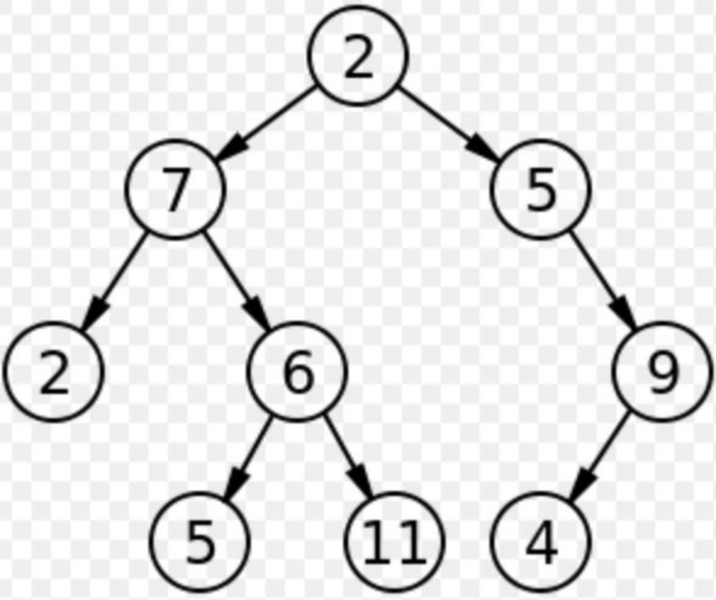
Tree && Graph

Presenter: Haotao (Eric) Lai
Contact: haotao.lai@gmail.com

What is a tree?



1. Node
2. Relation between node

1. General tree
2. Binary tree

1. AVL tree
2. Red-black tree
3. BK tree

Terminology of Tree (1)

Root: The top node in a tree.

Child: A node directly connected to another node when moving away from the Root.

Parent: The converse notion of a child.

Siblings: A group of nodes with the same parent.

Descendant: A node reachable by repeated proceeding from parent to child.

Ancestor: A node reachable by repeated proceeding from child to parent.

Leaf (External node): A node with no children.

Internal node: A node with at least one child.

Terminology of Tree (2)

Degree: The number of sub trees of a node.

Edge: The connection between one node and another.

Path: A sequence of nodes and edges connecting a node with a descendant.

Level: The level of a node is defined by 1 + (the number of connections between the node and the root).

Height of node: The height of a node is the number of edges on the longest path between that node and a leaf.

Height of tree: The height of a tree is the height of its root node.

Depth: The depth of a node is the number of edges from the tree's root node to the node.

Forest: A forest is a set of $n \geq 0$ disjoint trees.

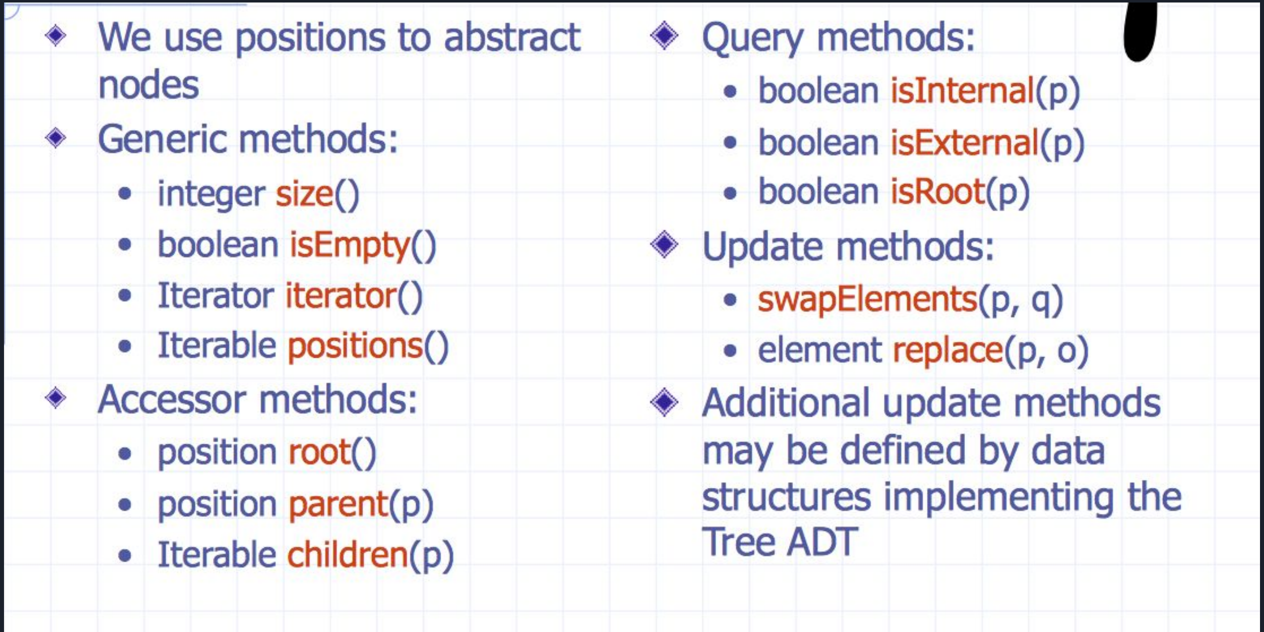
How to implement a tree?

Node and their Relation

Node, easy to understand, just create a Node as a container to wrap the actual data you need

Relation (Pointer), but NO pointer in Java. We already have node (reference)!!!

Tree ADT



- We use positions to abstract nodes
- Generic methods:
  - integer size()
  - boolean isEmpty()
  - Iterator iterator()
  - Iterable positions()
- Accessor methods:
  - position root()
  - position parent(p)
  - Iterable children(p)
- Query methods:
  - boolean isInternal(p)
  - boolean isExternal(p)
  - boolean isRoot(p)
- Update methods:
  - swapElements(p, q)
  - element replace(p, o)
- Additional update methods may be defined by data structures implementing the Tree ADT

How to construct a binary tree?

Rule:

1. Left child’s value is less or equal to its parent’s value;
2. Right child’s value is greater than its parent’s value;

1. Linked structure (kind of pointer?)
2. Array structure (use the index)

Try to do it !

Three ways to traverse a Tree

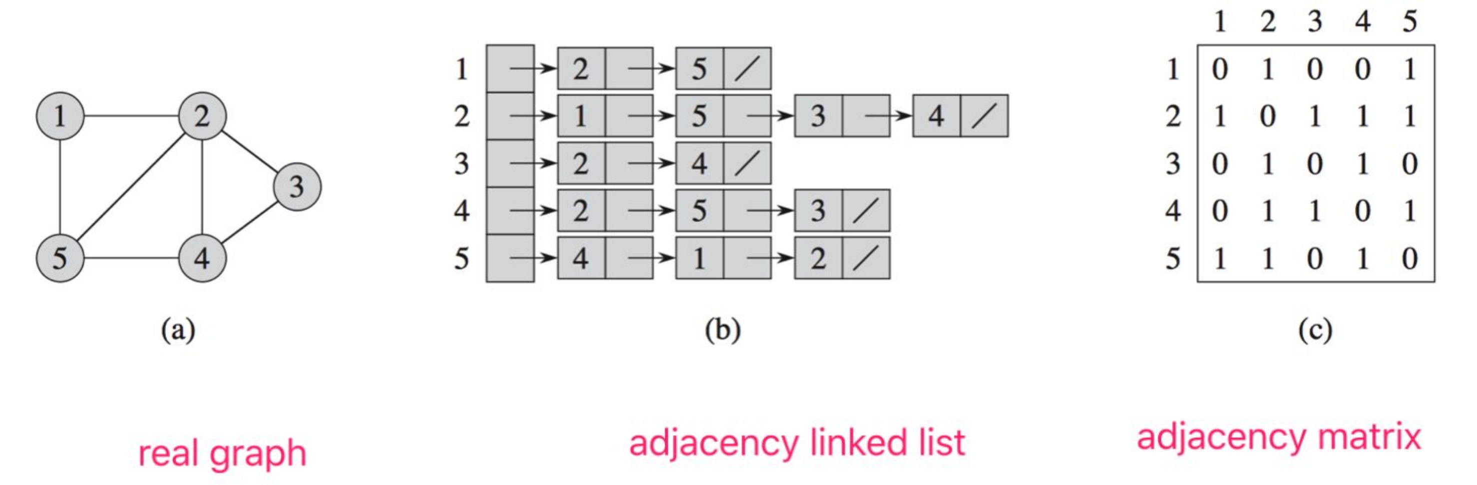
Pre-order: root, left child, right child

In-order: left child, root, right child

Post-order: left child , right child, root

Graph

How can we represent a graph



$$
\begin{array}{c|ccccc|}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 0 & 0 & 1 \\
2 & 1 & 0 & 1 & 1 & 1 \\
3 & 0 & 1 & 0 & 1 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 1 & 1 & 0 & 1 & 0 \\
\hline
\end{array}
$$

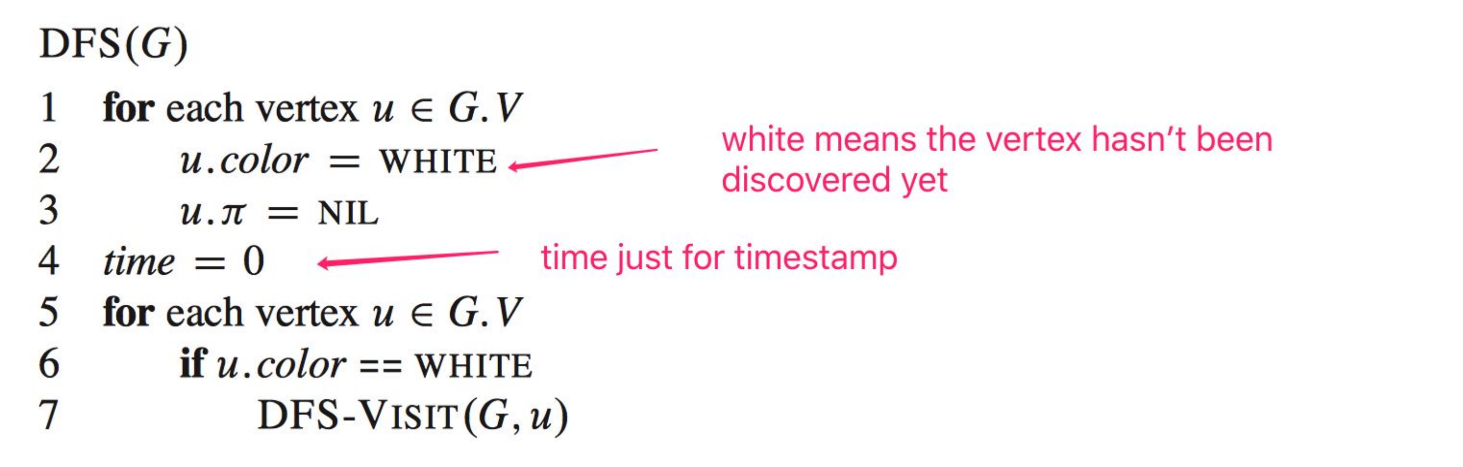
(a) (b) (c)

real graph

adjacency linked list

adjacency matrix

Recursion

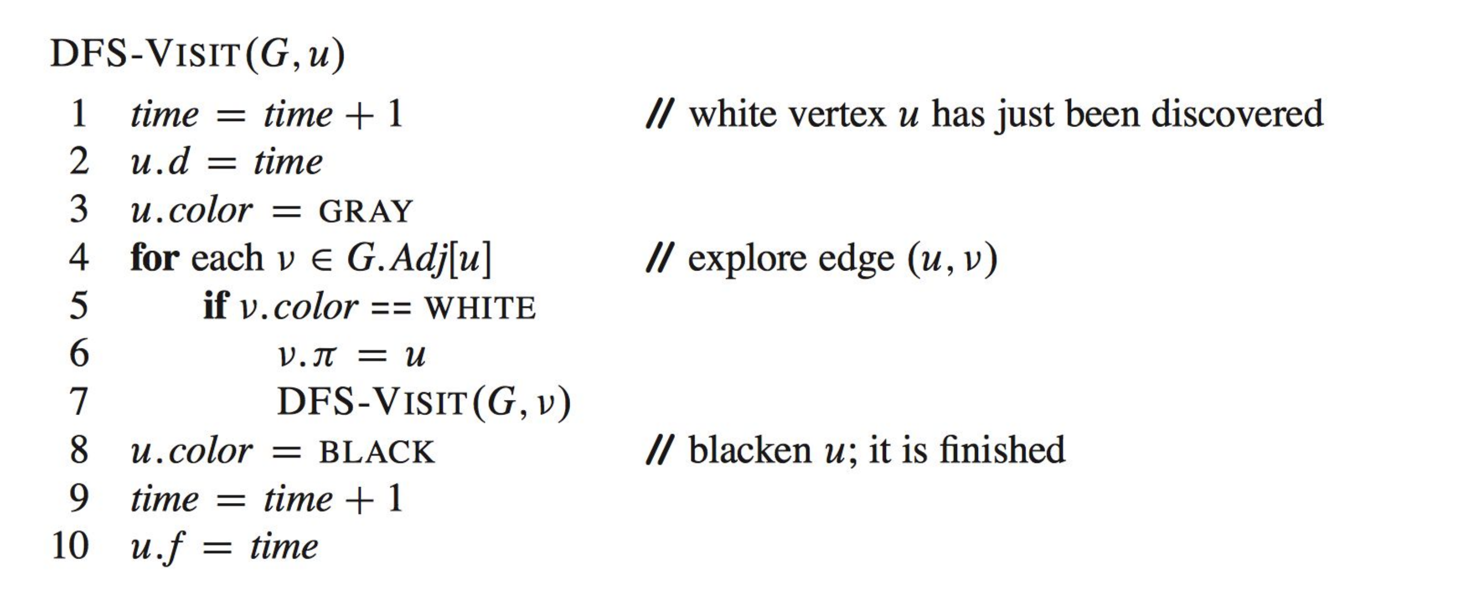


```
DFS($G$)
for each vertex $u \in G.V$
    $u.color = \text{WHITE}$
    $u.\pi = \text{NIL}$
$time = 0$
for each vertex $u \in G.V$
    if $u.color == \text{WHITE}$
        DFS-VISIT($G, u$)
```

white means the vertex hasn’t been discovered yet

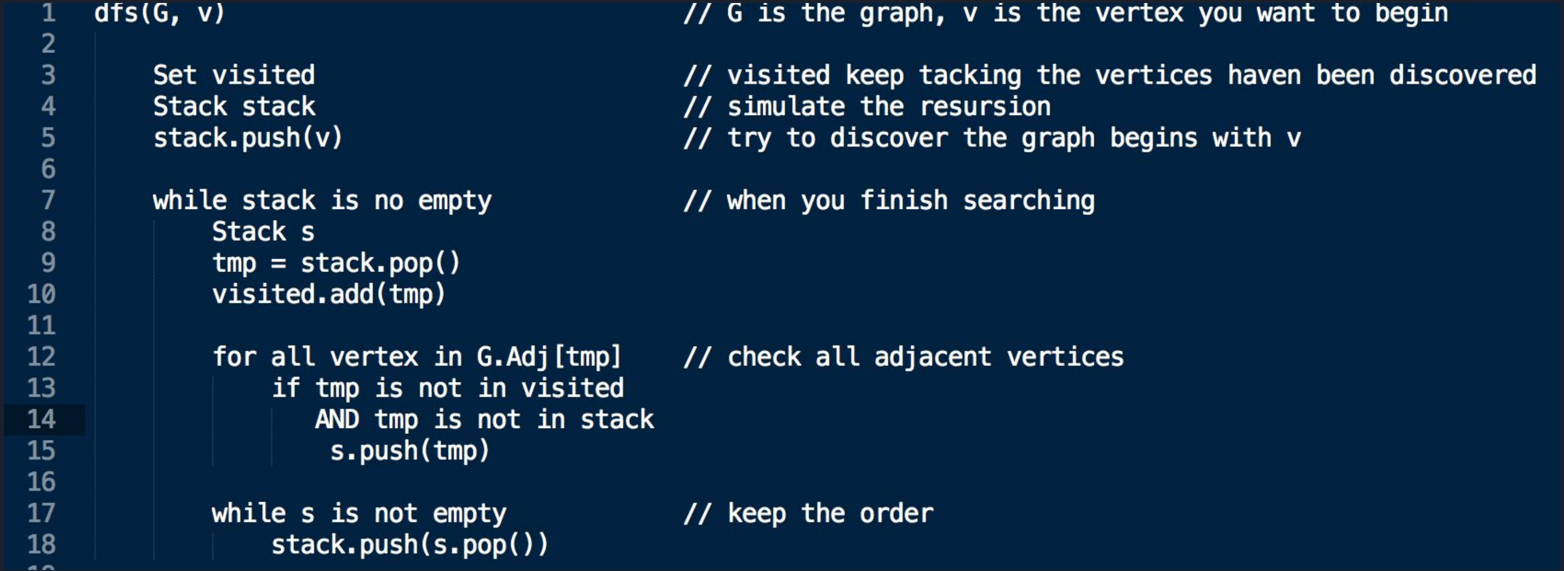
time just for timestamp

Recursion (continue)



```
DFS-VISIT$(G, u)$
$time = time + 1$                      // white vertex $u$ has just been discovered
$u.d = time$
$u.color$ = GRAY
for each $v \in G.Adj[u]$              // explore edge $(u, v)$
    if $v.color$ == WHITE
        $v.\pi = u$
        DFS-VISIT$(G, v)$
$u.color$ = BLACK                      // blacken $u$; it is finished
$time = time + 1$
$u.f = time$
```

Loop



```
dfs(G, v)                               // G is the graph, v is the vertex you want to begin

    Set visited                         // visited keep tacking the vertices haven been discovered
    Stack stack                         // simulate the resursion
    stack.push(v)                       // try to discover the graph begins with v

    while stack is no empty             // when you finish searching
        Stack s
        tmp = stack.pop()
        visited.add(tmp)

        for all vertex in G.Adj[tmp]    // check all adjacent vertices
            if tmp is not in visited
               AND tmp is not in stack
                s.push(tmp)

        while s is not empty            // keep the order
            stack.push(s.pop())
```

Levenshtein Distance

How to calculate Levenshtein Distance?

Assume we have string A and string B

1. Let d(i, j) represent the LD from the substring of A contains i characters to the substring of B contains j characters;
2. If i = 0 and j = 0?
3. If i = 0 and j > 0?
4. If i > 0 and j = 0?
5. If i >= 1 and j >= 1?