Lecture 16
Linked Lists

When to Use Different Data Structures for Collections (1/2)
- Many data structures store a collection of elements
- ArrayLists are called that because they implement Java-FX's List interface (defined soon) and are implemented using Arrays
- We can define a building block called Linked List, which can be an alternative to ArrayLists in some cases by avoiding data movement for insertion and deletion

When to Use Different Data Structures for Collections (2/2)
- Using Linked List of Nodes, can construct higher level abstractions to model collections (e.g., NodeList to parallel ArrayList, as well as Stacks, Queues, etc.)
- How to decide between data structures?
  - choose based on the way data is accessed and stored in your algorithm
  - access and store operations of different data structures can have very different impacts on an algorithm's overall efficiency - recall Big-O analysis

What is a Linked List? (1/2)
- A collection of linked nodes that form a sequence of elements
  - as with Arrays and ArrayLists, it can represent an unordered set or an ordered sequence of your data
  - algorithms can take advantage of fact that elements are stored sequentially, or not
- A Linked List holds reference to its first node (head) and its last node (tail)

What is a Linked List? (2/2)
- Each node holds an element and a reference to next node in list
- Most methods will involve:
  - "pointer-chasing" through the Linked List (for search and finding the correct place to insert or delete)
  - breaking and resetting the Linked List to perform the insertion or deletion
- But there won't be data movement! Hence efficient for dynamic collections

Ex: HTA Linked List

Note: this is an instance diagram, not a class diagram, because it has specific values!
Data Structure Comparison

Array
- indexed (explicit access to n° item)
- if user moves elements during insertion or deletion, their indices will change correspondingly
- cannot change size dynamically

ArrayList
- indexed (explicit access to n° item)
- indices of successor items automatically updated following an inserted or deleted item
- can grow/shrink dynamically
- use nodes instead of Arrays
- can insert or remove in the middle of list without data movement through the rest of the list

Linked List
- not indexed — in order to access the n° element, must start at the beginning and go to the next node n times — no random access!
- can grow/shrink dynamically
- use nodes instead of Arrays
- can insert or remove in the middle of list without data movement through the rest of the list

Linked List Implementations
- Find java.util implementation at: http://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html
- To learn list processing, define our own implementation of this data structure, MyLinkedList:
  - difference between MyLinkedList and Java's implementation is that Java uses something like our MyLinkedList to build a more advanced data structure that implements List
  - while there is overlap, there are also differences in the methods provided, their names, and their return types
- MyLinkedList is a building block for more specialized versions: Stacks, Queues, Sorted Linked Lists...
- We'll start by defining a Singly Linked List for both unsorted and sorted items, then we'll define a Doubly Linked List

Generic Unsorted Singly Linked List (1/3)

public class MyLinkedList<Type> {
    private Node<Type> _head;
    private int _size;
    public MyLinkedList() {
        // more on next slide
    }
    public Node<Type> getHead() {
        // still more on next slide
    }
    public Node<Type> getTail() {
        // still more on next slide
    }
    public boolean isEmpty() {
        // still more on next slide
    }
    public int size() {
        // still more on next slide
    }
    public boolean isEmpty() {
        // still more on next slide
    }
    public Node<Type> addFirst(Type e1) {
        // still more on next slide
    }
    public Node<Type> addLast(Type e1) {
        // still more on next slide
    }
    public Node<Type> remove(Type e1) {
        // still more on next slide
    }
    public Node<Type> search(Type e1) {
        // still more on next slide
    }
    // more on next slide
}

GNUC type parameter

Generic Unsorted Singly Linked List (2/3)

public class MyLinkedList<Type> {
    private Node<Type> _head;
    private int _size;
    public MyLinkedList() {
        // more on next slide
    }
    public Node<Type> getHead() {
        // still more on next slide
    }
    public Node<Type> getTail() {
        // still more on next slide
    }
    public boolean isEmpty() {
        // still more on next slide
    }
    public int size() {
        // still more on next slide
    }
    public boolean isEmpty() {
        // still more on next slide
    }
    public Node<Type> addFirst(Type e1) {
        // still more on next slide
    }
    public Node<Type> addLast(Type e1) {
        // still more on next slide
    }
    public Node<Type> remove(Type e1) {
        // still more on next slide
    }
    public Node<Type> search(Type e1) {
        // still more on next slide
    }
    // more on next slide
}

Generic Unsorted Singly Linked List (3/3)

public class MyLinkedList<Type> {
    private Node<Type> _head;
    private int _size;
    public MyLinkedList() {
        // more on next slide
    }
    public Node<Type> getHead() {
        // still more on next slide
    }
    public Node<Type> getTail() {
        // still more on next slide
    }
    public boolean isEmpty() {
        // still more on next slide
    }
    public int size() {
        // still more on next slide
    }
    public boolean isEmpty() {
        // still more on next slide
    }
    public Node<Type> addFirst(Type e1) {
        // still more on next slide
    }
    public Node<Type> addLast(Type e1) {
        // still more on next slide
    }
    public Node<Type> remove(Type e1) {
        // still more on next slide
    }
    public Node<Type> search(Type e1) {
        // still more on next slide
    }
    // more on next slide
}

Generic Singly Linked List Overview

public class MyLinkedList<Type> {
    private Node<Type> _head;
    private int _size;
    public MyLinkedList() {
        // more on next slide
    }
    public Node<Type> getHead() {
        // still more on next slide
    }
    public Node<Type> getTail() {
        // still more on next slide
    }
    public boolean isEmpty() {
        // still more on next slide
    }
    public int size() {
        // still more on next slide
    }
    public boolean isEmpty() {
        // still more on next slide
    }
    public Node<Type> addFirst(Type e1) {
        // still more on next slide
    }
    public Node<Type> addLast(Type e1) {
        // still more on next slide
    }
    public Node<Type> remove(Type e1) {
        // still more on next slide
    }
    public Node<Type> search(Type e1) {
        // still more on next slide
    }
    // more on next slide
}
### The Node Class

- Constructor initializes instance variables `_element` and `_next`
- Its methods are made up of accessors and mutators for these variables:
  - `getNext()` and `setNext()`
  - `getElement()` and `setElement()`

```java
public class Node<Type> {
    private Node<Type> _next;
    private Type _element;

    public Node(Type element) {
        _next = null;
        _element = element;
    }

    public Node<Type> getNext() {
        return _next;
    }

    public void setNext(Node<Type> next) {
        _next = next;
    }

    public Type getElement() {
        return _element;
    }

    public void setElement(Type element) {
        _element = element;
    }
}
```

### Ex: A pile of Books (1/2)

- Let’s use a Linked List to model a simple unorganized pile (i.e., set) of Books
- The elements in our pile will be of type `Book`
  - have titles, authors, dates, and ISBN numbers
  - we want a list that can store anything that “is a” `Book`

### Book Class

- Constructor stores author, date and ISBN number of `Book` as instance variables
- For each property, its `get` returns that property’s value
  - `getISBN()` returns `_isbn`

```java
public class Book {
    private String _author;
    private String _title;
    private int _isbn;

    public Book(String author, String title, int isbn) {
        _author = author;
        _title = title;
        _isbn = isbn;
    }

    public int getISBN() {
        return _isbn;
    }

    // other accessor methods elided
}
```

### PileOfBooks Class

- Contains a `MyLinkedList` of books as underlying data structure – it’s a thin wrapper
- `Book` is our generic Type
- Instantiating a `MyLinkedList` is entirely similar to instantiating an `ArrayList`

```java
public class PileOfBooks {
    private MyLinkedList<Book> _books;

    public PileOfBooks() {
        _books = new MyLinkedList<Book>();
    }

    // There could be many more methods here!
}
```

### Ex: MyLinkedList<Book>

- Note: The `LinkedList` class is in the instance with `_head` and `_tail` and the set of linked `Nodes` distributed in memory
**addFirst** — empty list

- If list is empty, _head and _tail will be null
  - Create a new Node/<Type> _head
  - Update new node's _next variable to null, which is where current _head points in this case
  - Update the _head and _tail variables to the new node

**addFirst** — non empty

- Create a new Node
  - Update its _next variable to current _head (in this case, some previously added Node _head that headed list)
  - Update _head variable to the new Node

**Constructor and addFirst Method (1/2)**

- Constructor
  - Initialize instance variables
- addFirst method
  - Increment _size by 1
  - Create new Node (constructor stores el in _element, null in _next)
  - Update new node's _next to first Node (pointed to by _head)
  - Update _head to point to new node
  - If _size is 1, _tail also points to new node
  - Return new node

**Constructor and addFirst Runtime (2/2)**

- addFirst method
  - Return new node
  - ...
addLast Runtime

```
public Node<Type> addLast(Type el) {
    Node<Type> newNode = new Node<Type>(el); // 1 op
    if (_size == 0) {
        // 1 op
        _head = newNode; // 1 op
        _tail = newNode; // 1 op
    } else {
        _tail.setNext(newNode); // 1 op
        _tail = newNode; // 1 op
    }
    _size++; // 1 op
    return newNode; // 1 op
}
```

\[ \text{addLast}(\text{Type el}) \in O(1) \]

size and isEmpty Methods

```
public int size() {
    return _size; // 1 op
}

public boolean isEmpty() {
    return _size == 0; // 1 op
}
```

\[ \text{size and isEmpty} \in O(1) \]

removeFirst Runtime

```
public Type removeFirst() {
    if (_size == 0) {
        System.out.println("List is empty"); // 1 op
        return null; // 1 op
    }
    Type removed = _head.getElement(); // 1 op
    _head = _head.getNext(); // 1 op
    _size--; // 1 op
    if (_size == 0) { // 1 op
        _tail = null; // 1 op
    }
    return removed; // 1 op
}
```

\[ \text{removeFirst} \in O(1) \]
removeLast Method

- As with removeFirst, remove Node by removing any references to it
- Pointer-chase in a loop to get predecessor to _tail and reset predecessor’s _next instance variable to null
  - pretty inefficient—stay tuned
- Last Node is thereby garbage-collected!

```java
public Type removeLast() {
    if (_size == 0) {
        return removed;
    }
    else if (_size == 1) {
        removed = _head.getElement();
        _head = null;
        _size = 0;
        return removed;
    }
    else { //classic pointer-chasing loop
        Node prev = null;
        Node curr = _head;
        while (curr != null) {
            prev = curr;
            curr = curr.getNext();
        }
        removed = curr.getElement();
        prev.setNext(null); //unlink last
        _tail = prev; //update _tail
        _size = _size - 1;
    }
    return removed;
}
```

removeLast Method

- Edge case(s)
  - can’t delete from an empty list
  - if there is only one Node, update _head and _tail references to null
- General case
  - iterate (pointer-chase) through list—common pattern
  - after loop ends, prev will point to Node just before last Node and curr will point to last Node

```java
public Type removeLast() {
    if (_size == 0) {
        System.out.println("List is empty");
        return removed;
    }
    else if (_size == 1) {
        removed = _head.getElement();
        _head = null;
        _size = 0;
        return removed;
    }
    else { //classic pointer-chasing loop
        Node prev = null;
        Node curr = _head;
        while (curr != null) {
            prev = curr;
            curr = curr.getNext();
        }
        removed = curr.getElement();
        prev.setNext(null); //unlink last
        _tail = prev; //update _tail
        _size = _size - 1;
    }
    return removed;
}
```
removeLast Method

```java
public Type removeLast() {
    Type removed = null;
    if (_size == 0) {
        System.out.println("List is empty");
        return removed;
    } else if (_size == 1) {
        removed = _head.getElement();
        _head = null;
        _size--;
    } else {
        Node<Type> prev = _head;
        Node<Type> curr = _head;
        while (curr.getNext() != null) {
            prev = curr;
            curr = curr.getNext();
        }
        prev.setNext(null); // unlink last
        removed = curr.getElement();
    }
    System.out.println("Not Empty");
    return removed;
}
```

search Method

- Let’s think back to our pile of Books example — what if we want to find a certain Book in the pile of Books? What if we want to search by ISBN, author, or title?
- Must compare each Book with one we are looking for — but in order to do this, we first need a way to check for the equality of two elements.
Java's Comparable<Type> interface (1/3)

- Previously we used == to check if two things are equal
  - this only works correctly for primitive data types (e.g., int), or when we are comparing two variables referencing the exact same object
  - in compare Strings, need a different way to compare strings
- We can implement the Comparable<Type> interface provided by Java
- Must define compareTo method, which returns an int
- Why don't we just use ==, even when using something like ISBN, which is an int?
  - Can't treat ISBNs as ints and compare them directly, but more generally we implement the Comparable<Type> interface, which could easily accommodate comparing strings such as author or title

Java's Comparable<Type> interface (2/3)

- The Comparable<Type> interface is specialized (think of it as parameterized) using generics
  ```java
  public interface Comparable<Type> {
    public int compareTo(Type toCompare);
  }
  ```
- Call compareTo on a variable of same type as specified in implementation of interface (Book, in our case)
- currentBook.compareTo(bookToFind);

Java's Comparable<Type> interface (3/3)

- compareTo method must return an int
  - negative if element on which compareTo is called is less than element passed in as the parameter of the search
  - 0 if element is equal to element passed in
  - positive if element is greater than element passed in
- compareTo not only used for numerical comparisons — it could be used for alphabetical or geometric comparisons as well — depends on how you implement compareTo

“Comparable” Book Class

```java
public class Book implements Comparable<Book> {
    public int compareTo(Book bookToCompare) {
        return (_isbn < bookToCompare.getISBN()) ? -1 :
        (_isbn == bookToCompare.getISBN()) ? 0 : 1;
    }
}
```

“Comparable” Singly Linked List

```java
public class MyLinkedList implements Comparable<Book> {
    public int compareTo(Book bookToCompare) {
        return (_isbn < bookToCompare.getISBN()) ? -1 :
        (_isbn == bookToCompare.getISBN()) ? 0 : 1;
    }
}
```

search Method for myLinkedList

```java
public Node<Book> search(Book el) {
    Node<Book> curr = _head;
    while (curr != null) {
        if (curr.compareTo(el) == 0) {
            return curr;
        }
        curr = curr.getNext(); // hop pointer
    }
    return null; // element not found
}
```
remove Method

- We have implemented methods to remove the first and the last elements of MyLinkedList
- What if we want to remove any element from MyLinkedList?
- Let's write a general remove method
  - will be similar to the search algorithm

remove Runtime

```
public Type remove(Type itemToRemove){
  if (this.isEmpty()) { // 1 op
    return null;
  }
  if (itemToRemove.compareTo(_head.getElement()) == 0) { // 1 op
    return this.removeFirst();
  }
  if (itemToRemove.compareTo(_tail.getElement()) == 0) { // 1 op
    return this.removeLast();
  }
  System.out.println("List is empty");
  return null; //return null if itemToRemove is not found
}
```

Ex: A sorted bookshelf

- Faster to find (and remove!) books in a sorted bookshelf
- Use a sorted linked list
  - makes several of our methods more efficient:
    - search
    - insert
    - delete
- Sort in increasing order
  - maintain sort order when inserting
**Search Method [For Sorted Linked Lists]**

- Must iterate through list until `toFind` is found.
- Compare `toFind` to `curr`'s element.
  - If `toFind` is equal to `curr`, we're done!
  - If `curr`'s element is greater than `toFind`, stop search.
    - Any following node's elements will also be greater since list is sorted.
    - Don't usually have to maintain sort order, i.e., by having to insert in the right place.

**Search Runtime [For Sorted Linked Lists]**

- Linear time! (Big O run time!)

**Insert Method**

- Once again, iterate through nodes with a while loop.
- Unlike insertion into unsorted linked list, there is one correct spot in list for new node.
- End iteration if current node's value is greater than new node's value—break the chain and insert there!
- Update next pointers of new node and previous node.

**What Did Sorting Buy Us?**

- Search still O(n)—not better worst-case performance, but better for normal cases because can usually stop earlier for items not in the list.
- This comes at the cost of having to maintain sort order.
  - If an algorithm does lots of searching compared to insertion and deletion, this efficiency would pay off; conversely, if an algorithm does a lot of adding and deleting compared to searching, it wouldn't since the algorithm would use the simple O(1) `insertFirst` or `insertLast`...
insert Method
[for Sorted Linked Lists]
- Edge case
  o if list is empty, all we have to do is reset _head_tail
- General case
  o iterate over lists until curr's element is greater than newItem
  o need loop's prev, so we can re-link list to integrate the new node
  o or if not found, special case

```java
public Node<Type> insert(Type newItem)
{
    Node<Type> toAdd = new Node<Type>(newItem);
    Node<Type> prev = _head;
    Node<Type> curr = _head.getNext();
    while (curr != null)
    {
        if (curr.getElement().compareTo(newItem) == 0)
        {
            return null;
        }
        else if (curr.getElement().compareTo(newItem) > 0)
        {
            toAdd.setNext(curr);
            curr.setPrev(toAdd);
            return toAdd;
        }
        prev = curr;
        curr = curr.getNext();
    }
    return null;
}
```

remove Method
[for Sorted Linked Lists]
- Loop through nodes until an _element matches ItemToRemove
  o since list is sorted, we can end loop early — stay tuned
- Re-link predecessor of node (again using a previous node) to successor of node (its _next reference)
- With no more reference to node, it is garbage collected at the termination of the method

```java
public Node<Type> remove(Type itemToRemove)
{
    if (this.isEmpty())
    {
        return null;
    }
    if (itemToRemove.compareTo(_head.getElement()) == 0)
    {
        _head = _head.getNext();
        _head.setPrev(null);
        if (_head != null)
        {
            _head.setPrev(null);
        }
        return itemToRemove;
    }
    else
    {
        Node<Type> curr = _head;
        Node<Type> prev = null;
        while (curr != null)
        {
            if (curr.getElement().compareTo(itemToRemove) < 0)
            {
                prev = curr;
                curr = curr.getNext();
            }
            else if (curr.getElement().compareTo(itemToRemove) > 0)
            {
                return null;
            }
            else
            {
                prev.setNext(curr.getNext());
                curr.setPrev(prev);
                return itemToRemove;
            }
        }
        return null;
    }
}
```

insert Runtime [for Sorted Linked Lists]
- Edge case(s)
  o if list is empty, return null
  o if ItemToRemove in the _head_tail, use same code as removeFirst
    removeLast in MyLinkedList
- General case
  o iterate over list until either:
    • ItemToRemove is found (equal to curr), so we have to re-link node and return found item
    • or if list is empty, return null
      Or if list is empty, return null
      But it does not affect Big-O runtime

```java
public void insert(Node<Type> toAdd)
{
    if (_head != null)
    {
        Node<Type> prev = _head;
        Node<Type> curr = _head.getNext();
        while (curr != null)
        {
            if (curr.getElement().compareTo(toAdd) == 0)
            {
                return;
            }
            else if (curr.getElement().compareTo(toAdd) > 0)
            {
                toAdd.setNext(curr);
                curr.setPrev(toAdd);
                return;
            }
            prev = curr;
            curr = curr.getNext();
        }
        return;
    }
}
```

remove Runtime [for Sorted Linked Lists]
- If ItemToRemove is found (equal to curr), so we have to re-link node and return found item
- Or if list is empty, return null
  • Or if list is empty, return null
  • But it does not affect Big-O runtime

remove Method [for Sorted Linked Lists]
- Edge case(s)
  o if list is empty, return null
  o if ItemToRemove in the _head_tail, use same code as removeFirst
    removeLast in MyLinkedList
- General case
  o iterate over list until either:
    • ItemToRemove is found (equal to curr) so we have to re-link node and return found item
    • or if list is empty, return null
      Or if list is empty, return null
      But it does not affect Big-O runtime

```java
public void remove(Type itemToRemove)
{
    if (this.isEmpty())
    {
        return;
    }
    if (itemToRemove.compareTo(_head.getElement()) == 0)
    {
        _head = _head.getNext();
        _head.setPrev(null);
        if (_head != null)
        {
            _head.setPrev(null);
        }
        return;
    }
    else
    {
        Node<Type> curr = _head;
        Node<Type> prev = null;
        while (curr != null)
        {
            if (curr.getElement().compareTo(itemToRemove) < 0)
            {
                prev = curr;
                curr = curr.getNext();
            }
            else if (curr.getElement().compareTo(itemToRemove) > 0)
            {
                return;
            }
            else
            {
                prev.setNext(curr.getNext());
                curr.setPrev(prev);
                return;
            }
        }
        return;
    }
}
```

Clicker Question
How do sorted and unsorted lists differ?

A. Sorted lists are more efficient than unsorted lists and are more easily searched but their insertion takes longer.
B. Sorted lists are less efficient than unsorted lists and are less easily searched, but their insertion takes less time.
C. Sorted lists are more efficient than unsorted lists and are more easily searched and their insertion takes less time.
D. Sorted lists are more efficient than unsorted lists, but are less easily searched and their insertion takes longer.
Doubly Linked List (1/3)

- Is there an easier/faster way to get to the previous node while removing a node?
  - With Doubly Linked Lists, nodes have references to both next and previous nodes.
  - Can traverse list both backwards and forwards—Linked List still stores reference to front of the list with _head and back of the list with _tail.
  - Modify Node class to have two pointers: _next and _prev.

Doubly Linked List (2/3)

- For Singly Linked List, the processing typically goes from first to last node, e.g., search, finding the place to insert or delete.
- Sometimes, particularly for a sorted list, need to go in the opposite direction.
  - e.g., sort students on their final grades. Find the lowest numeric grade that will be recorded as an "A". Now ask: who is close to getting an "A", i.e., borderline?

Doubly Linked List (3/3)

- This kind of backing up can't easily be done with the Singly Linked List implementation we have so far.
  - Could build our own specialized search method, which would scan from the _head and be, at a minimum, O(n).
- It is trivial for Doubly Linked Lists:
  - Find student with the lowest "A" using search.
  - Use the _prev pointer, which points to the predecessor of a node (O(1)), and back up until hit end of B+/A-grey area.

Remove method [For Sorted Doubly Linked List]

- This is pseudo-code for a remove method for a Sorted Doubly Linked List.
  - Note dot notation for variables—a common practice for pseudo-code.
  - This is incomplete: does not deal with edge cases (element not found, last in list, etc.).
- Other methods are analogous.

```
Node [For Doubly Linked Lists]

public class Node<TYPE> {

  private TYPE _element;
  private Node<TYPE> _next;
  private Node<TYPE> _prev;

  public Node(TYPE element) {
    _element = element;
    _next = null;
    _prev = null;
  }

  public void setNext(Node<TYPE> next) {
    _next = next;
  }

  public Node<TYPE> getNext() {
    return _next;
  }

  public void setPrev(Node<TYPE> prev) {
    _prev = prev;
  }

  public Node<TYPE> getPrev() {
    return _prev;
  }
}
```

Clicker Question

What methods do Doubly Linked Lists Nodes have that Singly Linked Lists Nodes do not have?

A. getNext(), setNext()  
B. getElement(), setElement() 
C. getPrev(), setPrev()  
D. getLocation(), setLocation()
Summary of Linked Lists

- Linked Lists can be grouped into two categories:
  - Unsorted vs. Sorted Linked List
  - Unsorted Linked List has two different implementations with identical method signatures and return types
  - Sorted Linked List also has two different implementations with identical method signatures and return types

- Implementation matters! While an Unsorted Singly Linked List and an Unsorted Doubly Linked List have the same functionality, they have different runtimes for their methods.
  - Pick implementation based on how you expect to use the data structure

Circular Doubly Linked Lists

- No beginning or end
- Example: Rolodex
- In operating systems, these structures are called rings

When to use a Linked List?

- Might use a Linked List if...
  - do not need random access (based on an index)
  - often need to insert elements into or delete from the interior of a sorted list
  - do not know how many elements will be stored beforehand (then still need to trade-off using List vs. ArrayList)

Linked List Exercises

How To Build A Node List

- Now that we have a building block, there are a number of methods we can implement to make a higher-level NodeList that implements Java's List interface (like ArrayList does)
  - Note: List interface is very general...
- Main addition List mandates is to support indexing into the NodeList. Let's write one of the simpler ones:
  - get(int i) method that returns element(Type) at that index

search Private Helper Method

- First define a search helper method to return node at a particular index
- Want to use this helper method in the class, but don't want to expose found nodes publicly, that would violate encapsulation - make helper private
- If a provided index is out of bounds, return null (print line is an optional error message)
- Otherwise, iterate through list until node at desired index is reached and return that node
search Private Helper Method Runtime

```java
private Node<Type> search(int i) {
    if (i >= _size || i < 0) {
        // 1 op
        System.out.println("Invalid index");
        // 1 op
        return null;
    }
    Node<Type> curr = _head;
    // 1 op
    for (int counter = 0; counter < i; counter++) {
        // n ops
        curr = curr.getNext();
    }
    return curr;
    // 1 op
}
```

Public Wrapper Method

- Write the publicly accessible wrapper code for the NodeList's get method.

```java
public Type get(int i) {
    return this.search(i).getElement();
}
```

Solution A

- If list is empty or has 1 node, return list
- Otherwise, create a new list of same type as input list
- Iterate through input list, removing first element each time and adding it as first element of new list

```java
public MyLinkedList<Type> reverse(MyLinkedList<Type> toReverse) {
    if (toReverse.size() < 2) {
        return toReverse;
    }
    MyLinkedList<Type> newList = new MyLinkedList<Type>();
    int origSize = toReverse.size();
    while (newList.size() < origSize) {
        newList.addFirst(toReverse.removeFirst());
    }
    return newList;
}
```

Solution B (1/2)

- Is there a better way?
- First algorithm reversed in O(n) time
  - but it wasn't in-place – (had to create a new list)
- Can write a method within MyLinkedList that reverses itself without creating new nodes
  - still O(n) but in-place and therefore more efficient

```java
public void reverse() {
    Node<Type> prev = null;
    Node<Type> curr = _head;
    Node<Type> next = null;
    _tail = _head;
    // set tail to head
    while (curr != null) {
        next = curr.getNext();
        curr.setNext(prev);
        prev = curr;
        curr = next;
    }
    _head = prev;
}
```

Solution B (2/2)

- Keep track of previous, current, and next node
- While current node isn't null, iterate through nodes, resetting node pointers in reverse
- In doing so, must be careful not to delete any references further on in the list
- Finally, set the _head pointer to what had been the last node (held in the prev variable)
- If the list is empty _curr will be null, so the loop will never begin and _head will continue to point to null

```java
public void reverse() {
    Node<Type> prev = null;
    Node<Type> curr = _head;
    Node<Type> next = null;
    _tail = _head;
    // set tail to head
    while (curr != null) {
        next = curr.getNext();
        curr.setNext(prev);
        prev = curr;
        curr = next;
    }
    _head = prev;
}
```
Solution B Walkthrough (1/15)

prev = null
curr = null

Solution B Walkthrough (2/15)

next = null
prev = null

Solution B Walkthrough (3/15)

prev = null
curr = null

Solution B Walkthrough (4/15)

prev = null
curr = null

Solution B Walkthrough (5/15)

prev = null
curr = null

Solution B Walkthrough (6/15)

prev = null
curr = null
while(curr!=null) {
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}

while(curr!=null) {
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
Solution B Walkthrough (13/15)

Solution B Walkthrough (14/15)

Solution B Walkthrough (15/15)

Announcements

- DoodleJump due dates
  - on-time hand-in is today at 11:59 pm
  - late hand-in is Saturday at 10:00 pm
- We are now in MacMillan 117!