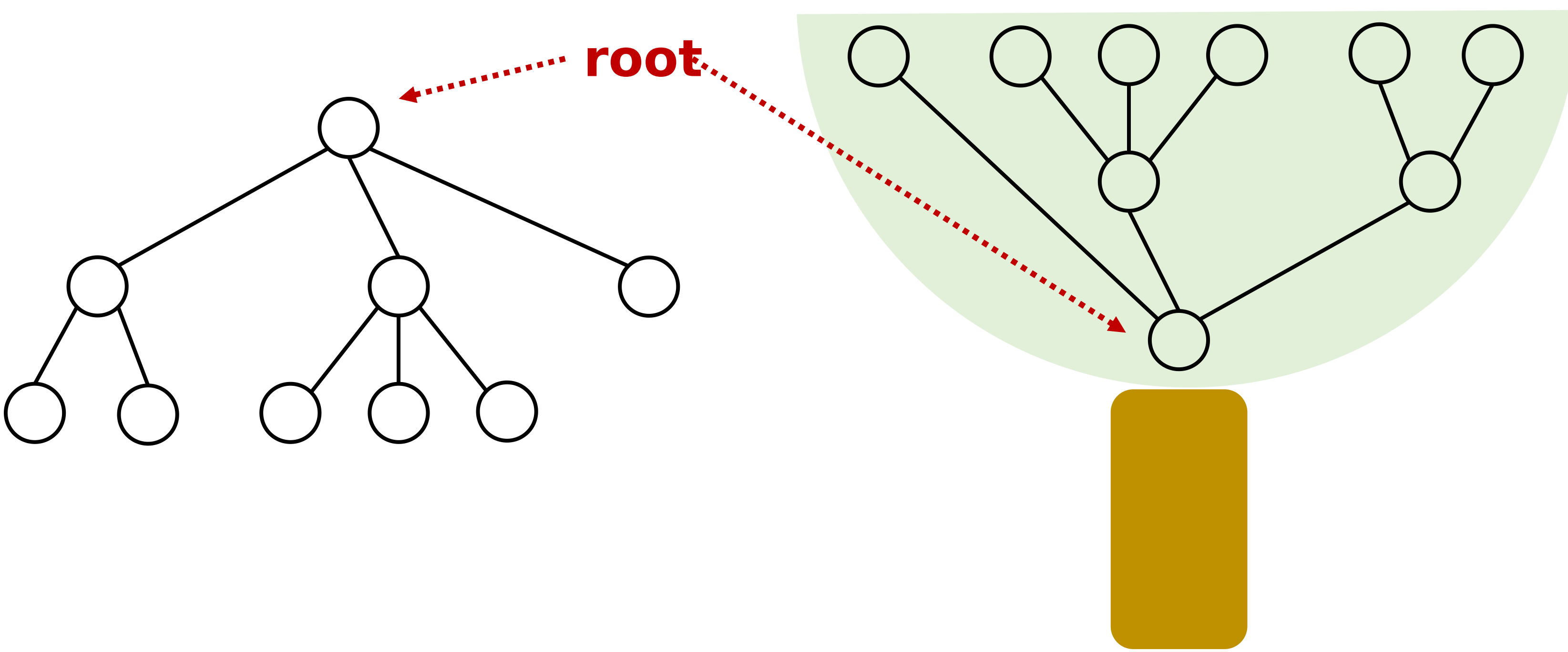
Trees

Data Structures and Algorithms

Nanjing University, Fall 2021
郑朝栋

Trees

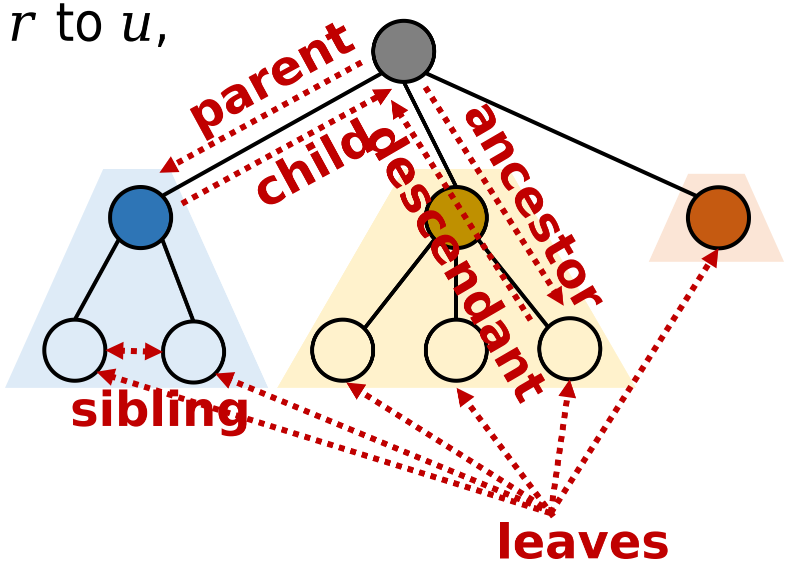
- A tree is a connected, acyclic undirected graph.
- In CS, we often study **rooted** trees.



Recursive definition of

Trees

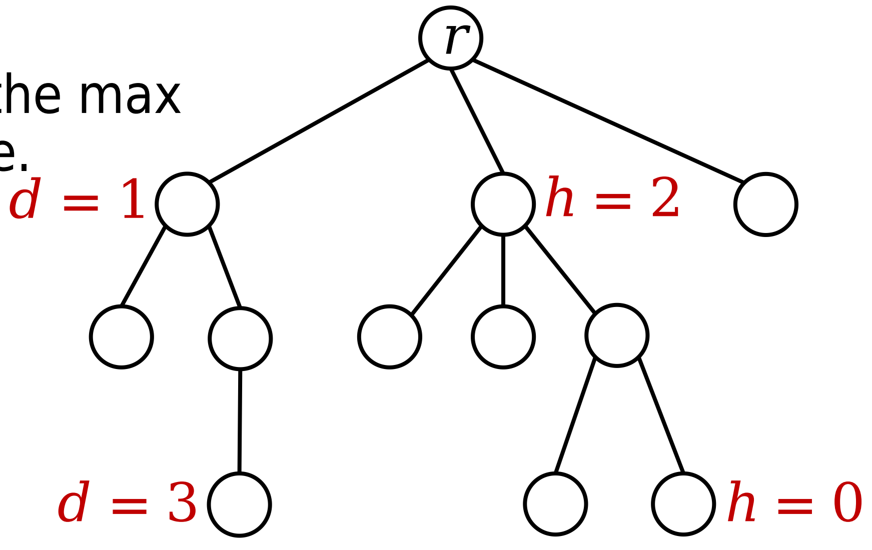
- A tree is either empty, or has a root r that connects to the roots of zero or more non-empty (sub)trees.
  - Root of each subtree is a **child** of r, and r is the **parent** of each subtree's root.
  - Nodes with no children are **leaves**.
  - Nodes with same parent are **siblings**.
  - If a node v is on the path from r to u, then v is an **ancestor** of u, and u is a **descendant** of v.



More terminology on

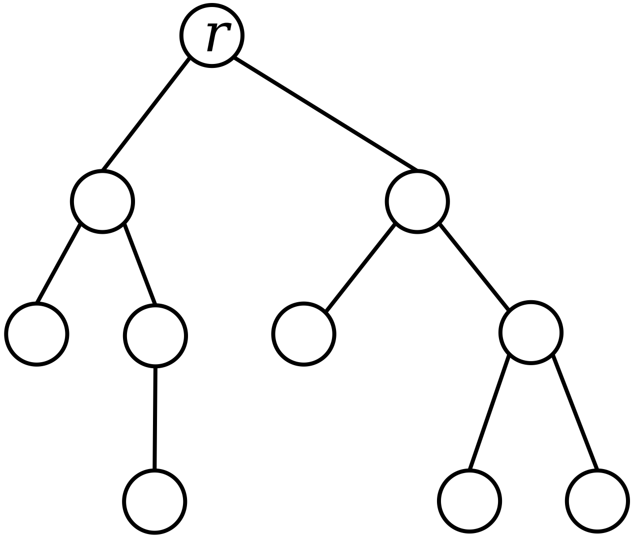
Trees

- The **depth** of a node u is the length of the path from u to r.
- The **height** of a node u is the length of the longest path from u to one of its descendants.
  - Height of a leaf node is zero.
  - Height of a non-leaf node is the max height of its children plus one.

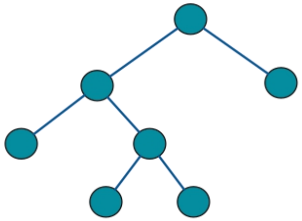


Binary Trees

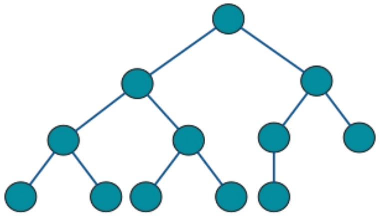
- A **binary tree** is a tree in which each node has at most two children.
  - Often call these children as **left child** and **right child**.



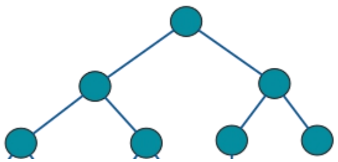
More terminology on
Binary Trees



- A **full binary tree** is a binary tree where each node has either zero or two children.
  - A full binary tree is either a single node, or a tree in which the two subtrees of the root are full binary trees.



- A **complete binary tree** is a binary tree where every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.
  - A complete binary tree can be efficiently represented using an array.

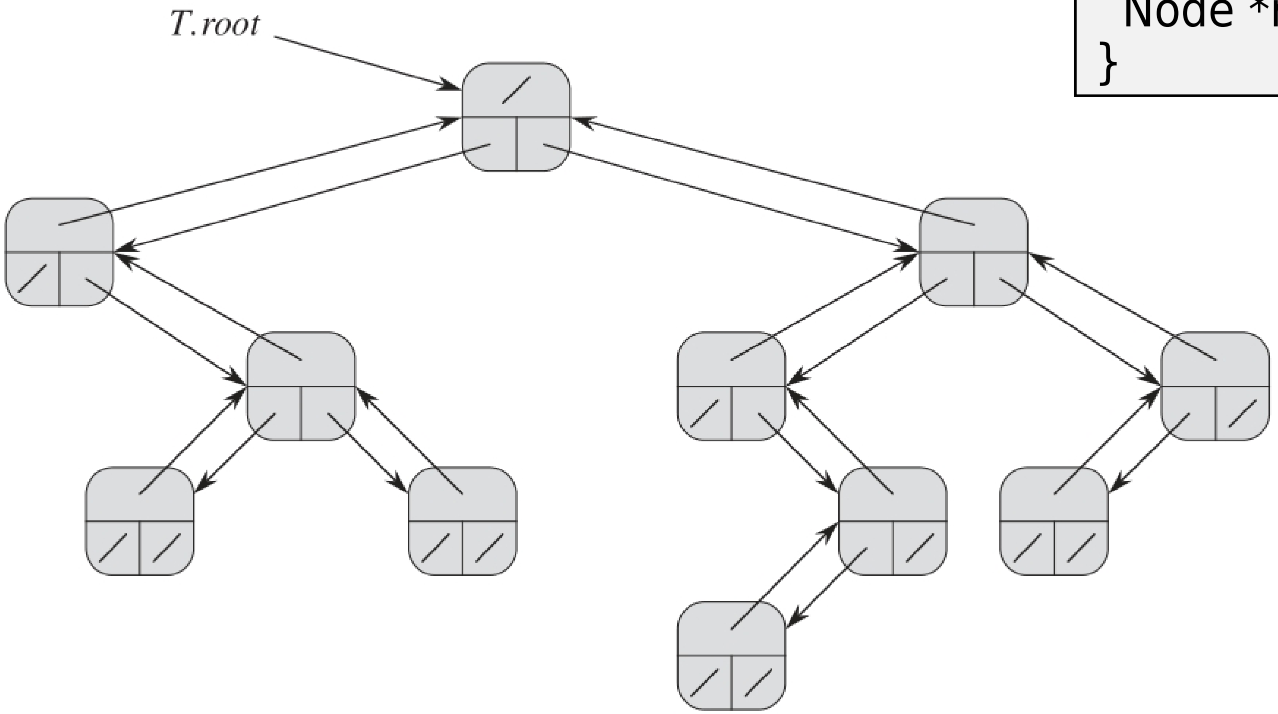


- A **perfect binary tree** is a binary tree where all non-leaf nodes have two children and all leaves have same depth.
  - CLRS call perfect binary trees as complete binary trees

Representing Binary Trees

What if nodes have more children?

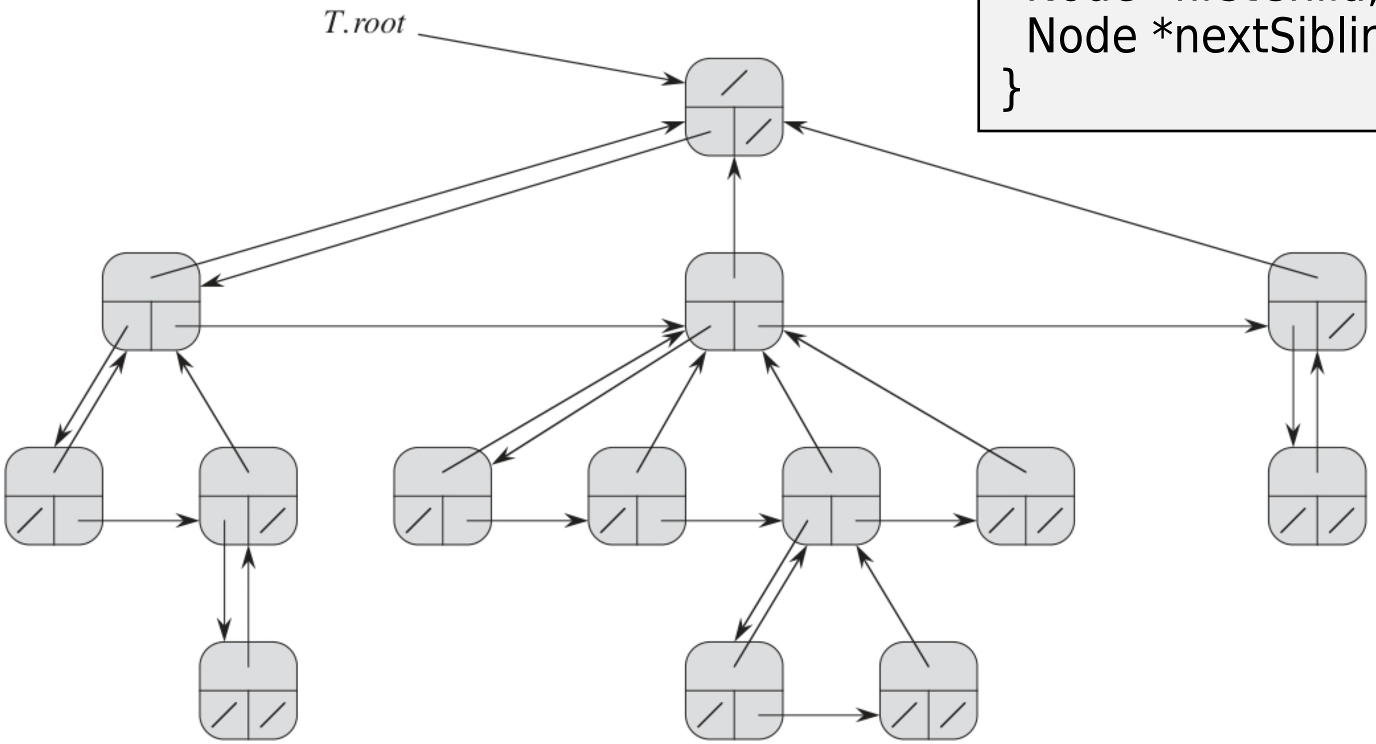
```
struct Node {
  Data data;
  Node *parent;
  Node *left;
  Node *right;
}
```



Representing General Trees

"Left-child, right-sibling representation."

```
struct Node {
  Data data;
  Node *parent;
  Node *firstChild;
  Node *nextSibling;
}
```



Tree Traversals

PreorderTrav(r):

```
if (r != NULL)
  Visit(r)
  for (each child u of r)
    PreorderTrav(u)
```

t all
finiti
ing to
empty subtrees.

PostorderTrav(r):

```
if (r != NULL)
  for (each child u of r)
    PostorderTrav(u)
  Visit(r)
```

- Many ways to traverse a tree (recursively):
  - **Preorder traversal:** given a tree with root r, first visit r, then visit subtrees rooted at r's children, using preorder traversal.
  - **Postorder traversal:** given a tree with root r, first visit subtrees rooted at r's children using postorder traversal, then visit r.
  - **Inorder tr** with root r, first visit subtree finally visit subtree root

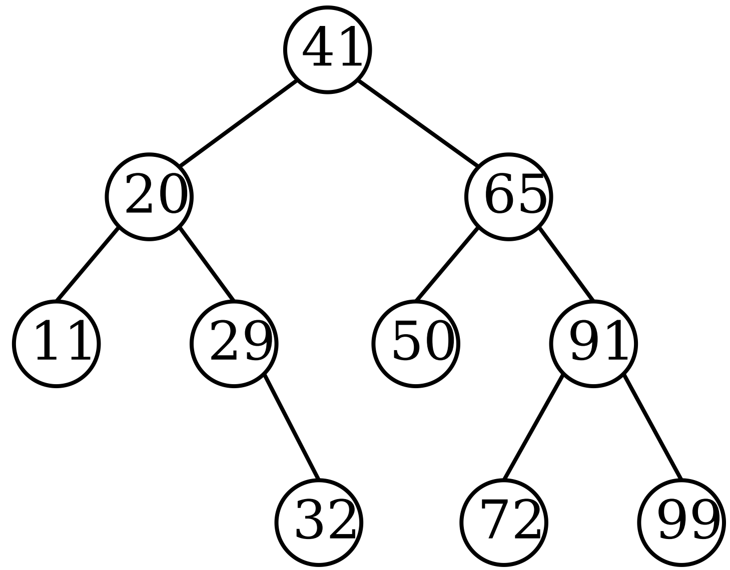
InorderTrav(r):

```
if (r != NULL)
  InorderTrav(r.left)
  Visit(r)
  InorderTrav(r.right)
```

Preorder Traversal

PreorderTrav(r):

```
if (r != NULL)
  Visit(r)
  for (each child u of r)
    PreorderTrav(u)
```

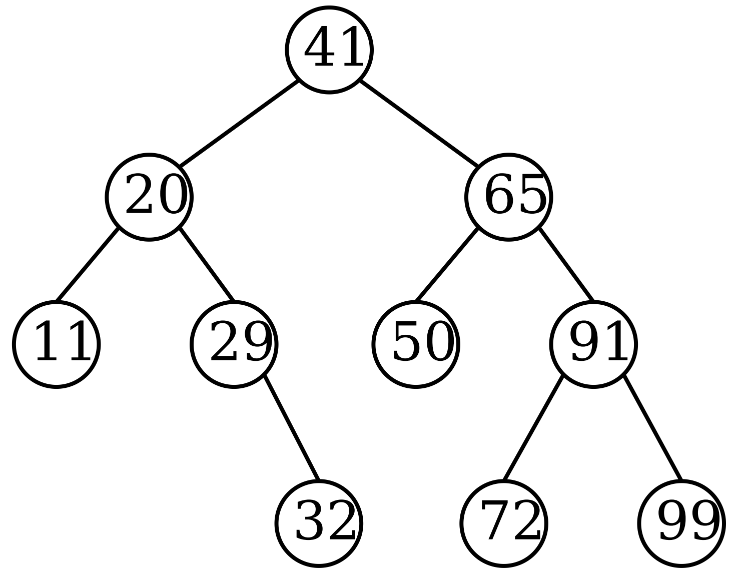


41 20 11 29 32 65 50 91 72 99

Postorder Traversal

PostorderTrav(r):

```
if (r != NULL)
  for (each child u of r)
    PostorderTrav(u)
  Visit(r)
```

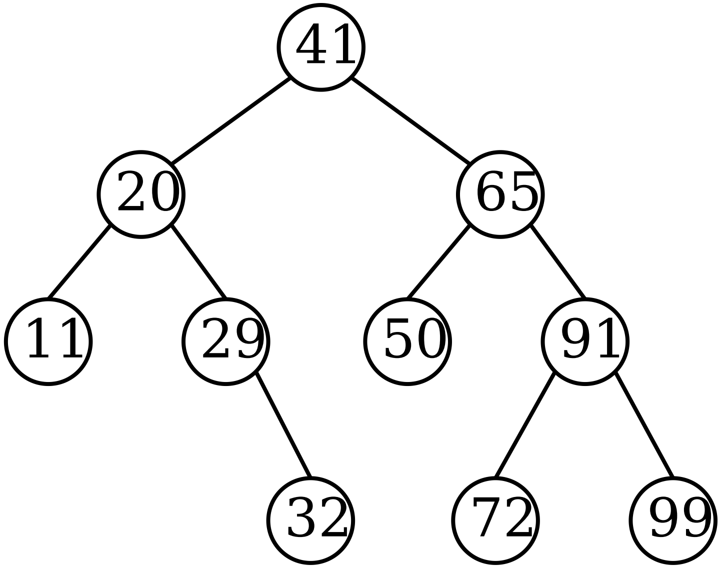


11 32 29 20 50 72 99 91 65 41

Inorder Traversal

InorderTrav(r):

```
if (r != NULL)
  InorderTrav(r.left)
  Visit(r)
  InorderTrav(r.right)
```



11 20 29 32 41 50 65 72 91 99

Complexity of recursive traversal

PreorderTrav(r):

```
if (r != NULL)
  Visit(r)
  for (each child u of r)
    PreorderTrav(u)
```

PostorderTrav(r):

```
if (r != NULL)
  for (each child u of r)
    PostorderTrav(u)
  Visit(r)
```

InorderTrav(r):

```
if (r != NULL)
  InorderTrav(r.left)
  Visit(r)
  InorderTrav(r.right)
```

Time complexity for a size n tre

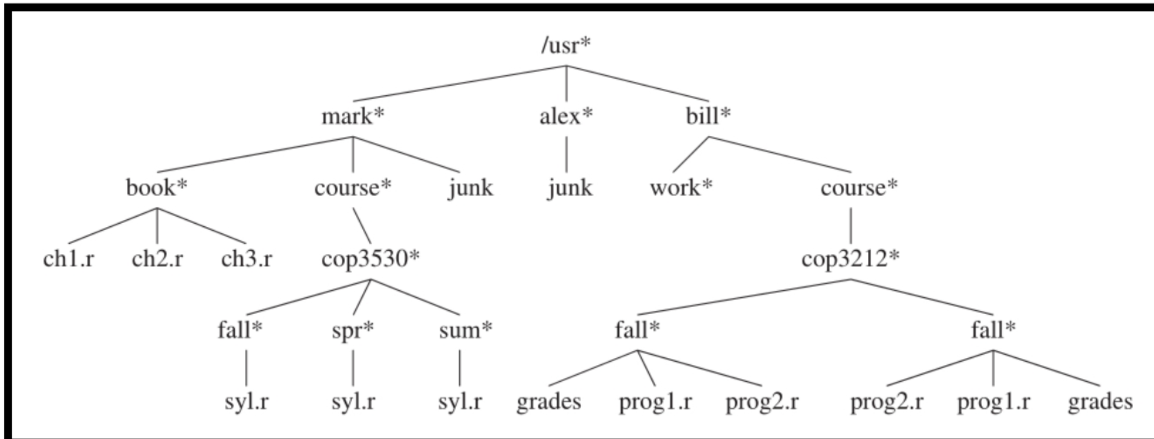
$\Theta(n)$ as processing each node takes $\Theta($

Space complexity for a size n tr

$O(n)$ as worst-case call stack depth is Θ

Sample application of preorder traversal

Directory Listing



PreorderTrav(r):

if (
V
fo

ListDir(obj, depth):

```
if (obj != NULL)
  PrintName(obj,depth)
  if (IsDirectory(obj))
    for (each obj in directory)
      ListDir(obj,depth+1)
```

```
/usr
    mark
        book
            ch1.r
            ch2.r
            ch3.r
        course
            cop3530
                fall
                    syl.r
                spr
                    syl.r
                sum
                    syl.r
        junk
    alex
        junk
    bill
        work
        course
            cop3212
                fall
                    grades
                    prog1.r
                    prog2.r
                fall
                    prog2.r
                    prog1.r
                    grades
```

Iterative tree traversal

Basic idea: simulate the recursive process with the help of a stack.

PreorderTrav(r):

```
if (r != NULL)
  Visit(r)
  for (each child u of r)
    PreorderTrav(u)
```

```
struct Frame {
  Node *node;
  bool visit;
  Frame(Node* n,bool v) {
    node = n;
    visit = v;
  }
}
```

PreorderTravIter(root):

```
Stack s
s.push(Frame(root,false))
while (!s.empty())
  f = s.pop()
  if (f.node != NULL)
    if (f.visit)
      Visit(f.node)
    else
      for (each child u of f.node)
        s.push(Frame(u,false))
      s.push(Frame(f.node,true))
```

What about postorder traversal?

Visit node or the subtree rooted at node.

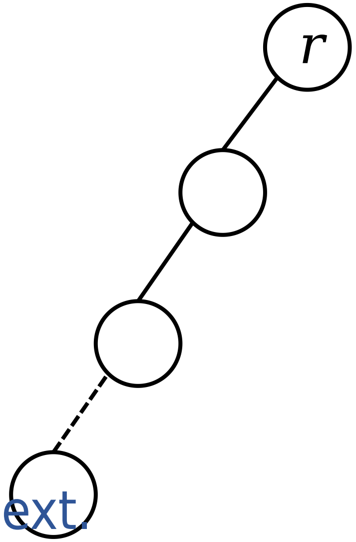
What about inorder traversal?

Iterative inorder tree traversal

<u>InorderTravIter(root):</u>

```
Stack s
s.push(Frame(root,false))
while (!s.empty())
  f = s.pop()
  if (f.node != NULL)
    if (f.visit)
      Visit(f.node)
    else
      s.push(Frame(f.node->right,false))
      s.push(Frame(f.node,true))
      s.push(Frame(f.node->left,false))
```

```
struct Frame {
  Node *node;
  bool visit;
  Frame(Node* n,bool v) {
    node = n;
    visit = v;
  }
}
```



What is the time complexity? $\Theta(n)$

What is the space complexity? $\mathcal{O}(n)$

When do we need $\Theta(n)$ space?

Can we have better space complexity?

Yes! Knowing last visited node tells us what to do next.

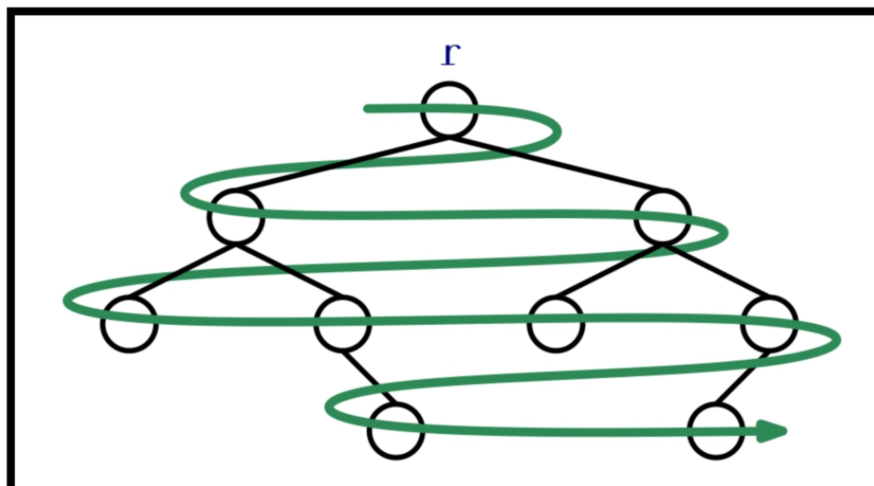
Level-order traversal of trees

Recursive MergeSort is somewhat like a postorder traversal of the recursion tree.

Iterative MergeSort is somewhat like a level-order traversal of the recursion tree, but bottom-up…

<u>LevelorderTrav(r):</u>

```
Queue q
q.add(r)
while (!q.empty())
  node = q.remove()
  if (node != NULL)
    Visit(node)
    q.add(node->left)
    q.add(node->right)
```



What is the time complexity? $\Theta(n)$

What is the space complexity? $\Theta(n)$ in the worst-case.

Reading

- [CLRS] Ch.10 (10.4)
- [Weiss] Ch.4 (4.1-4.2)
- [Morin] Ch.6 (6.1)