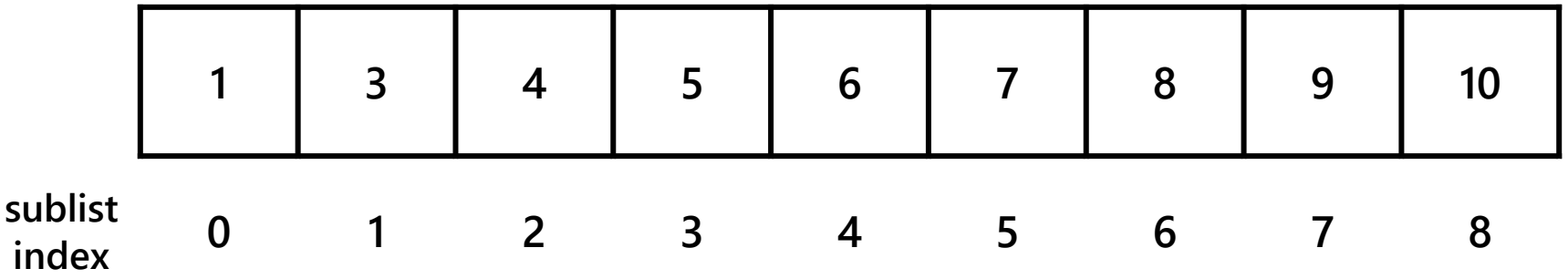
Binary Search

- one reason that we care about sorting is that it is much faster to search a sorted list compared to sorting an unsorted list
- the classic algorithm for searching a sorted list is called *binary search*
- to search a list of size n for a value v:
  - look at the element e at index $\left(\frac{n}{2}\right)$
  - if $e > v$ recursively search the sublist to the left
  - if $e < v$ recursively search the sublist to the right
  - if $e == v$ then done

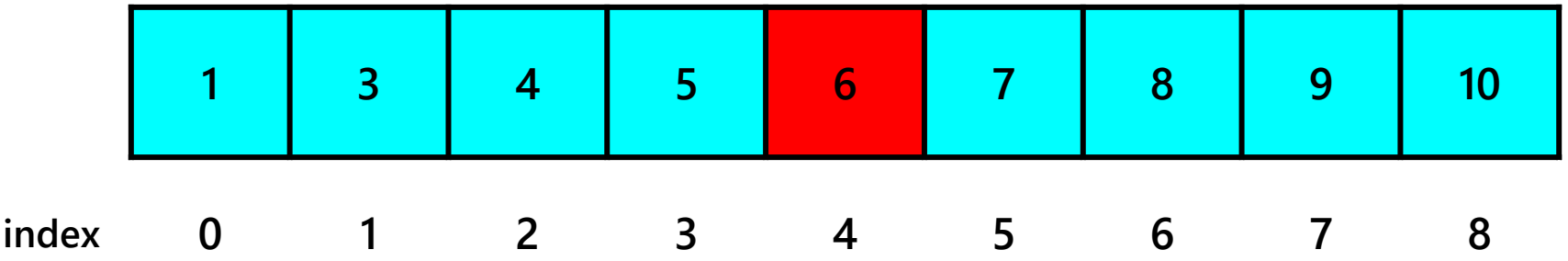
Binary Search

- consider the sorted list of size $n = 9$



Binary Search

- search for $v = 3$



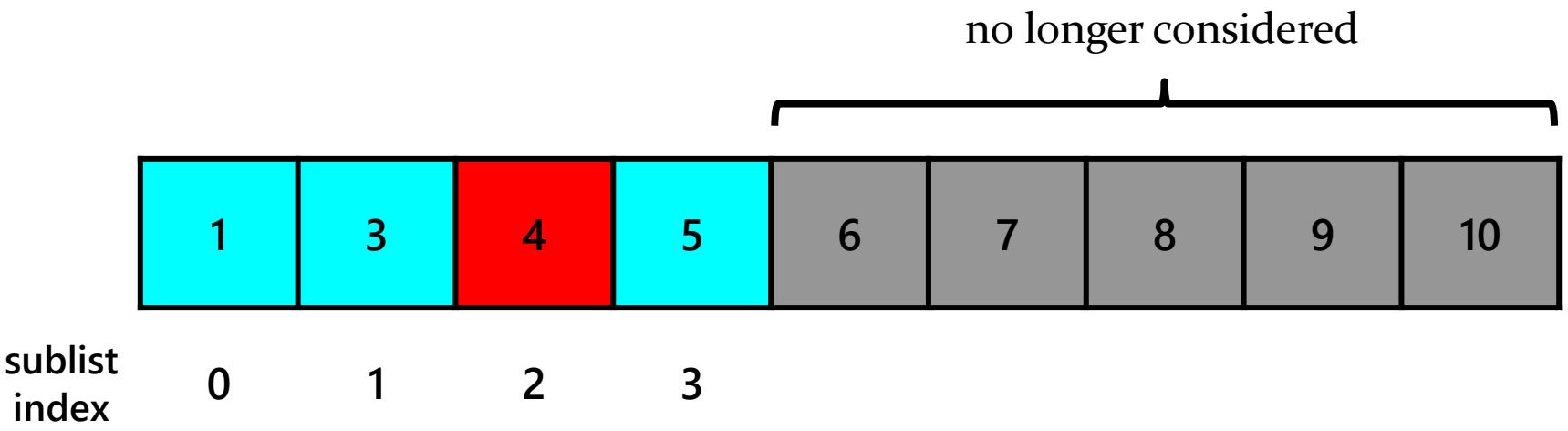
$$mid = \frac{9}{2} = 4$$

$$e = 6$$

$v < e$, recursively search the left sublist

Binary Search

- search for $v = 3$



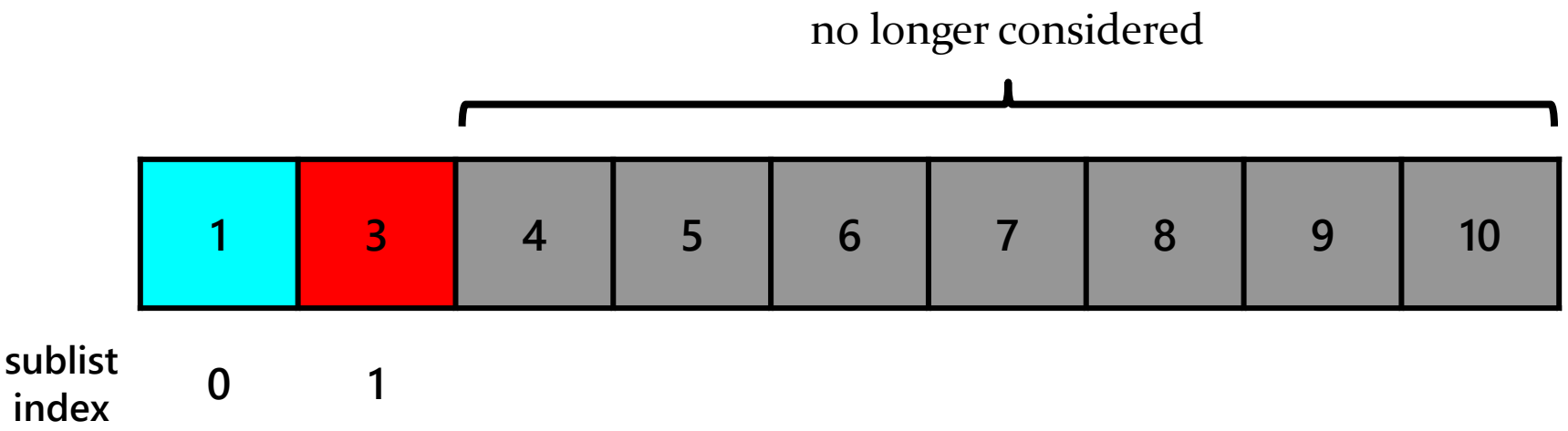
$$mid = \frac{4}{2} = 2$$

$$e = 4$$

$v < e$, recursively search the left sublist

Binary Search

- search for $v = 3$



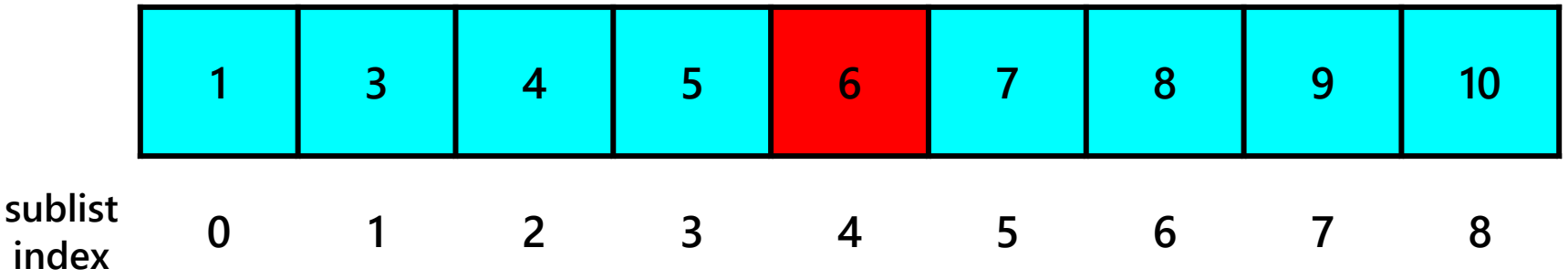
$$mid = \frac{2}{2} = 1$$

$$e = 3$$

$v == e$, done

Binary Search

- search for $v = 2$



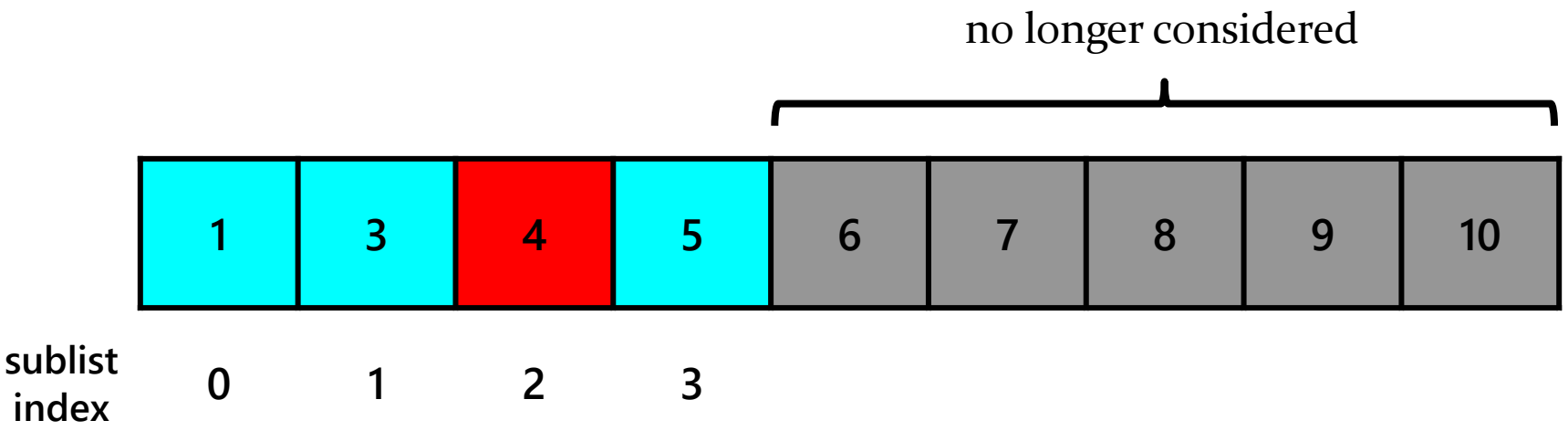
$$mid = \frac{9}{2} = 4$$

$$e = 6$$

$v < e$, recursively search the left sublist

Binary Search

- search for $v = 2$



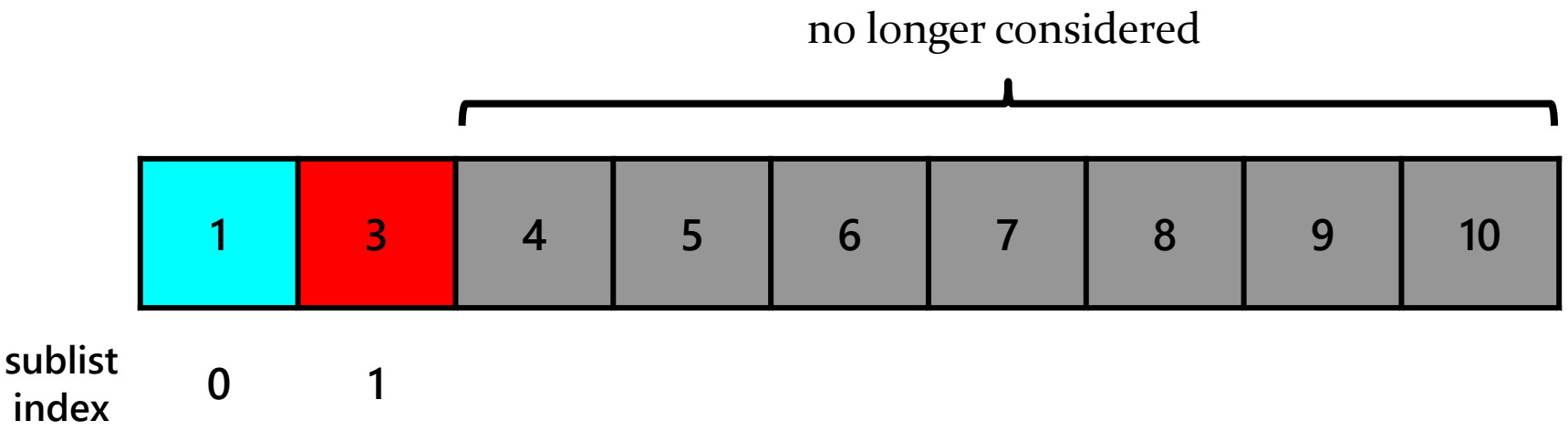
$$mid = \frac{4}{2} = 2$$

$e = 4$

$v < e$, recursively search the left sublist

Binary Search

- search for $v = 2$



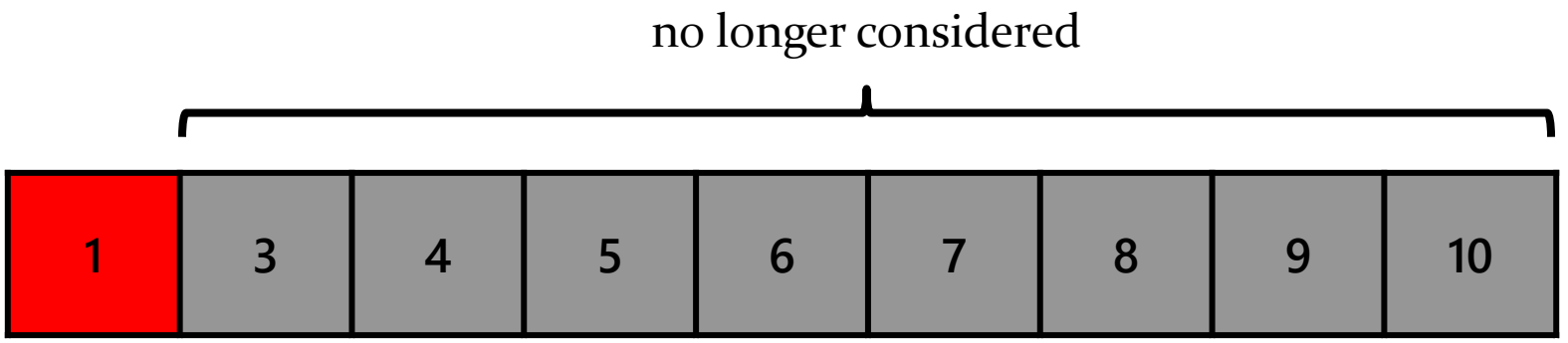
$$mid = \frac{2}{2} = 1$$

$e = 3$

$v < e$, recursively search the left sublist

Binary Search

- search for $v = 2$



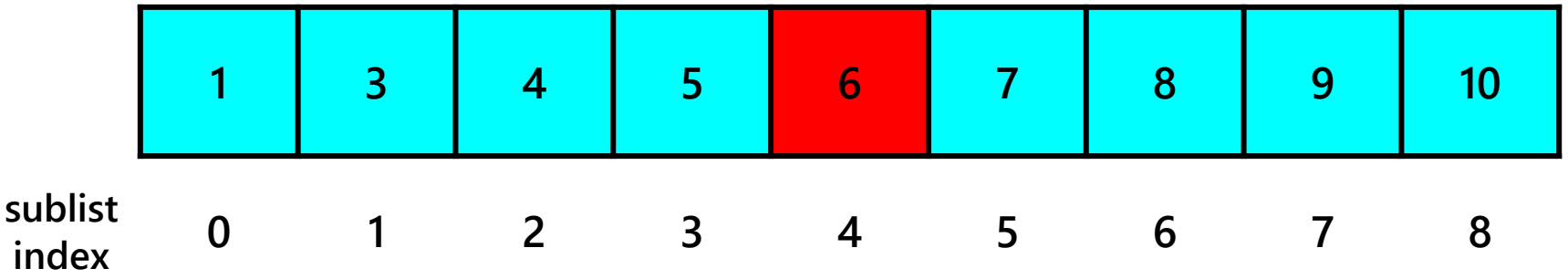
$$mid = \frac{1}{2} = 0$$

$e = 1$

$v > e$, recursively search the right sublist; right sublist is empty, done

Binary Search

- search for $v = 9$



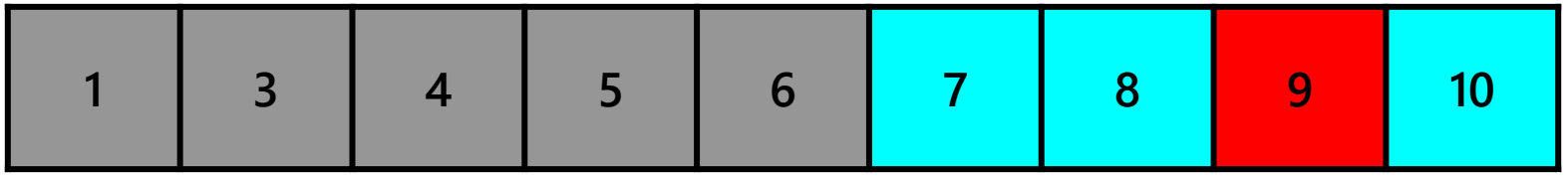
$$mid = \frac{9}{2} = 4$$

$$e = 6$$

$v > e$, recursively search the right sublist

Binary Search

- search for $v = 9$



sublist index

0 1 2 3

$$mid = \frac{4}{2} = 2$$

$$e = 9$$

$v == e$, done

```java
/**
 * Searches a sorted list of integers for a given value using binary search.
 *
 * @param v the value to search for
 * @param t the list to search
 * @return true if v is in t, false otherwise
 */
public static boolean contains(int v, List<Integer> t) {
  if (t.isEmpty()) {
    return false;
  }
  int mid = t.size() / 2;
  int e = t.get(mid);
  if (e == v) {
    return true;
  }
  else if (v < e) {
    return Recursion.contains(v, t.subList(0, mid));
  }
  else {
    return Recursion.contains(v, t.subList(mid + 1, t.size()));
  }
}
```

Binary Search

- what is the recurrence relation?
- what is the big-O complexity?

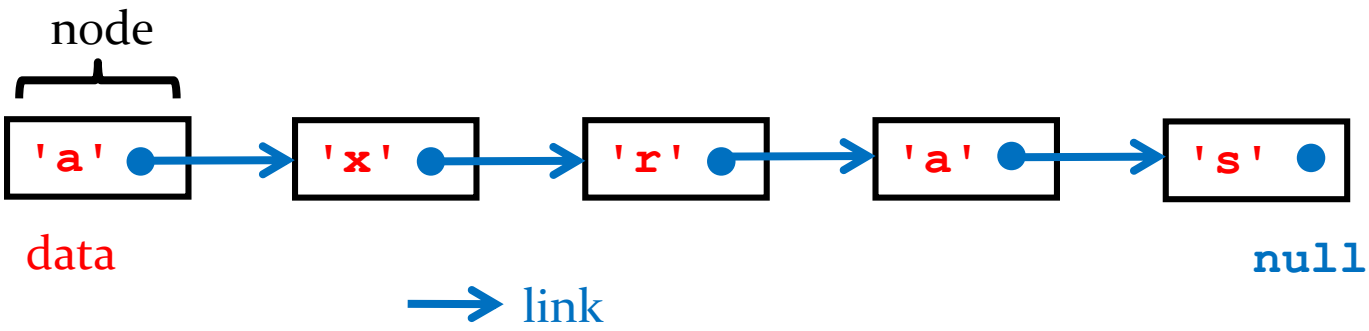
Revisiting Linked List

Recursive Objects

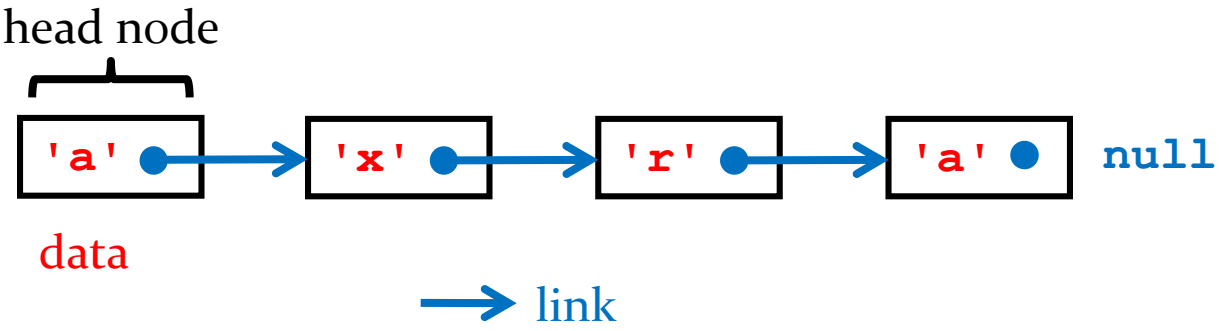
- an object that holds a reference to its own type is a recursive object
  - linked lists and trees are classic examples in computer science of objects that can be implemented recursively

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:



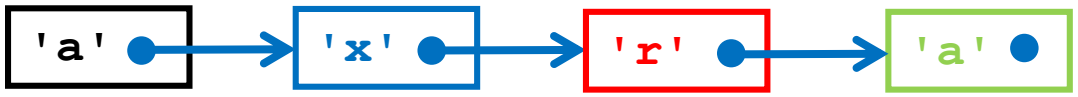
Singly Linked List



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

Linked List

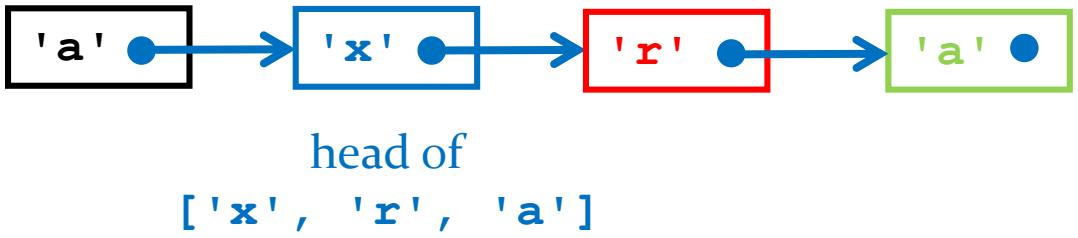
- each node can be thought of as the head of a smaller list



head of
`['a', 'x', 'r', 'a']`

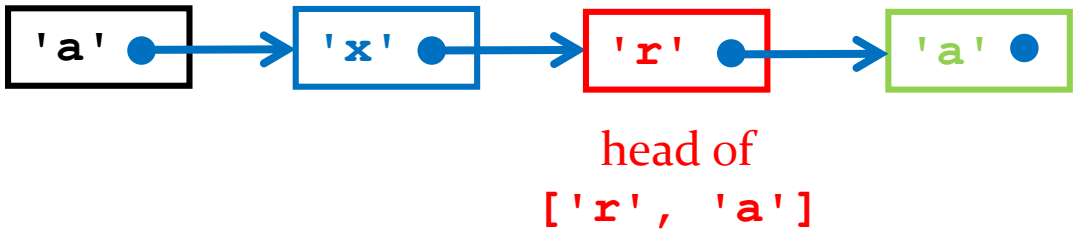
Linked List

- each node can be thought of as the head of a smaller list



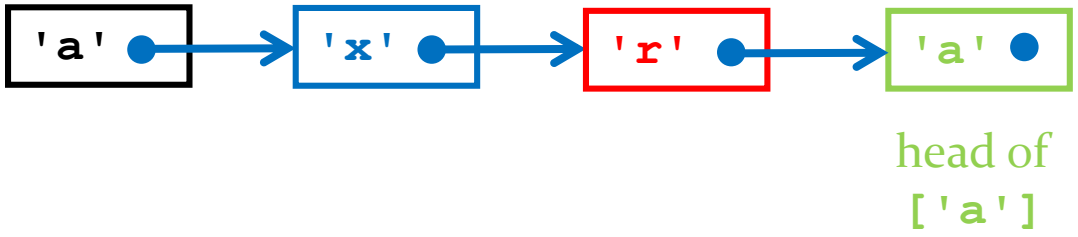
Linked List

- each node can be thought of as the head of a smaller list



Linked List

- each node can be thought of as the head of a smaller list



Linked List

- the recursive structure of the linked list suggests that algorithms that operate on a linked list can be implemented recursively
  - e.g., **`getNode(int index)`** from Lab 5

```java
/**
 * Returns the node at the given index.
 *
 * <p>
 * NOTE: This method is extremely useful for implementing many of the methods
 * of this class, but students should try to use this method only once in each
 * method.
 *
 * <p>
 * NOTE: This method causes a privacy leak and would not normally be
 * part of the public API; however, it is useful for testing purposes.
 *
 * @param index
 *            the index of the node
 * @return the node at the given index
 * @throws IndexOutOfBoundsException
 *             if index is less than 0 or greater than or equal the size of this
 *             list
 */
public Node getNode(int index) {
    this.checkIndex(index);
    return LinkedIntList.getNodeImpl(this.head, index);  // private static method
}
```

```
/**
 * Returns the node located at the specified index in the
 * list with specified head node.
 *
 * @param head the head node of a linked list
 * @param index the index of the element
 * @return the node located index elements from the specified node
 */
private static Node getNodeImpl(Node head, int index) {
```

- base case(s)?
- recursive case?
- precondition(s)?

```
}
```

```java
/**
 * Returns the node located at the specified index in the
 * list with specified head node.
 *
 * @param head the head node of a linked list
 * @param index the index of the element
 * @return the node located index elements from the specified node
 */
private static Node getNodeImpl(Node head, int index) {
  if (index == 0) {
    return head;
  }
  return LinkedIntList.getNodeImpl(head.getNext(), index - 1);
}
```

Linked List

- recursive version of **`contains`**

```java
/**
 * Returns true if this list contains the specified element,
 * and false otherwise.
 *
 * @param elem the element to search for
 * @return true if this list contains the specified element,
 * and false otherwise
 */
public boolean contains(int elem) {
  if (this.size == 0) {
    return false;
  }
  return LinkedIntList.contains(this.head, elem);
}
```

```java
/**
 * Returns true if the linked list with the specified head node contains
 * the specified element, and false otherwise.
 *
 * @param head the head node
 * @param elem the element to search for
 * @return true if the linked list with the specified head node contains
 * the specified element, and false otherwise
 */
private static boolean contains(Node head, int elem) {



}
```

- base case(s)?
- recursive case?
- precondition(s)?

```java
/**
 * Returns true if the linked list with the specified head node contains
 * the specified element, and false otherwise.
 *
 * @param head the head node
 * @param elem the element to search for
 * @return true if the linked list with the specified head node contains
 * the specified element, and false otherwise
 */
private static boolean contains(Node head, int elem) {
  if (head.getData() == elem) {
    return true;
  }
  if (head.getNext() == null) {
    return false;
  }
  return LinkedIntList.contains(head.getNext(), elem);
}
```