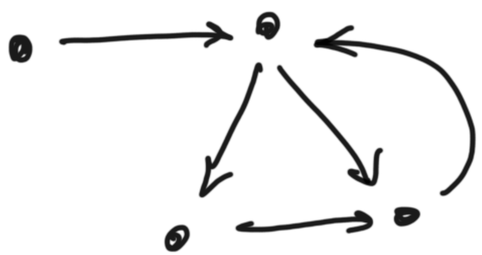
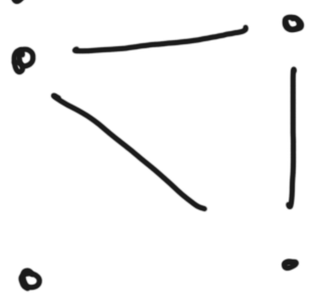
CS 2110 : Trees

Graph: nodes (vertices) connected by edges



directed



Undirected

linked list: directed acyclic graph (DAG)

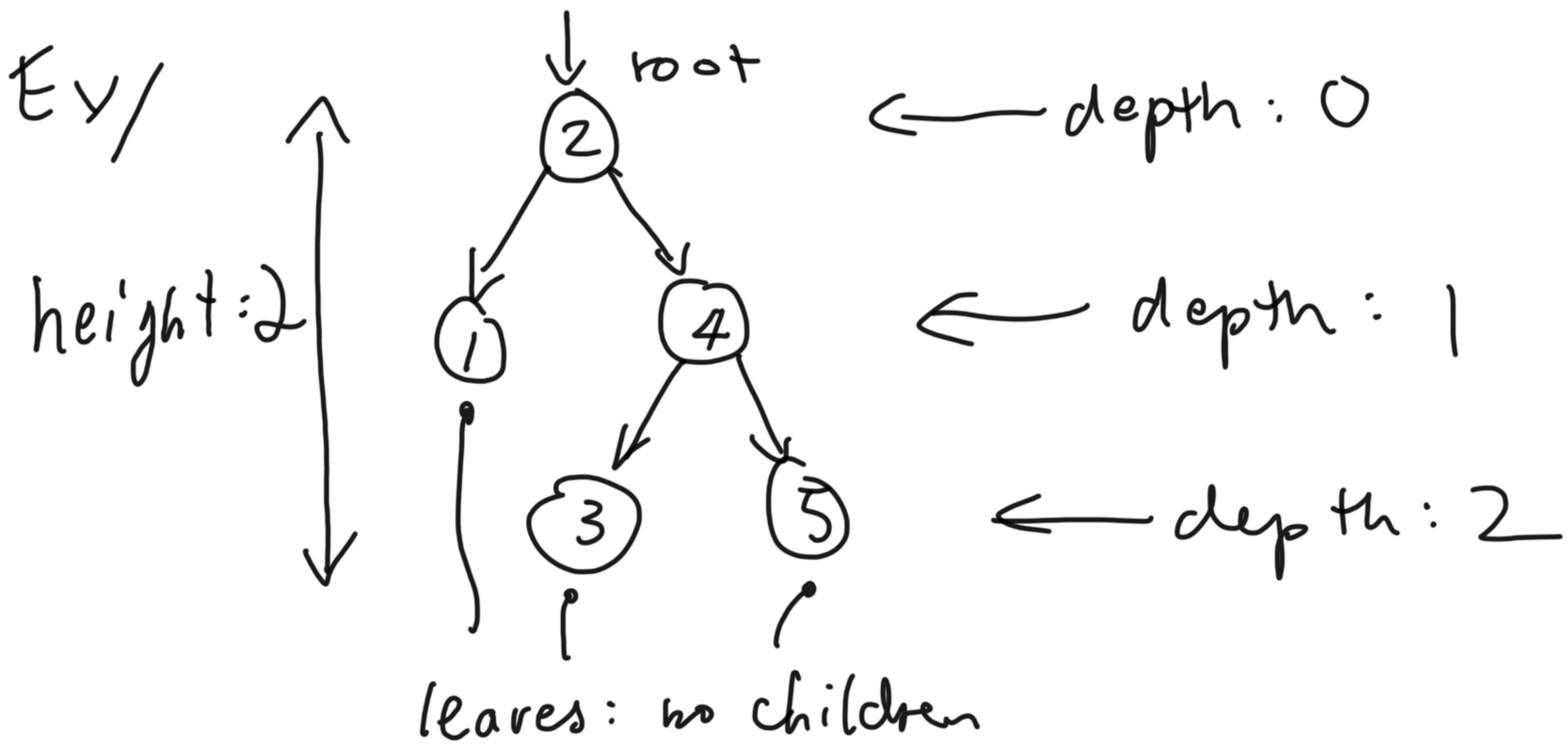
- edges are arrows
- no cycles

Each node has
- 0 or 1 successor
- 0 or 1 predecessor

Time to find node w/ data?

A: O(1)
B: O(n)
C: O(lg n)
D: O(n²)

tree: DAG where nodes can have >1 successor (children)
& 1 predecessor node (exc. root node)



In code:

```
class BinaryNode<T> {
    T data;
    BinaryNode<T> left, right;  // may be null
}
```

— node of a binary tree: up to 2 distinguished children (left, right)

N-ary tree?

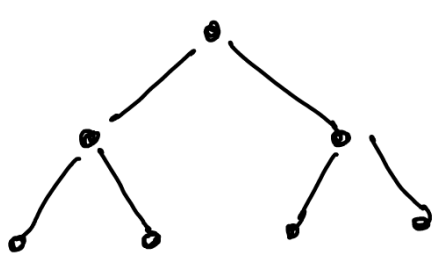
```
class NAryNode<T> {
    T data;
    NAryNode<T>[] children;
}
```

— or: linked list, set, ...

Why trees?

1) Some information has natural tree structure
   - org charts
   - class hierarchy
   - game states
   - genealogy
2) Way to find information quickly

Consider **full binary tree**: all leaves at same depth, all non-leaves have 2 children.



$h = 2,\ n = 7$

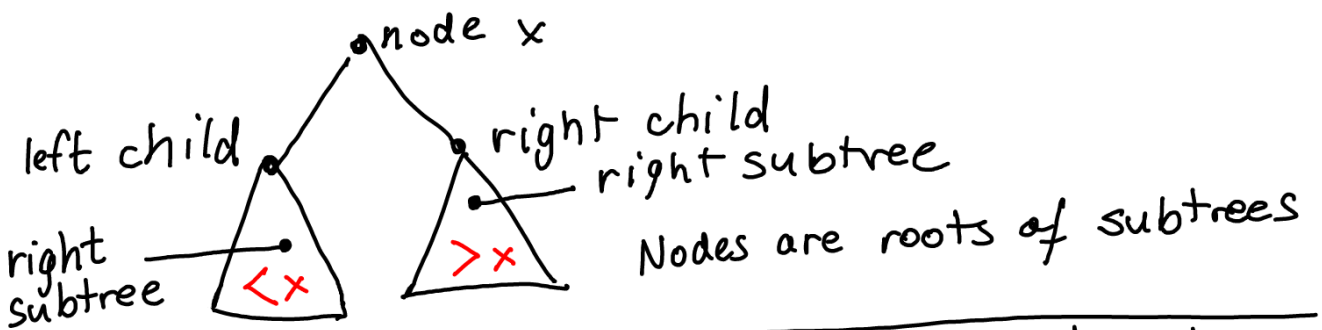
$$n = 1 + 2 + 4 + 8 + \cdots + 2^h = 2^{h+1} - 1$$

$$h = \lg(n+1) - 1 = O(\lg n)$$

⇒ following a path from root takes logarithmic time

⇒ can find elems in logarithmic time if:
- path is known
- tree is balanced: $h = O(\lg n)$

Binary Search Trees (BSTs)



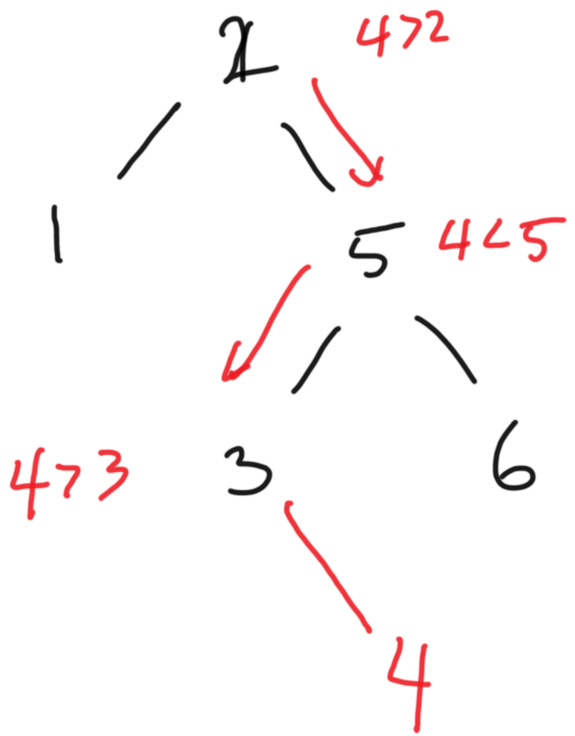
To find element e in tree with root value x:

- Compare e to x
  - e=x? done.
  - e<x? recurse in left subtree unless empty (null)
  - e>x? recurse in right subtree unless empty (null)

BST Invariant

Given node holding data x:

- In left subtree, all data < x
- In right subtree, all data > x
- Left & Right subtrees satisfy BST invariant (recursively)



add(4)

Performance: O(h)

Adding in sorted order? 1, 2, 3, ..., n

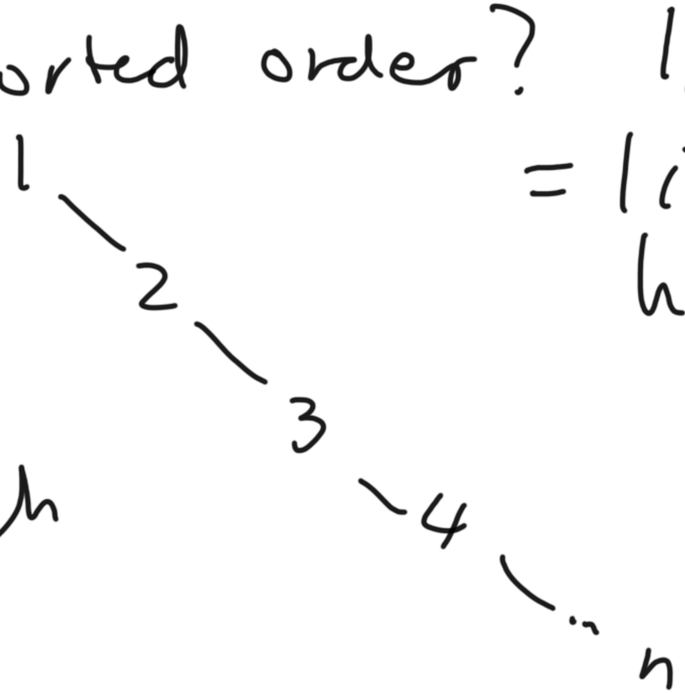
= linked list

h = O(n)

Add in random order

E(h) = O(lg n)

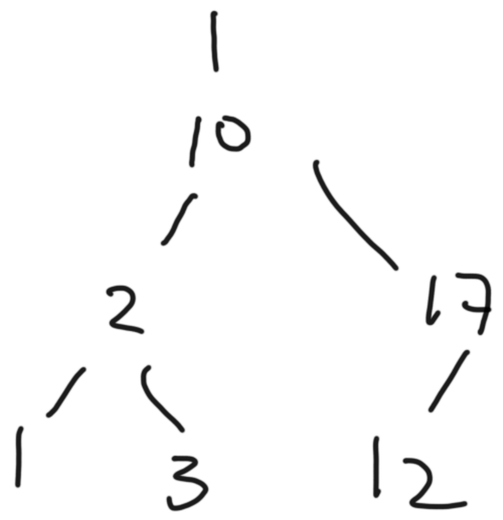
balanced with high probability



Traversals

visit tree nodes

In-order traversal.



1, 2, 3, 10, 12, 17

```
void traverse() {
    if (left != null) {
        left.traverse();
    }
    visit(data); // do something
    if (right != null) {
        right.traverse();
    }
}
```

In-order BST traversal sorts elements

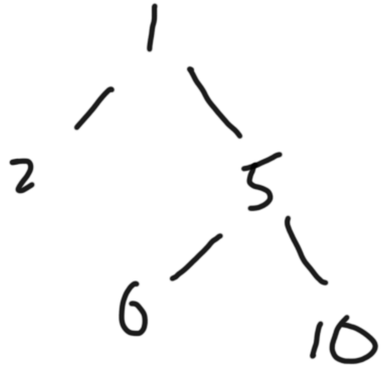
Preorder traversal

visit(data)
left traverse
right traverse.

Postorder traversal

left traverse
right traverse
visit (data)

Node visited after all descendants.



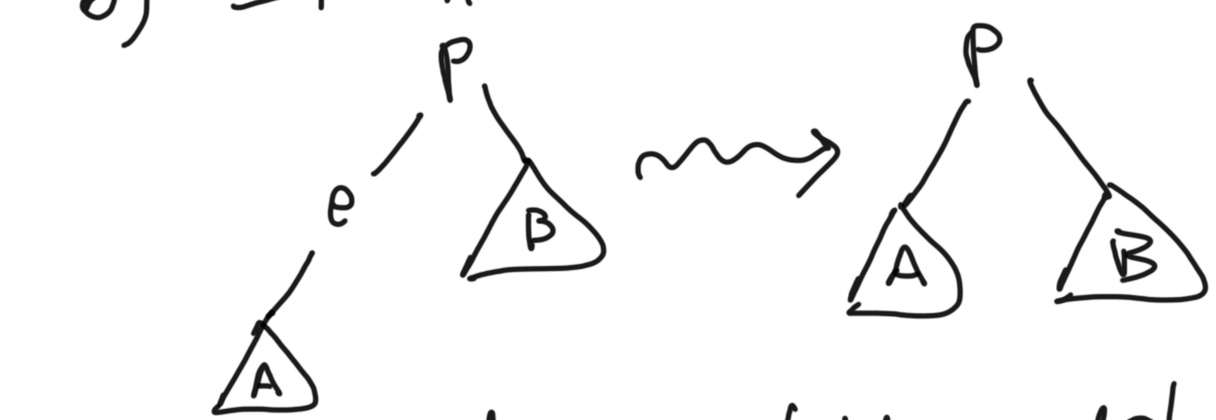
2, 6, 10, 5, 1

Removing elements from BST

1. search to node n containing element e.
   let p be parent of n.

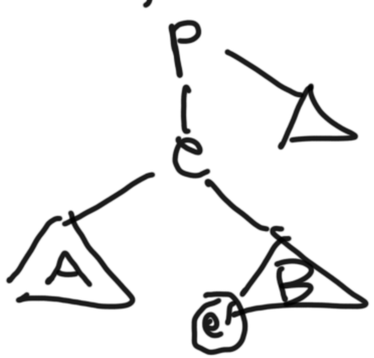
2. a) If n is a leaf, prune

b) If n has <2 children, splice



preserves BST invariant

c) If n has 2 children, replace



- Find next element e'
  = walk right, then left as far as possible
- Remove e' (prune/splice)
- $e.data = e'.data$