Topic

- Binary Search
- Reference-Based Implementation
Sorted List

- In the case of an array-based sorted list, a more efficient approach is available for searching the list: the binary search algorithm.

- Normal way: start at the beginning and compare the target element to each element in the list sequentially. (find)
Improving Linear Search

• Simply stop searching when we find an element larger than the target element.
  • Check the first element.
  • If it matches, we are done; otherwise proceed with the next element.
  • If we exhaust the list or reach a value greater than the desired, the desired value not present.
Binary Search Algorithm

1. Check the middle element.

2. If it matches, we are done.

3. If it is less than the target, search only the upper half of the list; otherwise, search only the lower half.
Recursive Impl

protected boolean find(T target) {
    return find1(target, 0, numElements-1);
}

private boolean find1(T target, int lo, int hi) {
    if (lo > hi)
        return false;
    int location = (lo + hi) / 2;
    if (list[location].compareTo(target) == 0)
        return true;
    if (list[location].compareTo(target) < 0)
        return find1(target, location+1, hi);
    return find1(target, lo, location-1);
Tail Recursion

- A function call is said to be tail recursive if there is nothing to do after the function returns except return its value.

- Since the current recursive instance is done executing at that point, saving its stack frame is a waste. Specifically, creating a new stack frame on top of the current, finished, frame is a waste.

- A compiler is said to implement Tail Recursion if it recognizes this case and replaces the caller in place with the callee, so that instead of nesting the stack deeper, the current stack frame is reused.
Tail VS Not Tail

factorial(n) {  
  if (n == 0) return 1;  
  return n * factorial(n - 1);  
}

This definition is not tail-recursive since the recursive call to factorial is not the last thing in the function.

factorial1(n, accumulator) {  
  if (n == 0) return accumulator;  
  return factorial1(n - 1, n * accumulator);  
}

factorial(n) {  
  return factorial1(n, 1);  
}
protected boolean find(T target) {
    int lo=0, hi=numElements-1;
    while (lo <= hi) {
        location = (lo + hi) / 2;
        if (list[location].compareTo(target) == 0)
            return true;
        if (list[location].compareTo(target))
            lo=location+1;
        else
            hi=location-1;
    }
    return false;
}

tail recursive methods can be converted to iterative form fairly easily.
Complexity

- Linear search: $O(N)$
- Binary search: $O(\log N)$
package ch06.lists;

import support.LLNode;

public class RefUnsortedList<T extends Comparable<T>> implements ListInterface<T> {
    protected int numElements = 0;
    protected LLNode<T> currentPos = null;
    protected LLNode<T> location;
    protected LLNode<T> previous;
    protected LLNode<T> list = null;
RefUnsortedList: Add

```java
public void add(T element) {
    LLNode<T> newNode = new LLNode<T>(element);
    newNode.setLink(list);  // new becomes first
    list = newNode;
    numElements++;
}
```

Since the list is not sorted we can add elements wherever it is most convenient and for a single linked list, that is at the front (i.e., right after the RefUnsortedList<T> node itself).
RefUnsortedList: Size

- Since we explicitly maintain a count of the number of elements currently residing on the list, size() is trivial.

- public int size() { return numElements; }
RefUnsortedList: Contains

public boolean contains(T element) {
    return find(element);
}

protected boolean find(T target) {
    location = list;
    while (location != null)
        if (location.getInfo().equals(target))
            return true
    previous = location;
    location = location.getLink();
    return false;
}
RefUnsortedList: Remove

public boolean remove(T element) {
    boolean found = find(element);
    if (found) {
        if (location = list) // first element
            list = location.getLink();
        else
            previous.setLink(location.getLink());
        numElements--;
    }
    return found;
}
RefUnsortedList: get()

public T get(T element) {
    if find(element)
        return location.getInfo();
    else
        return null;
}
RefUnsortedList: reset() / getNext() 

```java
public void reset() { currentPos = list; }

public T getNext() {
    T next = currentPos.getInfo();
    if (currentPos.getLink() == null)
        currentPos = list;
    else
        currentPos = currentPos.getLink();
    return next;
}
	// end of class RefUnsortedList<T>
```

1. The list is not empty.
2. The list has been reset().
3. This has not been modified since the most recent reset.
RefSortedList (RefsSortedList<T>)

- Take advantage of similarities between the unsorted and sorted lists by having RefSortedList extend RefUnsortedList
- The only method we need to override is the add method.
- Sorted List: The new add method must insert an element into the correct position on the list
public void add(T element) {
    LLNode<T> prevLoc = null;
    LLNode<T> location = list;
    while (location!=null &&
        location.getInfo().compareTo(element)<0) {
        prevLoc = location;
        location = location.getLink();
    }
    LLNode<T> newNode = new LLNoDE<T>(element);
    if (prevLoc == null) {   // add at front of list
        newNode.setLink(list);
        list = newNode;
    } else {                 // add elsewhere
        newNode.setLink(location);
        prevLoc.setLink(newNode);
    }
    numElements++;
}
} // end of class RefSortedList<T>