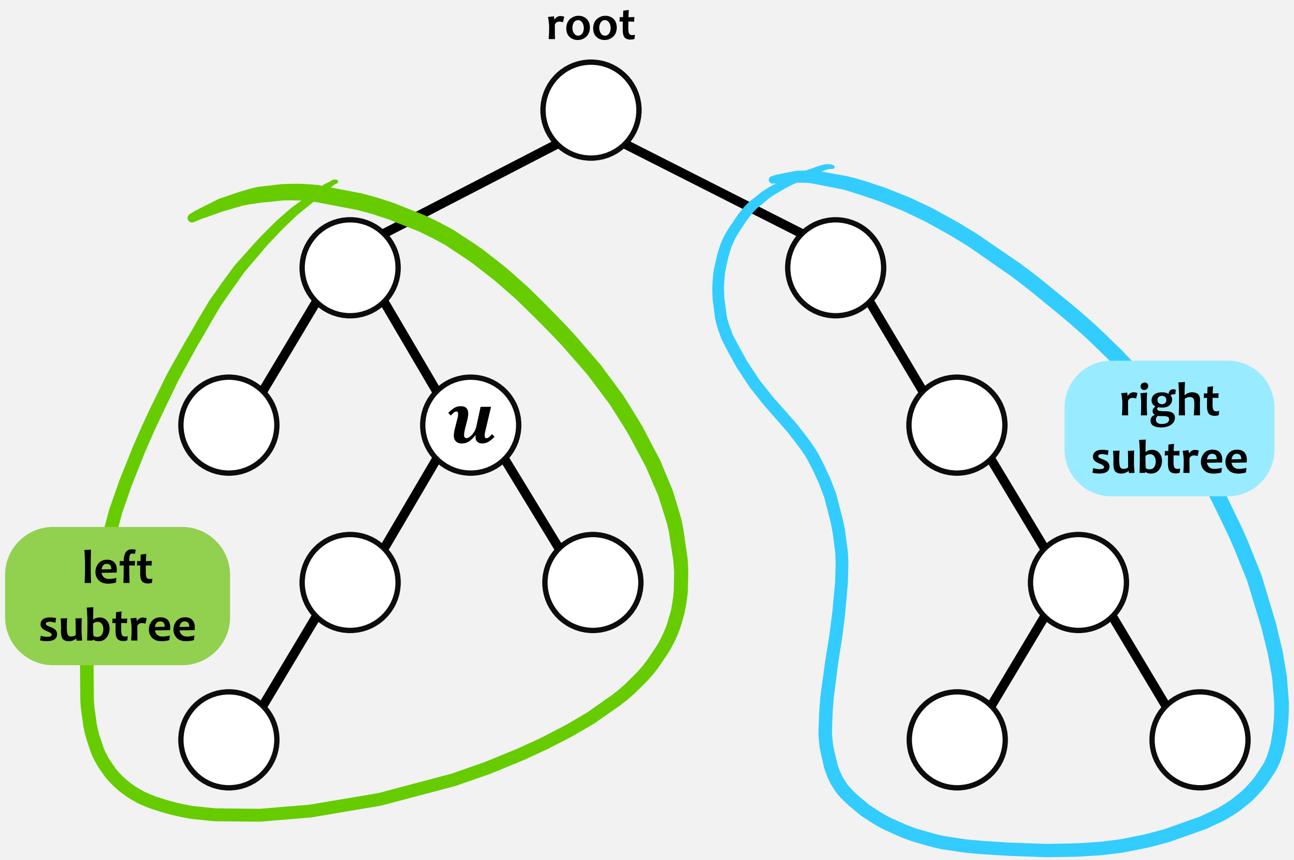
Binary Trees

Carleton University
Alina Shaikhet

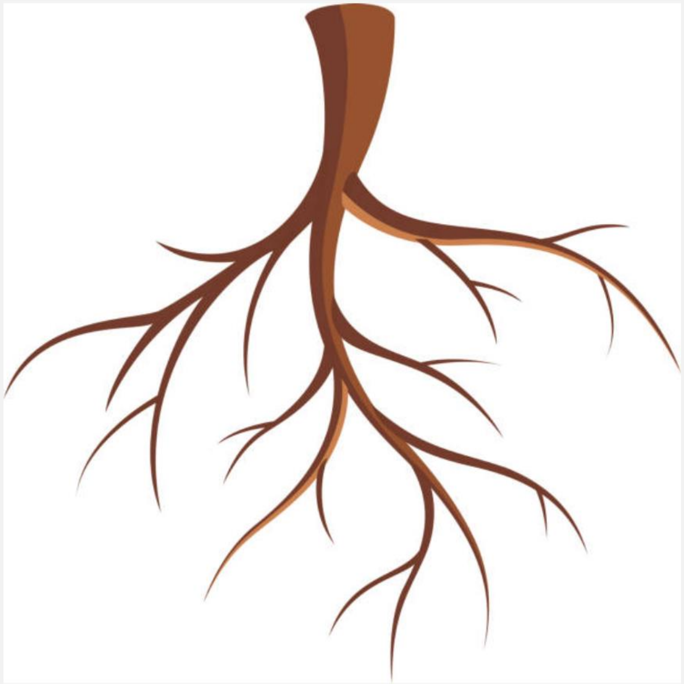
Rooted ordered binary trees

A **binary tree** is a connected, undirected, finite graph with no cycles, and no vertex of degree greater than three.

Recursive definition: **binary tree** is either empty (null), or it has a root node that has a right subtree and a left subtree. Left subtree and right subtree are also binary trees (possibly empty).



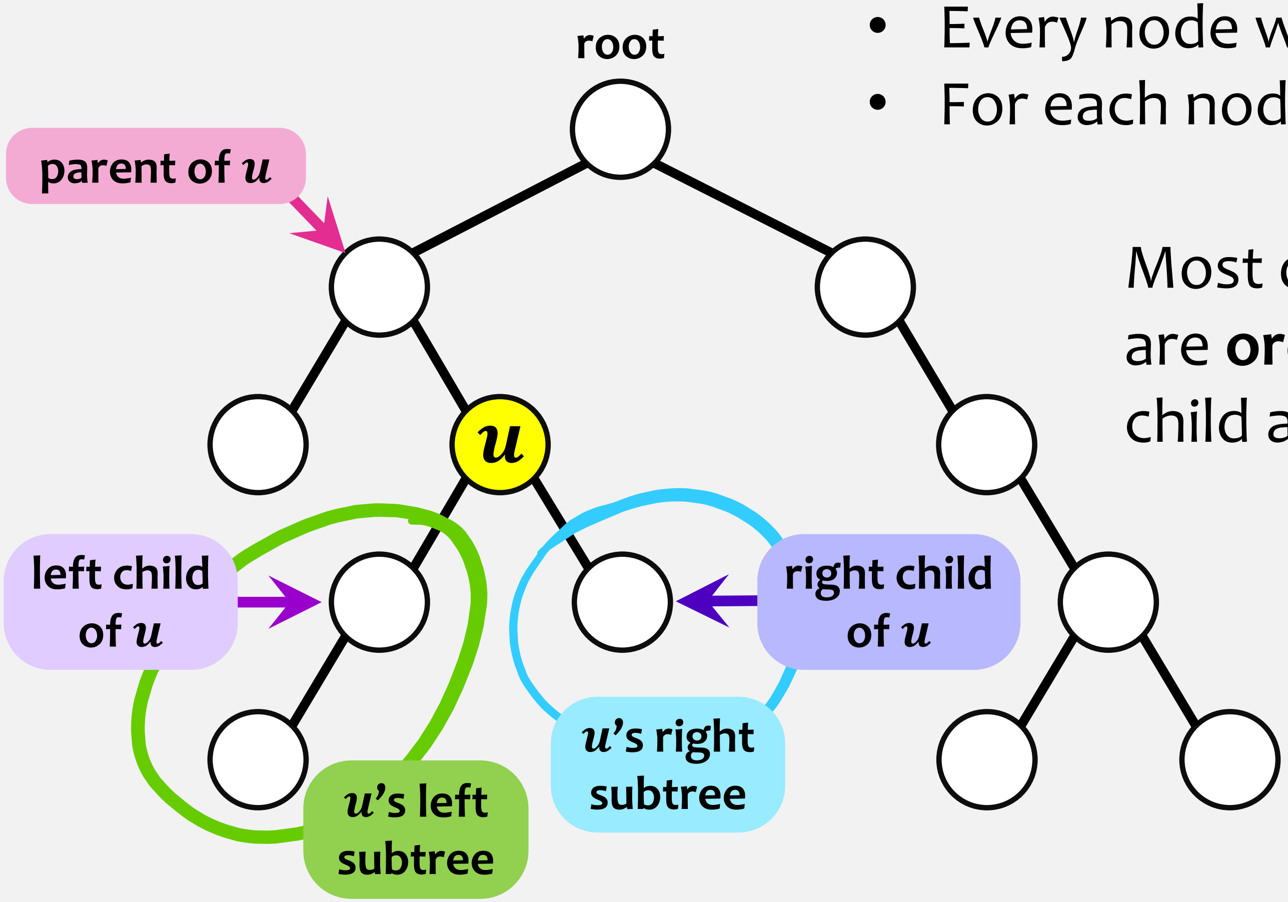
Trees in CS grow upside down



Rooted ordered binary trees

A binary tree $\boldsymbol{T}$ is a set of nodes that store elements based on a parent-child relationship:

- If $\boldsymbol{T}$ is non-empty, it has a node called the **root** of $\boldsymbol{T}$, that has no parent.
- Each node $\boldsymbol{v}$, other than the root, has a unique **parent** node $\boldsymbol{w}$.

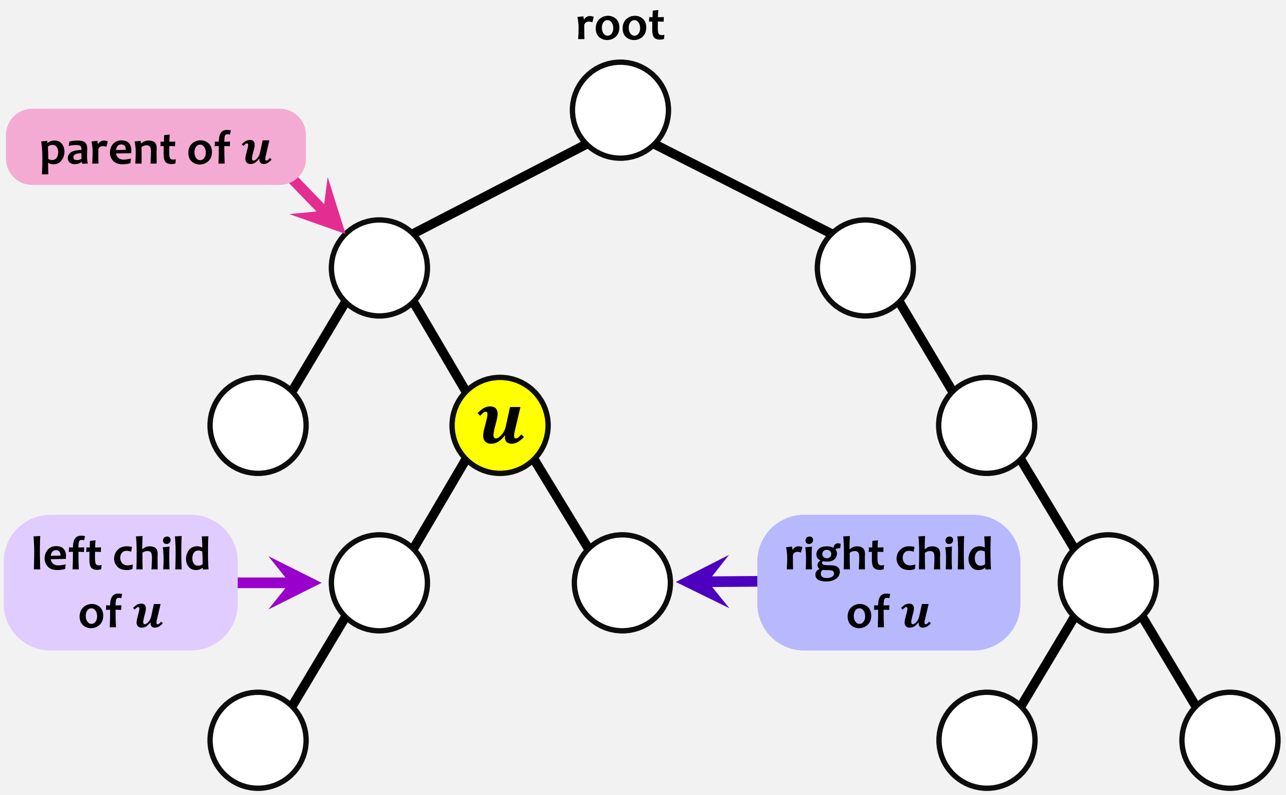


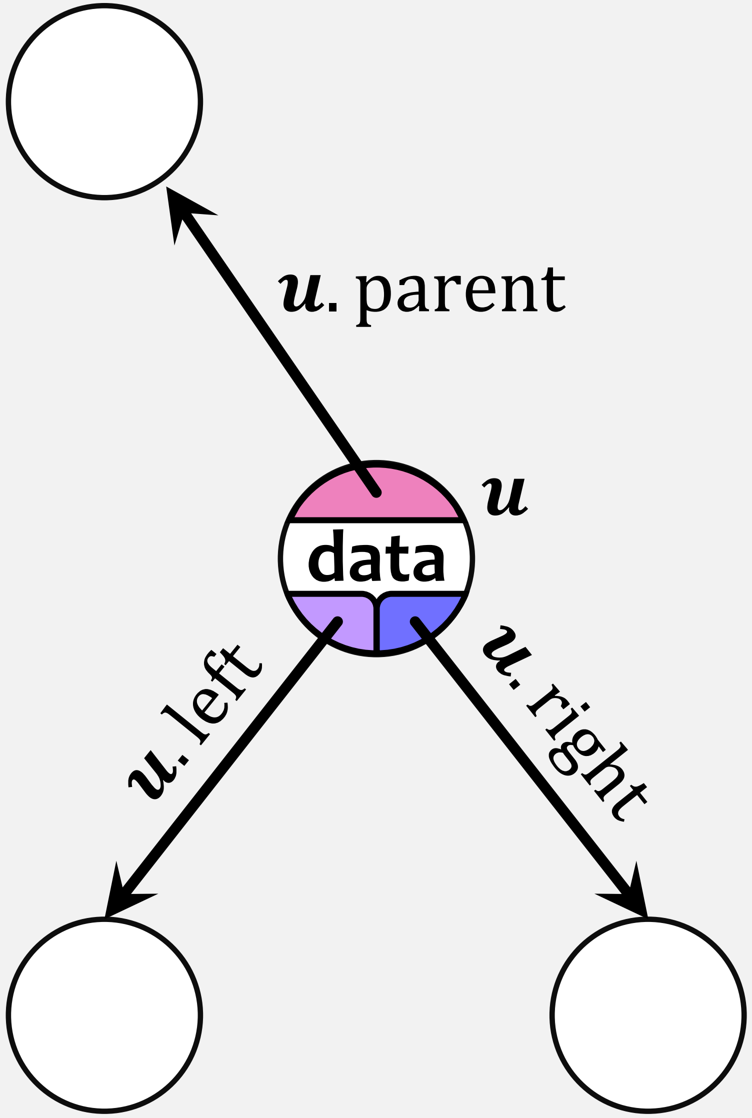
- Every node with parent $\boldsymbol{w}$ is a **child** of $\boldsymbol{w}$.
- For each node $\boldsymbol{v}$, the max number of children of $\boldsymbol{v}$ is $\mathbf{2}$.

Most of the binary trees we are interested in are **ordered**, so we distinguish between the left child and right child of $\boldsymbol{u}$ (for every node $\boldsymbol{u}$).

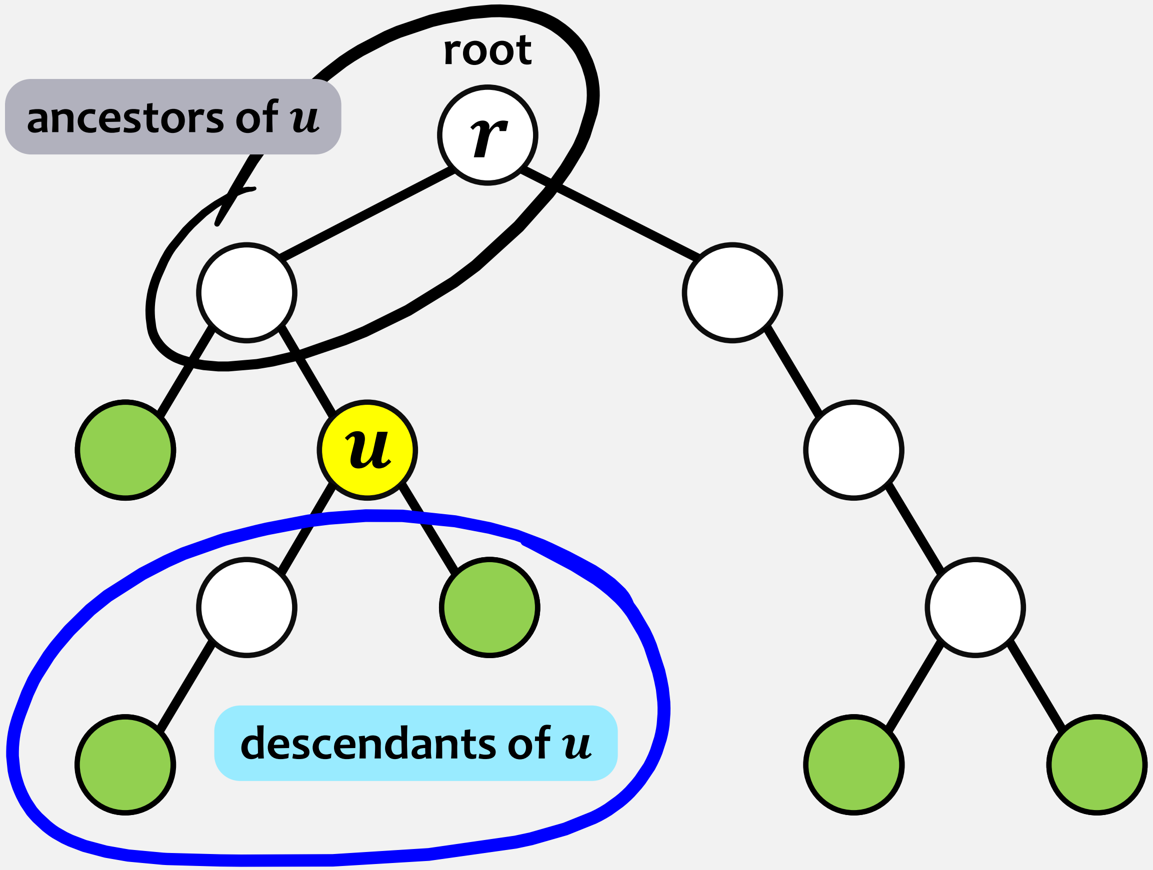
Implementation

```
public static class BTNode<Node extends BTNode<Node>> {
        public Node left;
        public Node right;
        public Node parent;
    }
```





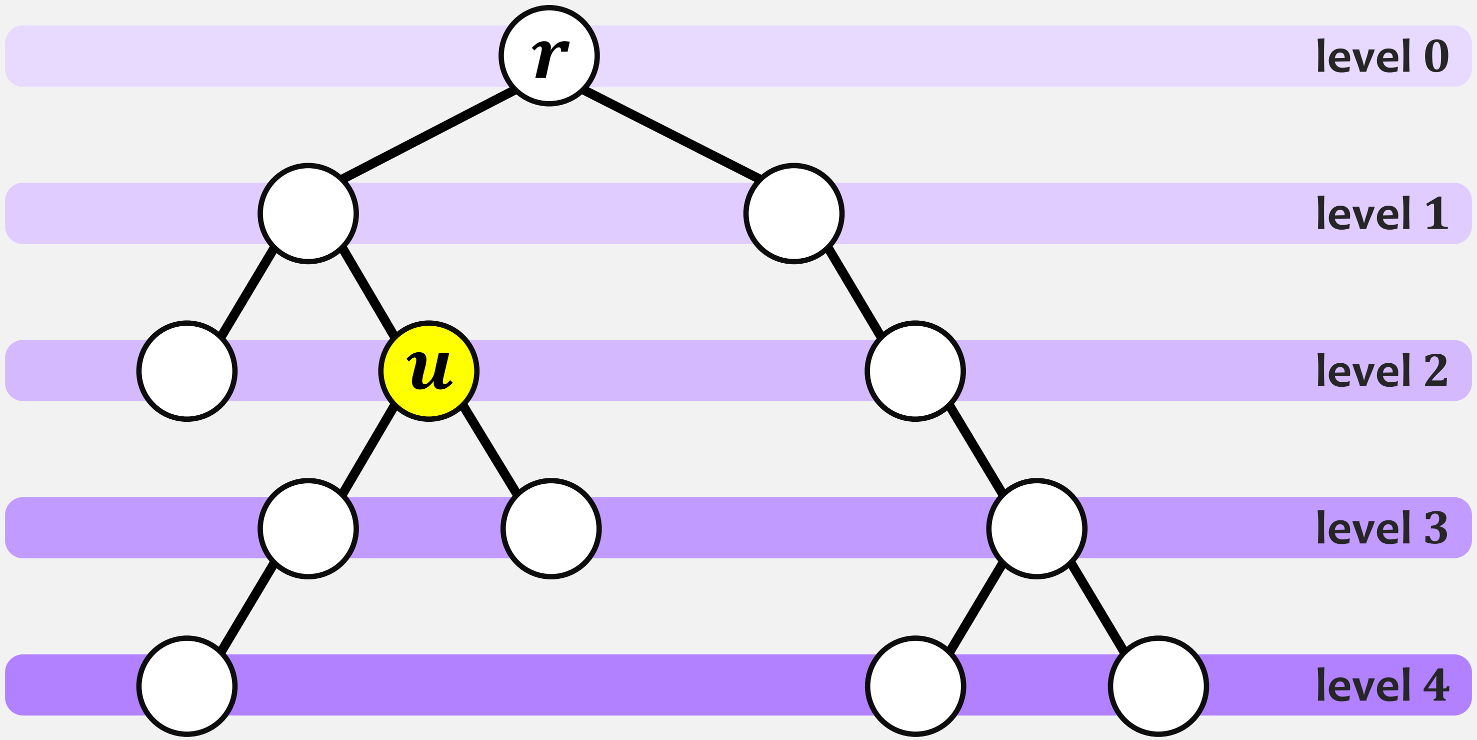
More terminology



- If a node w is on the path from u to r, then w is called an **ancestor** of u and u a **descendant** of w.
- The **subtree** of a node u is the binary tree that is rooted at u and contains all of u's descendants.
- A node u is a **leaf** if it has no children.
- Parent of u, the left child of u, and the right child of u are **neighbours** of u.

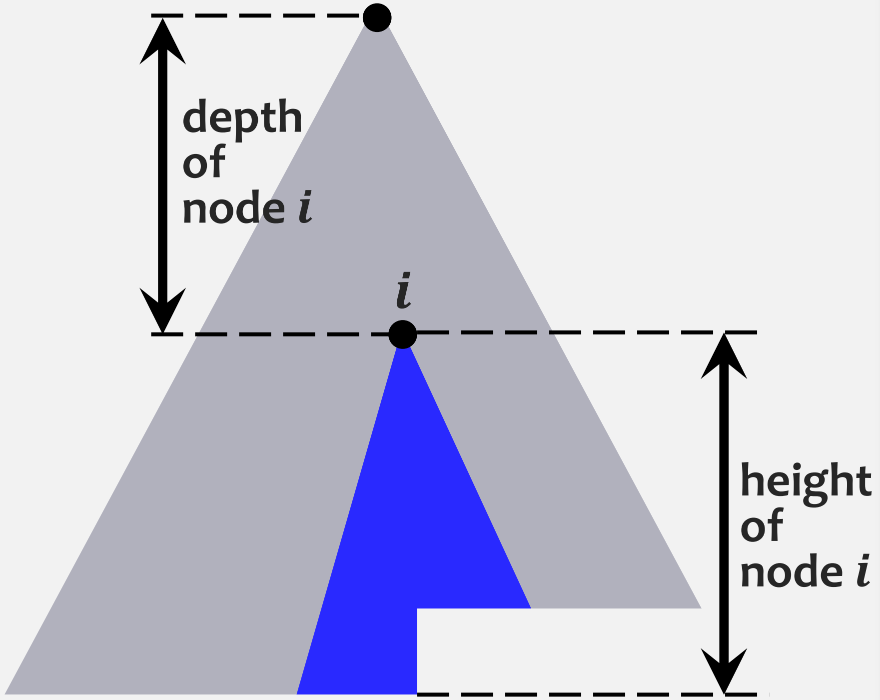
Depth & Height

- The **depth** of a node u in a binary tree is the length of the path (in # of edges) from u to the root of the tree r.
- Level i contains all the nodes of depth i.



the height of this tree is **4**

- The **height** of a node u is the length of the longest path from u to one of its descendants.
- The **height of a tree** is the height of its root.



Depth & Height

We can compute the **depth** of a node $\boldsymbol{u}$ in a binary tree by counting the number of steps on the path from $\boldsymbol{u}$ to the root $\boldsymbol{r}$.

```
int depth(Node u):
    int d = 0;
    while (u ≠ r)
        u = u.parent;
        d++;
    return d;
```

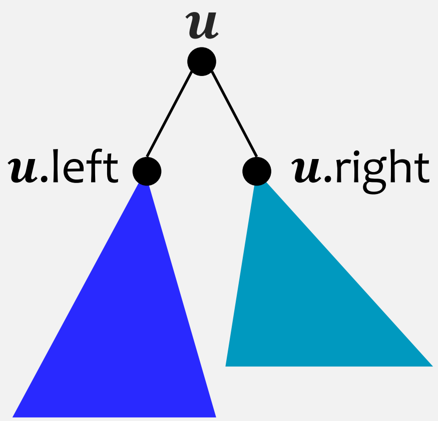
To compute the **height** of a node $\boldsymbol{u}$, we can **recursively** compute the height of $\boldsymbol{u}$'s two subtrees, take the maximum, and add $\mathbf{1}$:

What is the height of this tree?

$\boldsymbol{h} = \mathbf{0}$

```
int height(Node u):
    if (u == null)
        return −1;  // the height of the empty tree
    return 1 + max{height(u.left), height(u.right)};
```

note this function is **recursive**

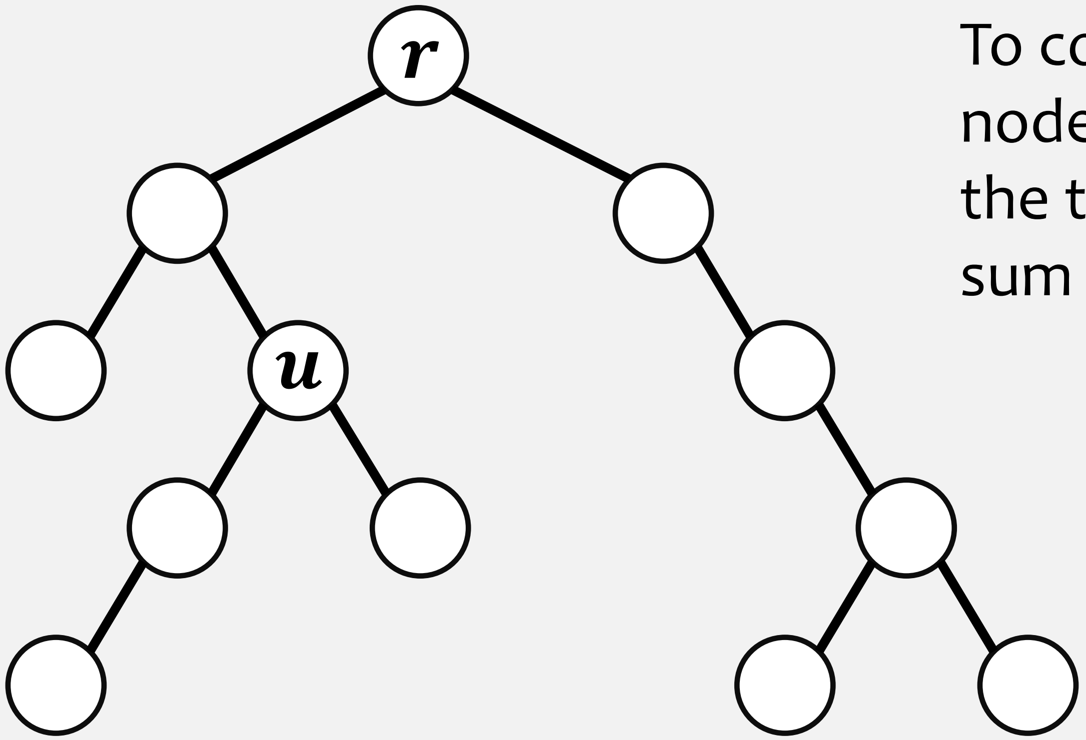


Size

- The **size** of a binary tree is the number of nodes in it.

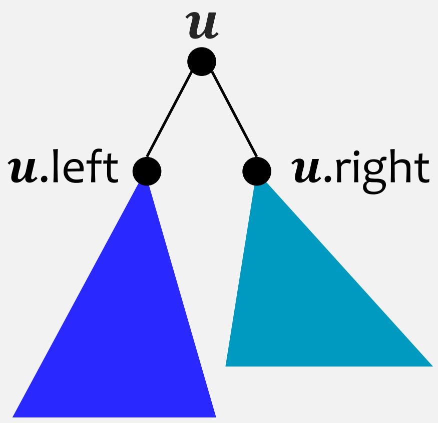
note, this function is recursive

```
int  size(Node u):
      if  (u  ==  null)   // empty tree
          return 0;
      return  1 + size(u.left) + size(u.right);
```



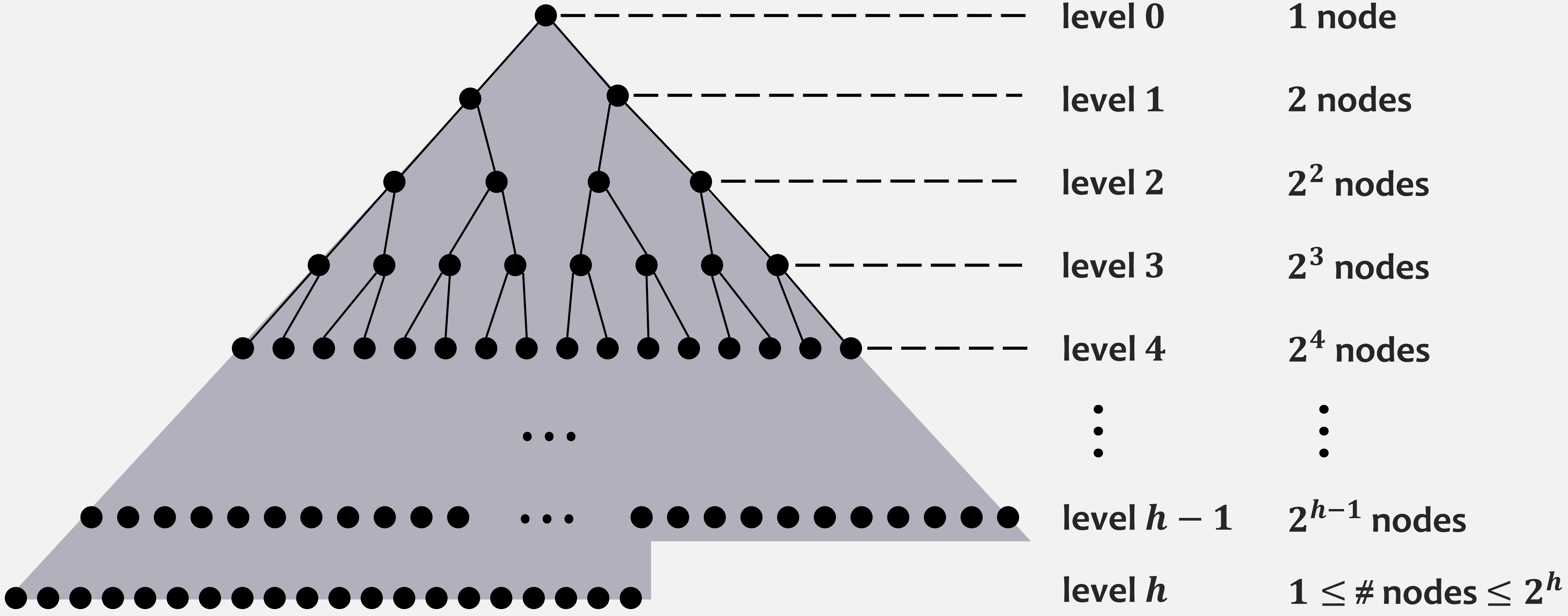
the size of this tree is **12**

To compute the size of a binary tree rooted at node u, we **recursively** compute the sizes of the two subtrees rooted at the children of u, sum up these sizes, and add $\mathbf{1}$.



Complete Binary Tree

A **complete** binary tree is a binary tree in which every level, except possibly the last, is completely full, and all nodes are as far left as possible.



Complete Binary Tree

$$2^0 + 2^1 + 2^2 + \cdots + 2^k = 2^{k+1} - 1$$

What is the **height** (let us call it $\boldsymbol{h}$) of a complete binary tree with $\boldsymbol{n}$ nodes?

$$1 + 2 + 2^2 + \cdots + 2^{h-1} + 1 \le n \le 1 + 2 + 2^2 + \cdots + 2^{h-1} + 2^h$$

$$(2^h - 1) + 1 \le n \le 2^{h+1} - 1$$

$$2^h \le n < 2^{h+1}$$

$$h \le \log n < h + 1$$

$$h \le \log n$$

$$\log n < h + 1$$

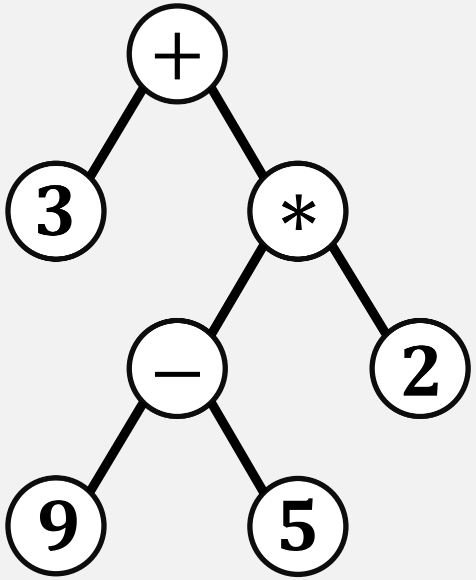
$$-1 + \log n < h$$

$$\log n - 1 < h \le \log n$$

$$h = \lfloor \log n \rfloor$$

Expression Trees

- If node is a leaf, then value is **data** (e.g., number, variable, constant,…)
- If node is internal, then value calculated by applying **operations** on its children



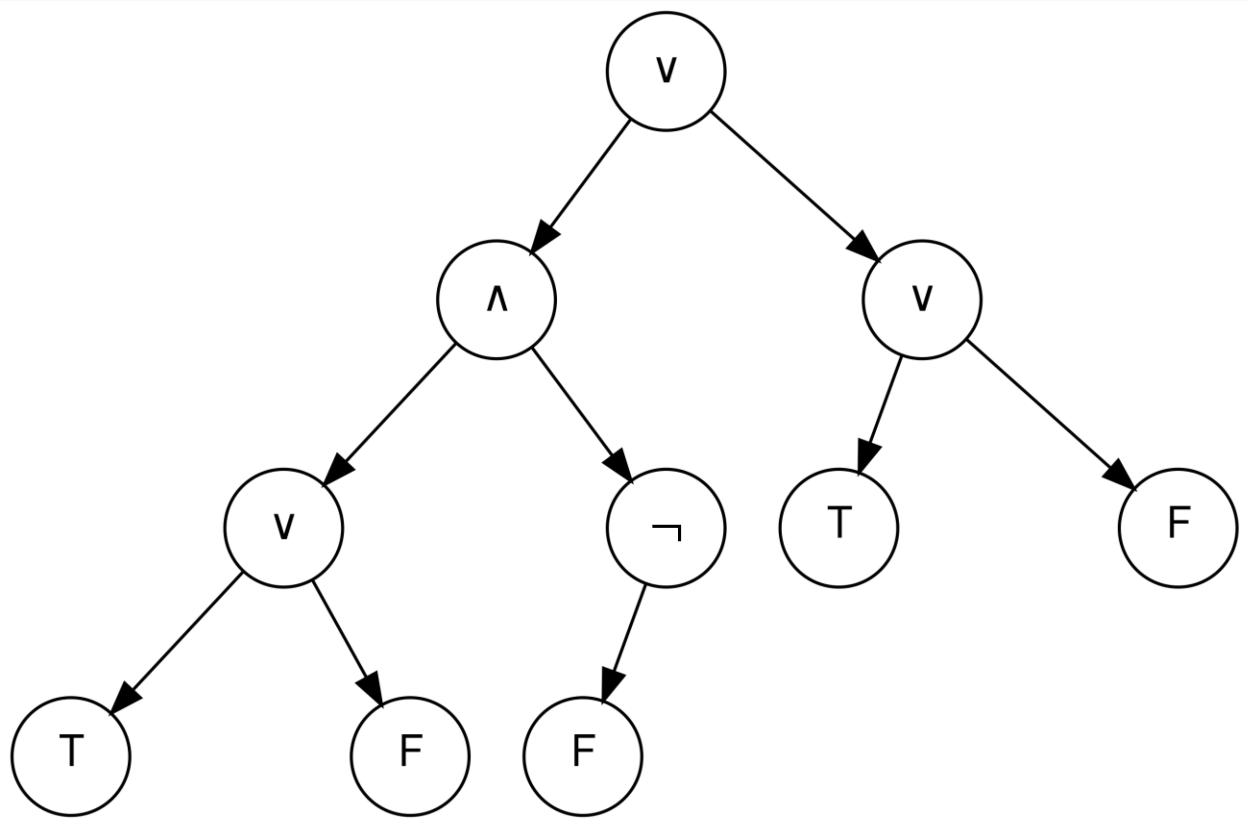
$$\mathbf{3} + ((\mathbf{9} - \mathbf{5}) * \mathbf{2})$$

```
T  evaluate(u):
      if  (u is a leaf)  then
            return  u.value;
      x = evaluate(u.left);
      y = evaluate(u.right);
      return  (x  u.operation  y);
```

note this function is **recursive**

Expression Trees

Boolean expressions



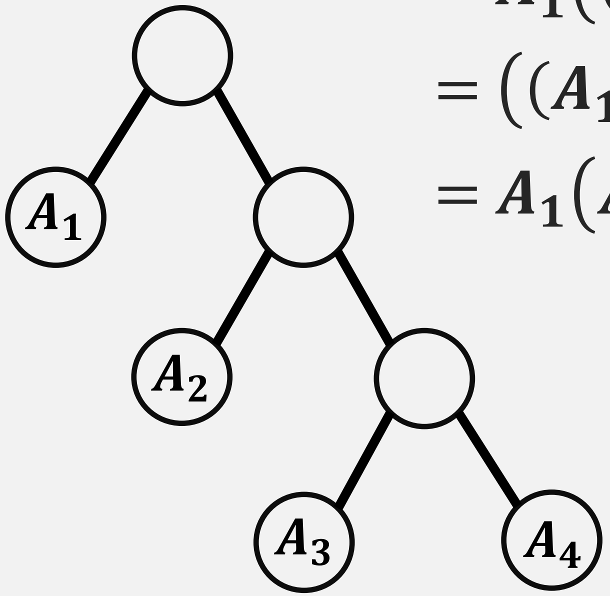
$$\Big(\big((\boldsymbol{T} \lor \boldsymbol{F}) \land \neg \boldsymbol{F}\big) \lor (\boldsymbol{T} \lor \boldsymbol{F})\Big)$$

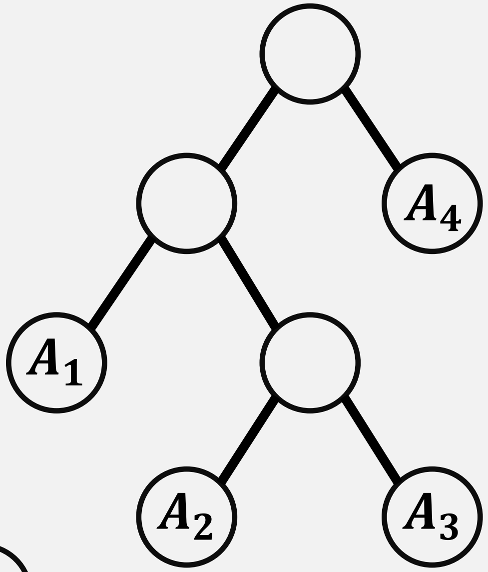
https://en.wikipedia.org/wiki/Binary_expression_tree

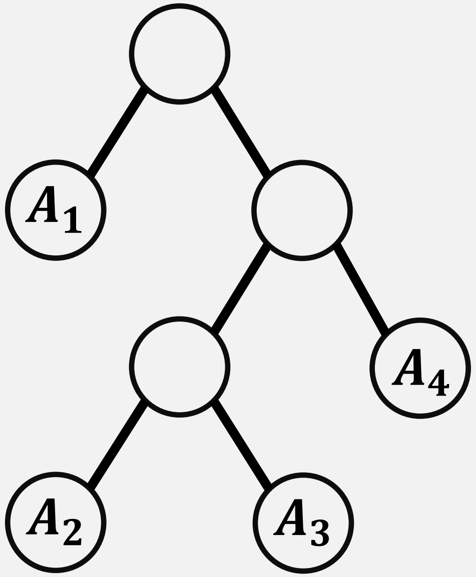
Matrix Chain Multiplication

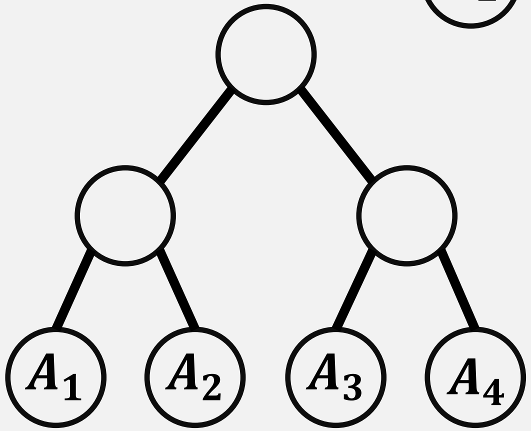
In how many ways can we group the matrices?

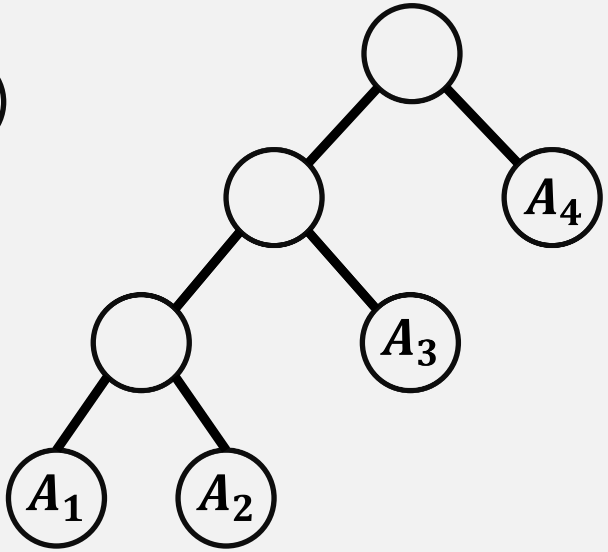
$$
\begin{aligned}
A_1A_2A_3A_4 &= \big(A_1(A_2A_3)\big)A_4 \\
&= (A_1A_2)(A_3A_4) \\
&= A_1\big((A_2A_3)A_4\big) \\
&= \big((A_1A_2)A_3\big)A_4 \\
&= A_1\big(A_2(A_3A_4)\big)
\end{aligned}
$$









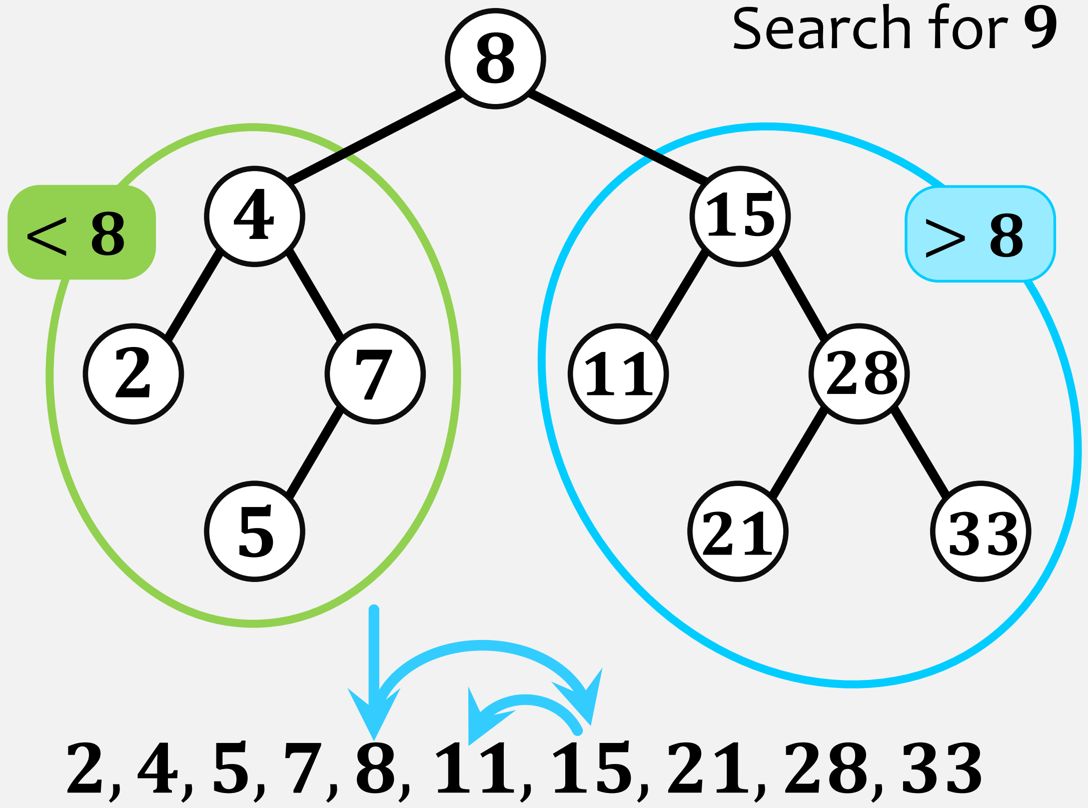


Binary Search Trees

Binary Search Tree property:
For a node $\boldsymbol{u}$ every data value stored in the subtree rooted at $\boldsymbol{u}$.left is less than $\boldsymbol{u.x}$, and every data value stored in the subtree rooted at $\boldsymbol{u}$.right is greater than $\boldsymbol{u.x}$.

Search for **5**

Search for **9**



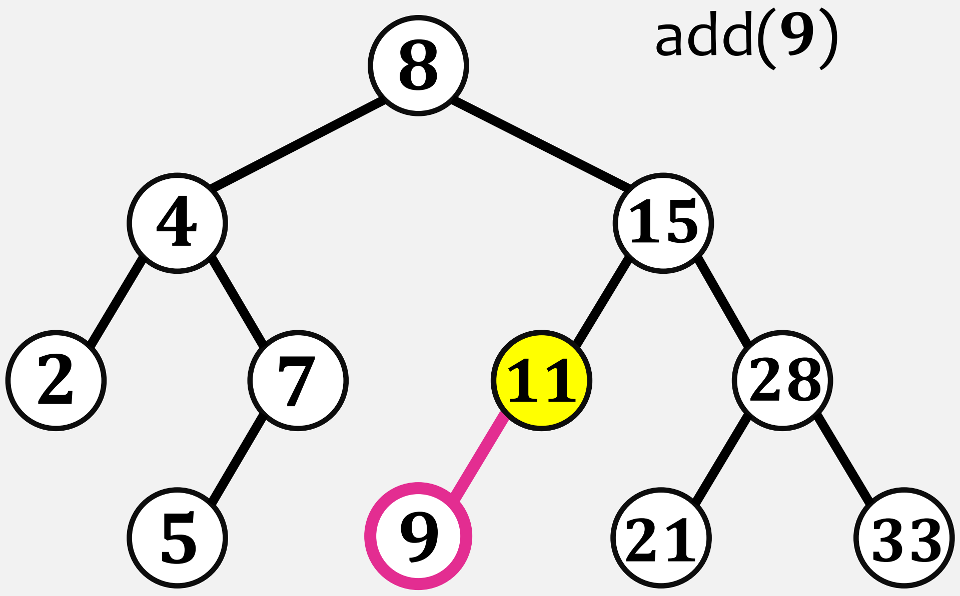
We start searching for $\boldsymbol{x}$ at the root.

If $\boldsymbol{u} =$ null, we conclude that $\boldsymbol{x}$ is not in the binary search tree

When examining a node $\boldsymbol{u}$:

1. If $\boldsymbol{x} < \boldsymbol{u.x}$, then the search proceeds to $\boldsymbol{u}$.left;
2. If $\boldsymbol{x} > \boldsymbol{u.x}$, then the search proceeds to $\boldsymbol{u}$.right;
3. If $\boldsymbol{x} = \boldsymbol{u.x}$, then we have found the node $\boldsymbol{u}$ containing $\boldsymbol{x}$.

Binary Search Trees – add(x)



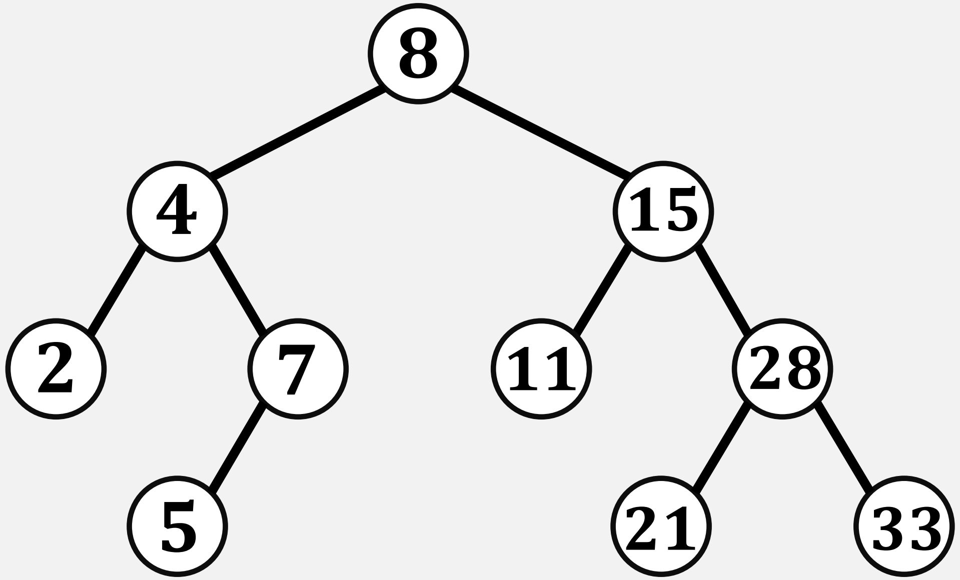
$2, 4, 5, 7, 8, 9, 11, 15, 21, 28, 33$

To add a new value x to a **BinarySearchTree:**

- we first search for x.
- If we find it, then there is no need to insert it.
- Otherwise, we store x at a leaf child of the last node p encountered during the search for x.
- Whether the new node is the left or right child of p depends on the result of comparing x and $p.x$.

Binary Search Trees – remove($\boldsymbol{x}$)

remove($\mathbf{2}$)

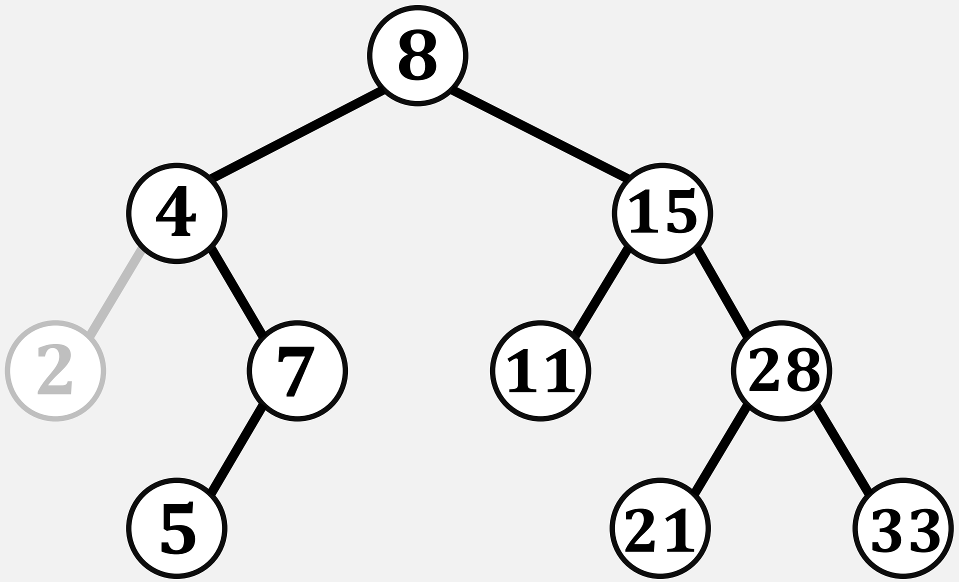


To delete a value stored in **BinarySearchTree:**

- Find node $\boldsymbol{u}$ that contains value $\boldsymbol{x}$.
- If $\boldsymbol{u}$ is a leaf, then we can just detach $\boldsymbol{u}$ from its parent.

Binary Search Trees – remove($\boldsymbol{x}$)

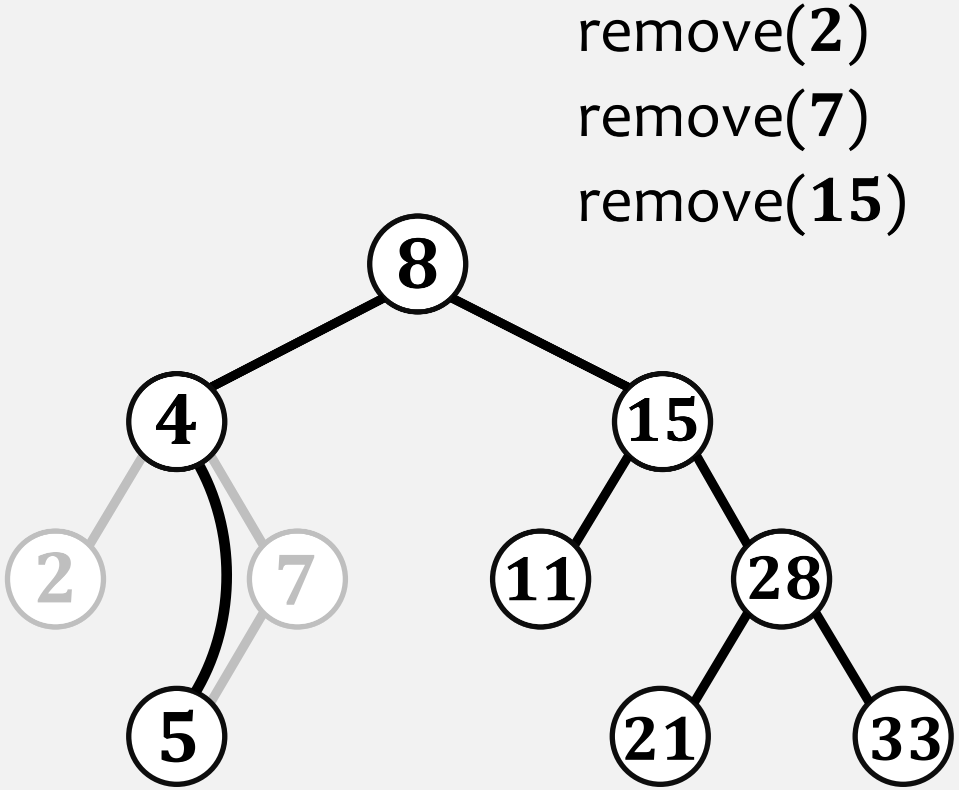
remove($\mathbf{2}$)
remove($\mathbf{7}$)



To delete a value stored in **BinarySearchTree:**

- Find node $\boldsymbol{u}$ that contains value $\boldsymbol{x}$.
- If $\boldsymbol{u}$ is a leaf, then we can just detach $\boldsymbol{u}$ from its parent.
- If $\boldsymbol{u}$ has only one child, then we can splice $\boldsymbol{u}$ from the tree by having $\boldsymbol{u}$.parent adopt $\boldsymbol{u}$'s child.

Binary Search Trees – remove($\boldsymbol{x}$)



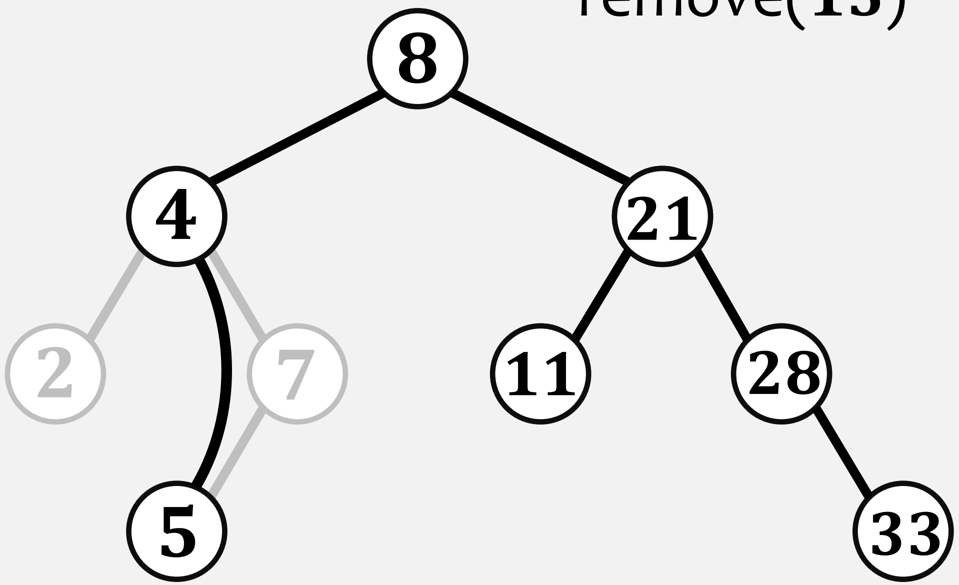
To delete a value stored in **BinarySearchTree:**

- Find node $\boldsymbol{u}$ that contains value $\boldsymbol{x}$.
- If $\boldsymbol{u}$ is a leaf, then we can just detach $\boldsymbol{u}$ from its parent.
- If $\boldsymbol{u}$ has only one child, then we can splice $\boldsymbol{u}$ from the tree by having $\boldsymbol{u}$.parent adopt $\boldsymbol{u}$'s child.
- If $\boldsymbol{u}$ has two children, then find a node $\boldsymbol{w}$, that has less than two children, such that $\boldsymbol{w.x}$ can replace $\boldsymbol{u.x}$.

Choose $\boldsymbol{w}$, such that $\boldsymbol{w.x}$ is the smallest value in the subtree rooted at $\boldsymbol{u}$.right. This node has no left child.

Binary Search Trees – remove($\boldsymbol{x}$)

remove($\mathbf{2}$)
remove($\mathbf{7}$)
remove($\mathbf{15}$)



To delete a value stored in **BinarySearchTree:**

- Find node $\boldsymbol{u}$ that contains value $\boldsymbol{x}$.
- If $\boldsymbol{u}$ is a leaf, then we can just detach $\boldsymbol{u}$ from its parent.
- If $\boldsymbol{u}$ has only one child, then we can splice $\boldsymbol{u}$ from the tree by having $\boldsymbol{u}$.parent adopt $\boldsymbol{u}$'s child.
- If $\boldsymbol{u}$ has two children, then find a node $\boldsymbol{w}$, that has less than two children, such that $\boldsymbol{w.x}$ can replace $\boldsymbol{u.x}$.

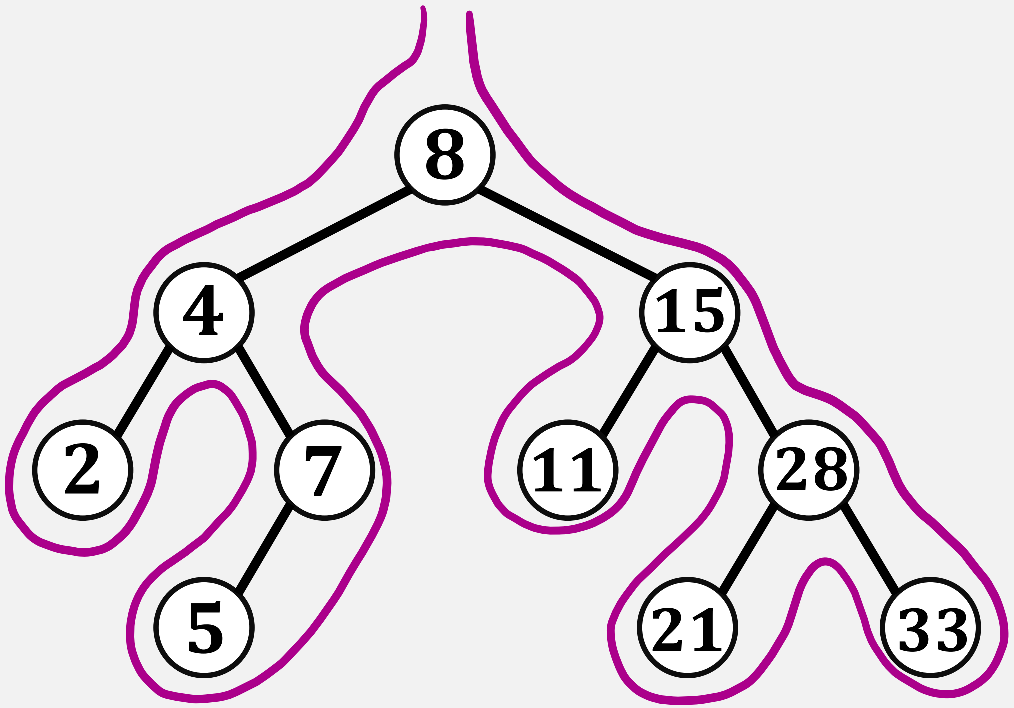
Choose $\boldsymbol{w}$, such that $\boldsymbol{w.x}$ is the smallest value in the subtree rooted at $\boldsymbol{u}$.right. This node has no left child.

Theorem 6.1

BinarySearchTree implements the **SSet** interface and supports the operations add($\boldsymbol{x}$), remove($\boldsymbol{x}$), and find($\boldsymbol{x}$) in $\boldsymbol{O}(\boldsymbol{n})$ time per operation.

Traversals

To **traverse** (or **walk**) the binary tree is to print each node in the binary tree exactly once



$$\mathbf{2, 4, 5, 7, 8, 11, 15, 21, 28, 33}$$

In-Order Traversal

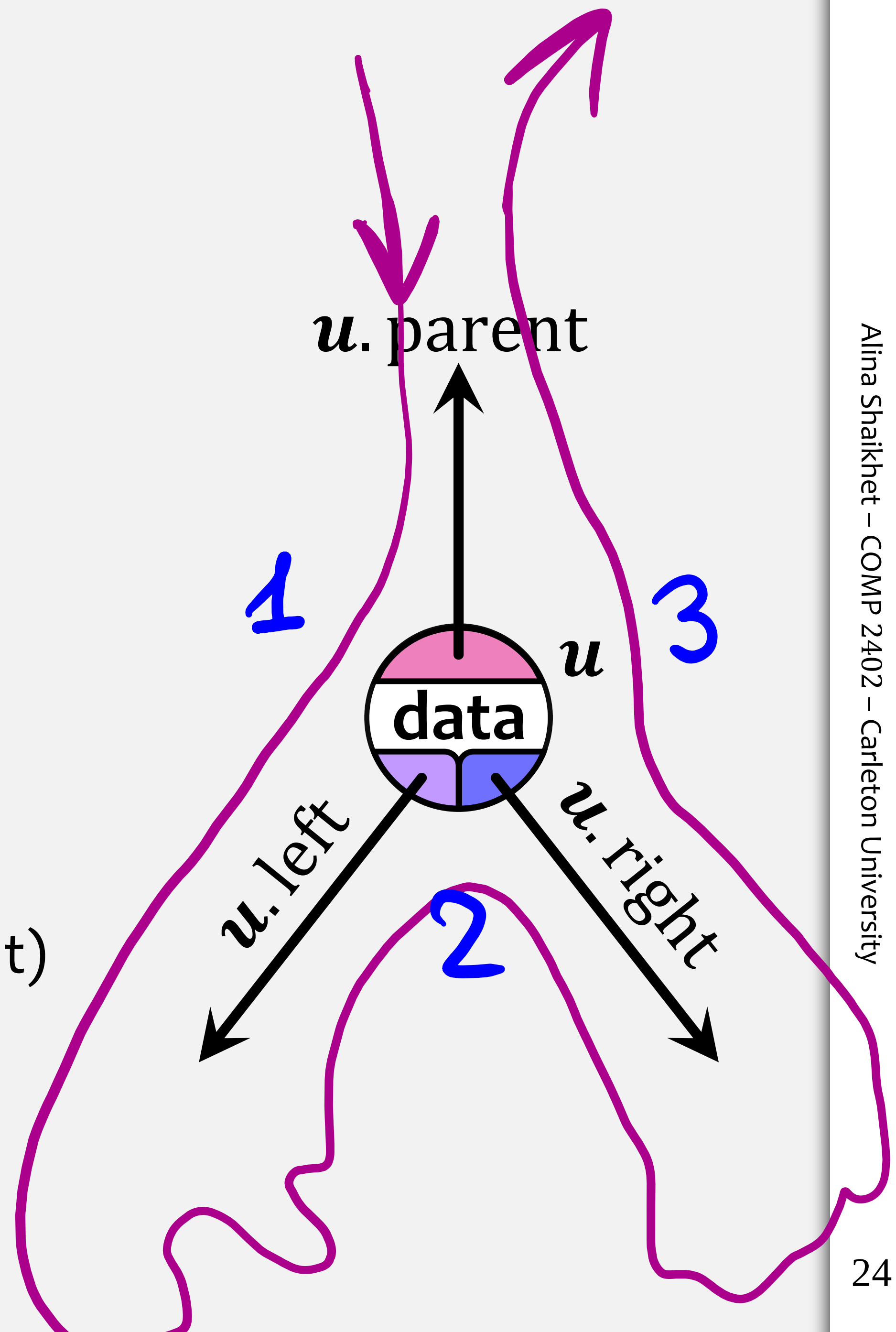
```
void  print-bst(u):
    if  (u = null)  then
        return;
    print_bst(u.left);
    print(u.x);
    print_bst(u.right);
```

note this function is **recursive**

Traversals – non-recursive

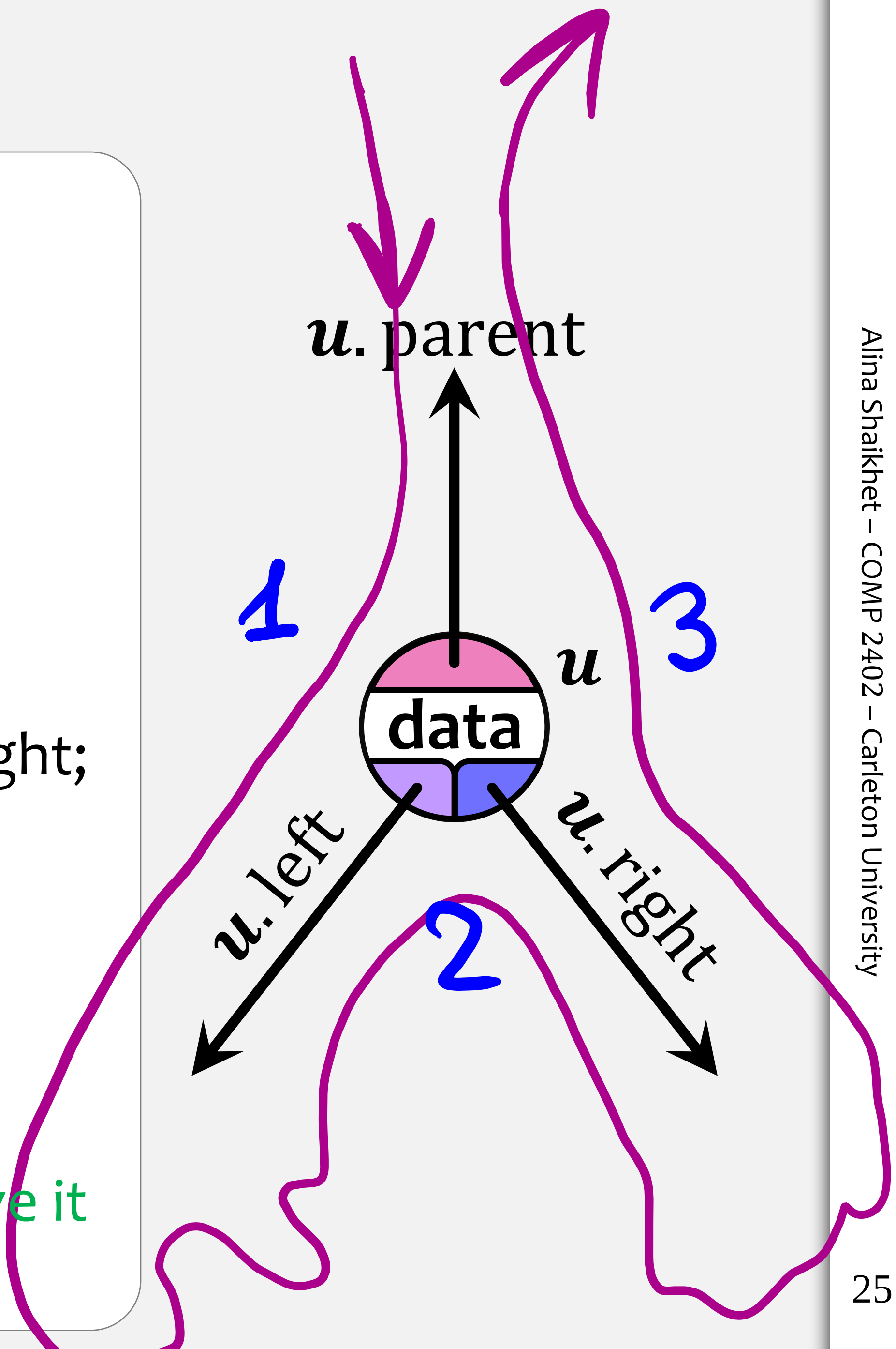
We see each node three times:

1. when we first visit it from the parent
   (so we need to go left)
2. when we come back from its left child
   (so we need to go right)
3. when we come back from its right child
   (so we need to go back to parent)



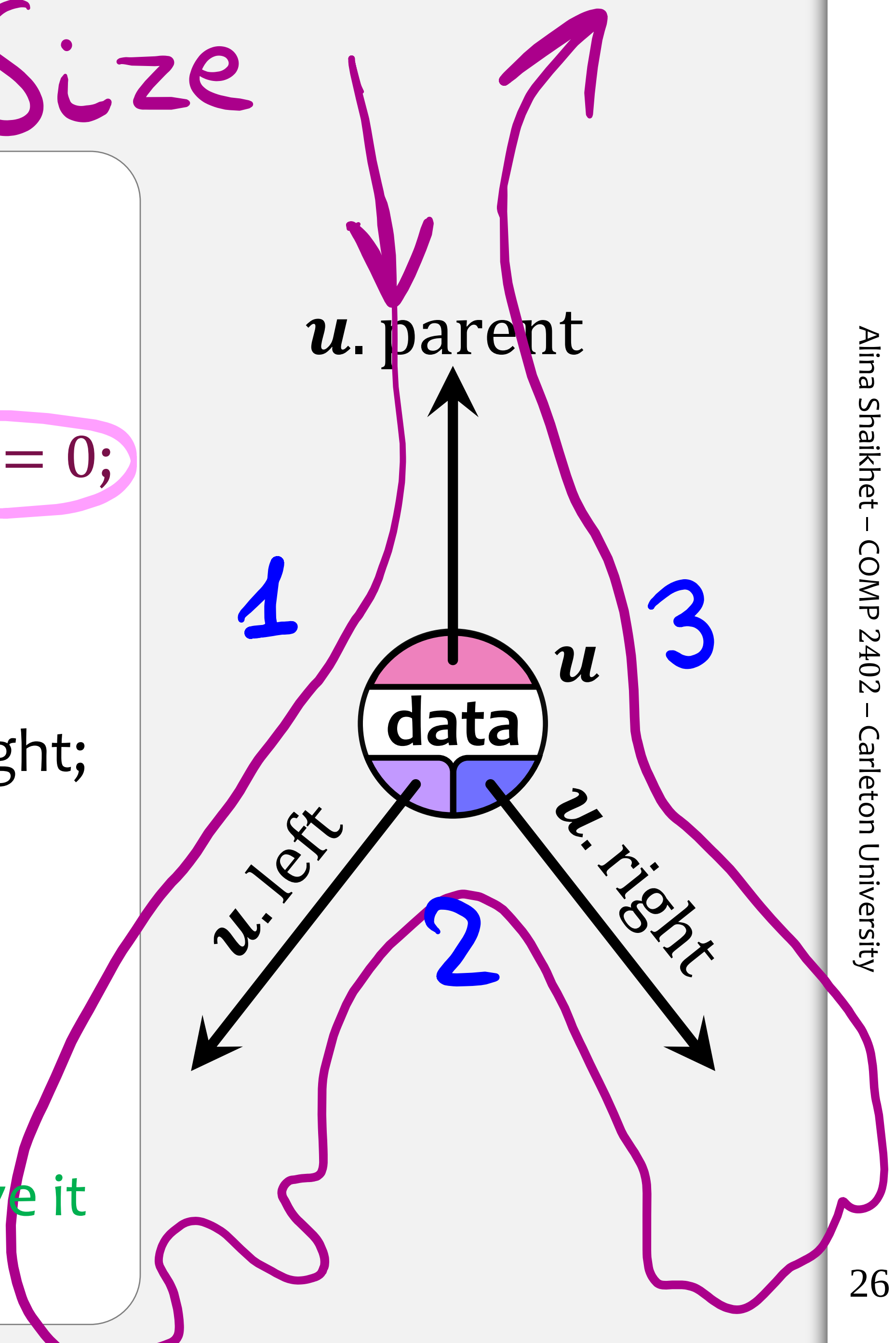
Traversals – non-recursive

```
void  traverse():
      Node u = root;
      Node prev = null;  // previous node we were at
      Node next;
      while  (u ≠ null)
         if  (prev == u.parent )  then
              if  (u.left ≠ null)   then  next = u.left;
              else  if  (u.right ≠ null)  then  next = u.right;
              else  next = u.parent;
         else if  (prev == u.left )  then
              if  (u.right ≠ null)  then  next = u.right;
              else  next = u.parent;
         else  next = u.parent;
         prev = u;  // we are about to leave u, so we save it
         u = next;
```



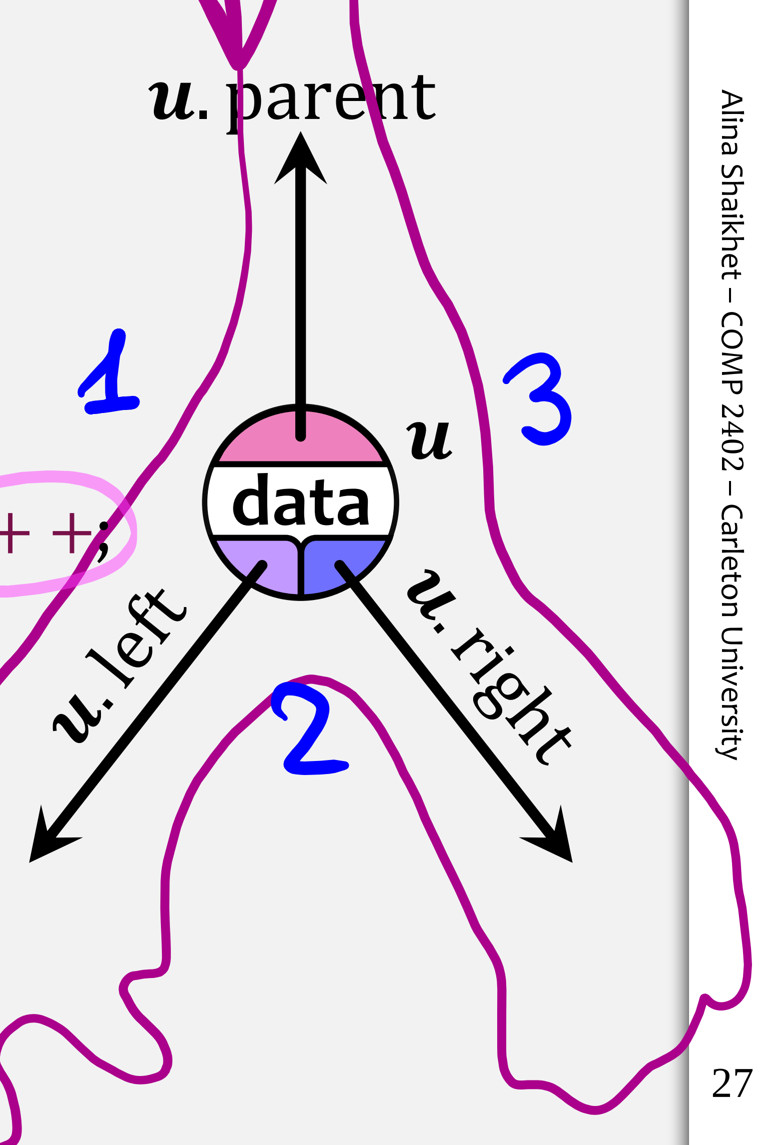
Traversals – non-recursive Size

```
void traverse():
    Node u = root;
    Node prev = null; // previous node we were at
    Node next; int size = 1; if root == null then size = 0;
    while (u ≠ null)
        if (prev == u.parent) then size++;
            if (u.left ≠ null) then next = u.left;
            else if (u.right ≠ null) then next = u.right;
            else next = u.parent;
        else if (prev == u.left) then
            if (u.right ≠ null) then next = u.right;
            else next = u.parent;
        else next = u.parent;
        prev = u; // we are about to leave u, so we save it
        u = next;
```



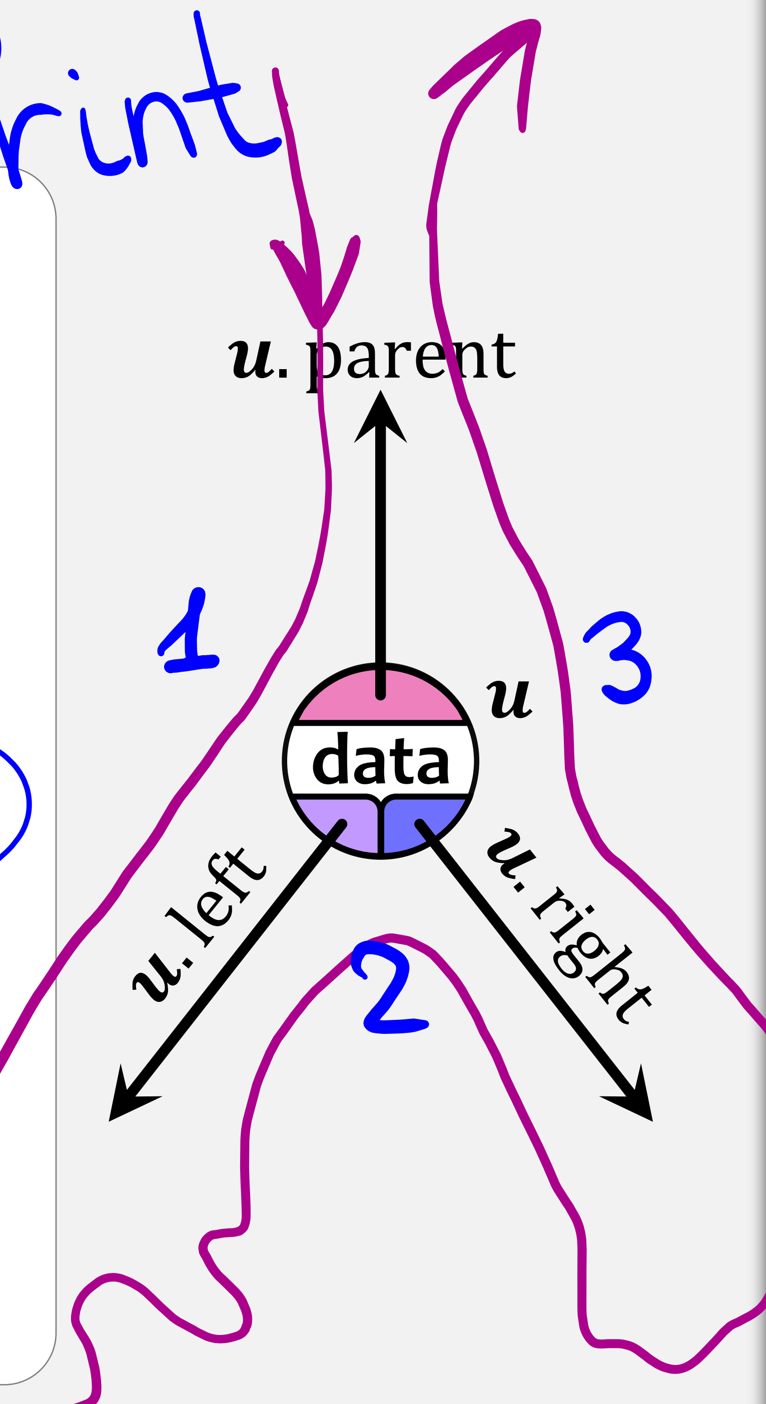
Traversals – non-recursive Height

```
void traverse():
    Node u = root;
    Node prev = null;   int height = 0;
    Node next;   int d = 0; // keep track of the depth of u
    while (u ≠ null)
        if (prev == u.parent ) then
            if (u.left ≠ null) then next = u.left;  d++;
            else if (u.right ≠ null) then next = u.right; d++;
            else next = u.parent; d--;
        else if (prev == u.left ) then
            if (u.right ≠ null) then next = u.right; d++;
            else next = u.parent; d--;
        else next = u.parent; d--;
        prev = u; // we are about to leave u, so we save it
        u = next;  height=max{height, d}
```



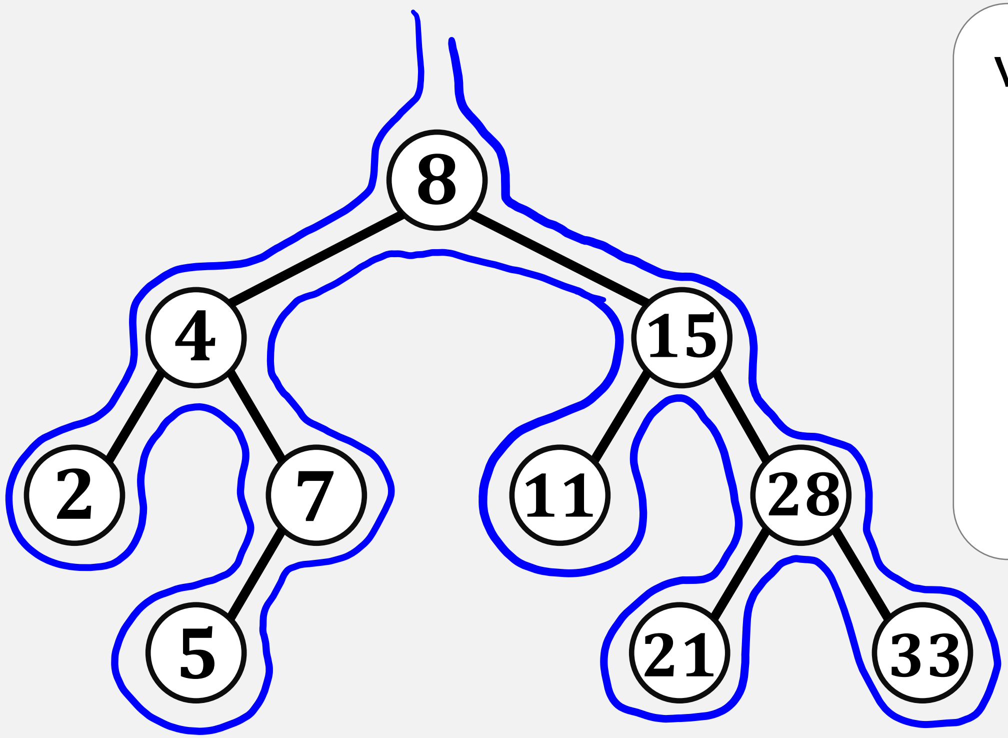
Traversals – non-recursive

```
void  traverse():
      Node u = root;
      Node prev = null;  // previous node we were at
      Node next;
      while  (u ≠ null)
          if  (prev == u.parent )  then
              if  (u.left ≠ null)   then  next = u.left;
              else  if  (u.right ≠ null)  then  next = u.right;
you are a leaf → else  next = u.parent;  Print(u.x);  Print(u.x);
          else if  (prev == u.left )  then  print(u.x);
              if  (u.right ≠ null)  then  next = u.right;
              else  next = u.parent;
          else  next = u.parent;
          prev = u;  // we are about to leave u, so we save it
          u = next;
```



Traversals

Pre-Order:	root (action), left subtree, right subtree	$8, 4, 2, 7, 5, 15, 11, 28, 21, 33$
In-Order:	left subtree, root (action), right subtree	$2, 4, 5, 7, 8, 11, 15, 21, 28, 33$
Post-Order:	left subtree, right subtree, root (action)	$2, 5, 7, 4, 11, 21, 33, 28, 15, 8$



Post-Order

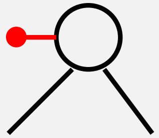
```
void print-bst(u):
    if (u = null) then
        return;
    print_bst(u.left);
    print_bst(u.right);
    print(u.x);
```

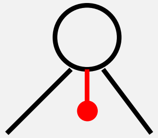
In-Order

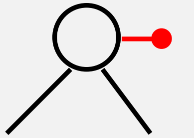
```
void print-bst(u):
    if (u = null) then
        return;
    print_bst(u.left);
    print(u.x);
    print_bst(u.right);
```

Tree traversals using “flags”

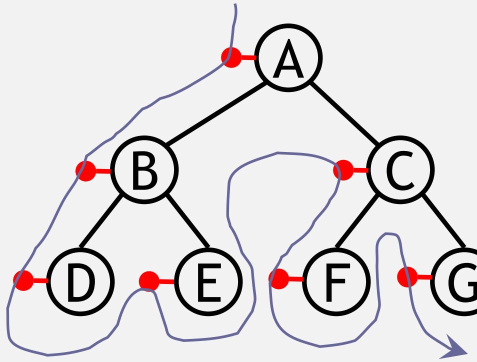
The order in which the nodes are visited during a tree traversal can be easily determined by imagining there is a “flag” attached to each node, as follows:


preorder

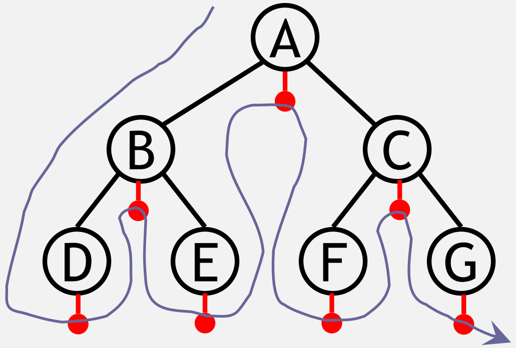

inorder


postorder

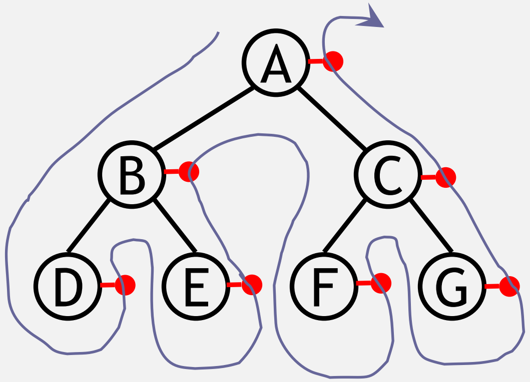
To traverse the tree, collect the flags:



A B D E C F G

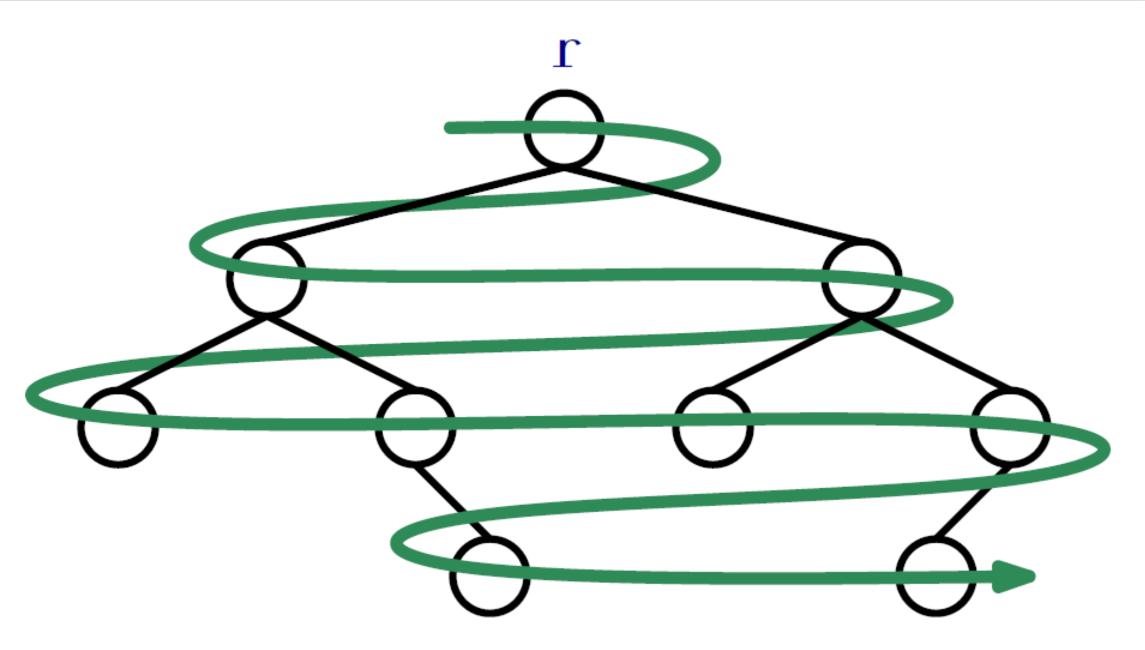


D B E A F C G



D E B F G C A

BFS traversal



from ODS book

void **bfTraverse**():

    Queue<Node> $\boldsymbol{q}$ = new LinkedList<Node>();
    if ($\boldsymbol{r} \neq$ null) then
        $\boldsymbol{q}$.add($\boldsymbol{r}$);
    while ($\boldsymbol{q}$ is not empty)
        Node $\boldsymbol{u} = \boldsymbol{q}$.remove();
        if ($\boldsymbol{u}$.left $\neq$ null) then $\boldsymbol{q}$.add($\boldsymbol{u}$.left);
        if ($\boldsymbol{u}$.right $\neq$ null) then $\boldsymbol{q}$.add($\boldsymbol{u}$.right);