DSC40B: Theoretical Foundations of Data Science II

Lecture 8: *Binary search tree*

Instructor: Yusu Wang

(Dynamic) Set operations

- Imagine you are maintaining a database indexed by some keys (real values), and you hope to support the following operations:

First approach: sort the array of keys

- Search — $\Theta(\lg n)$
- Maximum — $\Theta(1)$
- Minimum — $\Theta(1)$
- Successor — $\Theta(1)$
- Predecessor — $\Theta(1)$

Static operations!

Using a sorted array can handle all static operations efficiently

- Insert
- Delete — $\Theta(n)$
- Extract-Max — $\Theta(n)$
- Increase-key

Dynamic operations!

How to have a good data structure so we can support all these operations efficiently?

Today

- Binary search tree
  - support all the operations from previous slide
    - in time proportional to height of tree
- (Review): how to implement key operations, and time complexity
  - search, insert (and delete)
- Extension to balanced binary search tree
- *Select* query: augmenting data structure
  - median, order statistics

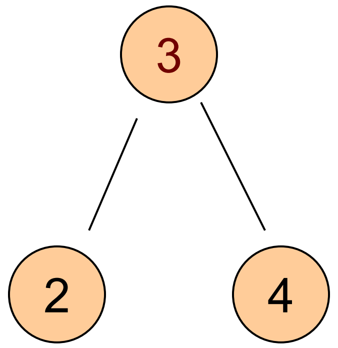
Part A:
What is binary search tree?

First: Binary tree

- A binary tree is a rooted tree
  - where each node has at most 2 children

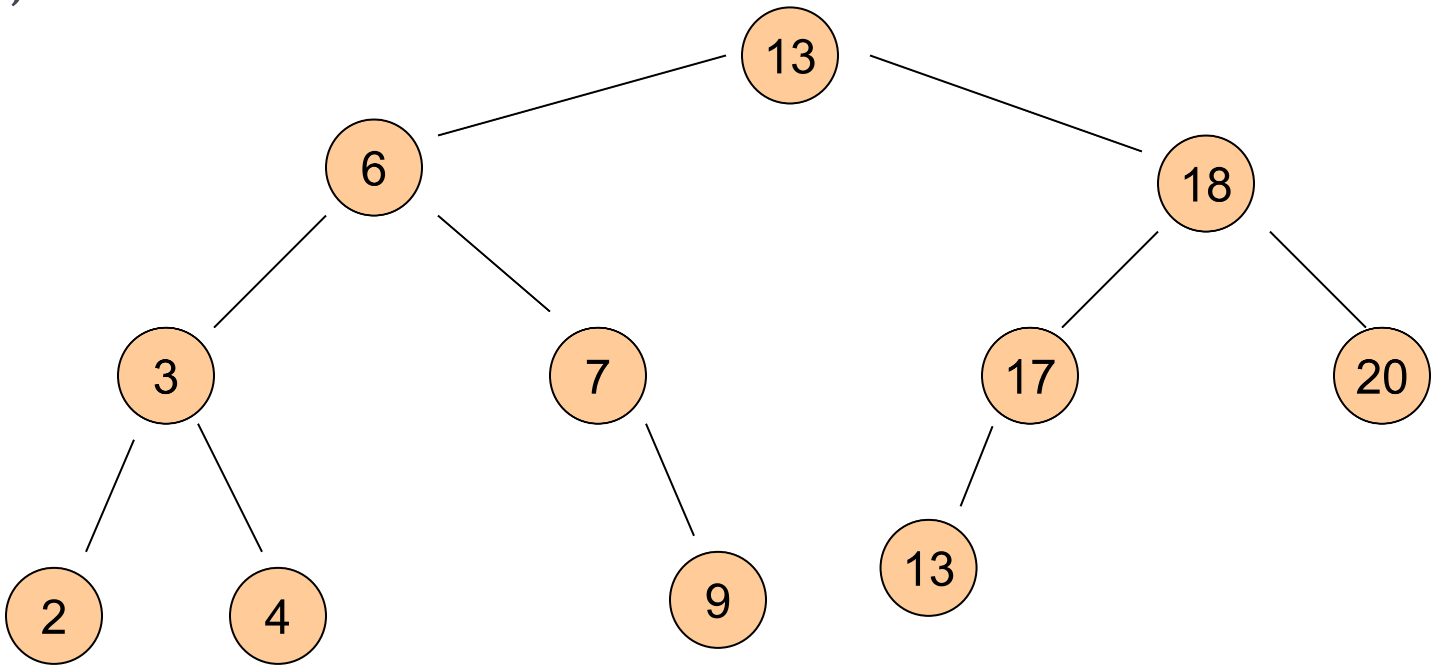
- Represented by a linked data structure

- Each node contains at least fields:
  - *Key*
  - *Left*
  - *Right*
  - *Parent*



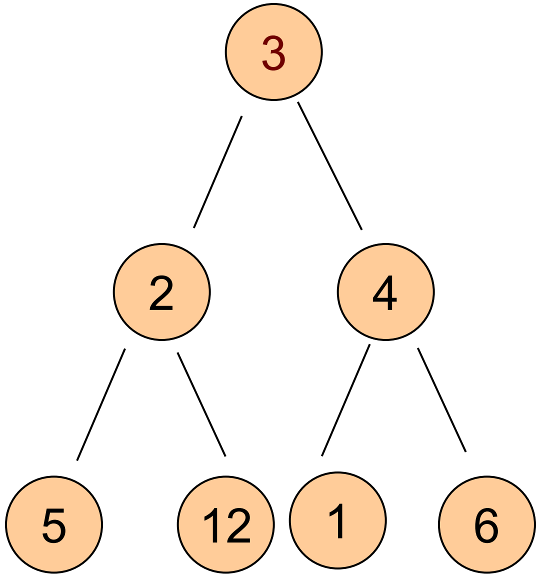
Example

- From root, following left pointers, we will visit
  - $13, 6, 3, 2, Nil$



Binary tree

- A binary tree is a rooted tree where
  - each node has at most 2 children
- A node is the root of the tree
  - if its parent is Nil
- A node is a leaf
  - if both children are Nil
- Left sub-tree, right sub-tree



- A complete binary tree is a binary tree
  - where each node has two children other than leaves
  - and each level (except possibly last level) is filled, and all nodes are as left as possible.

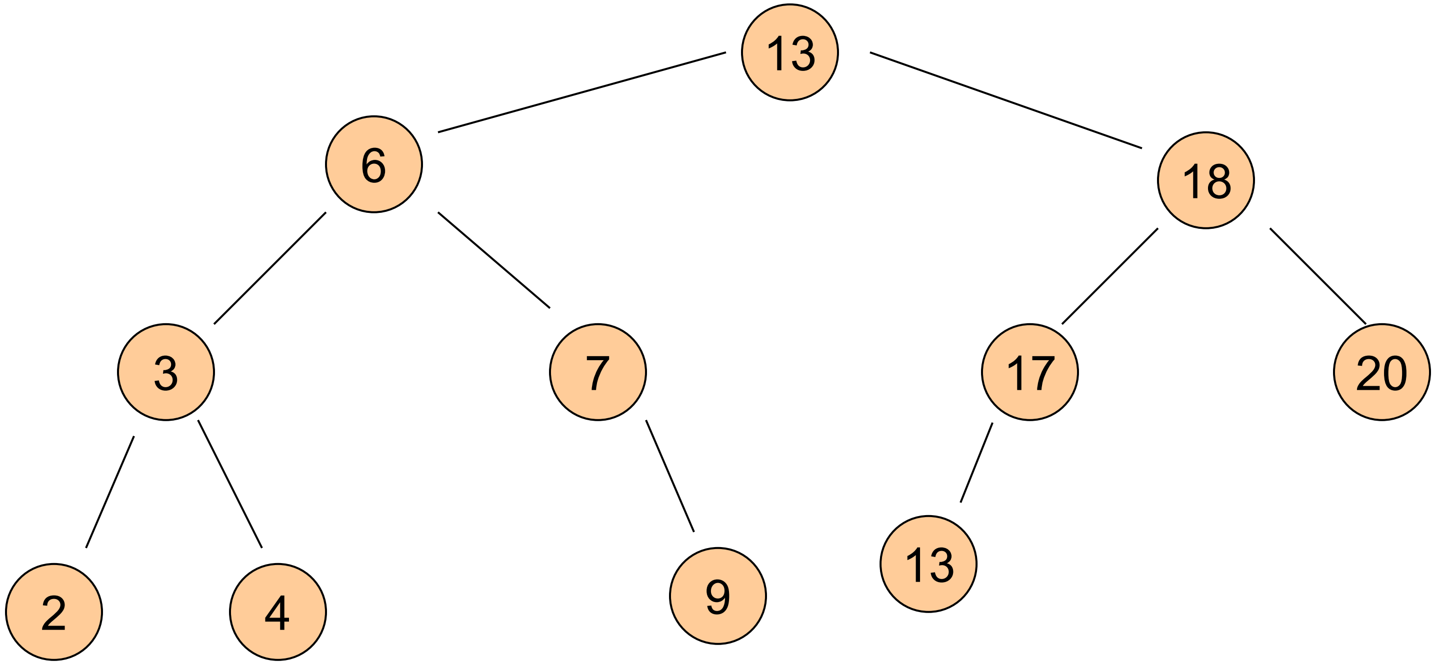
Binary search tree (BST)

- Binary-search-tree property
  - For any node $x \in T$,
    - $x.Key \geq y.Key$ if y is in the left subtree of x; and
    - $x.Key \leq y.Key$ if y is in the right subtree of x

- A binary tree T is a binary search tree (BST) if
  - it satisfies the binary search tree property

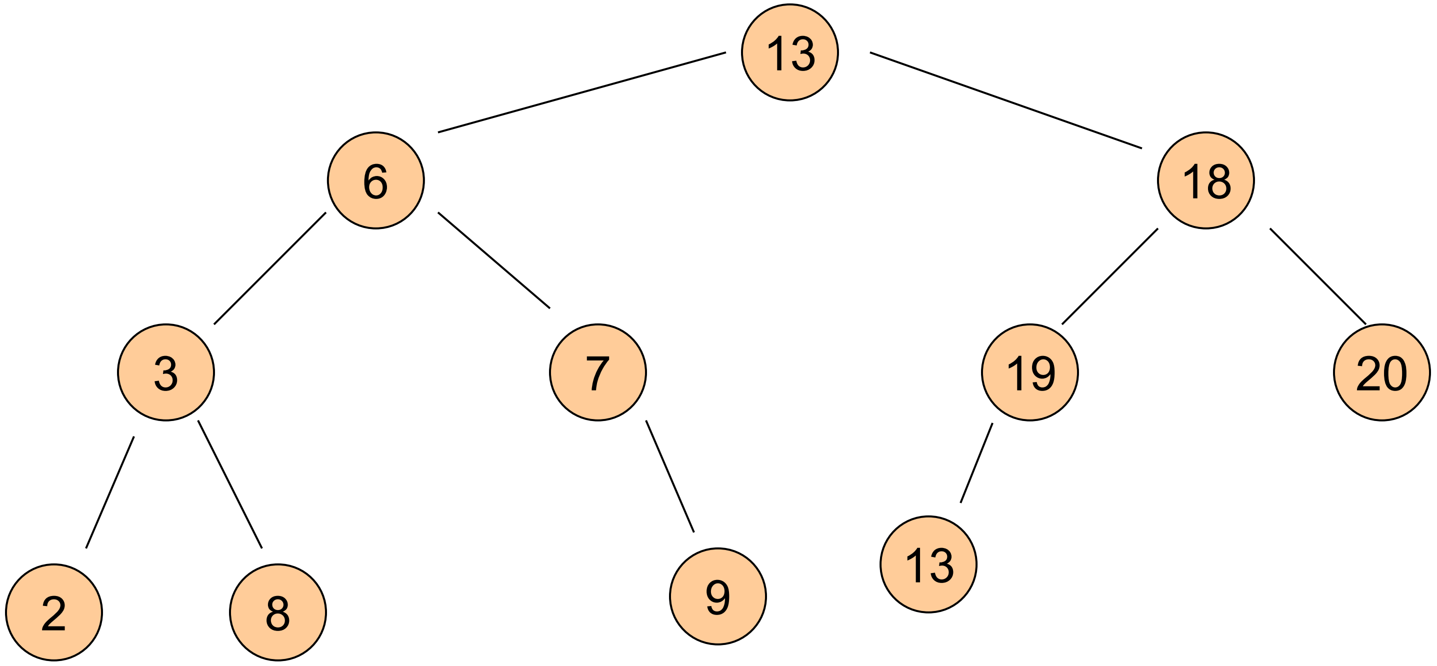
Example

- A valid BST



Example

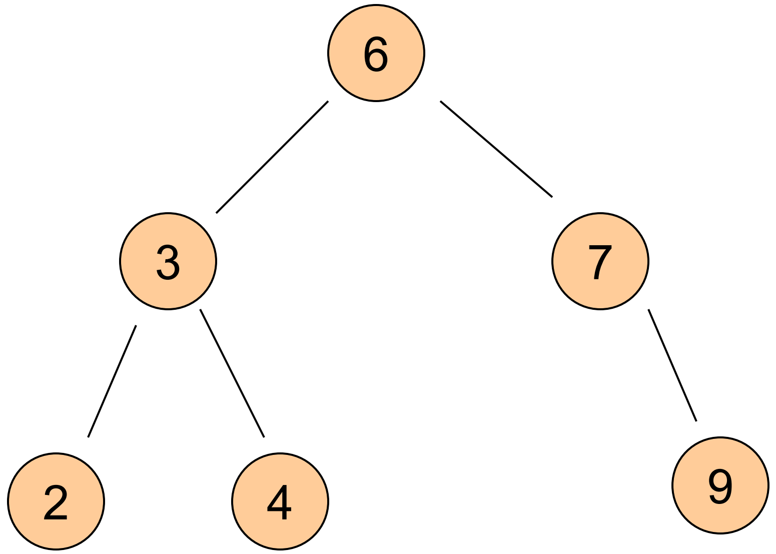
- ?



Properties

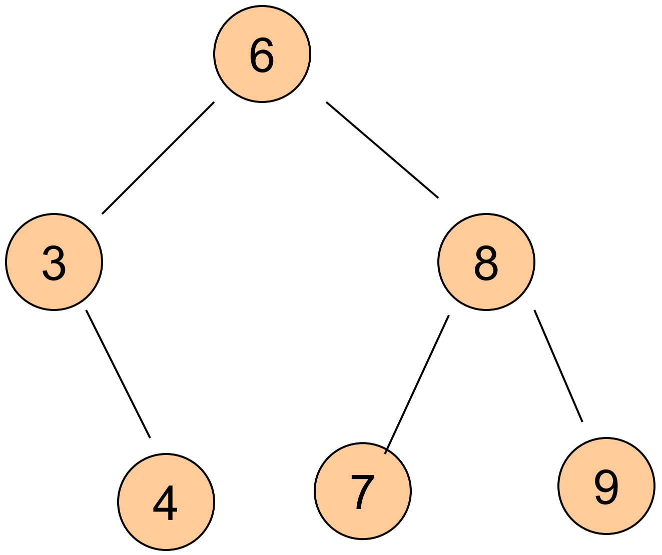
- Given the same set of elements
  - there are many possible BSTs over them

- Minimum?
  - Does it have to be a leaf?



Properties

- Given the same set of elements
  - there are many possible BSTs over them
- Minimum?
  - Does it have to be a leaf?
- Maximum?



- Given n nodes,
  - Tallest possible BST tree has height $h =$ n
  - Shortest possible BST tree has height $h =$ $\log_2 n = \Theta(\lg n)$

Part B:
Operations in BST

Search operation

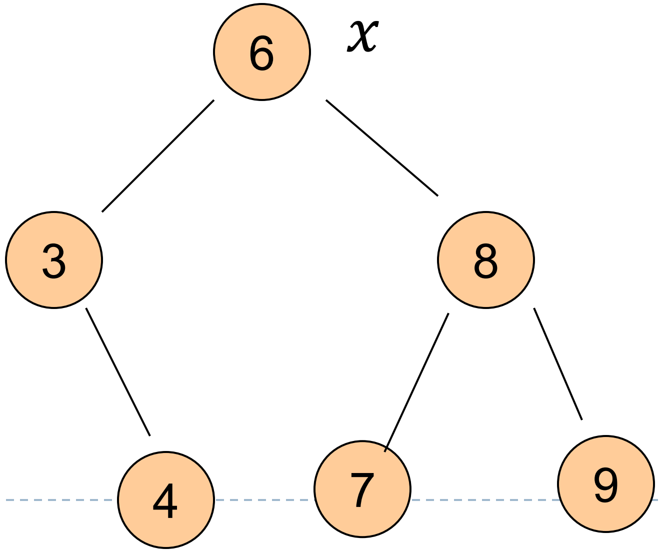
- A BST T with n nodes can be viewed as a way to store n keys in a smart way, so that queries among these keys become easy.

- Tree-search(x, k)
  - Input: given a tree node x and a query key k
  - Output: search whether k is in the tree rooted at x
    - if it is in, return a node y s.t. $y.key = k$
    - otherwise, returns NIL

Tree-search$(x, 8)$

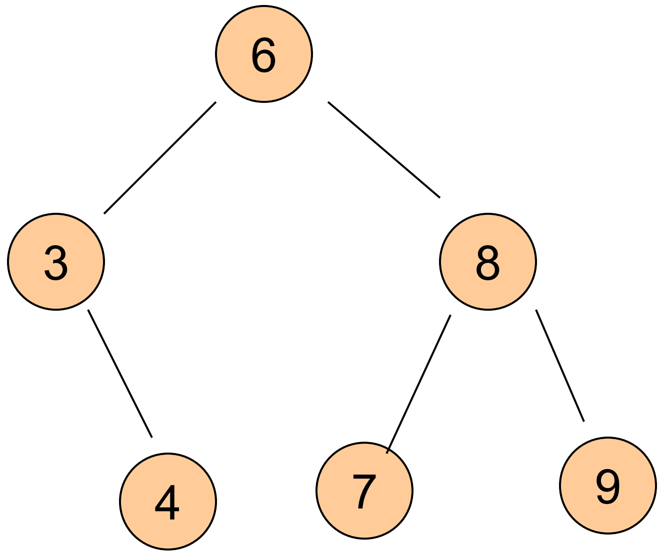
Tree-search$(x, 4)$

Tree-search$(x, 5)$



Tree-search algorithm, recursive version

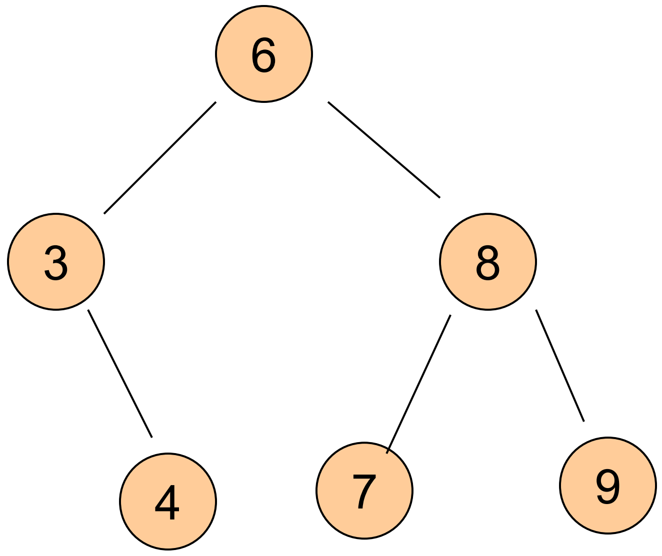
```
Tree-search ( x, k)
      if x = Nil or k = x.key
        then return x
      if k < x.key
        then return Tree-search( x.left, k )
        else  return Tree-search( x.right, k )
```



- Given an input tree T and a key k
  - we will start by calling Tree-search(T.root, k)

Tree-search algorithm, recursive version

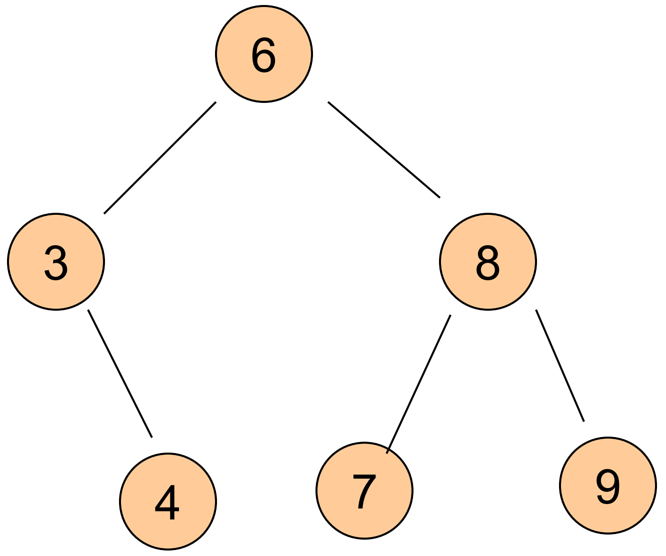
```
Tree-search ( x, k)
    if x = Nil or k = x.key
      then return x
    if k < x.key
      then return Tree-search( x.left, k )
      else  return Tree-search( x.right, k )
```



- Time complexity analysis
  - let $T(n)$ denote the worst case time complexity of procedure Tree-search() on any tree of n nodes

Tree-search algorithm, recursive version

```
Tree-search ( x, k)
      if x = Nil or k = x.key
        then return x
      if k < x.key
         then return Tree-search( x.left, k )
         else  return Tree-search( x.right, k )
```



- Time complexity analysis
  - other than recursive call, $\Theta(1)$ within each Tree-search call
  - thus, $T(n)$ is proportional to the number of nodes x we will call Tree-search on
  - $T(n) = \Theta(\text{tree-height}) = O(n)$

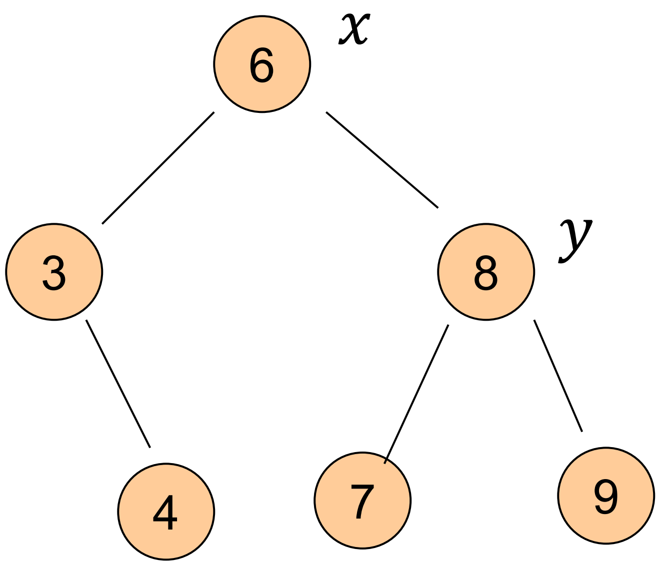
Tree-search: iterative version

procedure `IterativeTreeSearch`(x,K)

while (x = **NIL**) **and** (K $\neq x$.key) **do**
    if (K $\leq x$.key) **then**
        $x \leftarrow x$.left;
    else
        $x \leftarrow x$.right;
    end
end
return (x);

Minimum / Maximum

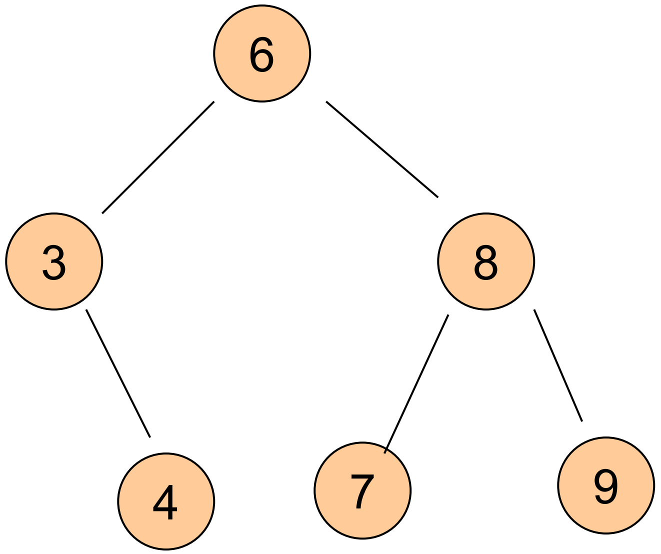
- Tree-minimum(x)
  - Input: a node x of a BST T
  - Output: return the node containing minimum key in the subtree rooted at x



Minimum / Maximum

- Tree-minimum(x)
  - Input: a node x of a BST T
  - Output: return the node containing a minimum key in the subtree rooted at x

```
Tree-minimum(x)
        while ( x.left ≠ Nil)
          do   x = x.left;
        return x;
```

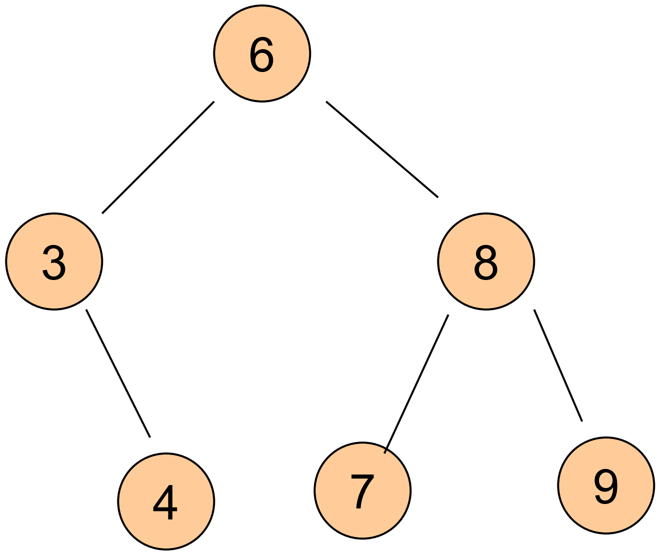


- Time complexity
  - $T(n) = \Theta(h)$ where h is height of input tree

Minimum / Maximum

- Tree-maximum(x)
  - Input: a node x of a BST T
  - Output: return the node containing a maximum key in the subtree rooted at x

```
Tree-maximum(x)
        while ( x.right ≠ Nil)
          do   x = x.right;
        return x;
```

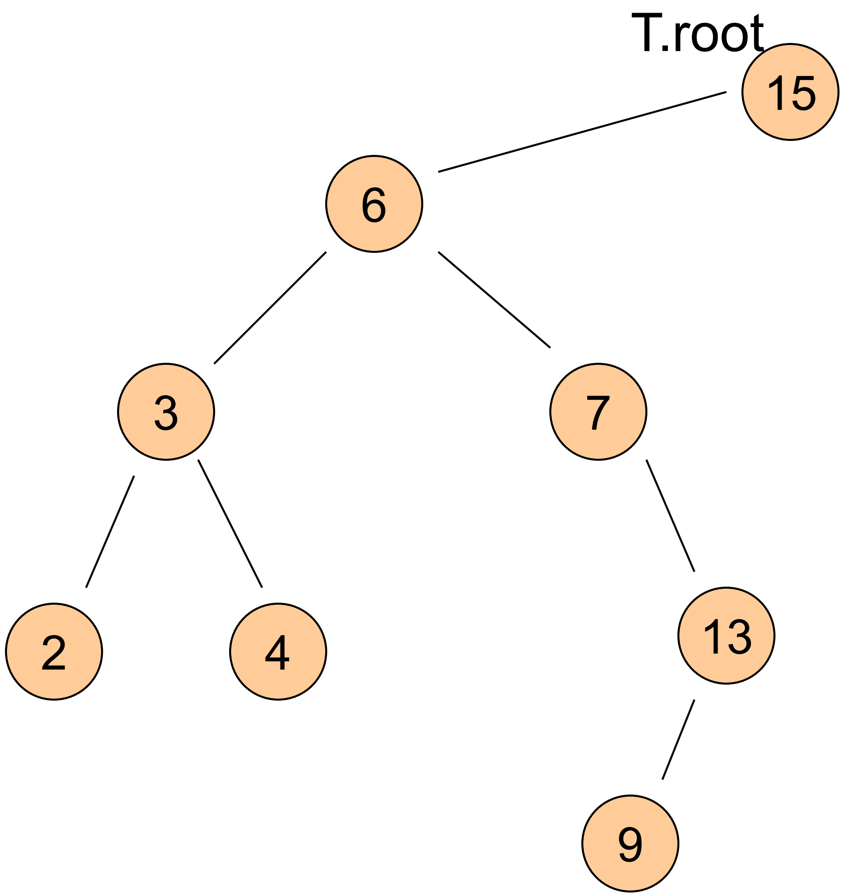


- Time complexity
  - $T(n) = \Theta(h)$ where h is height of input tree

Tree-insert

- Tree-insert(x, k)
  - Input: a BST tree node x and a key k
  - Output: insert k to the tree rooted at x such that the resulting tree is still a binary search tree

Examples



Tree-Insert(T.root, 8)

Tree-Insert(T.root, 6.5)

Use tree-search !

Tree-insert

Tree-insert(T, k)

```
y = Nil;   x = T.root
z.key = k; z.left = Nil; z.right=Nil
while (x ≠ Nil) do
        y = x
        if ( z.key < x.key )
            then  x = x.left
            else  x = x.right
 z.parent = y
 if (y = Nil) then T.root = z
 else if  (z.key < y.key)
        then     y.left = z
        else     y.right = z
```

- z is the new node to be inserted
- Locate potential parent y of z.

- Set up z as appropriate child of y

Tree-insert

Tree-insert(T, k)

```
y = Nil;   x = T.root
z.key = k; z.left = Nil; z.right=Nil
while (x ≠ Nil) do
        y = x
        if ( z.key < x.key )
            then  x = x.left
            else  x = x.right
z.parent = y
if (y = Nil) then T.root = z
else if (z.key < y.key)
        then    y.left = z
        else    y.right = z
```

- Time complexity
  - $T(n) = \Theta(h)$, where h is height of input tree

Summary: BST is good for both static and dynamic operations

- Suppose n input keys are already stored in a BST of height h

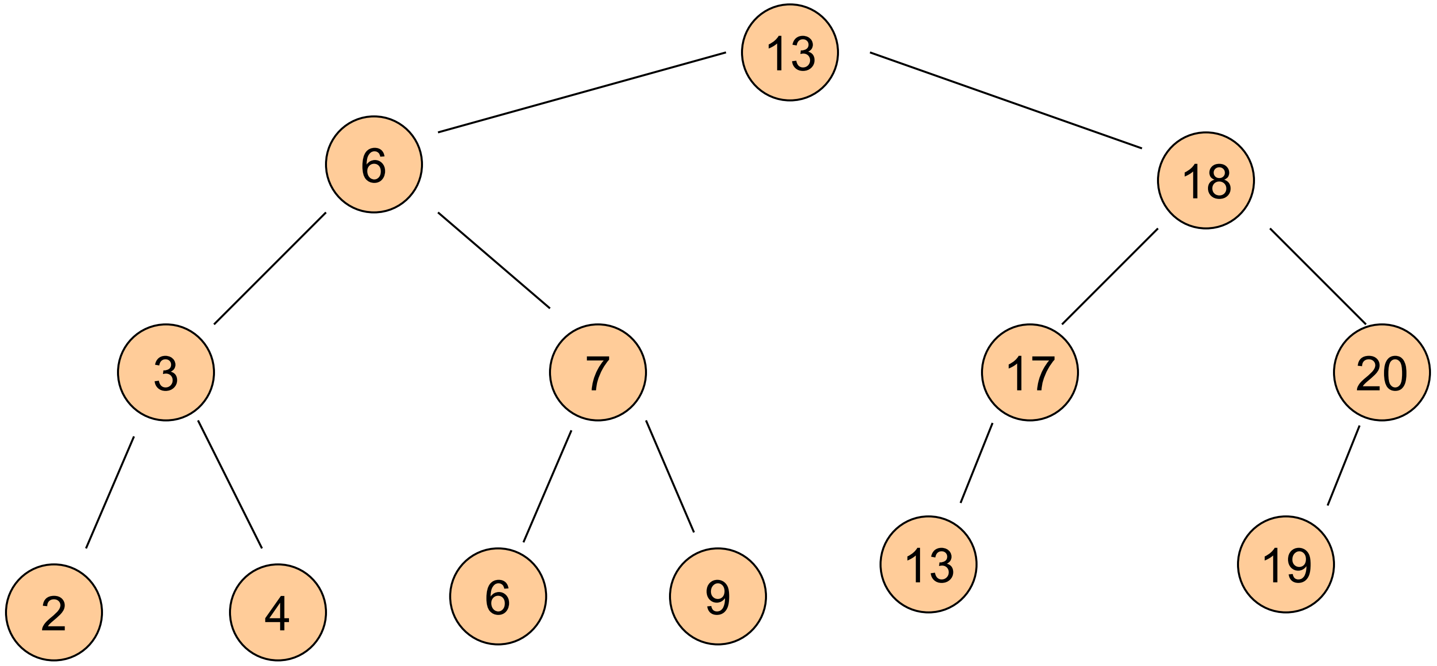
	Time complexity
Search	$\Theta(h)$
Maximum	$\Theta(h)$
Minimum	$\Theta(h)$
Successor	$\Theta(h)$
Predecessor	$\Theta(h)$
Insert	$\Theta(h)$
Delete	$\Theta(h)$
Extract-Max	$\Theta(h)$
Increase-key	$\Theta(h)$

- However, performance depending on height!
- Height $h = O(n)$ and $h = \Omega(\lg n)$

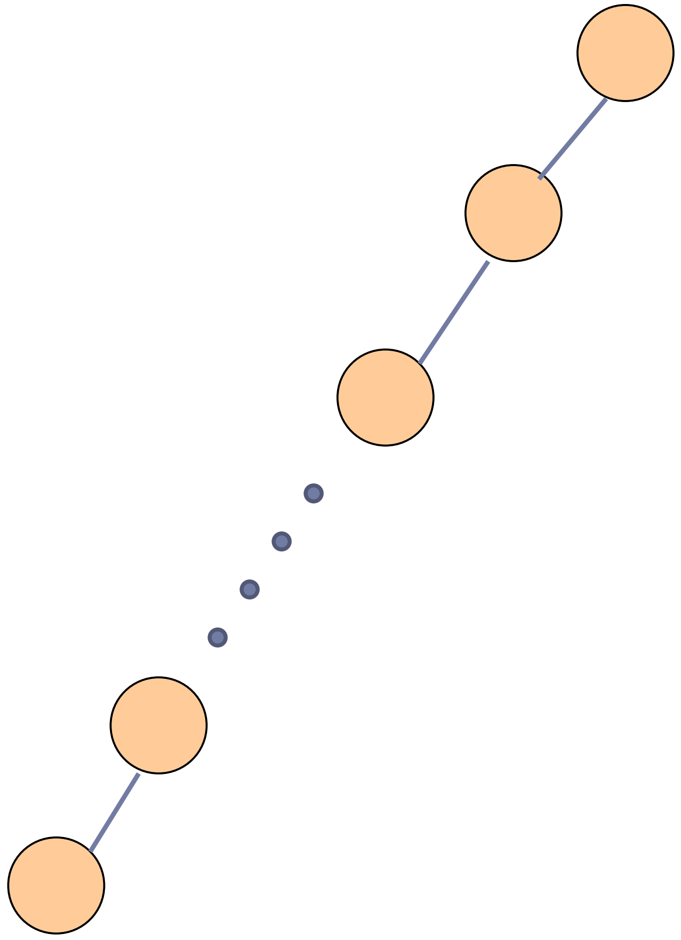
- To have good performance, we want to keep the tree height low!

Part C:
Balanced binary search tree

Good tree



Bad Tree



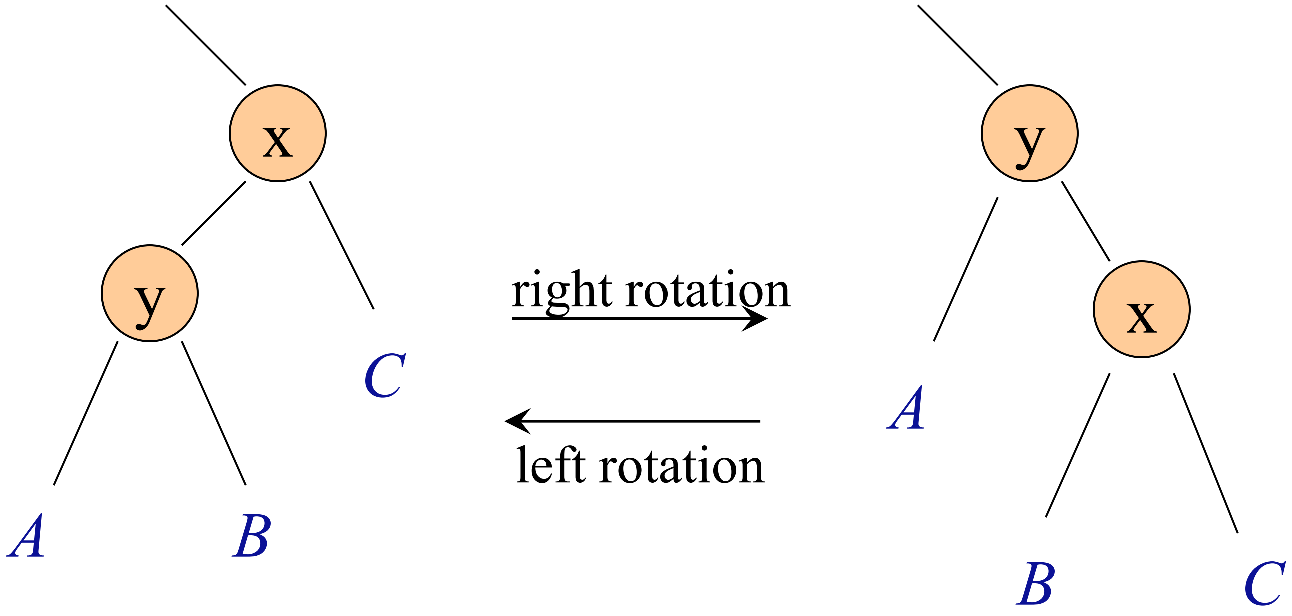
Balanced binary search tree

- It turns out that there are ways to add extra conditions to binary search trees, so that their height is $\Theta(\lg n)$
  - E.g, red-black tree, AVL tree, etc

- Once such a tree is created,
  - it can support search, minimum, maximum etc in $\Theta(h) = \Theta(\lg n)$ time using the same algorithms described before
  - the extra work comes at handling dynamic operations: insertion, deletion, and so on. Re-balancing is needed
  - however, for standard balanced BSTs, all these operations can be handled in $\Theta(\lg n)$ time.

Rotation operation

- Left rotation or Right rotation to keep tree height low



With balanced BST

- Suppose n input keys are already stored in a balanced BST

	Time complexity
Search	$\Theta(\lg n)$
Maximum	$\Theta(\lg n)$
Minimum	$\Theta(\lg n)$
Successor	$\Theta(\lg n)$
Predecessor	$\Theta(\lg n)$
Insert	$\Theta(\lg n)$
Delete	$\Theta(\lg n)$
Extract-Max	$\Theta(\lg n)$
Increase-key	$\Theta(\lg n)$

- Height of tree will be $\Theta(\lg n)$, where n is number of nodes in the tree

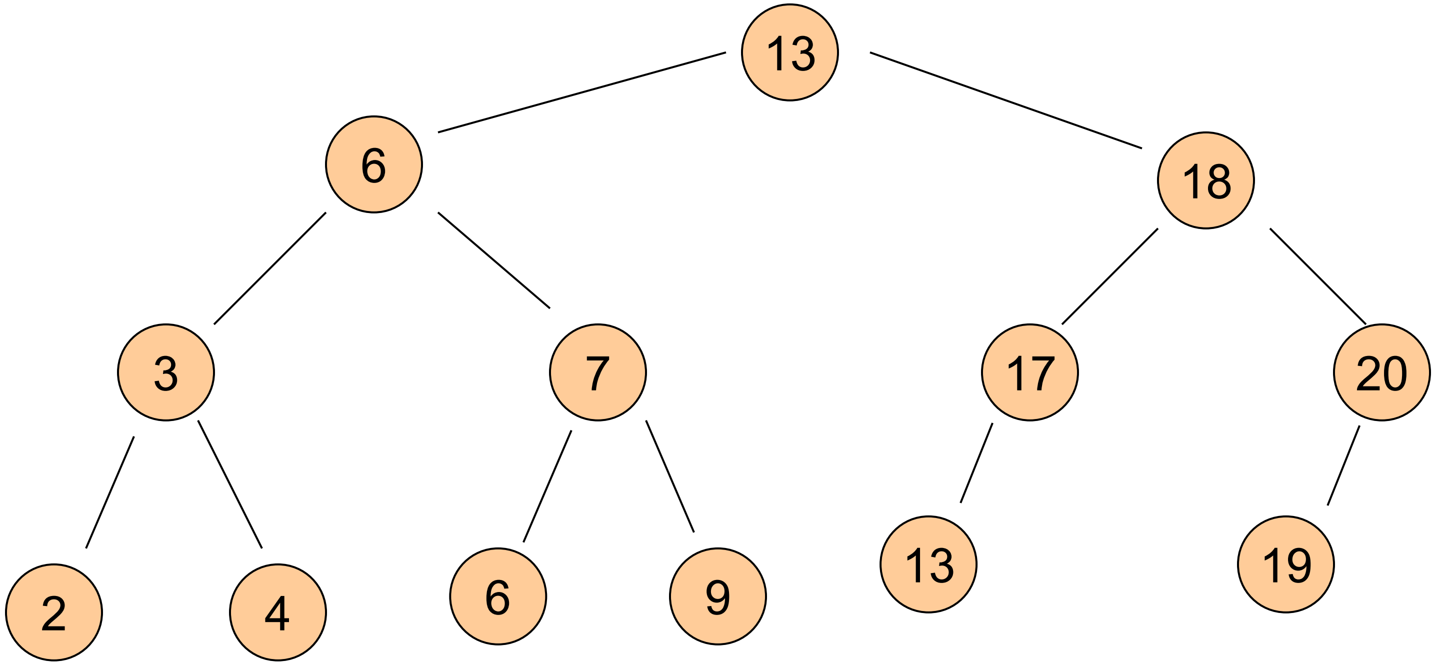
Part D:
Select queries
augmenting data structure

- What if we also want to perform Select operation

- BST-Select (x, k):
  - Given a list of records whose keys are stored in a tree rooted at x, return the node whose key has rank k.

- We can use QuickSelect to do this in linear time.. Why are we not satisfied?
  - What if we have to do it many times?
    - Sort it first
  - But what if we also have dynamic changes?
    - Need a data structure that can support Select under dynamic changes

BST



- What if we also want to perform Select operation

- BST-Select (*x*, *k*):
  - Given a list of records whose keys are stored in a tree rooted at *x*, return the node whose key has rank *k*.

- We can do linear search to find it. But can we do better?

- Goal:
  - Augment the binary search tree data structure so as to support Select (*x*, *k*) efficiently

In particular,

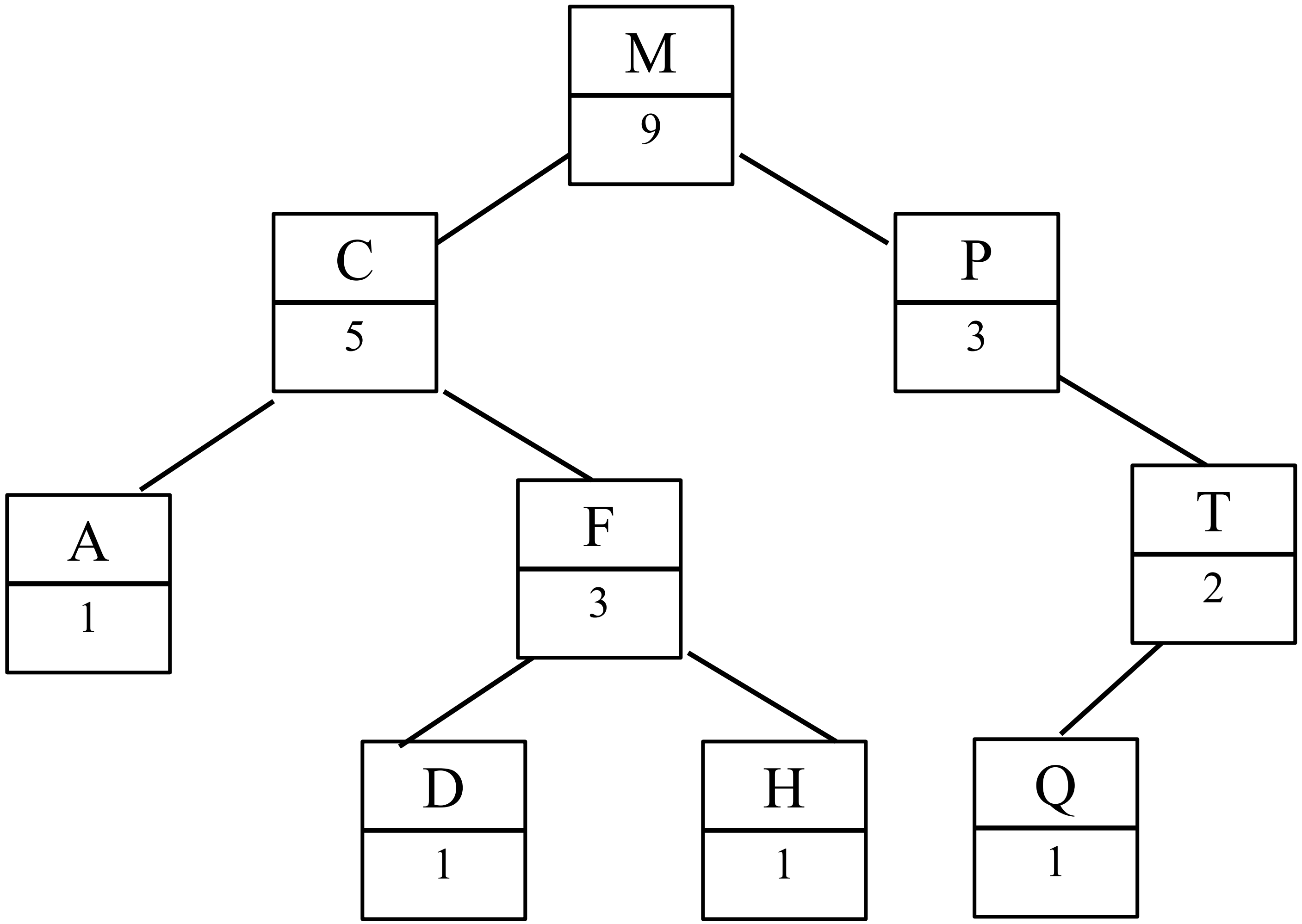
- BST-Select (*x*, *k*)
- Goal:
  - Augment the binary search tree data structure so as to support BST-Select (*x*, *k*) efficiently

- Ordinary binary search tree T
  - *O(h)* time for BST-Select(*T.root*, *k*) where *h is height of tree T*
- Using balanced search tree)
  - *O(lg n)* time for BST-Select(*T.root*, *k*)

How do we augment a BST T?

- At each node *x* of the tree *T*
  - store *x.size* = # nodes in the subtree rooted at *x*
    - Include x itself
    - If a node (leaf) is NIL, its size is 0.
- Space of an augmented tree:
  - $\Theta(n)$

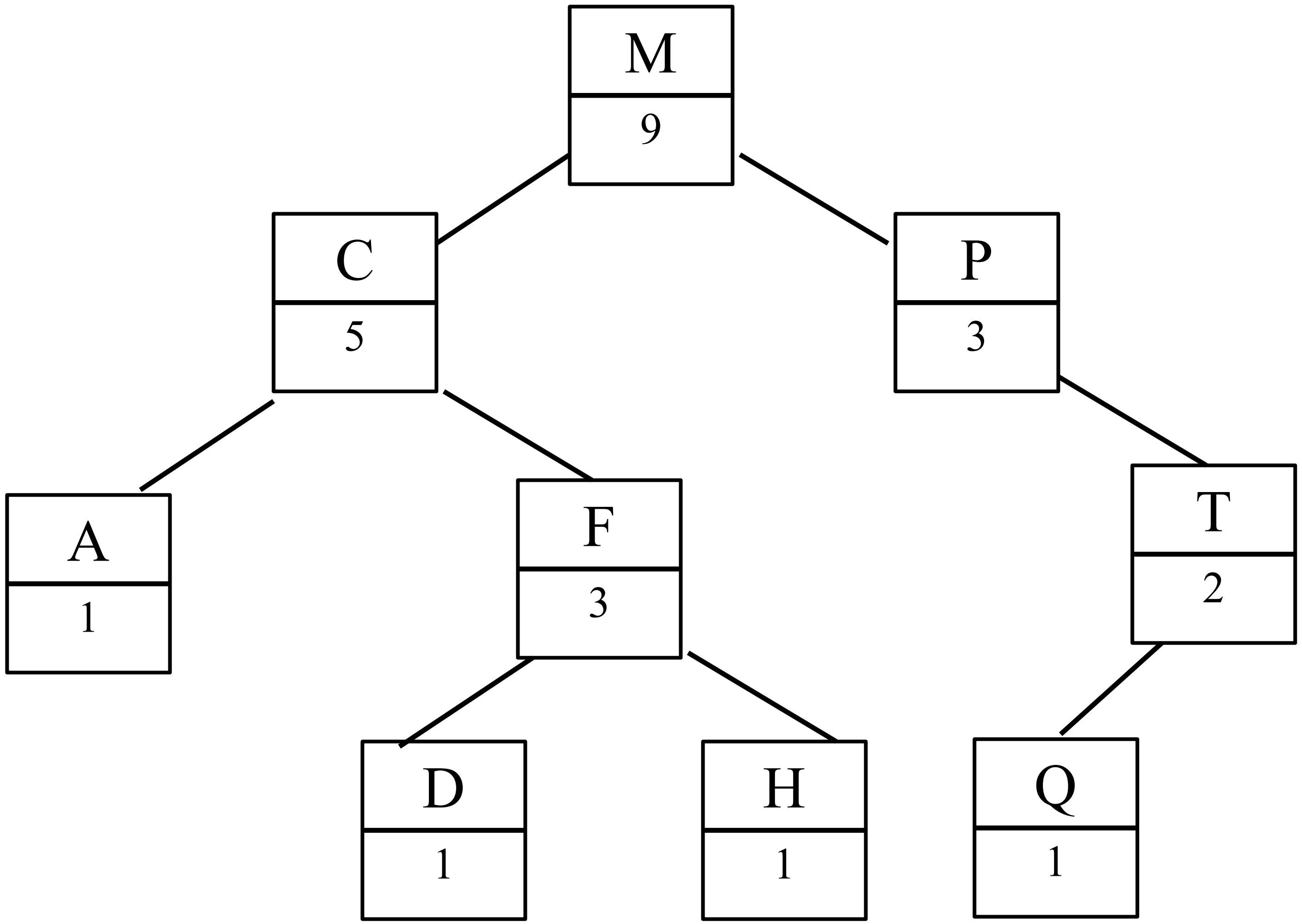
An example



How do we augment a BST T?

- At each node *x* of the tree *T*
  - store *x.size* = # nodes in the subtree rooted at *x*
    - Include x itself
    - If a node (leaf) is NIL, its size is 0.
- Space of an augmented tree:
  - $\Theta(n)$
- Basic property:
  - $x.size = x.left.size + x.right.size + 1$

How to set up size information ?



How to setup size information?

- procedure AugmentSize($treenode\ \ x$)

  If ($x \neq NIL$) then

  $Lsize$ = AugmentSize($x.left$);

  $Rsize$ = AugmentSize($x.right$);

  $x.size = Lsize + Rsize + 1;$

  Return($x.size$);

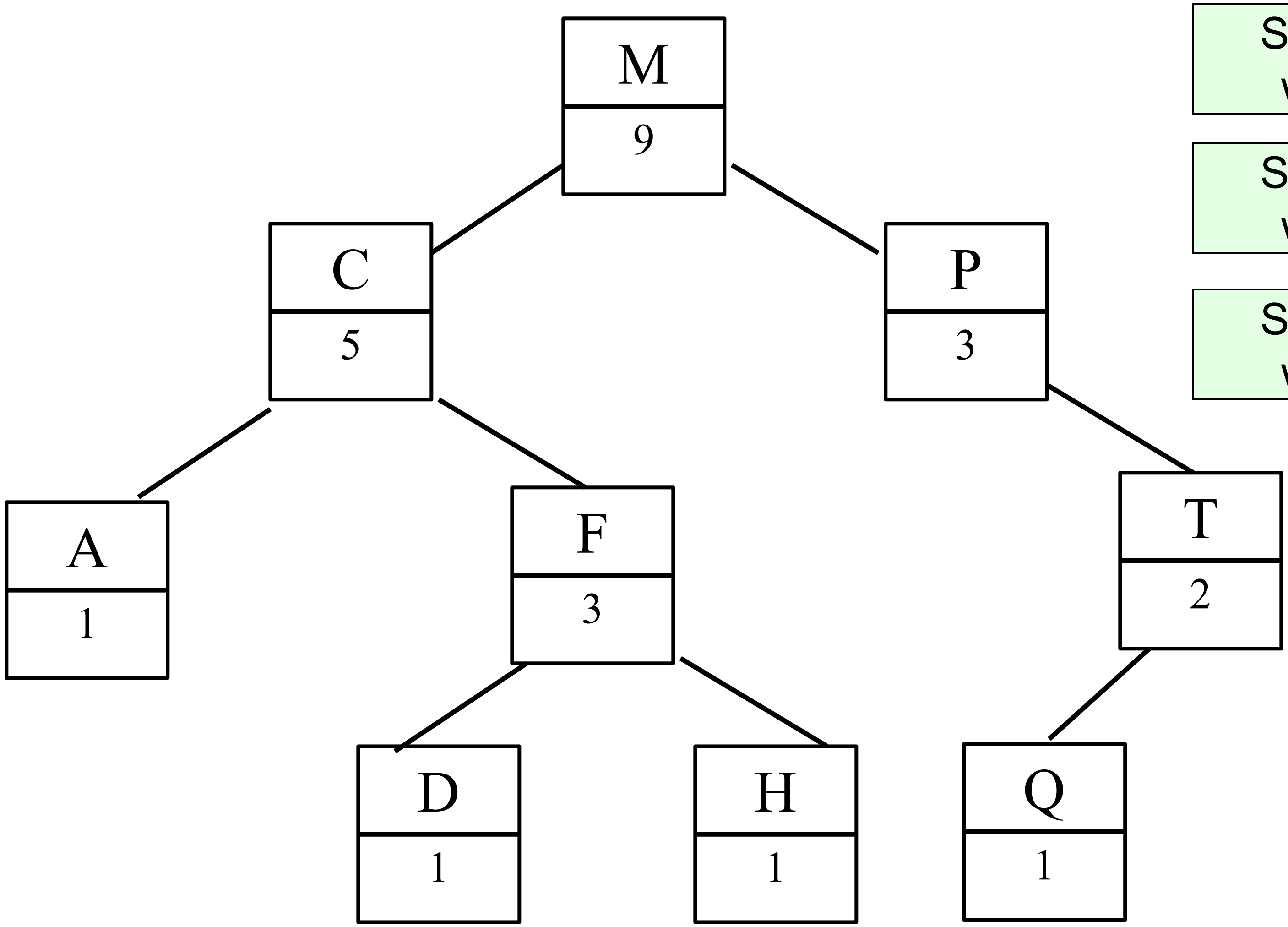
  end

  Return (0);

Postorder traversal
of the tree !

Time complexity for Augmentsize:
$\Theta(size\ of\ tree)$

How to perform select with aug-BST?



Select element with rank 3?

Select element with rank 7?

Select element with rank 6?

- Let *T* be an augmented binary search tree
- *BST-Select(x, k)*:
  - Return the *k*-th smallest element in the subtree rooted at *x*
  - *BST-Select(T.root, k)* returns the *k*-th smallest elements in the entire tree.

- Using ideas just described, BST-Select(x, k) can be implemented to have $\Theta(height\ of\ tree)$ time complexity
  - which is $\Theta(\lg n)$ for a balanced binary tree.
  - See homework.

Are we done?

- Need to maintain the augmented information under dynamic changes of the tree!
  - i.e, under insertions / deletions
  - in this case, just adjusting this size count as we update nodes, or under rotations, and it does not increase asymptotic time complexity of these operations

- Remark:
  - Select() in an sorted array can be done in $\Theta(1)$ time.
  - However, an array does not support dynamic operations (insert/delete) efficiently. That's augmented BST is a better data structure in this case.

Summary

- Simple example of augmenting data structures
- In general, the augmented information can be quite complicated
  - Can be a separate data structure!
- Need to consider how to maintain such information under dynamic changes

FIN