CS151 Intro to Data Structures

Iterators
Recursion

Binary Search

Announcements

- Lab5 and HW3 due Friday!
  - Stacks and queues

- Midterm after break

- If you're missing some assignments, chance to submit them over fall break...

Outline

- Iterators
- Recursion Review
- Binary Search

Iterators

- represents a sequence of elements and provides a way to iterate, or traverse, through those elements one at a time

Iterators

- Abstracts the process of scanning through a sequence of elements (traversal)
- provides a way to iterate, or traverse, through elements one at a time

hasNext( ):	Returns true if there is at least one additional element in the sequence, and false otherwise.
next( ):	Returns the next element in the sequence.

- Combination of these two methods allow a generic traversal structure

```
while(iter.hasNext()) {
  iter.next();
}
```

Iterators

- **code**

- Can an iterator go backwards? NO. Only can do `next()`

`Iterable` Interface

- What can i use an `iterator` on? Anything that implements the `iterable` interface.
- Each call to `iterator()` returns a new iterator instance, thereby allowing traversals of a collection
- `List` interface extends `Iterable` and `ArrayList` implements `List`

`Iterable` Interface

An interface with a single method:

- `iterator()`: returns an iterator of the elements in the collection

Iterator Interface

`Iterator` Interface

Another interface that supports iteration

- `boolean hasNext()`
- `E next()`
- `void remove()`

- `Scanner` implements `Iterator<String>`
- `ArrayList` inner class `ArrayListIterator` implements `Iterator`

Let’s make ExpandableArray iterable

`Iterable` versus `Iterator`?

- `Iterable`
  - `java.lang`
  - override `iterator()`
  - Doesn't store the iteration state
  - Removing elements during iteration isn't allowed

- `Iterator`
  - `java.util`
  - Override `hasNext()`, `next()`
  - Optional `remove()`
  - Stores iteration state (list cursor)
  - Removing elements during iteration supported

Outline

- Runtime
- **Recursion**
- Binary Search

Recursive functions – base case

Conditional statement that prevents infinite repetitions

Usually handles cases where:

- input is empty
- problem is at its smallest size

Recursion Example - Factorial

- What is a factorial? n!
- product of all integers less than or equal to n
  - n! = n * n-1 * n-2 ….. 1
  - 5! = 5 * 4 * 3 * 2 * 1
    - 5! = 5 * 4!
  - 4! = 4 * 3 * 2 * 1
  - 3! = 3 * 2 * 1
  - 2! = 2 * 1
  - 1! = 1

Visualizing recursion – Factorial example

factorial(5) =

= 5 * factorial(4)

= 5 * 4 * factorial(3)

= 5 * 4 * 3 * factorial(2)

= 5 * 4 * 3 * 2 * factorial(1)

= 5 * 4 * 3 * 2 * 1

Recursion Example – Contains letter

Write a method called “containsLetter” that determines if a String contains a given character

Question: What are the parameters?

1. The character to look for
2. The string to be looking in

Question: What is the return type?

Code it!

Recursion Visualization – Contains letter

contains(“l”, “apple”) =

contains(“l”, “apple”)

contains(“l”, “pple”)

contains(“l”, “ple”)

contains(“l”, “le”)

return true

Recursive Method

Break problem down into smaller subproblem that we can repeat

Base case(s):

- no recursive calls are performed
- every chain of recursive calls must reach a base case eventually

Recursive calls:

- Calls to the same method in a way that progress is made towards a base case
- Often called “the rule”

Outline

- Runtime
- Recursion
- **Binary Search**

Binary Search

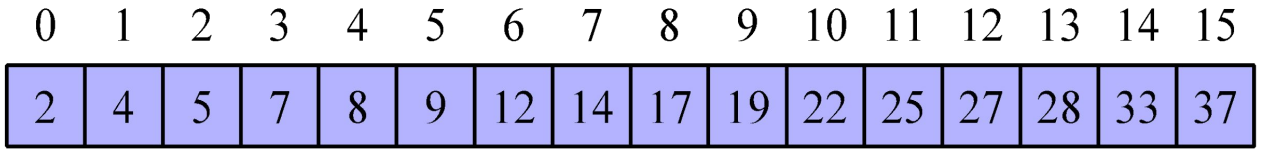
- efficient search in a sorted list
- can be implemented **recursively**

Search steps:

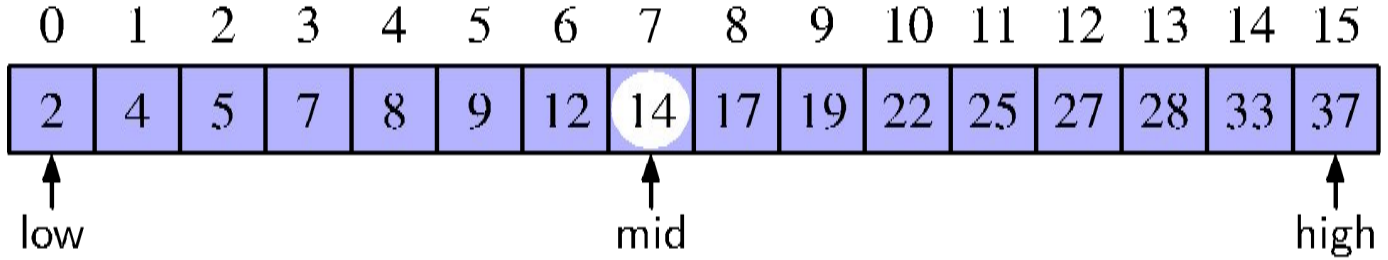
1. Calculate midpoint
2. Compare the value at the midpoint with the target value
   a. if equal:
      i. return index
   b. if target value < midpoint value:
      i. **search** the left portion of the list
   c. if target value > midpoint value:
      i. **search** the right portion of the list

Binary Search

Search for an integer (22) in an ordered list

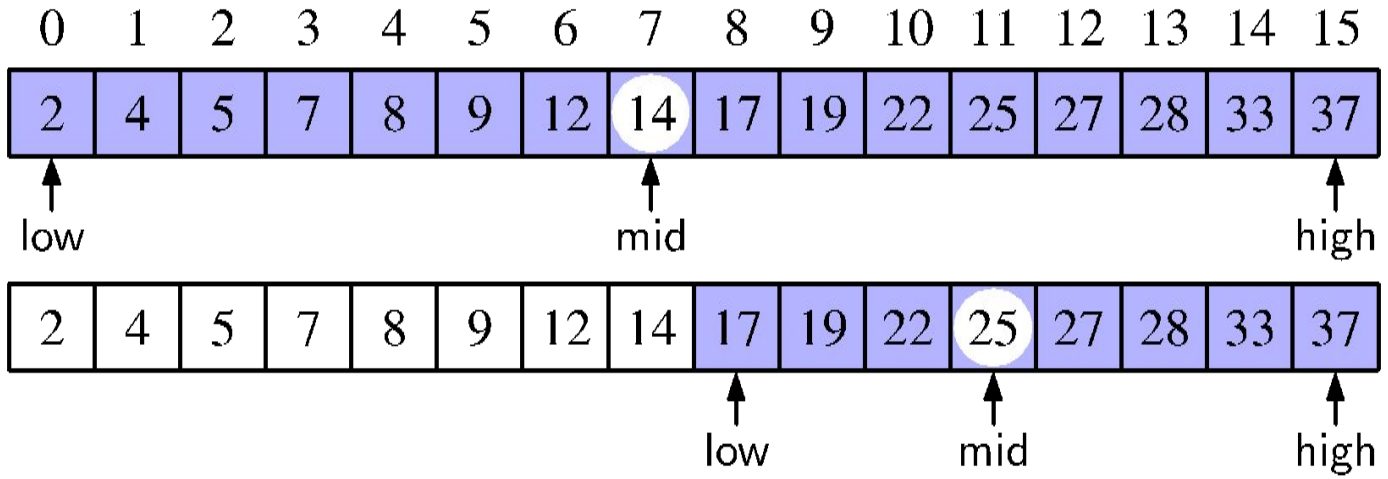


target = 22



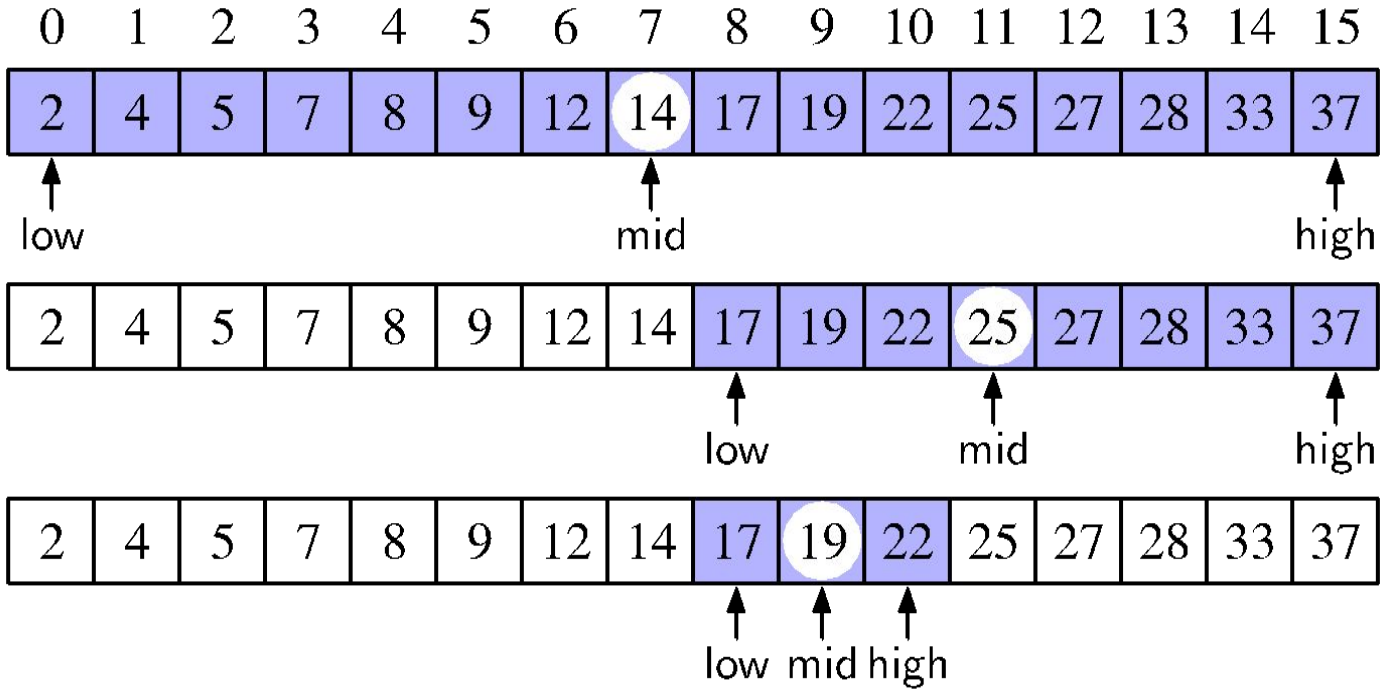

```
int mid = (high + low) / 2; //index
int midVal = arr[mid]; //14
```

target = 22

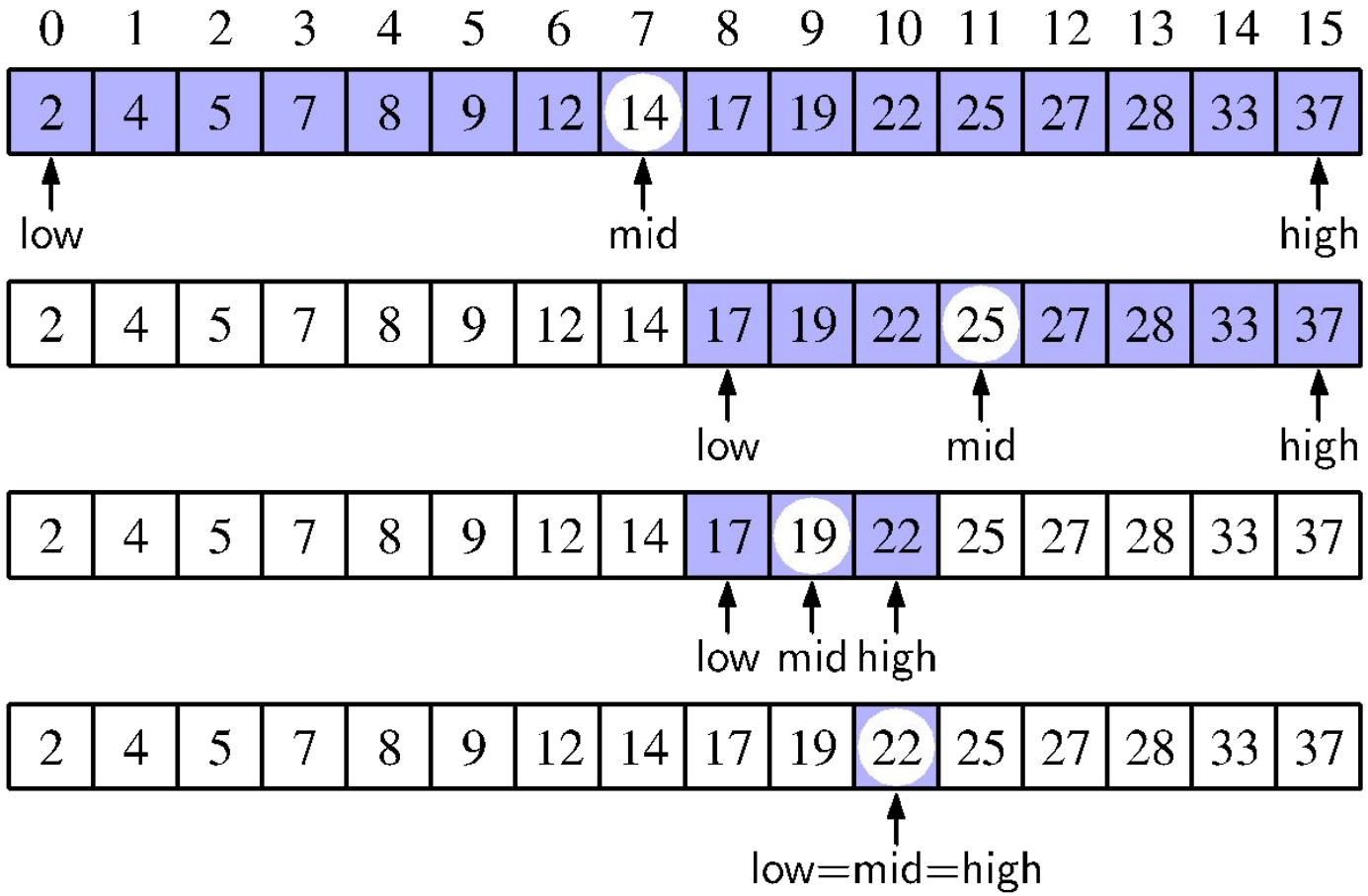


low = mid+1;

target = 22


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low
mid
high
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low
mid
high
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low mid high

target = 22


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low
mid
high
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low
mid
high
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low mid high
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low=mid=high

Binary Search Implementation

Binary Search Analysis

- Each recursive call divides the array in half

If the array is of size n, it divides (and searches) at most $logn$ times before the current half is of size 1

$O(logn)$

Comparable

Binary search on a list of objects requires that the objects have natural ordering

In other words, the objects must implement `Comparable`

`compareTo`

Summary

- iterators
  - What is the **Iterable** interface? iterator()
  - What is the **Iterator** interface?
    - next(), hasNext()

- Binary Search
  - runtime complexity? O(logn)
  - more, less, or equal efficiency to a linear search?
    - O(n)
    - O(logn)