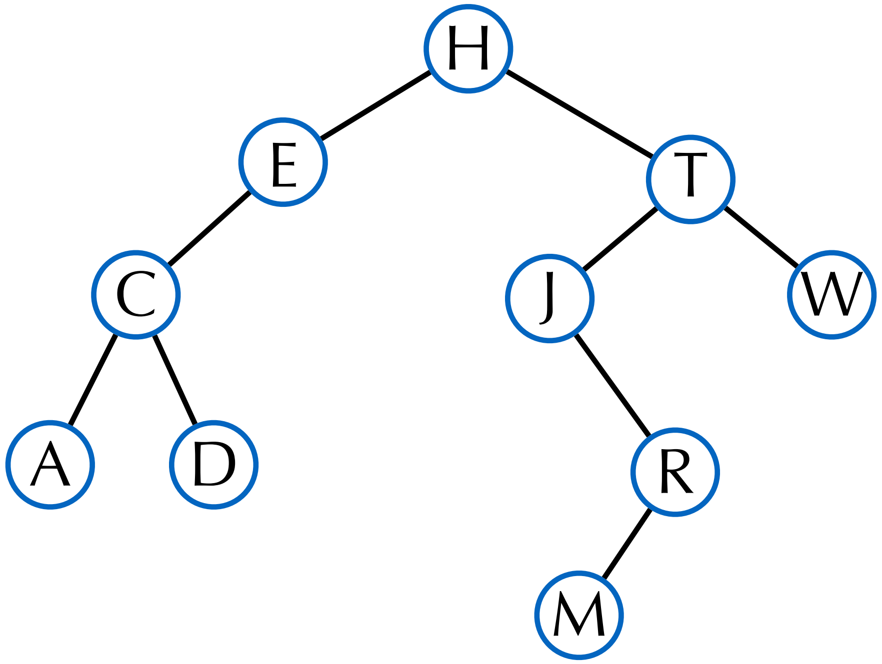
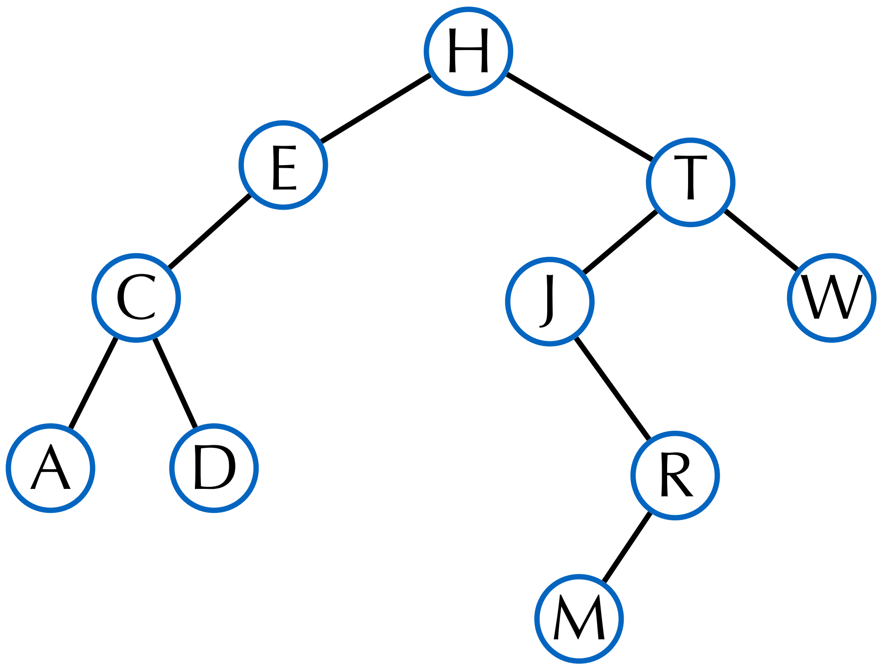
CSE202
Design and Analysis of Algorithms

Week 10 — Balance against Worst-Case

II. Binary Search Trees



Recall Definition (CSE101 & 102)



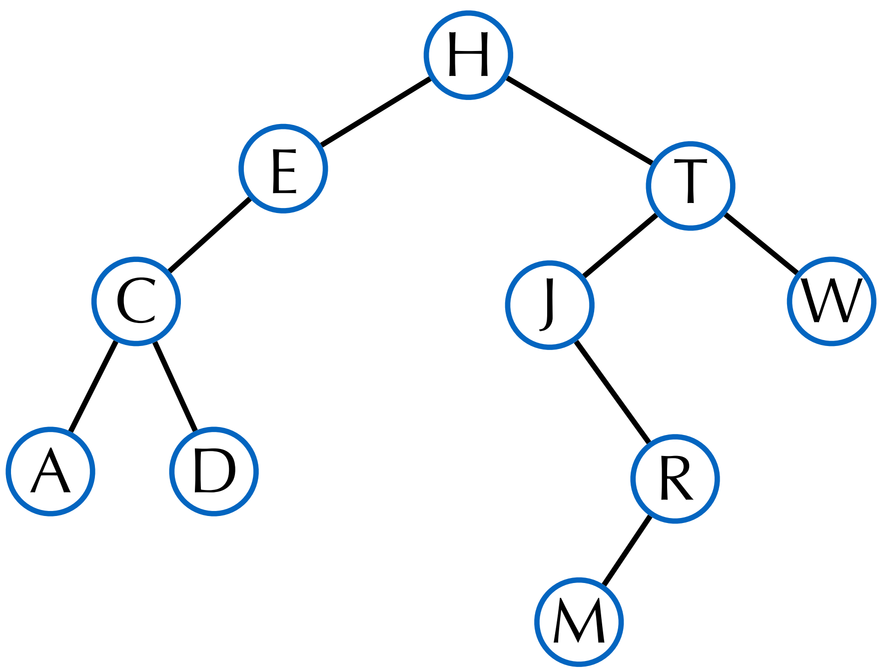
Smaller elements to the left,
larger elements to the right

```
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

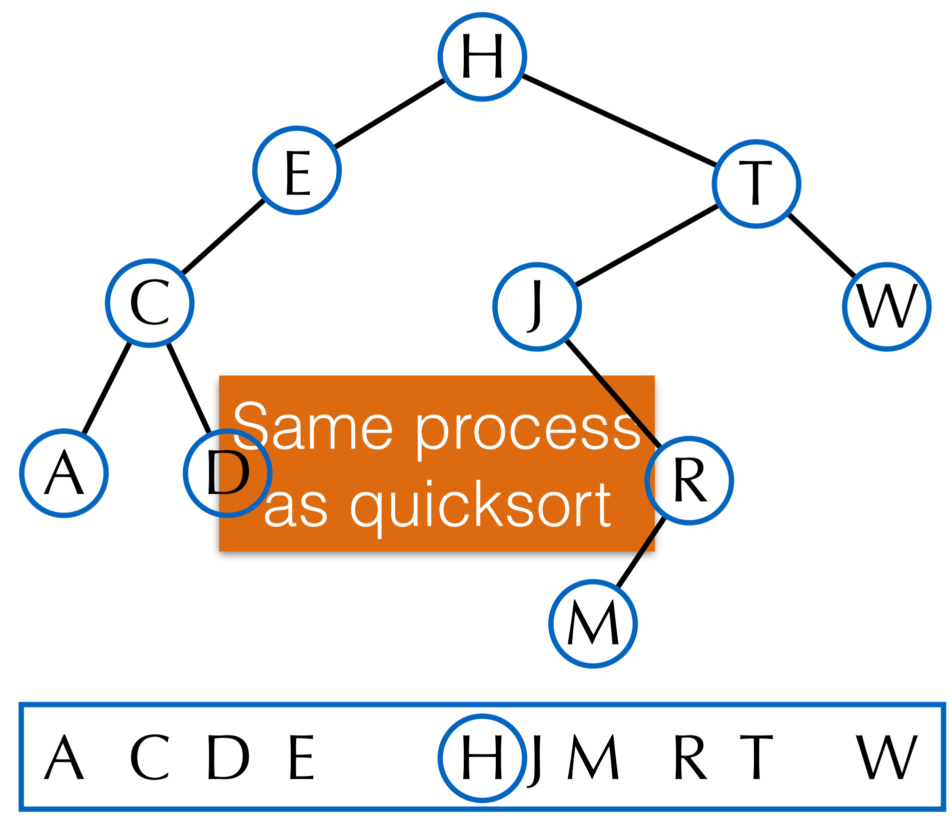
Internal path length:

$P_n :=$ 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

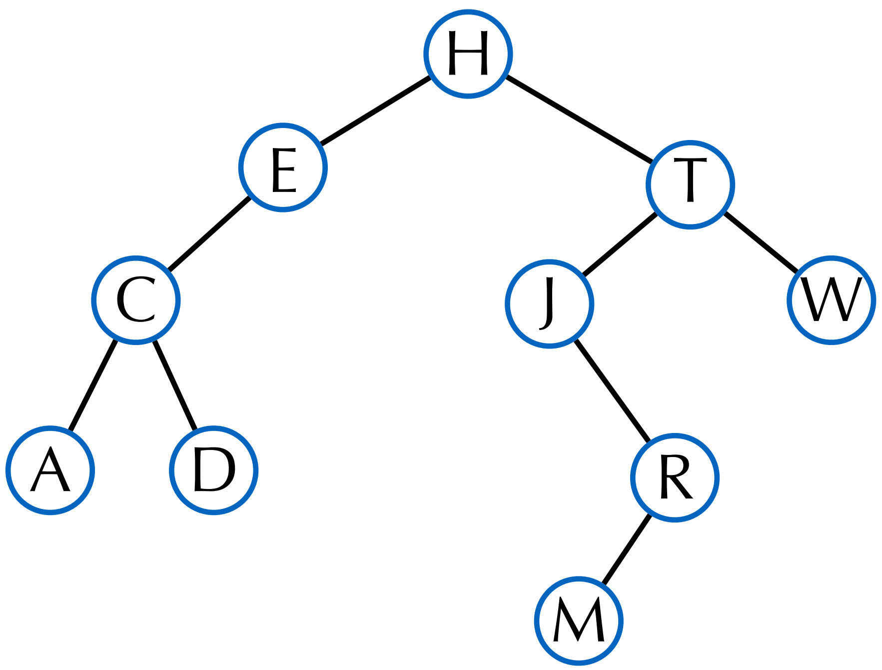
$$1.39 \log_2 n \; + \; O(1)$$

$$P_0 = P_1 = 0$$

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

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

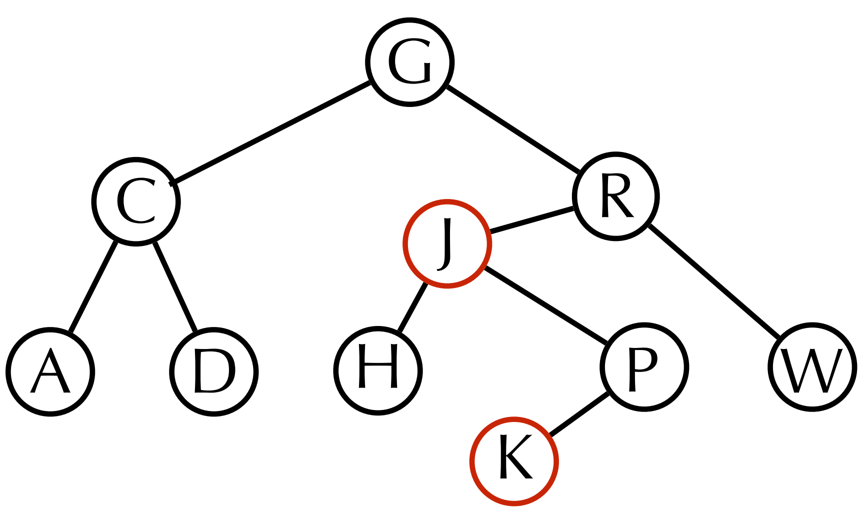
`key,left,right,size`

```
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
```

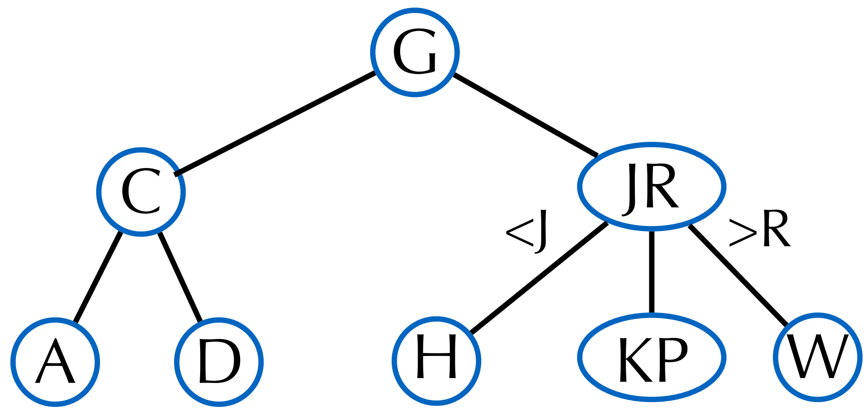
All these operations have cost bounded by the height, which is logarithmic on average.

Generalizes to higher dimensions (quadtrees).

III. Red-Black BST



WarmUp: 2-3 Search Trees

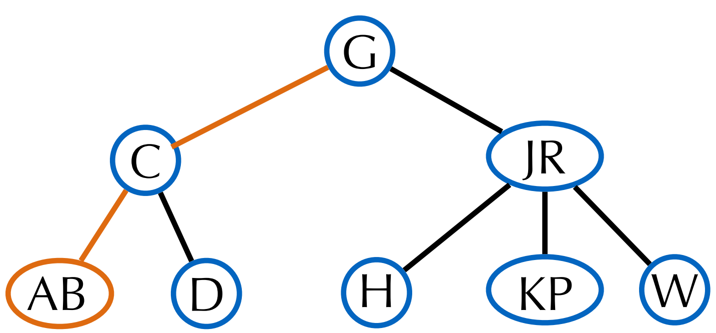


Find: same as BST

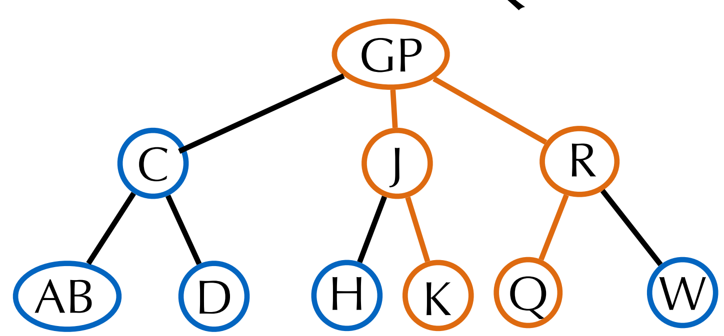
All leaves at the same level

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

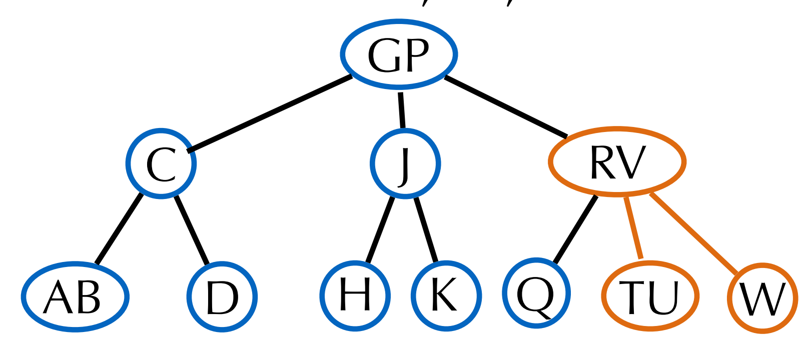
Insert B



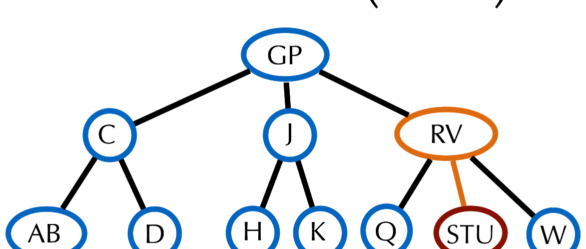
Insert Q



Insert V,U,T

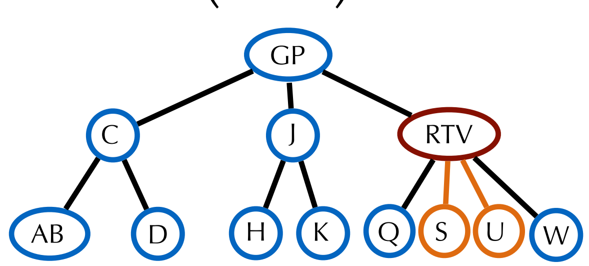


Insert S (1/4)

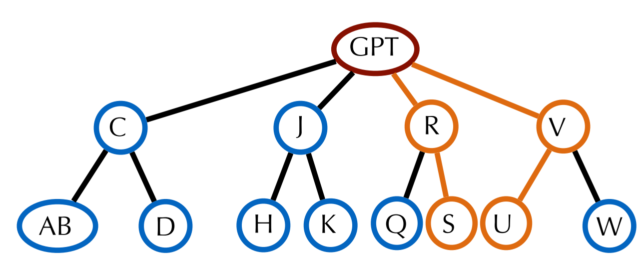


has to disappear

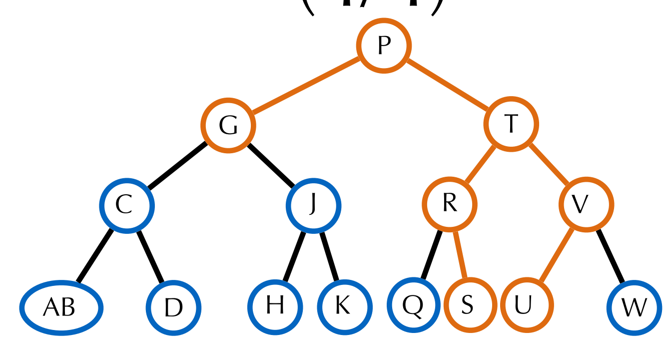
(2/4)



(3/4)

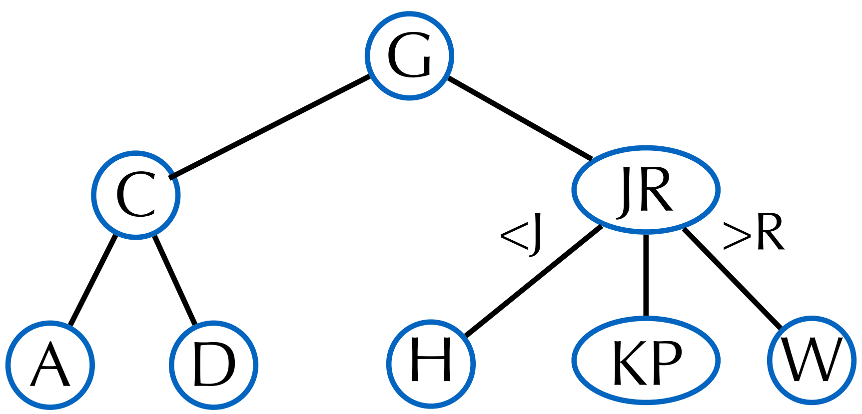


(4/4)

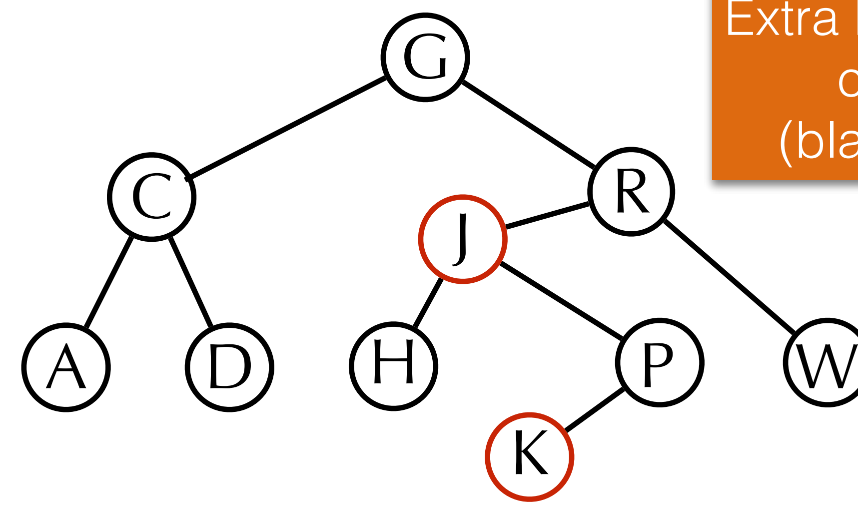


Worst-case number of nodes visited for find/insert: $\log_2 n$

(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.

`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

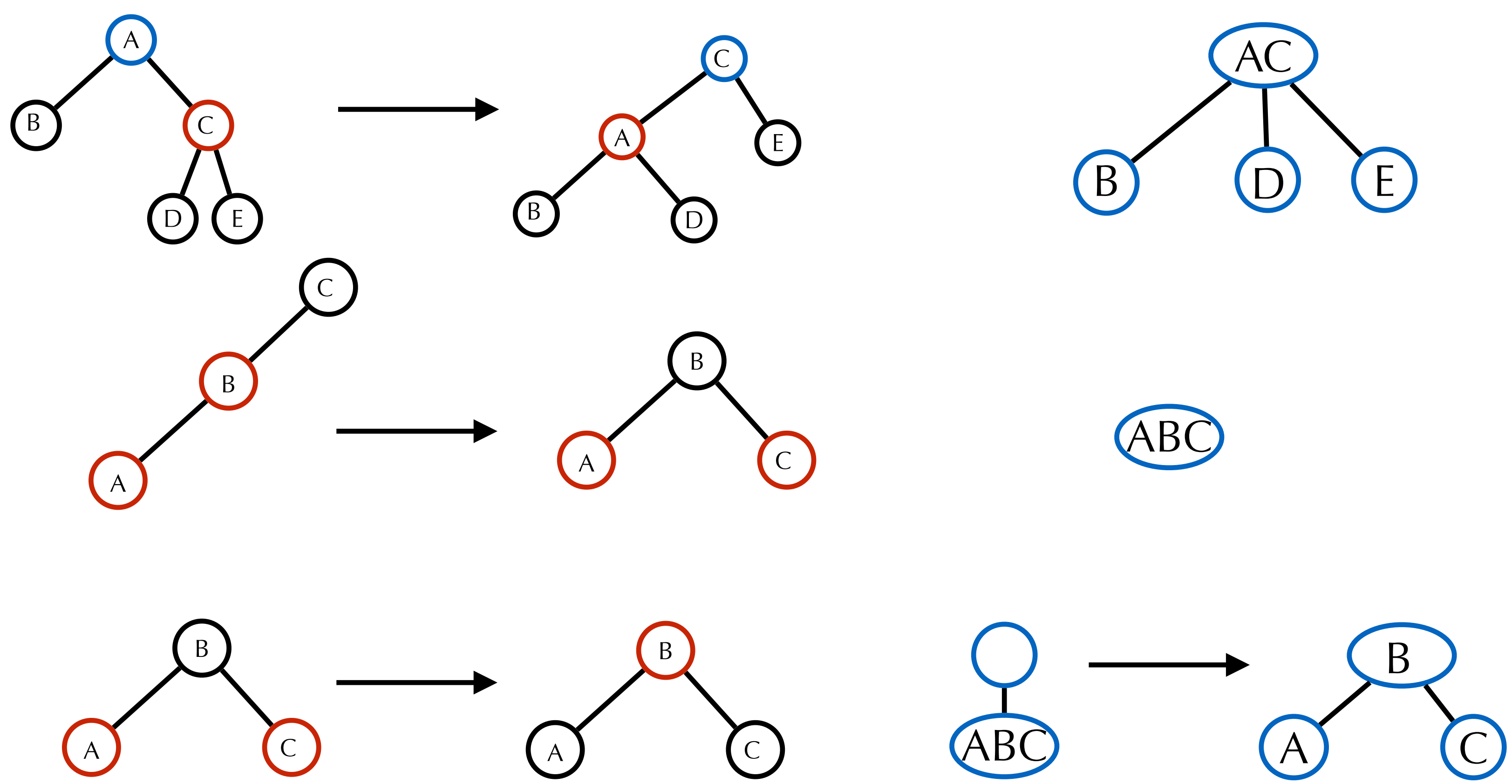
red-black trees

2-3 tree

rotateleft

rotateright

flipcolors



Check that order & black balance are preserved

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)

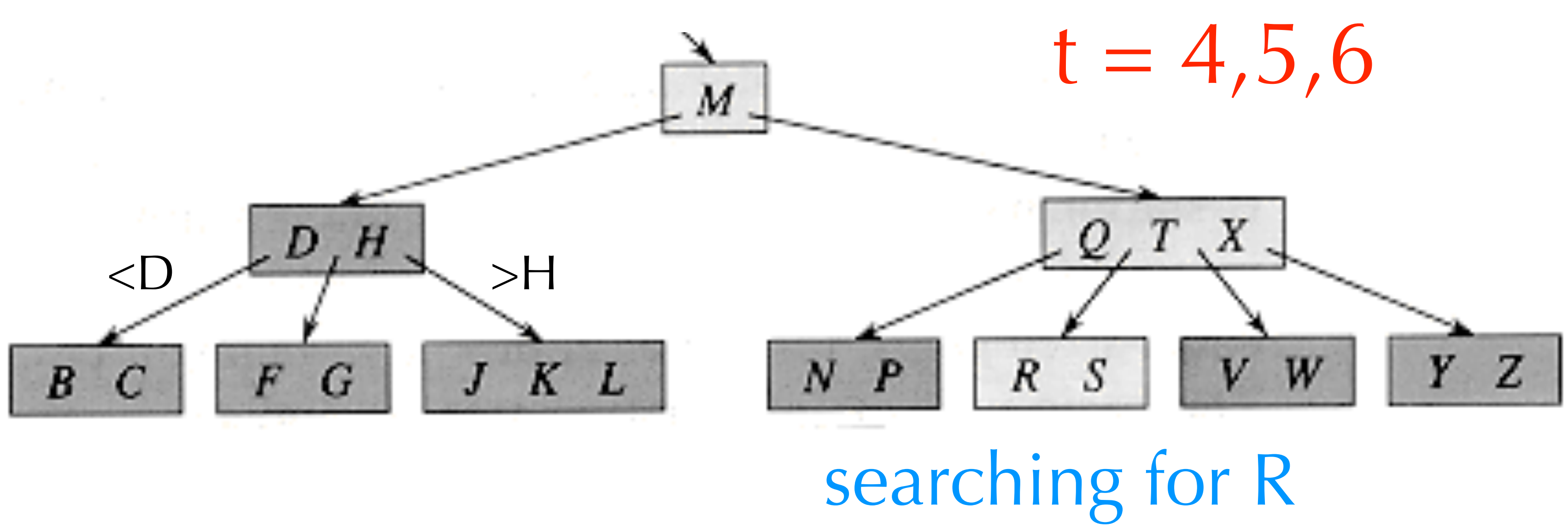
1. B Trees

B-Trees: Definition

Multi-way search trees commonly used in database systems (disk storage) - extension of 2-3 search trees

Definition (B-tree of order t ≥ 3):

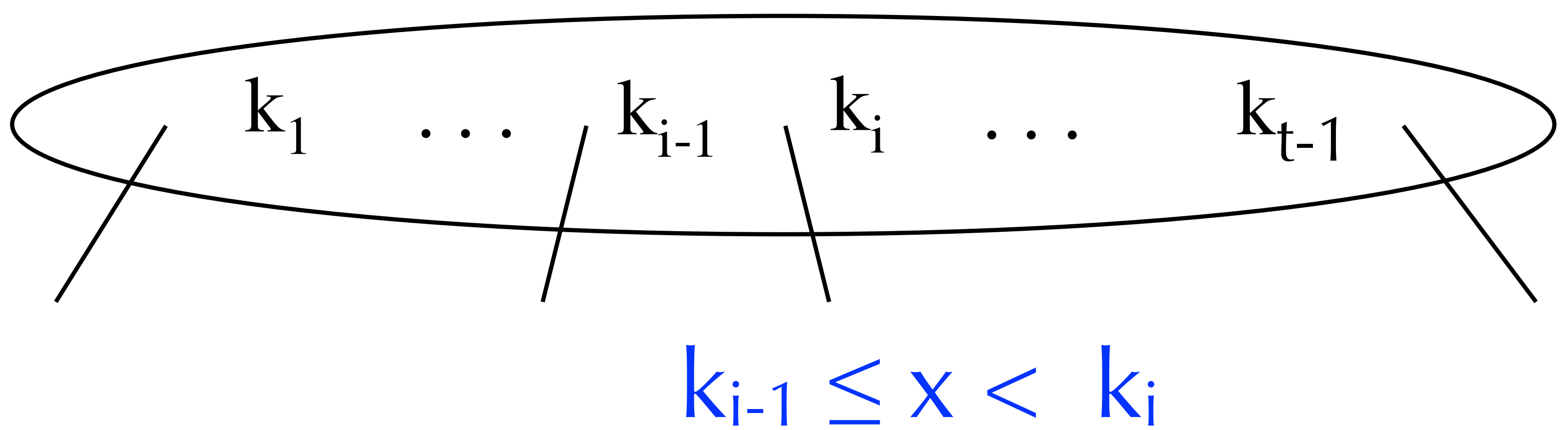
1. The root is either a leaf or has between 2 and t children;
2. Non-leaf nodes (except the root) have between ⌈t/2⌉ and t children;
3. All leaves are at the same depth. Each leaf stores between ⌈t/2⌉ and t items



B-Trees: Internal Nodes

Each internal node of a B-tree has:

- between ⌈t/2⌉ and t children
- up to t-1 keys $k_1 < k_2 < \ldots < k_{t-1}$



Keys are ordered so that:

$k_1 < k_2 < \ldots < k_{t-1}$

B-Trees: Find

For a B-tree of order t:

- each internal node has up to t-1 keys to search
- each internal node has between $\lceil t/2 \rceil$ and t children
- depth of B-tree storing N items: $O(log_{\lceil t/2 \rceil} N)$

B-Trees: Find

For a B-tree of order t:

- each internal node has up to t-1 keys to search
- each internal node has between $\lceil t/2 \rceil$ and t children
- depth of B-tree storing N items: $O(log_{\lceil t/2 \rceil} N)$

Complexity:

- $O(log_2\ t)$ to binary search which branch to take at a node
- Total time to find an item:

$$O(log_2\ t \cdot log_{\lceil t/2 \rceil} N) = O(log_2 N)$$

B-Trees: Find

```
class Node:

    def __init__(self,leaf=False):
        self.leaf = leaf
        self.keys = []
        self.children = []
```

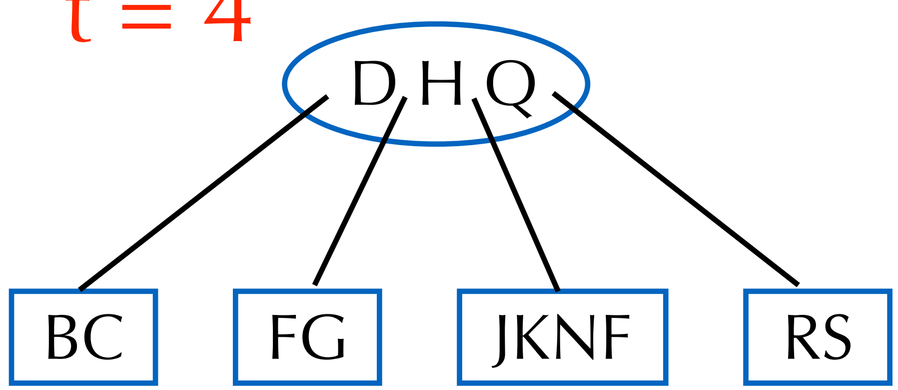
```
class BTree:

    def __init__(self, t):
        self.root = Node(True)
        self.t = t

    def search(self, k):
       x = self.root
       i = 0
       while i < len(x.keys) and k > x.keys[i][0]:
          i += 1
       if i < len(x.keys) and k == x.keys[i][0]:
          return (x, i)
       elif x.leaf:
          return None
       else:
          return self.search(k, x.child[i])
```

B-Trees: Insertion

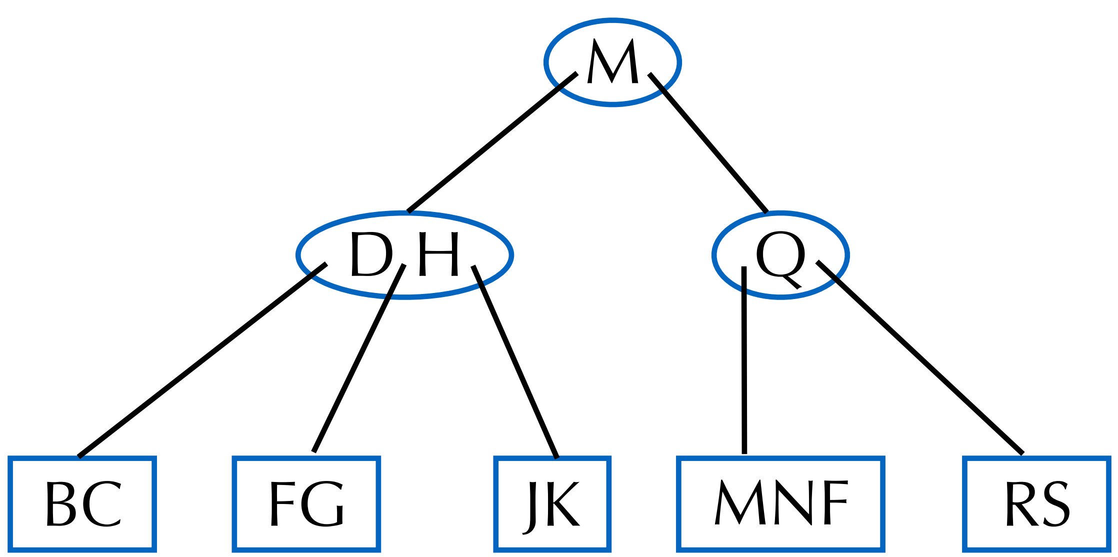
Insert x (similar to 2-3 search trees): Do a find on x and find appropriate leaf node

- if leaf node is not full, fill in empty slot with x
- if leaf node is full (has t items):
  - split into two nodes with $\lfloor (t+1)/2 \rfloor$ and $\lceil (t+1)/2 \rceil$ children
  - adjust parents up to the root node

t = 4



Insert M

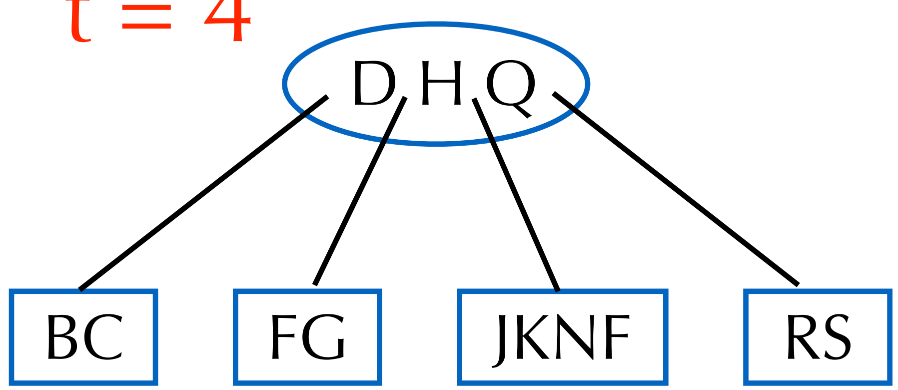


B-Trees: Insertion

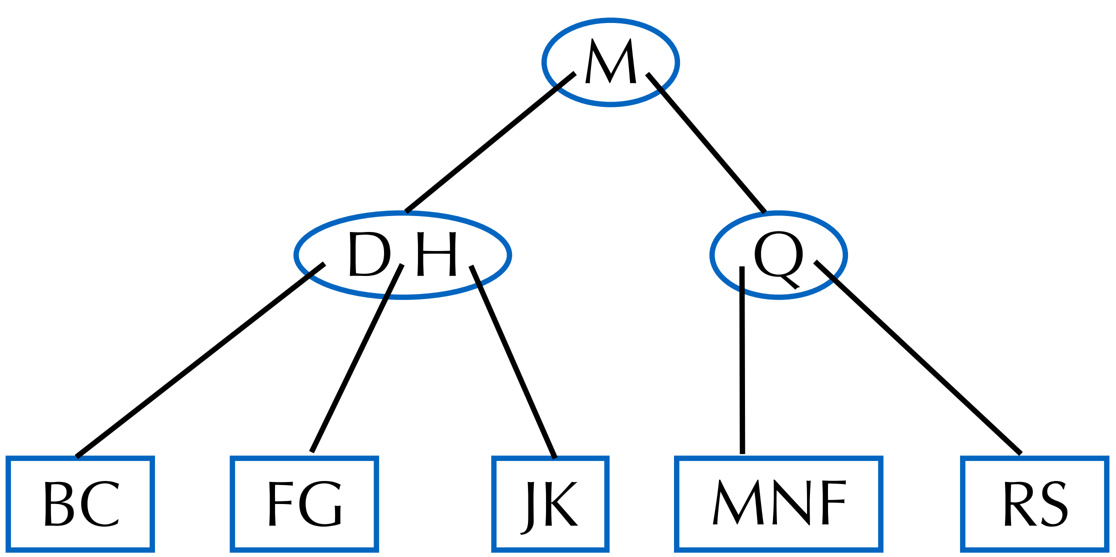
Insert x (similar to 2-3 search trees): Do a find on x and find appropriate leaf node

- if leaf node is not full, fill in empty slot with x
- if leaf node is full (has t items):
  - split into two nodes with $\lfloor (t+1)/2 \rfloor$ and $\lceil (t+1)/2 \rceil$ children
  - adjust parents up to the root node

t = 4



Insert M



$$O(t \cdot log_{\lceil t/2 \rceil} N) = O((t/log_2 t) \cdot log_2 N)$$

B-Trees: Insertion

```python
class Node:

    def __init__(self,leaf=False):
        self.leaf = leaf
        self.keys = []
        self.children = []
```

```python
def insert(self, k):
  root = self.root
  if len(root.keys) == self.t - 1:
    temp = Node()
    self.root = temp
    temp.child.insert(0, root)
    self.split_child(temp, 0)
    self.insert_non_full(temp, k)
  else:
    self.insert_non_full(root, k)
```

```python
def insert_non_full(self, x, k):
  i = len(x.keys) - 1
  if x.leaf:
    x.keys.append((None, None))
    while i >= 0 and k[0] < x.keys[i][0]:
      x.keys[i + 1] = x.keys[i]
      i -= 1
    x.keys[i + 1] = k
  else:
    while i >= 0 and k[0] < x.keys[i][0]:
      i -= 1
    i += 1
    if len(x.child[i].keys) == self.t - 1:
      self.split_child(x, i)
      if k[0] > x.keys[i][0]:
        i += 1
      self.insert_non_full(x.child[i], k)
```

B-Trees: Insertion

```
class Node:

    def __init__(self,leaf=False):
        self.leaf = leaf
        self.keys = []
        self.children = []
```

```
def split_child(self, x, i):
  t = self.t
  y = x.child[i]
  z = Node(y.leaf)
  x.child.insert(i + 1, z)
  x.keys.insert(i, y.keys[ceil(t/2)])
  z.keys = y.keys[ceil(t/2): t]
  y.keys = y.keys[0: ceil(t/2)]
  if not y.leaf:
    z.child = y.child[ceil(t/2): t]
    y.child = y.child[0: ceil(t/2)]
```

```
def insert(self, k):
  root = self.root
  if len(root.keys) == self.t - 1:
    temp = Node()
    self.root = temp
    temp.child.insert(0, root)
    self.split_child(temp, 0)
    self.insert_non_full(temp, k)
  else:
    self.insert_non_full(root, k)
```

B-Trees: Order Values

Tree in internal memory: t = 3 or 4

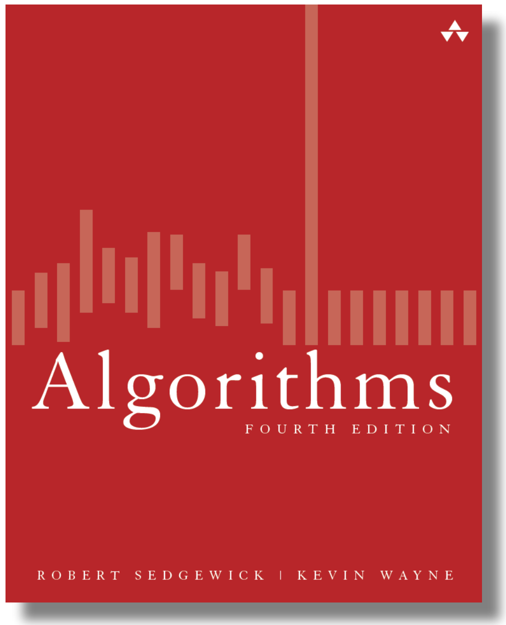
Tree on disk: t = 32 to 256 (interior and leaf nodes fit on 1 disk block)

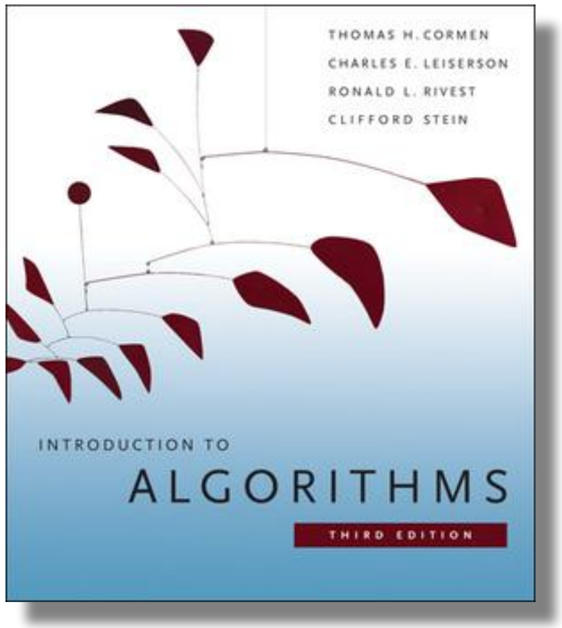
- depth = 2 or 3 → fast access to data in databases

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

NO Assignment

Next tutorial: K-Dimensional Trees

Next week: String Algorithms 1 (Video)

Feedback

Moodle

Questions: constantin.enea@polytechnique.edu