CSC 172– Data Structures and Algorithms

Lecture #9

Spring 2018

SINGLY LINKED LIST

Linked lists

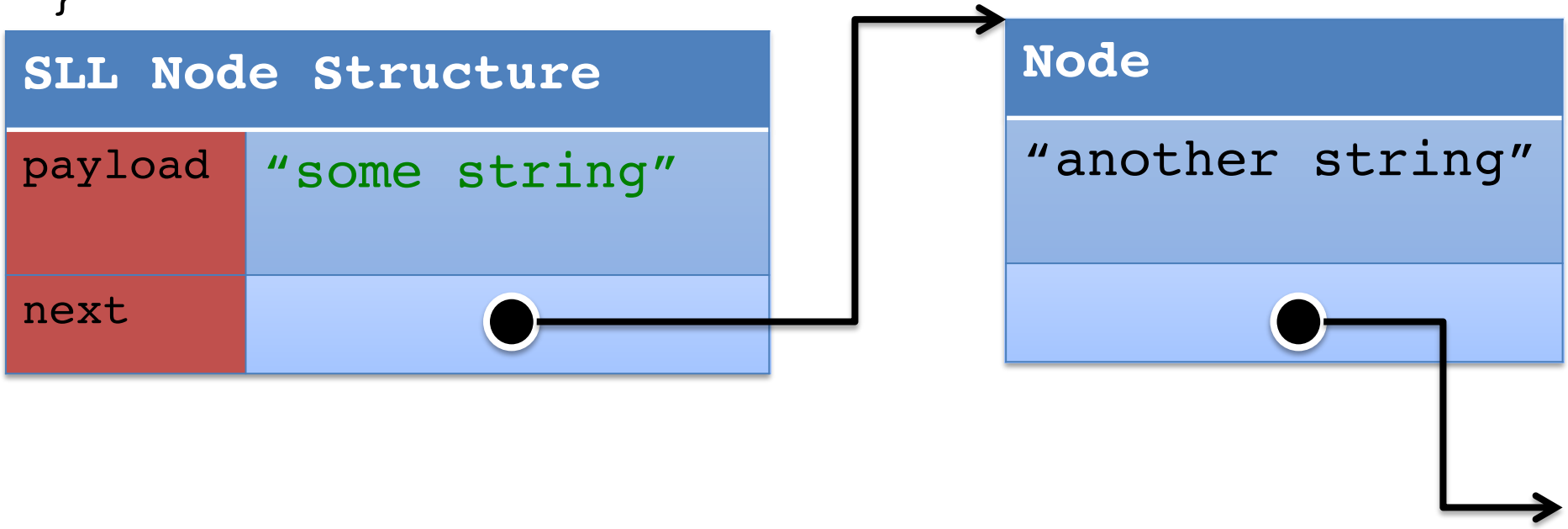
3.1.3

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;
    }
}
```

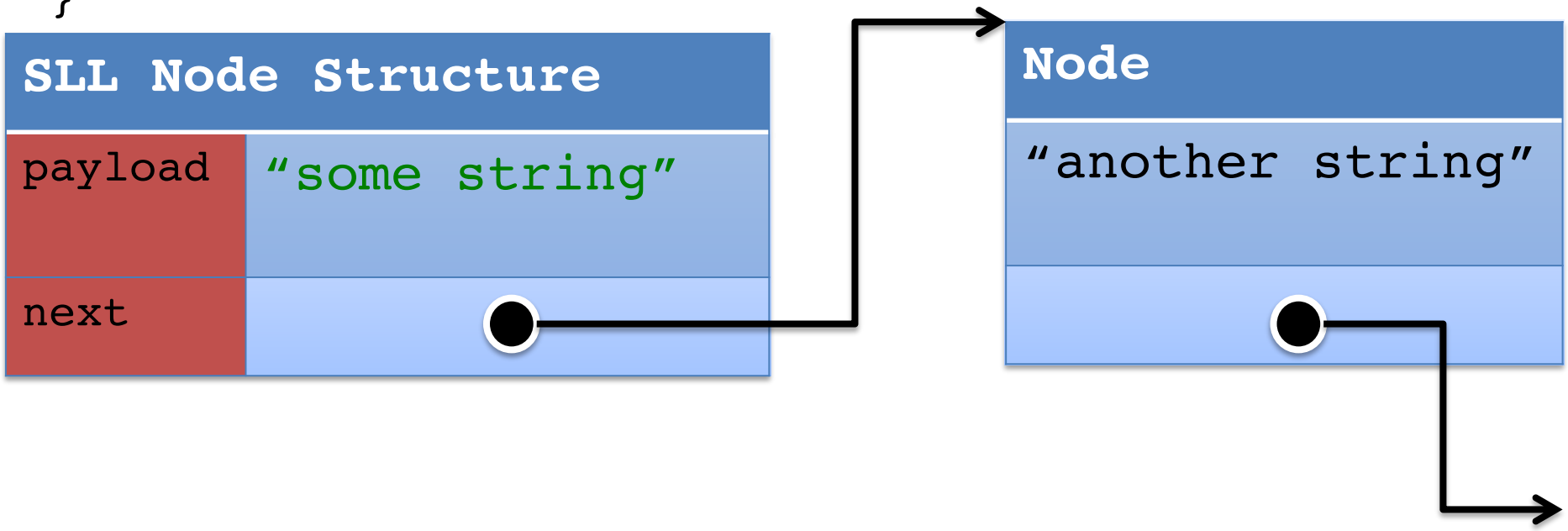


Linked lists

- Overcome the cons 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

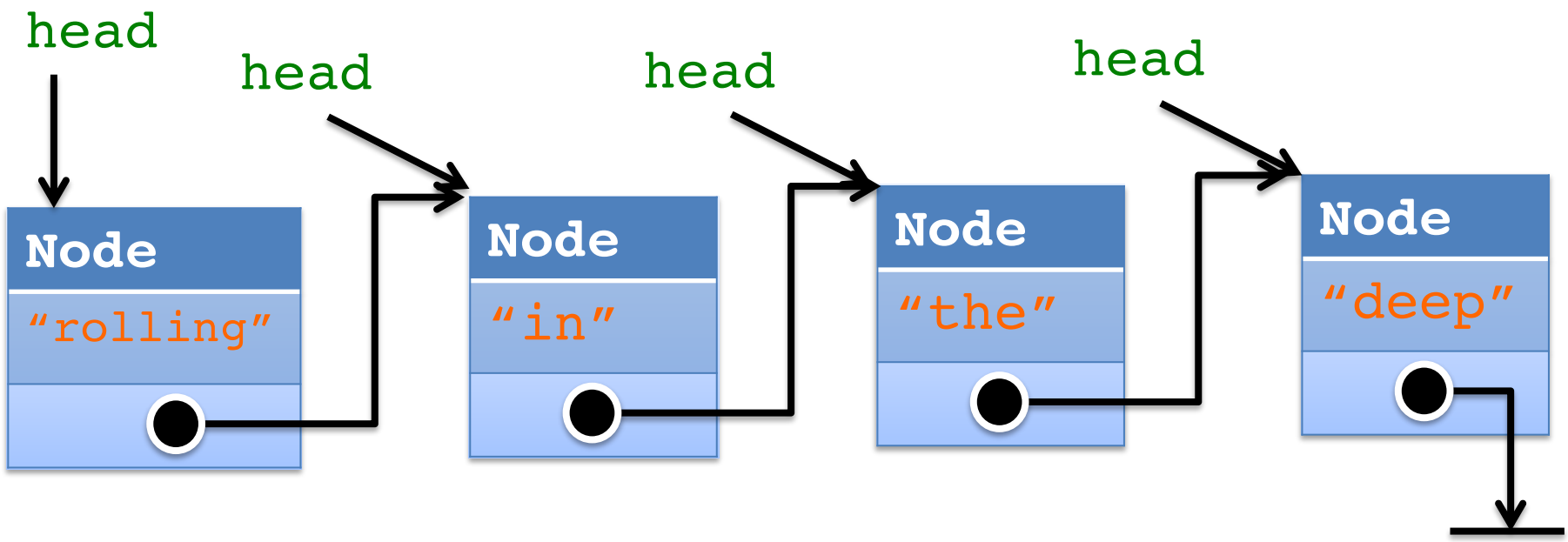
Basic Singly Linked List

```
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;
}
```

Linear Search (Recursive Function)

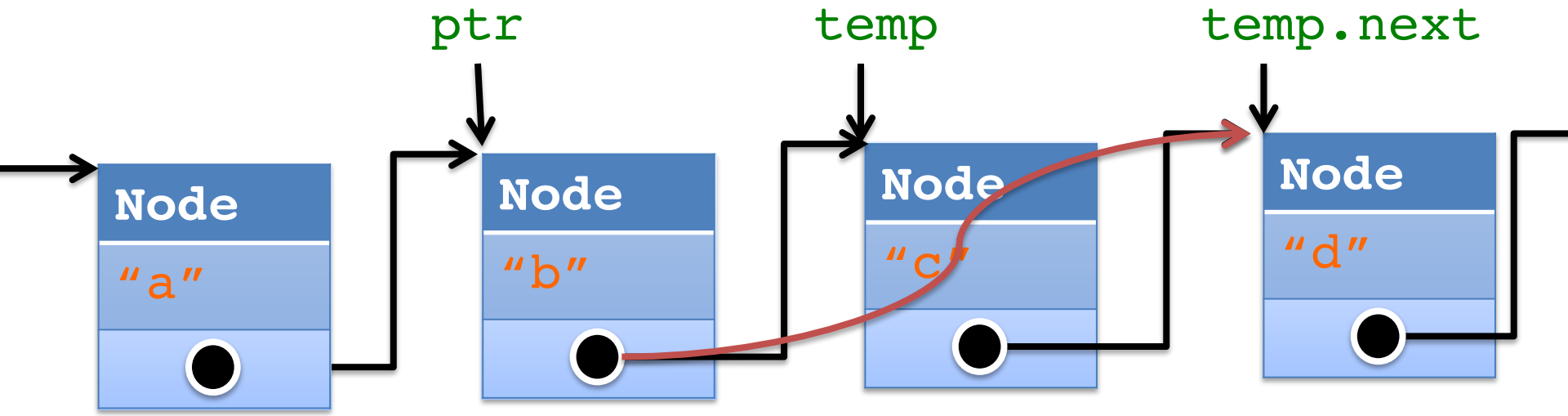
```
Node recursive_search(String key, Node ptr) {
    if (ptr != null && !key.equals(ptr.payload))
        return recursive_search(key, ptr.next);
     else
        return ptr;
}
```

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: $O(n \log n)$

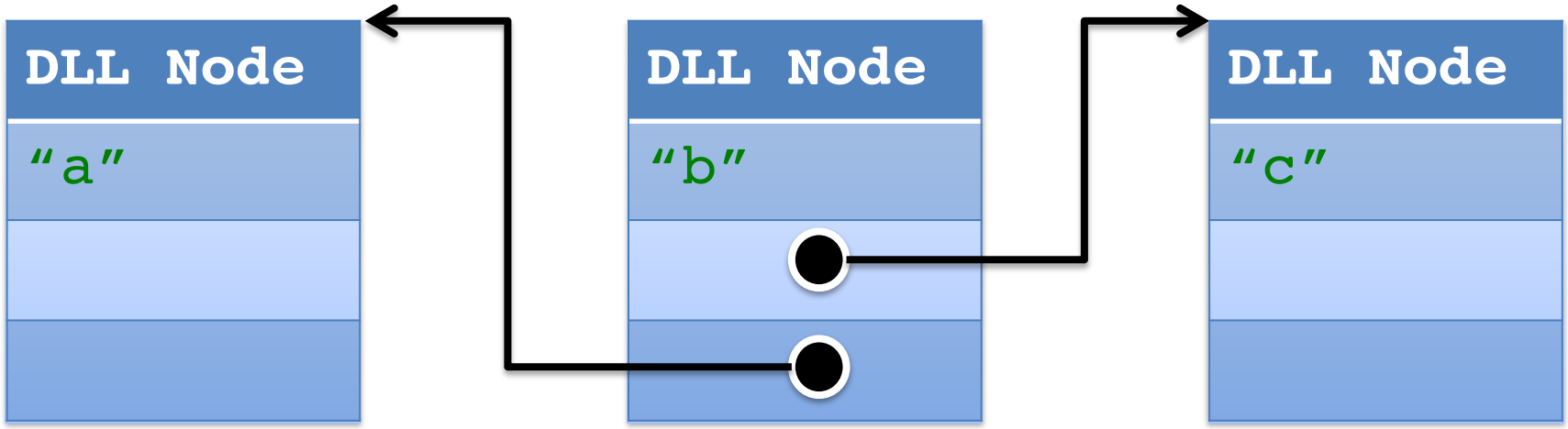
Properties of Singly Linked Lists

- 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++;
    }
```

Summary

- ArrayList
  - Random access
  - Contiguous Memory
  - Insertion and Deletion at the beginning takes time
- LinkedList
  - Two varieties:
    - Singly LinkedList
    - Doubly LinkedList
  - No Random access (sequential access only)
  - Requires more memory to store

ArrayList

3.1.3.1

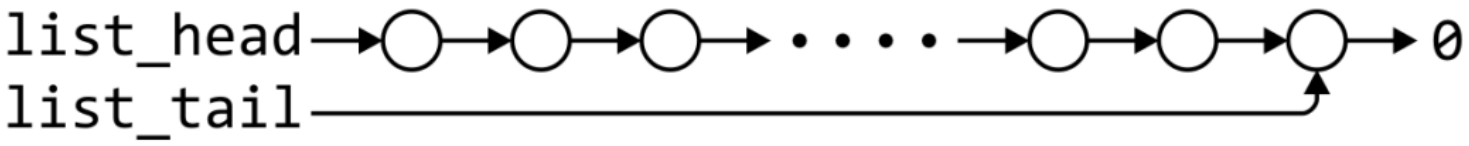
	Front/1^{st} elem	k^{th} elem	Back/n^{th} elem
Find	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Insert Before	$O(n)$	$O(n)$	$\Theta(1)$
Insert After	$O(n)$	$O(n)$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Erase	$O(n)$	$O(n)$	$\Theta(1)$
Next	$\Theta(1)$	$\Theta(1)$	n/a
Previous	$\Theta(1)$	$\Theta(1)$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

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	$\Theta(1)$	$O(n)$	$O(n)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$O(n)$	$O(n)$

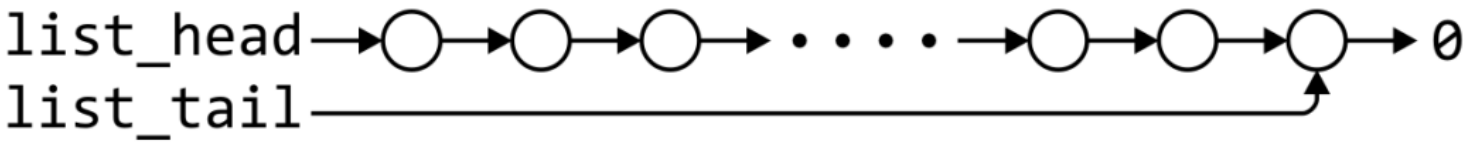
* These assume we have already accessed the k^{th} entry—an $O(n)$ operation



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	$O(n)$	$\Theta(n)$

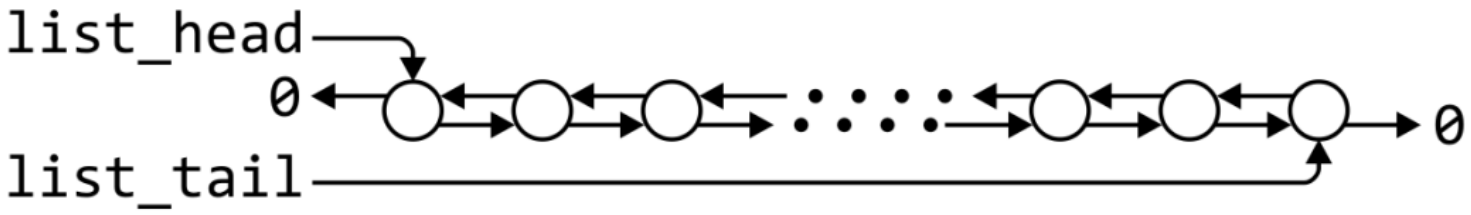
By replacing the value in the node in question, we can speed things up



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	$\Theta(1)^*$	$\Theta(1)$

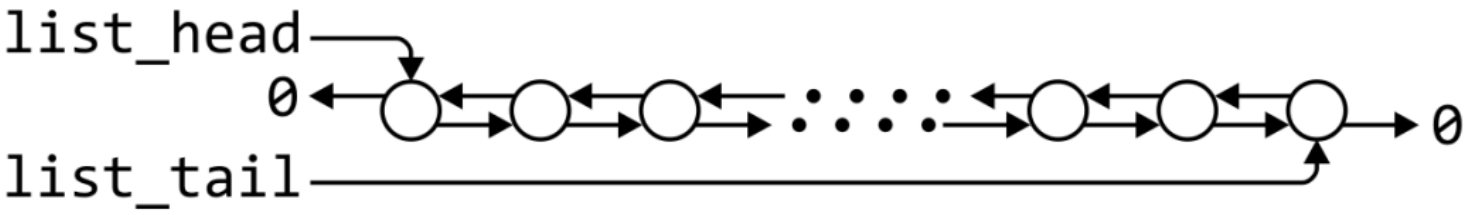
* These assume we have already accessed the k^{th} entry—an $O(n)$ operation



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

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

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)

Lab 5: Your own List

- We will give you the interface: URList

- You need to implement URArrayList and URLinkedList!

(Note: You NEED TO make the classes generic.)

ITERATOR (BONUS SLIDES)

Iterator

- Lab 3:
  - Different ways of traversing an ArrayList
  - One option was using an Iterator

Iterator Example

```
Integer[] inputArray = {2, 3, 5, 6, 7, 12, 15, 16};

ArrayList<Integer> arrayList =
new ArrayList<Integer>(Arrays.asList(inputArray));

Iterator<Integer> arrayListIterator = arrayList.iterator();

while (arrayListIterator.hasNext()) {
                System.out.println("Next number = " +
            arrayListIterator.next());
}
```

```
Next number = 2
Next number = 3
Next number = 5
Next number = 6
Next number = 7
Next number = 12
Next number = 15
Next number = 16
```

Custom Iterator

- Dummy problem:
- What if I have a class which contains an array of Integers
- I want to create an iterator which iterates through only the even integers contained in an object of this class.

The object contains {2, 3, 5, 6, 7, 12, 15, 16};

The iterator should iterate through {2, 6, 12, 16};

public interface Iterable<T>

- Implementing this interface allows using iterators.

- You must implement the method iterator() which returns an Iterator

Iterator<T>	iterator() Returns an iterator over elements of type T.

What is an Iterator

- An iterator over a collection
  - The name says it all!
  - All class implementing an Iterator interface must implement the following two methods:

- **hasNext()** --- Returns Boolean (true if the iteration has more elements; false otherwise)
- **next()** --- Returns T (the next element in the iteration)

Another optional method: **remove()**
Removes from the underlying collection the last element returned by this iterator

Time to Implement!

```
public class LearnIterator implements Iterable<Integer> {...}
```

```
private Integer[] intArray;

LearnIterator(Integer[] inputArray){
    intArray = inputArray.clone();
}
```

main Method contains:

```
Integer[] inputArray = {2, 3, 5, 6, 7, 12, 15, 16};
LearnIterator testItr = new LearnIterator(inputArray);
```

Code vs It's output

How I want to use Iterator

```
Integer[] inputArray = {2, 3, 5, 6, 7, 12, 15, 16};
LearnIterator testItr = new LearnIterator(inputArray);
Iterator<Integer> evenIterator = testItr.iterator();

while (evenIterator.hasNext()) {
    System.out.println("Next Even number="+ evenIterator.next());
}
```

Expected output:

```
Next Even number= 2
Next Even number= 6
Next Even number= 12
Next Even number= 16
```

Class LearnIterator implements

```
@Override
    public Iterator<Integer> iterator() {
        return new CustomEvenIterator();
    };
```

```
private class CustomEvenIterator implements Iterator<Integer> {

        private int currentIndex;

        CustomEvenIterator() {
             currentIndex = 0;
        }

        @Override
        public boolean hasNext() {
             if (currentIndex >= intArray.length) return false;
           else {
                 for ( int i = currentIndex; i < intArray.length; i++) {
                      if (intArray[i] % 2 == 0) {
                           return true;
                      }
                 }
           }
             return false;
        }

       @Override
       public Integer next() {
             if (!hasNext())
               throw new NoSuchElementException();

             while(intArray[currentIndex] % 2 != 0) {
                  currentIndex++;
             }
             return intArray[currentIndex++];
       }

       @Override
       public void remove() {
           throw new UnsupportedOperationException();
       }
   }
```

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.