CSC 172– Data Structures and Algorithms

Lecture #8

Spring 2018

Announcements

- Project 1 due tonight
- Project 2 will be out soon

Q & A

- Reminder:
  - For sharing your concern anonymously, you can always go to:
  - http://www.cs.rochester.edu/courses/172/spring2018/
  - Forms → Feedback Form

Concerns

- I see that it is possible to get extra credit on Project 1 by scheduling an appointment with a TA. However I was wondering if it would be possible to implement a similar system as I had in CSC 171 where you could explain why you deserve extra credit in your README and the TAs decide at their discretion.

Reply

- You should always explain the extra features your code provides.

- Why Demo:
  - 0: Builds rapport
  - 1. You should feel good to demo your code!
  - 2. It will boost your confidence.
  - 3. Code walkthrough would motivate you to write comments
  - 4. If you are not confident enough to demo your code, you probably do not deserve the extra credit.
  - 5. Last, but not the least, it will help us detect plagiarism

Plagiarism

Note: It's better not to submit your code rather than cheat.

Someone who cheats must get lower grade than someone who does not submit his/her work

Using code available online (with/without citation if exceeds 10% of your code) or getting code from friends will be treated as plagiarism

Concerns

- When people ask or answer questions, it's often hard to hear them because of the acoustics of the auditorium. Would you be willing to make an effort to repeat what someone has said when they ask/answer questions so that the whole class can hear it?

- Certainly I will. But please let me know immediately if I ever forget to do so.

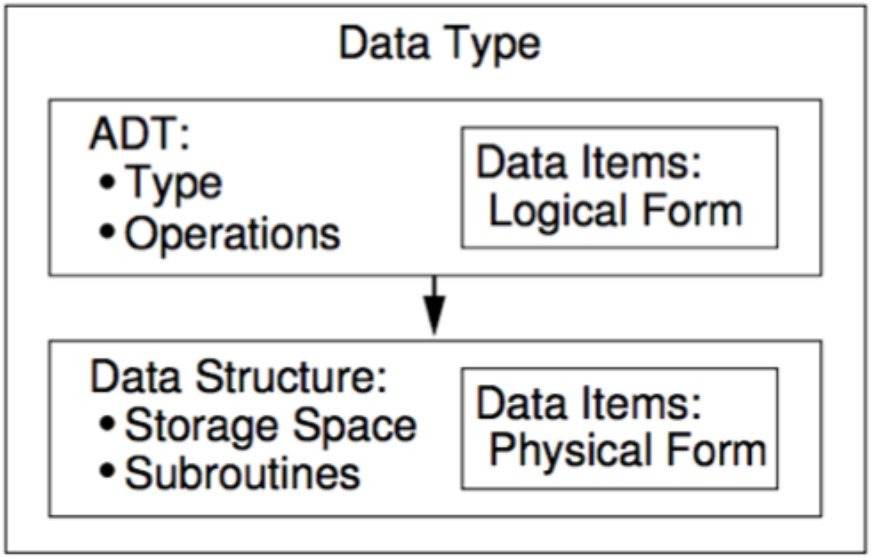
Concerns (Website on iPad)

- If you could link the mobile css on your website so that it works on iphone and ipad that would be great. The header does not load which makes it impossible to use. It is funny that website issues are only prevalent for sites for the computer science department.

- I believe we have fixed it this time! Let me know if you can access now.

Agenda

- ADT vs Data Structures



List <E> is an ADT whereas
ArrayList<E> and LinkedList<E> are Data Structures

Outline

We will now look at our first abstract data structure

- Linear ordering
- Operations
- Implementations of an abstract list with:
  - Linked lists
  - Arrays
- Memory requirements

ArrayList<E> and LinkedList<E>

How are
ArrayList<E> and LinkedList<E> fundamentally different?

Mainly due to Memory Allocation

MEMORY ALLOCATION

Outline

This topic will describe:

- The concrete data structures that can be used to store information
- The basic forms of memory allocation
  - Contiguous
  - Linked
- The prototypical examples of these:
  - ArrayLists and LinkedLists
  - Finally, we will discuss the run-time of queries and operations on ArrayLists and LinkedLists

Memory Allocation

Memory allocation can be classified as either

- Contiguous
- Linked

Prototypical examples:

- Contiguous allocation: Arrays
- Linked allocation: Linked lists

Contiguous Allocation

- An array stores n objects in a single contiguous space of memory

- Unfortunately, if more memory is required, a request for new memory usually requires copying all information into the new memory

  – In general, you cannot request for the operating system to allocate to you the next n memory locations

Linked Allocation

Linked storage such as a linked list associates two pieces of data with each item being stored:

- The object itself, and
- A reference to the next item

LIST

Definition

3.1

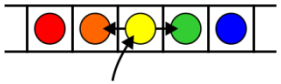
An Abstract List (or List ADT) is linearly ordered data where the programmer explicitly defines the ordering

- The most obvious implementation is to use either an array or linked list

Operations

3.1.1

Given access to the k^{th} object, gain access to either the previous or next object



Given two abstract lists, we may want to

- Concatenate the two lists
- Determine if one is a sub-list of the other

Locations and run times

We will consider the amount of time required to perform actions such as

- finding,
- inserting new entries before or after, or
- erasing entries at
  - the first location (the *front*)
  - an arbitrary (k^{th}) location
  - the last location (the *back* or n^{th})

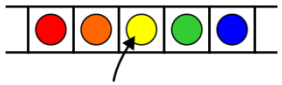
The run times will be $\Theta(1)$, $O(n)$ or $\Theta(n)$

Operations

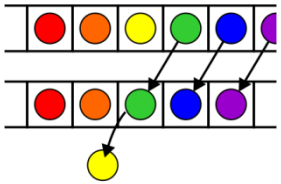
3.1.1

Operations at the k^{th} entry of the list include:

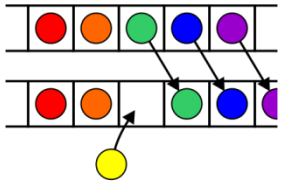
Access to the object



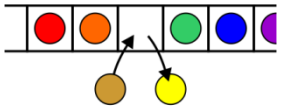
Erasing an object



Insertion of a new object



Replacement of the object



Recall: List<E>

- public interface **List<E>** extends Collection<E>

List<E> Methods

Modifier and Type	Method and Description
boolean	**add**(E e) Appends the specified element to the end of this list (optional operation).
void	**add**(int index, E element) Inserts the specified element at the specified position in this list (optional operation).
boolean	**addAll**(Collection<? extends E> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).
boolean	**addAll**(int index, Collection<? extends E> c) Inserts all of the elements in the specified collection into this list at the specified position (optional operation).
void	**clear**() Removes all of the elements from this list (optional operation).
boolean	**contains**(Object o) Returns true if this list contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns true if this list contains all of the elements of the specified collection.
boolean	**equals**(Object o) Compares the specified object with this list for equality.
E	**get**(int index) Returns the element at the specified position in this list.
int	**hashCode**() Returns the hash code value for this list.

List<E> Methods

`int`	**`indexOf(Object o)`** Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
`boolean`	**`isEmpty()`** Returns `true` if this list contains no elements.
`Iterator<E>`	**`iterator()`** Returns an iterator over the elements in this list in proper sequence.
`int`	**`lastIndexOf(Object o)`** Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
`ListIterator<E>`	**`listIterator()`** Returns a list iterator over the elements in this list (in proper sequence).
`ListIterator<E>`	**`listIterator(int index)`** Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
`E`	**`remove(int index)`** Removes the element at the specified position in this list (optional operation).
`boolean`	**`remove(Object o)`** Removes the first occurrence of the specified element from this list, if it is present (optional operation).
`boolean`	**`removeAll(Collection<?> c)`** Removes from this list all of its elements that are contained in the specified collection (optional operation).
`boolean`	**`retainAll(Collection<?> c)`** Retains only the elements in this list that are contained in the specified collection (optional operation).
`E`	**`set(int index, E element)`** Replaces the element at the specified position in this list with the specified element (optional operation).
`int`	**`size()`** Returns the number of elements in this list.
`List<E>`	**`subList(int fromIndex, int toIndex)`** Returns a view of the portion of this list between the specified `fromIndex`, inclusive, and `toIndex`, exclusive.
`Object[]`	**`toArray()`** Returns an array containing all of the elements in this list in proper sequence (from first to last element).
`<T> T[]`	**`toArray(T[] a)`** Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.

How does Java Implements List

- http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/ArrayList.java

- http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/LinkedList.java

Java ArrayList (Core Members and Constructor)

```
Object[] elementData;  // This is the core container
private int size;
private static final int DEFAULT_CAPACITY = 10;
```

```
public ArrayList(int initialCapacity) {
        if (initialCapacity > 0) {
            this.elementData = new Object[initialCapacity];
        } else if (initialCapacity == 0) {
            this.elementData = EMPTY_ELEMENTDATA;
        } else {
            throw new IllegalArgumentException("Illegal Capacity: "+
                                               initialCapacity);
        }
    }
```

How to ensure capacity?

```
private void grow(int minCapacity) {
        int oldCapacity = elementData.length;
        int newCapacity = oldCapacity + (oldCapacity >> 1);
        elementData = Arrays.copyOf(elementData, newCapacity);
    }
```

One sample function: add (E e)

```
public boolean add(E e) {
        ensureCapacityInternal(size + 1);
        elementData[size++] = e;
        return true;
    }
```

Note: You need to update size

Performance of ArrayList

- The good
  - Dynamic size
  - Adding and removing at the end takes O(1)
  - Sorting takes O(n log n)
  - Allows binary search
- The bad
  - add & remove in the middle takes O(n)
  - Requires contiguous memory block

Java LinkedList (Core Members and Constructor)

```
int size = 0;
Node<E> first;
Node<E> last;
```

```
public LinkedList() {}
```

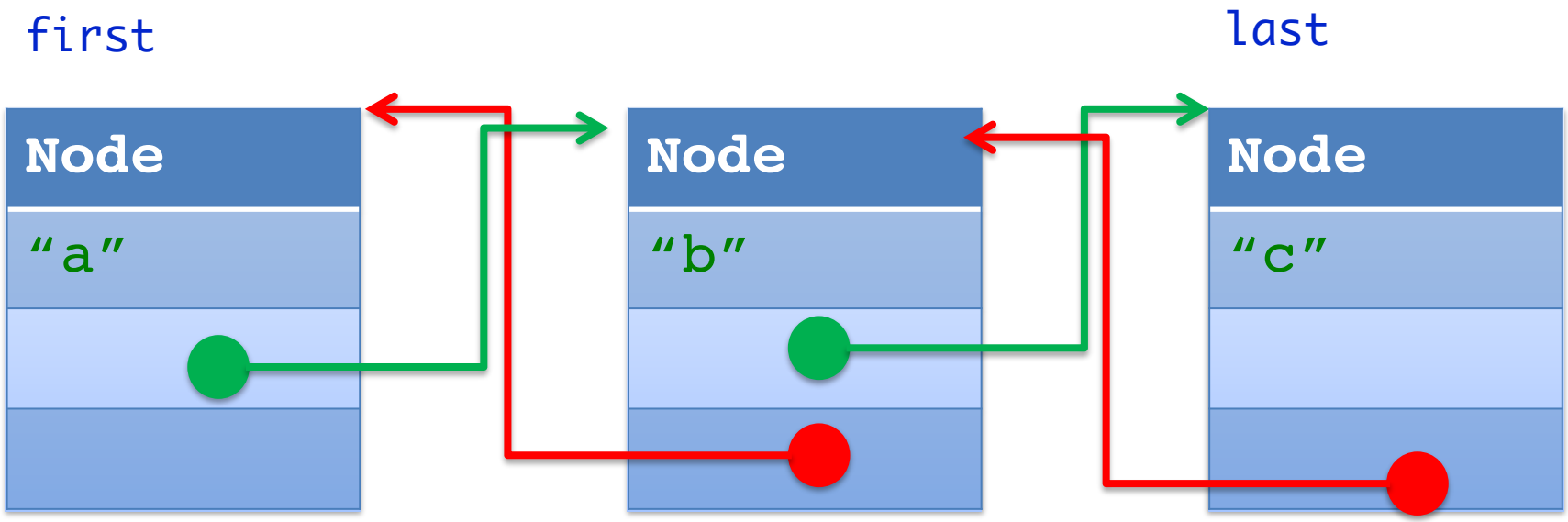
Java Node Class For LinkedList

```
private static class Node<E> {
        E item;
        Node<E> next;
        Node<E> prev;

        Node(Node<E> prev, E element, Node<E> next) {
            this.item = element;
            this.next = next;
            this.prev = prev;
        }
    }
```

Doubly Linked Lists

- Can go back and forth



One sample function: add (E e)

```
public boolean add(E e) {
        linkLast(e);
        return true;
    }
```

```
void linkLast(E e) {
        final Node<E> l = last;
        final Node<E> newNode = new Node<>(l, e, null);
        last = newNode;
        if (l == null)
            first = newNode;
        else
            l.next = newNode;
        size++;
        modCount++;
    }
```

Lab 5: Your own List

- We will give you the interface: URList
- You need to implement URArrayList and URLinkedList!

(Note: You do not have to make the classes generic. But, by this time, you know how should you proceed if you had to!)

SINGLY LINKED LIST

Linked lists

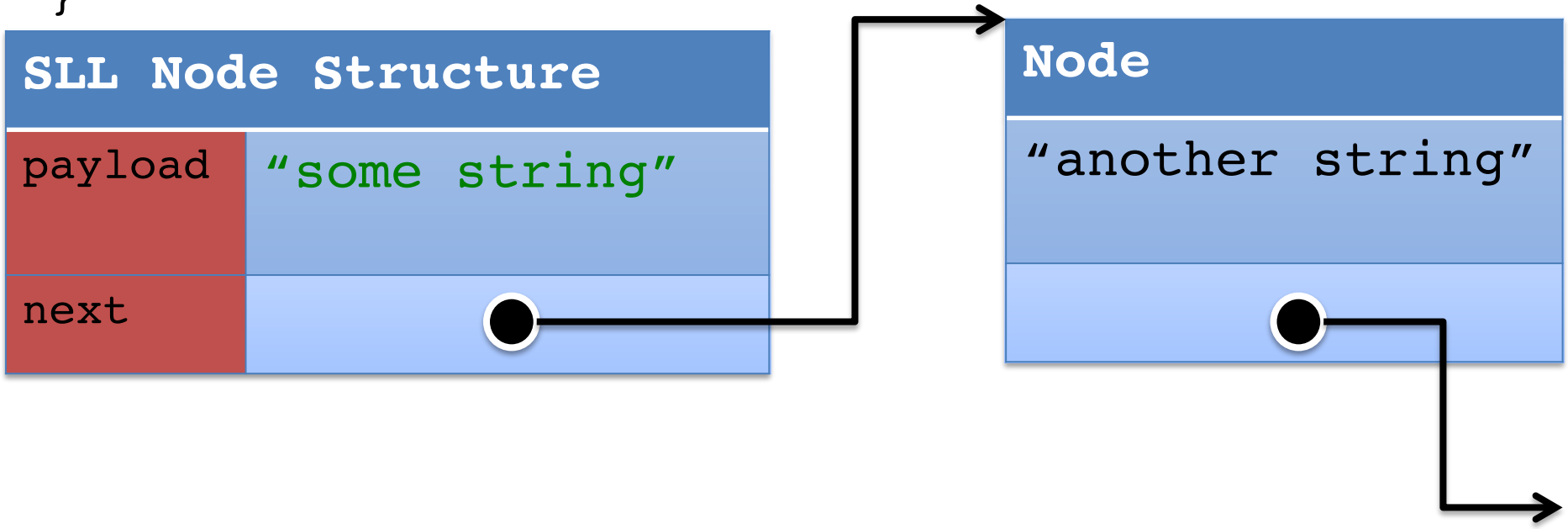
3.1.3

We will consider these for

- Singly linked lists
- Doubly linked lists

Basic Singly Linked List

```
class Node {
     Node  next;
     String payload;
     Node(String payload, Node next) {
        this.payload = payload;
        this.next = next;
    }
}
```

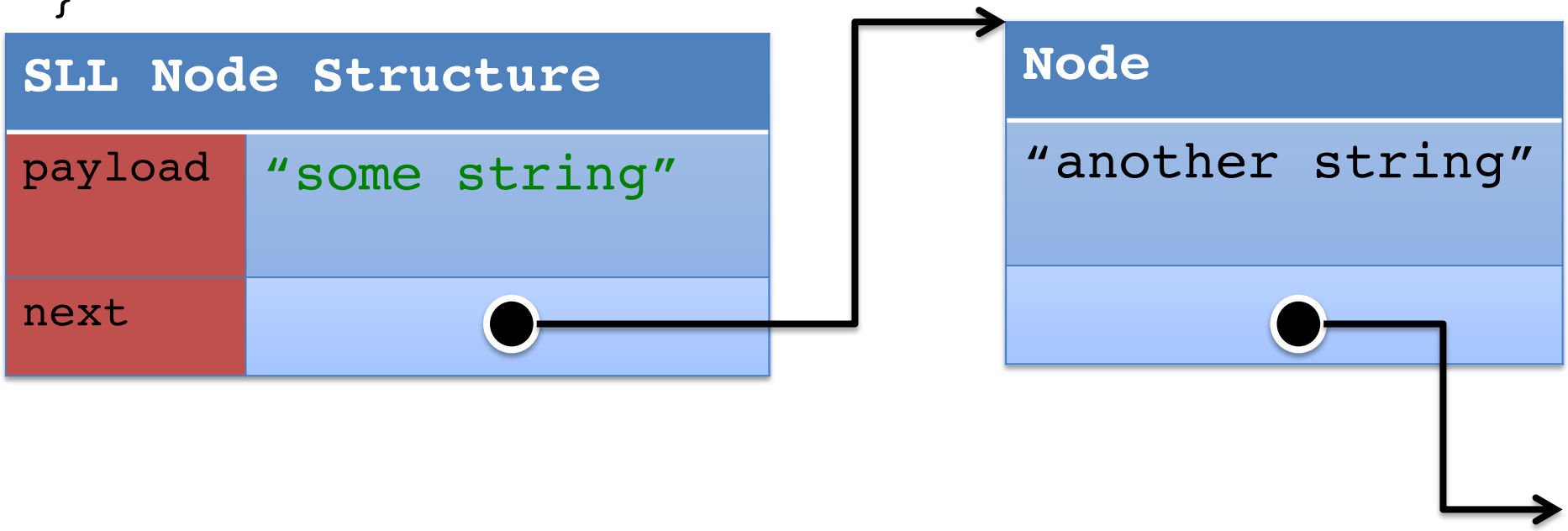


Linked lists

- Overcome the bad of ArrayLists/arrays
  - Add & remove in O(1)-time
  - Does not need contiguous block of memory
  - Can still grow/shrink dynamically
  - Can still sort in O(n log n), with care
- However
  - No random access
  - Can't do binary search even if list is sorted
  - Unless we use more advanced linked lists

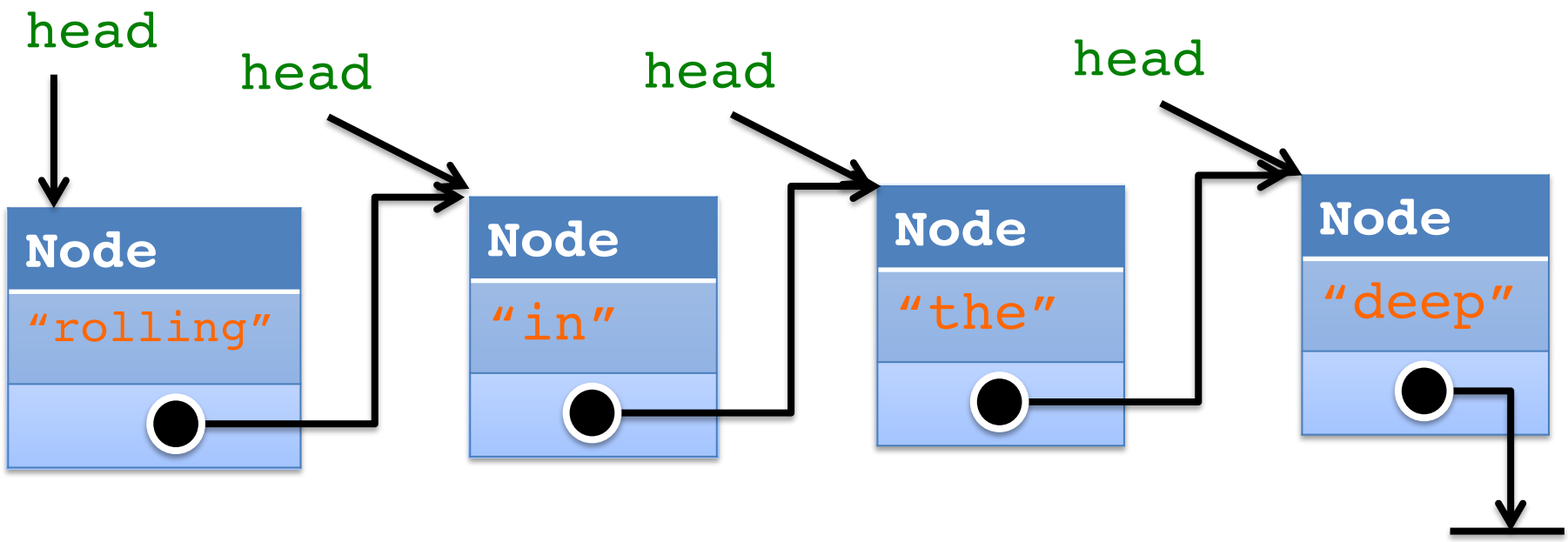
Basic Singly Linked List

```
class Node {
     Node  next;
     String payload;
     Node(String payload, Node next) {
     this.payload = payload;
     this.next = next;
    }
}
```



Constructing a SLL

```
public static void main(String[] args) {
    Node head = new Node("deep", null);
    head = new Node("the", head);
    head = new Node("in", head);
    head = new Node("rolling", head);
    print_list(head);
}
```



Traverse a SLL, Iteratively

```
void print_list(Node ptr) {
    while (ptr != NULL) {
    System.out.println (ptr.payload);
        ptr = ptr.next;
    }
}
```

Recursive version?

Traverse a SLL, Recursively

```
void print_list(Node ptr) {
     if (ptr != NULL) {
        System.out.println (ptr.payload);
        print_list(ptr.next);
     }
}
```

Linear Search

```
Node iterative_search(String key, Node ptr)
{
    while (ptr != NULL &&
!key.equals(ptr.payload))
        ptr = ptr.next;
    return ptr;
}
```

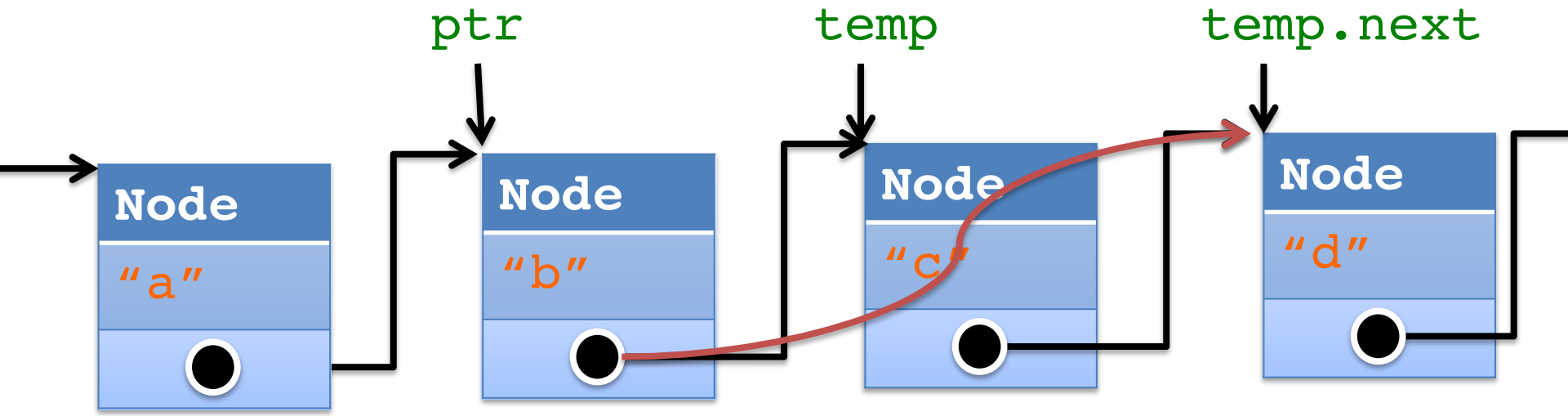
```
Node recursive_search(String key, Node ptr) {
    if (ptr != NULL && !key.equals(ptr.payload))
        return recursive_search(key, ptr.next);
     else
        return ptr;
}
```

Delete: Must Have a Predecessor Pointer

- `void del_successor(Node ptr) {`

Delete: Must Have a Predecessor Pointer

```
void del_successor(Node ptr) {
    if (ptr == null|| ptr.next == null) return;
    Node temp = ptr.next;
    ptr.next = temp.next;
}
```



Reverse a Linked List

- Given a head pointer, return the head pointer to the reversed list

```
Node reverse_sll(Node head) {
    Node prev = NULL, temp;
    while (head != NULL) {
        temp = head.next;
        head.next = prev;
        prev = head;
        head = temp;
    }
    return prev;
}
```

Insert Into a Sorted List

```
Node insert_into_sorted_list(Node head, Node node_ptr) {
    // insert in the beginning
    if (head == NULL ||
            node_ptr.payload.compareTo(head.payload)) < 0) {
        node_ptr.next = head;
        return node_ptr;
    }

    // insert in the middle, first look for spot
    Node prev = head, temp = head.next;
    while (temp != NULL &&
 temp.payload.compareTo(node_ptr.payload) < 0) {
        prev = temp;
        temp = temp.next;
    }

    prev.next     = node_ptr;
    node_ptr.next = temp;
    return head;
}
```

Properties of Singly Linked Lists

- Delete & Insert: O(1)-time if we know where
  - Especially great if we operate on the two ends
  - O(1)-time for stack & queue operations
- Search: O(n)-time even if list already sorted
- Waste O(n)-space for all the pointers
  - However, this O(n) is only on the pointers!
- What about sorting?
  - Insertion sort: $O(n^2)$
  - Merge sort: $O(n \log n)$

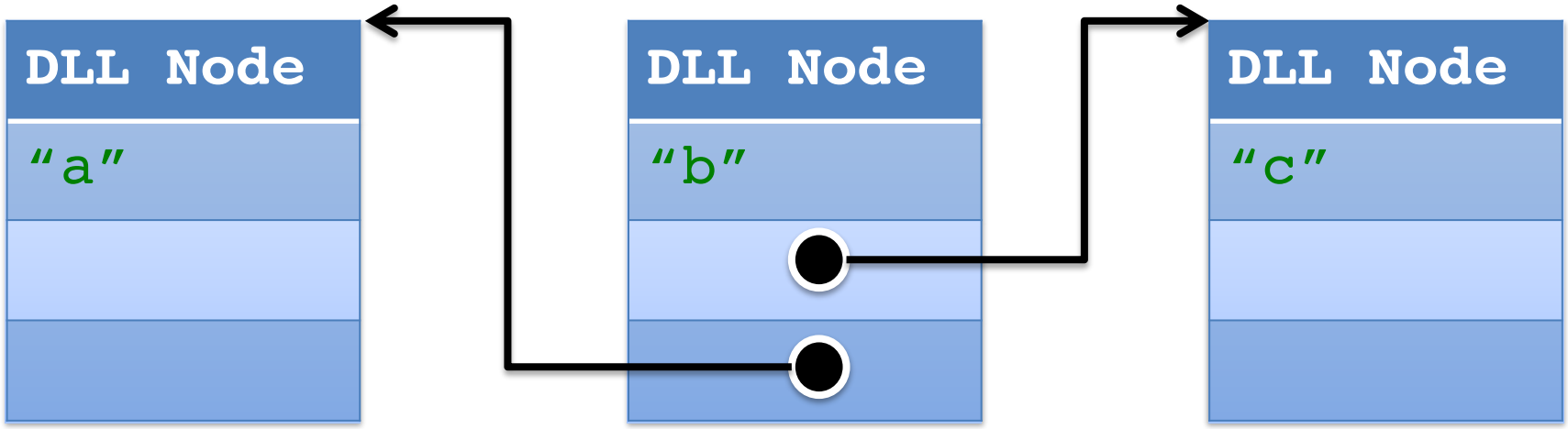
Properties of Singly Linked Lists

- Many other types of computation can be done iteratively (or recursively sometimes)
  - Count # of members in the list
  - Remove duplicate elements
  - Swap 2 sub-blocks of two lists
  - Remove elements of a given key
  - Etc.

- Can't go backward

Doubly Linked Lists

- Can go back and forth
- Waste one extra pointer per element



Recall: One sample function: add (E e)

```
public boolean add(E e) {
        linkLast(e);
        return true;
    }
```

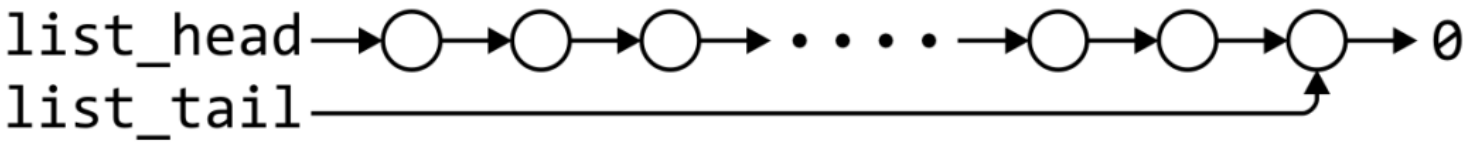
```
void linkLast(E e) {
        final Node<E> l = last;
        final Node<E> newNode = new Node<>(l, e, null);
        last = newNode;
        if (l == null)
            first = newNode;
        else
            l.next = newNode;
        size++;
        modCount++;
    }
```

Singly linked list

3.1.3.1

	Front/1^{st} node	k^{th} node	Back/n^{th} node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$O(n)$	$\Theta(n)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$O(n)$	$\Theta(n)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$O(n)$	$\Theta(n)$

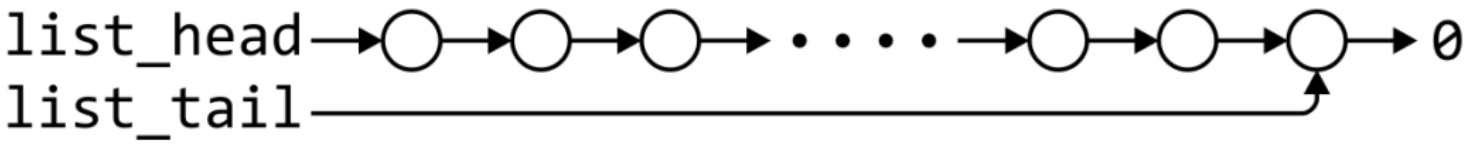
* These assume we have already accessed the k^{th} entry—an $O(n)$ operation



Singly linked list

	Front/1^{st} node	k^{th} node	Back/n^{th} node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(1)^*$	$\Theta(n)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$O(n)$	$\Theta(n)$

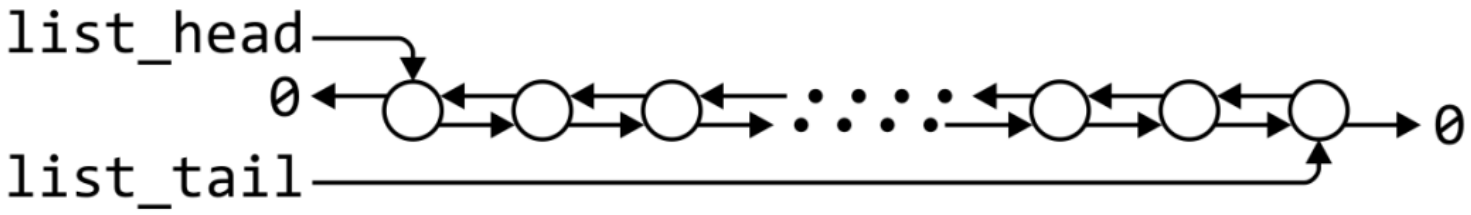
By replacing the value in the node in question, we can speed things up



Doubly linked lists

	Front/1^{st} node	k^{th} node	Back/n^{th} node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$\Theta(1)^*$	$\Theta(1)$

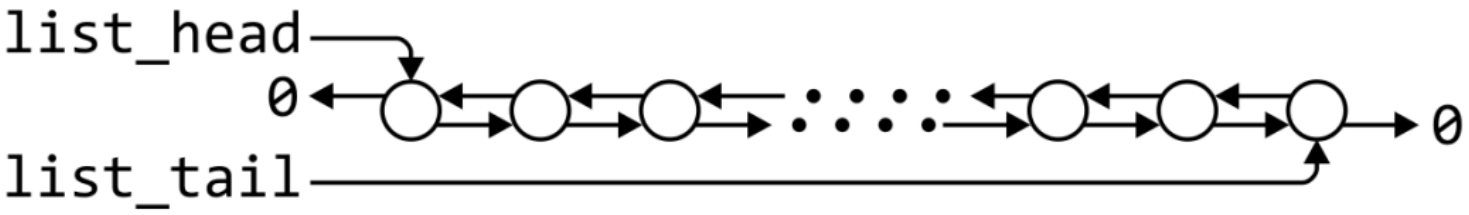
* These assume we have already accessed the k^{th} entry—an $O(n)$ operation



Doubly linked lists

Accessing the k^{th} entry is $O(n)$

	k^{th} node
Insert Before	$\Theta(1)$
Insert After	$\Theta(1)$
Replace	$\Theta(1)$
Erase	$\Theta(1)$
Next	$\Theta(1)$
Previous	$\Theta(1)$



Other operations on linked lists

Other operations on linked lists include:

- Allocation the memory requires $\Theta(n)$ time
- Concatenating two linked lists can be done in $\Theta(1)$
  - This requires a tail pointer

Main Problem with Linked List

Operation	Time
Search	O(n)
Insert	Search + O(1)
Delete	Search + O(1)
Insert front/back	O(1)
Delete front/back	O(1)

Acknowledgement

- Douglas Wilhelm Harder.
  - Thanks for making an excellent set of slides for ECE 250 *Algorithms and Data Structures* course

- Prof. Hung Q. Ngo:
  - Thanks for those beautiful slides created for CSC 250 (Data Structures) course at UB.

- Many of these slides are taken from these two sources.