Lecture 18
Data Structures I:
Linked Lists

Outline
• Linked Lists
• Stacks and Queues (next lecture)
• Trees (next lecture)
• HashSets and HashMaps (next lecture)

Linked Lists
Tribute 1 → Tribute 2 → Tribute 3 → Tribute 4
**What is a LinkedList? (1/2)**

- Collection of nodes stored anywhere in memory linked in a "daisy chain" to form sequence of elements
  - as with Arrays and ArrayLists, it can represent an unordered set or an ordered (sorted) sequence of data elements
- A LinkedList holds a reference (pointer) to its first node (head) and its last node (tail) – internal nodes maintain list via their references to their next nodes

**What is a LinkedList? (2/2)**

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

**Ex: HTA LinkedList**

Note that this is an instance diagram, not a class diagram, because it has specific values!
When to Use Different Data Structures for Collections (1/2)

- **ArrayLists** get their name because they implement Java's `List` interface (defined soon) and are implemented using `Arrays`.
- **LinkedLists** also implement the `List` interface and are an alternative to `ArrayLists` that avoid data movement for insertion and deletion.
  - Uses pointer manipulation rather than moving elements in an array.

When to Use Different Data Structures for Collections (2/2)

- 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.
  - Even without N very large, there can be significant performance differences.
  - Roughly, **Arrays** if mostly static collection, **ArrayLists** if need more update dynamics while retaining easy accessibility, and **LinkedList** if more updates than accesses.

Data Structure Comparison

<table>
<thead>
<tr>
<th>Array</th>
<th>ArrayList</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexed (explicit access to ith item)</td>
<td>Indexed (explicit access to ith item)</td>
<td>Not indexed — to access the nth element, must start at the beginning and go to the next node n times — no random access!</td>
</tr>
<tr>
<td>If user moves elements during insertion or deletion, their indices will change correspondingly</td>
<td>Indices of successor items automatically updated following an inserted or deleted item</td>
<td>Can grow/shrink dynamically</td>
</tr>
<tr>
<td>Can't change size dynamically</td>
<td>Can grow/shrink dynamically</td>
<td>Uses nodes and pointers instead of <code>Arrays</code></td>
</tr>
<tr>
<td>Java uses an <code>Array</code> as underlying data structure (and does data shuffling itself)</td>
<td></td>
<td>Can insert or remove nodes anywhere in the list without data movement through the rest of the list</td>
</tr>
</tbody>
</table>
**Linked List Implementations (1/2)**

- Find Java.util implementation at: http://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html

- To learn list processing, we'll make our own implementation of this data structure, MyLinkedList (Mll):
  - difference between Mll and Java's implementation is that Java uses something like our Mll to build a more advanced data structure that implements Java's List interface
  - while there is overlap, there are also differences in the methods provided, and their names/return types
  - in CS200, you will use LinkedLists in your own programs

**Linked List Implementations (2/2)**

- MyLinkedList (Mll) is a general building block for more specialized data structures we'll build: 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—users of these data structures don't see any of these internals!
  - will implement Mll as a Singly Linked List in next few slides

**Singly Linked List (1/3)**

- Mll doesn't implement full List interface
- Linked list is maintained by head and tail pointers; internal structure changes dynamically
- Constructor initializes instance variables
  - head and tail are initially set to null
  - size set to 0
- addFirst() appends Node to front of list and updates head to reference it
- addLast() appends Node to end of list and updates tail to reference it

```java
class MyLinkedList {
    private Node head;
    private Node tail;
    private int size;

    public MyLinkedList() {
        this.head = null;
        this.tail = null;
        this.size = 0;
    }

    public Node addFirst(CS15TA el) {
        //...
    }

    public Node addLast(CS15TA el) {
        //...
    }
    // more on next slide
}
```
Singly Linked List (2/3)

- `removeFirst()` removes first Node and returns element
- `removeLast()` removes last Node and returns element
- `remove()` removes first occurrence of Node containing element `el` and returns it (implicit search)

```java
public Node<CS15TA> removeFirst() {
    //...
}
public Node<CS15TA> removeLast() {
    //...
}
public Node<CS15TA> remove(CS15TA el) {
    // still on next slide
```

Singly Linked List (3/3)

- `search()` finds and returns Node containing `el`
- `size()` returns size of list
- `isEmpty()` checks if list is empty (returns boolean)
- `getHead/getTail()` return reference to head/tail Node of list

```java
public Node<CS15TA> search(CS15TA el) {
    //...
}
public int size() {
    //...
}
public boolean isEmpty() {
    //...
}
public Node<CS15TA> getHead() {
    //...
}
public Node<CS15TA> getTail() {
    //...
```

Singly Linked List Summary

```java
public class MyLinkedList<CS15TA> {
    private Node<CS15TA> head;
    private Node<CS15TA> tail;
    private int size;

    public MyLinkedList() {
    //...
}
    public Node<CS15TA> addFirst(CS15TA el) {
    //...
}
    public Node<CS15TA> addLast(CS15TA el) {
    //...
}
    public Node<CS15TA> removeFirst() {
    //...
}
    public Node<CS15TA> removeLast() {
    //...
}
    public Node<CS15TA> search(CS15TA el) {
    //...
}
```
The Node Class

- Also uses generics: user of MLL specifies type and Java substitutes specified type in Node class’ methods
- Constructor initializes instance variables element and next
- Its methods are made up of accessors and mutators for these variables:
  - o getNext() and setNext()
  - o getElement() and setElement()
- Type is a placeholder for whatever object Node will hold

```java
public class Node<Type> {
    private Node<Type> next;
    private Type element;

    public Node(Type element) {
        this.next = null;
        this.element = element;
    }

    public Node<Type> getNext() {
        return this.next;
    }

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

    public Type getElement() {
        return this.element;
    }

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

Ex: A pile of Books

- Before implementing LinkedList’s internals, let’s see how to use one to model a simple pile of Books
  - o ‘user’ here is another programmer using the MyLinkedList we’re making
- Elements in our pile will be of type Book
  - o each has title, author(s), date and ISBN (International Standard Book Number)
  - o want list that can store any Book

```java
public class Book {
    private String author;
    private String title;
    private int isbn;

    public Book(String author, String title, int isbn) {
        this.author = author;
        this.title = title;
        this.isbn = isbn;
    }

    public int getISBN()
    { return this.isbn;
    }

    // other mutator and accessor methods elided
}
```

Book Class

- Book’s constructor stores author, date and ISBN number of Book as instance variables
- For each property, get method returns that property’s value
  - o ex. getISBN() returns isbn

```java
public class Book {
    private String author;
    private String title;
    private int isbn;

    public Book(String author, String title, int isbn) {
        this.author = author;
        this.title = title;
        this.isbn = isbn;
    }

    public int getISBN(){
        return this.isbn;
    }

    // other mutator and accessor methods elided
}
```
Implementation: addFirst - empty list

- If list is empty, head and tail are null
  - let's only show list pointers
- Create new Node<ElementType>
- Update new node's next variable to where head points to, which is null in this case
  - constructor already had null – we're accounting for general case
- Update head and tail variables to new node

addFirst - non empty

- Construct new Node
- Update its next variable to current head (in this case, some previously added Node that headed list)
- Update MLL's head variable to the new Node
Constructor and addFirst Method (1/2)

- Constructor — as shown before
  - Initialize instance variables
  - addFirst method
    - Increment size by 1
    - Create new Node (S15: constructor stores el in element, null in next)
    - Update newNode's next to first Node (pointed to by head)
    - Update MLL's head to point to newNode
    - If size is 1, tail must also point to newNode (edge case)
    - Return newNode

 Constructor and addFirst Runtime (2/2)

- Constructor is O(1)
- addFirst(Type el) is O(1)

addLast Method (1/2)

- MLL's tail already points to the last Node in the list
- Create a new Node<Type>
- Update tail's node's next pointer to the new node
- Then, update tail to the new Node
addLast Method (2/2)

- **Edge Case**
  - If list is empty, update head and tail variables to newNode

- **General Case**
  - Update next of current last Node (to which tail is pointing) to new last Node
  - Update tail to that new last Node
  - New Node’s next variable already points to null

```java
public Node<Type> addLast(Type el) {
    Node<Type> newNode = new Node<Type>(el);
    if (this.size == 0) {
        this.head = newNode;
        this.tail = newNode;
    } else {
        this.tail.setNext(newNode);
        this.tail = newNode;
    }
    this.size++;
    return newNode;
}
```

addLast Runtime

```java
public Node<Type> addLast(Type el) {
    Node<Type> newNode = new Node<Type>(el);
    if (this.size == 0) {
        this.head = newNode;
        this.tail = newNode;
    } else {
        this.tail.setNext(newNode);
        this.tail = newNode;
    }
    this.size++; // 1 op
    return newNode;
}
```

size and isEmpty Methods and Runtime

```java
public int size() { // 1 op
    return this.size;
}

public boolean isEmpty() { // 2 ops
    return this.size == 0;
}
```

- Edge Case
  - If list is empty, update head and tail variables to newNode

- General Case
  - Update next of current last Node (to which tail is pointing) to new last Node
  - Update tail to that new last Node
  - New Node’s next variable already points to null
**removeFirst Method (1/2)**

- Remove reference to original first Node by setting `head` variable to second Node, i.e., first Node's successor Node, via first's `next`
- Node to remove is garbage-collected after termination of method

```
Node<Type> head  
```

**removeFirst Method (2/2)**

- Edge case for empty list
  - `println` is optional, just one way to handle error checking; caller should check for null in any case
- Store data element from first Node to removed
- Then unchain first Node by resetting `head` to point to first Node's successor
- If list is now empty, update `tail` to null (what did `head` get set to?)
- Node to remove is garbage-collected at method's end

```
public Type removeFirst() {
    if (this.size == 0) {
        System.out.println("List is empty");
        return null;
    }
    Type removed = this.head.getElement();
    this.head = this.head.getNext();
    this.size --;
    if (this.size == 0) {
        this.tail = null;
    }
    return removed;
}
```

**removeFirst Runtime**

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

++ removeFirst() is O(1)
Review: Accessing Nodes Via Pointers

- This does not get `next` field of `head`, which doesn't have such a field, being just a pointer
- Instead, read this as "get `next` field of the node `head` points to"
- What does `this.tail.getNext()` produce?
- What does `this.tail.getElement()` produce?
- Note we can access a variable by its unique name, index, contents, or here, via a pointer

```
Node<Type> head
Node<Type> Node<Type> next
Type element
...
Node<Type>
Node<Type> Node<Type> next
Type element
Node<Type>
Node<Type> next
Type element
null
```

TopHat Question

Given a Linked List of Nodes,

```
A -> B -> C -> D
```

where `head` points to node `A`, what is `this.head.getNext().getNext()`?

A. Nothing, throws a `NullPointerException`
B. B
C. C
D. D

```
Node<Type> head
Node<Type> tail
Node<Type> last node
null
```

removeLast Method

- As with `removeFirst`, remove `Node` by removing any references to it. Need to know predecessor, but no pointer to it!
- "Pointer-chase" in a loop until predecessor's `next` is `tail` and reset predecessor's `next` instance variable to null
- Very inefficient, stay tuned
- Update `tail`
- Last Node is thereby garbage-collected!
removeLast Method

• Edge case(s)
  o can’t delete from empty list
  o if there’s only one Node, update 'head' and 'tail' references to null

• General case
  o iterate ("pointer-chase") through list – common pattern using pointers to current and previous node in lockstep
  o after loop ends, prev will point to last Node and curr will point to last Node

public Type removeLast() {
  Type removed = null;
  if (this.size == 0) {
    System.out.println("List is empty");
    this.head = null;
    this.tail = null;
    this.size = 0;
  } else if (this.size == 1) {
    removed = this.head.getElement();
    this.head = null;
    this.size --;
  } else {
    // classic pointer-chasing loop
    Node curr = this.head;
    Node prev = null;
    while (curr.getNext() != null) {
      // bop the pointers
      prev = curr;
      curr = curr.getNext();
    }
    removed = curr.getElement();
    prev.setNext(null); // unlink last
    this.tail = prev;
    this.size --;
  }
  return removed;
}

• Edge case(s)
  o can’t delete from empty list
  o if there’s only one Node, update 'head' and 'tail' references to null

• General case
  o iterate ("pointer-chase") through list – common pattern using pointers to current and previous node in lockstep
  o after loop ends, prev will point to last Node and curr will point to last Node

public Type removeLast() {
  Type removed = null;
  if (this.size == 0) {
    System.out.println("List is empty");
    this.head = null;
    this.tail = null;
    this.size = 0;
  } else if (this.size == 1) {
    removed = this.head.getElement();
    this.head = null;
    this.size --;
  } else {
    // classic pointer-chasing loop
    Node curr = this.head;
    Node prev = null;
    while (curr.getNext() != null) {
      // bop the pointers
      prev = curr;
      curr = curr.getNext();
    }
    removed = curr.getElement();
    prev.setNext(null); // unlink last
    this.tail = prev;
    this.size --;
  }
  return removed;
}
TopHat Question
Given that `animals` is a Singly Linked List of `n` animals, what is `node` pointing to?

```java
curr = this.head;
prev = null;
while (curr.getNext().getNext() != null) {
    prev = curr;
    curr = curr.getNext();
} 
node = curr.getNext();
```

A. Nothing useful, throws a `NullPointerException`  
B. Points to the last node on the list  
C. Points to the second node on the list  
D. Points to the head of the list

---

search Method for `MyLinkedList`

- Loop through list until element is found or end is reached (`curr==null`)  
- If a Node’s element is same as the argument, return `curr`  
- If no elements match, return `null`

```java
public Node<Type> search(Type el) {
    Node<Type> curr = this.head;
    while (curr != null) {
        if (curr.getElement().equals(el)) {
            return curr;
        }
        curr = curr.getNext();
    }
    return null; // got to end of list w/o finding
}
```
remove Method

- We have implemented methods to remove first and last elements of MyLinkedList
- What if we want to remove any element from MyLinkedList?
- Let's write a general remove method
  - think of it in 2 phases:
    - a search loop to find correct element (or end of list)
    - breaking the chain to jump over the element to be removed

```java
public Type remove(Type itemToRemove)
{
    if (this.isEmpty())
    {
        System.out.println("List is empty");
        return null;
    }
    if (itemToRemove.equals(this.head.getElement()))
    {
        return this.removeFirst();
    }
    if (itemToRemove.equals(this.tail.getElement()))
    {
        return this.removeLast();
    }
    //advance to 2nd item
    Node<Type> curr = this.head.getNext();
    Node<Type> prev = this.head;
    while (curr != null)
    {
        if (curr.getElement().equals(itemToRemove))
        {
            prev.setNext(curr.getNext());
            //jump over node
            this.size--;
            //decrement size
            return curr.getElement();
        }
        prev = curr;
        curr = curr.getNext();
    }
    return null;
    //return null if itemToRemove is not found
}
```

Note: caller of remove can find out if item was successfully found (and removed) by testing for != null.
remove (Runtime)

```java
public Type remove(Type itemToRemove) {
    if (this.isEmpty()) {
        System.out.println("List is empty");
        return null;
    }
    if (itemToRemove.equals(this.head.getElement())) {
        return this.removeFirst();
    }
    if (itemToRemove.equals(this.tail.getElement())) {
        return this.removeLast();
    }
    Node<Type> curr = this.head.getNext();
    Node<Type> prev = this.head;
    while (curr != null) {
        if (itemToRemove.equals(curr.getElement())) {
            prev.setNext(curr.getNext());
            this.size--;
            return curr.getElement();
        }
        prev = curr;
        curr = curr.getNext();
    }
    return null;
}
```

TopHat Question

Given that `animals` is a Singly Linked List of `n` animals, `curr` points to the node with an animal to be removed from the list, that `prev` points to `curr`'s predecessor, and that `curr` is not the tail of the list, what will this code fragment do?

A. List is unchanged, prints out removed animal
B. List is unchanged, prints out the animal after the one that got removed
C. List loses an animal, prints out removed animal
D. List loses an animal, prints out the animal after the one that was removed

Doubly Linked List (1/3)

- Is there an easier/faster way to get to previous node while removing a node?
  - with Doubly Linked Lists, nodes have references both to next and previous nodes
  - can traverse list both backwards and forwards – Linked List still stores reference to front of list with `head` and back of list with `tail`
  - modify `Node` class to have two pointers: `next` and `prev`
  - eliminates pointer-chasing loop because `prev` points to predecessor of every `Node`, at cost of second pointer
    - classic space-time tradeoff!
Doubly Linked List (2/3)

- For Singly Linked List, processing typically goes from first to last node, e.g. search, finding place to insert or delete.
- Sometimes, particularly for sorted list, need to go in the opposite direction.
  - e.g., sort CS15 students on their final grades in ascending order. Find lowest numeric grade that will be recorded as an "A". Then ask: who has a lower grade but is closer to the "A" cut off, i.e., in the grey area, and therefore should be considered for "benefit of the doubt"?

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 simpler for Doubly Linked Lists:
  - find student with lowest "A" using search.
  - use prev pointer, which points to the predecessor of a node (O(1)), and back up until hit end of B+/A- grey area.

Announcements

- Tetris is out!
  - early handin: Saturday 11/11
  - on-time handin: Monday 11/13
  - late handin: Wednesday 11/15
  - Tetris Code-Along 11/08 7:00pm Friedman Hall
    - Recording on Website
- HTA hours in Friedman 101 Friday 3pm-4pm
  - come and chat about course registration, the upcoming final project or any other concerns you may have.
Cybersecurity and the Future of Warfare
CS15 Fall 2023

Cybersecurity: A Brief History

Andy with IBM Graphics Display Unit, 1968

Source: History of Computer Security

What is Cybersecurity?

"Cybersecurity is the art of protecting networks, devices, and data from unauthorized access or criminal use and the practice of ensuring confidentiality, integrity, and availability of information."
— United States Cybersecurity & Infrastructure Security Agency

Image Sources: Flat Icon
Chat GPT’s Popularity Leveraged to Spread Malware

Threat actors using Adam Erhat, a well-known YouTuber market strategist, to earn trust and facilitate this campaign

https://google.drive.com/u0/uc...

Link to malware

How Hackers Use Data: Ransomware

“Ransomware is a type of malware that locks a victim’s data or device and requires the victim to pay a ransom to the attacker.”
— IBM

Case Study: Colonial Pipelines Ransomware Attack

Example ransom message from DarkSide, the group that hacked Colonial Pipelines
~500,000 email addresses were compromised in the 2021 cyberattack — this is the message leaked emails would receive.

Case Study: SolarWinds Cyber Attack

"As of today, 9 federal agencies and about 100 private sector companies were compromised." — Anne Neuberger, Deputy National Security Advisor

Cybersecurity + International Affairs

As cyberattacks become more common...

...cybersecurity groups work together globally!

Groups that helped neutralize the Russian malware "Snake," a cyber-espionage malware found in over 50 countries.
Future of cybersecurity

Executive Order on Improving the Nation’s Cybersecurity

Source: NYTimes, The White House

Cybersecurity at Brown

Courses at Brown:
- CSCI 1040: The Basics of Cryptographic Systems
- CSCI 1360: Humans Factors in Cybersecurity
- CSCI 1660: Introduction to Computer Security
- CSCI 1800: Cybersecurity and International Relations
- CSCI 1870: Cybersecurity Ethics
- CSCI 2660: Computer Security