Recursive Objects

Singly Linked Lists

Recursive Objects

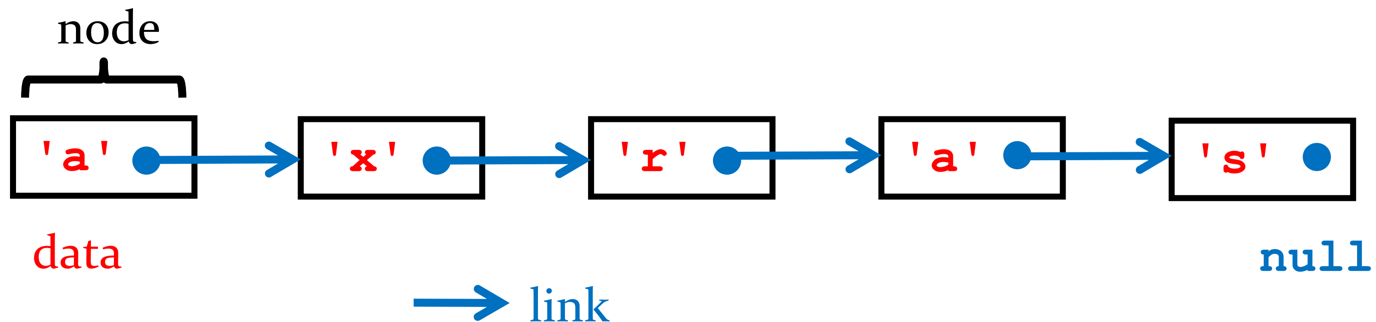
- an object that holds a reference to its own type is a recursive object
  - linked lists and trees are classic examples in computer science of objects that can be implemented recursively

Data Structures

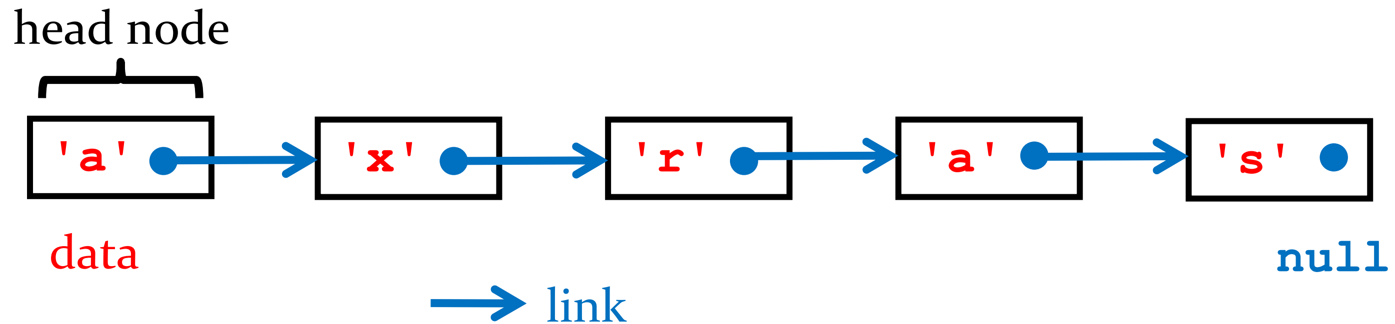
- data structures (and algorithms) are one of the foundational elements of computer science
- a data structure is a way to organize and store data so that it can be used efficiently
  - list – sequence of elements
  - set – a group of unique elements
  - map – access elements using a key
  - many more…

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:

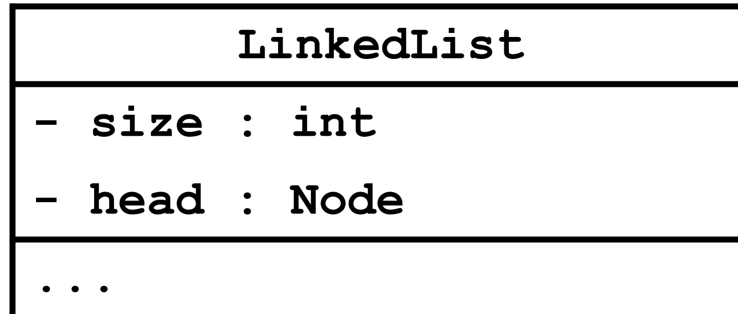


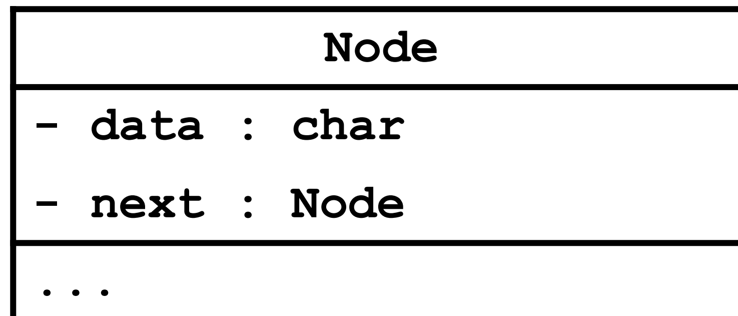
Singly Linked List

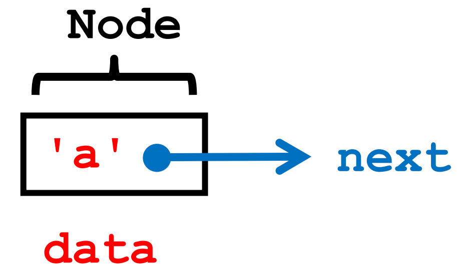


- the first node of the list is called the *head* node

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 LinkedList {

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

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

 // ...
}
```

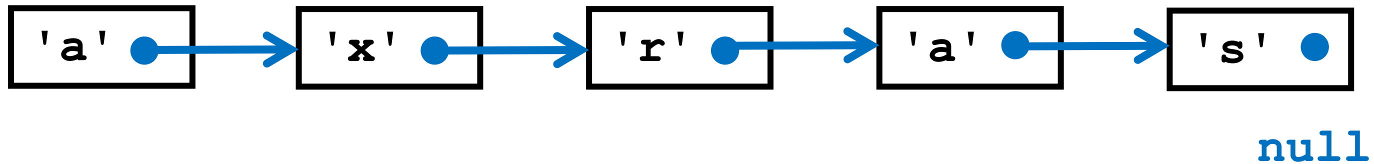
- **`Node`** is an *nested class*
- a nested class is a class that is defined inside of another class
- a *static nested class* behaves like a regular top-level class
  - does not have access to private members of the enclosing class
  - e.g., **`Node`** does not have access to the private fields of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

```java
/**
 * 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

```java
/**
 * 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)

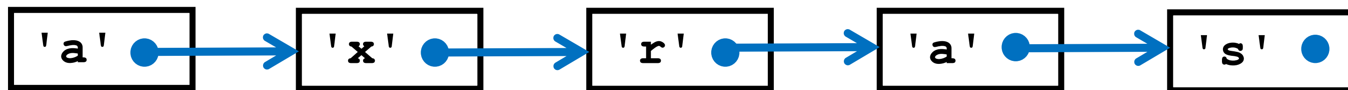
- adding to the end of the list can be done iteratively

```
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 *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




```
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`**

```java
/**
 * 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

```java
/**
 * 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

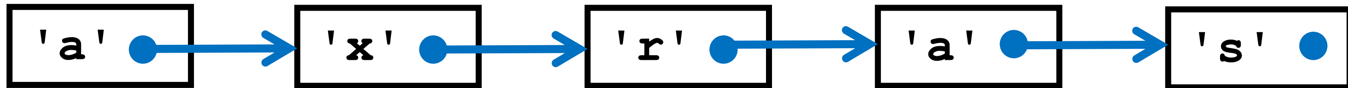
```java
/**
 * Sets the element at the specified position
 * in the list.
 *
 * @param index index of the element to set
 * @param c new value of element
 * @throws IndexOutOfBoundsException if the index
 *         is out of the range
 *         {@code (index < 0 || index >= list size)}
 */
public void 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

```java
/**
 * 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);
}
```

toString

- finding the string representation of a list can be done recursively



  - the string is
    `"[a, x, r, a, s]`
  - the string is
    `"[" + "a, " + toString(`*`the list['x', 'r', 'a', 's']`*`)`

toString

- base case: **`next`** is **`null`**
  - return the value of the link as a string + **`"]"`**
- recursive case: current link is not the last link
  - return the value of the link as a string + `", "` + the rest of the list as a string

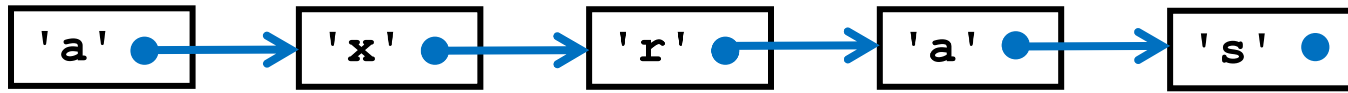
```
public String toString() {
  if (this.size == 0) {
    return "[]";
  }
  return "[" + LinkedList.toString(this.head);
}

private static String toString(Node n) {
  if (n.next == null) {
    return n.data + "]";
  }
  String s = n.data + ", ";
  return s + LinkedList.toString(n.next);
}
```

recursive method

Finding an element in the list

- often useful to ask if a list contains a particular element
- worst case: must visit every element of the list



  - e.g., `t.contains('z')`

Finding an element in the list

- contains can be solved recursively
  - base case: found the character we are looking for
    - i.e., node **`data`** is equal to the character we are searching for
      - return true
  - base case: at the end of the list
    - i.e., node **`next`** is **`null`**
      - return false
  - recursive case: have not found the character we are searching for and not at the end of the list
    - search the sublist starting at **`node.next`**
      - return result

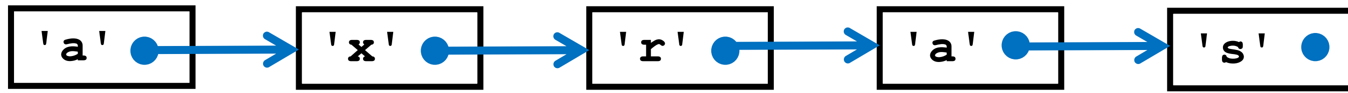
```
/**
 * Returns <code>true</code> if this list contains the specified
   element.
 *
 * @param c element to search for
 * @return <code>true</code> if this list contains the
 * specified element
 */
public boolean contains(char c) {
  if (this.size == 0) {
      return false;
    }
  return LinkedList.contains(c, this.head);
}
```

recursive method

```java
/**
 * Returns <code>true</code> if this list contains the specified
   element.
 *
 * @param c element to search for
 * @param node the node at the head of the current sublist
 * @return <code>true</code> if this list contains the
 * specified element
 */
private static boolean contains(char c, Node node) {
  if (node.data == c) {
    return true;
  }
  if (node.next == null) {
    return false;
  }
  return LinkedList.contains(c, node.next);
}
```

Finding an element in the list

- closely related to contains is finding the index of an element in the list
- worst case: must visit every element of the list



  - e.g., `t.indexOf('s')`

Finding an element in the list

- **`indexOf`** can be solved recursively
  - base case: found the character we are looking for
    - i.e., node **`data`** is equal to the character we are searching for
      - return 0
  - base case: at the end of the list
    - i.e., node **`next`** is **`null`**
      - return -1
  - recursive case: have not found the character we are searching for and not at the end of the list
    - search the sublist starting at **`node.next`**
      - return 1 + result

```
/**
 * Returns the index of the first occurrence of the
 * specified element in this list, or -1 if this list
 * does not contain the element.
 *
 * @param c
 *          element to search for
 * @return the index of the first occurrence of the
 *         specified element in this list, or -1 if this
 *         list does not contain the element
 */
public int indexOf(char c) {
  if (this.size == 0) {
    return -1;
  }
  return LinkedList.indexOf(c, this.head);
}
```

recursive method

```
// returns the index of the node containing c relative to node n

private static int indexOf(char c, Node n) {
  if (n.data == c) {
    return 0;
  }
  if (n.next == null) {
    return -1;
  }
  int i = LinkedList.indexOf(c, n.next);
  if (i == -1) {
    return -1;
  }
  return 1 + i;
}
```