CSE202
Design and Analysis of Algorithms

Week 10 — Balance against Worst-Case

Data-Structures for Ordered Data

Priority Queues: insert, findmax, deletemax

Ordered Search Trees: insert, find, delete, selectbyrank, floor, ceiling, countbetween,...

Sorting first is not an option

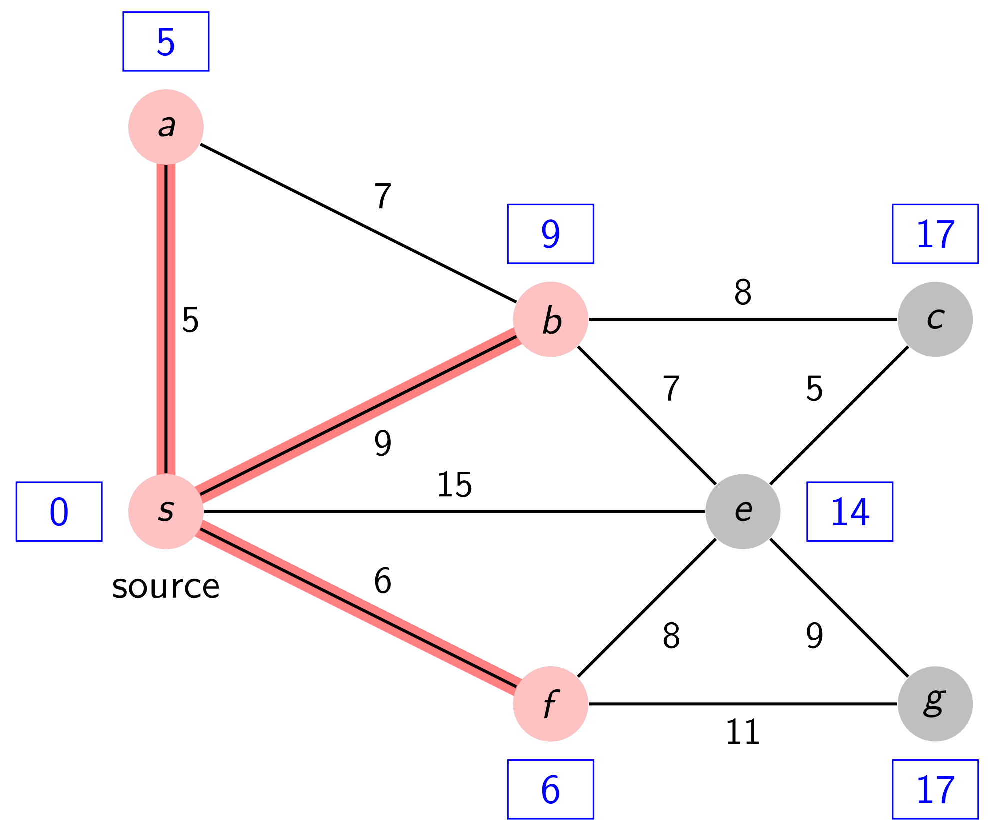
Def. All leaves in the same one or two levels

Balanced trees allow for all these operations in worst-case time $O(\log n)$.

n	$\log_2 n$
10^6	≈ 20
10^9	≈ 30
10^{12}	≈ 40

I. Priority Queues & Heap-ordered Trees

Recall Dijkstra's Algorithm (CSE103)



while PQ not empty:

remove first edge ((u,v),d(s,u)) from PQ

if v not in the tree

add v to the tree

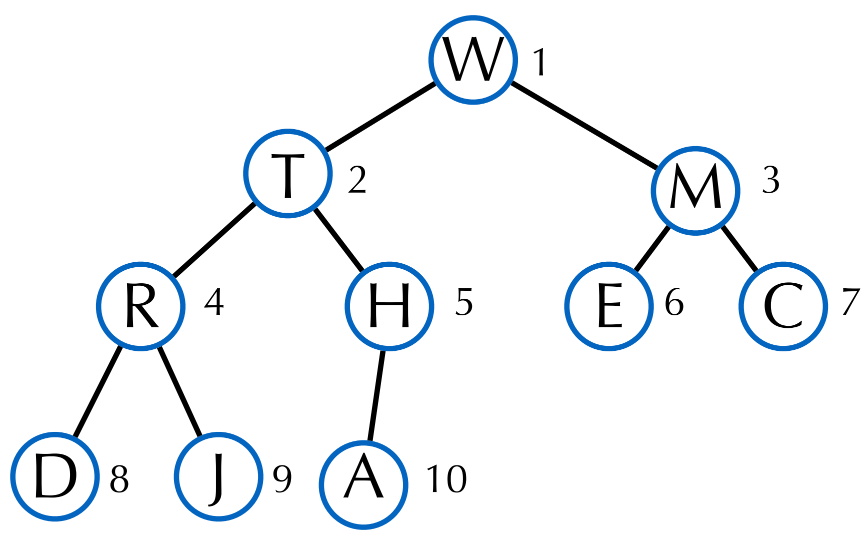
for all neighbours w of v

insert ((v,w),d(s,u)+d(v,w)) in PQ

Complexity depends on good priority queues

Heaps

Each node is larger than its children



Operations:
insert,
findmax,
deletemax

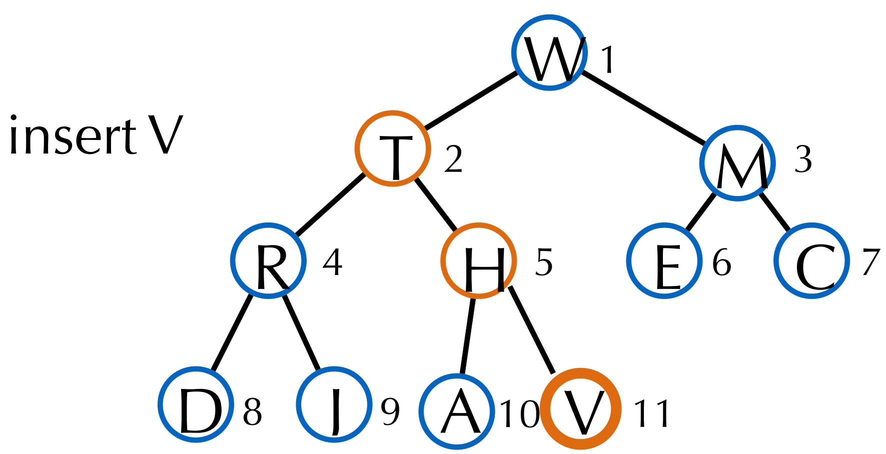
Simple application: find the M smallest elements in a stream in time $O(N \log M)$

Array representation:

	W	T	M	R	H	E	C	D	J	A						
0	1															16

Basic Operations

Insert & fixup

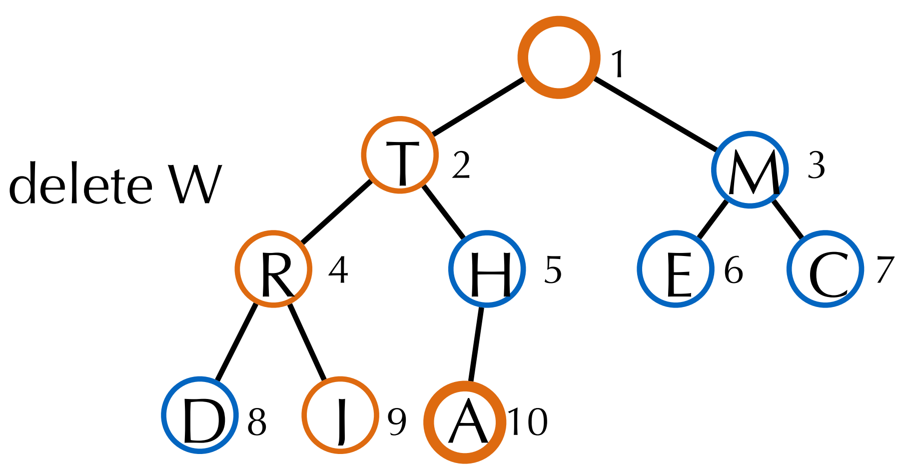


$\leq \log_2 n$ comparisons

```
def insert(self,key):
    self.size += 1
    self.PQ[self.size]=key
    self.fixup(self.size)
```

```
def fixup(self,ind):
    if ind==1: return
    parent = ind // 2
    if self.PQ[parent]>self.PQ[ind]: return
    self.exch(parent,ind)
    self.fixup(parent)
```

Deletemax & fixdown

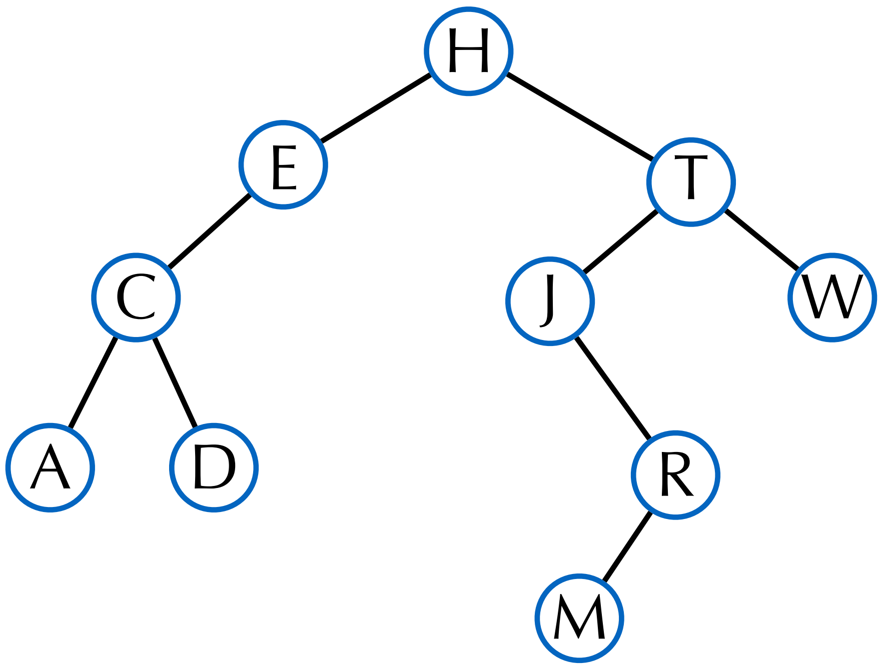


$\leq 2\log_2 n$ comparisons

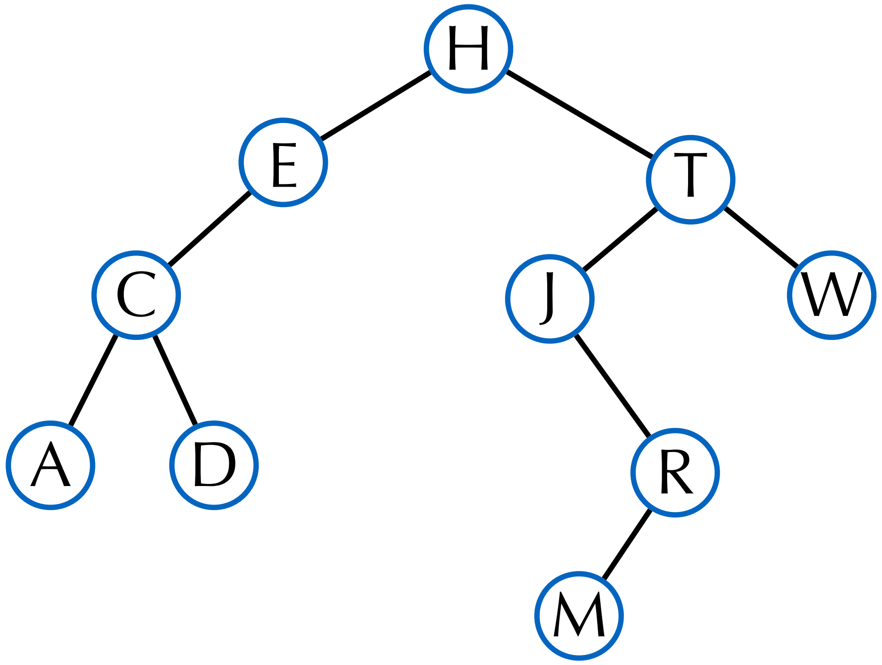
```
def deletemax(self):
    self.PQ[1] = self.PQ[self.size]
    self.size -= 1
    self.fixdown(1)
```

```
def fixdown(self,ind):
    child = 2*ind
    if child>self.size: return
    if child<self.size and \
       self.PQ[child+1]>self.PQ[child]:
        child +=1
    if self.PQ[ind]<self.PQ[child]:
        self.exch(ind,child)
        self.fixdown(child)
```

II. Binary Search Trees



Recall Definition (CSE101 & 102)



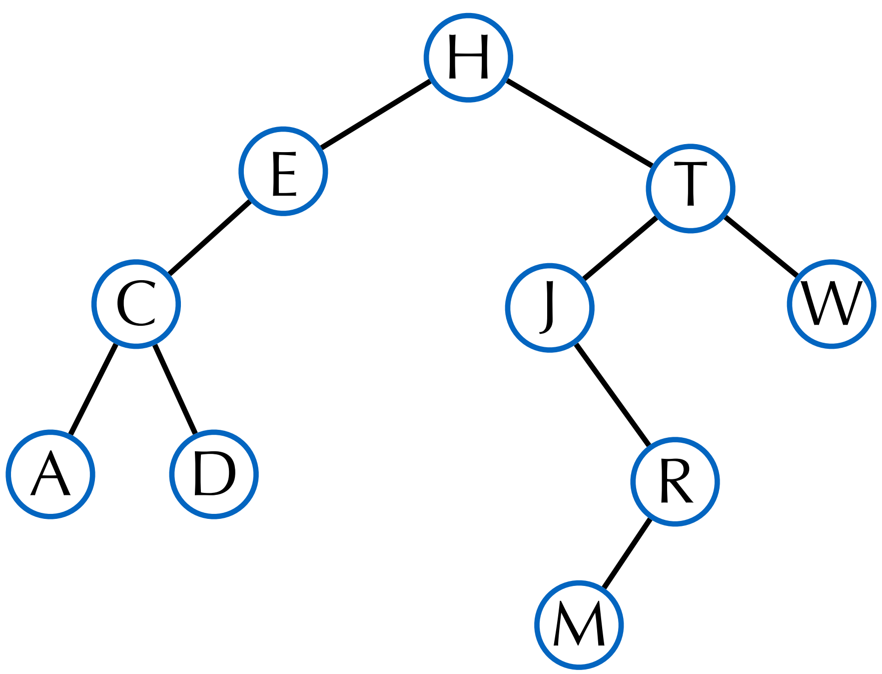
Smaller elements to the left,
larger elements to the right

```
class Node:

    def __init__(self,key,left=None,right=None):
        self.key = key
        self.left = left
        self.right = right
```

```
class BST:

    def __init__(self):
        self.root = None

    def find(self,key):
        return self._find(self.root,key)

    def insert(self,key):
        self.root = self._insert(self.root,key)

    def delete(self,key):
        self.root = self._delete(self.root,key)
```

Find/Insert

```
def _find(self,node,key):
    if node is None: return False
    if node.key > key: return self._find(node.left,key)
    if node.key < key: return self._find(node.right,key)
    return True
```




```
def _insert(self,node,key):
    if node is None: return Node(key)
    if node.key > key:
        node.left = self._insert(node.left,key)
    elif node.key < key:
        node.right = self._insert(node.right,key)
    return node
```

Delete slightly more complicated (CSE102)

Worst-case: search in $O(n)$ comparisons for a BST built from n keys.

Average-Case Analysis

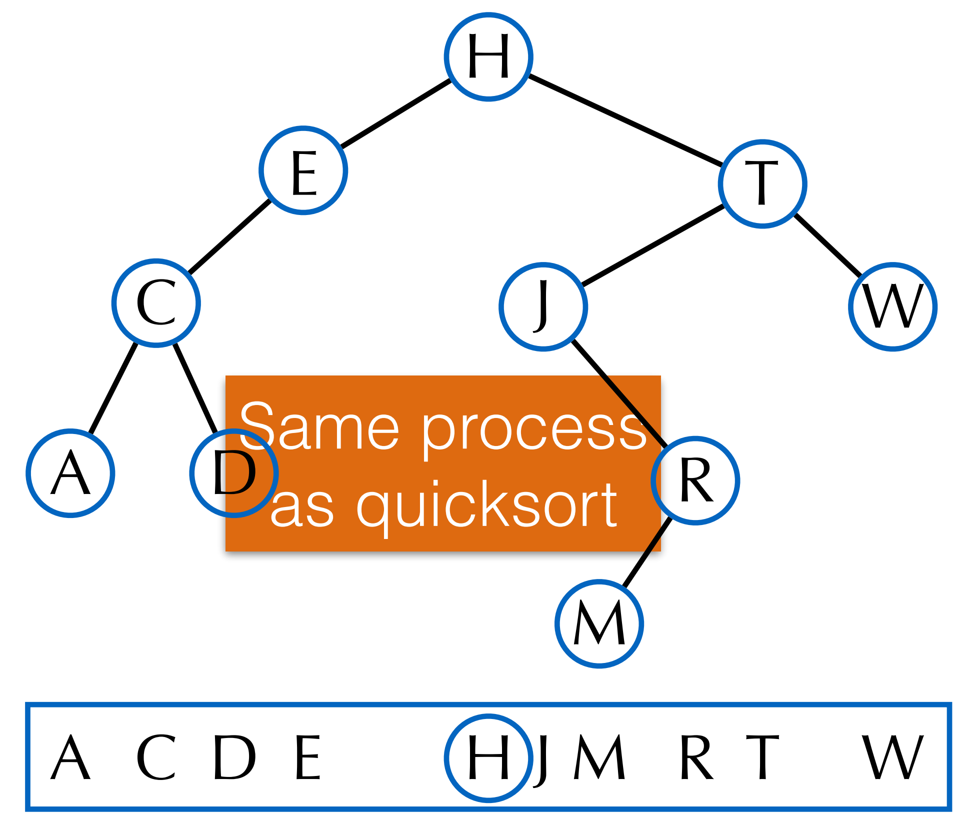
Internal path length:

$P_n :=$ sum depths of all nodes

$P_n/n + 1$: average successful search

$P_n/n + 3$: average unsuccessful search (= insert)

Blackboard proof



Prop. In a BST built from n random keys, the average number of comparisons for a search is

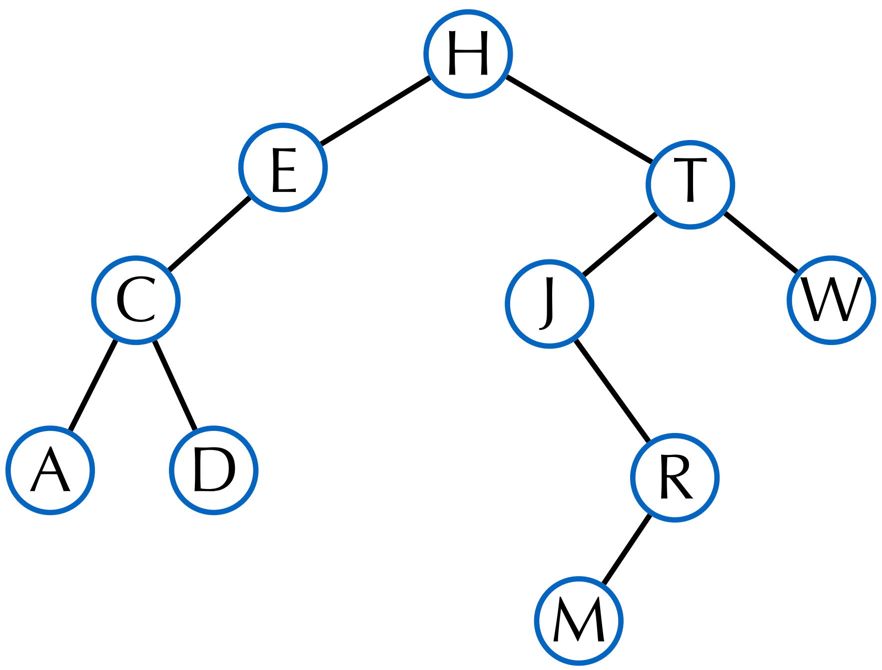
$$2 \log n + O(1) \approx 1.39 \log_2 n$$

$$P_0 = P_1 = 0$$

$$\mathbb{E}P_n = n - 1 + \sum_{i=1}^{n} \frac{\mathbb{E}P_{i-1} + \mathbb{E}P_{n-i}}{n}$$

Same recurrence as in the analysis of quicksort.

Select



min, max, floor, ceiling: easy

floor: largest key smaller than input

median,select:

change nodes into

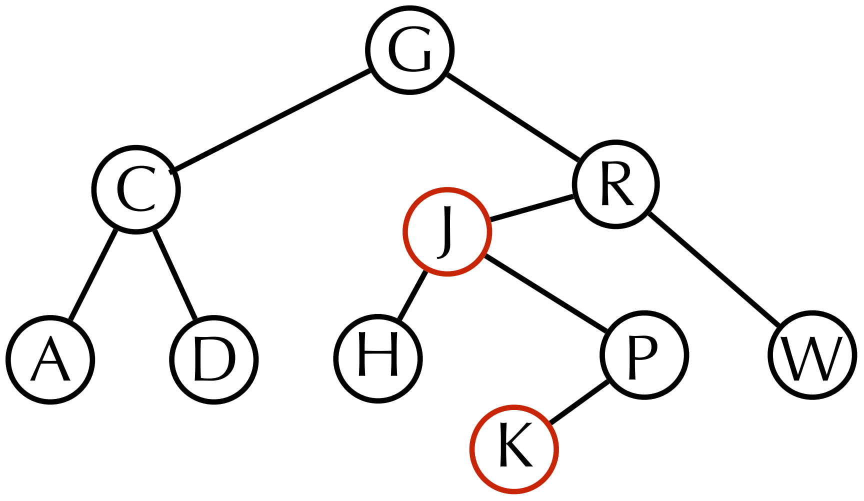
`key,left,right,size`

```
def _insert(self,node,key):
    if node is None: return Node(key)
    if node.key > key:
        node.left = self._insert(node.left,key)
    elif node.key < key:
        node.right = self._insert(node.right,key)
    node.size = 1+size(node.left)+size(node.right)
    return node
```

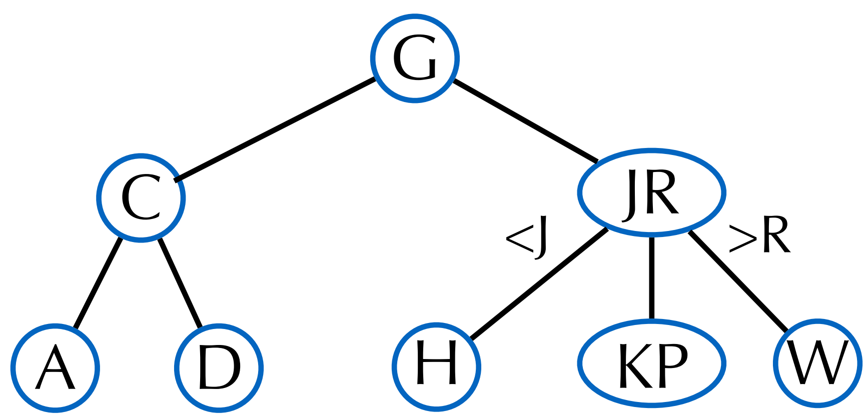
All these operations have cost bounded by the height, which is logarithmic on average.

Generalizes to higher dimensions (quadtrees).

III. Red-Black BST



WarmUp: 2-3 Search Trees

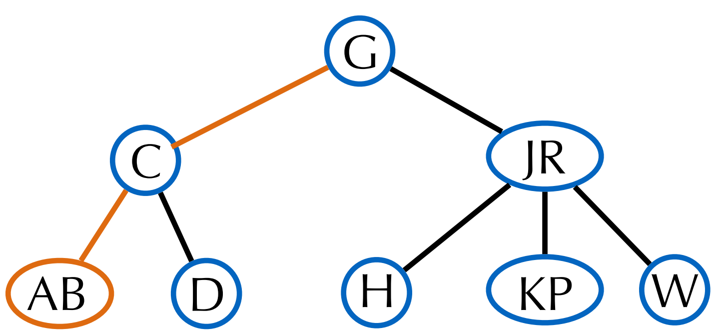


All leaves at the same level

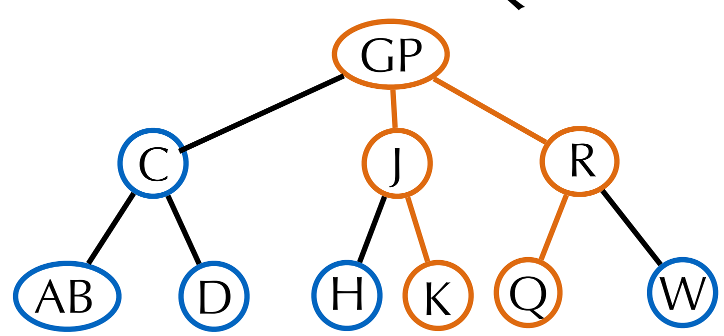
Find: same as BST

Insert maintaining perfect balance: search, insert at bottom and propagate upwards

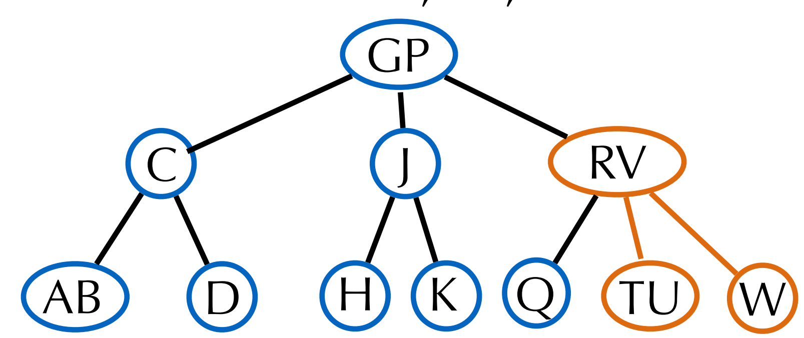
Insert B



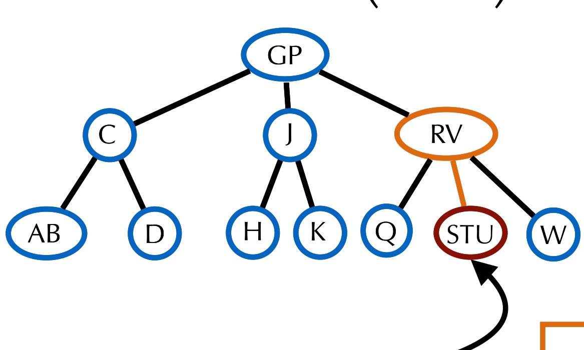
Insert Q



Insert V,U,T

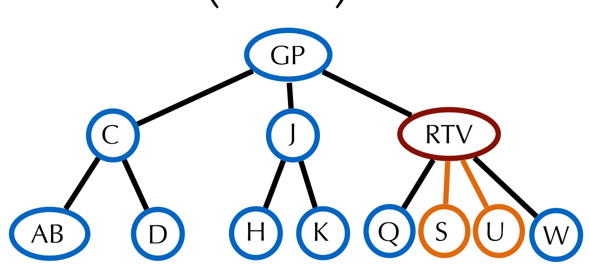


Insert S (1/4)

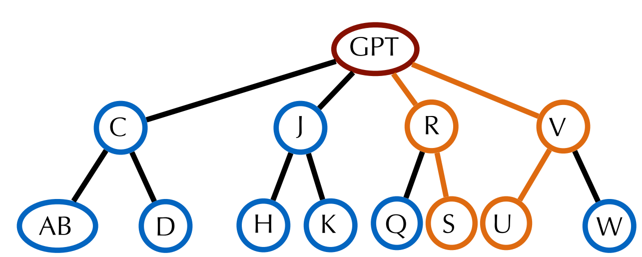


has to disappear

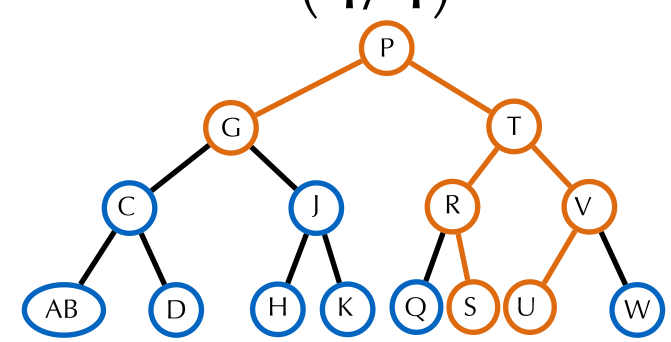
(2/4)



(3/4)

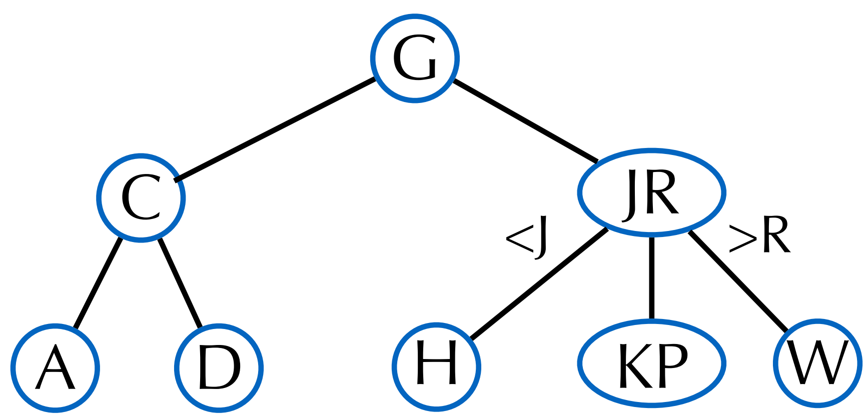


(4/4)

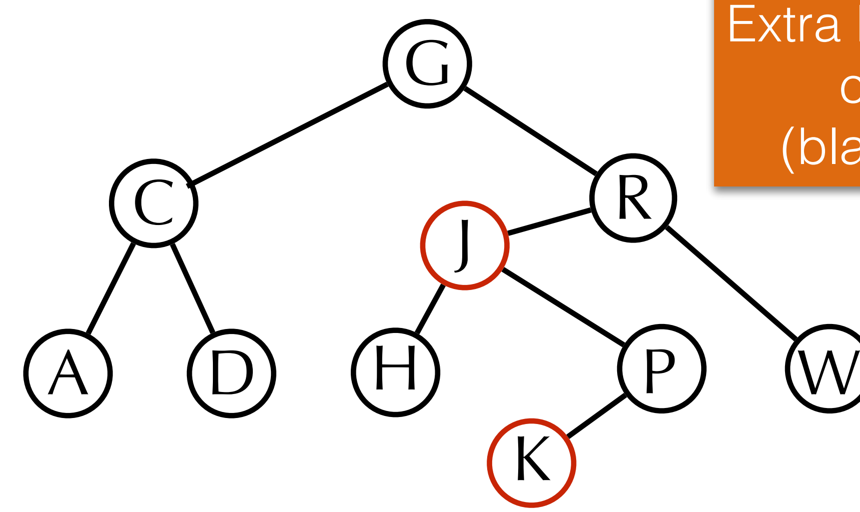


Worst-case number of nodes visited for find/insert: $\log_2 n$

(Left-Leaning) Red-Black Trees



stored as a coloured BST



Extra Node Info: colour (black root)

Properties:
1. red nodes are left children;
2. red nodes have black children;
3. every path from the root to a leaf has the same number of black nodes.

Black balance

Red-black trees with these properties are in 1-to-1 correspondance with 2-3 trees.

Sedgewick (2008)

`find`, `select`: code for BST unchanged! Just faster.

Insertion

Insert maintaining order & black balance: search, insert red node at bottom and propagate upwards

```
def _insert(self,node,key):
    if node is None: return Node(key,red=True)
    if node.key > key:
        node.left = self._insert(node.left,key)
    elif node.key < key:
        node.right = self._insert(node.right,key)
    if isRed(node.right) and not isRed(node.left): node = rotateleft(node)
    if isRed(left.red) and isRed(node.left.left): node = rotateright(node)
    if isRed(node.left) and isRed(node.right): flipcolors(node)
    node.size = 1+size(node.left)+size(node.right)
    return node
```

Local fixes

red-black trees

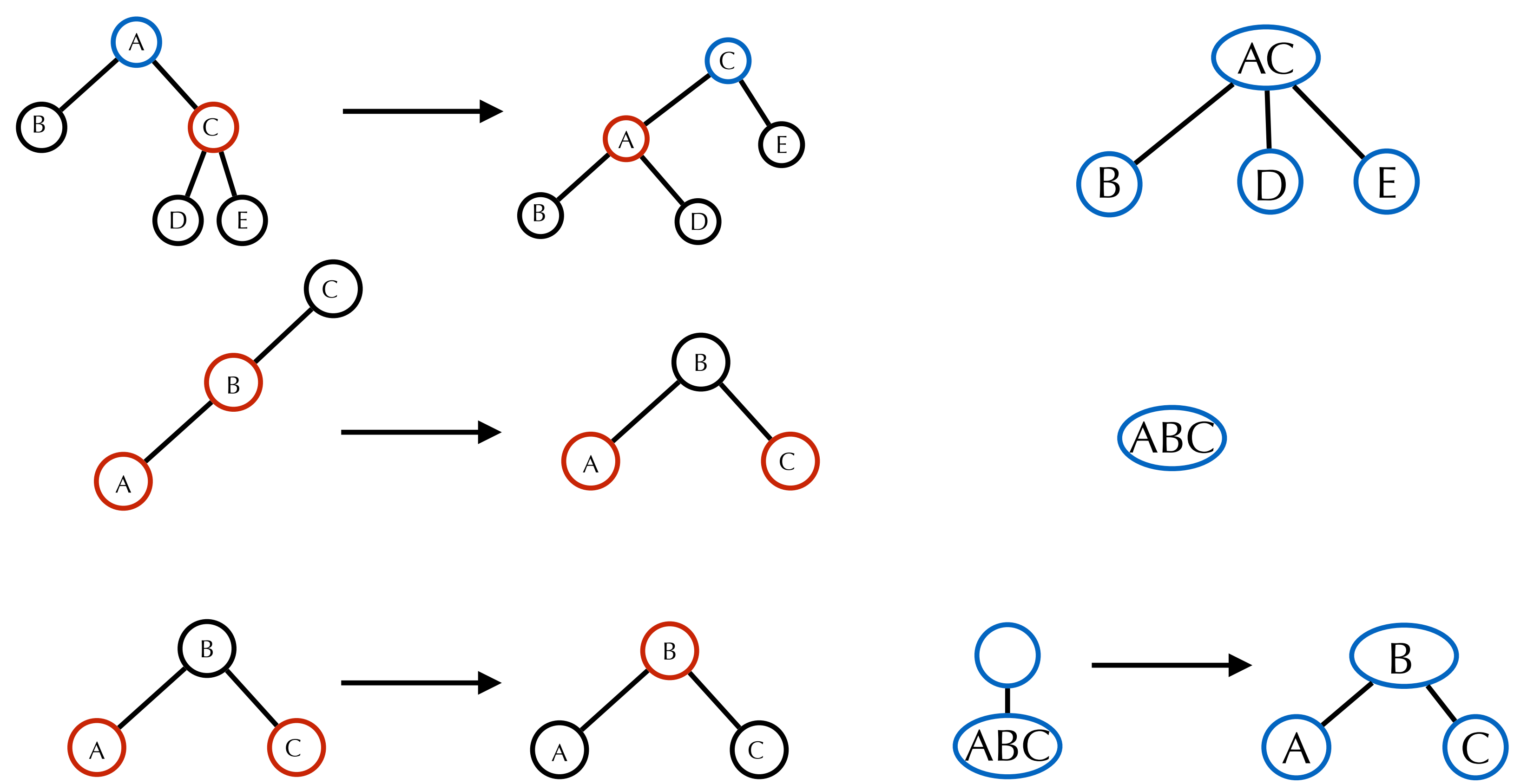
2-3 tree

Check that order & black balance are preserved

rotateleft

rotateright

flipcolors



Delete more complicated

Worst-Case Analysis

Prop. The height of a red-black BST with n nodes is bounded by $2\log_2 n$.

Proof: exercise.

Summary

algorithm (data structure)	worst-case cost (after N inserts)		average-case cost (after N random inserts)	
	search	insert	search hit	insert
sequential search (unordered linked list)	N	N	$N/2$	N
binary search (ordered array)	$\lg N$	N	$\lg N$	$N/2$
binary tree search (BST)	N	N	$1.39 \lg N$	$1.39 \lg N$
2-3 tree search (red-black BST)	$2 \lg N$	$2 \lg N$	$1.00 \lg N$	$1.00 \lg N$

Empirical.
No proof yet.

(Sedgewick-Wayne 2011)

References for this lecture

The slides are designed to be self-contained.

They were prepared using the following
book that I recommend if you want to learn more:

Next

Assignment: balanced BST

Next tutorial: Sudoku by WalkSat

Next week: String Algorithms 1

Feedback

Moodle for the slides, tutorials and exercises.

Questions or comments: Bruno.Salvy@inria.fr