Algorithms and Data Structures (ECS529)

Nikos Tzevelekos

Lecture 9

Trees

Data structures

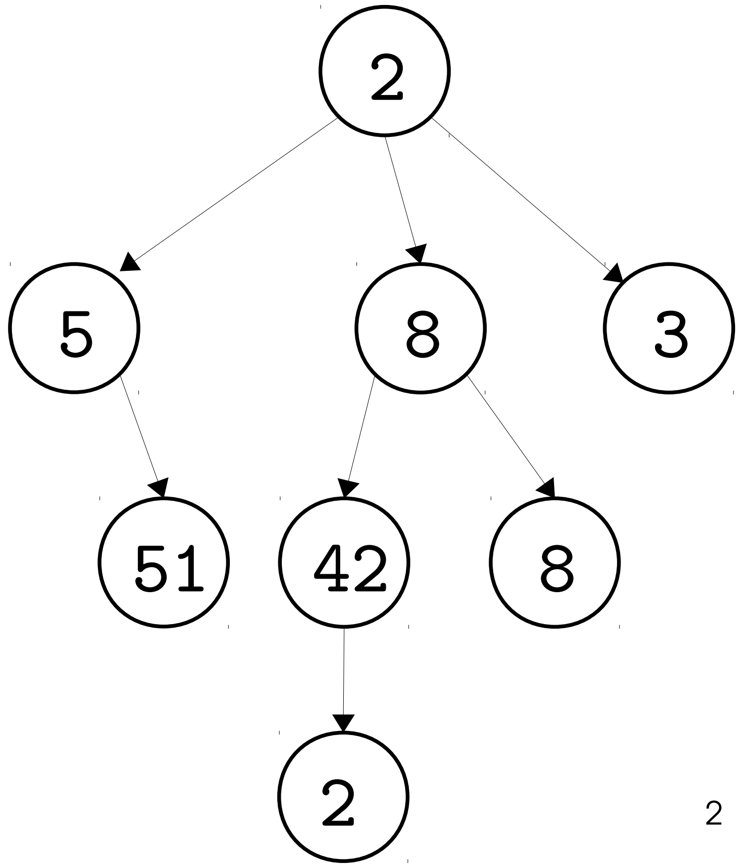
All the data structures we have seen so far have been linear:

- a collection of data put in line, in some way
- we access the data either by indexing or by traversing the data structure

In this lecture we look at trees:

not

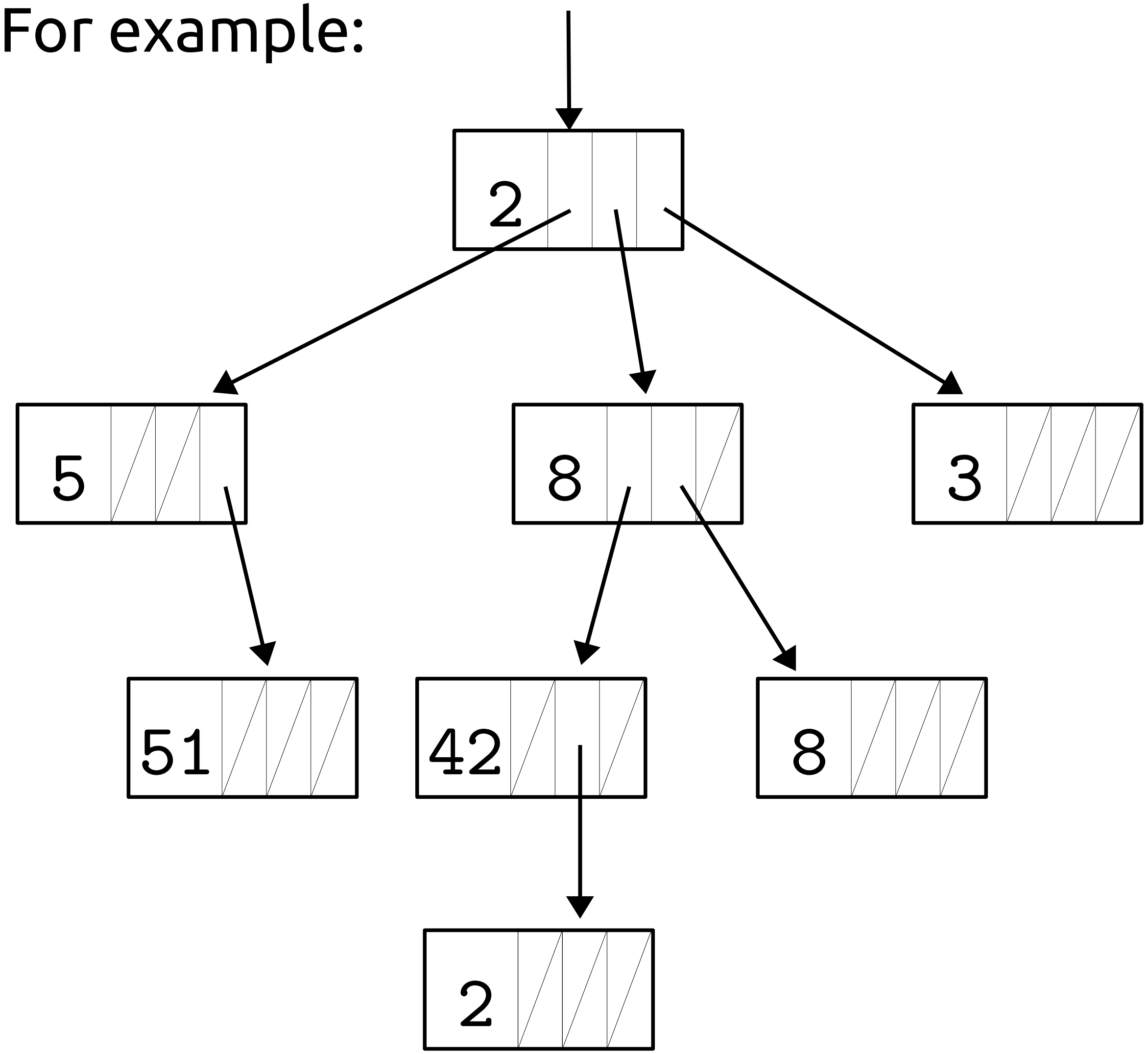
but



What is a tree?

Trees are linked lists where each node can point to more than one "next" node, which are called its **children**.

For example:



Useful terminology:

root:
the 'head' of the tree, i.e. the node from which every other node can be reached

leaf:
a node that has no children

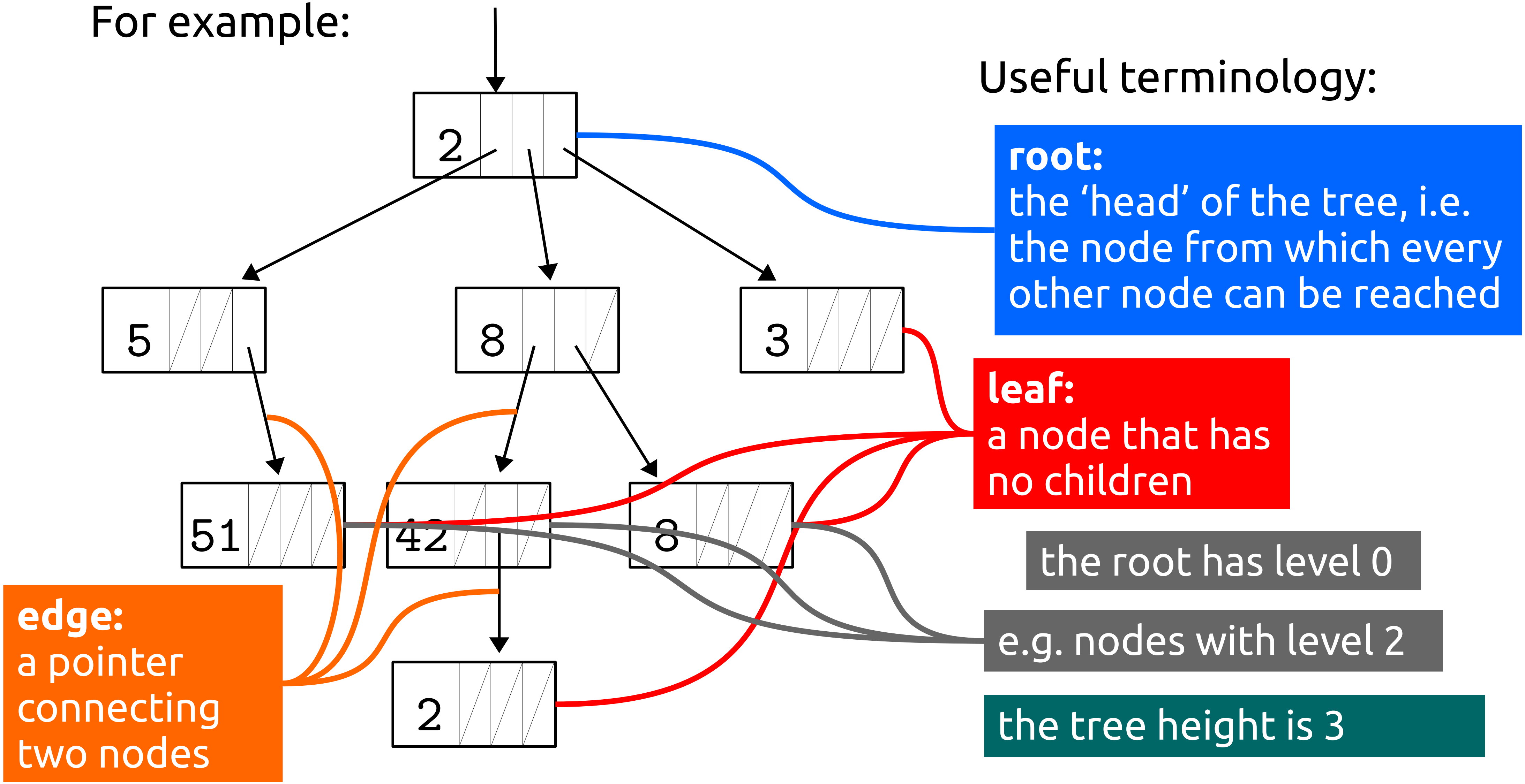
level of a node:
its distance from the root

height of a tree:
the greatest level in it

What is a tree?

Trees are linked lists where each node can point to more than one "next" node, which are called its **children**.

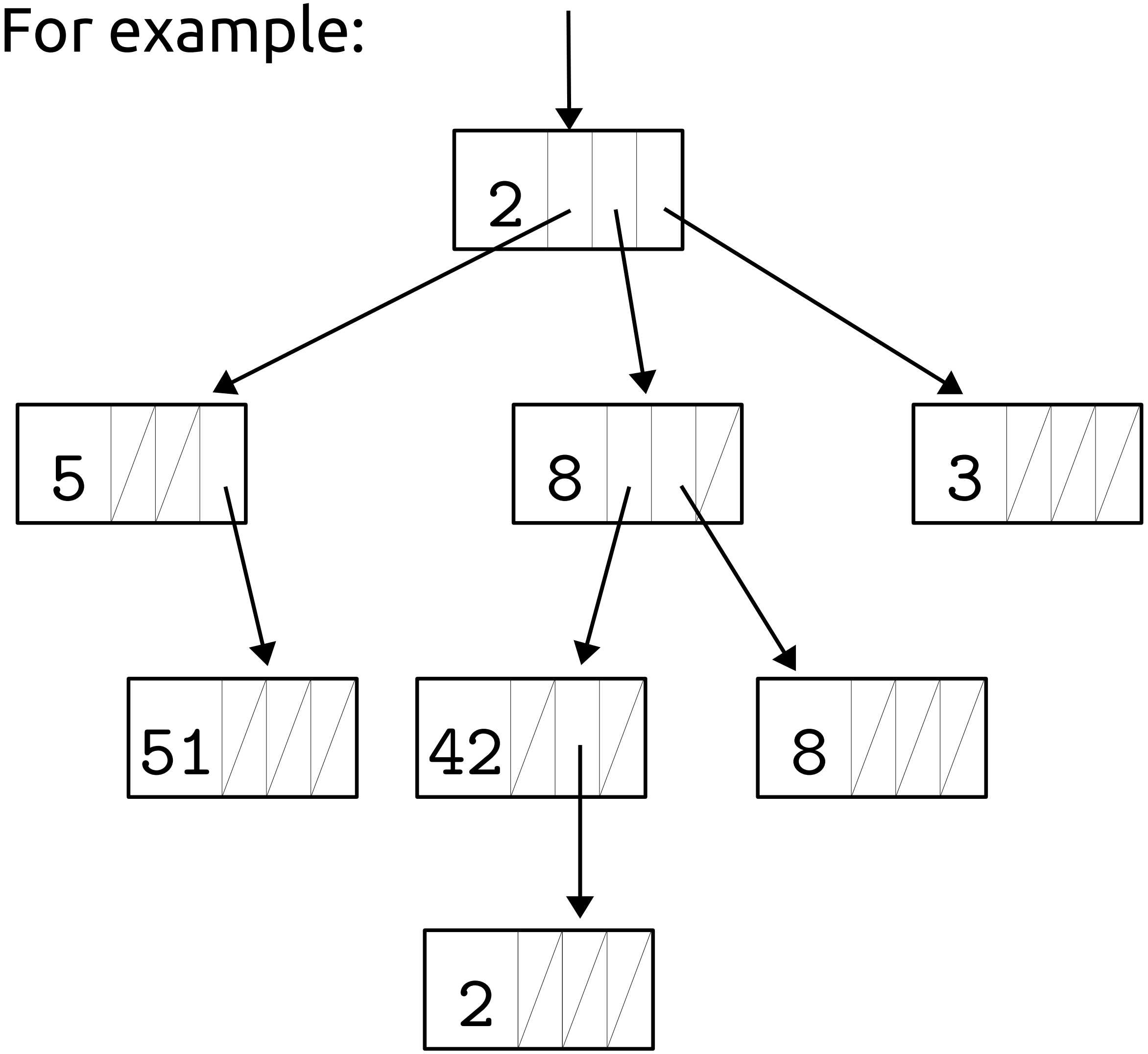
For example:



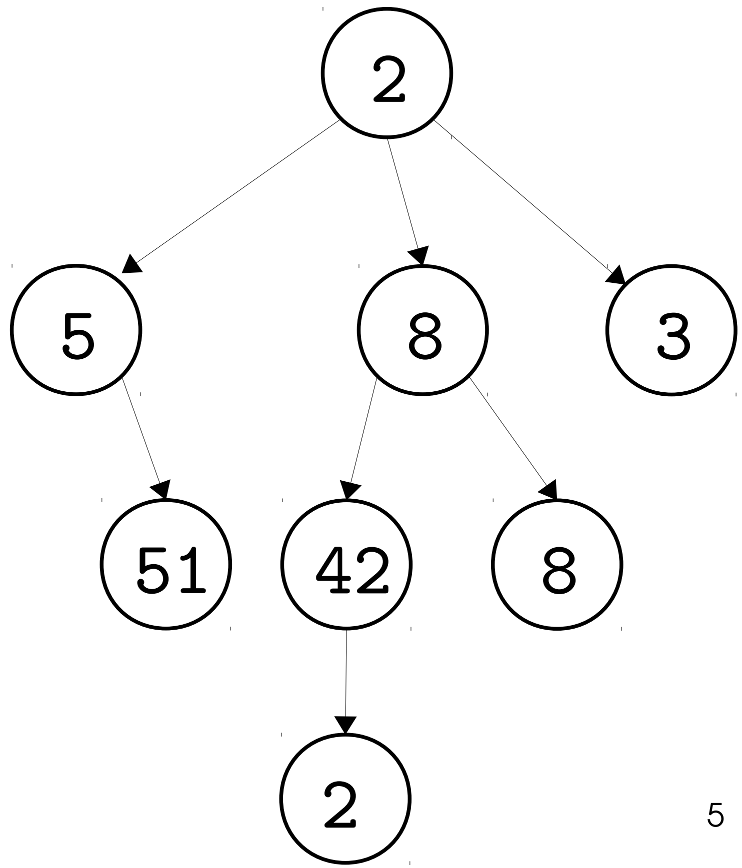
What is a tree?

Trees are linked lists where each node can point to more than one "next" node, which are called its **children**.

For example:

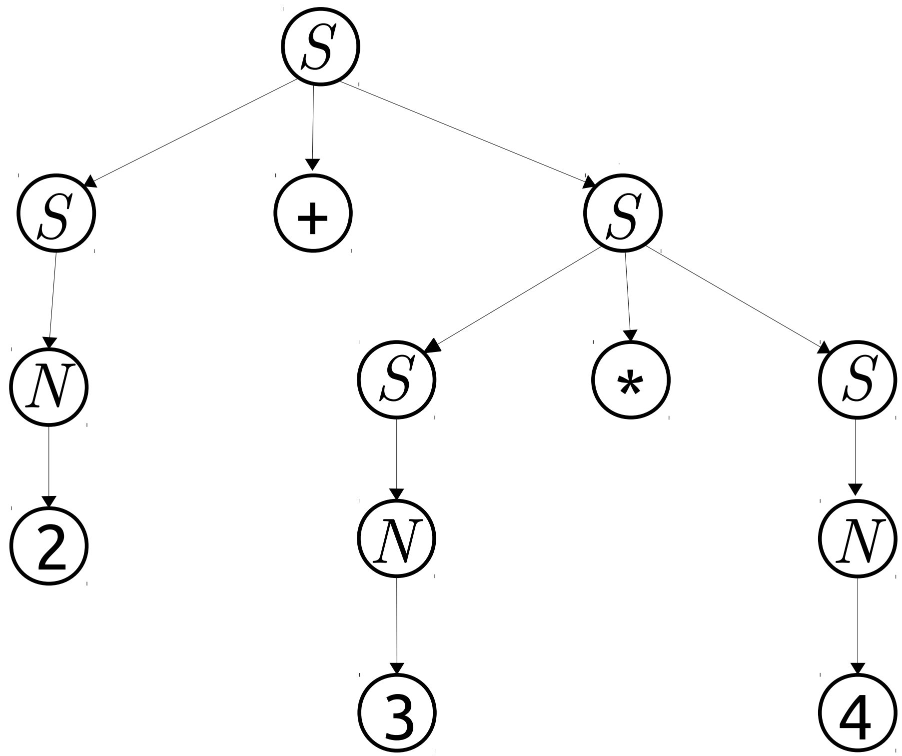


we usually draw trees without the pointer boxes and head pointer:



Example: parse trees

Remember parse trees for context-free grammars:



$$\left(\begin{array}{lcl} S & \to & N \mid S{+}S \mid S{*}S \mid (S) \\ N & \to & 1 \mid 2 \mid 3 \mid \ldots \mid 1000 \end{array}\right)$$

Example: HTML document tree

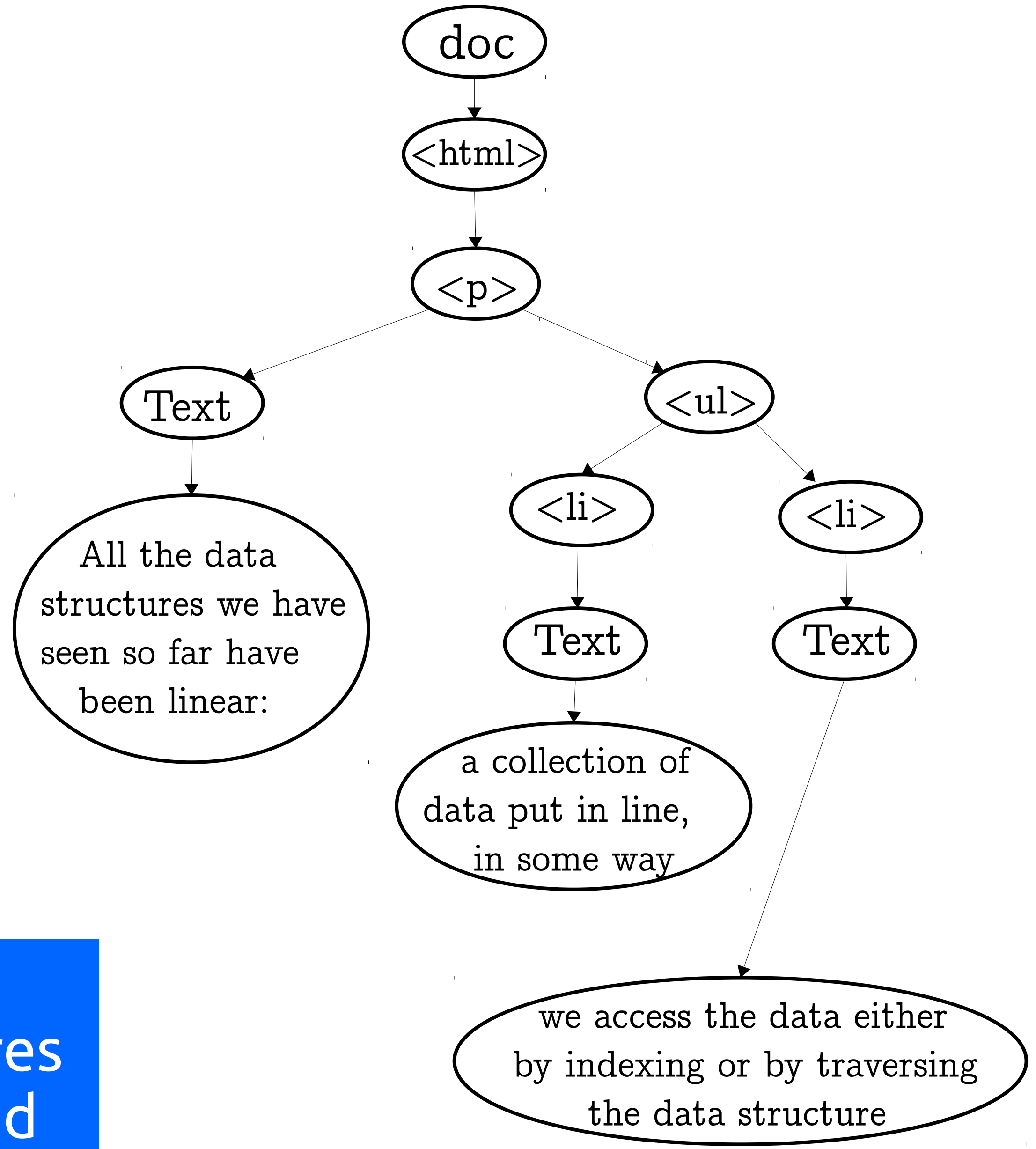
<html><p>

All the data structures we have seen so far have been linear:

 a collection of data put in line, in some way

 we access the data either by indexing or by traversing the data structure

</p><html>



A web browser needs to parse documents into tree data structures (DOM) in order to display them and react to user inputs

Other examples

Tree representations of data are very common:

- in text processing
- in compilers
- on the web (html, xml, json)
- in databases (we are going to see how)
- in file systems (and hierarchical systems more generally)
- etc.

The benefit of using trees is that we can have quicker access to elements than in linear data structures:

> following one pointer in a tree amounts to going forward a whole level of nodes!

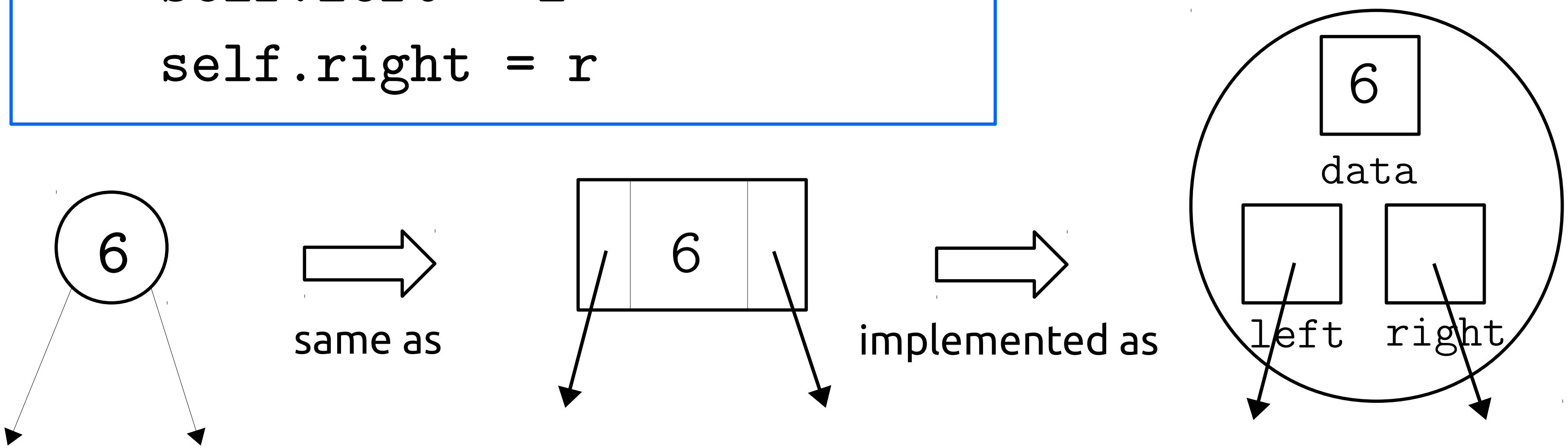
Binary trees

We call a tree **binary** when each node in it has at most 2 children – we refer to the children as: left, right.

We will next focus on binary trees.

Here is an implementation of binary tree nodes in Python:

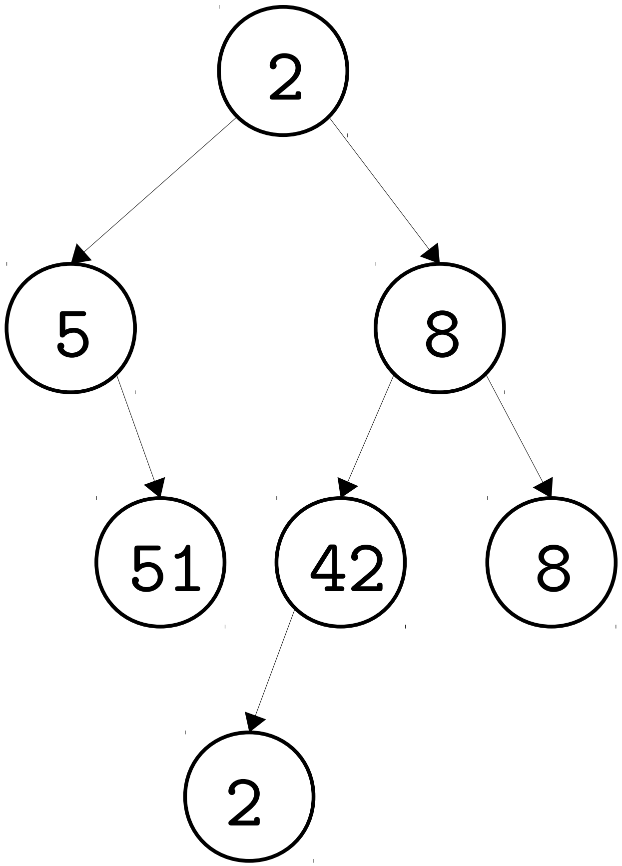
```
class BTNode:
  def __init__(self, d, l, r):
    self.data = d
    self.left = l
    self.right = r
```



Binary tree creation (bottom up)

The easiest way to create a tree is starting from its leaves.

E.g. the tree on the left is created by the code on the right:

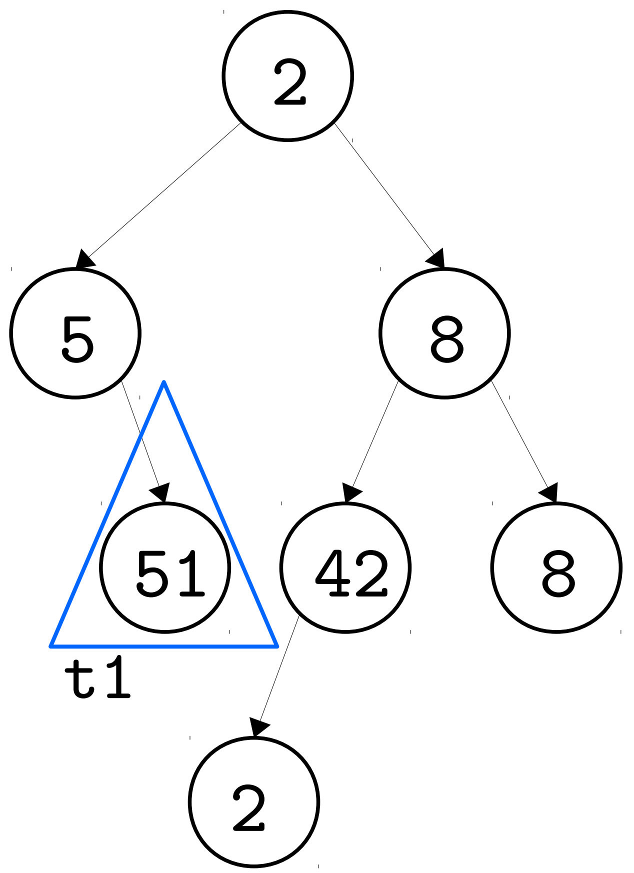



```
t1 = BTNode(51,None,None)
t2 = BTNode(5,None,t1)
t3 = BTNode(2,None,None)
t4 = BTNode(42,t3,None)
t5 = BTNode(8,None,None)
t6 = BTNode(8,t4,t5)
t = BTNode(2,t2,t6)
```

Binary tree creation (bottom up)

The easiest way to create a tree is starting from its leaves.

E.g. the tree on the left is created by the code on the right:



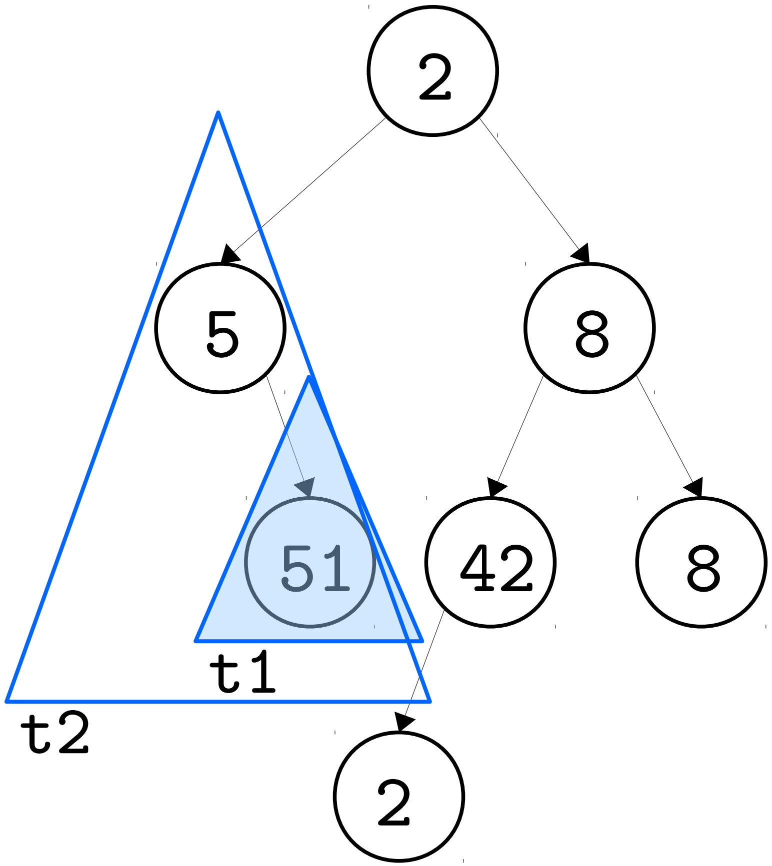

```
t1 = BTNode(51,None,None)
t2 = BTNode(5,None,t1)
t3 = BTNode(2,None,None)
t4 = BTNode(42,t3,None)
t5 = BTNode(8,None,None)
t6 = BTNode(8,t4,t5)
t = BTNode(2,t2,t6)
```

let's look at this step by step ...

Binary tree creation (bottom up)

The easiest way to create a tree is starting from its leaves.

E.g. the tree on the left is created by the code on the right:



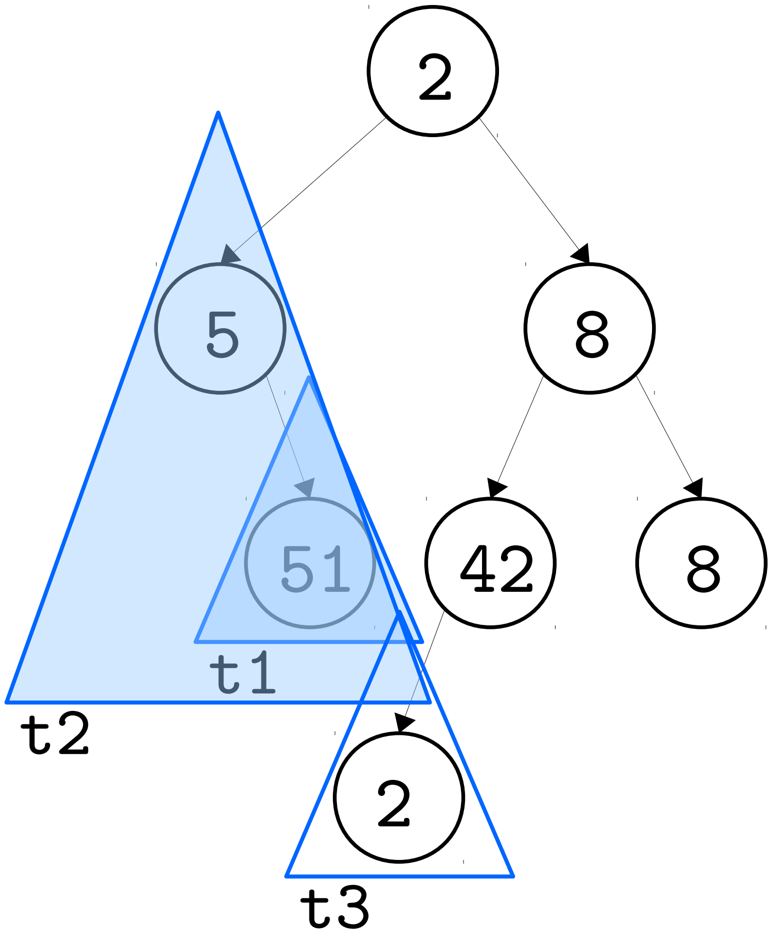

```
t1 = BTNode(51,None,None)
t2 = BTNode(5,None,t1)
t3 = BTNode(2,None,None)
t4 = BTNode(42,t3,None)
t5 = BTNode(8,None,None)
t6 = BTNode(8,t4,t5)
t  = BTNode(2,t2,t6)
```

let's look at this step by step ...

Binary tree creation (bottom up)

The easiest way to create a tree is starting from its leaves.

E.g. the tree on the left is created by the code on the right:



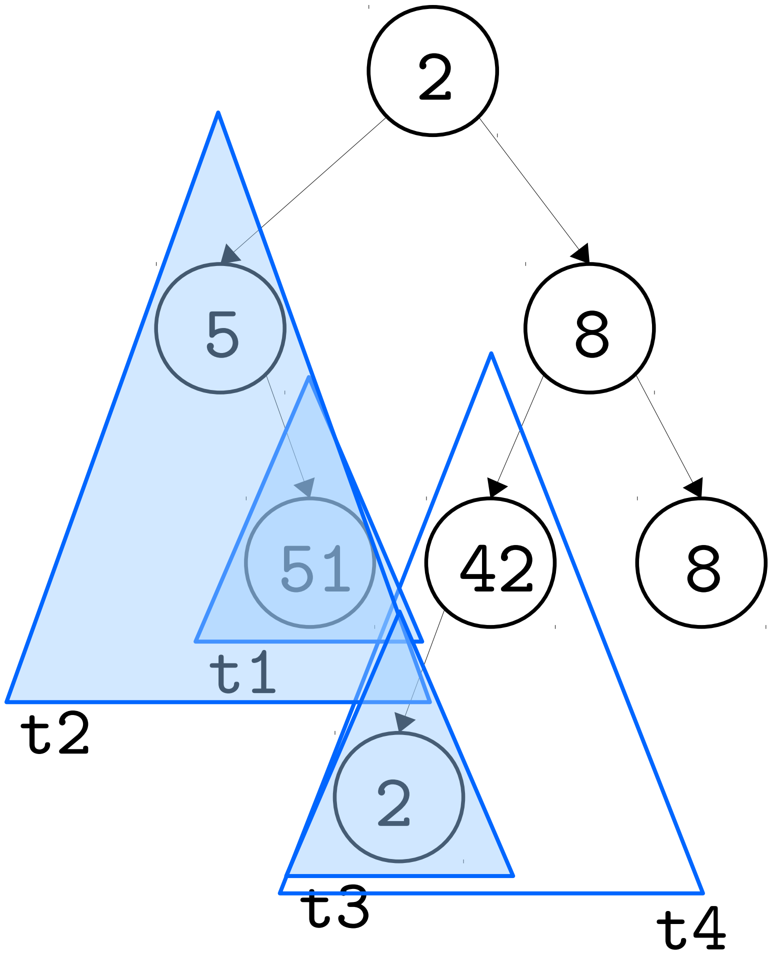

```
t1 = BTNode(51,None,None)
t2 = BTNode(5,None,t1)
t3 = BTNode(2,None,None)
t4 = BTNode(42,t3,None)
t5 = BTNode(8,None,None)
t6 = BTNode(8,t4,t5)
t  = BTNode(2,t2,t6)
```

let's look at this step by step ...

Binary tree creation (bottom up)

The easiest way to create a tree is starting from its leaves.

E.g. the tree on the left is created by the code on the right:



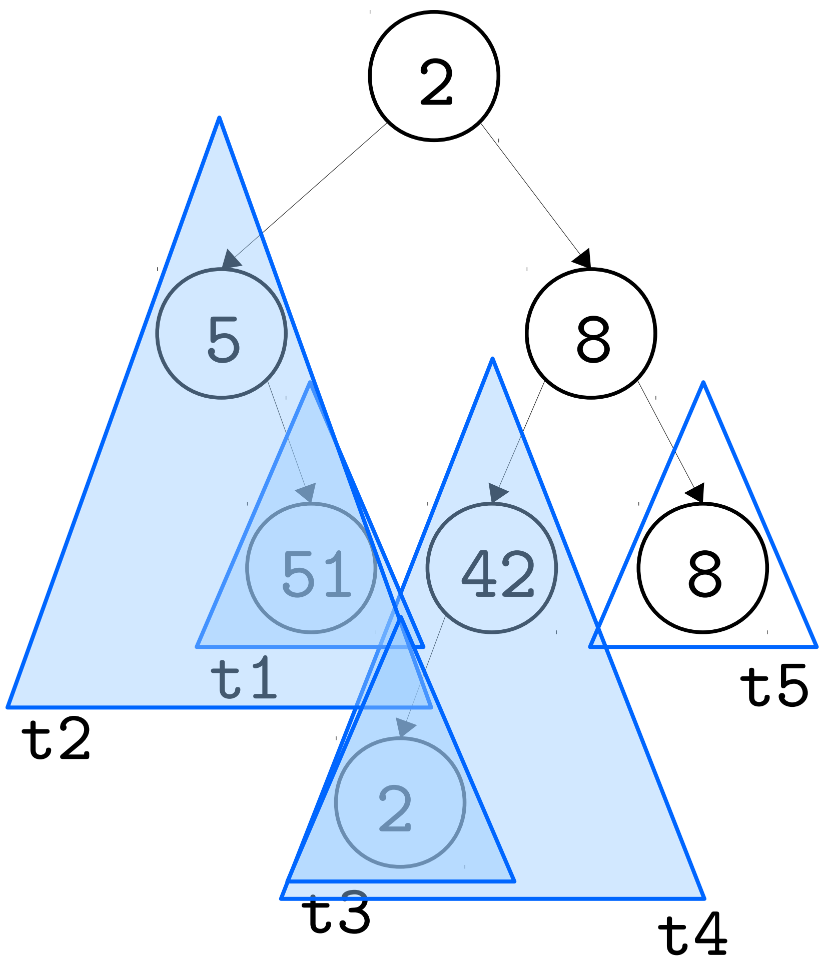

```
t1 = BTNode(51,None,None)
t2 = BTNode(5,None,t1)
t3 = BTNode(2,None,None)
t4 = BTNode(42,t3,None)
t5 = BTNode(8,None,None)
t6 = BTNode(8,t4,t5)
t  = BTNode(2,t2,t6)
```

let's look at this step by step ...

Binary tree creation (bottom up)

The easiest way to create a tree is starting from its leaves.

E.g. the tree on the left is created by the code on the right:



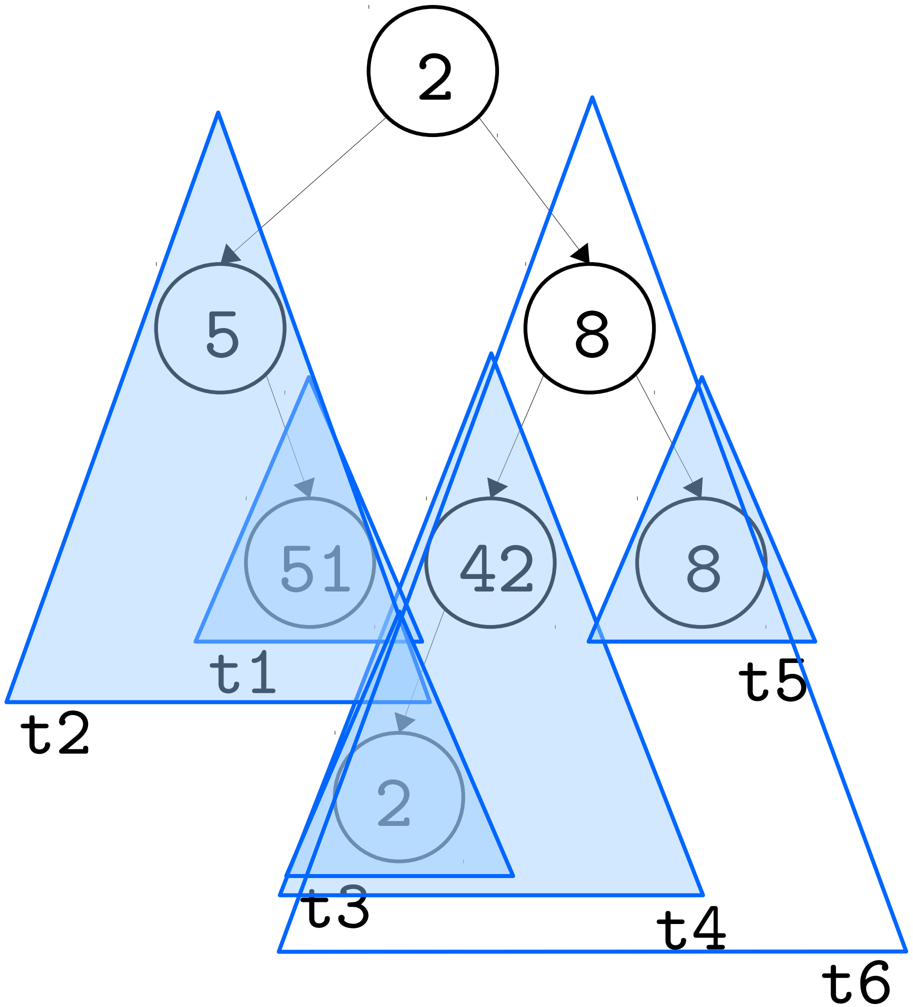

```
t1 = BTNode(51,None,None)
t2 = BTNode(5,None,t1)
t3 = BTNode(2,None,None)
t4 = BTNode(42,t3,None)
t5 = BTNode(8,None,None)
t6 = BTNode(8,t4,t5)
t  = BTNode(2,t2,t6)
```

let's look at this step by step ...

Binary tree creation (bottom up)

The easiest way to create a tree is starting from its leaves.

E.g. the tree on the left is created by the code on the right:



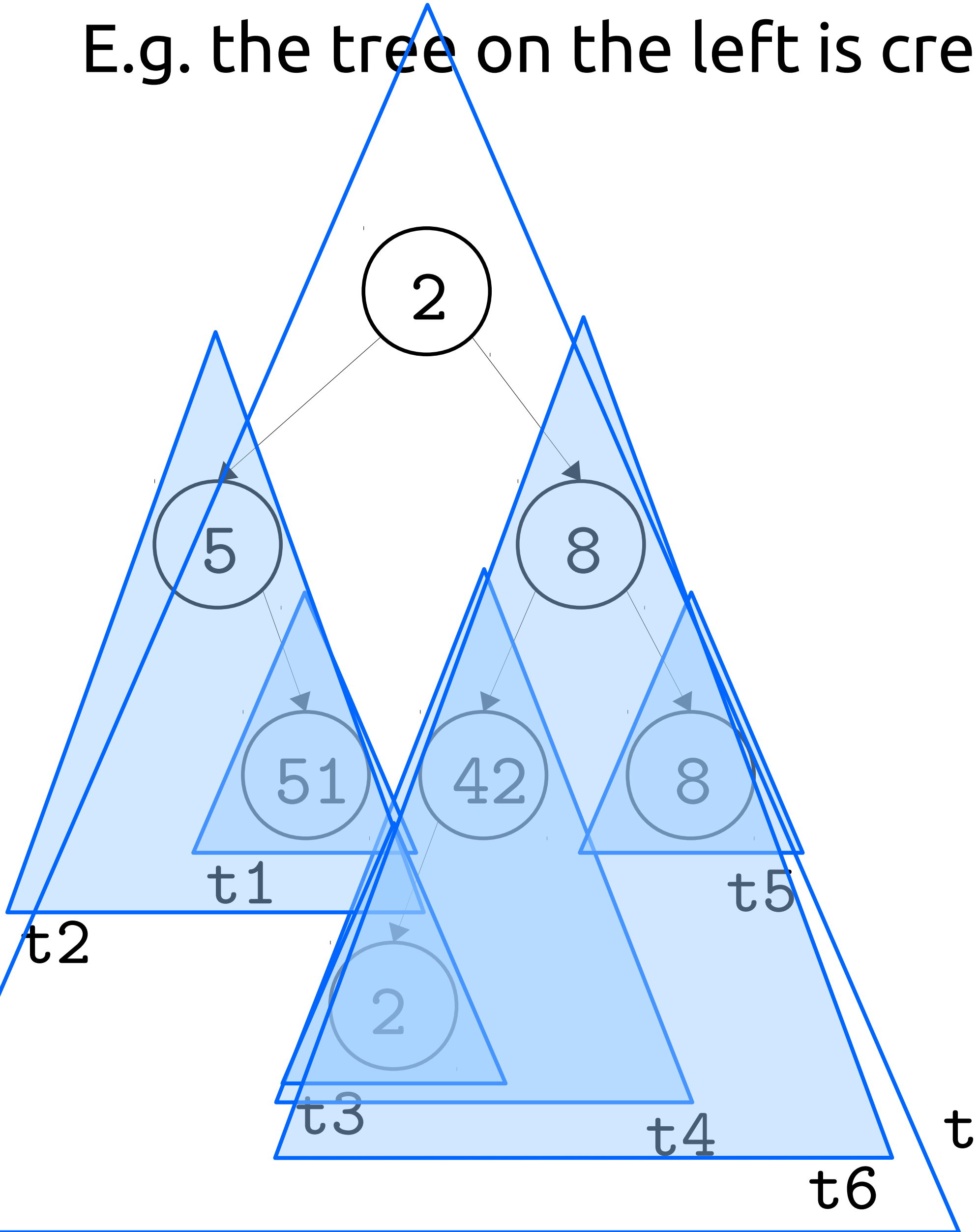

```
t1 = BTNode(51,None,None)
t2 = BTNode(5,None,t1)
t3 = BTNode(2,None,None)
t4 = BTNode(42,t3,None)
t5 = BTNode(8,None,None)
t6 = BTNode(8,t4,t5)
t = BTNode(2,t2,t6)
```

let's look at this step by step ...

Binary tree creation (bottom up)

The easiest way to create a tree is starting from its leaves.

E.g. the tree on the left is created by the code on the right:



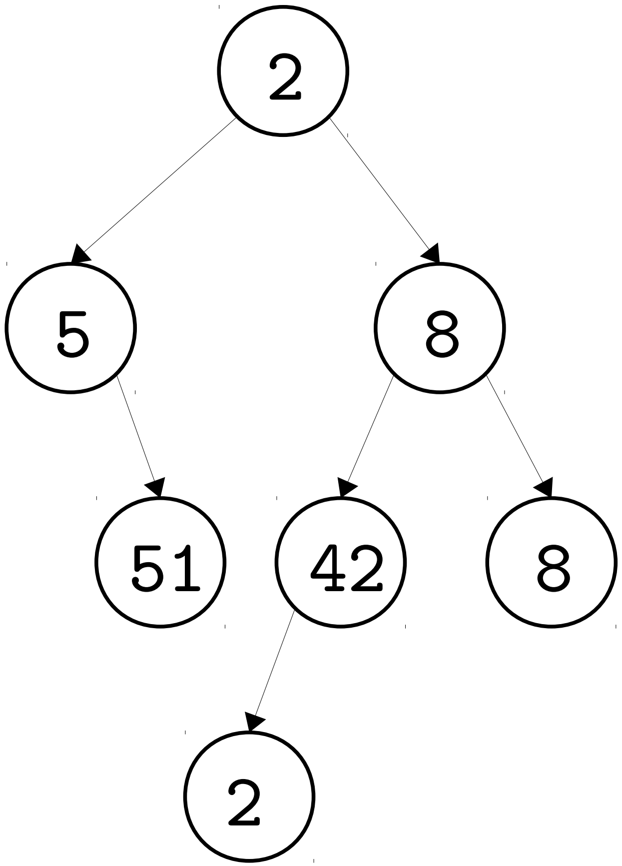

```
t1 = BTNode(51,None,None)
t2 = BTNode(5,None,t1)
t3 = BTNode(2,None,None)
t4 = BTNode(42,t3,None)
t5 = BTNode(8,None,None)
t6 = BTNode(8,t4,t5)
t = BTNode(2,t2,t6)
```

Binary tree creation (bottom up)

The easiest way t

E.g. the tree on t

we can re-use variables storing nodes which we have already connected to their parents, so the same tree can be built like this:

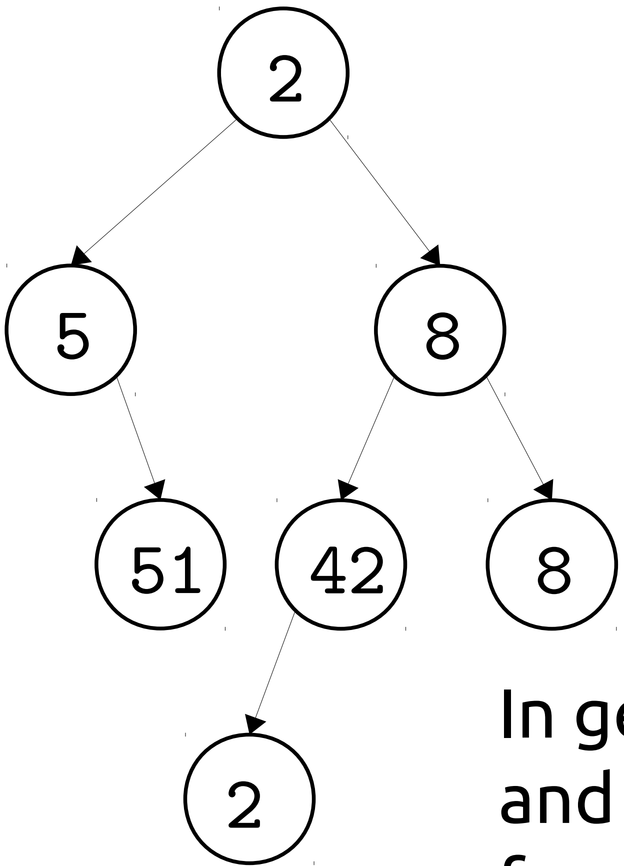



```
t = BTNode(51,None,None)
t1 = BTNode(5,None,t)
t = BTNode(2,None,None)
t = BTNode(42,t,None)
t2 = BTNode(8,None,None)
t2 = BTNode(8,t,t2)
t = BTNode(2,t1,t2)
```

Binary tree creation (top down)

We can also create a tree from its root.

E.g. the tree on the left is created by the code on the right:



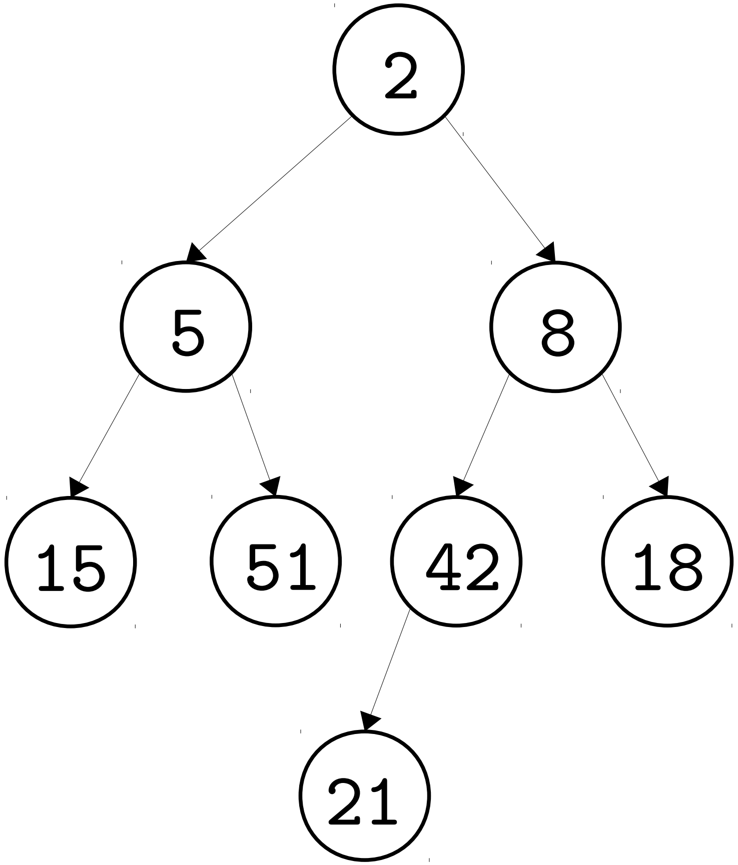

```
t = BTNode(2,None,None)
t.left = BTNode(5,None,None)
t.left.right = BTNode(51,None,None)
t.right = BTNode(8,None,None)
t.right.left = BTNode(42,None,None)
t.right.left.left = BTNode(2,None,None)
t.right.right = BTNode(8,None,None)
```

In general, depending on the kind of the tree and the use we intend for it, we write functions for adding elements to it and we usually combine top-down and bottom-up

Tree traversal

Storing data in a tree is because we want to access it.

How can we do this systematically? How can we traverse a tree?



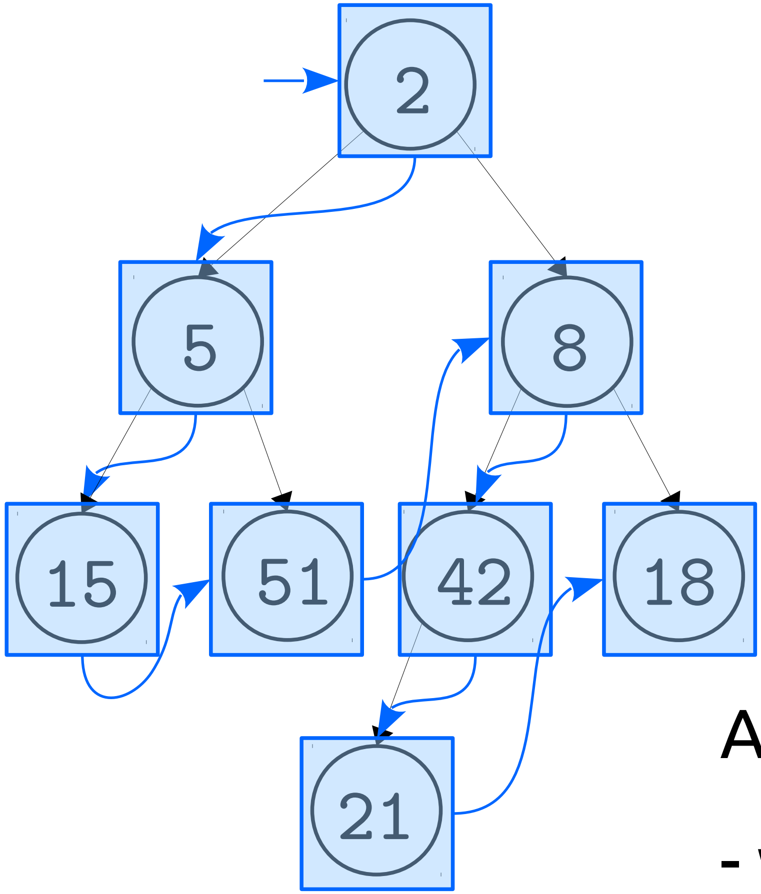
For example:

- we could go level-by-level, from the top, going through each level from left to right
- we could start from the root, go left and traverse all the left subtree, then go right and traverse the right subtree;
  each subtree is traversed in the same way (root, left subtree, then right subtree)
- as above, but root -> right -> left
- something else

We look at two general ways of traversing, or *searching*, a tree.

Depth-first search

Idea: pick one direction (e.g. left), and go through whole tree on that direction; then, go through whole subtree in other direction.



In each subtree, we follow the same idea.

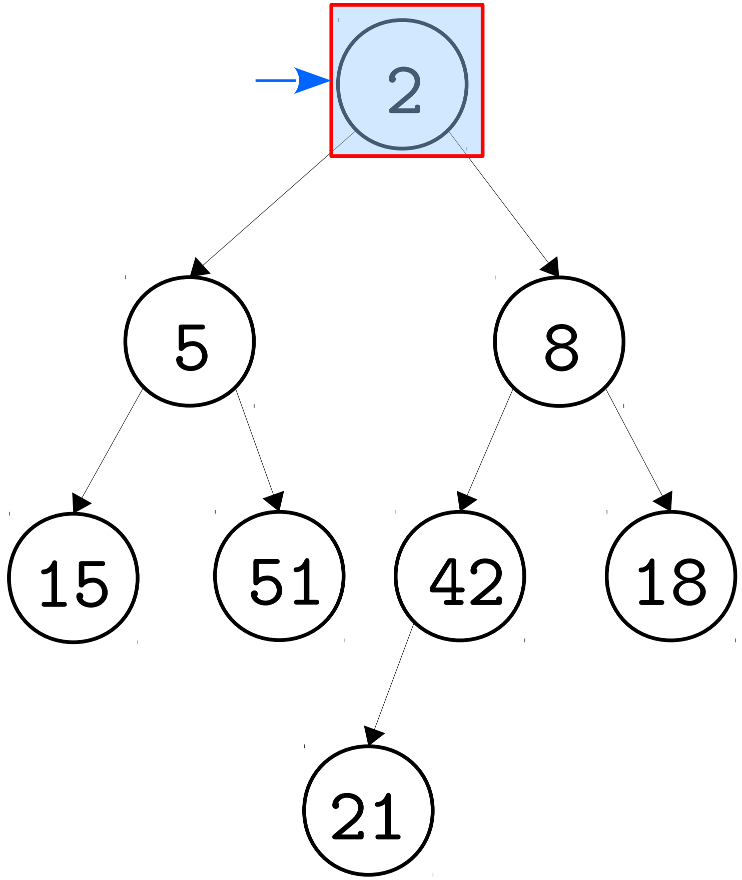
This is called *depth-first search*, because we traverse the tree down (by going left) until we reach its leaves, then we backtrack and continue with the lowest right node that we have not traversed already.

What algorithm can we use for this?

A recursive one:

- we start from the root
- go left and recursively search left subtree
- go right and recursively search right subtree

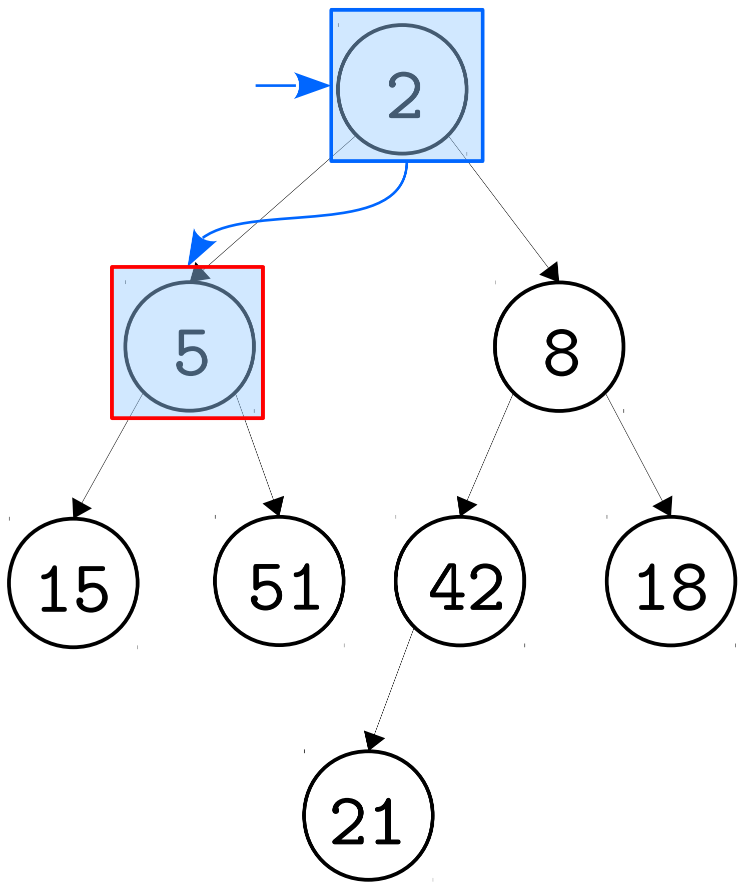
Depth-first search example



- start from 2

Algorithm:
- start from the root
- recursively search left subtree
- recursively search right subtree

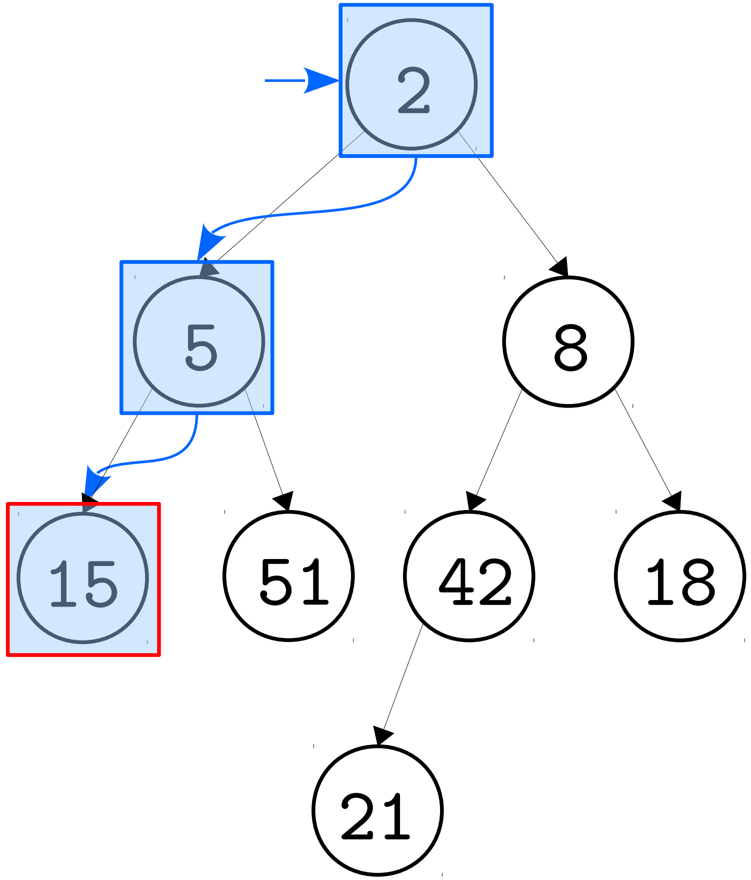
Depth-first search example



- start from 2
- go left (subtree with root 5)
  - start from 5

Algorithm:
- start from the root
- recursively search left subtree
- recursively search right subtree

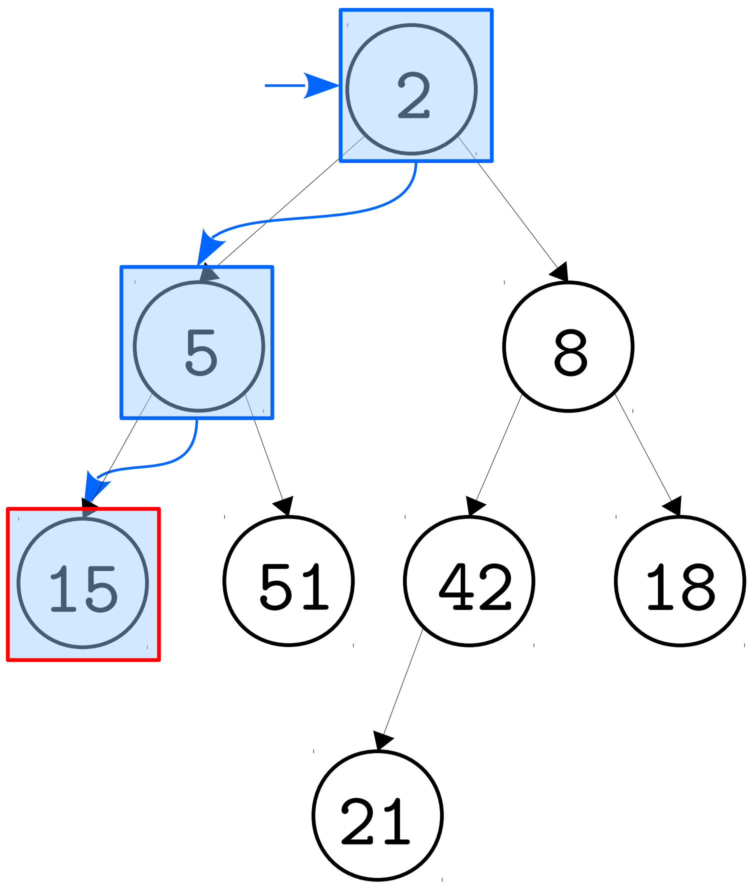
Depth-first search example



- start from 2
- go left (subtree with root 5)
  - start from 5
  - go left (subtree with root 15)
    - start from 15

Algorithm:
- start from the root
- recursively search left subtree
- recursively search right subtree

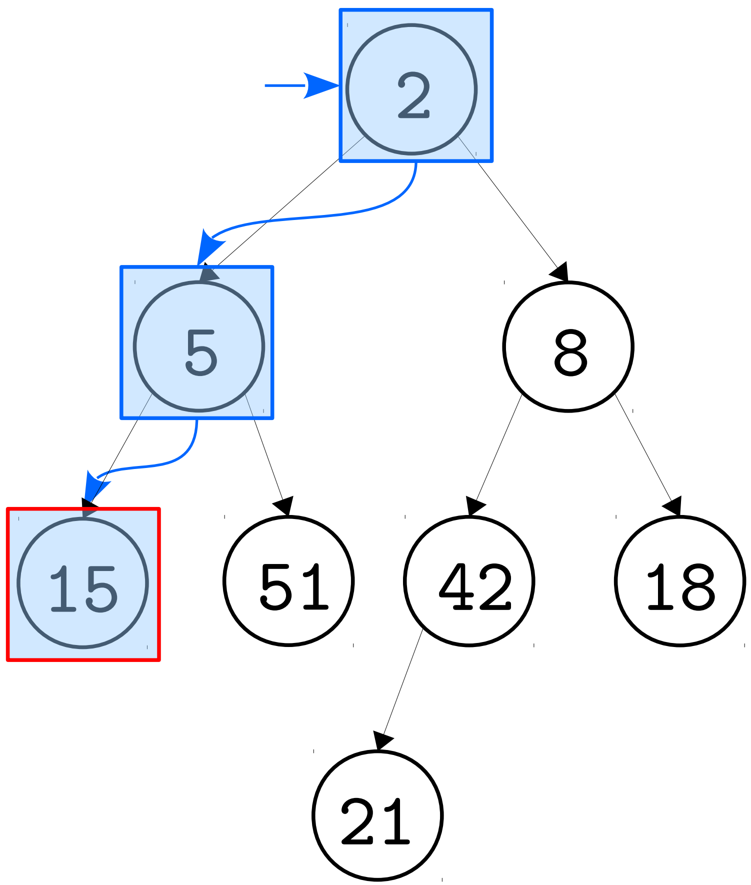
Depth-first search example



- start from 2
- go left (subtree with root 5)
  - start from 5
  - go left (subtree with root 15)
    - start from 15
    - cannot go left → **done**

Algorithm:
- start from the root
- recursively search left subtree
- recursively search right subtree

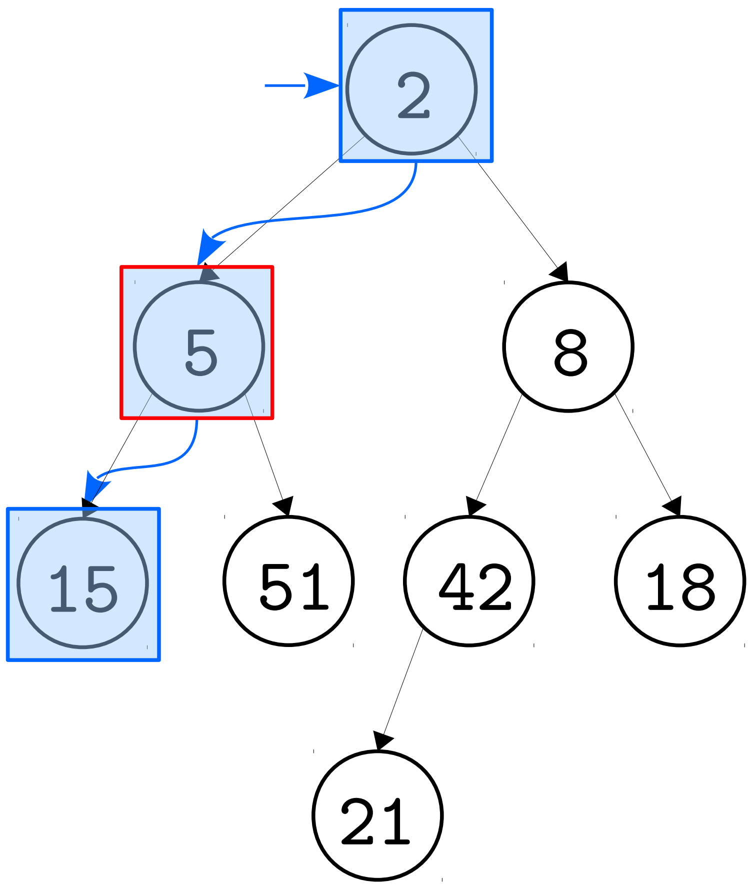
Depth-first search example



- start from 2
- go left (subtree with root 5)
  - start from 5
  - go left (subtree with root 15) → **done**
    - start from 15
    - cannot go left → **done**
    - cannot go right → **done**

Algorithm:
- start from the root
- recursively search left subtree
- recursively search right subtree

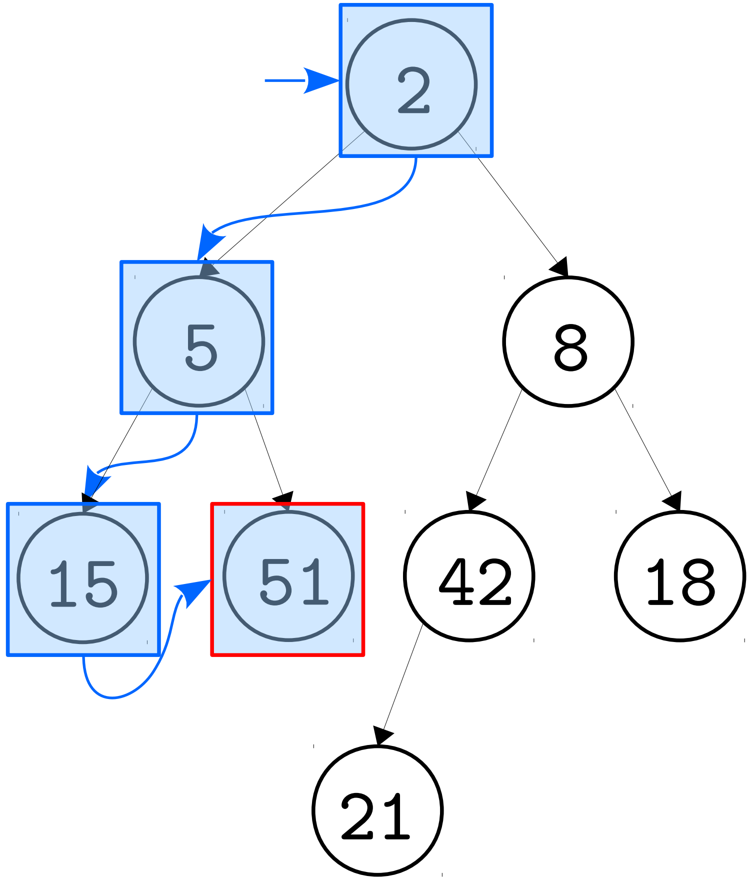
Depth-first search example



- start from 2
- go left (subtree with root 5)
  - start from 5
  - go left (subtree with root 15) → **done**
    - start from 15
    - cannot go left → **done**
    - cannot go right → **done**
  - go right (subtree with root 51)

Algorithm:
- start from the root
- recursively search left subtree
- recursively search right subtree

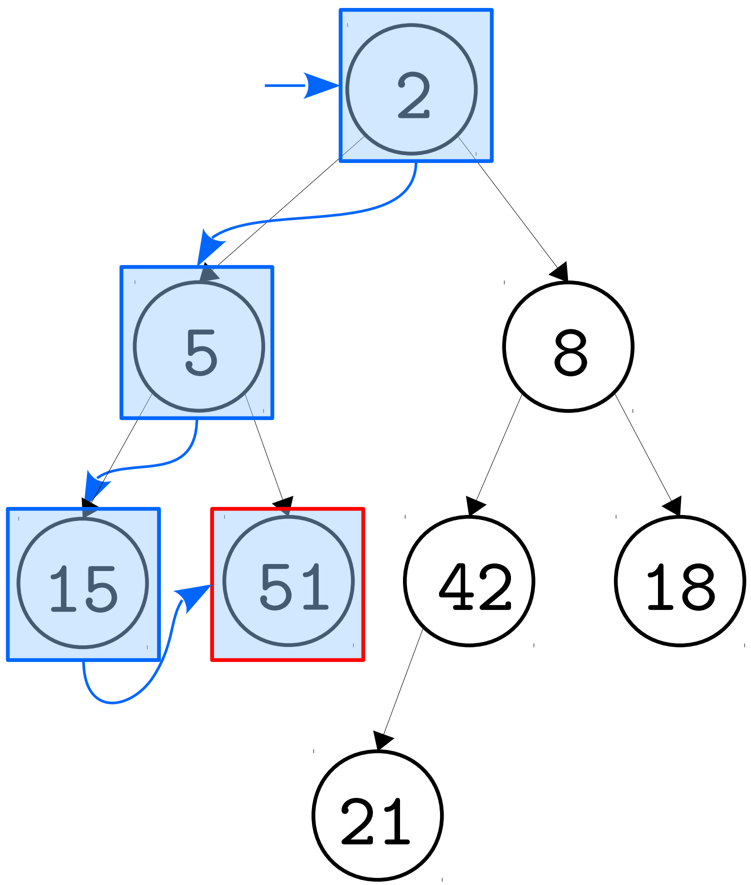
Depth-first search example



- start from 2
- go left (subtree with root 5)
  - start from 5
  - go left (subtree with root 15) → **done**
    - start from 15
    - cannot go left → **done**
    - cannot go right → **done**
  - go right (subtree with root 51)
    - start from 51

Algorithm:
- start from the root
- recursively search left subtree
- recursively search right subtree

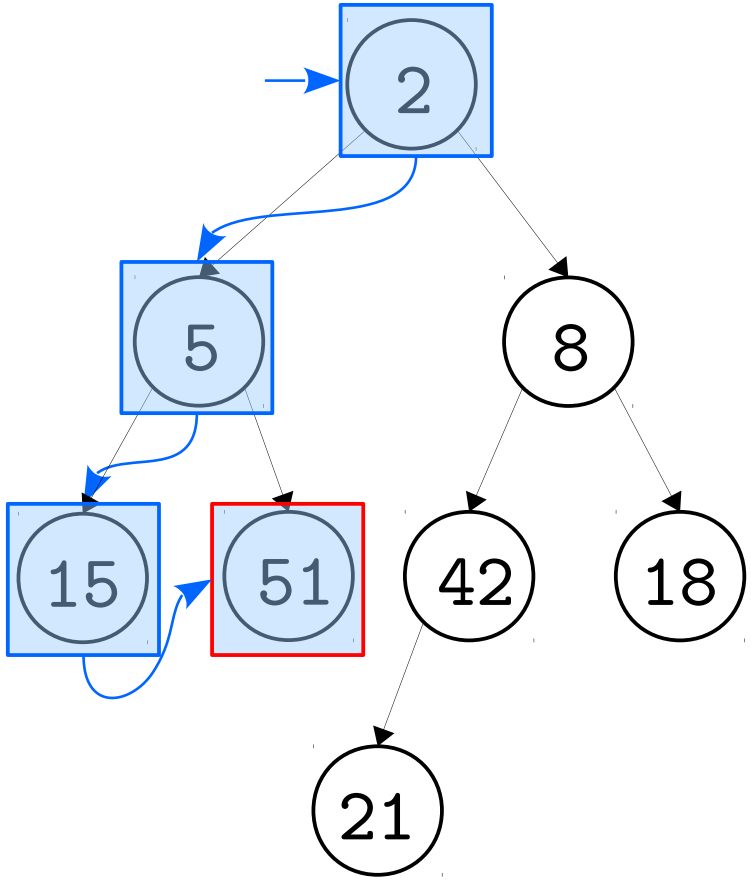
Depth-first search example



- start from 2
- go left (subtree with root 5)
  - start from 5
  - go left (subtree with root 15) → **done**
    - start from 15
    - cannot go left → **done**
    - cannot go right → **done**
  - go right (subtree with root 51)
    - start from 51
    - cannot go left → **done**

Algorithm:
- start from the root
- recursively search left subtree
- recursively search right subtree

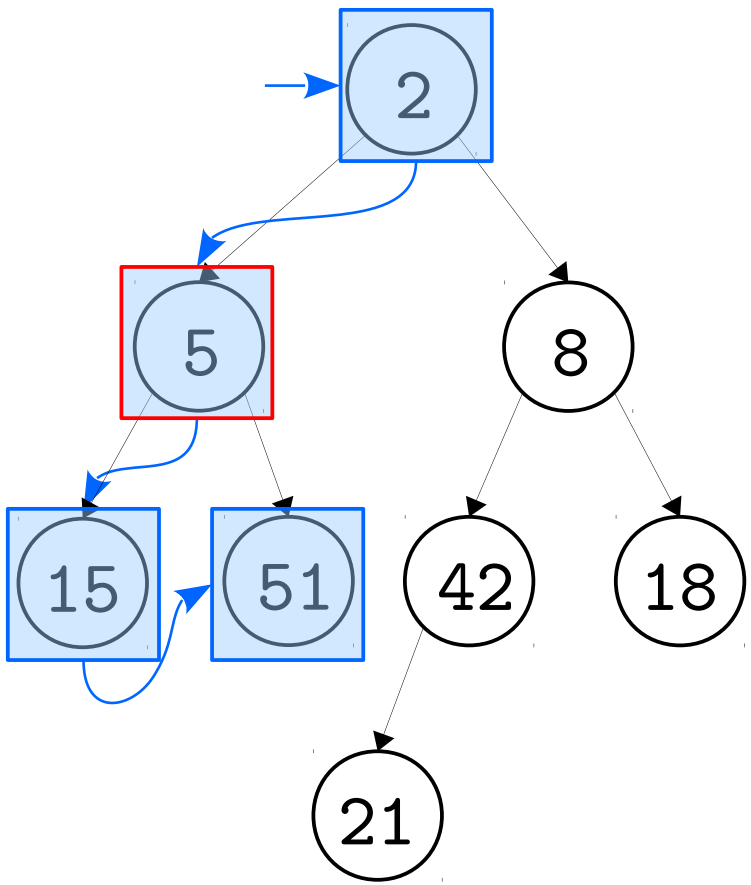
Depth-first search example



- start from 2
- go left (subtree with root 5)
  - start from 5
  - go left (subtree with root 15) → **done**
    - start from 15
    - cannot go left → **done**
    - cannot go right → **done**
  - go right (subtree with root 51) → **done**
    - start from 51
    - cannot go left → **done**
    - cannot go right → **done**

Algorithm:
- start from the root
- recursively search left subtree
- recursively search right subtree

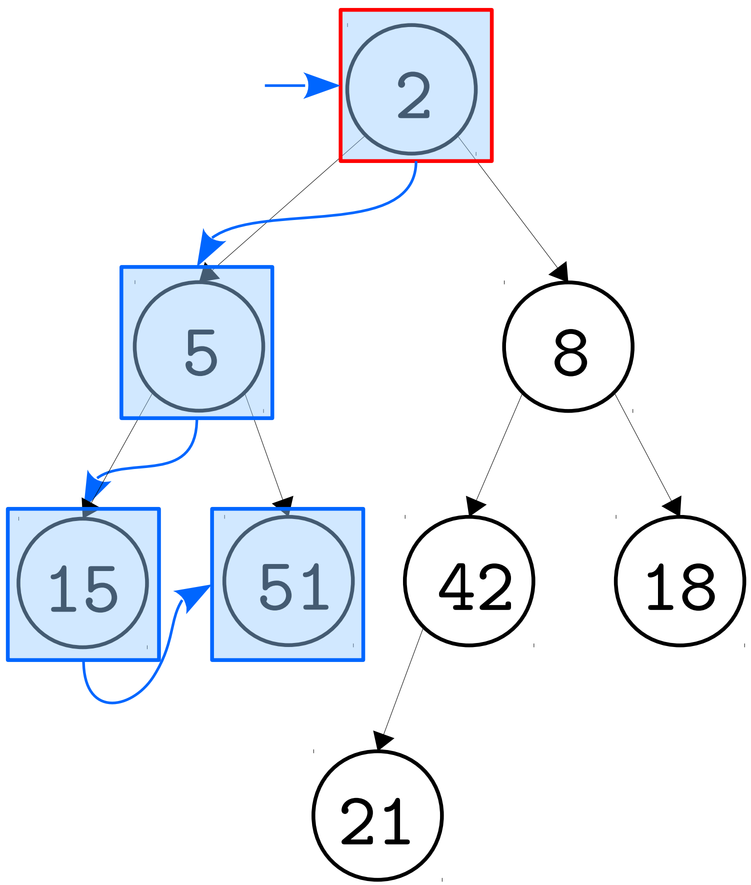
Depth-first search example



- start from 2
- go left (subtree with root 5) → **done**
  - start from 5
  - go left (subtree with root 15) → **done**
    - start from 15
    - cannot go left → **done**
    - cannot go right → **done**
  - go right (subtree with root 51) → **done**
    - start from 51
    - cannot go left → **done**
    - cannot go right → **done**

Algorithm:
- start from the root
- recursively search left subtree
- recursively search right subtree

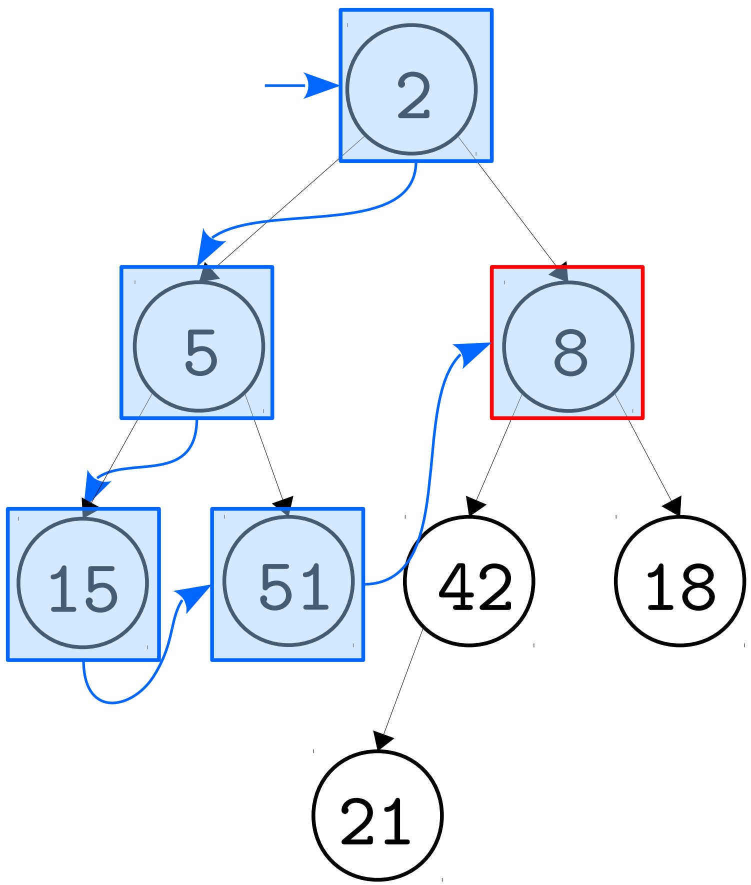
Depth-first search example



- start from 2
- go left (subtree with root 5) → **done**
- go right (subtree with root 8)

Algorithm:
- start from the root
- recursively search left subtree
- recursively search right subtree

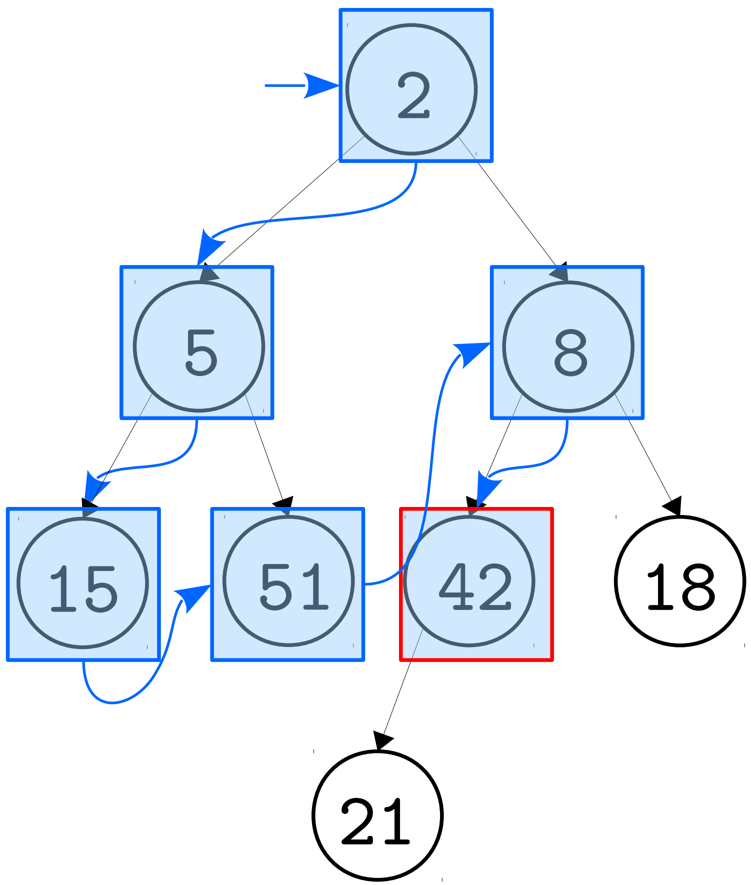
Depth-first search example



- start from 2
- go left (subtree with root 5) → **done**
- go right (subtree with root 8)
  - start from 8

Algorithm:
- start from the root
- recursively search left subtree
- recursively search right subtree

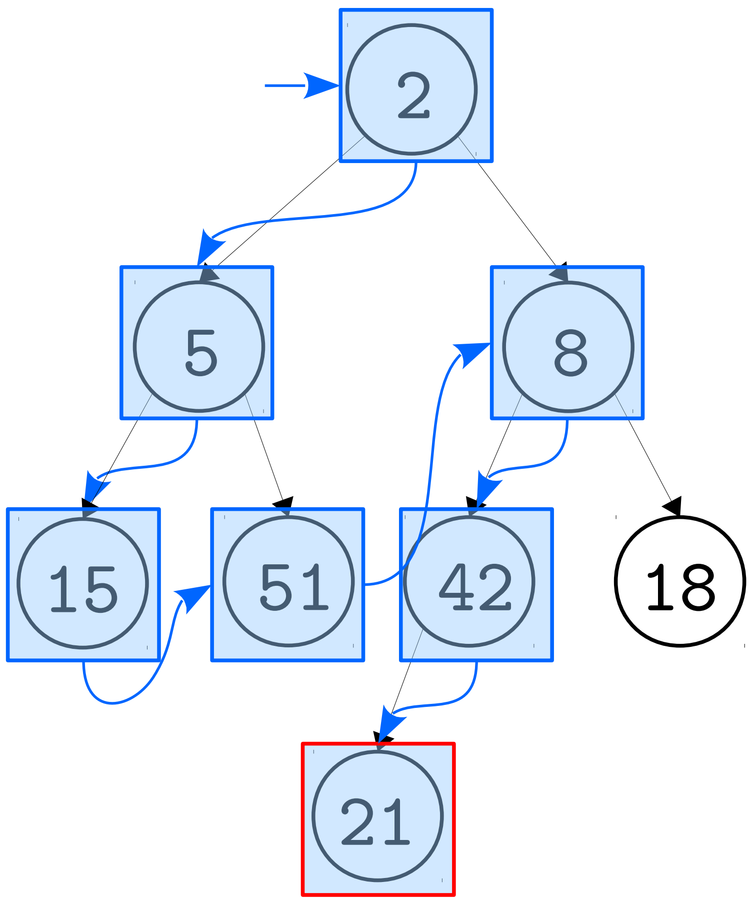
Depth-first search example



- start from 2
- go left (subtree with root 5) → **done**
- go right (subtree with root 8)
  - start from 8
  - go left (subtree with root 42)
    - start from 42

Algorithm:
- start from the root
- recursively search left subtree
- recursively search right subtree

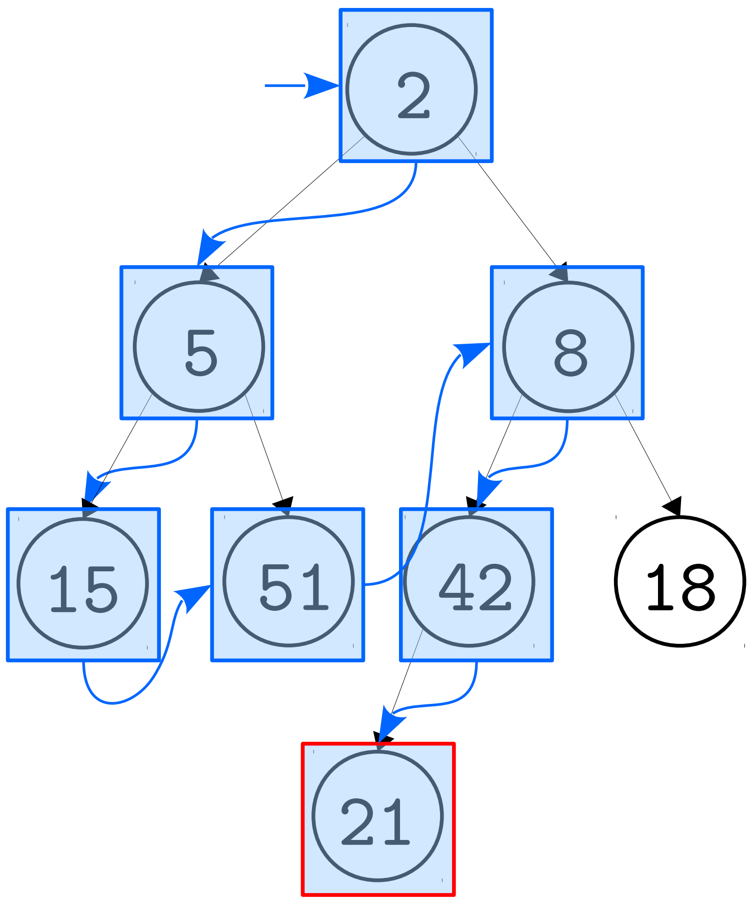
Depth-first search example



- start from 2
- go left (subtree with root 5) → **done**
- go right (subtree with root 8)
  - start from 8
  - go left (subtree with root 42)
    - start from 42
    - go left (subtree with root 21)
      - start from 21

Algorithm:
- start from the root
- recursively search left subtree
- recursively search right subtree

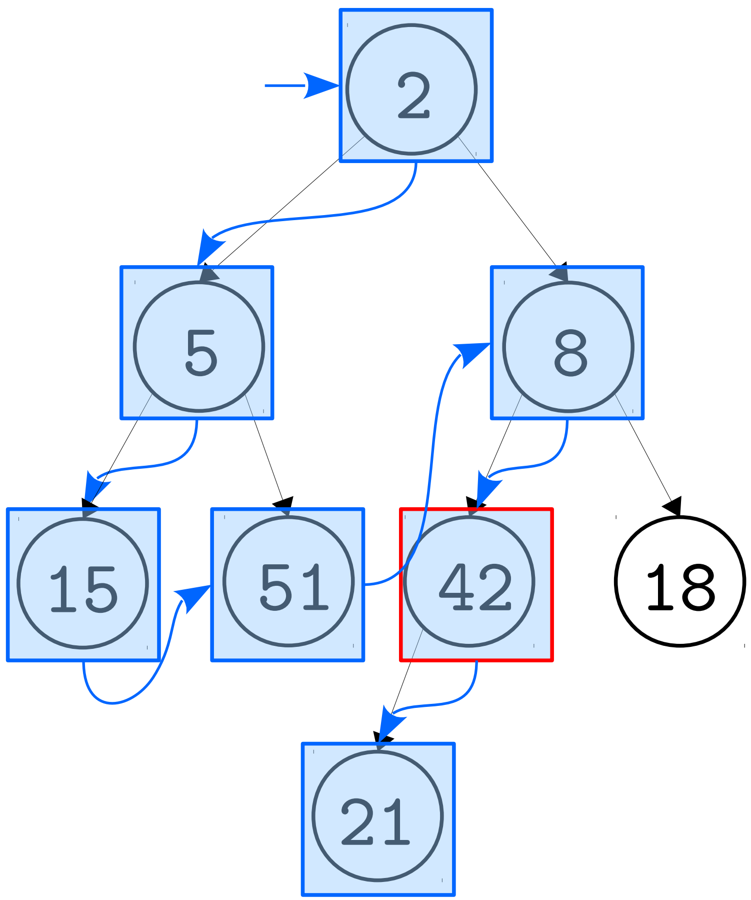
Depth-first search example



- start from 2
- go left (subtree with root 5) → **done**
- go right (subtree with root 8)
  - start from 8
  - go left (subtree with root 42)
    - start from 42
    - go left (subtree with root 21) → **done**
      - start from 21
      - cannot go left or right → **done**

Algorithm:
- start from the root
- recursively search left subtree
- recursively search right subtree

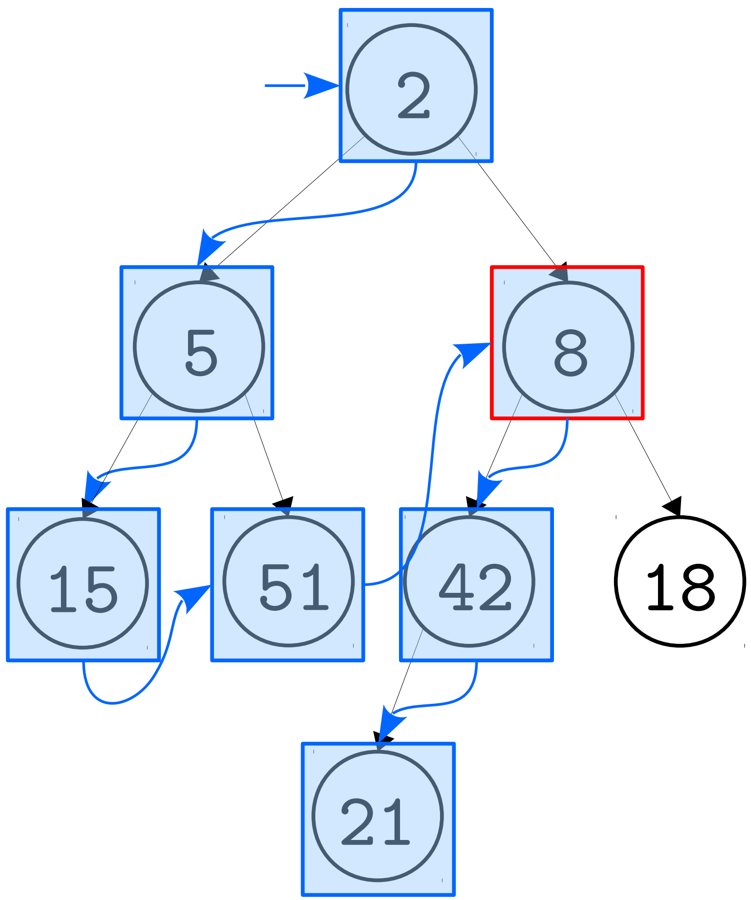
Depth-first search example



- start from 2
- go left (subtree with root 5) → **done**
- go right (subtree with root 8)
  - start from 8
  - go left (subtree with root 42) → **done**
    - start from 42
    - go left (subtree with root 21) → **done**
      - start from 21
      - cannot go left or right → **done**
    - cannot go right → **done**

Algorithm:
- start from the root
- recursively search left subtree
- recursively search right subtree

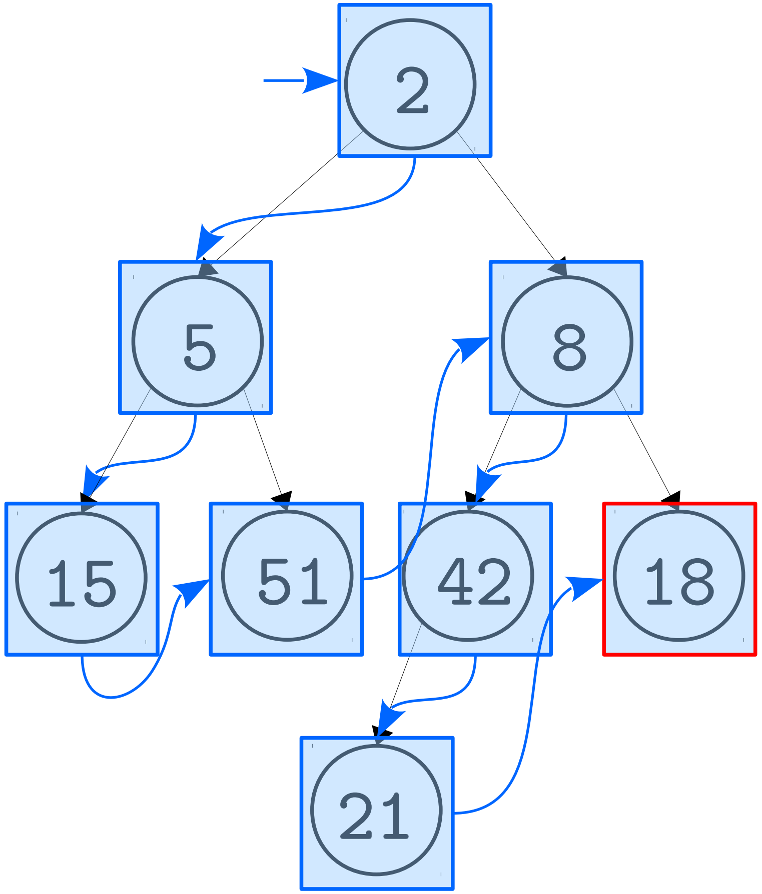
Depth-first search example



- start from 2
- go left (subtree with root 5) → **done**
- go right (subtree with root 8)
  - start from 8
  - go left (subtree with root 42) → **done**
    - start from 42
    - go left (subtree with root 21) → **done**
      - start from 21
      - cannot go left or right → **done**
    - cannot go right → **done**
  - go right (subtree with root 18)

Algorithm:
- start from the root
- recursively search left subtree
- recursively search right subtree

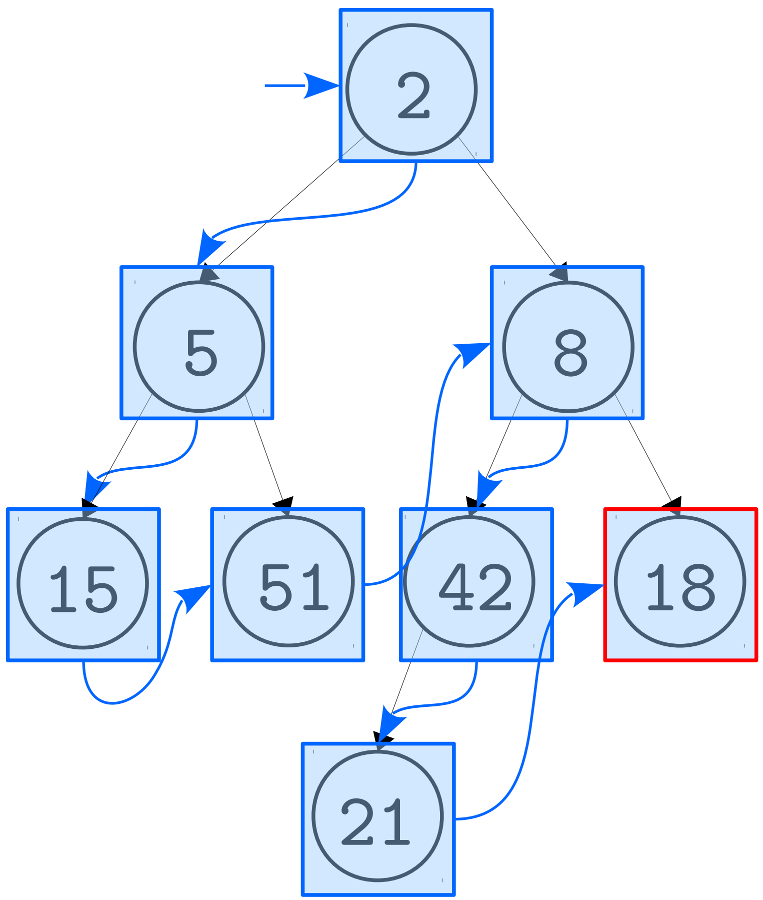
Depth-first search example



- start from 2
- go left (subtree with root 5) → **done**
- go right (subtree with root 8)
  - start from 8
  - go left (subtree with root 42) → **done**
    - start from 42
    - go left (subtree with root 21) → **done**
      - start from 21
      - cannot go left or right → **done**
    - cannot go right → **done**
  - go right (subtree with root 18)
    - start from 18

Algorithm:
- start from the root
- recursively search left subtree
- recursively search right subtree

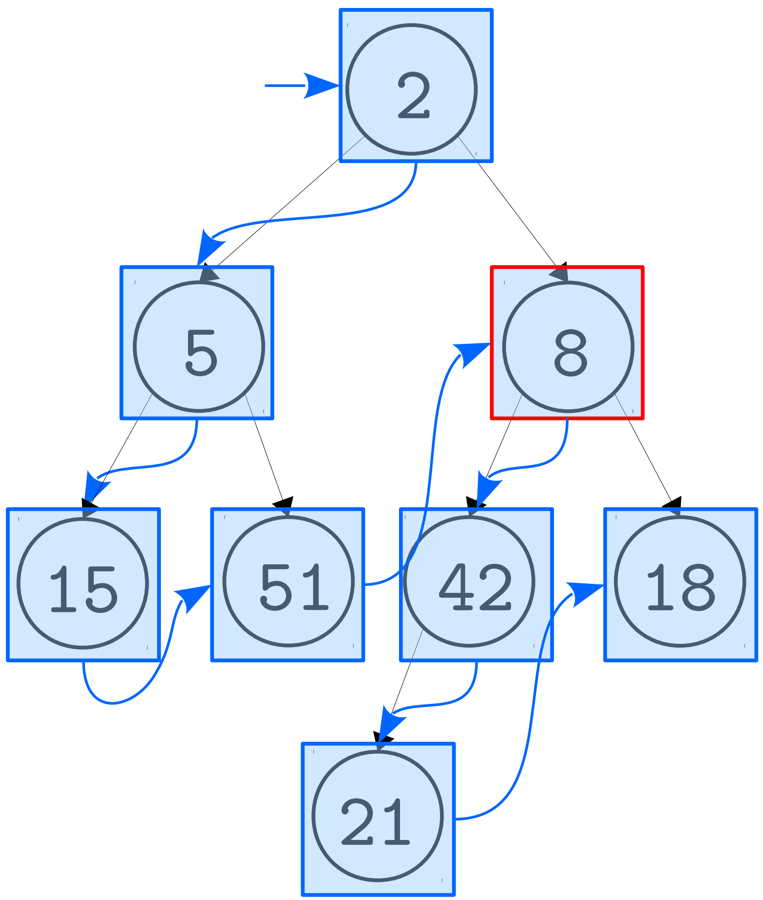
Depth-first search example



- start from 2
- go left (subtree with root 5) → **done**
- go right (subtree with root 8)
  - start from 8
  - go left (subtree with root 42) → **done**
    - start from 42
    - go left (subtree with root 21) → **done**
      - start from 21
      - cannot go left or right → **done**
    - cannot go right → **done**
  - go right (subtree with root 18) → **done**
    - start from 18
    - cannot go left or right → **done**

Algorithm:
- start from the root
- recursively search left subtree
- recursively search right subtree

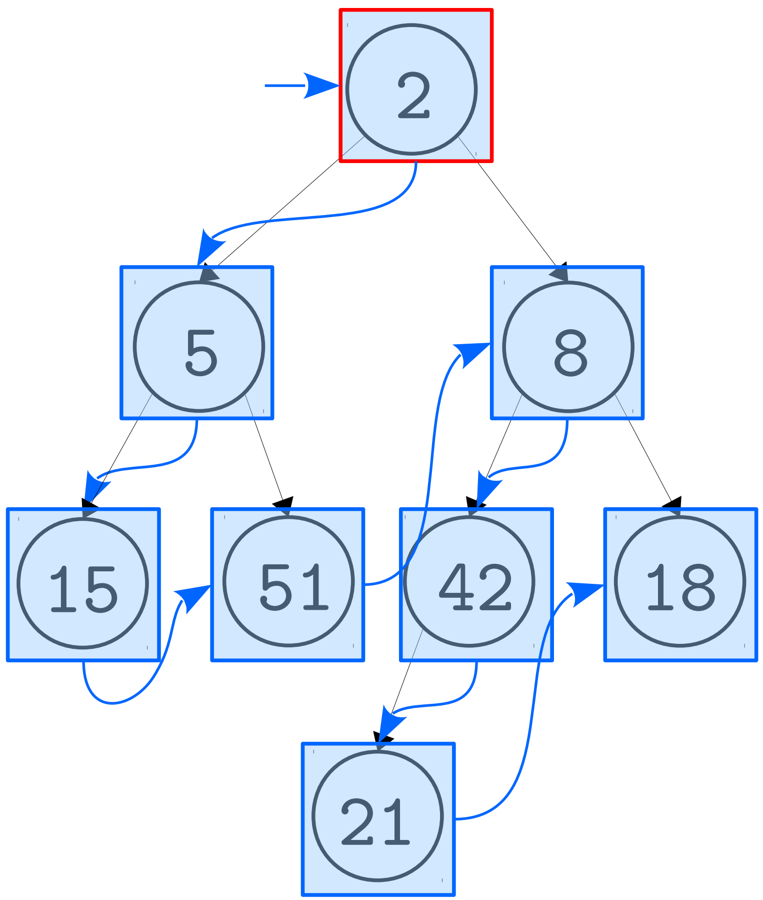
Depth-first search example



- start from 2
- go left (subtree with root 5) → **done**
- go right (subtree with root 8) → **done**
  - start from 8
  - go left (subtree with root 42) → **done**
    - start from 42
    - go left (subtree with root 21) → **done**
      - start from 21
      - cannot go left or right → **done**
    - cannot go right → **done**
  - go right (subtree with root 18) → **done**
    - start from 18
    - cannot go left or right → **done**

Algorithm:
- start from the root
- recursively search left subtree
- recursively search right subtree

Depth-first search example



Algorithm:
- start from the root
- recursively search left subtree
- recursively search right subtree

→ **done**

- start from 2
- go left (subtree with root 5) → **done**
- go right (subtree with root 8) → **done**
  - start from 8
  - go left (subtree with root 42) → **done**
    - start from 42
    - go left (subtree with root 21) → **done**
      - start from 21
      - cannot go left or right → **done**
    - cannot go right → **done**
  - go right (subtree with root 18) → **done**
    - start from 18
    - cannot go left or right → **done**

Example: searching for an element in a tree

Depth-first search (DFS) of a tree is easy to implement using recursion.

For example, here is how we can search a tree `t` for the value `d`:

```
def search(t, d):

    if t == None: return False
    if t.data == d: return True
    if search(t.left,d): return True
    return search(t.right,d)
```

if `t` is None, there is nothing to search, so return False

if the data in `t` is `d` then we found it, so return True

otherwise, go and search recursively for `d` in the left subtree, and if `d` is found there then return True

if `d` not found in the left subtree then go and search for it in the right subtree, and return whatever that search returns

Example: print all elements of a tree

DFS is a general traversal algorithm and can be used for any task that requires going through all the elements of the tree.

E.g. we can write the following function to print all the elements of a tree using DFS:

```
def printElemsDFS(t):
    if t == None: return
    print(t.data)
    printElemsDFS(t.left)
    printElemsDFS(t.right)
```

Print a tree in a string

We can print a whole tree `t` in a string using the following representation:

```
t.data -> [<string for t.left>, <string for t.right>]
```

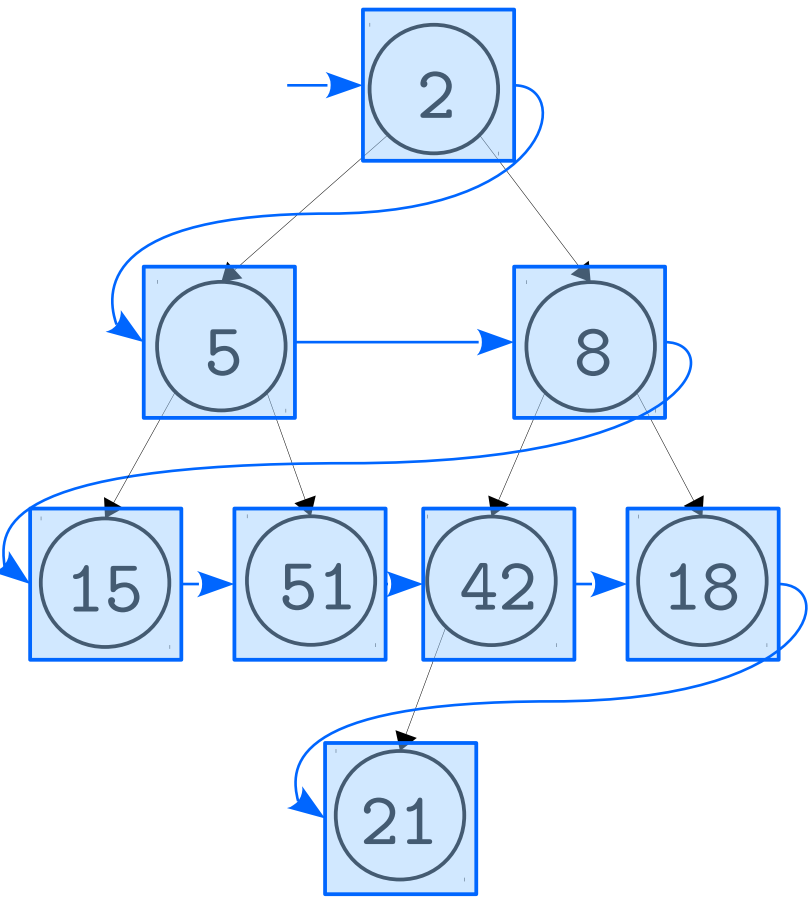
where the strings for `t.left` and `t.right` are produced in the same way, i.e. recursively.

Here is a function implementing this:

```
def toStringDFS(t):
    if t == None: return "None"
    st = str(t.data)+" -> ["
    st += toStringDFS(t.left)+", "
    st += toStringDFS(t.right)+"]"
    return st
```

Breadth-first search

Breadth-first search goes through the nodes of the tree level-by-level and left-to-right:



What algorithm can we use for this?

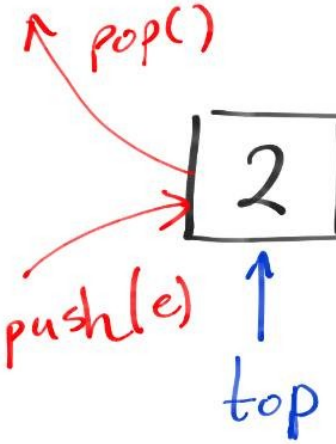
- we need a queue to store to-visit nodes

Idea:

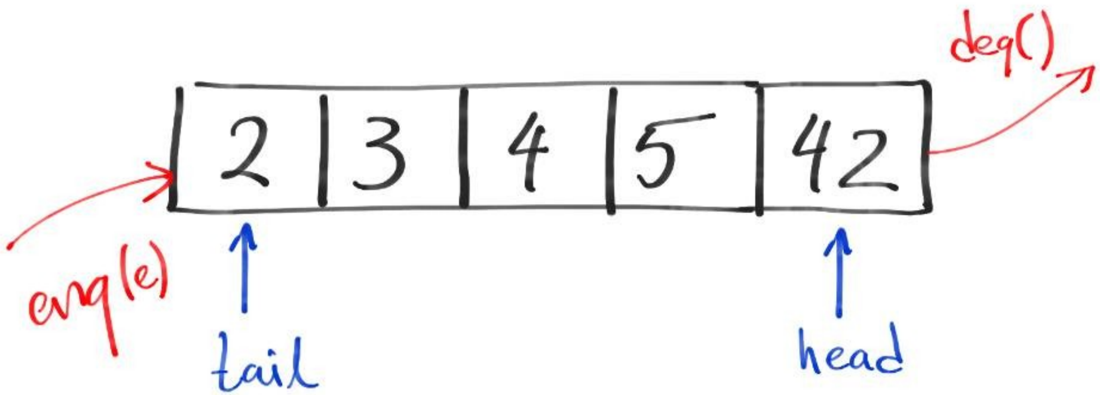
- start from the root 2
- put all children of 2 in a queue
- then keep doing this:
    - go to next node in the queue
    - dequeue it and enqueue all its children

Queues and Stacks reminder

Stack:



Queue:

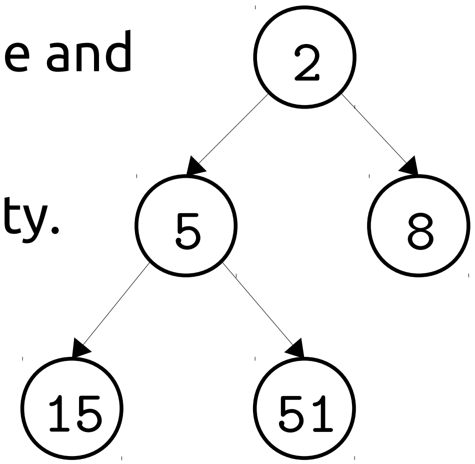


Breadth-first search code

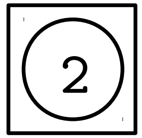
```
def searchBF(t, d):
    q = Queue()
    q.enq(t)
    while q.size() > 0:
        ptr = q.deq()
        if ptr == None:
            continue
        if ptr.data == d:
            return True
        q.enq(ptr.left)
        q.enq(ptr.right)
    return False
```

Note: the `continue` command makes us go to the next iteration of the while loop (i.e. we skip the 4 lines after it)

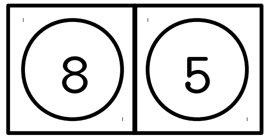
For example, let `t` be this tree and suppose we search for 42:



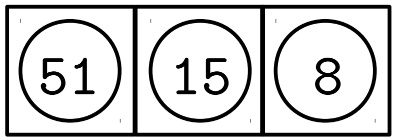
- the queue `q` is initially empty.
- we then enqueue `t` in it:



- we next dequeue the node 2 and, since its data is not 42, enqueue its children:



- we next dequeue the node 5 and, since its data is not 42, enqueue its children:



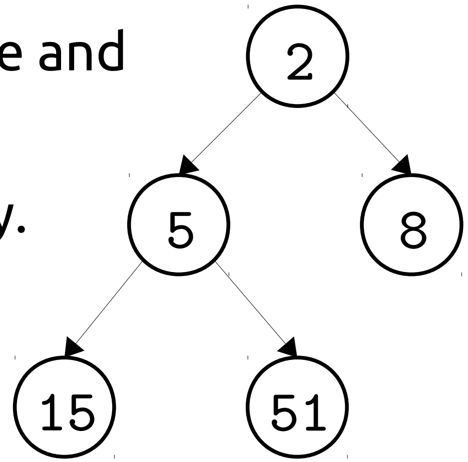
- we next dequeue the node 8, etc.

What if we used a stack?

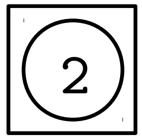
We would get again depth-first search!

```
def searchDF(t, d):
    s = Stack()
    s.push(t)
    while s.size() > 0:
        ptr = s.pop()
        if ptr == None:
            continue
        if ptr.data == d:
            return True
        s.push(ptr.right)
        s.push(ptr.left)
    return False
```

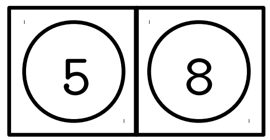
For example, let `t` be this tree and suppose we search for 42:



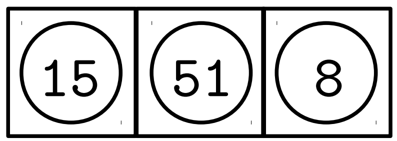
- the stack `s` is initially empty.
- we then push `t` in it:



- we next pop the node 2 and, since its data is not 42, push its children:



- we next pop the node 5 and, since its data is not 42, push its children:



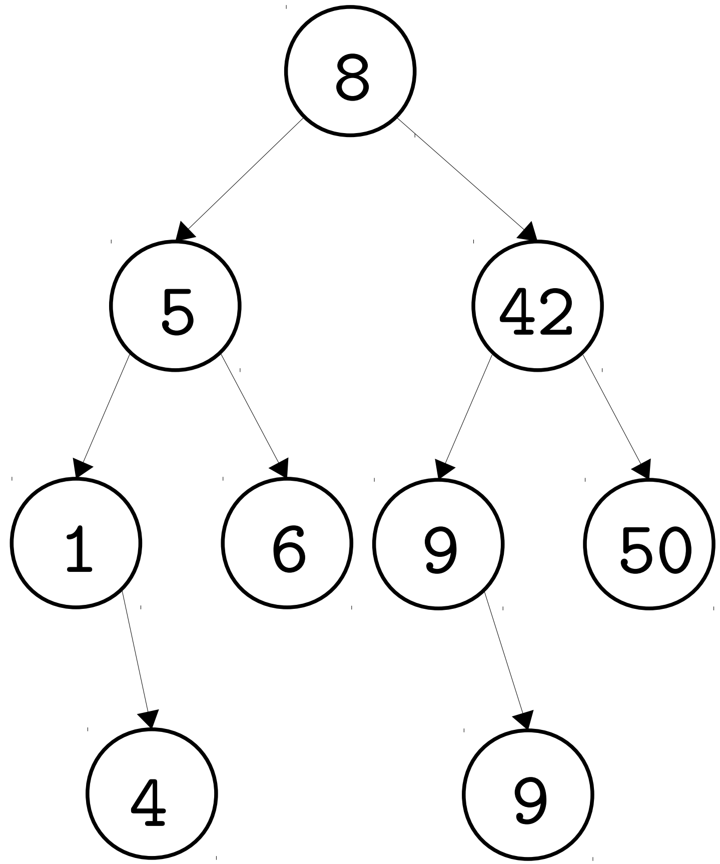
- we next pop the node 15, etc.

Application: Binary Search Trees (BSTs)

Binary search trees are binary trees whose nodes are ordered in a very specific way (the **BST discipline**).

From a given node `t`:

- all nodes on the left of `t` (i.e. the left child of `t`, and all its children, and all its children's children, etc.) have data values smaller than that of `t`
- all nodes on the right of `t` (i.e. the right child of `t`, and all its children, and all its children's children, etc.) have data values greater or equal than that of `t`

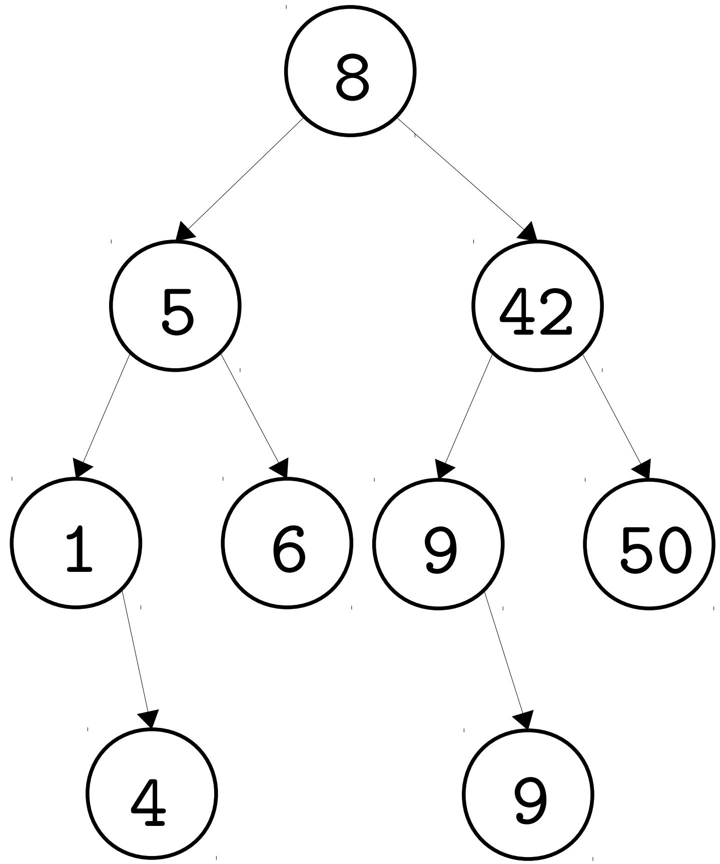


Properties of BSTs

The BST discipline is extremely useful for searching an element in the tree – it basically allows us to do binary search.

Other properties:

- where is the smallest element of the tree?
- the largest?
- how can we find how many times does an element occur?
- how many nodes do we need to look at before finding an element we are searching for, or figuring out it is not there?

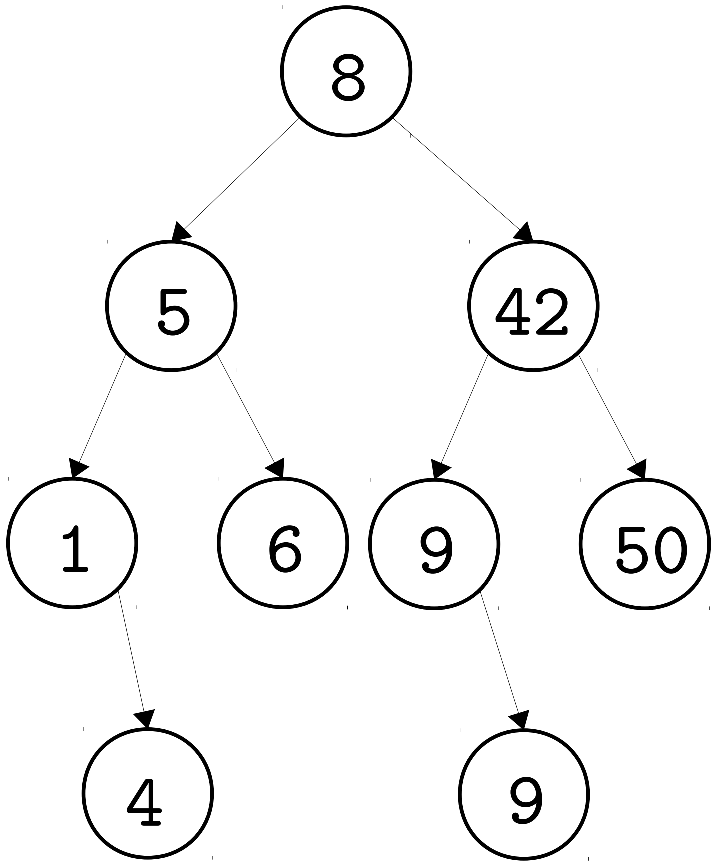


Searching in BSTs

Searching in BSTs is done similarly to a binary search!

Suppose we want to search this tree for the value 9.

- we start from the root of the tree
  - since the value we are searching is greater than the value of the root, we move right
- now our node is 42
  - since the value we are searching is smaller than 42, we move left
- now our node is 9
  - since the value we are searching is in fact 9, we stop and return True (meaning: the element was found)

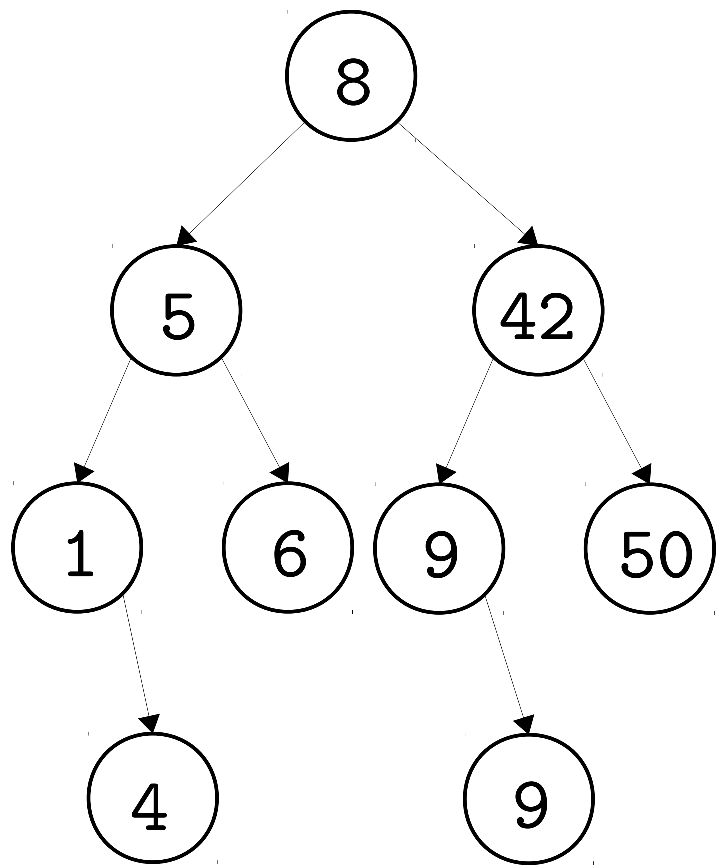


Searching in BSTs

Searching in BSTs is done similarly to a binary search!

Suppose we want to search this tree for the value 19.

- we start from the root of the tree
  - since the value we are searching is greater than the value of the root, we move right
- now our node is 42
  - since the value we are searching is smaller than 42, we move left
- now our node is 9
  - since the value we are searching is greater than 9, we move right
- now our node is 9, so we move right
- now our node is None, so we return False (i.e. 19 was not found)



A class for BSTs

We will be writing an implementation of BSTs as we go along explaining how they work. So, we can start with:

```
class BTNode:
    def __init__(self,d,l,r):
        self.data = d
        self.left = l
        self.right = r

class BST:
    def __init__(self):
        self.root = None
        self.size = 0
```

A class for BSTs

Next we implement searching as we described earlier:

```
def search(self, d):
    ptr = self.root
    while ptr != None:
        if d == ptr.data:
            return True
        if d < ptr.data:
            ptr = ptr.left
        else:
            ptr = ptr.right
    return False
```

set `ptr` to the root of the BST

do binary search using `ptr`:

- if `ptr` contains `d` then we found `d` and return True
- otherwise:
  - if `d < pt.data` then we need to go left, so we set `ptr` to `ptr.left`
  - otherwise, we need to go right, so we set `ptr` to `ptr.right`

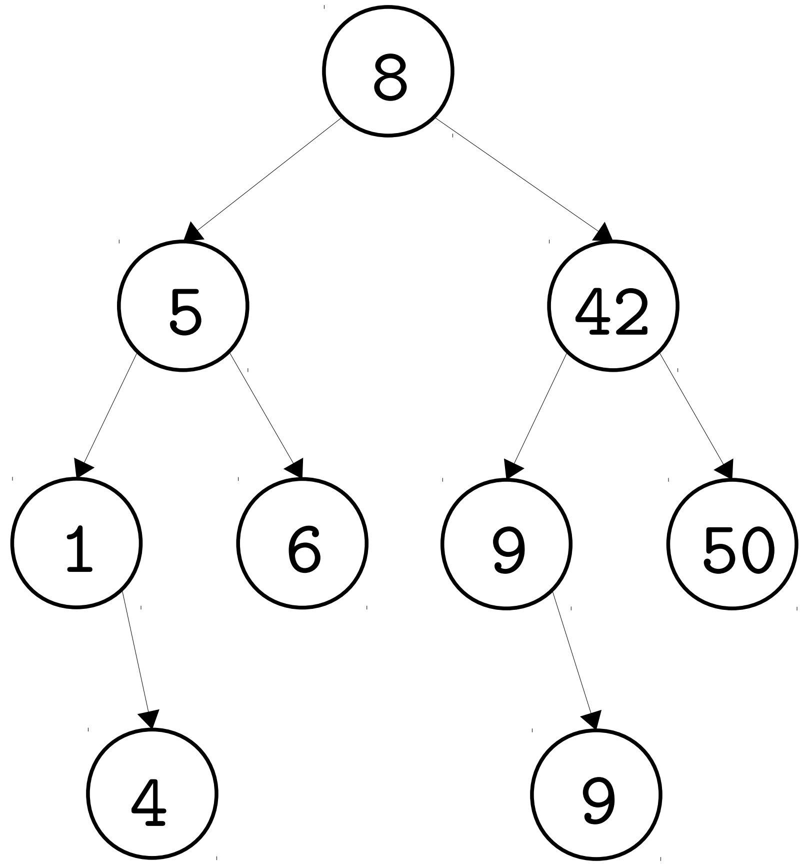
If `ptr` reaches None then `d` is not in the BST so we return False

Adding to BSTs

To add a new element in a BST we look for the right position where to insert using the same idea as in searching:

Suppose we want to add in this tree the value 7.

- we start from the root of the tree
  - since the value to insert is smaller than the value of the root, we move left
- now our node is 5
  - since the value to insert is greater than 5, we move right
- now our node is 6
  - since the value to insert is greater than 6, we need to move right
  - since right is None -> **insert here**

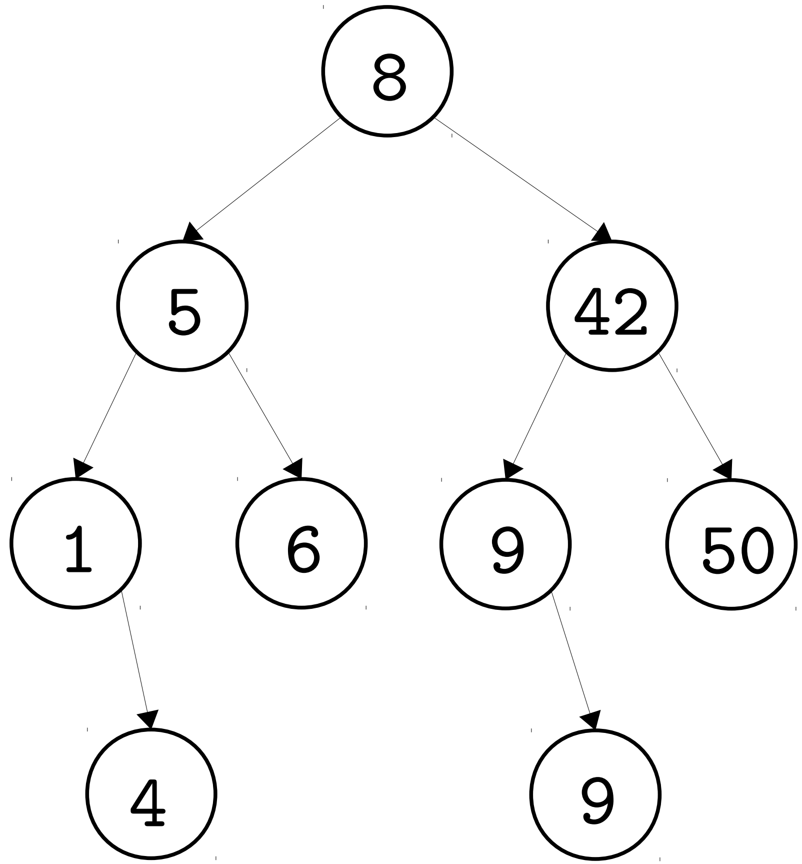


Adding to BSTs

To add a new element in a BST we look for the right position where to insert using the same idea as in searching:

Suppose we want to add in this tree the value 42.

- we start from the root of the tree
  - since the value to insert is greater than the value of the root, we move right
- now our node is 42
  - since the value to insert is equal to 42, we move right
- now our node is 50
  - since the value to insert is smaller than 50, we need to move left
  - since left is None -> **insert here**



Implementation in the BST class

```
def add(self, d):
    if self.root == None:
        self.root = BTNode(d,None,None)
    else:
        ptr = self.root
        while True:
            if d < ptr.data:
                if ptr.left == None:
                    ptr.left = BTNode(d,None,None)
                    break
                ptr = ptr.left
            else:
                if ptr.right == None:
                    ptr.right = BTNode(d,None,None)
                    break
                ptr = ptr.right
    self.size += 1
```

if the BST is empty then we create a node with `d` and make it the root

do binary search in order to find position to add `d`, using `ptr`.

The search is done as in the `search` function only that we do not aim to find d – rather, we are looking for an empty child to insert a new node with `d`.

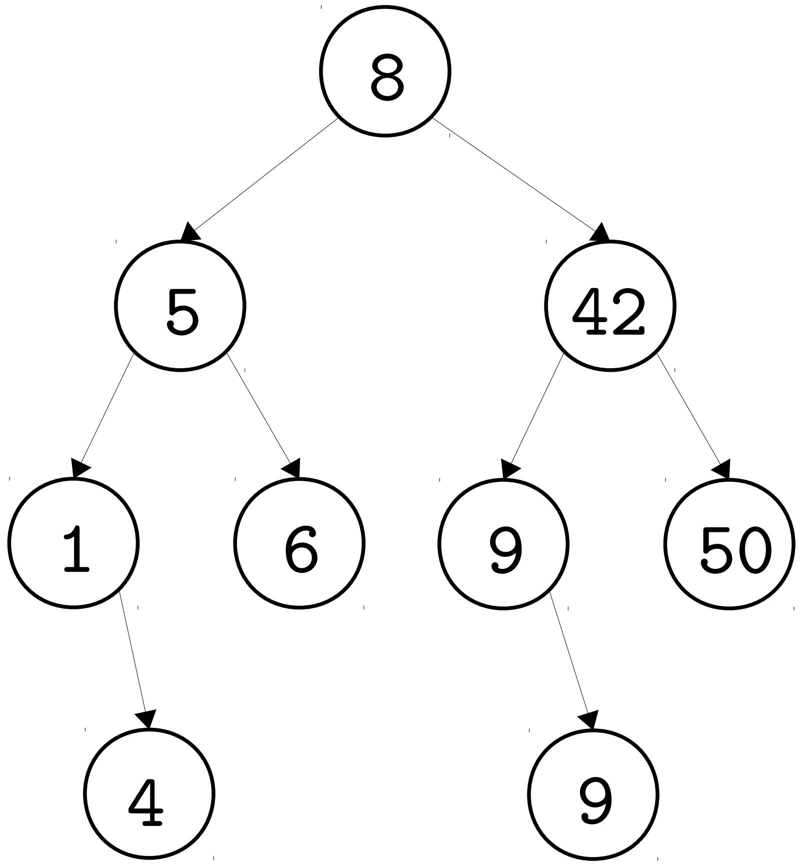
the position to add d is found when binary search tells us to move to a child that is None – this is where we add our new node with `d`

Removing from BSTs

Removing an element from a BST is more involved. The crux is that:

simply removing a node can break the tree and its BST discipline

We split node removals in 3 categories:



1. Removal of a leaf node (no children)

   - *easy*

2. Removal of a node with exactly 1 child

   - *medium*

3. Removal of a node with 2 children

   - hard

Removing from BSTs

Removing an element from a BST is more involved. The crux is that:

simply removing a node can break the tree and its BST discipline

We split node removals in 3 categories:

1. Removal of a leaf node (no children)

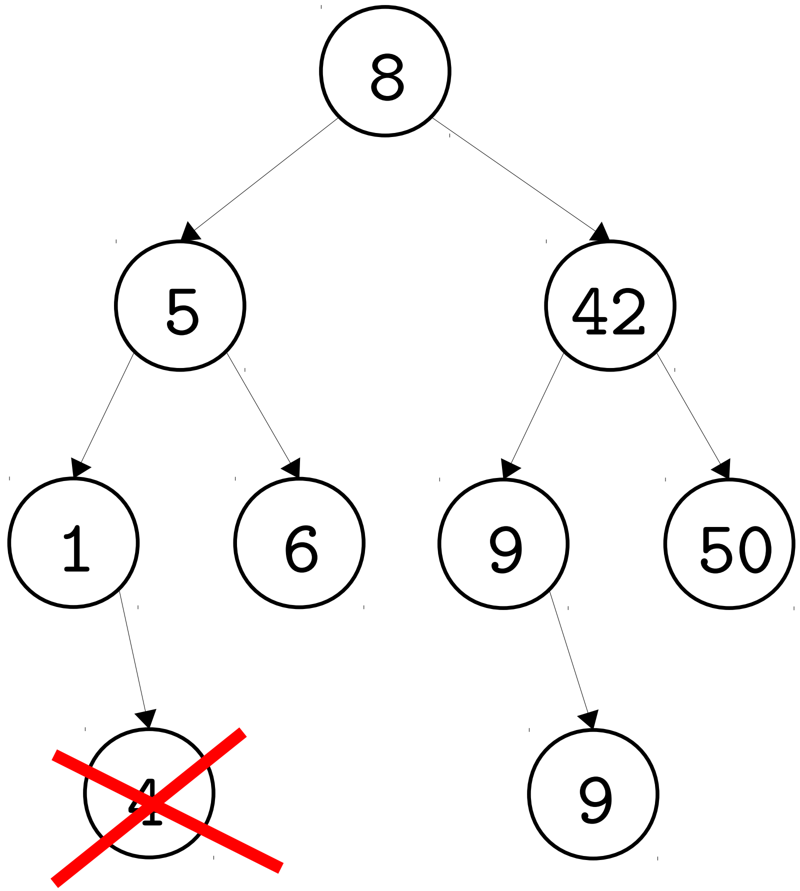
   - *just remove it!*

2. Removal of a node with exactly 1 child

   - *medium*

3. Removal of a node with 2 children

   - ***hard***



Removing from BSTs

Removing an element from a BST is more involved. The crux is that:

simply removing a node can break the tree and its BST discipline

We split node removals in 3 categories:

1. Removal of a leaf node (no children)

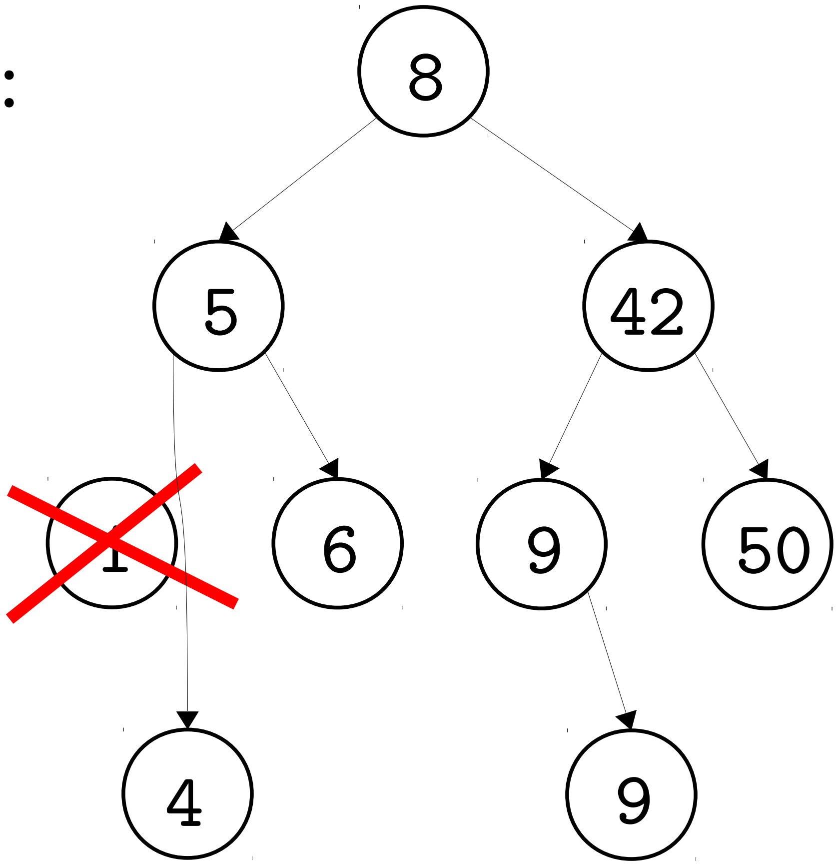
   - *just remove it!*

2. Removal of a node with exactly 1 child

   - *just bypass it!*

3. Removal of a node with 2 children

   - ***hard***



Removing from BSTs

Removing an element from a BST is more involved. The crux is that:

simply removing a node can break the tree and its BST discipline

We split node removals in 3 categories:

1. Removal of a leaf node (no children)

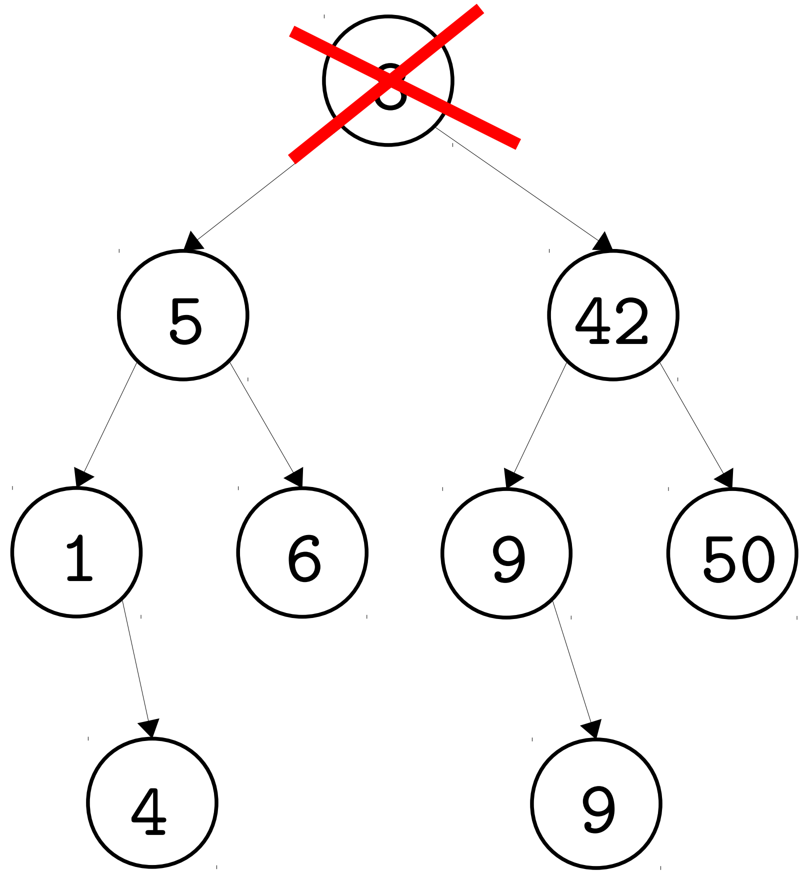
   - *just remove it!*

2. Removal of a node with exactly 1 child

   - *just bypass it!*

3. Removal of a node with 2 children

   - ??



The hard case

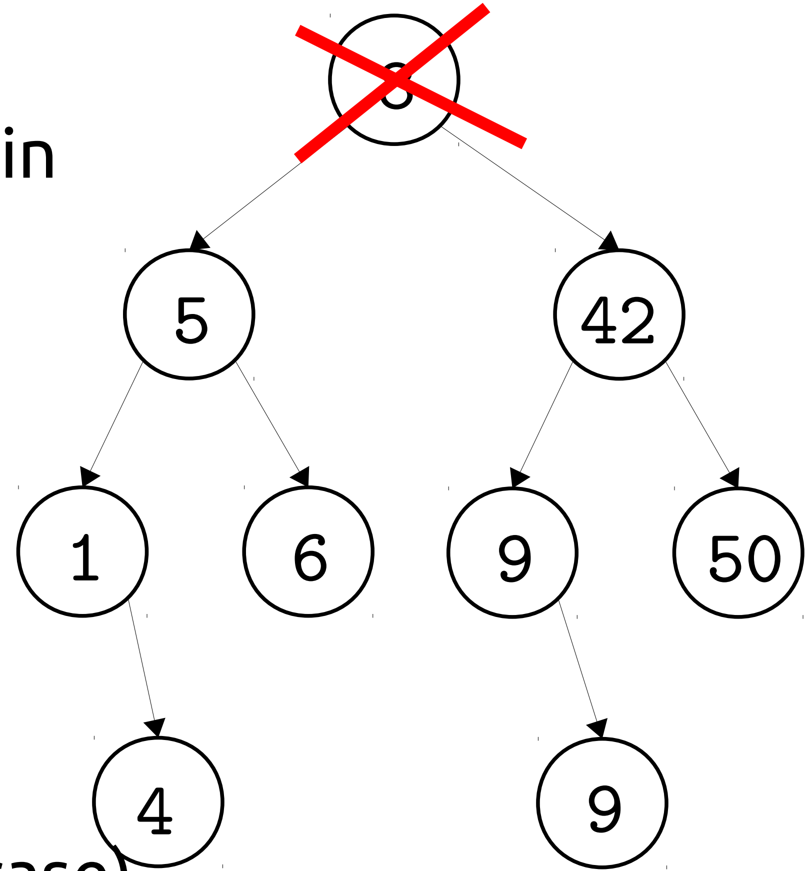
Removing a node (call it `ptr`) with 2 children is done in 3 phases:

I. we find the node whose data is the next greatest-or-equal in the tree after `ptr`

   i.e. the node with the smallest value in the right of `ptr`

   this can be found by moving right from `ptr` and then keep going left until we reach `None`

   call this node `minNode`



II. we remove `minNode` (easy/medium case)

III. we replace `ptr` with `minNode`

Implementation in the BST class

Several things need care in the `remove` function:

- the function takes as input the data to remove, not a node.
  So, it needs to first find the node to remove, if it exists, and then remove it
- removing a node requires to have a pointer to its parent, not the node itself (remember e.g. removing from a linked list)
- the root node has no parent, so extra care is needed in order to remove it

Finding the node to remove

```
def remove(self,d):
    if self.root == None:
        return
    if self.root.data == d:
        return self._removeRoot()
    parentNode = None
    currentNode = self.root
    while currentNode != None and currentNode.data != d:
        if d < currentNode.data:
            parentNode = currentNode
            currentNode = currentNode.left
        else:
            parentNode = currentNode
            currentNode = currentNode.right
    if currentNode != None:
        return self._removeNode(currentNode,parentNode)
```

Finding the node to remove

```
def remove(self,d):
    if self.root == None:
        return
    if self.root.data == d:
        return self._removeRoot()
    parentNode = None
    currentNode = self.root
    while currentNode != None and currentNode.data != d:
        if d < currentNode.data:
            parentNode = currentNode
            currentNode = currentNode.left
        else:
            parentNode = currentNode
            currentNode = currentNode.right
    if currentNode != None:
        return self._removeNode(currentNode,parentNode)
```

if the tree is empty, there is nothing to remove, so return

if the node to remove is the root, call the `_removeRoot` function and return

otherwise, search for the node to remove using binary search. Store the node to remove in `currentNode`, and its parent in `parentNode`

at the end, either the node to remove was found (so `currentNode` is not `None`) and we call `_removeNode` to remove it, or there is no such node on the tree (so `currentNode` is `None`)

The actual node removal

```
def _removeNode(self,currentNode,parentNode):
    self.size -= 1
    if currentNode.left == currentNode.right == None:
        parentNode.updateChild(currentNode,None)
    elif currentNode.left == None or currentNode.right == None:
        if currentNode.left != None:
            parentNode.updateChild(currentNode,currentNode.left)
        else:
            parentNode.updateChild(currentNode,currentNode.right)
    else:
        parentMinNode = currentNode
        minNode = currentNode.right
        while minNode.left != None:
            parentMinNode = minNode
            minNode = minNode.left
        parentMinNode.updateChild(minNode,minNode.right)
        parentNode.updateChild(currentNode,minNode)
        minNode.left = currentNode.left
        minNode.right = currentNode.right
```

The actual node removal

```
def _removeNode(self,currentNode,parentNode):
    self.size -= 1
    if currentNode.left == currentNode.right == None:
        parentNode.updateChild(currentNode,None)
    elif currentNode.left == None or currentNode.right == None:
        if currentNode.left != None:
            parentNode.updateChild(currentNode,currentNode.left)
        else:
            parentNode.updateChild(currentNode,currentNode.right)
    else:
        parentMinNode = currentNode
        minNode = currentNode.right
        while minNode.left != None:
            parentMinNode = minNode
            minNode = minNode.left
        parentMinNode.updateChild(minNode,minNode.right)
        parentNode.updateChild(currentNode,minNode)
        minNode.left = currentNode.left
        minNode.right = currentNode.right
```

first, reduce the tree size by 1 (since we are removing a node from it)

Case 1: the node to remove (i.e. `currentNode`) is a leaf – just remove it!

Case 2: the node to remove has exactly one child – just bypass it!

How? Connect the node's parent directly to the node's child

Case 3: the node to remove has both children

I. find `minNode` (the node with minimum value on the right of `currentNode`)

II. remove `minNode`

III. replace `currentNode` with `minNode`

Two missing pieces

In `BTNode`, we add this method for changing a child in a node:

```
def updateChild(self, oldChild, newChild):
    if self.left == oldChild:
        self.left = newChild
    elif self.right == oldChild:
        self.right = newChild
    else: raise Exception("updateChild error")
```

Then, back in `BST`, to remove the root node we do a hack: we temporarily add a parent to the root, remove as usual, and then discard the temporary parent:

```
def _removeRoot(self):
    parentNode = BTNode(None,self.root,None)
    self._removeNode(self.root,parentNode)
    self.root = parentNode.left
```

Exercises

1. Draw a binary tree containing the numbers:

   4,5,1,45,23,65,12,65,12,67,12

   as data, in whichever order you prefer. Next, write down the numbers of the tree you constructed, starting from the root and using:
   a) depth-first search
   b) breadth-first search
2. Let t be the root node of the tree you drew in Question 1. Using `t`, write a command that:
   a) changes the value of the node of the tree containing 1 to 42.
   b) adds a new node with value 24 as the right child of the rightmost leaf in your tree.
   c) removes the leftmost leaf in your tree.
3. Draw the binary search tree we obtain if we start from the empty tree and add consecutively the numbers:

   24,15,1,11,45,23,65,12,5,12,67,32

   Now, on your paper, perform the following:
   a) remove the node containing 1
   b) remove the node containing 11
   c) remove the root

   in the way that the BST `remove` method works.
4. Write a BST function

   ```
   def _searchNode(self, ptr, d)
   ```

   that searches using DFS the subtree starting from node `ptr` and returns the first node that contains `d`, or `None` if `d` is not stored in the subtree.
   Use `_searchNode` to define a function

   ```
   def count(self, d)
   ```

   that returns the number of times that `d` occurs in the tree.
5. Find the bug(s) in the following simpler implementation of `add` for BSTs:

   ```
   def add(self, d):
       ptr = self.root
       while ptr != None:
           if d < ptr.data:
               ptr = ptr.left
           else:
               ptr = ptr.right
       ptr = BTNode(d,None,None)
       self.size += 1
   ```
6. Implement `remove` without creating a dummy parent node in case the root is to be removed

BST code

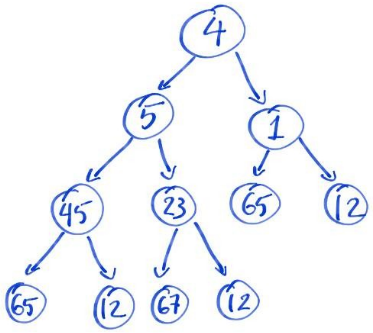
```
class BTNode:
    def __init__(self,d,l,r):
        self.data = d
        self.left = l
        self.right = r

    def updateChild(self, oldChild, newChild):
        if self.left == oldChild:
            self.left = newChild
        elif self.right == oldChild:
            self.right = newChild
        else:
            raise Exception("updateChild error")


class BST:
    def __init__(self):
        self.root = None
        self.size = 0

    def search(self, d):
        ptr = self.root
        while ptr != None:
            if d == ptr.data:
                return True
            if d < ptr.data:
                ptr = ptr.left
            else:
                ptr = ptr.right
        return False
```

```
def add(self, d):
    if self.root == None:
        self.root = BTNode(d,None,None)
    else:
        ptr = self.root
        while True:
            if d < ptr.data:
                if ptr.left == None:
                    ptr.left = BTNode(d,None,None)
                    break
                ptr = ptr.left
            else:
                if ptr.right == None:
                    ptr.right = BTNode(d,None,None)
                    break
                ptr = ptr.right
    self.size += 1

def remove(self,d):
    if self.root == None: return
    if self.root.data == d:
        return self._removeRoot()
    parentNode = None
    currentNode = self.root
    while (currentNode != None
              and currentNode.data !=d):
        if d < currentNode.data:
            parentNode = currentNode
            currentNode = currentNode.left
        else:
            parentNode = currentNode
            currentNode = currentNode.right
    if currentNode != None:
        return self._removeNode(currentNode,parentNode)
```

```
# removes the node currentNode from the tree altogether
def _removeNode(self,currentNode,parentNode):
    self.size -= 1
    # there are 3 cases to consider:
    # 1. the node to be removed is a leaf (no children)
    if currentNode.left == currentNode.right == None:
        parentNode.updateChild(currentNode,None)
    # 2. the node to be removed has exactly one child
    elif (currentNode.left == None
          or currentNode.right == None):
        if currentNode.left != None:
            parentNode.updateChild(currentNode,
                                   currentNode.left)
        else:
            parentNode.updateChild(currentNode,
                                   currentNode.right)
    # 3. the node to be removed has both children
    else:
        parentMinNode = currentNode
        minNode = currentNode.right
        while minNode.left != None:
            parentMinNode = minNode
            minNode = minNode.left
        parentMinNode.updateChild(minNode,minNode.right)
        parentNode.updateChild(currentNode,minNode)
        minNode.left = currentNode.left
        minNode.right = currentNode.right

def _removeRoot(self):
    parentNode = BTNode(None,self.root,None)
    self._removeNode(self.root,parentNode)
    self.root = parentNode.left
```

Exercises – Solutions

1. Here is one such tree:



   We can read off its elements as follows:

   a) DFS: 4,5,45,65,12,23,67,12,1,65,12

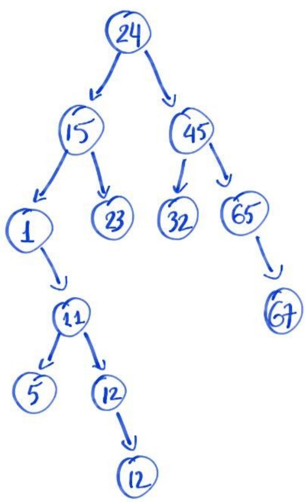
   b) BFS: 4,5,1,45,23,65,12,65,12,67,12

2. a) `t.right.data = 42`

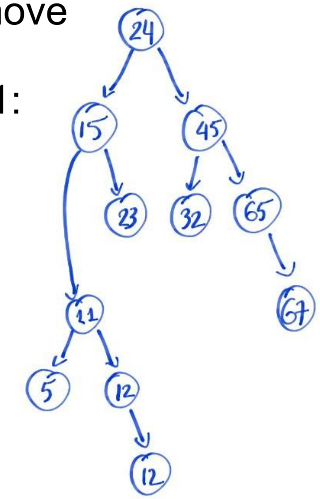
   b) `t.right.right.right = Node(24,None,None)`

   c) `t.left.left.left = None`

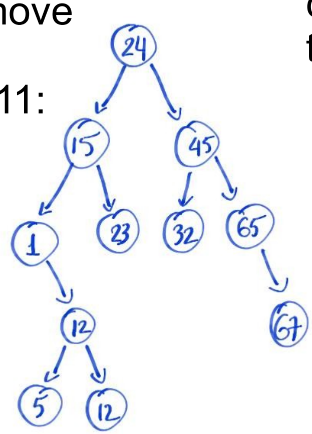
3. Here is the (unique!) BST we obtain:



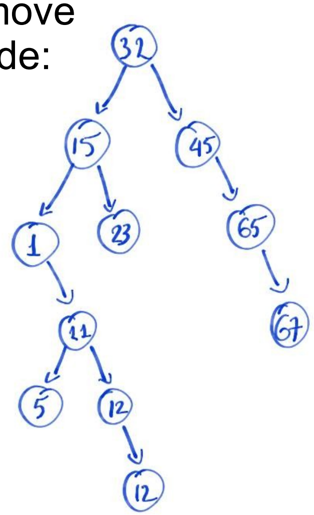
a) if we remove the node containing 1:



b) if we remove the node containing 11:



c) if we remove the root node:



4. Code given in `lecture9.ipynb`.

5. There are at least two problems with it:

   a) the while loop will find the position where the new node needs to be added, but not its parent, So, after the while loop we have no way to insert the new node in the required position.

   b) The assignment

      `ptr = BTNode(d,None,None)`

      creates the new node that we want to add but does not connect it to the tree in any way.

6. We can use an alternative function `_removeRoot2` which copies the code of `_removeNode` but uses `self.root` in place of `parentNode`. Code given in `lecture9.ipynb`.