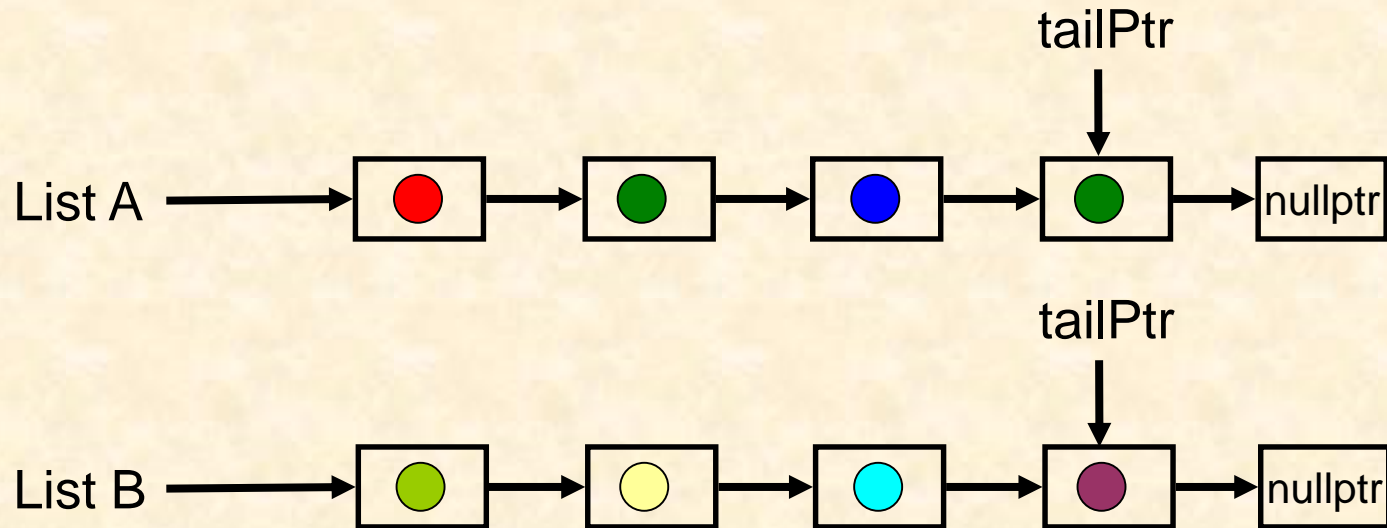


Given the head pointer, how do you determine whether a linked list is or is not circular?

Merging Lists



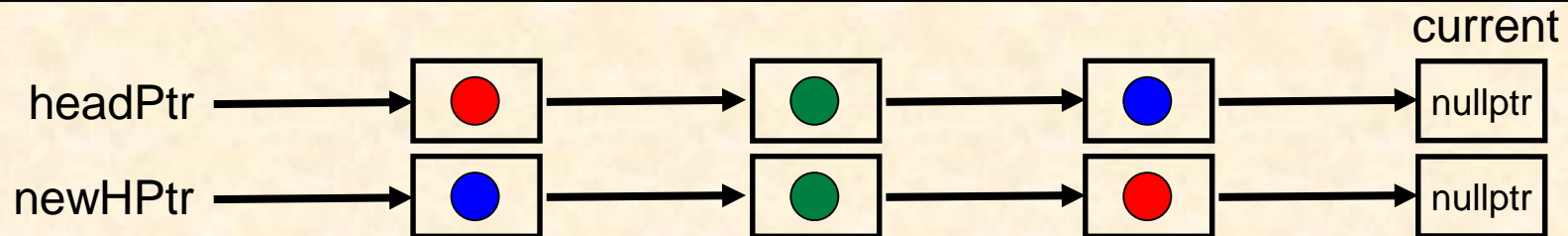
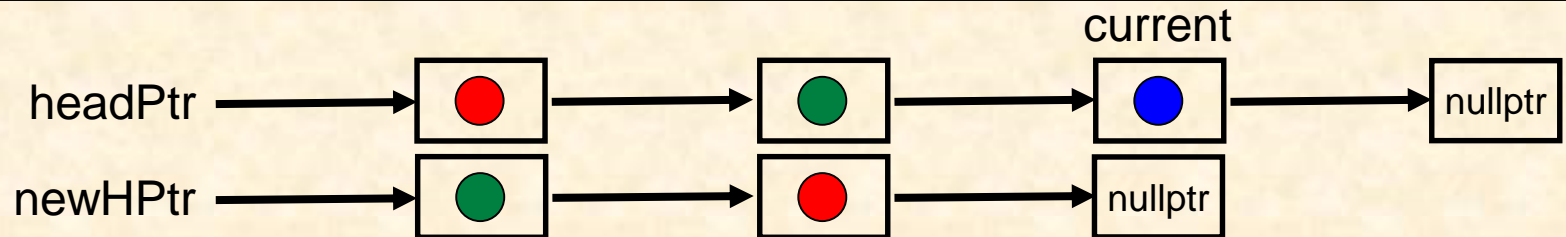
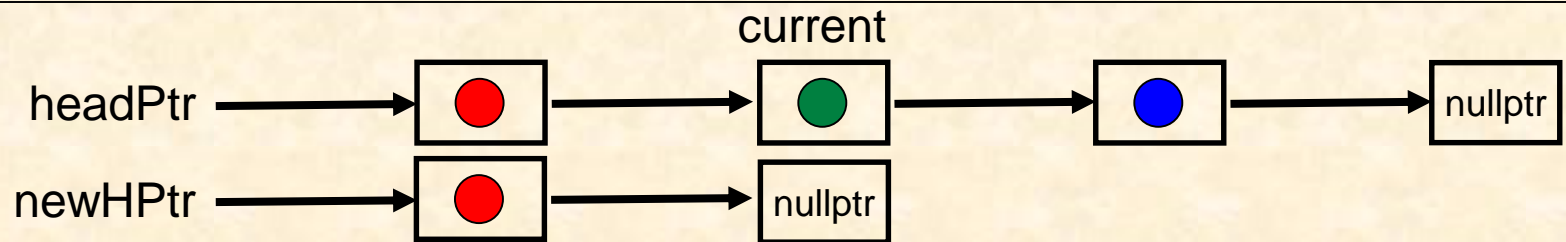
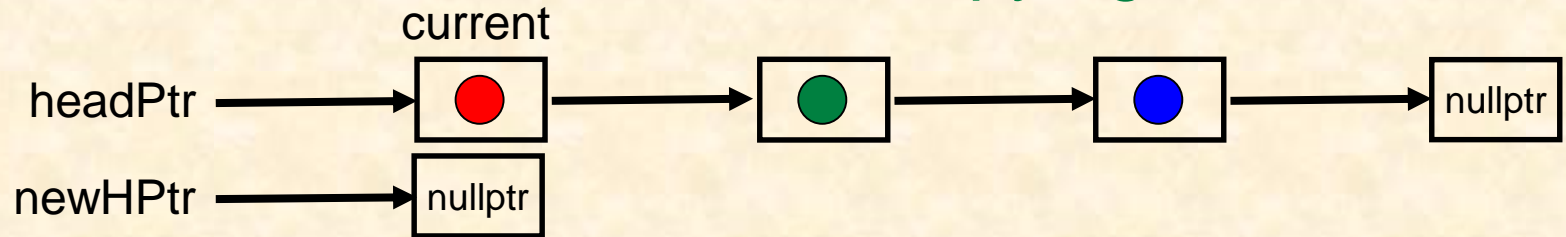
How long will it take to merge List A and List B into one list?



What if both lists have **tail pointers**?

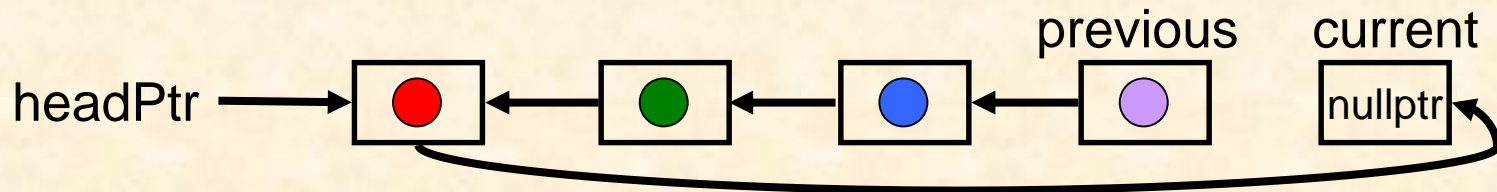
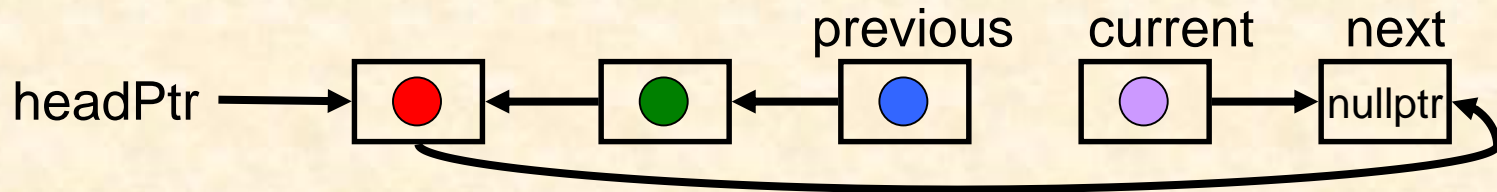
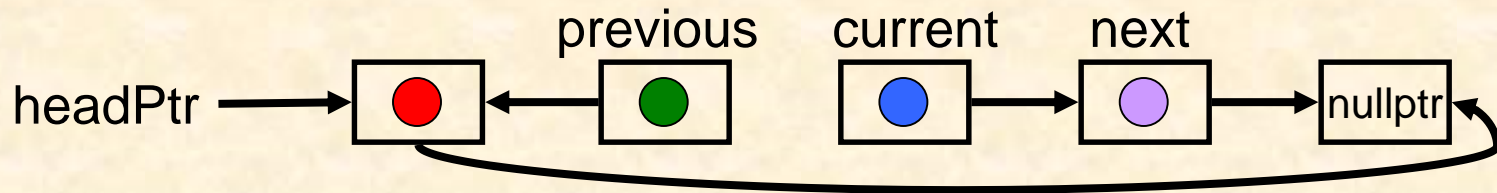
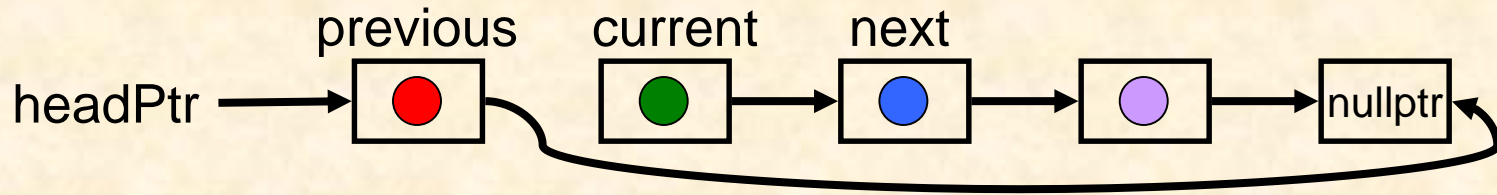
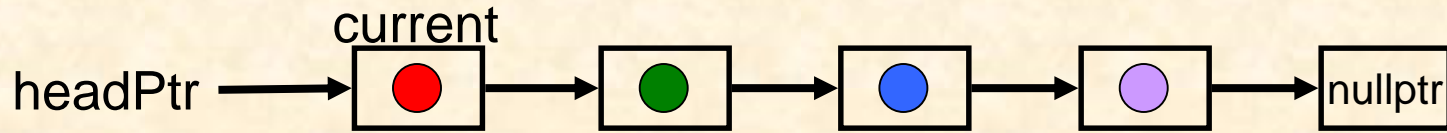
Reversing a Linked List: Solution 1

Similar to copying

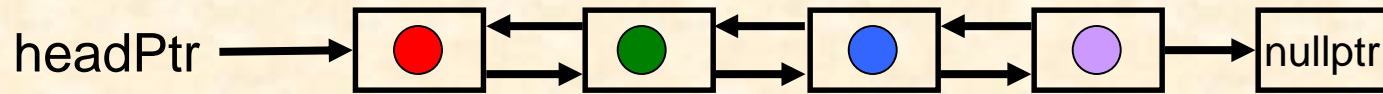



Now delete old list and set **headPtr = newHPtr**

Reversing a Linked List: Solution 2



Reversing a Doubly-linked List



- Can we reverse this in $O(1)$ memory?
- What about in $O(1)$ time? 

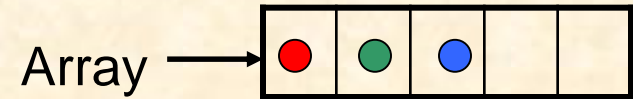
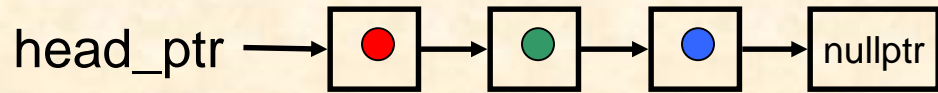
What if we add a tail pointer?

- Does this reduce memory needed?
- Does this reduce time?

STL lets you access it in reverse order in $O(1)$ time
using a `reverse_iterator`

Traversing Linked Lists and Arrays

Printing requires a traversal



```
void LinkedList::print() {  
    node *current = headPtr;  
  
    while(current != nullptr) {  
        cout << current->item << endl;  
        current = current->next;  
    } // while  
} // print()
```



I'm seeing
double!

```
void Array::print() {  
    int *current = data;  
  
    while(current != data + size) {  
        cout << *current << endl;  
        current++;  
    } // while  
} // print()
```


I'm seeing
double!




Can the same code work with linked lists and arrays?

Generic Programming

- Datatype-independent way of programming
- Code reused for data structures & algorithms supporting same interface
 - Example: Lists and Arrays
 - Algorithm Example: Square or ++ every number in a container


```
1 // Generic code to print a series of items
2 template<class InputIterator>
3 void genPrint(InputIterator itBegin, InputIterator itEnd) {
4     while (itBegin != itEnd) { 
5         cout << *itBegin << endl;
6         itBegin++;
7     } // while
8 } // genPrint()
```

What is an InputIterator?

```
// new LinkedList print code
void LinkedList::print() {
    genPrint(this->begin(),
              this->end()); 
} // print()
```

```
// new Array print code
void Array::print() {
    genPrint(this->begin(),
              this->end());
} // print()
```

Iterators for Linked Lists

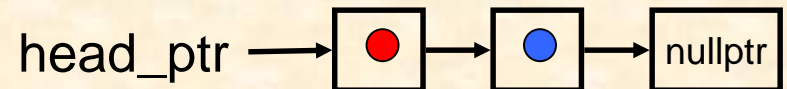
```
1  class ListIter {
2      Node *ptr;
3
4  public:
5      ListIter() : ptr(nullptr) {}
6      ListIter(Node *n) : ptr(n) {}
7
8      // allows access of data using *
9      const double& operator*() {
10         return ptr->item;
11     } // operator*()
12
13     // allows itr++ 
14     void operator++(int) {
15         ptr = ptr->next;
16     } // operator++()
17
18     // allows ++itr
19     void operator++() {
20         ptr = ptr->next;
21     } // ++operator()
22     //--- Insert more methods here
23 }; // ListIter
```

```
// Add these to LinkedList

ListIter LinkedList::begin() {
    return ListIter(headPtr);
} // begin()

ListIter LinkedList::end() {
    return ListIter();
} // end()
```

Why does end() return an empty ListIter?

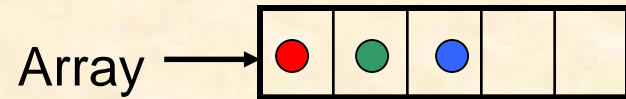


What other methods would be useful?

Iterators may be read-write or const (similar to read-write/const pointers)

Iterators for Arrays

```
1 // Add these to Array class
2
3 int *Array::begin() {
4     return data;
5 } // begin()
6
7 int *Array::end() {
8     return data + size;
9 } // end()
```



Why don't we need an iterator class for arrays
(the built-in type)?

Types of Iterators

- Access members of a container class
- Similar to pointers

<code>input_iterator</code>	Read values with forward movement. Can be incremented, compared, and dereferenced.
<code>output_iterator</code>	Write values with forward movement . Can be incremented, and dereferenced.
<code>forward_iterator</code>	Read or write values with forward movement. Can be incremented, compared, dereferenced, and store the iterators value.
<code>bidirectional_iterator</code>	Same as <code>forward_iterator</code> but can also decrement (default in C++).
<code>random_iterator</code>	Same as <code>bidirectional_iterator</code> but can also do pointer arithmetic and pointer comparisons.
<code>reverse_iterator</code>	An iterator adaptor (that inherits from either a <code>random_iterator</code> or a <code>bidirectional_iterator</code>) whose <code>++</code> operation moves in reverse.

Destroying a Linked List

- Since we allocated dynamic memory via `new`, we should at least have a destructor
 - Should also add copy constructor and `operator=()`

Terrible way to Destruct

```
~LinkedList() {  
    deleteList(headPtr);  
}
```

```
deleteList(Node *start) {  
    if (start) {  
        deleteList(start->next);  
        delete start;  
    }
```



Top 10 Study Questions

1. When would a linked list be preferred over an array?
2. How are linked lists and arrays similar?
3. What methods are faster with doubly-linked lists?
4. What methods are faster with a tail pointer?
5. What does it mean to check that a linked list is consistent?
6. How can you tell if a linked list is circular?
7. What is generic programming?
8. What is the difference between traversal and iteration?
9. What is an iterator?
10. Why are iterators useful?

