Data Structures and Algorithms

Week 2

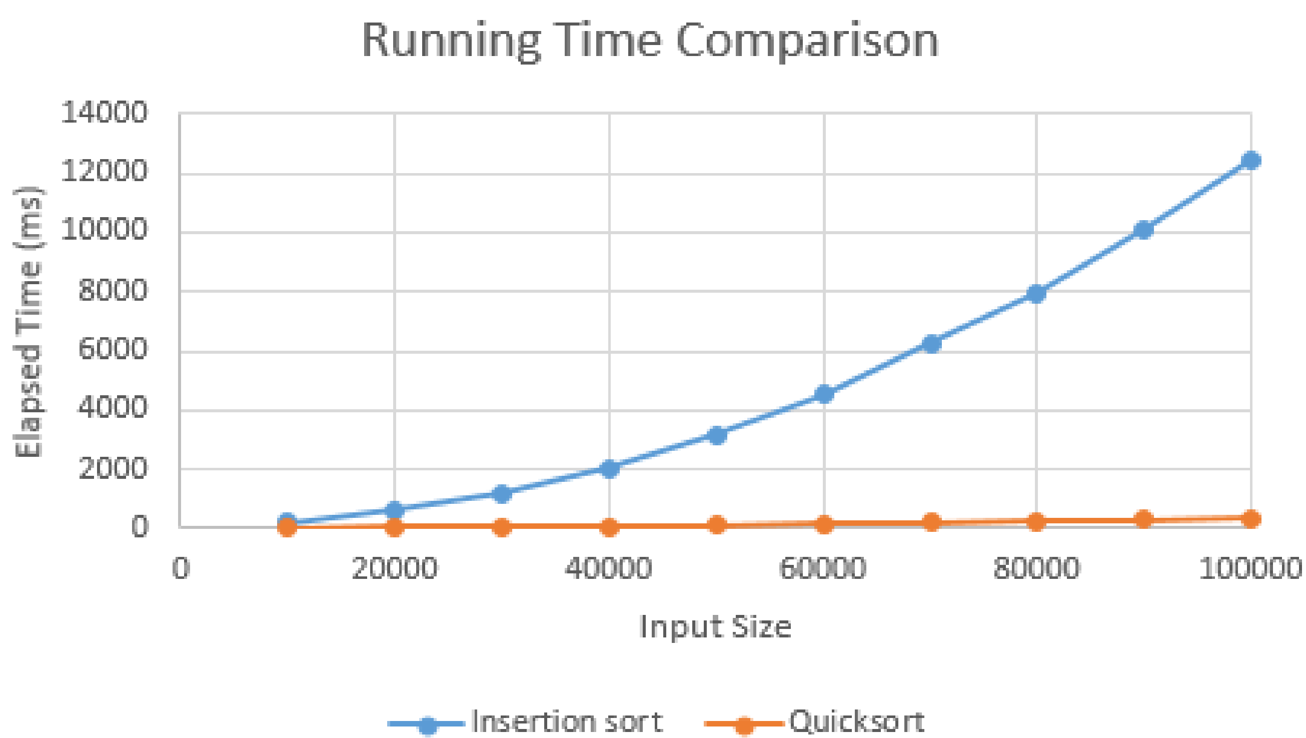
Algorithm Analysis
Basic Concepts

- An algorithm is a finite sequence of steps which solves a problem.
- Efficiency of algorithms can be analyzed in terms of memory/space usage and in terms of running time.
- We will focus on running time analysis.

- Running time of an algorithm depends on the input size.
- We express running time as a function of the input size n.

Algorithm Analysis
Basic Concepts

- Running times of two sorting algorithms



Algorithm Analysis
Basic Concepts

- Running times of an algorithm may be different for different inputs of the same size.
- For example, elapsed times of insertion sort algorithm on an array of 100,000 integers:
  - Best case: 1 ms, when elements are sorted in nondecreasing order
  - Average case: 12,145 ms, when elements are randomly distributed
  - Worst case: 24,810 ms, when elements are sorted in the reverse order
- Often, we perform only the worst-case analysis

Algorithm Analysis
Mathematical Functions

- $f(n) = c$ (constant)
- $f(n) = c \log n$ (log n)
- $f(n) = cn$ (linear)
- $f(n) = cn \log n$ (n log n)
- $f(n) = cn^2$ (quadratic)

Algorithm Analysis

Rate of Growth

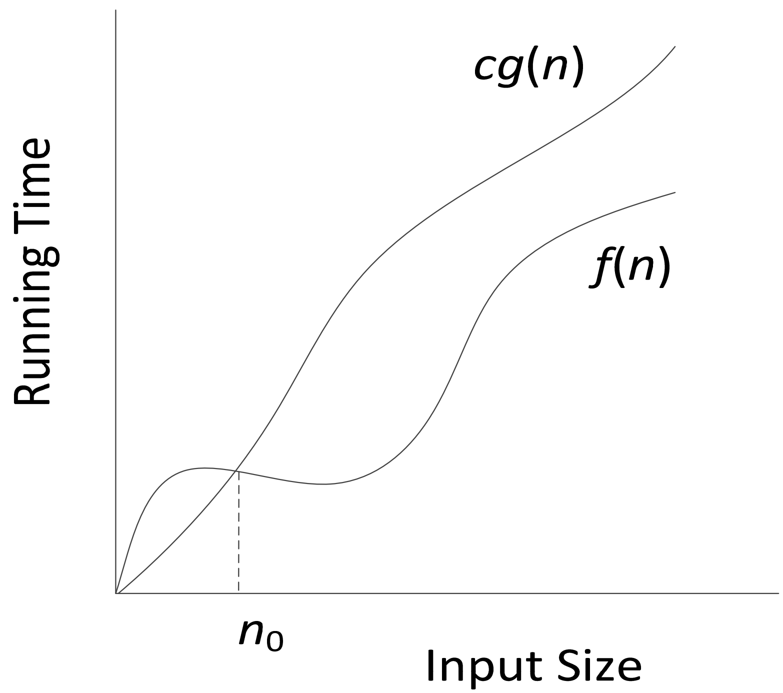
- When we analyze the running time of algorithms, we do not look at the actual running times.
- Instead, we focus on the rate of growth, i.e., how fast or slowly the running time grows as the input size increases infinitely.
- This is called *asymptotic analysis*.

- Notations: O (big-oh), Ω (big-omega), and Θ (big-theta)

  (There are also small-oh, small-omega, and small-theta)

Algorithm Analysis
Rate of Growth

- $O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for $n \geq n_0\}$



Algorithm Analysis
Rate of Growth

- $f(n) = 3n + 2 \Rightarrow f(n) = O(n)$

  Proof:

  Let $g(n) = n$, $c = 4$, and $n_0 = 2$. Then,

  $f(n) = 3n + 2 \le 4n$ for all $n \ge 2$, or

  $f(n) \le cg(n)$ for all $n \ge n_0$

  So, $f(n) = O(n)$

Algorithm Analysis

Rate of Growth

- $f(n) = 5n^3 + 2n^2 + 8n + 4 \Rightarrow f(n) = O(n^3)$

  Proof:

  $$f(n) = 5n^3 + 2n^2 + 8n + 4$$
  $$\leq 5n^3 + 2n^3 + 8n^3 + 4n^3$$
  $$= 19n^3$$

  If we let $g(n) = n^3$, $c = 19$ and $n_0 = 1$, then

  $f(n) \leq cg(n)$ for all $n \geq n_0$

  So, $f(n) = O(n^3)$

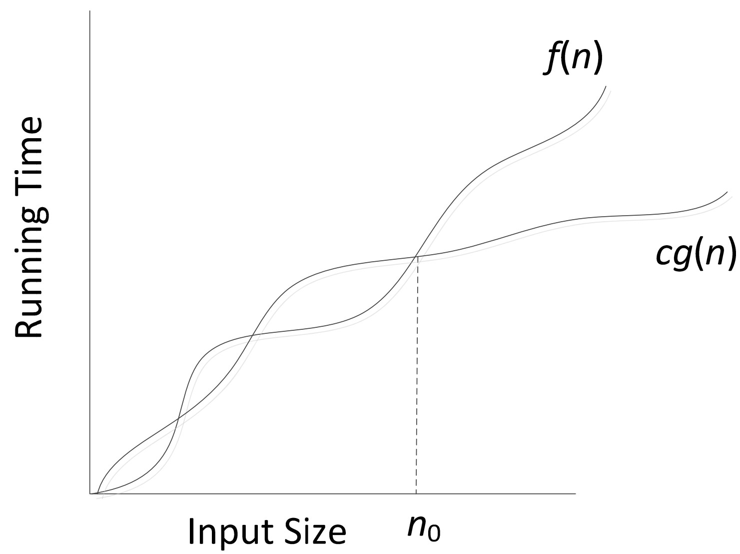
Algorithm Analysis

Rate of Growth

- $f(n) = 3n + 2 \Rightarrow O(n)$
- $f(n) = 5n^3 + 2n^2 + 8n + 4 \Rightarrow O(n^3)$
- $f(n) = 2n^2 + 2n \log n + 2n + 4 \Rightarrow O(n^2)$
- $f(n) = 2n \log n + 10n - 6 \Rightarrow O(n \log n)$
- $f(n) = 5n + 23\log n \Rightarrow O(n)$
- $f(n) = 3\log n + 10 \Rightarrow O(\log n)$

Algorithm Analysis
Rate of Growth

- $\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $f(n) \geq cg(n)$ for $n \geq n_0\}$

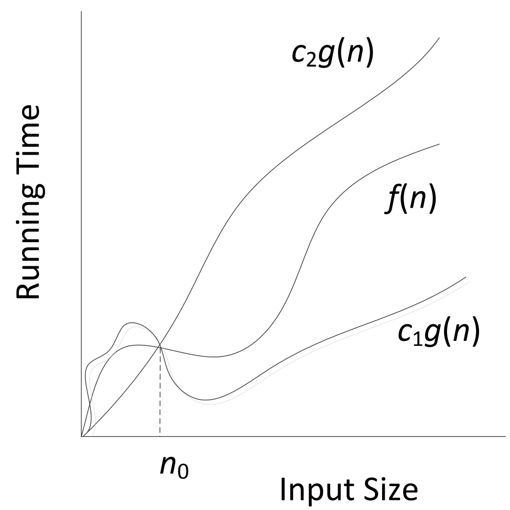


- $f(n) = 3n \log n - 2n \Rightarrow f(n) = \Omega(n \log n)$
- $f(n) = 5n^3 + 2n^2 + 8n + 4 \Rightarrow f(n) = \Omega(n^3)$

Algorithm Analysis

Rate of Growth

- $\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ for all $n \geq n_0\}$



- $f(n) = 3n \log n + 4n + 5\log n \Rightarrow f(n) = \Theta(n \log n)$
- $f(n) = 5n^3 + 2n^2 + 8n + 4 \Rightarrow f(n) = \Theta(n^3)$

Algorithm Analysis
Rate of Growth

- Rate of growth of different functions:

n	log n	n	n log n	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4096	65536
32	5	32	160	1024	32768	4294967296
64	6	64	384	4096	262144	1.84467E+19
128	7	128	896	16384	2097152	3.40282E+38
256	8	256	2048	65536	16777216	1.15792E+77
512	9	512	4608	262144	134217728	1.3408E+154

Algorithm Analysis
Rate of Growth

- Example: Find the largest element

```
public static double arrayMax(double[ ] data) {
  int n = data.length;                    // c1
  double currentMax = data[0];    // c2
  for (int j=1; j < n; j++)               // loop executed n – 1 times
    if (data[j] > currentMax)            // loop body takes c3
      currentMax = data[j];
  return currentMax;                     // c4
}
```

- Total running time, $f(n) = c1 + c2 + c3(n - 1) + c4$
- $f(n) = O(n)$

Algorithm Analysis
Rate of Growth

- Example: Three-way set disjointness problem (solution 1)

```
public static boolean disjoint1(int[ ] groupA, int[ ] groupB, int[ ] groupC) {
  for (int a : groupA)
    for (int b : groupB)
      for (int c : groupC)
        if ((a == b) && (b == c))
          return false
  return true;
}
```

- $f(n) = O(n^3)$

Algorithm Analysis

Rate of Growth

- Example: Three-way set disjointness problem (solution 2)

```
public static boolean disjoint1(int[ ] groupA, int[ ] groupB, int[ ] groupC) {
  for (int a : groupA)
    for (int b : groupB)
      if (a == b)
        for (int c : groupC)
          if (b == c)
            return false
  return true;
}
```

- $f(n) = O(n^2)$

Algorithm Analysis
Rate of Growth

- Example: Element uniqueness problem (solution 1)

```
public static boolean unique1(int[ ] data) {
  int n = data.length;
  for (int j=0; j < n-1; j++)
    for (int k=j+1; k < n; k++)
      if (data[j] == data[k])
        return false;                    // found duplicate pair
  return true;                           // if we reach this, elements are unique
}
```

- In the worst case, $f(n) = (n - 1) + (n - 2) + \ldots + 1 = O(n^2)$

Algorithm Analysis
Rate of Growth

- Example: Element uniqueness problem (solution 2)

```
public static boolean unique2(int[ ] data) {
  int n = data.length;
  int[ ] temp = Arrays.copyOf(data, n);   // make copy of data
  Arrays.sort(temp);                      // and sort the copy, O(n log n)
  for (int j=0; j < n-1; j++)             // for loop takes O(n)
    if (temp[j] == temp[j+1])             // check neighboring entries
      return false;                       // found duplicate pair
  return true;                            // if we reach this, elements are unique
}
```

- In the worst case, $f(n) = O(n \log n) + O(n) = O(n \log n)$

Proof Techniques

- To disprove a statement, it is sufficient to show a counterexample.
- To prove a statement, we must show it is true for all objects in the domain under consideration.

- Exhaustive proof, direct proof, proof by contraposition, proof by contradiction, mathematical induction

- Loop invariant method: to prove the correctness of an algorithm (or a program), which involves a loop.

Proof Techniques
Proof by Contradiction

- To prove P → Q: Assume Q is false and find a contradiction.
- Example: If an even integer is added to another even integer, the result is an even integer.

- Proof:
  - Let x and y be two even integers. Let $z = x + y$.
  - Let's assume that z is an odd integer (negating the conclusion of the given statement).
  - Since x is even, it can be rewritten as $x = 2n$, for some integer n.
  - Since y is even, it can be rewritten as $y = 2m$, for some integer m.
  - Since z is odd (this we assumed), it can be rewritten as $z = 2k + 1$, for some integer k.

Proof Techniques
Proof by Contradiction

- Proof (continued)
  - Then, we have:
  - $x + y = z$
  - $2n + 2m = 2k + 1$
  - $2n + 2m - 2k = 1$
  - $2(n + m - k) = 1$
  - This is a contradiction because the left hand side is an even number and the right hand side is 1, which is odd.

Proof Techniques
Induction

- Consists of base case (or base step) and inductive step.
- To prove a predicate P(n) is true for all positive integers n.
  - Base case: Show that P(1) is true
  - Inductive step: Assume that P(k) is true, and prove P(k + 1) is also true. This assumption is called *inductive hypothesis*.

- See the example in the next slide.

Proof Techniques
Induction

- Example: Prove that for any positive integer n, $2^n > n$

Base case: $n = 1$

LHS = $2^1 = 2$, RHS = 1. So, LHS > RHS

Induction step: ($n \geq 1$).

Inductive hypothesis: it is true for $n = k$ ($k \geq 1$), i.e., $2^k > k$ for $k \geq 1$.

We show that it is also true for $n = k + 1$, i.e., $2^{k+1} > k + 1$

LHS = 2^{k+1}

$= 2 \times 2^k$

$> 2k$ (by the inductive hypothesis)

$= k + k \geq k + 1$

= RHS.

So, LHS > RHS.

Proof Techniques
Loop Invariant

- **Loop invariant**:
  - A property (or a statement) that is true at the beginning of each iteration of the loop
- To prove an algorithm is correct
  - (1). **Identify** a loop invariant and **state** it.
    Choose one that will help prove the correctness
  - (2). Prove the loop invariant is true. This involves two steps – *base case* and *inductive step*
  - (3). Use the loop invariant to prove the correctness of the algorithm.

Proof Techniques
Loop Invariant

- Insertion Sort

```
public class InsertionSort {
  public static void insertionSort(char[ ] data) {
    int n = data.length;
    for (int k = 1; k < n; k++) {    // begin with second element
        char cur = data[k];              // save data[k] in cur
        int j = k;                              // find correct index j for cur
        while (j > 0 && data[j-1] > cur) {  // thus, data[j-1] must go after cur
            data[j] = data[j-1];                    // shift data[j-1] rightward
             j--;                                           // and consider previous j for cur
       } // while
       data[j] = cur;                          // this is the proper place for cur
     } // for
}                                                   // running time: O(n^2)
```

Proof Techniques

Loop Invariant

- **Loop invariant (L.I.):** At the start of each iteration of the **for** loop of lines 4 - 12, the subarray *data*[0 . . k −1] consists of the elements originally in *data*[0 . . k − 1] but in sorted order.

- **Base case:** Just before the first iteration, k = 1. The subarray *data*[0 . . k − 1] has only one element *data*[0], and it is trivially sorted.

- **Inductive step:**
  - Assume the L.I. is true at the start of an iteration and prove it is also true at the start of the next iteration.

  - Assume the loop invariant is true at the start of the m-th iteration, i.e. when k = m, the subarray *data*[0 . . m - 1] consists of the elements originally in *data*[0 . . m - 1] but in sorted order.

Proof Techniques
Loop Invariant

- **Inductive step: (continued)**
  - During the execution of the *m*-th iteration, the *while* loop in lines 7 – 10 moves *data*[*m*] into the correct position in *data*[0 . . *m* – 1].
  - So, at the start of the (*m*+1)-th iteration, *data*[0 . . *m*] consists of the elements originally in *A*[0 . . *m*] but in sorted order.

- The **for** loop ends when *k* = *n*.

- According to the loop invariant, the subarray *data*[0 . . *n – 1*] consists of the elements originally in *data*[0 . . *n – 1*] but in sorted order. In other words, the entire array is sorted.

Recursion

- A recursive function is a function which is defined in terms of itself.
- A recursion, in programming, is a way of implementing repeated execution of statements (or a method), where a method invokes itself.

- Example: Factorial

$$
n! = \begin{cases} 1 & \text{if } n = 0 \\ n*(n-1)! & \text{if } n \geq 1 \end{cases}
$$

Recursion
Factorial

- Java implementation

```
public static int factorial(int n) throws IllegalArgumentException {
  if (n < 0)
      throw new IllegalArgumentException();  // argument must be
                                             // nonnegative
  else if (n == 0)
      return 1;                              // base case
  else
      return n * factorial(n-1);             // recursive case
}
```

Recursion
Factorial

- Recursion trace

factorial(4) returns 4*6 = 24

4 * factorial(3) returns 3*2 = 6

3 * factorial(2) returns 2*1 = 2

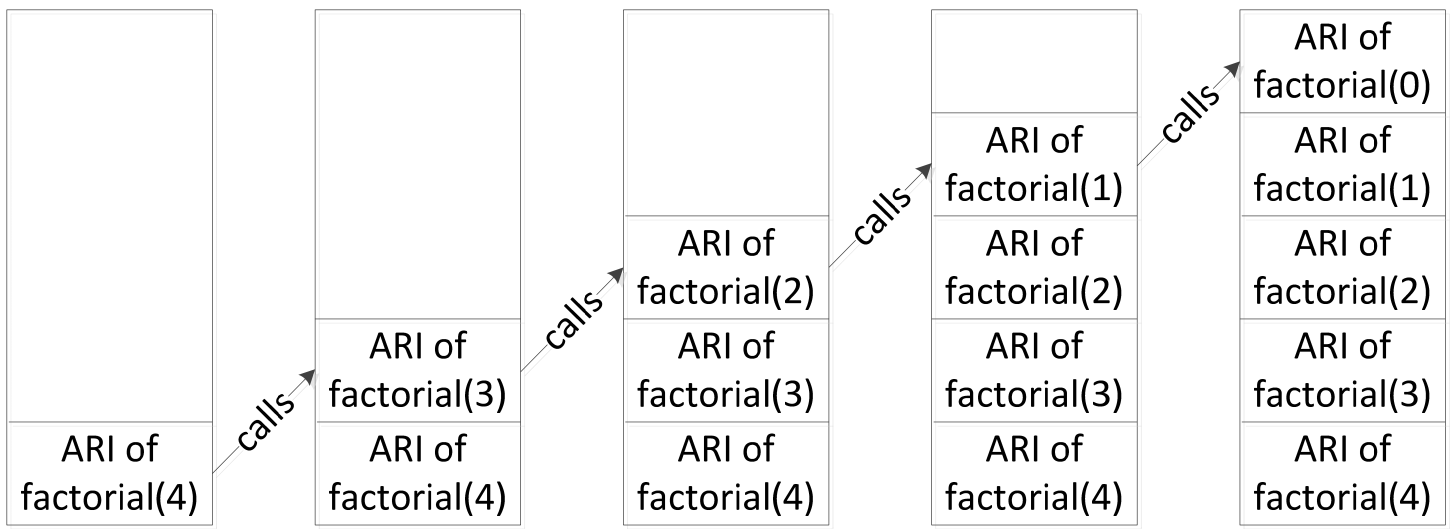
2 * factorial(1) returns 1*1 = 1

1 * factorial(0)

returns 1

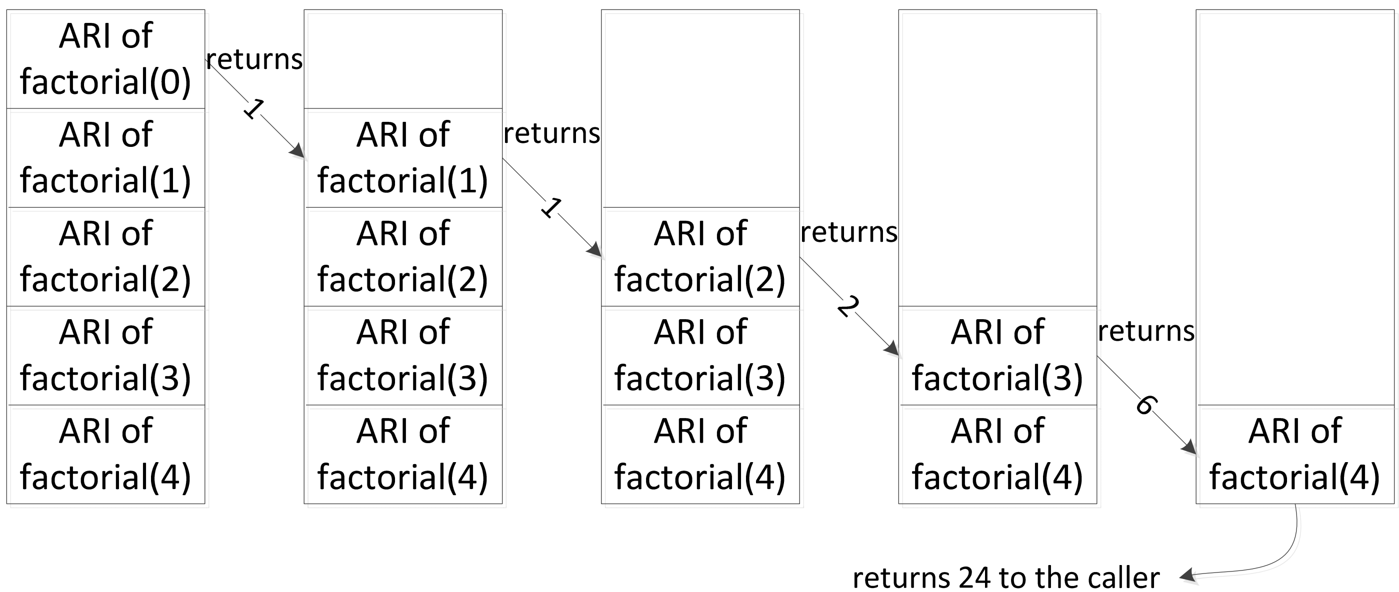
Recursion
Factorial

- Recursive calls



Recursion
Factorial

- Returning from calls



- Running time of *factorial*: $O(n)$

Recursion
Binary Search

- Search a sequence of n elements for a target element.
- Linear search
  - Examine each element while scanning the sequence
  - Best case: one comparison, or $O(1)$
  - Worst case: n comparisons, or $O(n)$
  - On average: $n/2$ comparisons, or $O(n)$

- Binary search
  - If the sequence is sorted, we can use binary search
  - Running time is $O(\log n)$

Recursion
Binary Search

- Pseudocode

```
Algorithm binarySearch(int[ ] data, int target, int low, int high)
  If low > high                    // target is not found
     return false
  else
     mid = floor((low + high)/2) // median candidate
     if target = data[mid]         // target found
        return true
     else if target < data[mid]
        search data[low .. mid-1] recursively
     else
        search data[mid+1 .. high] recursively
```

Recursion
Binary Search

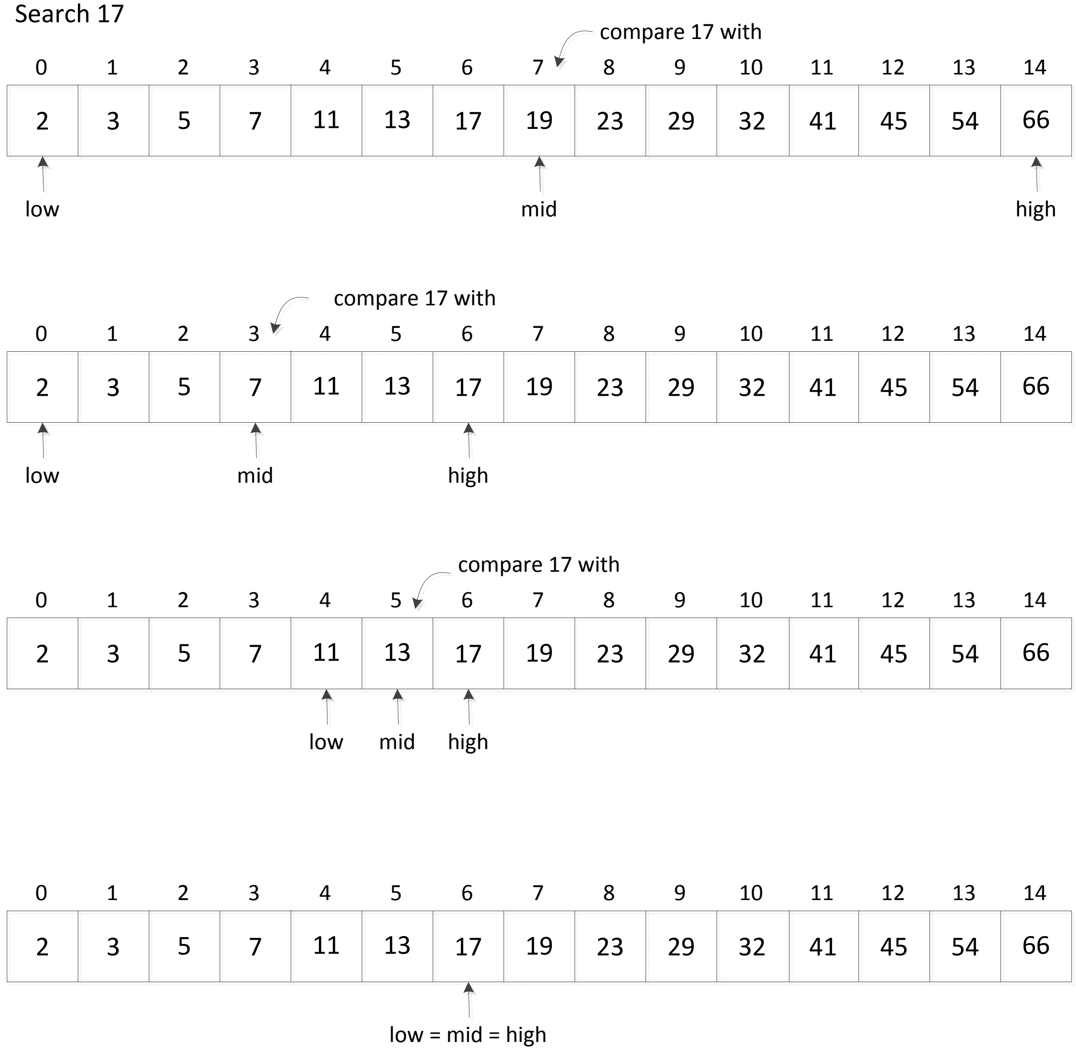
- Java implementation

```
public static boolean binarySearch(int[ ] data, int target,
                                      int low, int high) {
if (low > high)
    return false;                  // interval empty; no match
else {
    int mid = (low + high) / 2;
    if (target == data[mid])
        return true;               // found a match
    else if (target < data[mid]) // recurse left of the middle
        return binarySearch(data, target, low, mid - 1);
    else  // recurse right of the middle
        return binarySearch(data, target, mid + 1, high);
  }
}
```

Recursion

Binary Search

Search 17



Recursion
Binary Search

- Running time analysis
  - Execution of one call takes $O(1)$.
  - Each time binary search is (recursively) invoked, the number of elements to be searched is reduced to at most half.
  - Initially, there are n elements.
  - In the first recursive call, there are at most $n/2$ elements.
  - In the second recursive call, there are at most $n/4$ elements.
  - and so on …

Recursion

Binary Search

- Running time analysis (continued)
  - In the j-th recursive call, there are at most n / (2^j) elements.
  - In the worst case, the target is not in the sequence. In this case, recursion stops when there is no more elements to be searched.
  - The max. number of recursive calls is the smallest integer r such that $\frac{n}{2^r} < 1$
  - Or, r is the smallest integer such that $r > \log n$
  - Therefore, $r = \lfloor \log n \rfloor + 1$
  - So, the total running time is $O(\log n)$

Recursion
More Examples

- Print array elements recursively - Pseudocode

```
Algorithm printArrayRecursively(data, i)
    if i = n, return
    else
        print data[i]
        i = i + 1
        printArrayRecursively(data, i)
```

Recursion
More Examples

- Print array elements recursively – Java code

```
public static void printArrayRecursive(int[ ] data, int i){
  if (i == data.length)
    return;
 else{
    System.out.print(data[i++] + " ");
    printArrayRecursive(data, i);
 }
}
```

Recursion
More Examples

- Reverse sequence recursively – Pseudocode

```
Algorithm reverseArray(data, low, high)
    if low >= high, return
    else
        swap data[low] with data[high]
        reverseArray(data, low+1, high-1)
```

Recursion
More Examples

- Reverse sequence recursively – Java code

```
public static void reverseArray(int[] data, int low,
                                            int high) {
  if (low < high) {
    int temp = data[low];
    data[low] = data[high];
    data[high] = temp;
    reverseArray(data, low + 1, high - 1);
  }
}
```

Recursion
More Examples

- Binary sum – Java code

```
public static int binarySum(int[] data, int low, int high) {
  if (low > high)            // zero elements in subarray
    return 0;
  else if (low == high)      // one element in subarray
    return data[low];
  else {
    int mid = (low + high) / 2;
    return binarySum(data, low, mid) +
                           binarySum(data, mid+1, high);
  }
}
```

Recursion
Computing Powers

- Definition: *power*(x, n) = x^n
- Recursive definition

$$power(x, n) = \begin{cases} 1 & \text{if } n = 0 \\ x * power(x, n-1) & \text{otherwise} \end{cases}$$

- Direct implementation

```
public static double power(double x, int n) {
  if (n == 0)
    return 1;
  else
    return x * power(x, n-1);
}
```

- Execution of each method call takes $O(1)$.
- The method is invoked ($n + 1$) times.
- Running time is $O(n)$

Recursion
Computing Powers

- There is an efficient method.
- Let $k = \left\lfloor \frac{n}{2} \right\rfloor$
- If *n* is even, $k = \frac{n}{2}$ and if *n* is odd, $k = \frac{n-1}{2}$
- So,

$$\left(x^k\right)^2 = \left(x^{\frac{n}{2}}\right)^2 = x^n$$ if *n* is even

$$\left(x^k\right)^2 = \left(x^{\frac{n-1}{2}}\right)^2 = x^{n-1}$$ if *n* is odd

Recursion
Computing Powers

- Then, we can redefine *power*(x, n) as follows:

$$power(x, n) = \begin{cases} 1 & \text{if } n = 0 \\ \left(power\left(x, \left\lfloor \frac{n}{2} \right\rfloor\right)\right)^2 & \text{if } n \text{ is even} \\ \left(power\left(x, \left\lfloor \frac{n}{2} \right\rfloor\right)\right)^2 \cdot x & \text{if } n \text{ is odd} \end{cases}$$

Recursion
Computing Powers

- Illustration

power(2,5) partial = 4; result = 4 * 4 = 16; n is odd; result = 16 * 2 = 32

calls returns 4

power(2,2) partial = 2; result = 2 * 2 = 4;

calls returns 2

power(2,1) partial = 1; result = 1 * 1 = 1; n is odd; result = 1 * 2 = 2

calls returns 1

power(2,0)

Recursion
Computing Powers

- Implementation

```
public static double power(double x, int n) {
  if (n == 0)
    return 1;
  else {
    double partial = power(x, n/2); // use integer division of n
    double result = partial * partial;
    if (n % 2 == 1)      // if n odd, include extra factor of x
      result *= x;
    return result;
  }
}
```

- Execution of one call takes $O(1)$.
- The method is invoked $O(\log n)$ times.
- Running time is $O(\log n)$

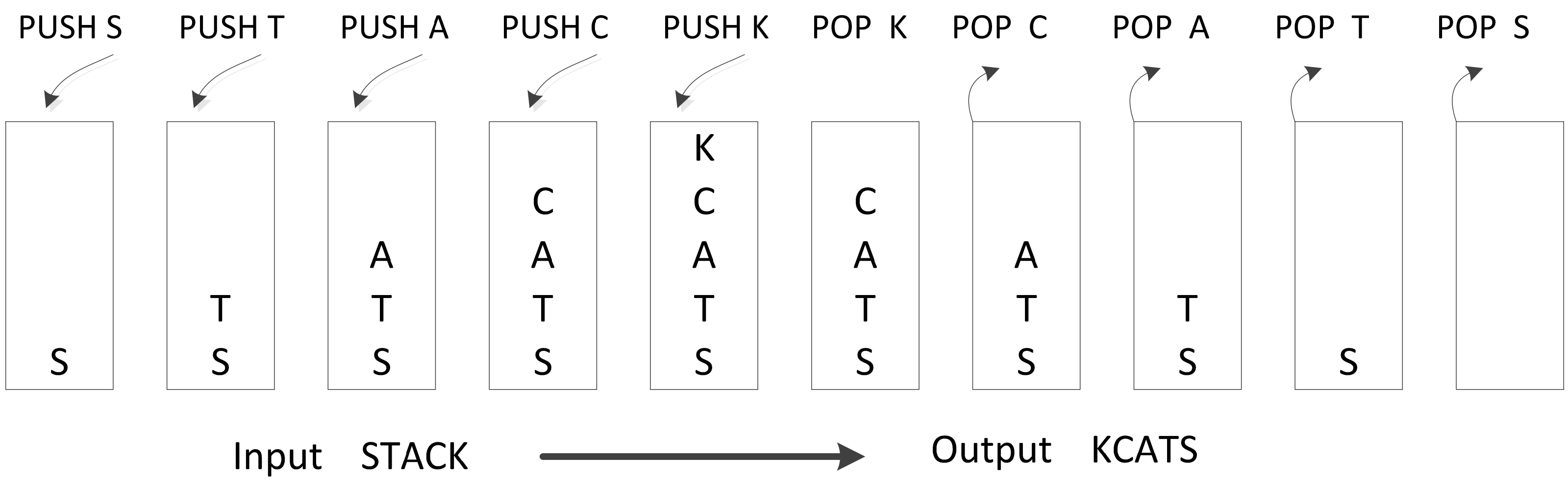
Recursion
Designing Recursive Algorithms

- Two components: base case and recursion
- Base case:
  - Recursive call stops when a certain condition is met.
  - This is usually referred to as *base case*.
- Recursion: When the condition of the base case is not met, the algorithm invokes itself recursively.

- Sometimes, a helper method is necessary.
- When poorly designed, very inefficient.
- Make sure the base case is always reached to avoid infinite recursion.

Stacks

- Illustration



Stacks

- Stack ADT supports the following operations:
  - push(*e*): Adds element *e* to the stack top.
  - pop( ): Removes and returns the top element from the stack. Returns null if the stack is empty.
  - top( ): Returns the top element of the stack without removing it. Returns null if the stack is empty.
  - size( ): Returns the number of elements in the stack.
  - isEmpty( ): Returns true if the stack is empty and false otherwise.

Stacks

- Illustration

Operation	Return Value	Stack Contents → top
push(10)	-	(10)
push(20)	-	(10, 20)
push(5)	-	(10, 20, 5)
size( )	3	(10, 20, 5)
top( )	5	(10, 20, 5)
pop( )	5	(10, 20)
push(30)	-	(10, 20, 30)
pop( )	30	(10, 20)
pop( )	20	(10)
pop( )	10	( )
isEmpty( )	true	( )
pop( )	null	( )

Stacks

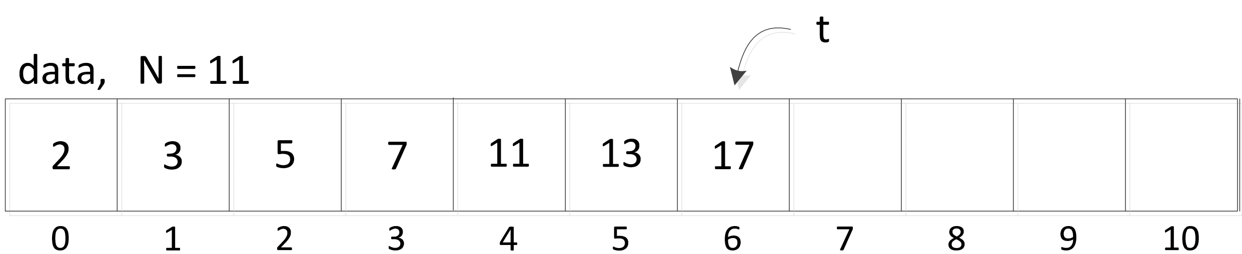
- Java's stack (*java.util.Stack* class)

Stack ADT in Textbook	Class java.util.Stack
size( )	size( )
isEmpty( )	empty( )
push(e)	push(e)
pop( )	pop( )
top( )	peek( )

- JavaStackDemo.java

Stacks

- Array-based implementation
  - The bottom element is stored in *data*[0].
  - The top element is stored in *data*[*t*], $0 \leq t < N$.
  - When the stack is empty, by convention, $t = -1$.



- ArrayStack.java

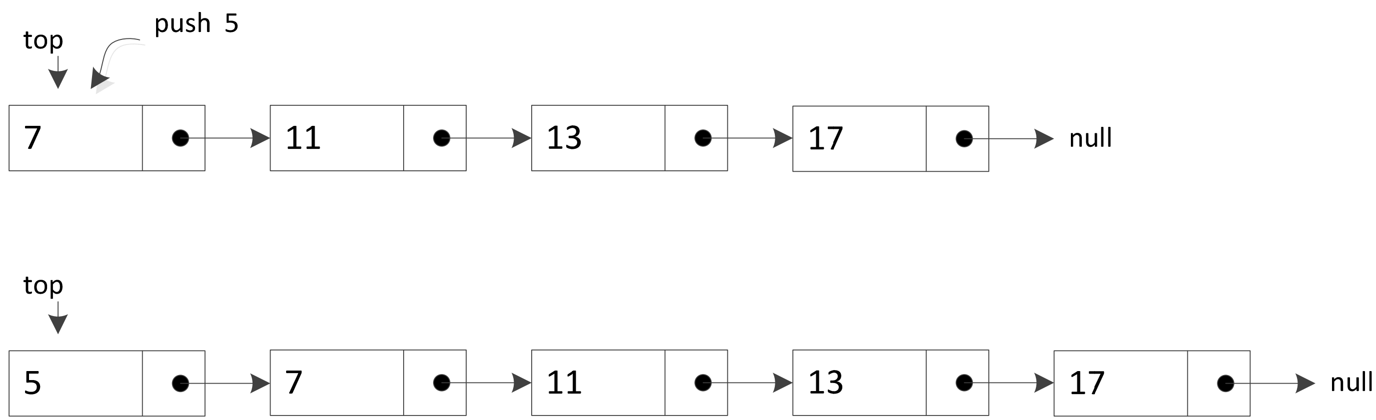
Stacks

- Running times of array-based implementation

Method	Running Time
size( )	O(1)
isEmpty( )	O(1)
push(e)	O(1)
pop( )	O(1)
top( )	O(1)

Stacks

- Stack implementation using singly-linked list.
  - Stack top element is stored at the head of a list.
  - All operations take $O(1)$.



- *LinkedStack.java* code

Stacks

- Reversing array elements

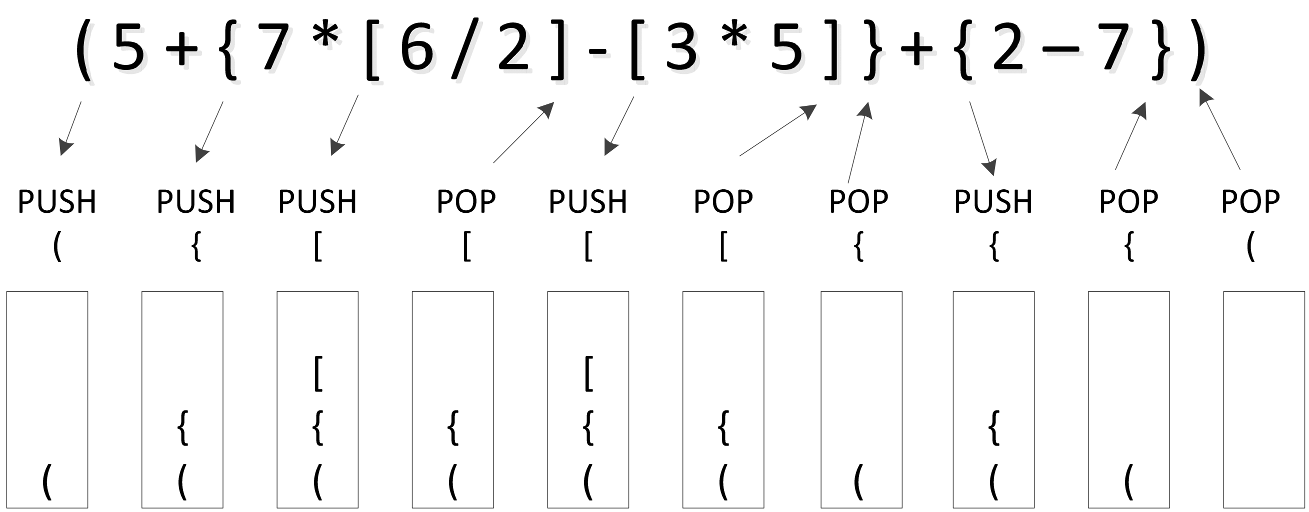
```
public static <E> void reverse(E[ ] a) {
  Stack<E> buffer = new ArrayStack<>(a.length);
  for (int i=0; i < a.length; i++)
    buffer.push(a[i]);
  for (int i=0; i < a.length; i++)
    a[i] = buffer.pop();
}
```

Stacks

- Matching parentheses
  - Scan the expression one character at a time from left to right
  - If the character is an opening delimiter, push it to the stack
  - If the character is a closing delimiter:
    - Pop a delimiter from the stack
    - Compare that with the closing delimiter being scanned
    - If they are a matching pair (for example, a left square bracket and a right square bracket), continue
    - Else, the expression is invalid

Stacks

- Matching parentheses (continued)

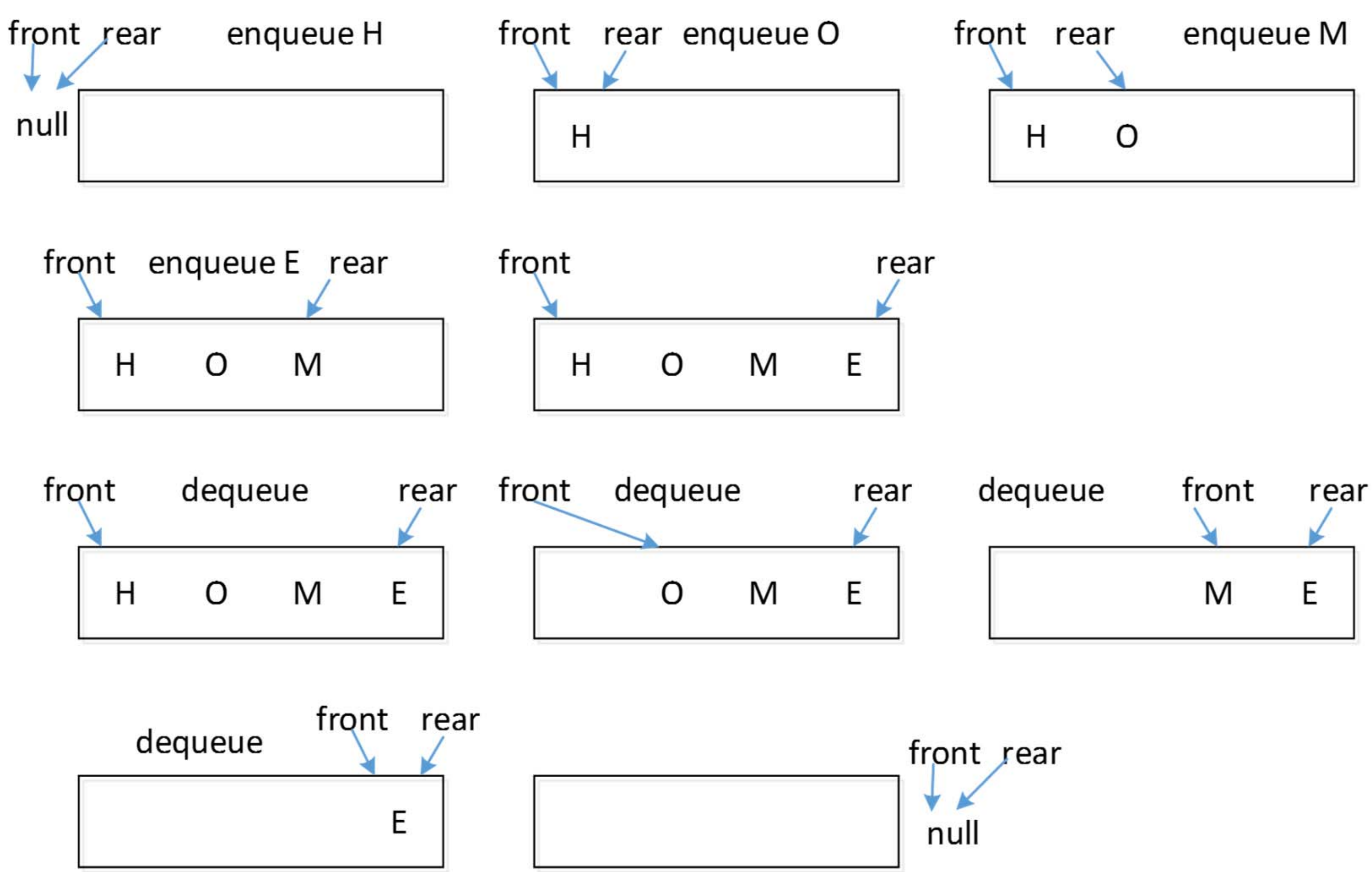


Queues

- A queue has a linear data structure (like stack)
- An object is added to one end called *rear*, and removed from the other end called *front*.
- FIFO (first in first out).
- Adding is called *enqueue*
- Removing is called *dequeue*

Queues

- Illustration



Queues

- Queue ADT operations:

  - enqueue(*e*): Adds element *e* to the back of queue.
  - dequeue( ): Remove and returns the first element from the queue. Returns null if the queue is empty.
  - first( ): Returns the first element of the queue, without removing it. Returns null if the queue is empty.
  - size( ): Returns the number of elements in the queue.
  - isEmpty( ): Returns true if the queue is empty and false otherwise.

Queues

- Illustration of operations:

Operation	Return Value	first ← Q ← last
enqueue(10)	-	(10)
enqueue(20)	-	(10, 20)
enqueue(5)	-	(10, 20, 5)
size( )	3	(10, 20, 5)
dequeue( )	10	(20, 5)
enqueue(30)	-	(20, 5, 30)
dequeue ( )	20	(5, 30)
dequeue ( )	5	(30)
dequeue ( )	30	( )
isEmpty( )	true	( )
dequeue( )	null	( )

Queues

- Java has the *java.util.Queue* interface.
- Java Queue interface operations:

Queue ADT	Interface java.util.Queue	
	throws exception	returns special value
enqueue(e)	add(e)	offer(e)
dequeue( )	remove( )	poll( )
first( )	element( )	peek( )
size( )	size( )	
isEmpty( )	isEmpty	

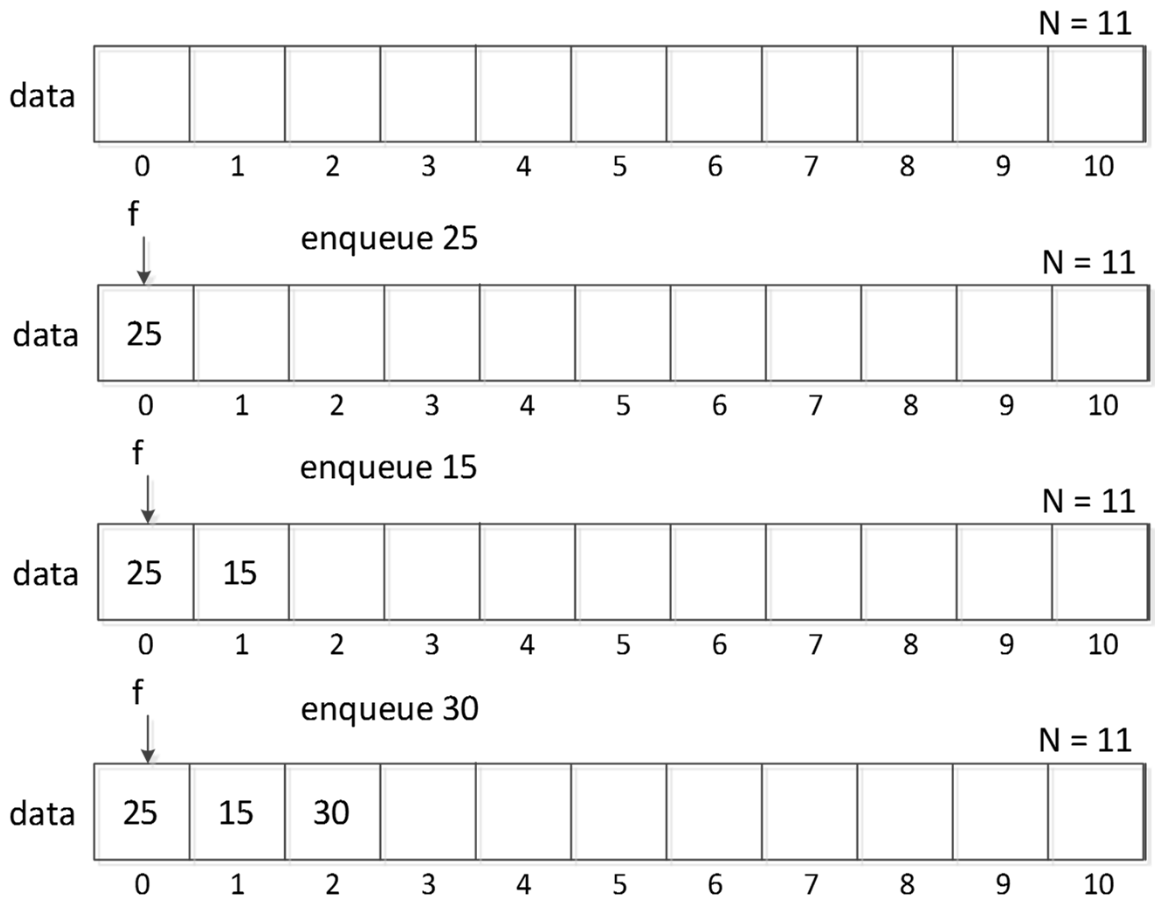
Queues

- In Java:
  - LinkedList class implements List and Deque interfaces.
  - Deque interface extends Queue interface.

- JavaQueueDemo.java

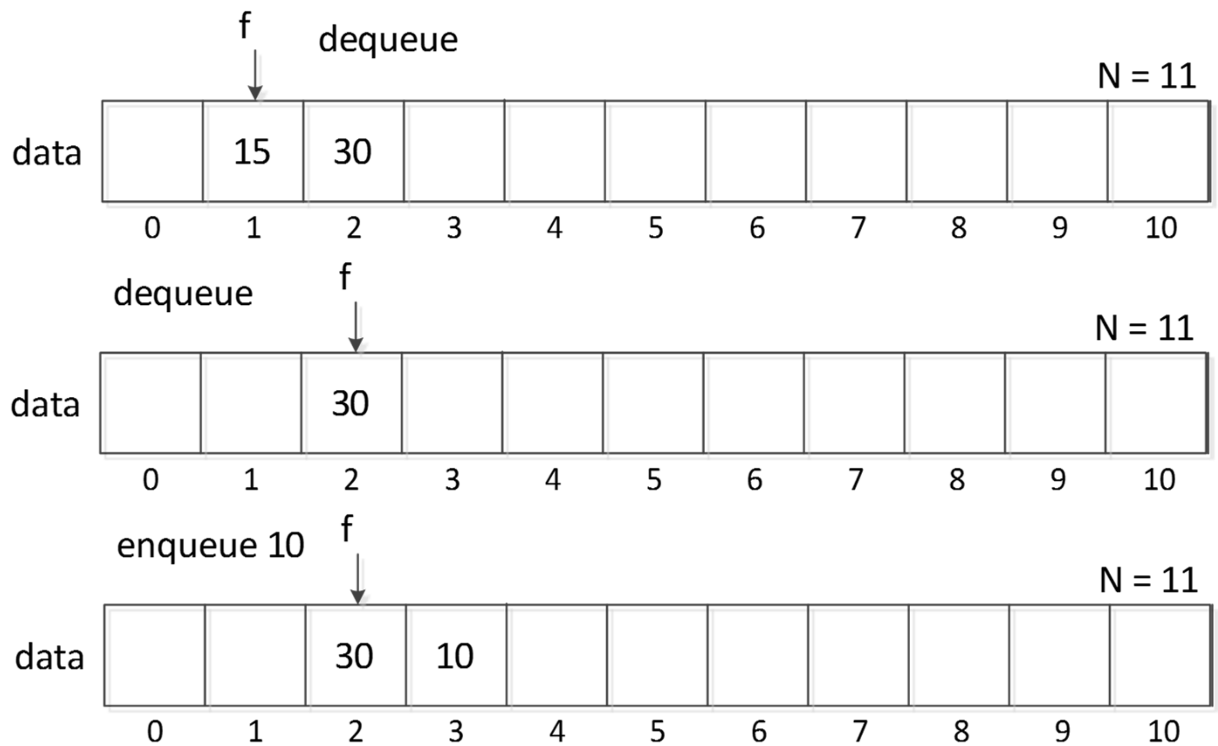
Queues

- Array-based implementation (*ArrayQueue.java*)



Queues

- Array-based implementation (continued)

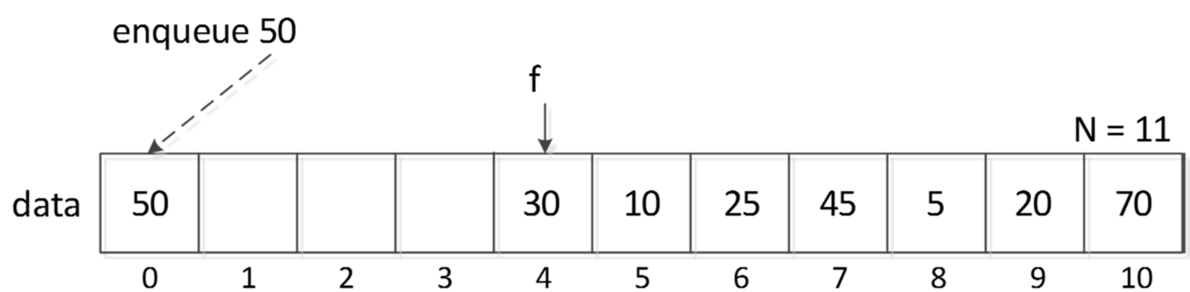


Queues

- Array-based implementation (continued)

- Elements are added in a wrap around manner:

```
public void enqueue(E e) throws IllegalStateException {
  if (sz == data.length)
      throw new IllegalStateException("Queue is full");
   int avail = (f + sz) % data.length;   // use modular arithmetic
   data[avail] = e;
   sz++;
}
```



Queues

- Singly linked list-based implementation

```
public class LinkedQueue<E> implements Queue<E> {
  private SinglyLinkedList<E> list = new SinglyLinkedList<>();
  public LinkedQueue() { }
  public int size() { return list.size(); }
  public boolean isEmpty() { return list.isEmpty(); }
  public void enqueue(E element) { list.addLast(element); }
  public E first() { return list.first(); }
  public E dequeue() { return list.removeFirst(); }
}
```

Queues

- Circular queue

```
public interface CircularQueue<E> extends
                                        Queue<E> {
    void rotate();
}
```

Queues

- Double-ended queue, called *deque* (pronounced “deck”).
  - Allows insertion and deletion at both ends.
  - Can be used as a stack or as a queue

- Queue and Deque (in Java)

Queue Method	Equivalent Deque Method
add(e)	addLast(e)
offer(e)	offerLast(e)
remove()	removeFirst()
poll()	pollFirst()
element()	getFirst()
peek()	peekFirst()

Queues

- Stack and Deque (in Java)

Stack Method	Equivalent Deque Method
push(e)	addFirst(e)
pop()	removeFirst()
peek()	peekFirst()

- *java.util.Deque* interface.
- *java.util.ArrayDeque* implements *Deque* interface.

References

- M.T. Goodrich, R. Tamassia, and M.H. Goldwasser, "Data Structures and Algorithms in Java," Sixth Edition, Wiley, 2014.