Arrays

Arrays

- 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

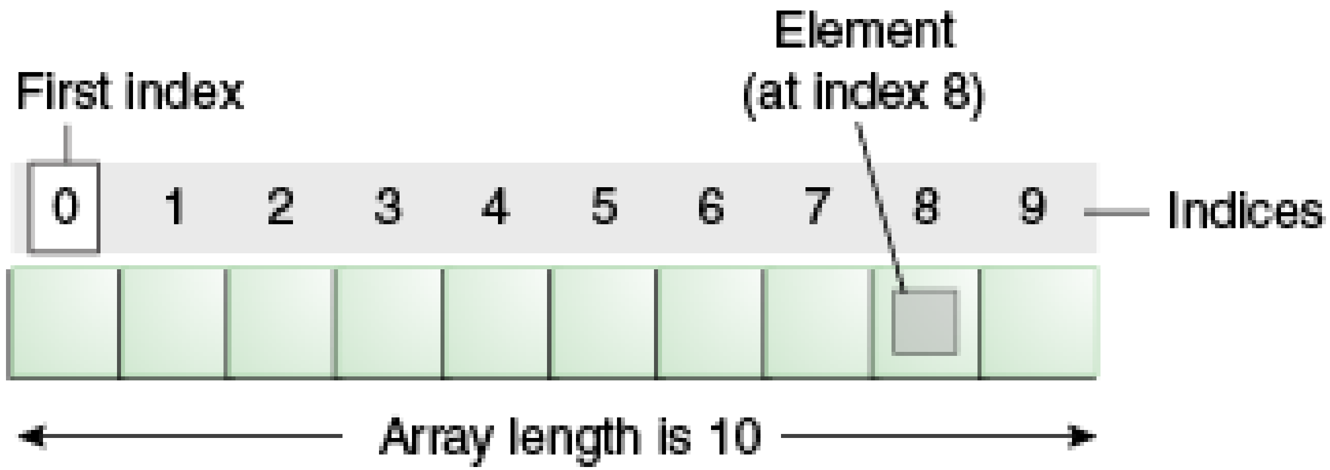
- the number of elements in the array is stored in the public field named length

```
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

Computational complexity

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)

Searching a list

```java
/**
 * Returns true if the specified array contains the specified value, and false
 * otherwise.
 *
 * @param arr
 *          an array to search
 * @param value
 *          a value to search for
 * @return true if the specified array contains the specified value, and false
 *         otherwise
 */
public static boolean contains(int[] arr, int value) {
  boolean result = false;
  for (int i = 0; i < arr.length; i++) {
    if (arr[i] == value) {
      result = true;
      break;
    }
  }
  return result;
}
```

size of problem, n, is the number of elements in the array **`arr`**

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
  - array access
  - return statement

Elementary instructions

- loops are technically more complicated, but for our purposes you can consider **`for(/* something */)`** to be a single elementary operation
  - but this is not true if the loop initialization, condition, or increment expression involves non-elementary operations

Estimating complexity

- count the number of elementary operations in each line of **`contains`**
  - discuss amongst yourselves…

Searching a list

```
public static boolean contains(int[] arr, int value) {
  boolean result = false;
  for (int i = 0; i < arr.length; i++) {
    if (arr[i] == value) {
      result = true;
      break;
    }
  }
  return result;
}
```

Estimating complexity

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

Searching a list

```java
public static boolean contains(int[] arr, int value) {
  boolean result = false;
  for (int i = 0; i < arr.length; i++) {
    if (arr[i] == value) {
      result = true;
      break;
    }
  }
  return result;
}
```

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
- multiply the number of elementary operations by the number of times each line runs

Total number of operations

- multiply the number of elementary operations by the number of times each line runs

Searching a list

```
public static boolean contains(int[] arr, int value) {
  boolean result = false;                        2 * 1
  for (int i = 0; i < arr.length; i++) {         1 * 1
    if (arr[i] == value) {                       3 * n
      result = true;                             1 * 1
      break;                                     1 * 1
    }
  }
  return result;                                 1 * 1
}
```

Running time

- the running time for **`contains`** is $f(n) = 3n + 6$

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 $f(n) = 3n + 6$ the constants 3 and 6 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 **contains** 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

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

Big-O notation

- Claim: $f(n) = 3n + 6 \in O(n)$
- Proof: $f(n) = 3n + 6, g(n) = n$

For $n > 1$, $f(n) > 0$ 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:

$$3n + 6 < Mn \text{ for all } n > m$$

For all $n > 1$ we have:

$$3n + 6 \leq 9n$$

$\therefore 3n + 6 < 9n$ for all $n > 1$ and $f(n) \in O(n)$

Big-O notation

- Proof 2: $f(n) = 3n + 6, g(n) = n$

For $n > 1$, $f(n) > 0$ 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:

$$3n + 6 < Mn \text{ for all } n > m$$

For $n > 1$ we have:

$$\frac{3n+6}{n} < \frac{3n+6n}{n} < \frac{9n}{n} < 9$$

$\therefore 3n + 6 < 9n$ for all $n > 1$ and $f(n) \in O(n)$

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 $n > 1$

$$\begin{aligned}
\frac{|3n^2 - n + 100|}{|n^2|} &< \frac{3n^2 \overset{\text{change to +}}{+} n + 100}{n^2} < \frac{3n^2 + \overset{\text{increase}}{n^2} + 100\overset{\text{increase}}{n^2}}{n^2} \\
&= \frac{104n^2}{n^2} \\
&= 104
\end{aligned}$$

$O(1)$

- $O(1)$ describes an algorithm that runs in constant time
  - i.e., the run time does not depend on the size of the input
  - examples:
    - determine if an integer is even or odd
    - **`get`** for **`ArrayList`**
    - **`contains`** for **`HashSet`**

$O(\log_2 n)$

- $O(\log_2 n)$ 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
  - examples:
    - **`Arrays.binarySearch`** (**`contains`** for a sorted array)
    - **`contains`** for **`TreeSet`**

$O(n)$

- $O(n)$ 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
  - examples:
    - finding the minimum or maximum value in an array or list
    - **`contains`** for an unsorted array

$O(n\log_2 n)$

- $O(n \log_2 n)$ 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
  - examples:
    - efficient sorting of an array or list

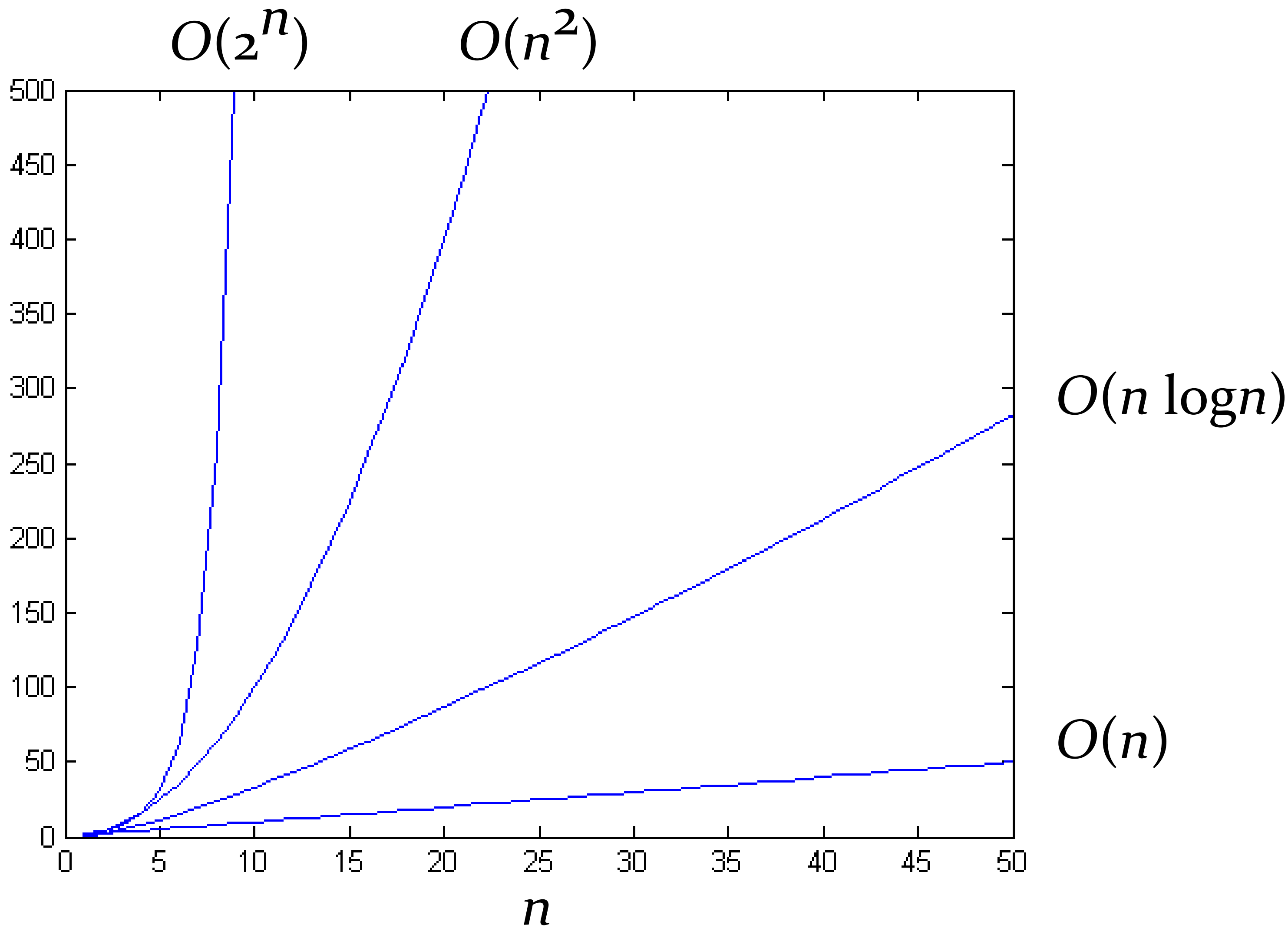
$O(n^2)$

- $O(n^2)$ 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
  - examples:
    - inefficient sorting of an array or list
    - checking if everything in one list is in another list

$O(2^n)$

- $O(2^n)$ 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
  - example:
    - trying to break a combination lock by trying every possible combination

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

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…

Implementing a list

- suppose that we wanted to implement our own list-of-strings class
  - we want to use an array to store the string references
- what public features should our class have?
  - discuss amongst yourselves here…

Implementing a list

- how does the choice of using an array affect the implementation of the list features?
  - discuss amongst yourselves here…

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

```java
public class StringList {

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

  // Initializes an empty list of strings having the given capacity.
  public StringList(int capacity) {
    if (capacity < 1) {
      throw new
        IllegalArgumentException("capacity must be positive");
    }
    this.capacity = capacity;
    this.size = 0;
    this.elements = new String[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
/**
 * Returns the string at the specified position in this list.
 *
 * @param index
 *            index of the string to return
 * @return the string at the specified position in this list
 * @throws IndexOutOfBoundsException
 *             if index is out of range (index is less than zero or
 *             index is greater than or equal to the size of this list)
 */
public String get(int index) {
  if (index < 0 || index >= this.size) {
    throw new IndexOutOfBoundsException("index: " + index);
  }
  return this.elements[index];
}
```

```java
/**
 * Replaces the string at the specified position in this list with the
 * specified string.
 *
 * @param index
 *            index of the element to replace
 * @param element
 *            string to be stored at the specified position
 * @return the string previously at the specified position
 */
public String set(int index, String element) {
  String 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
/**
 * Appends the specified string to the end of this list.
 *
 * @param element string to be appended to this list
 * @return true (consistent with java.util.List)
 */
public boolean add(T element) {
  if (this.size == this.capacity) {
    this.resize();
  }
  this.elements[this.size] = element;
  this.size++;
  return true;
}
```

```java
/**
 * Creates a new array twice the size of this.elements, and copies
 * the references from this.elements to the new array. Assigns the
 * new array to this.elements.
 */
private void resize() {
  int newCapacity = 2 * this.capacity;
  String[] newElements = new String[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

```java
/**
 * Inserts the specified string at the specified position in this list.
 * Shifts the string currently at that position (if any) and any subsequent
 * strings to the right (adds one to their indices).
 *
 * @param index
 *            index at which the specified element is to be inserted
 * @param element
 *            element to be inserted
 * @throws IndexOutOfBoundsException
 *             if index is out of range (index is less than zero or index is
 *             greater than or equal to the size of this list)
 */
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

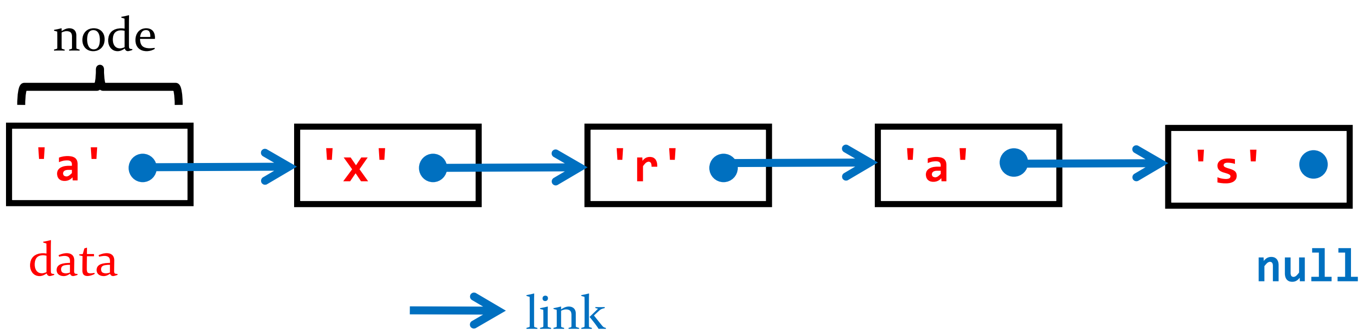
Singly Linked Lists

Recursive Objects

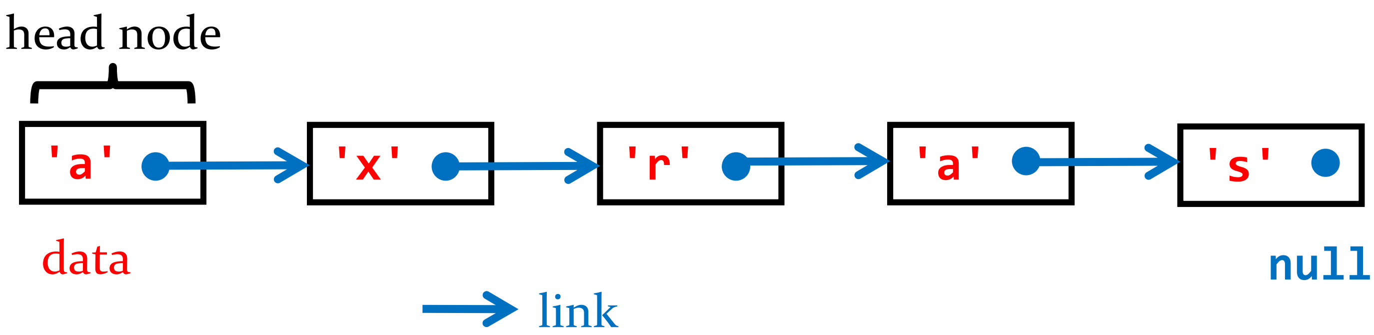
- an object that holds a reference to its own type is a recursive object
  - linked lists and trees are classic examples in computer science of objects that can be implemented recursively

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:

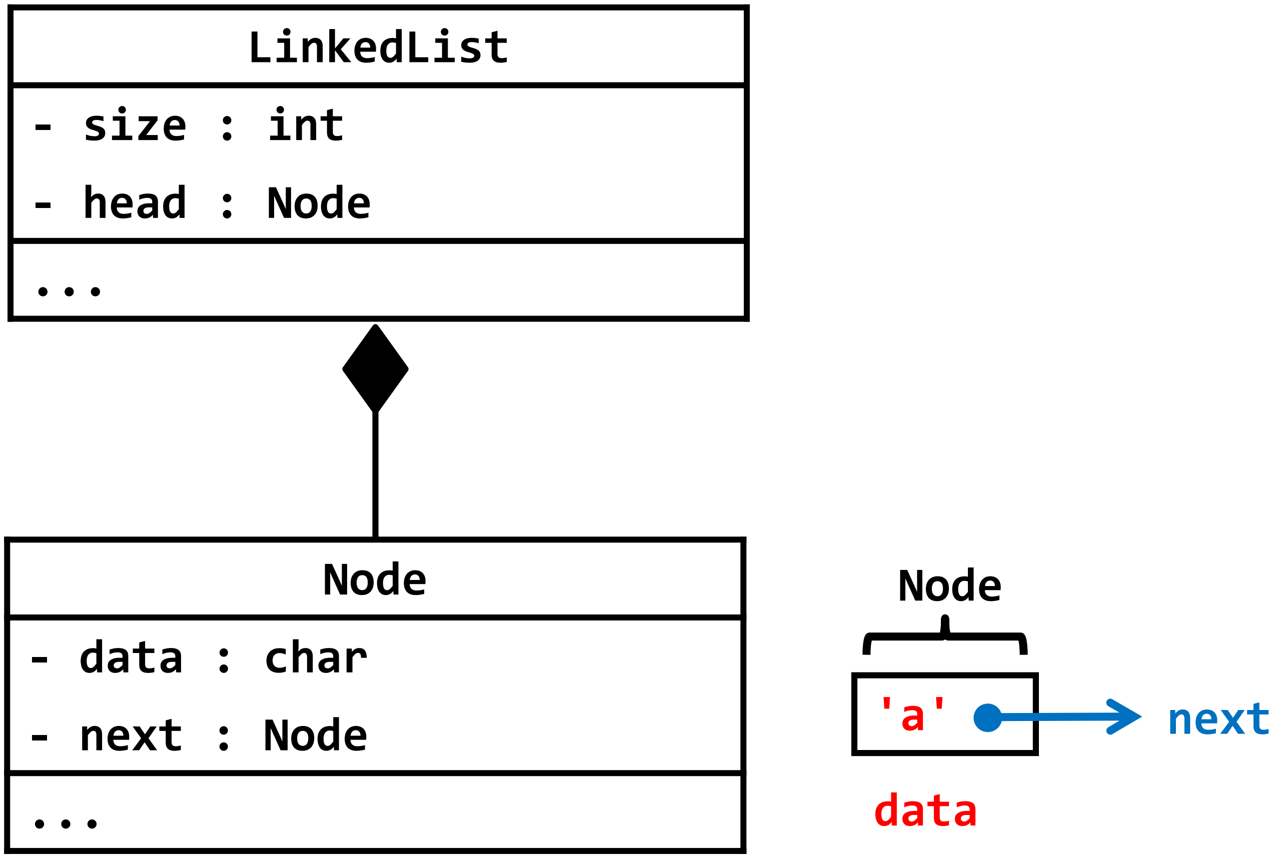


Singly Linked List



- 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

```java
public class LinkedCharList {

  /**
   * A class representing the internal nodes of the linked list.
   * A node is an aggregation of a data element and a link to the
   * next node in the sequence.
   *
   */
  public static class Node {
    private char data;
    private Node next;

    // see next slide for Node implementation
  }

}
```

```java
/**
 * Initialize this node to store the given data value and sets
 * the reference to the next node in the sequence to null.
 *
 * @param data
 *          the data element to store in this node
 */
public Node(char data) {
  this.data = data;
  this.next = null;
}
```

```java
/**
 * Returns the data element stored in this node.
 *
 * @return the data element stored in this node
 */
public char data() {
  return this.data;
}

/**
 * Returns the reference to the next node in the sequence after
 * this node.
 *
 * @return the reference to the next node in the sequence
 *         after this node
 */
public Node next() {
  return this.next;
}
```

Node details

- **`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 class
  - does not have access to private members of the enclosing class
    - **`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`**

Linked list fields

- at a minimum we need fields for:
  - size of the list (the number of elements in the list)
  - the first node of the list (the head node)

- do we need a field for the capacity?
  - discuss amongst yourselves…

```java
public class LinkedCharList {

  public static class Node {
    // see previous slides for Node implementation
  }

  private Node head;
  private int size;
```

No argument constructor

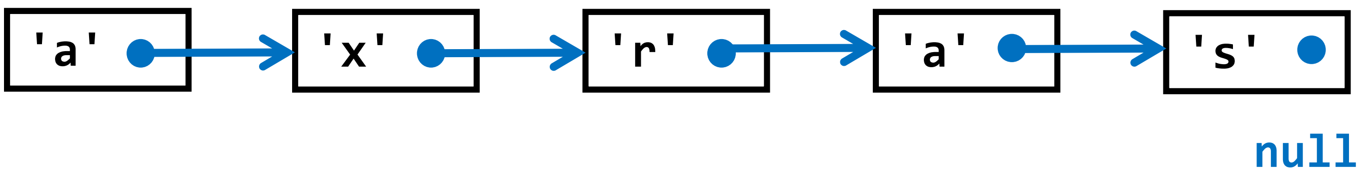
- the no argument constructor should create an empty list
  - the size of the list is equal to zero
  - there is no head node (because there is nothing in the list)

No argument constructor

```java
/**
 * Initialize the linked list to be empty (size == 0).
 */
public LinkedCharList() {
  this.head = null;
  this.size = 0;
}
```

Creating a linked list

- to create the following linked list:




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

Add to end of list

- to add an element to the end of the list we need to:
  - make a node to store the new element
  - get a reference to the current tail node
  - set the current tail node’s **`next`** field to point to the new node
  - increment **`size`**

```java
/**
 * Add an element to the end of this linked list.
 *
 * @param elem
 *          the element to add to the end of this linked list
 * @return true (to be consistent with java.util.Collection)
 */
public boolean add(char elem) {

  Node n = this.head;
  for (int i = 0; i < this.size; i++) {
    n = n.next;
  }
  n.next = new Node(elem);
  this.size++;

  return true;
}
```

What's wrong with this implementation?

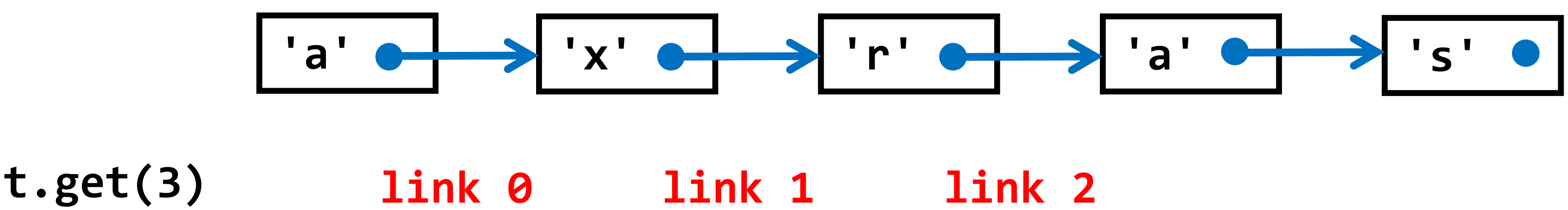
```java
/**
 * Add an element to the end of this linked list.
 *
 * @param elem
 *          the element to add to the end of this linked list
 * @return true (to be consistent with java.util.Collection)
 */
public boolean add(char elem) {

  Node n = this.head;
  for (int i = 0; i < this.size - 1; i++) {
    n = n.next;
  }
  n.next = new Node(elem);
  this.size++;
  return true;
}
```

What's wrong with this implementation?

```java
public boolean add(char elem) {
    if (this.head == null) {
      this.head = new Node(elem);
    }
    else {
      Node n = this.head;
      for (int i = 0; i < this.size - 1; i++) {
        n = n.next;
      }
      n.next = new Node(elem);
    }
    this.size++;
    return true;
  }
```

Getting an element in the list

- a client may wish to retrieve the *ith* 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 *ith* element in a linked list we need to sequentially follow the first (i - 1) links



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

Getting an element in the list

- to get an element from the list we need to:
  - validate the index
  - get a reference to the node at the specified index
  - return the **`data`** of the node

```java
/**
  * Get the element stored at the given index in this linked list.
  *
  * @param index
  *          the index of the element to get
  * @return the element stored at the given index in this linked list
  * @throws IndexOutOfBoundsException
  *           if (index < 0) || (index > size)
  */
 public char get(int index) {
   if (index < 0 || index >= this.size) {
     throw new IndexOutOfBoundsException();
   }
   Node n = this.head;
   for (int i = 0; i < index - 1; i++) {
     n = n.next;
   }
   return n.data;
 }
```

What's wrong with this implementation?

```java
/**
 * Get the element stored at the given index in this linked list.
 *
 * @param index
 *          the index of the element to get
 * @return the element stored at the given index in this linked list
 * @throws IndexOutOfBoundsException
 *           if (index < 0) || (index > size)
 */
public char get(int index) {
  if (index < 0 || index >= this.size) {
    throw new IndexOutOfBoundsException();
  }
  Node n = this.head;
  for (int i = 0; i < index; i++) {
    n = n.next;
  }
  return n.data;
}
```

Setting an element in the list

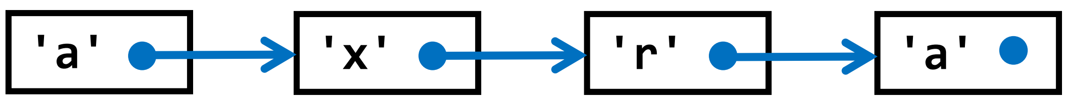
- setting the *i*th element is almost exactly the same as getting the *i*th element:
  - validate the index
  - get a reference to the node at the specified index
  - remember the current **`data`** of the node
  - set the **`data`** of the node
  - return the old **`data`** of the node

```java
/**
 * Sets the element stored at the given index in this linked list. Returns the
 * old element that was stored at the given index.
 *
 * @param index
 *          the index of the element to set
 * @param elem
 *          the element to store in this linked list
 * @return the old element that was stored at the given index
 * @throws IndexOutOfBoundsException
 *           if (index < 0) || (index > size)
 */
public char set(int index, char elem) {
  if (index < 0 || index >= this.size) {
    throw new IndexOutOfBoundsException();
  }
  Node n = this.head;
  for (int i = 0; i < index; i++) {
    n = n.next;
  }
  char oldData = n.data;
  n.data = elem;
  return oldData;
}
```

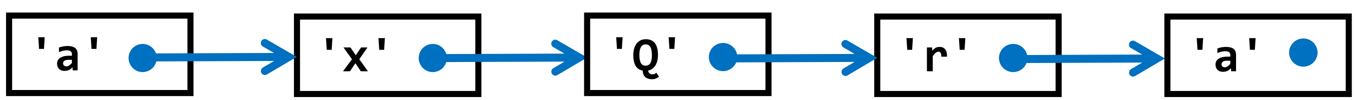
Adding in the middle of a list

- a client may wish to add an element at the *ith* index of a list

`t`




```
t.add(2, 'Q');
```

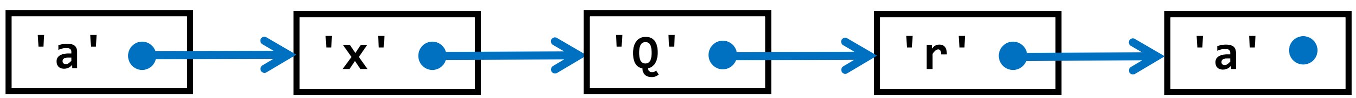


- what steps are required?
  - discuss amongst yourselves here…

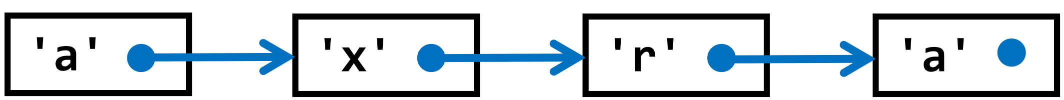
Removing an element

- a client may wish to remove an element at the *ith* index of a list

`t`



`t.remove(2)`



- what steps are required?
  - discuss amongst yourselves here…