# Recursion (Part 3)

# Computational complexity

- computational complexity is concerned with describing the amount of resources needed to run an algorithm
  - for our purposes, the resource is time
- complexity is usually expressed as a function of $n$ the size of the problem
  - the size of the problem is always a non-negative integer value (i.e., a natural number)

# Recursively Move Smallest to Front

```
public class Week10 {

  public static void minToFront(List<Integer> t) {
    if (t.size() < 2) {
      return;
    }
    Week10.minToFront(t.subList(1, t.size()));
    int first = t.get(0);
    int second = t.get(1);
    if (second < first) {
      t.set(0, second);
      t.set(1, first);
    }
  }
}
```

size of problem, $n$, is the number of elements in the list **t**

# Estimating complexity

- the basic strategy for estimating complexity:
  1. for each line of code, estimate its number of elementary instructions
  2. for each line of code, determine how often it is executed
  3. determine the total number of elementary instructions

# Elementary instructions

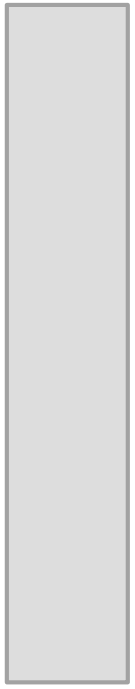
- what is an elementary instruction?
  - for our purposes, any expression that can be computed in a constant amount of time
- examples:
  - declaring a variable
  - assignment (=)
  - arithmetic (+, -, *, /, %)
  - comparison (<, >, ==, !=)
  - Boolean expressions (||, &&, !)
  - if, else
  - return statement

# Estimating complexity

- count the number of elementary operations in each line of **`minToFront`**
  - assume that the following are all elementary operations:
    - **`t.size()`**
    - **`t.get(0)`**
    - **`t.get(1)`**
    - **`t.set(0, ...)`**
    - **`t.set(1, ...)`**
    - **`t.subList(x, y)`**
  - leave the line with the recursive call blank for now

# Recursively Move Smallest to Front

```
public class Week10 {

  public static void minToFront(List<Integer> t) {
    if (t.size() < 2) {
      return;
    }
    Week10.minToFront(t.subList(1, t.size()));
    int first = t.get(0);
    int second = t.get(1);
    if (second < first) {
      t.set(0, second);
      t.set(1, first);
    }
  }
}
```



# Estimating complexity

- for each line of code, determine how often it is executed

# Recursively Move Smallest to Front

```
public class Week10 {

  public static void minToFront(List<Integer> t) {
    if (t.size() < 2) {                                  1
      return;                                            1 or 0
    }
    Week10.minToFront(t.subList(1, t.size()));           1 or 0
    int first = t.get(0);                                1 or 0
    int second = t.get(1);                               1 or 0
    if (second < first) {                                1 or 0
      t.set(0, second);                                  1 or 0
      t.set(1, first);                                   1 or 0
    }
  }
}
```

# Total number of operations

- before we can determine the total number of elementary operations, we need to count the number of elementary operations arising from the recursive call
- let $T(n)$ be the total number of elementary operations required by **minToFront(t)**

# Total number of operations

```
public class Week10 {

  public static void minToFront(List<Integer> t) {



    Week10.minToFront(t.subList(1, t.size()));

                                        1 elementary operation



  }
}
```

# Total number of operations

```
public class Week10 {

  public static void minToFront(List<Integer> t) {



    Week10.minToFront(t.subList(1, t.size()));

                      1 elementary operation

  }
}
```

# Total number of operations

```
public class Week10 {

  public static void minToFront(List<Integer> t) {

    Week10.minToFront(t.subList(1, t.size()));

  }
}
```

$T(n-1)$ elementary operations

# Total number of operations

```
public class Week10 {

  public static void minToFront(List<Integer> t) {



    Week10.minToFront(t.subList(1, t.size()));

  }
}
```

$T(n-1)$ elementary operations

**1** elementary operation

**1** elementary operation

$= T(n-1) + 2$

# Total number of operations

```
public class Week10 {

  public static void minToFront(List<Integer> t) {
    if (t.size() < 2) {
      return;
    }
    Week10.minToFront(t.subList(1, t.size()));
    int first = t.get(0);
    int second = t.get(1);
    if (second < first) {
      t.set(0, second);
      t.set(1, first);
    }
  }
}
```

these lines run if the base case is true

# Total number of operations

- base cases
  - $T(0) = T(1) = 4$

# Total number of operations

```
public class Week10 {

  public static void minToFront(List<Integer> t) {
    if (t.size() < 2) {
      return;
    }
    Week10.minToFront(t.subList(1, t.size()));
    int first = t.get(0);
    int second = t.get(1);
    if (second < first) {
      t.set(0, second);
      t.set(1, first);
    }
  }
}
```

this line runs if the base case is not true

these lines run if the base case is not true

these lines might run if the base case is not true

# Total number of operations

- when counting the total number of operations, we often consider the worst case scenario
  - let's assume that the lines that might run always run

# Total number of operations

- base cases
  - $T(0) = T(1) = 4$
- recursive case
  - $T(n) = T(n-1) + 15$

- the two equations above are called the *recurrence relation* for **minToFront**
- let's try to solve the recurrence relation

# Solving the recurrence relation

$T(0) = 4$
$T(1) = 4$
$T(n) = T(n-1) + 15$

- if we knew $T(n-1)$ we could solve for $T(n)$

$$T(n) = T(n-1) + 15$$
$$= (T(n-2) + 15)$$
$$= T(n-2) + 2(15)$$

$$T(n-1) = T(n-2) + 15$$

# Solving the recurrence relation

$T(0) = 4$
$T(1) = 4$
$T(n) = T(n-1) + 15$

- if we knew $T(n-2)$ we could solve for $T(n)$

$$T(n) = T(n-1) + 15$$
$$= (T(n-2) + 15) + 15$$
$$= T(n-2) + 2(15)$$
$$= (T(n-3) + 15) + 2(15)$$
$$= T(n-3) + 3(15)$$

$$T(n-1) = T(n-2) + 15$$

$$T(n-2) = T(n-3) + 15$$

# Solving the recurrence relation

$T(0) = 4$
$T(1) = 4$
$T(n) = T(n-1) + 15$

- if we knew $T(n-3)$ we could solve for $T(n)$

$$
\begin{aligned}
T(n) &= T(n-1) + 15 \\
&= (T(n-2) + 15) + 15 \\
&= T(n-2) + 2(15) \\
&= (T(n-3) + 15) + 2(15) \\
&= T(n-3) + 3(15) \\
&= (T(n-4) + 15) + 3(15) \\
&= T(n-4) + 4(15)
\end{aligned}
$$

$T(n-1) = T(n-2) + 15$

$T(n-2) = T(n-3) + 15$

$T(n-3) = T(n-4) + 15$

# Solving the recurrence relation

$T(0) = 4$
$T(1) = 4$
$T(n) = T(n-1) + 15$

- there is clearly a pattern

$T(n) = T(n-k) + k(15)$

# Solving the recurrence relation

$$T(0) = 4$$
$$T(1) = 4$$
$$T(n) = T(n-1) + 15$$

- substitute $k = n - 1$ so that we reach a base case

$$\begin{aligned} T(n) &= T(n-k) + k(15) \\ &= T\big(n-(n-1)\big) + (n-1)(15) \\ &= T(1) + 15n - 15 \\ &= 4 + 15n - 15 \\ &= 15n - 11 \end{aligned}$$

# Big-O notation

- when counting the number of elementary operations we assumed that all elementary operations would run in 1 unit of time
- in reality this isn't true and exactly what constitutes an elementary operation and how much time each operation requires depends on many factors
- in our expression $T(n) = 15n - 11$ the constants 15 and 11 are likely to be inaccurate
- big-O notation describes the complexity of an algorithm that is insensitive to variations in how elementary operations are counted

# Big-O notation

- using big-O notation we say that the complexity of **minToFront** is in $O(n)$

- more formally, a function $f(n)$ is an element of $O(g(n))$ if and only if there is a positive real number $M$ and a real number $m$ such that

$$| f(n) | < M | g(n) | \text{ for all } n > m$$

# Big-O notation

- Claim: $T(n) = 15n - 11 \in O(n)$
- Proof: $f(n) = 15n - 11, g(n) = n$

For $n \geq 1$, $f(n) > 0$ and $g(n) \geq 0$; therefore, we do not need to consider the absolute values. We need to find $M$ and $m$ such that the following is true:

$$15n - 11 < Mn \text{ for all } n > m$$

For all $n > 0$ we have:

$$15n - 11 < 15n$$

$\therefore 15n - 11 < 15n$ for all $n > 0$ and $T(n) \in O(n)$

# Big-O notation

- Proof 2: $f(n) = 15n - 11, g(n) = n$

For $n \geq 1$, $f(n) > 0$ and $g(n) \geq 0$; therefore, we do not need to consider the absolute values. We need to find $M$ and $m$ such that the following is true:

$$15n - 11 < Mn \text{ for all } n > m$$

For $n > 0$ we have:

$$\frac{15n - 11}{n} < \frac{15n}{n} < 15$$

$\therefore 15n - 11 < 15n$ for all $n > 0$ and $T(n) \in O(n)$

# Big-O notation

- the second proof uses the following recipe:
  1. Choose $m = 1$
  2. Assuming $n > 1$ derive $M$ such that

$$\frac{|f(n)|}{|g(n)|} < M \frac{|g(n)|}{|g(n)|} = M$$

- assuming $n > 1$ implies that $1 < n, n < n^2, n^2 < n^3$, etc. which means you can replace terms in the numerator to simplify the expression

# Big-O notation

- Claim: $f(n) = 3n^2 - n + 100 \in O(n^2)$
- Proof:

1. Choose $m = 1$
2. Assume $n > 1$

change to +　　increase　　increase

$$\frac{|3n^2 - n + 100|}{|n^2|} < \frac{3n^2 + n + 100}{n^2} < \frac{3n^2 + n^2 + 100n^2}{n^2}$$

$$= \frac{104n^2}{n^2}$$

$$= 104$$

# $O(1)$

- $O(1)$ describes an algorithm that runs in constant time
  - i.e., the run time does not depend on the size of the input

# $O(\log_2 n)$

- $O(\log_2 n)$ describes an algorithm whose runtime grows in proportion to the logarithm of the input size
  - i.e., doubling the size of the input increases the runtime by 1 unit of time
  - called logarithmic complexity

# $O(n)$

- $O(n)$ describes an algorithm whose runtime grows in proportion to the size of the input
  - i.e., doubling the input size double the runtime (approximately)
  - called linear complexity

# $O(n\log_2 n)$

- $O(n \log_2 n)$ describes an algorithm whose runtime complexity is slightly greater than linear
  - i.e., doubling the size of the input more than doubles the runtime (approximately)
  - called linearithmic complexity

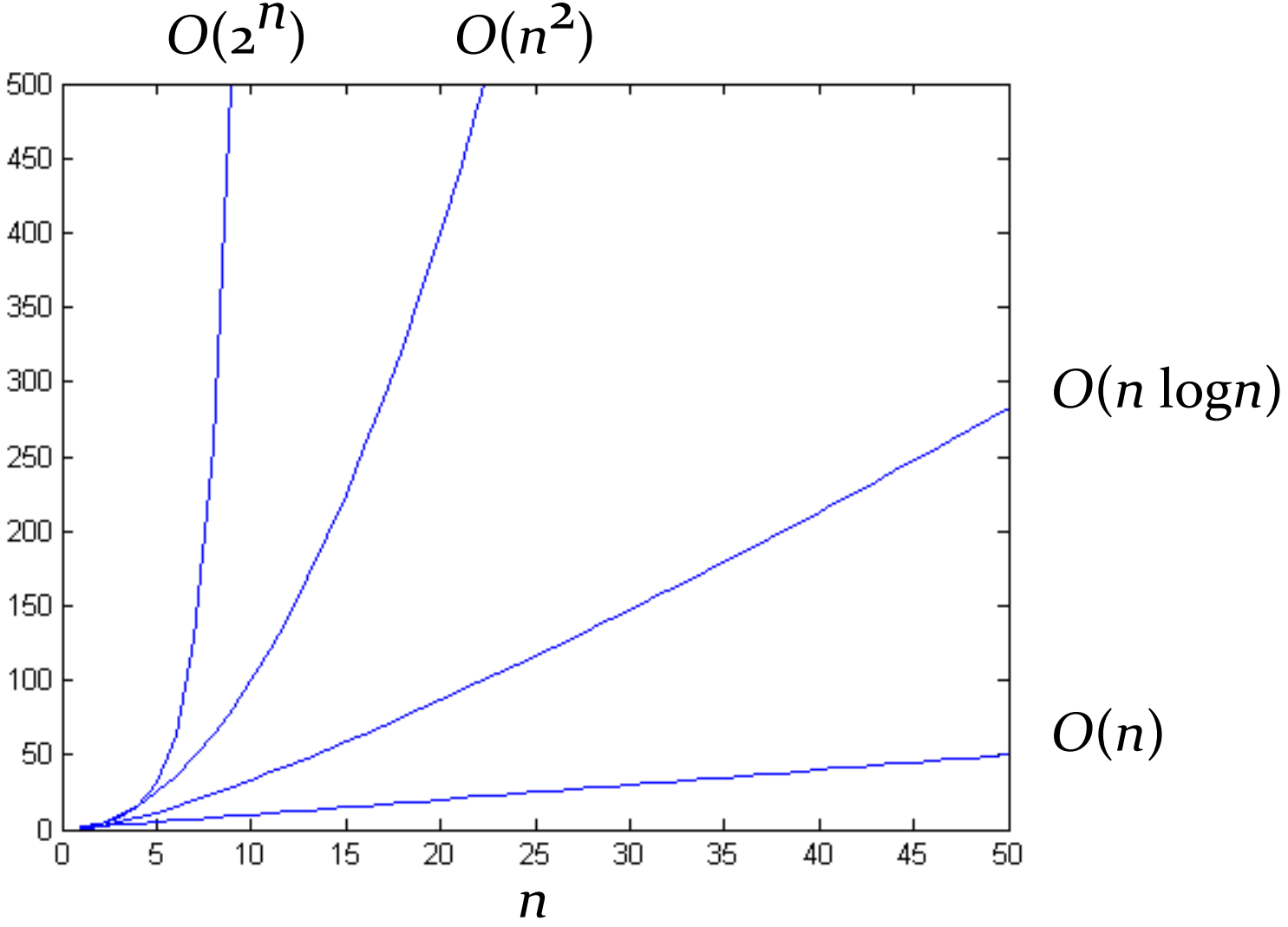
# $O(n^2)$

- $O(n^2)$ describes an algorithm whose runtime grows in proportion to the square of the size of the input
  - i.e., doubling the input size quadruples the runtime (approximately)
  - called quadratic complexity

# $O(2^n)$

- $O(2^n)$ describes an algorithm whose runtime grows exponentially with the size of the input
  - i.e., increasing the input size by 1 doubles the runtime (approximately)
  - called exponential complexity

# Comparing Rates of Growth



# Comments

- big-O complexity tells you something about the running time of an algorithm as the size of the input, *n*, approaches infinity
  - we say that it describes the limiting, or asymptotic, running time of an algorithm
- for small values of *n* it is often the case that a less efficient algorithm (in terms of big-O) will run faster than a more efficient one

# Proving correctness and terminaton

# Proving Correctness and Termination

- to show that a recursive method accomplishes its goal you must prove:
  1. that the base case(s) and the recursive calls are correct
  2. that the method terminates

# Proving Correctness

- to prove correctness:
  1. prove that each base case is correct
  2. assume that the recursive invocation is correct and then use the assumption to prove that what is done in the recursive case of the method is correct

# printItToo

```
public static void printItToo(String s, int n) {
  if (n == 0) {
    return;
  }
  else {
    System.out.print(s);
    printItToo(s, n - 1);
  }
}
```

# Correctness of printItToo

1. (prove the base case) If `n == 0` nothing is printed; thus the base case is correct.
2. Assume that `printItToo(s, n-1)` prints the string `s` exactly `(n - 1)` times. Then the recursive case prints the string `s` exactly `(n - 1)+1 = n` times; thus the recursive case is correct.

# Proving Termination

- to prove that a recursive method terminates:
  1. define the size of a method invocation; the size must be a non-negative integer number
  2. prove that each recursive invocation has a smaller size than the original invocation

# Termination of printItToo

1. **`printItToo(s, n)`** prints **`n`** copies of the string **`s`**; define the size of **`printItToo(s, n)`** to be **`n`**
2. The size of the recursive invocation **`printItToo(s, n-1)`** is **`n-1`** which is smaller than the original size **`n`**.

# countZeros

```
public static int countZeros(long n) {

  if(n == 0L) {  // base case 1
    return 1;
  }
  else if(n < 10L) {  // base case 2
    return 0;
  }

  boolean lastDigitIsZero = (n % 10L == 0);
  final long m = n / 10L;
  if(lastDigitIsZero) {
    return 1 + countZeros(m);
  }
  else {
    return countZeros(m);
  }
}
```

# Correctness of countZeros

1. (base cases) If the number has only one digit then the method returns `1` if the digit is zero and `0` if the digit is not zero; therefore, the base case is correct.
2. (recursive cases) Assume that `countZeros(n/10L)` is correct (it returns the number of zeros in the first `(d - 1)` digits of `n`). If the last digit in the number is zero, then the recursive case returns `1 +` the number of zeros in the first `(d - 1)` digits of `n`, otherwise it returns the number of zeros in the first `(d - 1)` digits of `n`; therefore, the recursive cases are correct.

# Termination of countZeros

1. Let the size of **`countZeros(n)`** be **`d`** the number of digits in the number **`n`**.
2. The recursive invocation is **`countZeros(n/10L)`**. The number of digits in **`n/10L`** is one less than the number of digits in **`n`**. Therefore, the size of the recursive invocation is **`(d-1)`** which is less than **`d`**.

# Implementing a list

# Data Structures

- data structures (and algorithms) are one of the foundational elements of computer science
- a data structure is a way to organize and store data so that it can be used efficiently
  - list – sequence of elements
  - set – a group of unique elements
  - map – access elements using a key
  - many more…

# Arrays

- a list can be implemented using an array
- in Java an array is a container object that holds a fixed number of values of a single type
- the length of an array is established when the array is created



https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html

# Arrays

- to declare an array you use the element type followed by an empty pair of square brackets

```
double[] collection;
// collection is an array of double values


collection = new double[10];
// collection is an array of 10 double values
```



https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html

# Arrays

- to create an array you use the new operator followed by the element type followed by the length of the array in square brackets

```
double[] collection;
// collection is an array of double values


collection = new double[10];
// collection is an array of 10 double values
```



https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html

# Arrays

- the number of elements in the array is stored in the public field named length

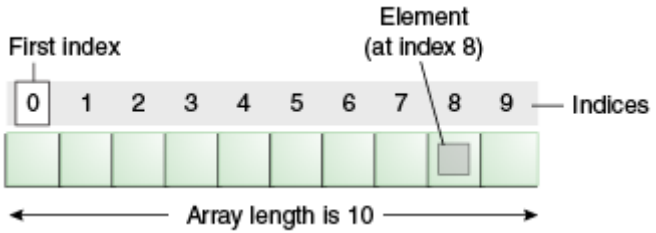
```
double[] collection;
// collection is an array of double values


collection = new double[10];
// collection is an array of 10 double values

int n = collection.length;
// the public field length holds the number of elements
```



https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html

# Arrays

- the values in an array are called elements
- the elements can be accessed using a zero-based index (similar to lists and strings)






https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html

# Arrays

- the elements can be accessed using a zero-based index (similar to lists and strings)

```
collection[0] = 100.0;
collection[1] = 100.0;
collection[2] = 100.0;
collection[3] = 100.0;
collection[4] = 100.0;
collection[5] = 100.0;
collection[6] = 100.0;
collection[7] = 100.0;
collection[8] = 100.0;
collection[9] = 100.0;  // set all elements to equal 100.0
collection[10] = 100.0; // ArrayIndexOutOfBoundsException
```



https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html

# Implementing a list using an array

- the *capacity* of a list is the maximum number of elements that the list can hold
  - note that the capacity is different than the size
    - the size of the list is the number of elements in the list whereas the capacity is the maximum number of elements that the list can hold
- the client can specify the capacity using a constructor
- if the clients tries to add more elements than the list can hold we have to increase the capacity

```
public class MyArrayList<T> implements List<T> {

  private Object[] elements;
  private int capacity;
  private int size;

  public MyArrayList(int capacity) {
    if (capacity < 1) {
      throw new
        IllegalArgumentException("capacity must be positive");
    }
    this.capacity = capacity;
    this.size = 0;
    this.elements = new Object[capacity];
  }
```

# Get and set

- to get and set an element at an index we simply get or set the element in the array at the given index
- because arrays are stored contiguously in memory, this operation has O(1) complexity (in theory)

```java
@Override
public T get(int index) {
  if (index < 0 || index >= this.size) {
    throw new IndexOutOfBoundsException("index: " + index);
  }
  return (T) this.elements[index];
}


@Override
public T set(int index, T element) {
  T oldElement = this.get(index);
  this.elements[index] = element;
  return oldElement;
}
```

# Adding to the end of the list

- when we add an element to the end of the list we have to check if there is room in the array to hold the new element
  - if not then we have to:
    1. make a new array with double the capacity of the old array
    2. copy all of the elements from the old array into the new array
    3. add the new element to the new array
- we say that adding to the end of an array-based list has O(1) amortized complexity

```java
@Override
public boolean add(T element) {
  if (this.size == this.capacity) {
    this.resize();
  }
  this.elements[this.size] = element;
  this.size++;
  return true;
}

private void resize() {
  int newCapacity = 2 * this.capacity;
  Object[] newElements = new Object[newCapacity];
  for (int i = 0; i < this.size; i++) {
    newElements[i] = this.elements[i];
  }
  this.capacity = newCapacity;
  this.elements = newElements;
}
```

# Inserting in the middle of an array

- when we insert an element into the middle of an array we have to:
  1. check if there is room in the array to hold the new element
     - resize if necessary
  2. shift the elements from the insertion index to the end of the array up by one index
  3. set the array at the insertion index to the new element

- Step 2 has O(n) complexity

```
@Override
public void add(int index, T element) {
  if (index < 0 || index > this.size) {
    throw new IndexOutOfBoundsException("index: " + index);
  }
  if (this.size == this.capacity) {
    this.resize();
  }
  for (int i = this.size - 1; i >= index; i--) {
    this.elements[i + 1] = this.elements[i];
  }
  this.set(index, element);
}
```

# Other list operations

- removing an element from the end of an array-based list takes O(1) time
- removing an element from the middle of an array-based list takes O(n) time
  - need to shift all elements from the removal index to the end of the array down by one index

- in most cases you should use an array-based list
- if you find yourself in a situation where most of your operations require inserting or removing elements near the front of a list then you should use a different kind of list

# Recursive Objects

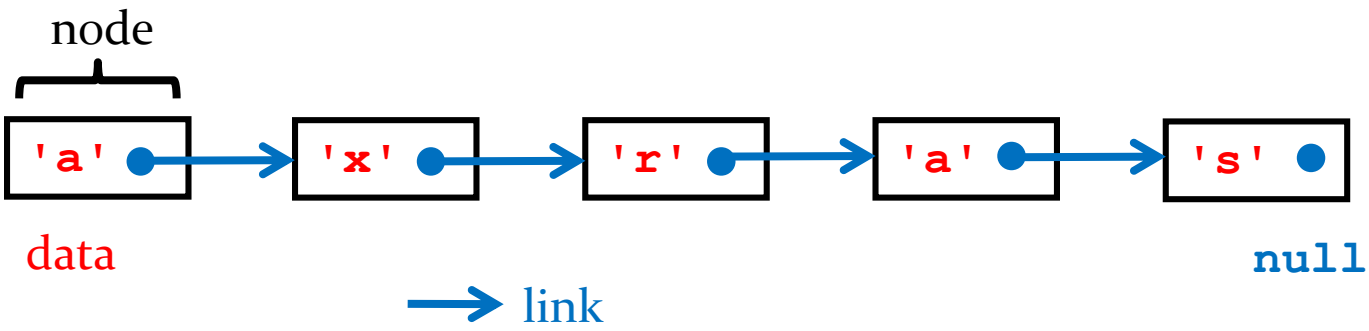
## Singly Linked Lists

# Recursive Objects

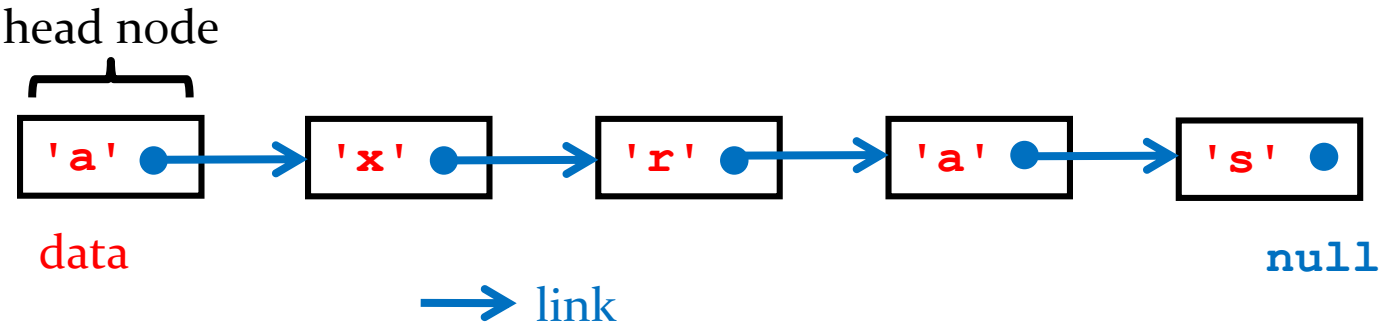
- an object that holds a reference to its own type is a recursive object
  - linked lists and trees are classic examples in computer science of objects that can be implemented recursively

# Singly Linked List

- a data structure made up of a sequence of nodes
- each node has
  - some data
  - a field that contains a reference (a *link*) to the **next** node in the sequence
- suppose we have a linked list that holds characters; a picture of our linked list would be:

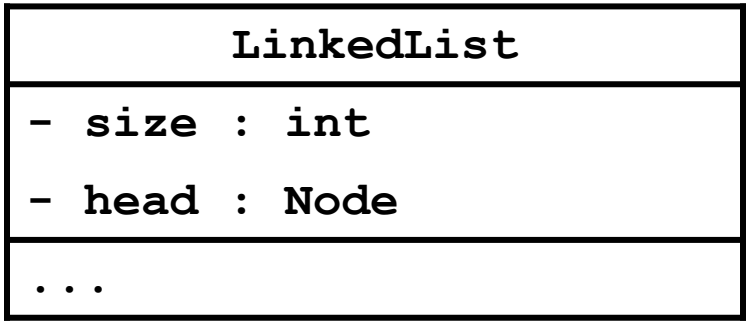


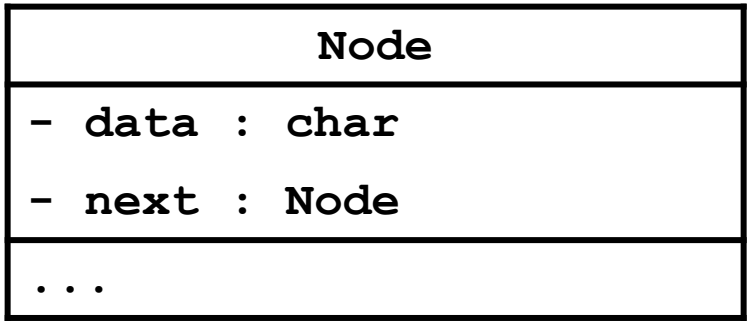
# Singly Linked List

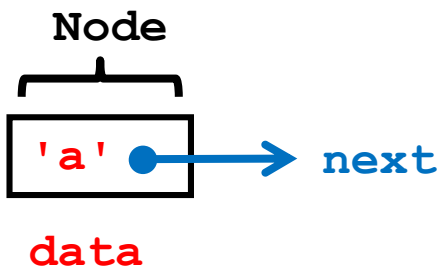



- the first node of the list is called the *head* node

# UML Class Diagram



# Node

- nodes are implementation details that the client does not need to know about
- **`LinkedList`** needs to be able to create nodes
  - i.e., needs access to a constructor
- if we create a separate **`Node`** class other clients can create nodes
  - no way to hide the constructor from every client except **`LinkedList`**
- Java allows the implementer to define a class inside of another class

```
public class LinkedList {

  private static class Node {
    private char data;
    private Node next;

    public Node(char c) {
      this.data = c;
      this.next = null;
    }
  }

 // ...
}
```

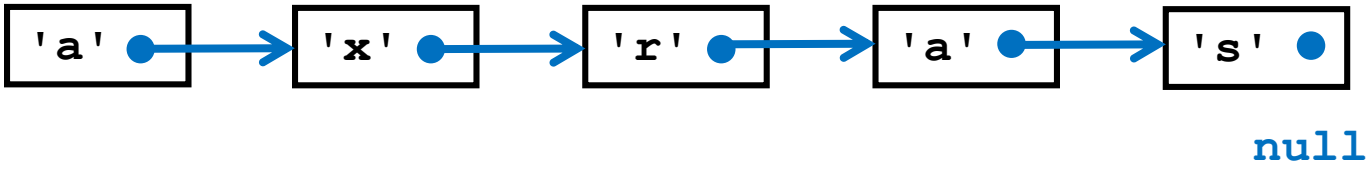
- **`Node`** is an *nested class*
- a nested class is a class that is defined inside of another class
- a *static nested class* behaves like a regular top-level class
  - does not have access to private members of the enclosing class
  - e.g., **`Node`** does not have access to the private fields of **`LinkedList`**
- a nested class is a member of the enclosing class
  - **`LinkedList`** has direct access to private features of **`Node`**

# LinkedList constructor

```
/**
 * Create a linked list of size 0.
 *
 */
public LinkedList() {
  this.size = 0;
  this.head = null;
}
```

# Creating a Linked List

- to create the following linked list:




```
LinkedList t = new LinkedList();
t.add('a');
t.add('x');
t.add('r');
t.add('a');
t.add('s');
```

# Add to end of list (recursive)

- methods of recursive objects can often be implemented with a recursive algorithm
  - notice the word "can"; the recursive implementation is not necessarily the most efficient implementation
- adding to the end of the list can be done recursively
  - base case: at the end of the list
    - i.e., **`next`** is **`null`**
    - create new node and append it to this link
  - recursive case: current link is not the last link
    - add to the end of **`next`**

```
/**
 * Adds the given character to the end of the list.
 *
 * @param c The character to add
 */
public void add(char c) {
  if (this.size == 0) {
    this.head = new Node(c);
  }
  else {
    LinkedList.add(c, this.head);
  }
  this.size++;
}
```

recursive method

```
/**
 * Adds the given character to the end of the list.
 *
 * @param c The character to add
 * @param node The node at the head of the current sublist
 */
private static void add(char c, Node node) {
  if (node.next == null) {
    node.next = new Node(c);
  }
  else {
    LinkedList.add(c, node.next);
  }
}
```

# Add to end of list (iterative)

- adding to the end of the list can be done iteratively

```
public void add(char c) {
  if (this.size == 0) {
    this.head = new Node(c);
  }
  else {
    Node n = this.head;
    while (n.next != null) {
      n = n.next;
    }
    n.next = new Node(c);
  }
  this.size++;
}
```

Starting from the head of the list, follow the links from node to node until you reach the last node.

# Getting an Element in the List

- a client may wish to retrieve the i*th* element from a list
  - the ability to access arbitrary elements of a sequence in the same amount of time is called *random access*
  - arrays support random access; linked lists do not
- to access the i*th* element in a linked list we need to sequentially follow the first ($i$ - 1) links


```
t.get(3)
```

link 0 link 1 link 2

  - takes $O(n)$ time versus $O(1)$ for arrays

# Getting an Element in the List

- validation?
- getting the *i*th element can be done recursively
  - base case:
    - `index == 0`
    - return the value held by the current link
  - recursive case:
    - get the element at `index - 1` starting from `next`

```
/**
 * Returns the item at the specified position
 * in the list.
 *
 * @param index index of the element to return
 * @return the element at the specified position
 * @throws IndexOutOfBoundsException if the index
 *         is out of the range
 *         {@code (index < 0 || index >= list size)}
 */
public char get(int index) {
  if (index < 0 || index >= this.size) {
    throw new IndexOutOfBoundsException("Index: " + index +
                                        ", Size: " + this.size);
  }
  return LinkedList.get(index, this.head);
}
```

recursive method

```
/**
 * Returns the item at the specified position
 * in the list.
 *
 * @param index index of the element to return
 * @param node The node at the head of the current sublist
 * @return the element at the specified position
 */
private static char get(int index, Node node) {
  if (index == 0) {
    return node.data;
  }
  return LinkedList.get(index - 1, node.next);
}
```

# Setting an Element in the List

- setting the *i*th element is almost exactly the same as getting the *i*th element

```
/**
 * Sets the element at the specified position
 * in the list.
 *
 * @param index index of the element to set
 * @param c new value of element
 * @throws IndexOutOfBoundsException if the index
 *         is out of the range
 *         {@code (index < 0 || index >= list size)}
 */
public void set(int index, char c) {
  if (index < 0 || index >= this.size) {
    throw new IndexOutOfBoundsException("Index: " + index +
                                        ", Size: " + this.size);
  }
  LinkedList.set(index, c, this.head);
}
```

recursive method

```
/**
 * Sets the element at the specified position
 * in the list.
 *
 * @param index index of the element to set
 * @param c new value of the element
 * @param node The node at the head of the current sublist
 */
private static void set(int index, char c, Node node) {
  if (index == 0) {
    node.data = c;
    return;
  }
  LinkedList.set(index - 1, c, node.next);
}
```