Recursion for Data Structures

CPSC 219: Introduction to Computer Science for Multidisciplinary Studies II
Fall 2023

Jonathan Hudson, Ph.D.
Instructor
Department of Computer Science
University of Calgary

Friday, 10 November 2023

UNIVERSITY OF CALGARY

Recursion

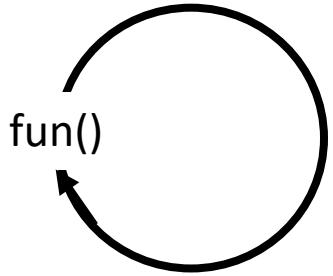
- Definition:
  - See Recursion
  - Defining something in terms of itself
    - Generally using a smaller or simpler version
- Recursive Function
  - A function that calls itself

Recursion

- A programming technique whereby a function calls itself either directly or indirectly

```
def fun ():
    :
    fun ()
    :
```

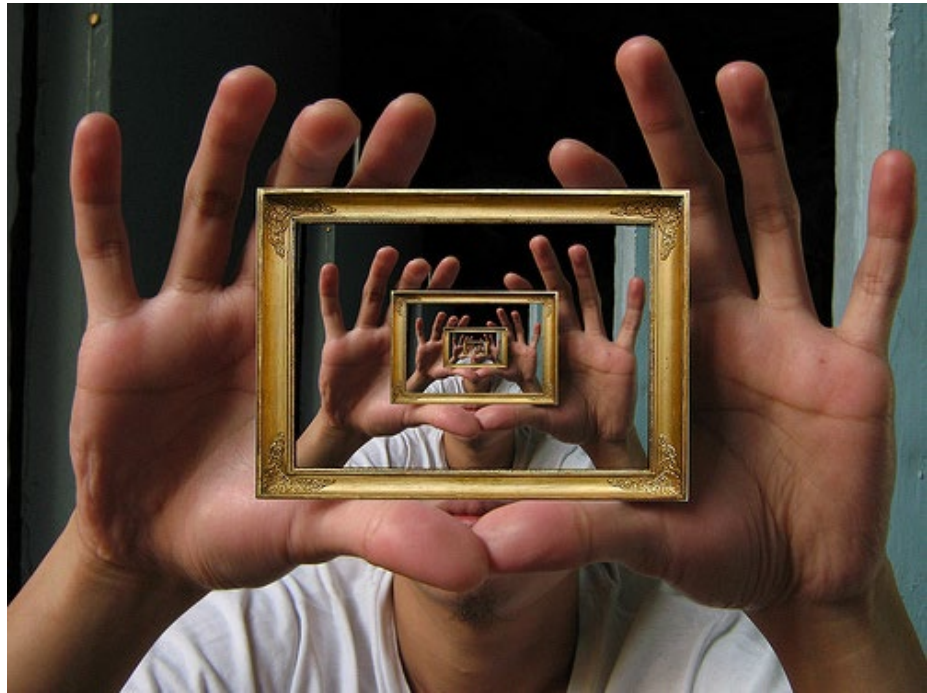
Directly




```
def fun1 ():
    fun2 ()

def fun2 ():
    fun1 ()
```

indirectly (mutual recursive)



Tail Recursion

- a tail call is a subroutine call performed as the final action of a function (Ex. A final return statement function call)
- If a tail call calls the function itself (as the last thing the function does), then this function is called tail recursive

```
#Not tail-recursive
def fact(n):

    if (n == 0):
        return 1

    return n * fact(n-1)
```

```
#Tail recursive
def fact(n, a = 1):

    if (n == 1):
        return a

    return fact(n - 1, n * a)
```

Tail Recursion

- Typically when a function is called a record of memory (stack frame) is stored to track state from the previous function call
- But a tail recursive function doesn’t need anything from before the final function call
- Modern compilers can identify this and reduce the process into an iterative loop for you. Although this is not guaranteed.
  - Java is a still a procedural language, there is a class of languages called functional languages which will often guarantee this behaviour
- Neither Java or Python optimizes tail recursion

Factorial

A Small Example - Factorial

Computing n factorial (n!):

- Defined as the result of multiplying all numbers from n to 1 for all n>0. If n == 0, then result is 1.

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1)! & n > 0 \end{cases}$$

- 0! → 1
- 1! → 1*1 = 1
- 2! → 2*1*1 = 2
- 3! → 3*2*1*1 = 6

A Small Example

Recursive Definition!

- Another solution
  - BASE CASE, 0! == 1
  - RECURSIVE CASE, n! == n * (n-1)!

A Small Example - Factorial

```
public static int factorial(int n){
    if(n < 0){
        throw new IllegalArgumentException("...");
    }
    if (n == 0){
        return 1;
    }
    return n * factorial(n-1);
}
```

A Small Example - Factorial

```
public static int factorial(int n){
    if(n < 0){
        throw new IllegalArgumentException("...");
    }
    if (n == 0){
        return 1;
    }
    return n * factorial(n-1);
}
```

We saw a BigInteger version in TicTacToe

Fibonacci

Fibonacci Numbers

- A sequence of values:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …
- Defined recursively:
  - By definition:
    - fib(0) is 0
    - fib(1) is 1
  - Remaining values:
    - fib(n) = fib(n-1)+fib(n-2)

Fibonacci Numbers

```
public int fibonacci(int n){
    if(n < 0){
        throw new IllegalArgumentException("...");
    }
    if(n == 0){
        return 0;
    }
    if(n == 1){
        return 1;
    }
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Recursion – Beyond Algorithms

Not all about code

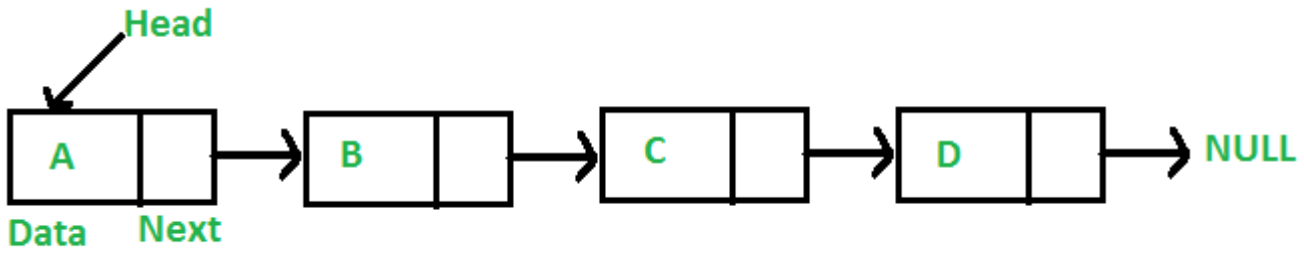
- We are going to make two recursive data structures

1. A Linkedlist
2. A BinarySearchTree

Linked List

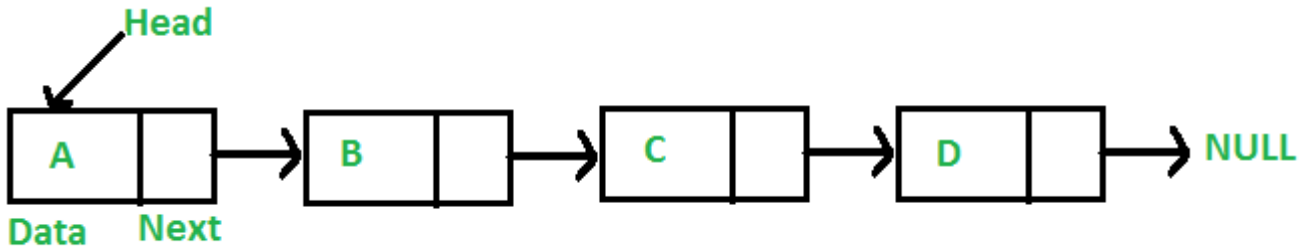
Linked List

- Data is stored structurally, 1 NODE per piece of data
- We have connection to start of list called HEAD
- Each NODE connects to another NODE to connect to more data, or NULL if there is no more data



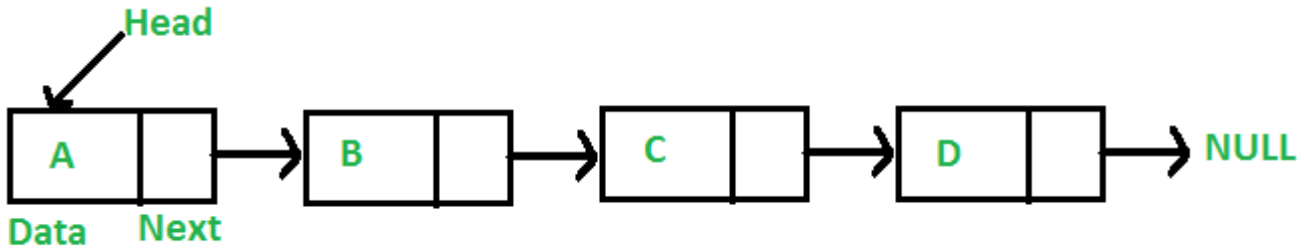
Linked List

- We want 2 classes
- One LinkedList and one for idea of Node



Linked List

- We want 2 classes
- One LinkedList and one for idea of Node

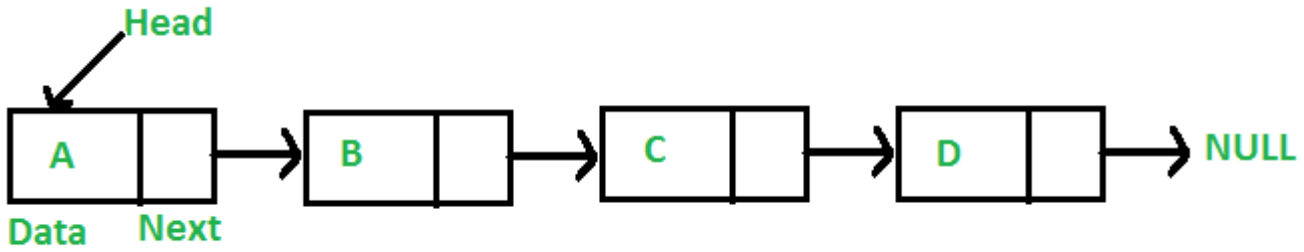



```
public class LinkedList {
}

public class Node {
}
```

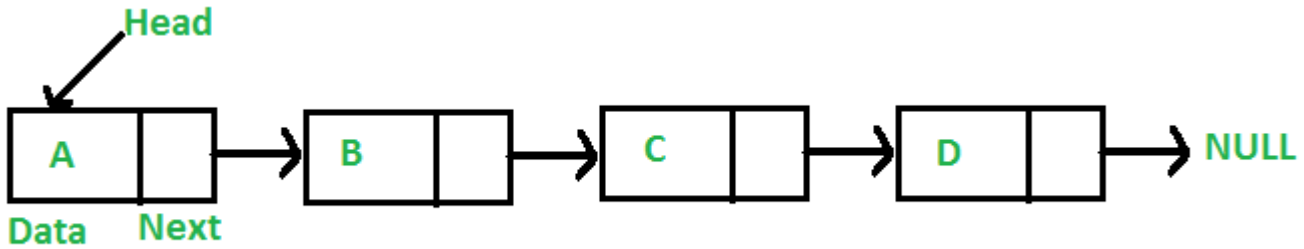
Linked List

- Does any other part of code need to know about a Node every?
- No! (Solution is internal class)




```
public class LinkedList {
    private class Node {
    }
}
```

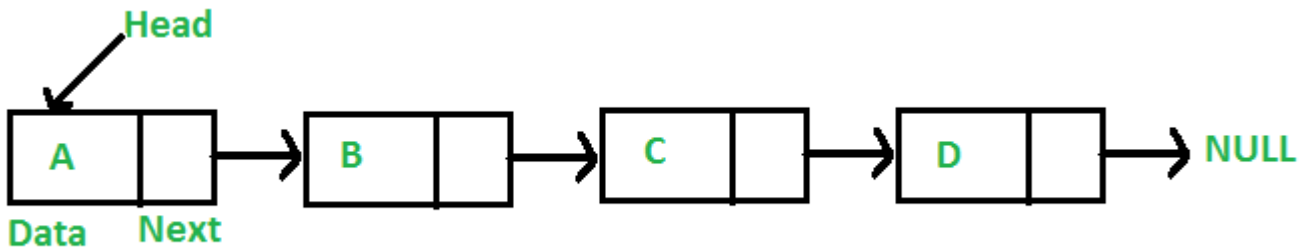
Linked List




```
public class LinkedList {
    private Node head;
    private class Node {
        Object data;
        Node next;
    }
}
```

- We don’t have to worry the same amount about modifiers for internal private class as the data in an node is never exposed in the same way
- We can still make it private and make accessors but for simplicity we’ll use Node more like just storage package of data

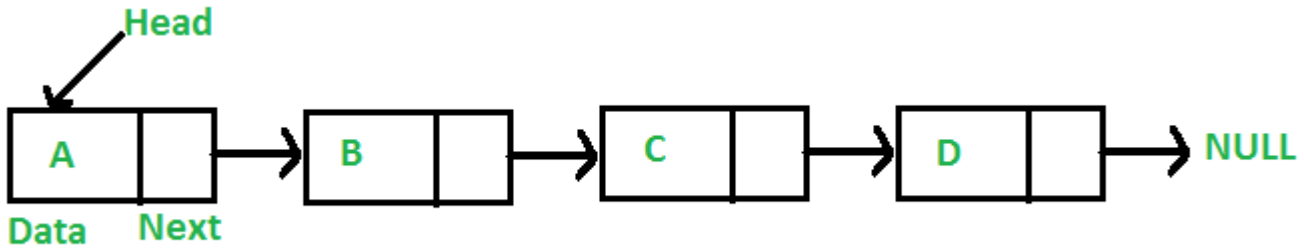
Linked List




```
public class LinkedList {
    private Node head;
    private class Node {
        Object data;
        Node next;
    }
}
```

- We don’t have to worry the same amount about modifiers for internal private class as the data in an node is never exposed in the same way
- We can off course still make it private and make accessors but for simplicity we’ll use Node more like just storage package of data

Linked List




```
public class LinkedList<E> {
    private Node head;
    private class Node {
        E data;
        Node next;
    }
}
```

- Generics!

Starting point

```
public class LinkedList<E> {
    private Node head;
    private class Node {
        E data;
        Node next;
    }
    public LinkedList(){
        head = null;
    }
}
```

Make it a Java List interface type

```
import java.util.*;

public class LinkedList<E> implements List<E> {
    private Node head;
    private class Node {
        E data;
        Node next;
    }
    public LinkedList(){
        head = null;
    }
```

We'll stick with easiest to start (non index)

```
import java.util.*;

public class LinkedList<E> implements List<E> {
    private Node head;
    private class Node {
        E data;
        Node next;
    }
    public LinkedList(){
        head = null;
    }
```

```
public int size()
public boolean isEmpty()
public boolean contains(Object o)
public boolean add(E e)
public boolean remove(Object o)
public void clear()
```

We'll stick with easiest to start (non index)

```
import java.util.*;

public class LinkedList<E> implements List<E> {
    private Node head;
    private class Node {
        E data;
        Node next;
    }
    public LinkedList(){
        head = null;
    }
```

```
public int size()
public boolean isEmpty()
public boolean contains(Object o)
public boolean add(E e)
public boolean remove(Object o)
public void clear()
```

Size()

- Right now the only way to know this is to loop and count how many things are stored!

Size()

- Let's instead track a efficient integer for this value

```
public class LinkedList<E> implements List<E> {
    private Node head;
    private int size;
    public LinkedList(){
        head = null;
        size = 0;
    }
    @Override
    public int size() {
        return size;
    }
```

```
@Override
public void clear() {
    head = null;
    size = 0;
}
```

```
@Override
public boolean isEmpty() {
    return size == 0;
}
```

contains()

- Iterative contains and recursive contains

```
@Override
public boolean contains(Object o) {
    Node curr = head;
    while (curr != null) {
        if (curr.data.equals(o)) {
            return true;
        }
        curr = curr.next;
    }
    return false;
}
```

```
@Override
public boolean contains(Object o) {
    return recContains(head, o);
}

private boolean recContains(Node node, Object o) {
    if(node == null){      //base case
        return false;
    }
    if(node.data.equals(o)){   // base case
        return true;
    }
    return recContains(node.next, o); // iterative case
}
```

toString()

```
@Override
public String toString(){
    StringBuilder sb = new StringBuilder();
    sb.append("[");
    Node curr = head;
    while(curr != null){
        sb.append(curr.data);
        if(curr.next != null) {
            sb.append(", ");
        }
        curr = curr.next;
    }
    sb.append("]");
    return sb.toString();
}
```

- This is an iterative version of toString()
- We can at the same time program a recursive version

toString()

```
@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    sb.append("[");
    recToString(head, sb);
    sb.append("]");
    return sb.toString();
}

private void recToString(Node node, StringBuilder sb) {
    if (node == null) {
        return;
    }
    sb.append(node.data);
    if (node.next != null) {
        sb.append(", ");
    }
    recToString(node.next, sb);
}
```

- This is a recursive version of toString()
- Base case is we are at end and are done!
- Recursive case is we add current node data, and then go to next node
  - (there is a sub-case where we add the comma only if we aren't last entry (indicated by next being null)

add()

- Iterative add
- We make a new head node if currently empty list

- Otherwise we loop until end of list and then add a new node there
- Slow!!!!!!!
- Solutions?

```
@Override
public boolean add(E e) {
   if(head == null){
      head = new Node();
      head.data = e;
      size++;
      return true;
   }
   Node curr = head;
   while (curr != null) {
      if (curr.next == null) {
         curr.next = new Node();
         curr.next.data = e;
         size++;
         return true;
      }
      curr = curr.next;
   }
   return false;
}
```

add()

- Iterative add
- We make a new head node if currently empty list

- Otherwise we loop until end of list and then add a new node there
- Slow!!!!!!!
- Solutions? Track a pointer to end of list node!

```
@Override
public boolean add(E e) {
   if(head == null){
      head = new Node();
      head.data = e;
      size++;
      return true;
   }
   Node curr = head;
   while (curr != null) {
      if (curr.next == null) {
         curr.next = new Node();
         curr.next.data = e;
         size++;
         return true;
      }
      curr = curr.next;
   }
   return false;
}
```

add()

- Recursive add
- We make a new head node if currently empty list
- Otherwise, explore (recursive case) until end of list and then add (base case)

```
@Override
public boolean add(E e) {
    if(head == null){
        head = new Node();
        head.data = e;
        size++;
        return true;
    }
    return recAdd(head, e);
}
```

```
private boolean recAdd(Node node, E e) {
    if(node.next == null){
        node.next = new Node();
        node.next.data = e;
        size++;
        return true;
    }
    return recAdd(node.next, e);
}
```

remove()

- Iterative remove

```
@Override
public boolean remove(Object o) {
    Node curr = head;
    Node prev = null;
    while (curr != null) {
        //What if I found it?
        prev = curr;
        curr = curr.next;
    }
    return false;
}
```

remove()

- Iterative remove
- If found at head, then we set head to skip past
- If found later we set previous to skip past current to the next

```
@Override
public boolean remove(Object o) {
   Node curr = head;
   Node prev = null;
   while (curr != null) {
      if (curr.data.equals(o)) {
         if (curr == head) {
            head = head.next;
         } else {
            prev.next = curr.next;
         }
         size--;
         return true;
      }
      prev = curr;
      curr = curr.next;
   }
   return false;
}
```

remove()

- Recursive remove
- Deal with empty start of list
- Then deal with removing at start of list, otherwise enter recursion

- Recursion actually checks delete of next node, and deletes by skipping past it from current node
- If we reach end we have base case to return false

```
@Override
public boolean remove(Object o) {
   if (head == null) {
      return false;
   }
   if (head.data.equals(o)) {
      head = head.next;
      return true;
   } else {
      return recRemove(head, o);
   }
}

public boolean recRemove(Node node, Object o) {
   if (node.next == null) {
      return false;
   }
   if (node.next.data.equals(o)) {
      node.next = node.next.next;
      size--;
      return true;
   }
   return false;
}
```

LinkedList (Make it a List)

```
public static void main(String[] args) {
    LinkedList<Integer> nal = new LinkedList <>();
    nal.add(1);
    nal.add(2);
    nal.add(3);
    System.out.println(nal);
}
```

[1, 2, 3]

Named LinkedList (Make it a List)

```
public static void main(String[] args) {
    LinkedList<Integer> nal = new LinkedList<>();
    System.out.println(nal.size() +" "+ nal.isEmpty());
    nal.add(1);
    System.out.println(nal.size() +" "+ nal.isEmpty());
    nal.add(2);
    System.out.println(nal.size() +" "+ nal.isEmpty());
    nal.add(3);
    System.out.println(nal.size() +" "+ nal.isEmpty());
}
```

```
0 true
1 false
2 false
3 false
```

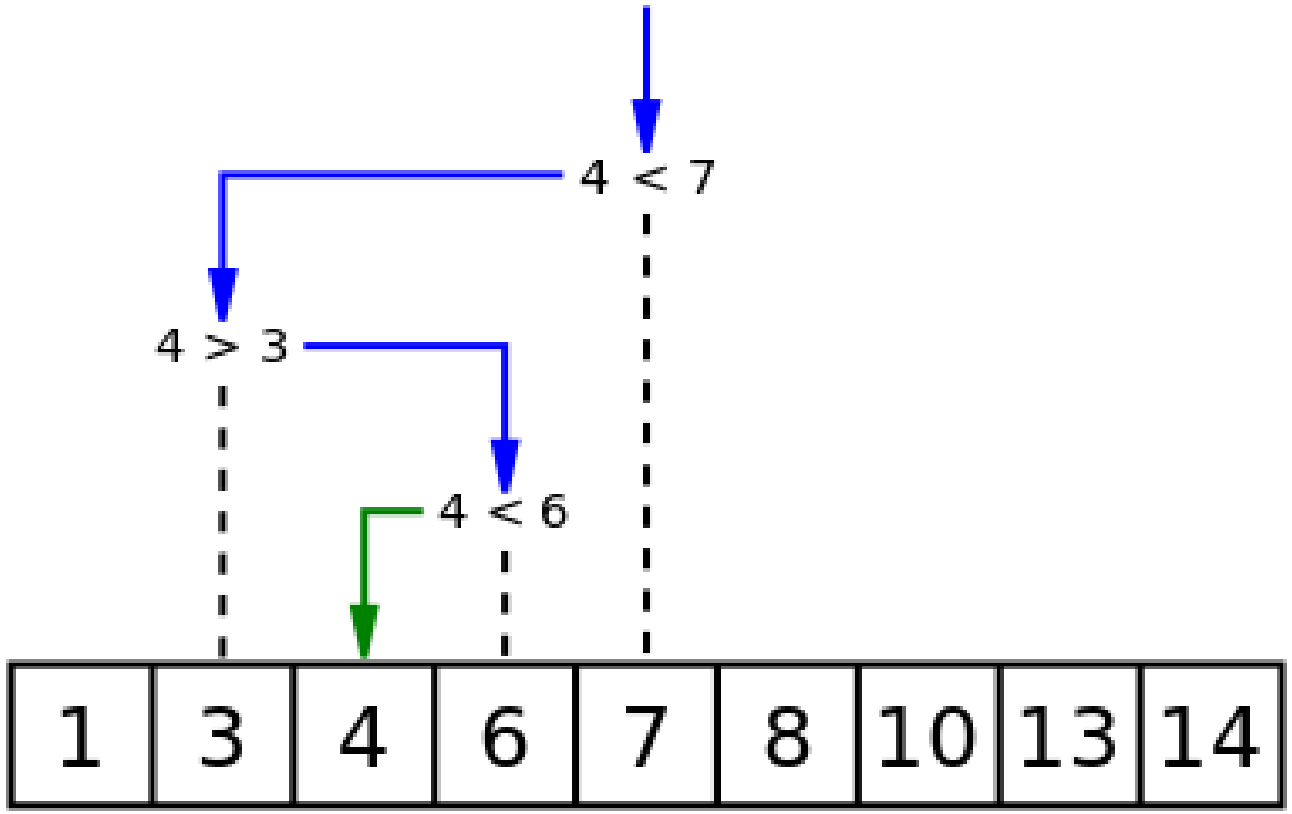
Named LinkedList (Make it a List)

```
public static void main(String[] args) {
    LinkedList<Integer> nal = new LinkedList<>();
    nal.add(1);       nal.add(2);       nal.add(3);
    System.out.println(nal);
    System.out.println(nal.contains(1));
    System.out.println(nal.contains(4));
    nal.remove(new Integer(1));
    System.out.println(nal);
    System.out.println(nal.contains(1));
    nal.clear();
    System.out.println(nal);
}
```

```
[1, 2, 3]
true
false
[2, 3]
false
[]
```

Binary Search

Binary Search



Binary Search

```
public static Integer binarySearch(Comparable[] values, Comparable search){
   return binarySearch(values, search, 0, values.length-1);
}
private static Integer binarySearch(Comparable[] values, Comparable search, int start, int end){
   if(start > end){
      return null;
   }
   int middle = (start + end) / 2;
   int compare = search.compareTo(values[middle]);
   if(compare == 0){
      return middle;
   }
   if (compare < 0){
      return binarySearch(values, search, start, middle-1);
   }else{
      return binarySearch(values, search, middle+1, end);
   }
}
```

Binary Search

```
Integer[] values = new Integer[]{0,2,4,6,8,10,12,14,16,18,20};
for (int i = -1; i <= 21; i++) {
    System.out.println(i+" -> "+binarySearch(values, new Integer(i)));
}
```

```
-1 -> null
0 -> 0
1 -> null
2 -> 1
3 -> null
4 -> 2
5 -> null
6 -> 3
7 -> null
8 -> 4
9 -> null
10 -> 5
11 -> null
12 -> 6
13 -> null
14 -> 7
15 -> null
16 -> 8
17 -> null
18 -> 9
19 -> null
20 -> 10
21 -> null
```

Binary Search

```
Integer[] values = new Integer[]{20,18,16,14,12,10,8,6,4,2,0};
for (int i = -1; i <= 21; i++) {
   System.out.println(i+" -> "+binarySearch(values, new Integer(i)));
}
```

```
-1 -> null
0 -> null
1 -> null
2 -> null
3 -> null
4 -> null
5 -> null
6 -> null
7 -> null
8 -> null
9 -> null
10 -> 5
11 -> null
12 -> null
13 -> null
14 -> null
15 -> null
16 -> null
17 -> null
18 -> null
19 -> null
20 -> null
21 -> null
```

Binary Search

```
Integer[] values = new Integer[]{20,18,16,14,12,10,8,6,4,2,0};
for (int i = -1; i <= 21; i++) {
   System.out.println(i+" -> "+binarySearch(values, new Integer(i)));
}
```

```
-1 -> null
0 -> null
1 -> null
2 -> null
3 -> null
4 -> null
5 -> null
6 -> null
7 -> null
8 -> null
9 -> null
10 -> 5
11 -> null
12 -> null
13 -> null
14 -> null
15 -> null
16 -> null
17 -> null
18 -> null
19 -> null
20 -> null
21 -> null
```

Binary Search

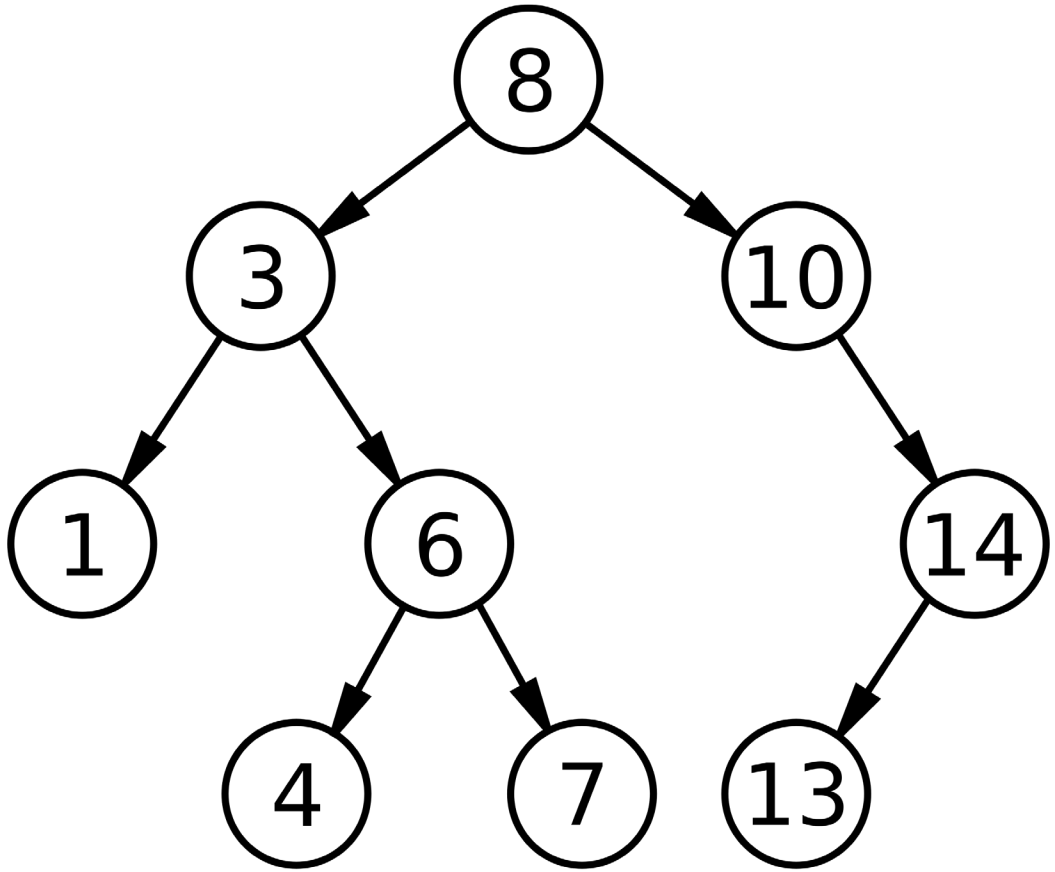
```
Integer[] values = new Integer[]{20,18,16,14,12,10,8,6,4,2,0};
 Arrays.sort(values);
for (int i = -1; i <= 21; i++) {
   System.out.println(i+" -> "+binarySearch(values, new Integer(i)));
}
```

```
-1 -> null
0 -> 0
1 -> null
2 -> 1
3 -> null
4 -> 2
5 -> null
6 -> 3
7 -> null
8 -> 4
9 -> null
10 -> 5
11 -> null
12 -> 6
13 -> null
14 -> 7
15 -> null
16 -> 8
17 -> null
18 -> 9
19 -> null
20 -> 10
21 -> null
```

Binary Search Tree

Binary Search Tree

- Data on left is always less than data in middle
- Data on right is always more than data in middle
- This is true for every node



Binary Search Tree

- Data on left is always less than data in middle
- Data on right is always more than data in middle
- This is true for every node

```
public class BinarySearchTree<K extends Comparable> {
    private Node root;
    private class Node{
        Comparable<K> key;
        Node left, right;
    }
```

add

- If empty then adding is simple otherwise we have to make choices of middle, left, or right

```
public boolean add(K key){
   if(root == null){
      root = new Node();
      root.key = key;
      return true;
   }
   return recAdd(root, key);
}
```

add

```
private boolean recAdd(Node node, K key) {
    int compare = key.compareTo(node.key);
    if(compare == 0){
        return false;
    }
    if(compare < 0) {
        if (node.left == null) {
            node.left = new Node();
            node.left.key = key;
            return true;
        }
        return recAdd(node.left, key);
    }
    else{
        if (node.right == null) {
            node.right = new Node();
            node.right.key = key;
            return true;
        }
        return recAdd(node.right, key);
    }
}
```

contains

- We start at root,
- Either we are at a null, find it, or go left, or right

```
public boolean contains(K key){
    return recContains(root, key);
}
```

```
private boolean recContains(Node node, K key) {
    if(node == null){
        return false;
    }
    int compare = key.compareTo(node.key);
    if(compare == 0){
        return true;
    }
    if(compare < 0) {
        return recContains(node.left, key);
    }else{
        return recContains(node.right, key);
    }
}
```

toString

- We build outer part, then we go through tree, left before current, current, then right
- This gets us sorted order

```
@Override
public String toString(){
   StringBuilder sb = new StringBuilder();
   sb.append("[");
   recToString(root, sb);
   sb.delete(sb.length()-2,sb.length());
   sb.append("]");
   return sb.toString();
}
```

```
private void recToString(Node node, StringBuilder sb) {
   if(node == null){
      return;
   }
   recToString(node.left, sb);
   sb.append(node.key);
   sb.append(", ");
   recToString(node.right, sb);
}
```

toString

```
Integer[] values = new Integer[]{0,2,4,6,8,10,12,14,16,18,20};
BinarySearchTree<Integer> bst = new BinarySearchTree<>();
for(Integer value: values){
    bst.add(value);
}
System.out.println(bst);
for (int i = -1; i <= 21; i++) {
    System.out.println(i+" -> "+bst.contains(new Integer(i)));
}
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
-1 -> false
0 -> true
1 -> false
2 -> true
3 -> false
4 -> true
5 -> false
6 -> true
7 -> false
8 -> true
9 -> false
10 -> true
11 -> false
12 -> true
13 -> false
14 -> true
15 -> false
16 -> true
17 -> false
18 -> true
19 -> false
20 -> true
21 -> false
```

toString

```
Integer[] values = new Integer[]{20,18,16,14,12,10,8,6,4,2,0};
BinarySearchTree<Integer> bst = new BinarySearchTree<>();
for(Integer value: values){
    bst.add(value);
}
System.out.println(bst);
for (int i = -1; i <= 21; i++) {
    System.out.println(i+" -> "+bst.contains(new Integer(i)));
}
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
-1 -> false
0 -> true
1 -> false
2 -> true
3 -> false
4 -> true
5 -> false
6 -> true
7 -> false
8 -> true
9 -> false
10 -> true
11 -> false
12 -> true
13 -> false
14 -> true
15 -> false
16 -> true
17 -> false
18 -> true
19 -> false
20 -> true
21 -> false
```

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

0 (root)

->

2

->

4

->

6

->

8

->

10

->

12

->

14

->

16

->

18

->

20

[20,18,16,14,12,10,8,6,4,2,0]

0

->

2

->

4

->

6

->

8

->

10

->

12

->

14

->

16

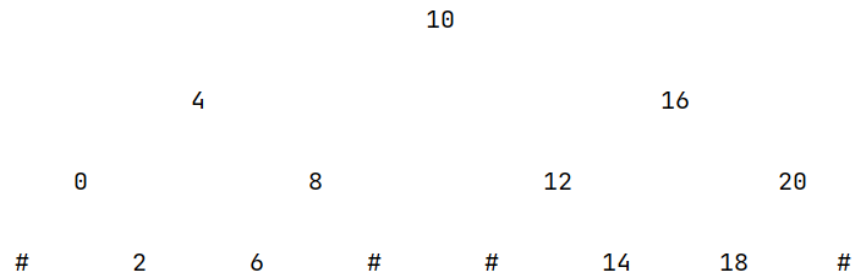
->

18

->

20 (root)

[10, 4, 16, 0, 2, 8, 6, 12, 14, 20, 18]



[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

0 (root)

->

2

->

4

->

6

->

8

->

10

->

12

->

14

->

16

->

18

->

20

Depth 11

[20,18,16,14,12,10,8,6,4,2,0]

0

->

2

->

4

->

6

->

8

->

10

->

12

->

14

->

16

->

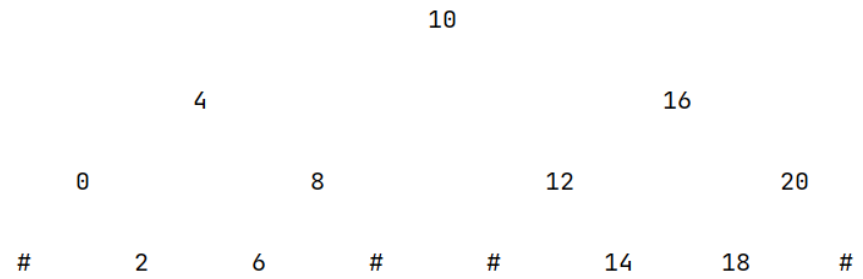
18

->

20 (root)

Depth 11

[10, 4, 16, 0, 2, 8, 6, 12, 14, 20, 18]



Depth 4

$\log_2 11 = 3.459 \ldots$

Onward to … optimization and profiling.

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY