# CSE202
# Design and Analysis of Algorithms

## *Week 9 — 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

# 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

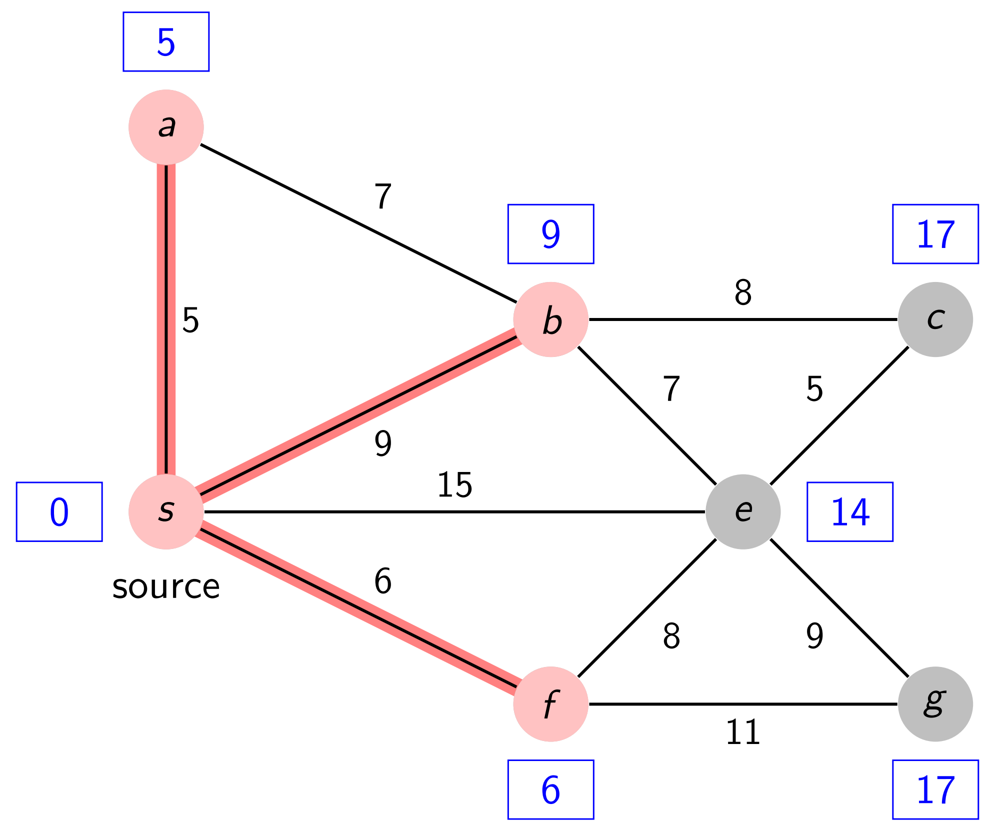
**Def.** All leaves in the same one or two levels

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

| $n$ | $\log_2 n$ |
|---|---|
| $10^6$ | $\approx 20$ |
| $10^9$ | $\approx 30$ |
| $10^{12}$ | $\approx 40$ |

# 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

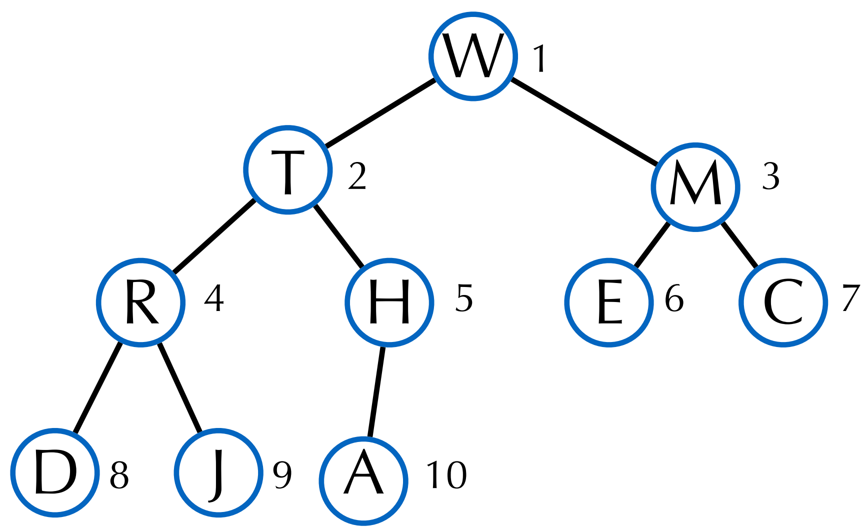
**for all** neighbours w of v

insert ((v,w),d(s,v)+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 | | | | | | |

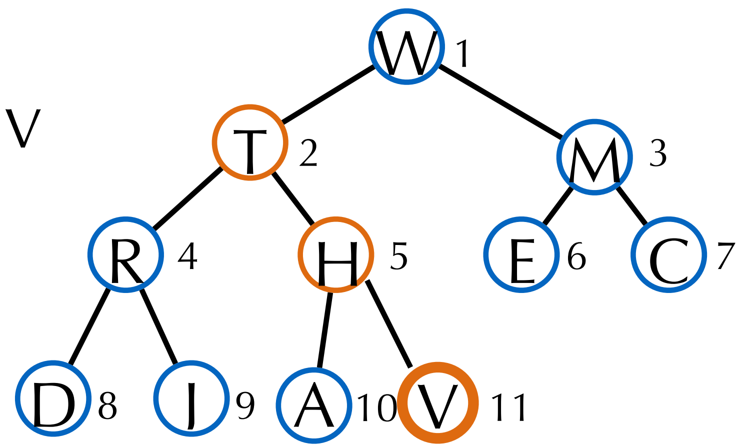
0 1 16

# Basic Operations

# Basic Operations

## Insert & fixup

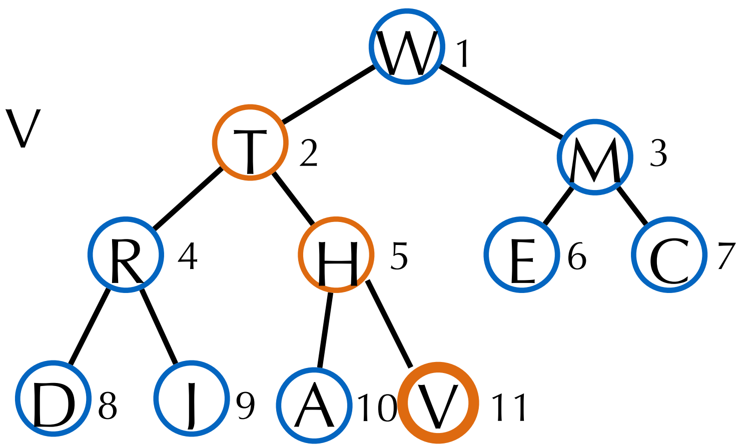
insert V



# Basic Operations

## Insert & fixup

insert V

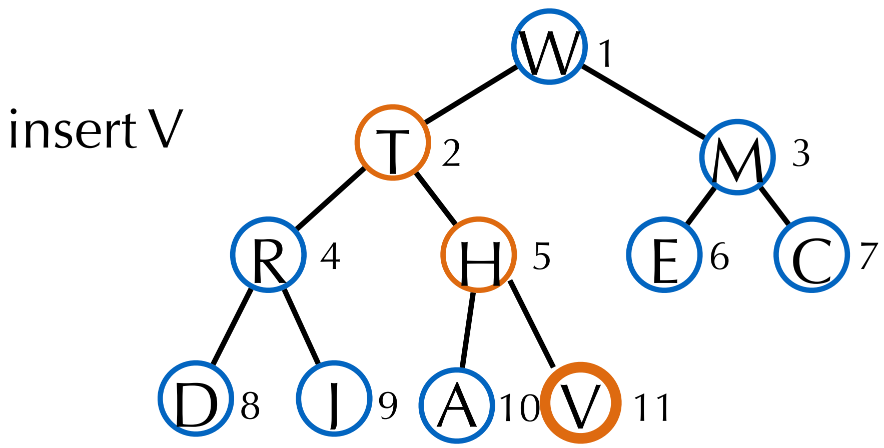



```
def insert(self,key):
    self.size += 1
    self.PQ[self.size]=key
    self.fixup(self.size)
```

```
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)
```

# Basic Operations

Insert & fixup




$\leq \log_2 n$ comparisons

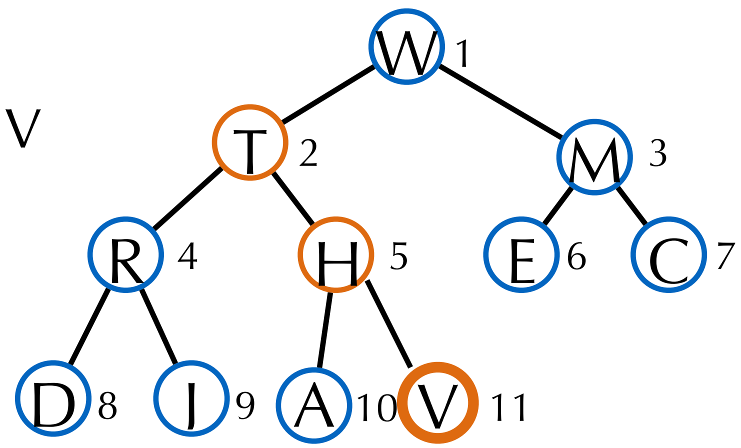
```
def insert(self,key):
    self.size += 1
    self.PQ[self.size]=key
    self.fixup(self.size)
```

```
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)
```

# Basic Operations

## Insert & fixup

insert V



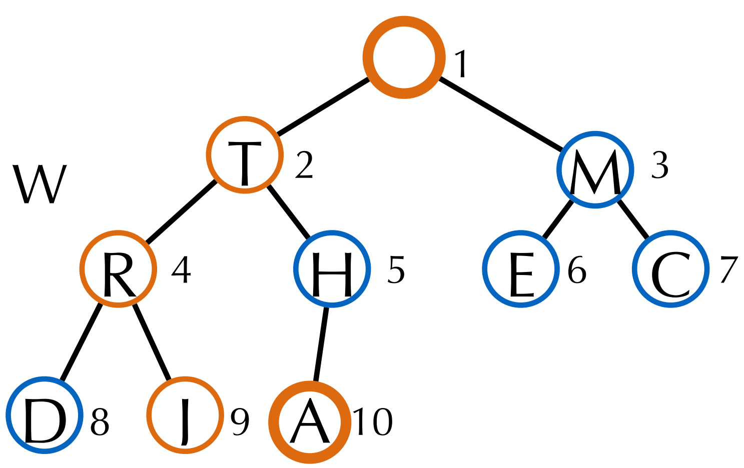

$\leq \log_2 n$ comparisons

```
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)
```

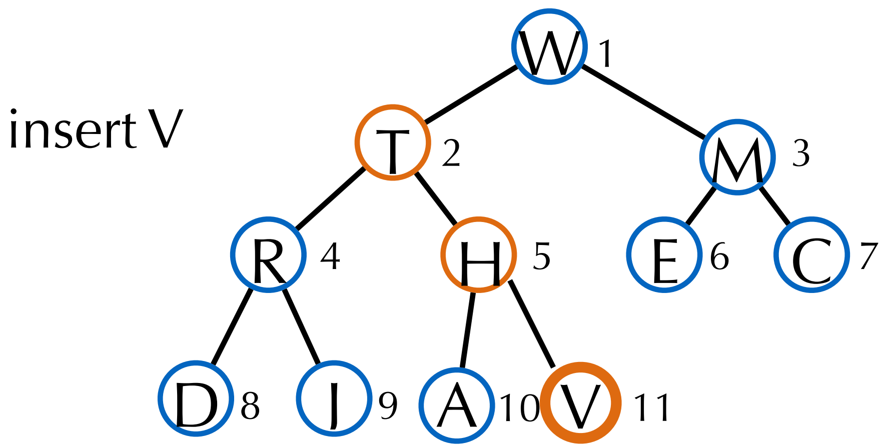
## Deletemax & fixdown

delete W



# Basic Operations

## Insert & fixup

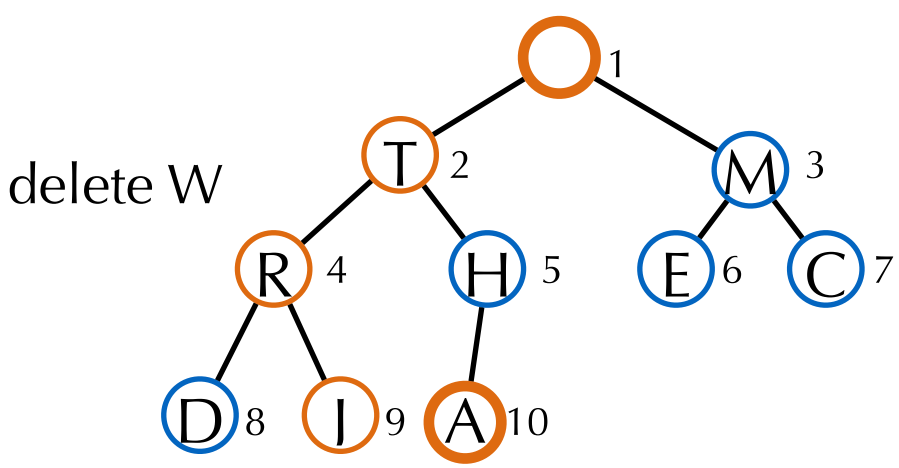



$\leq \log_2 n$ comparisons

```
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)
```

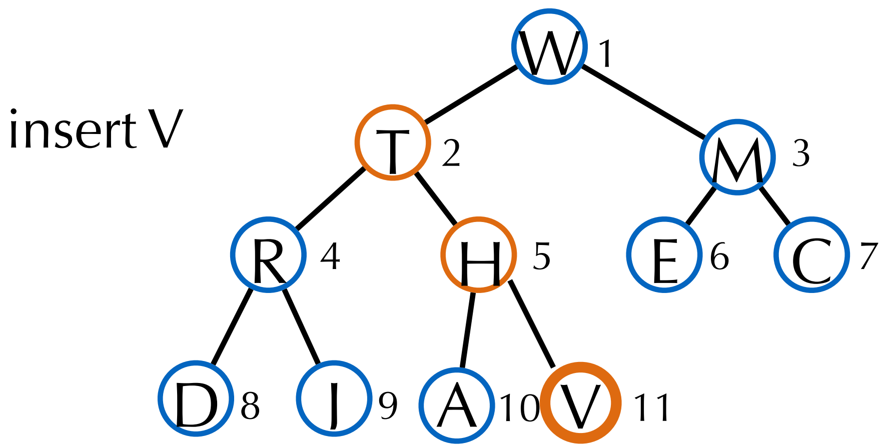
## Deletemax & fixdown




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

# Basic Operations

## Insert & fixup

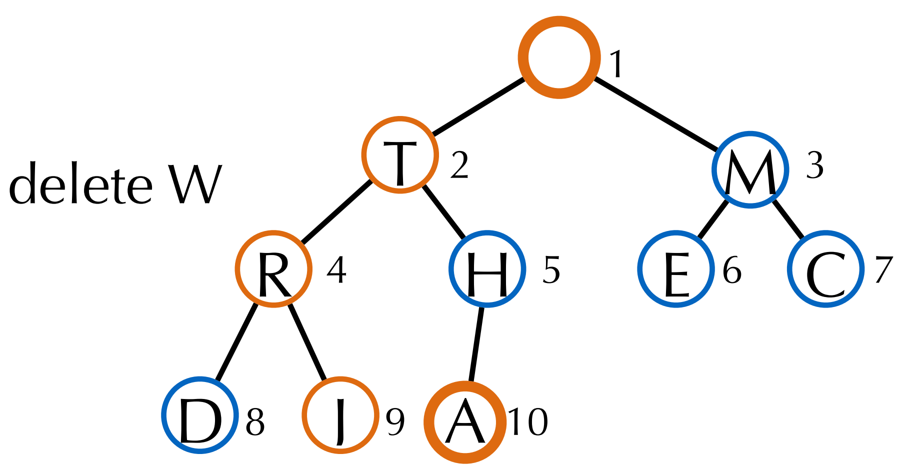



$\leq \log_2 n$ comparisons

```
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



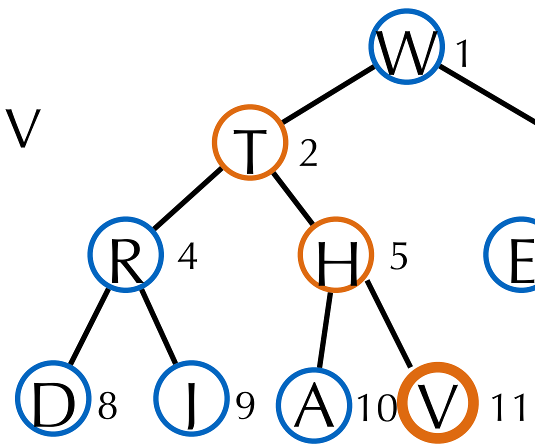

```
def deletemax(self):
    self.PQ[1] = self.PQ[self.size]
    self.size -= 1
    self.fixdown(1)
```

```
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)
```

# Basic Operations

## Insert & fixup

insert V



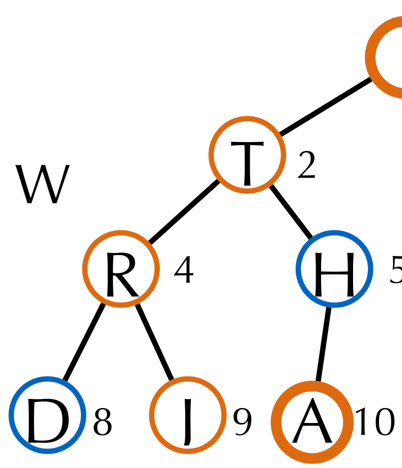

$\leq \log_2 n$ comparisons

```
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

delete W

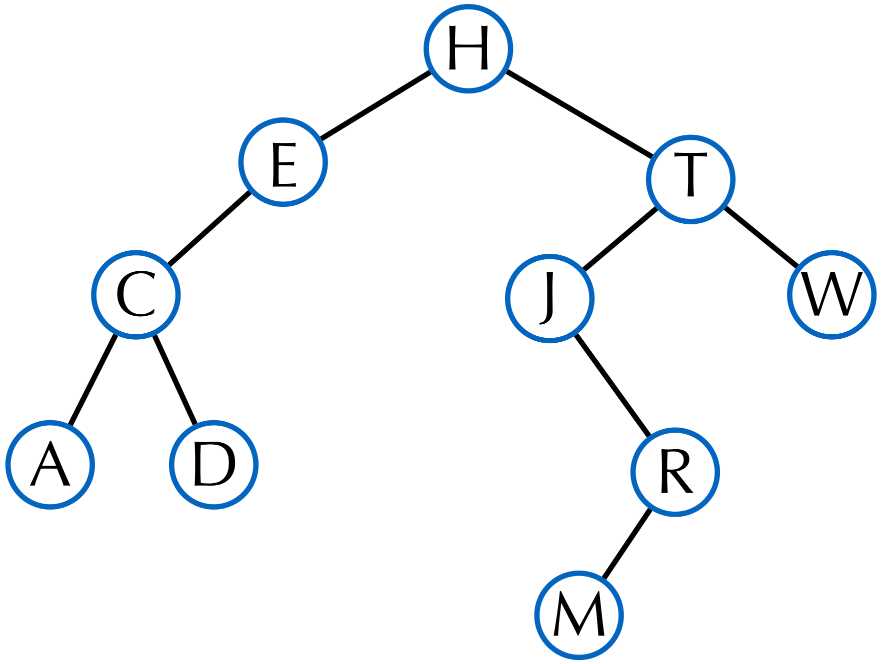



$\leq 2\log_2 n$ comparisons

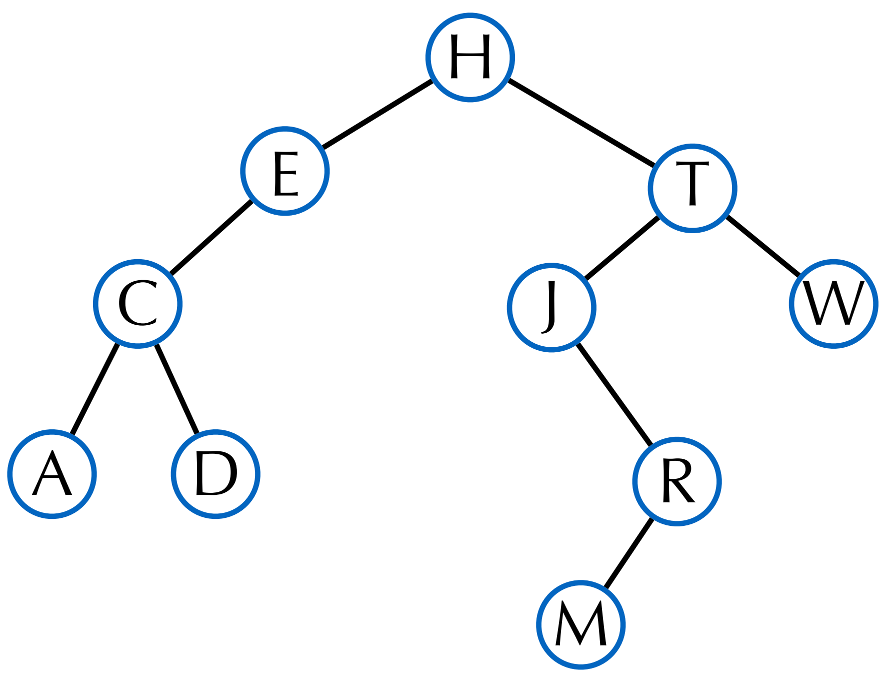
```
def deletemax(self):
    self.PQ[1] = self.PQ[self.size]
    self.size -= 1
    self.fixdown(1)
```

```
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)



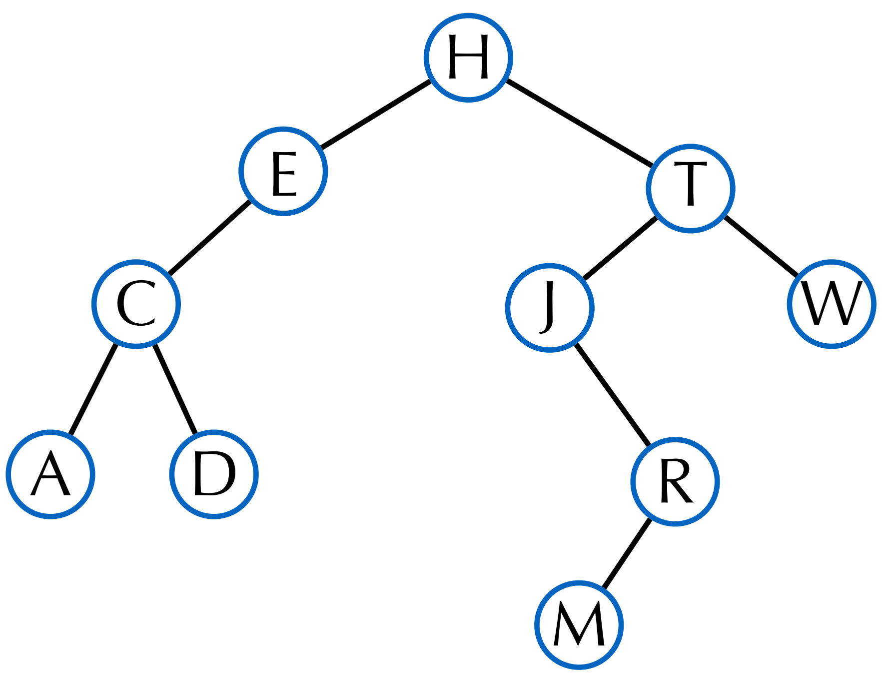

Smaller elements to the left, larger elements to the right

```python
class Node:

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

```python
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

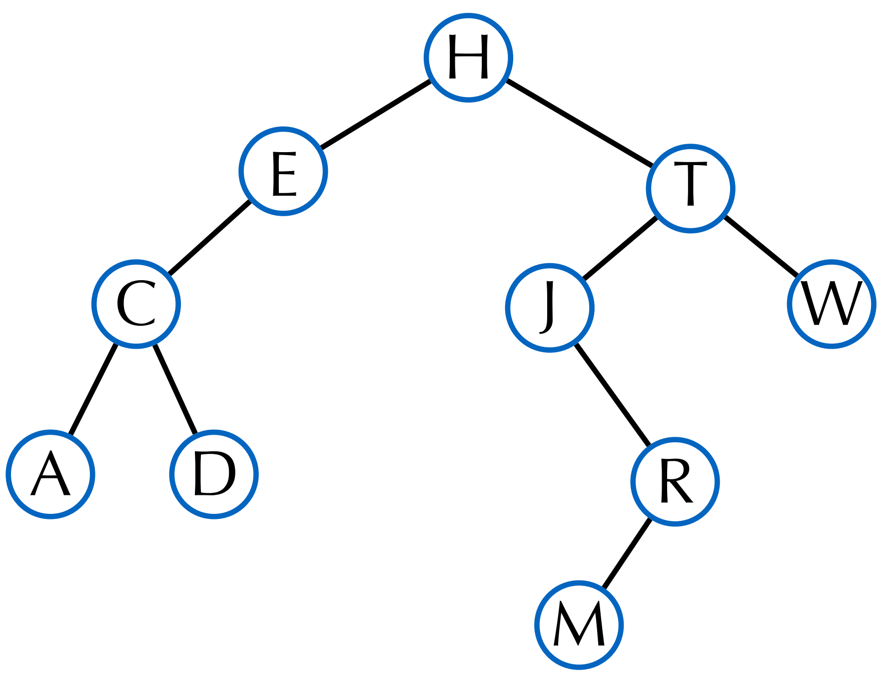
```
def _find(self,node,key):
    if node is None: return False
    if node.key > key: return self._find(node.left,key)
    if node.key < key: return self._find(node.right,key)
    return True
```




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

# 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
```

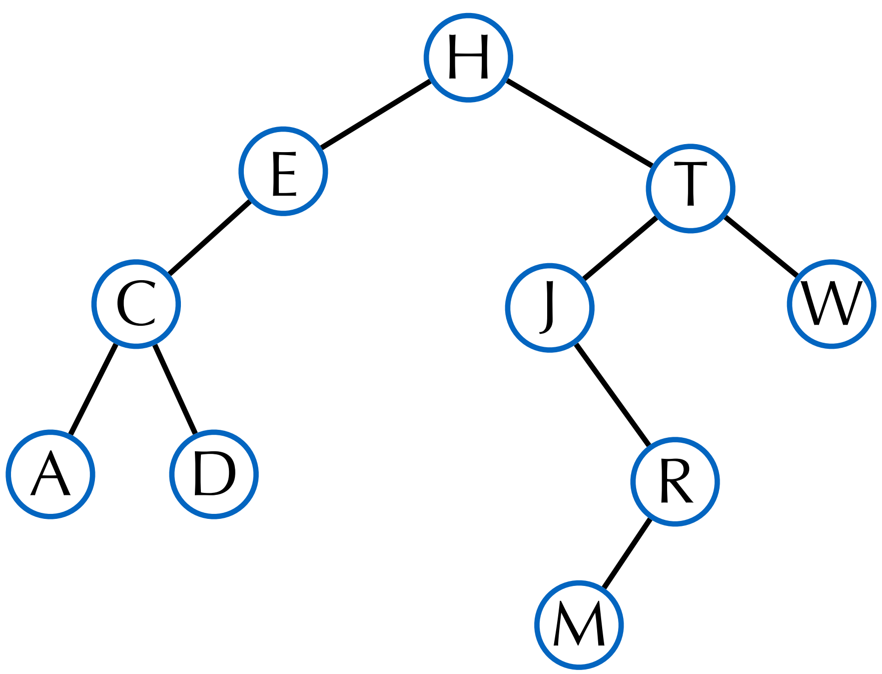



```
def _insert(self,node,key):
    if node is None: return Node(key)
    if node.key > key:
        node.left = self._insert(node.left,key)
    elif node.key < key:
        node.right = self._insert(node.right,key)
    return node
```

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

# 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
```

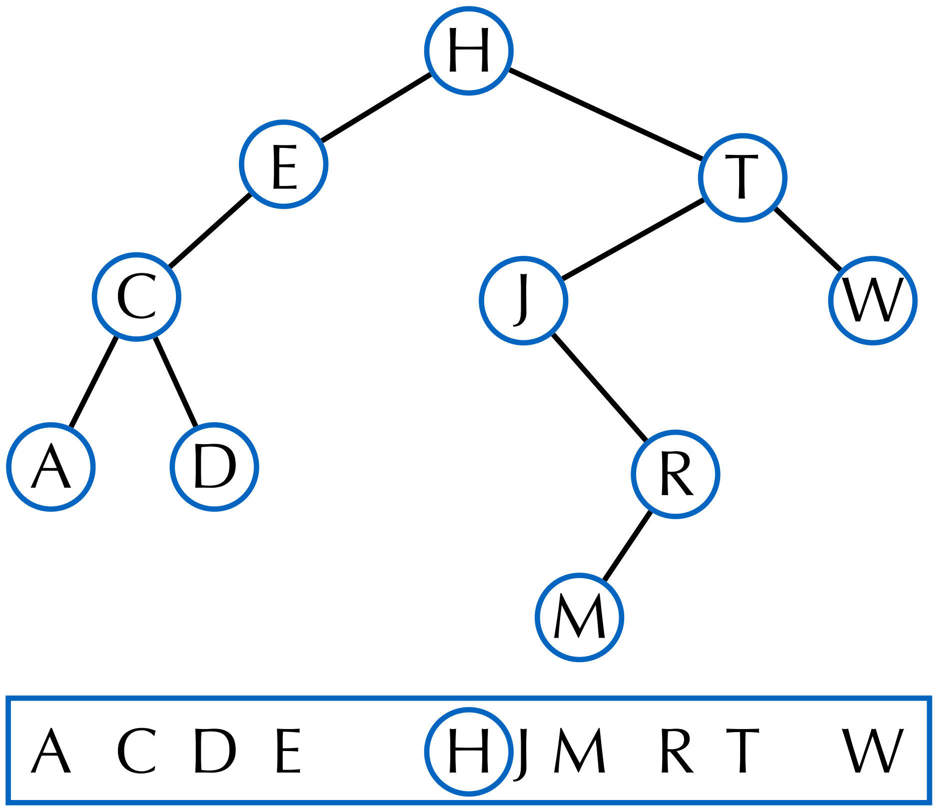



```
def _insert(self,node,key):
    if node is None: return Node(key)
    if node.key > key:
        node.left = self._insert(node.left,key)
    elif node.key < key:
        node.right = self._insert(node.right,key)
    return node
```

Delete slightly more complicated (CSE102)

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

# Average-Case Analysis




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

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

# Average-Case Analysis

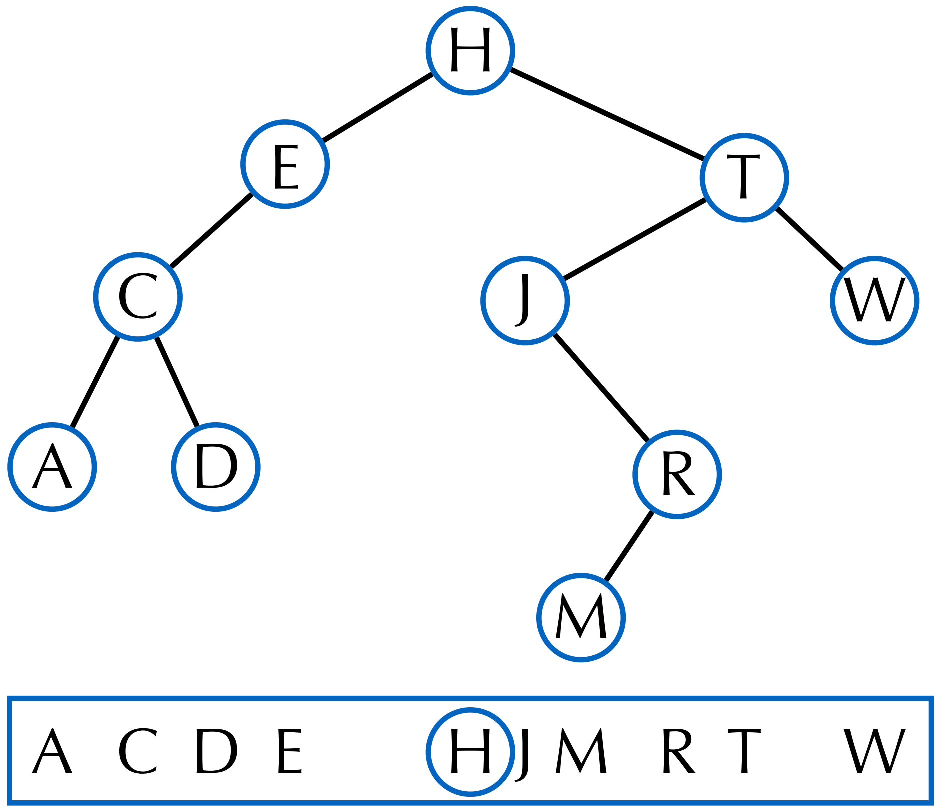
Internal path length:

$P_n$ := 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

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

# Average-Case Analysis

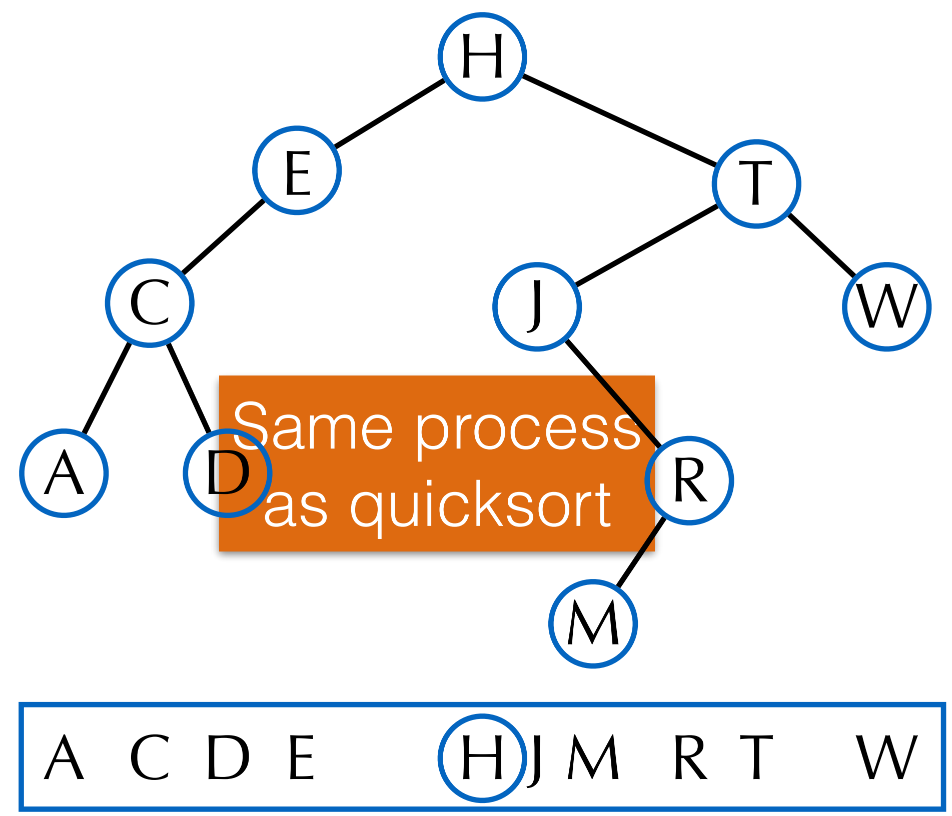
Internal path length:

$P_n :=$ 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

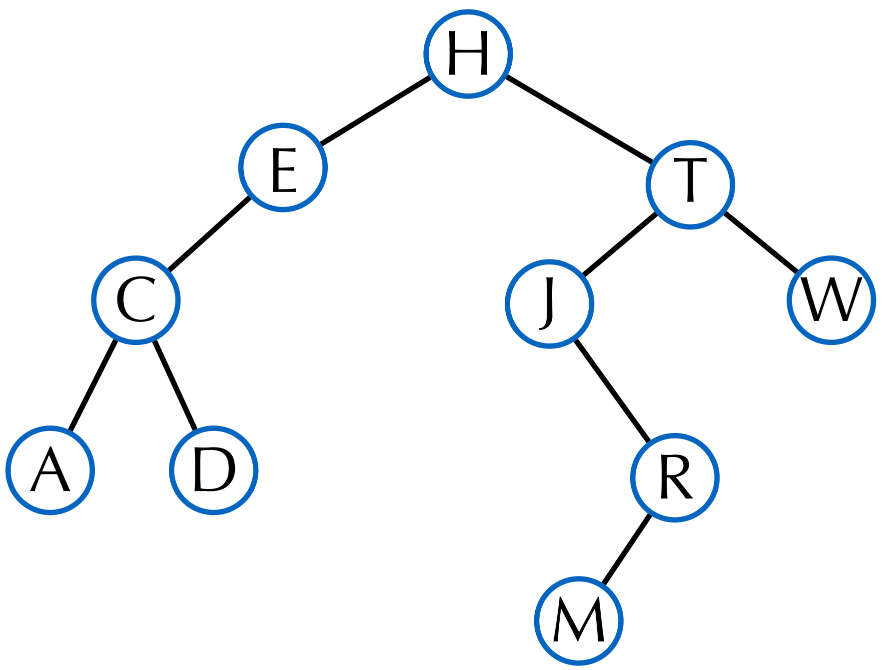
$$1.39 \log_2 n \; + \; O(1)$$

$$P_0 = P_1 = 0$$

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

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
```

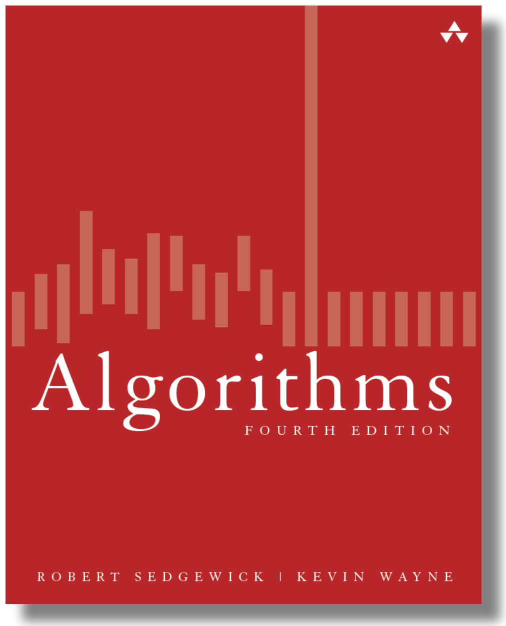
All these operations have cost bounded by the height, which is logarithmic on average.

Generalizes to higher dimensions (quadtrees).

# 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: Union-find

Next tutorial: Union-find

# Feedback

Moodle

Questions: constantin.enea@polytechnique.edu