CS217 - Data Structures & Algorithm Analysis (DSAA)

Lecture #9

Binary Search Trees

Prof. Pietro S. Oliveto
Department of Computer Science and Engineering
Southern University of Science and Technology (SUSTech)

olivetop@sustech.edu.cn
https://faculty.sustech.edu.cn/olivetop

Reading: Chapter 12

Aims of this lecture

- We’ve seen a lot of binary trees already
  - Recurrence tree for visualising runtime in recursive calls
  - HeapSort uses imaginary trees
  - Decision trees in the lower bound for comparison sorts
- Now: discussing binary trees more thoroughly, including how to prove **inductive statements about trees**.
- To introduce **binary search trees** and their typical operations.
- To work out the **running time** for operations on binary search trees.

Recall

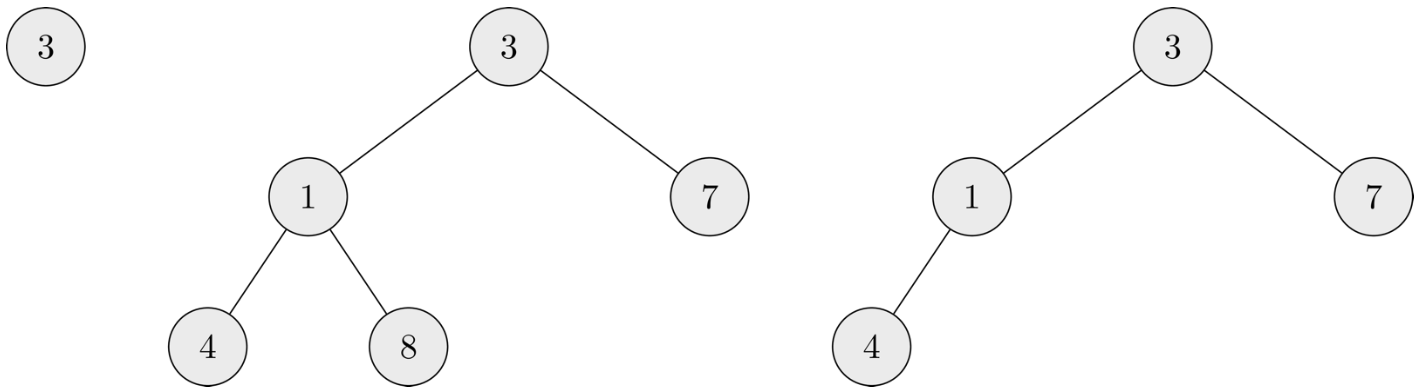
- Elements can contain **satellite data** and a **key** is used to identify the element.
- Typical operations:
  - **Search(S, k):** returns element x with **key k**, or NIL
  - **Insert(S, x):** adds **element x** to S
  - **Delete(S, x):** removes element x from S
  - **Minimum(S), Maximum(S):** return x resp. with smallest or largest key
  - **Successor(S, x), Predecessor(S, x):** next larger (smaller) than Key(x)
- **Time** often measured using n as the number of elements in S.

Binary trees

- Intuitively: trees where every node has at most two children.
- We can define binary trees recursively:
- A **binary tree** is a structure defined as finite set of nodes such that either
  - The tree is empty (no nodes) or
  - It is composed of a root node, a left subtree and a right subtree
- This view is very handy for proving statements about trees by induction (see later).
- The root of the left subtree of a node is called **left child**, that of the right subtree is called **right child**.

Definitions for binary trees

- We tacitly assume that all nodes are labelled by numbers.



- A **path** in a tree is a sequence of nodes linked by edges. The **length** of a path is the number of edges.
- A **leaf** of a tree is a node that has no children; otherwise it is called **internal node**.
- We speak about **siblings, parents, ancestor, descendant** in the obvious way.

Depth and height

- The **depth** of a node in a tree is the length of a (simple) path from that node to the root.
- A **level** of a tree is a set of nodes of the same depth.
- The **height** of a node in a tree is the length of the longest path from that node to a leaf.
- The **height of a tree** is the height of its root.
- A binary tree is **full** if each node is either a leaf or has exactly two children.
- It is **complete** if it is full and all leaves have the same level.

Inductive proofs on trees

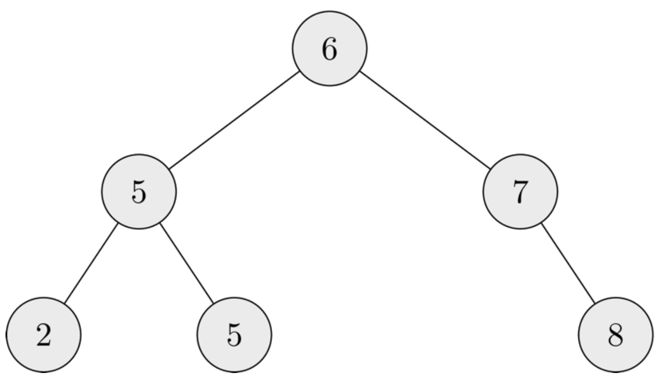
- We can use the recursive definition to prove statements about trees inductively. The general recipe is this:
- **Proof**:
  - **Base case**: show that the statement holds for the "smallest" tree, e.g. an empty tree or just the root node (depending on the statement).
  - **Induction step**: any larger tree has a root and two subtrees (possibly empty). Assume that the statement holds for both subtrees and show that it then holds for the whole tree.
- Caveat: if a statement reads "for all non-empty trees", in the induction step we may need to watch out for empty subtrees.

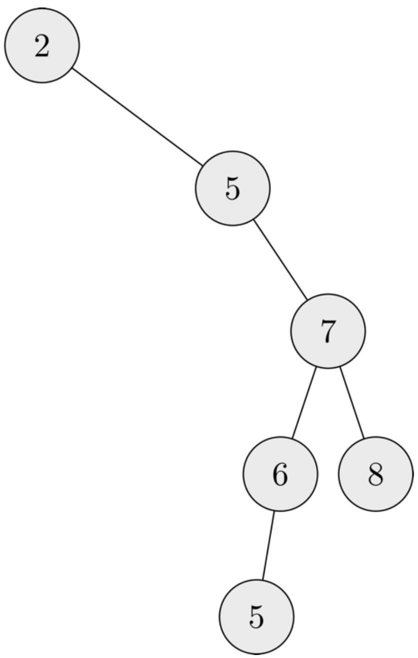
Inductive proofs on trees: example

- **Theorem**: A binary tree of height at most h has no more than 2^h leaves.
  - We have used this statement in the lower bound for comparison sorts. Now we prove it.
- **Proof**:
  - **Base case**: a tree of height 0 has no more than $2^0=1$ leaves.
  - **Induction step**: a tree of height $h>0$ has a root and two subtrees (possibly empty) of height at most $h-1$. Assume that the statement holds for both subtrees. Then the subtrees have at most 2^{h-1} leaves, so the whole tree has at most $2\cdot 2^{h-1}=2^h$ leaves.

Binary search trees

- A binary search tree (BST) is a binary tree where all labels (keys) satisfy the **binary search tree property**:
  - If y is a node in the left subtree of x, then y.key ≤ x.key.
  - If y is a node in the right subtree of x, then y.key ≥ x.key.

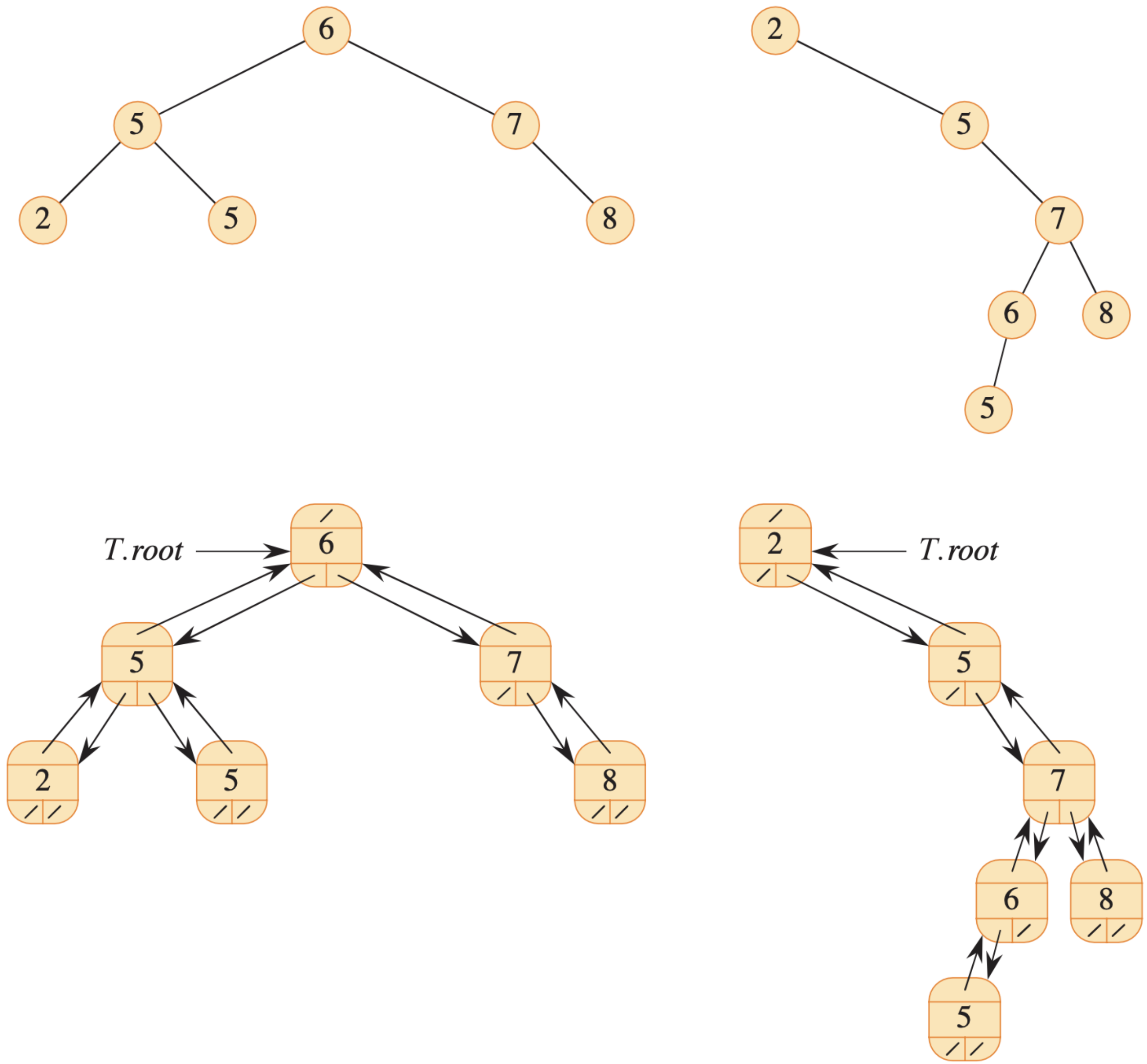




Binary search trees: representation

- Linked list
- Key
- Satellite data
- Attributes:
  - **T.Root**
  - **Left** child pointer
  - **Right** child pointer
  - **Parent** pointer

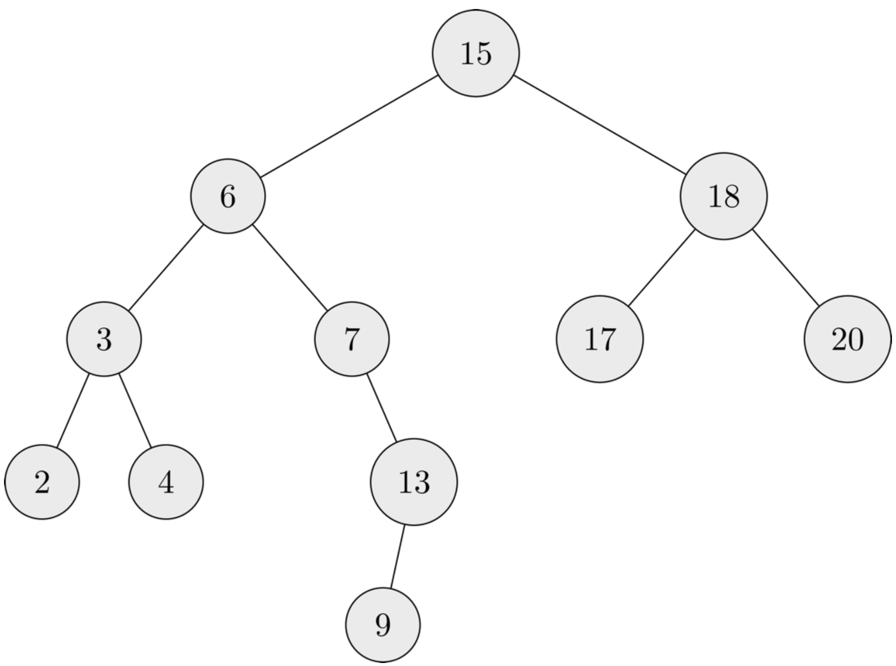
- Parent of T.root is NIL



(a)
(b)

Searching in a BST

- **Search(x, k):** returns the element with key k in a tree rooted in x, or NIL
- **Idea**: compare against current key and stop or go down left or right.



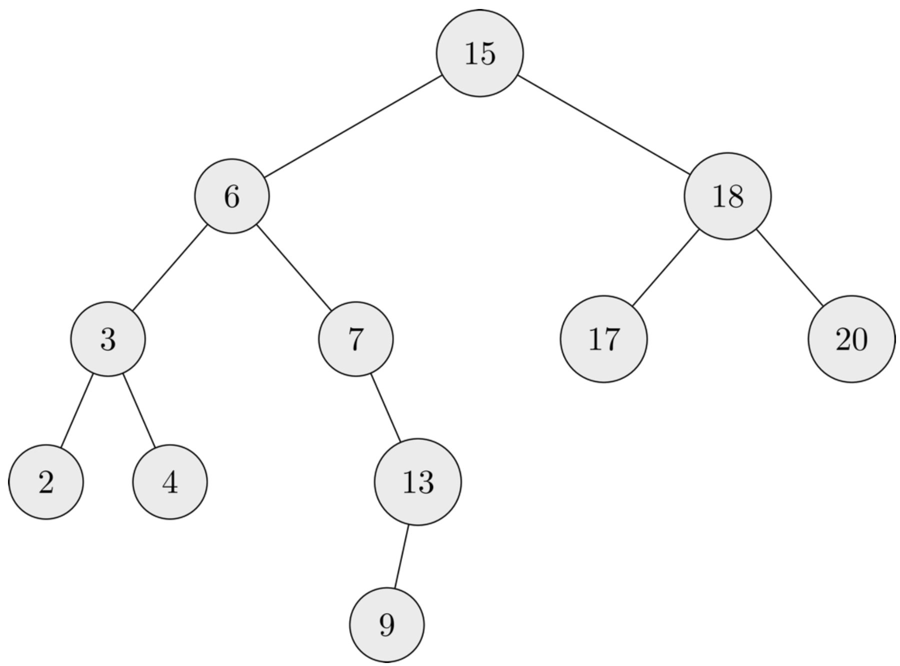

```
TREE-SEARCH(x, k)
    if x == NIL or k == x.key
        return x
    if k < x.key
        return TREE-SEARCH(x.left, k)
    else return TREE-SEARCH(x.right, k)

ITERATIVE-TREE-SEARCH(x, k)
    while x ≠ NIL and k ≠ x.key
        if k < x.key
            x = x.left
        else x = x.right
    return x
```

Runtime: *O(h), h* the height of the tree

Minimum, Maximum, Successor in a BST

Minimum: starting from the root, go left until the left child is NIL.

Maximum: starting from the root, go right until the right child is NIL.



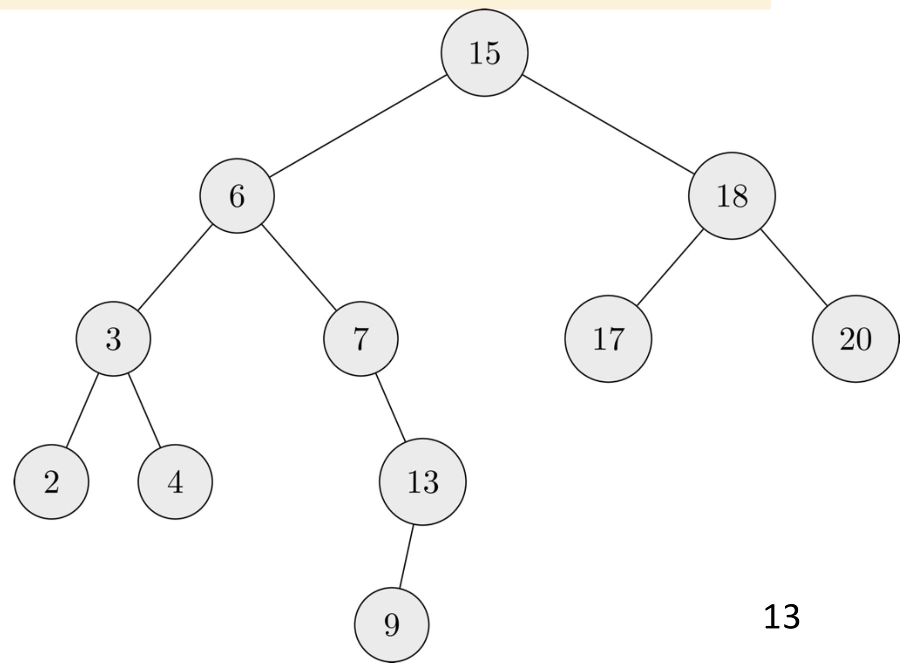

```
TREE-MINIMUM(x)
  while x.left ≠ NIL
      x = x.left
  return x

TREE-MAXIMUM(x)
  while x.right ≠ NIL
      x = x.right
  return x
```

Runtime: *O(h), h* the height of the tree

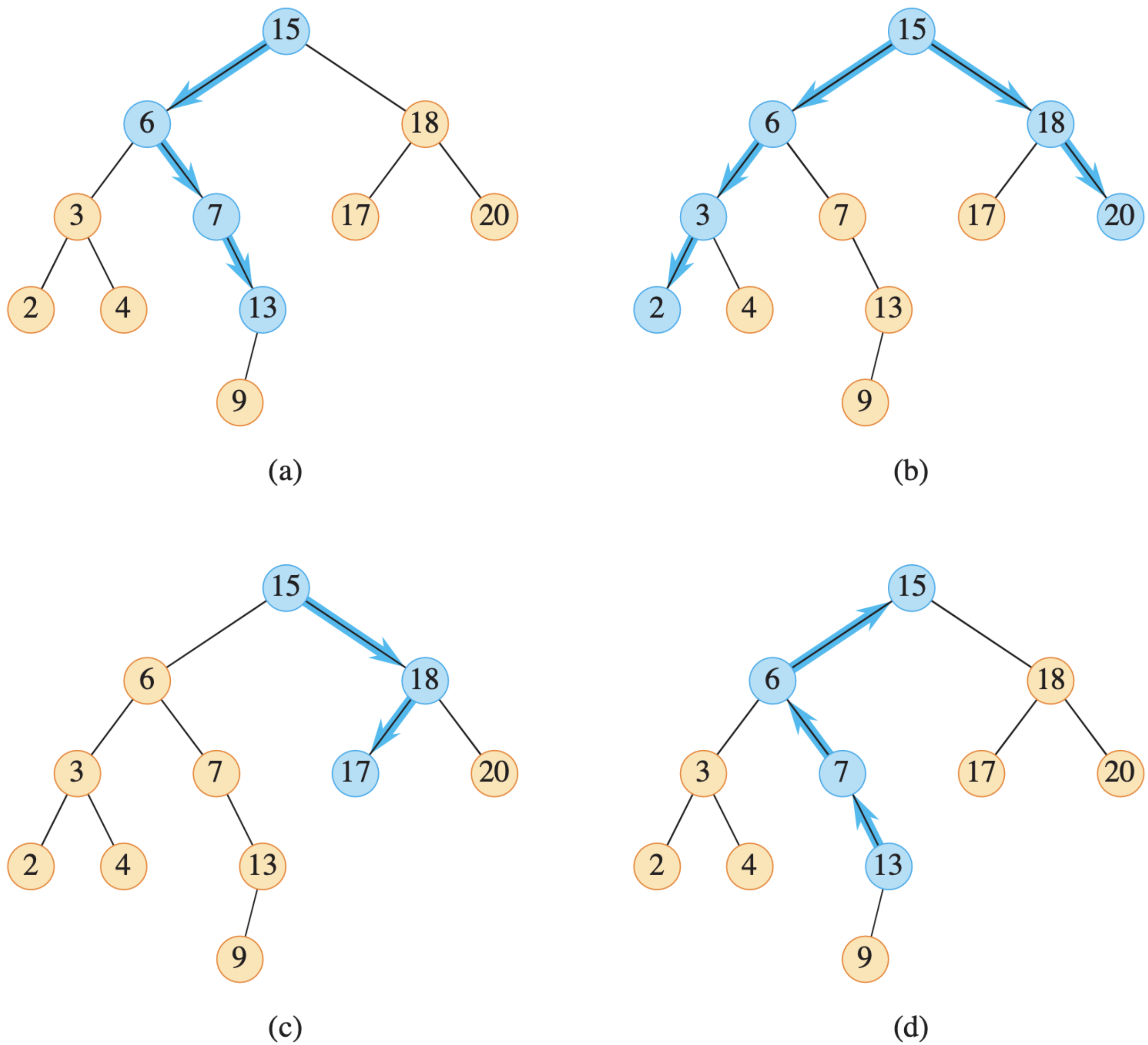
Minimum, Maximum, Successor in a BST

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)   // leftmost node in right subtree
else // find the lowest ancestor of x whose left child is an ancestor of x
    y = x.p
    while y ≠ NIL and x == y.right
        x = y
        y = y.p
    return y
```

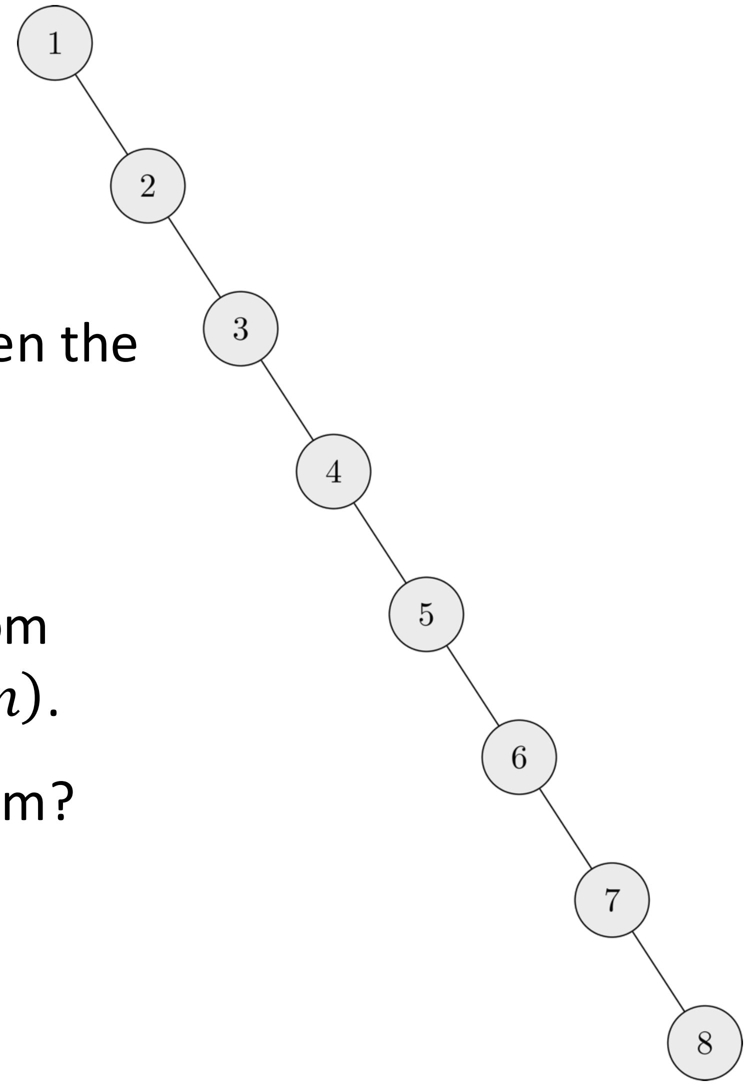
Runtime: *O(h), h* the height of the tree



Searching in a BST: Summary



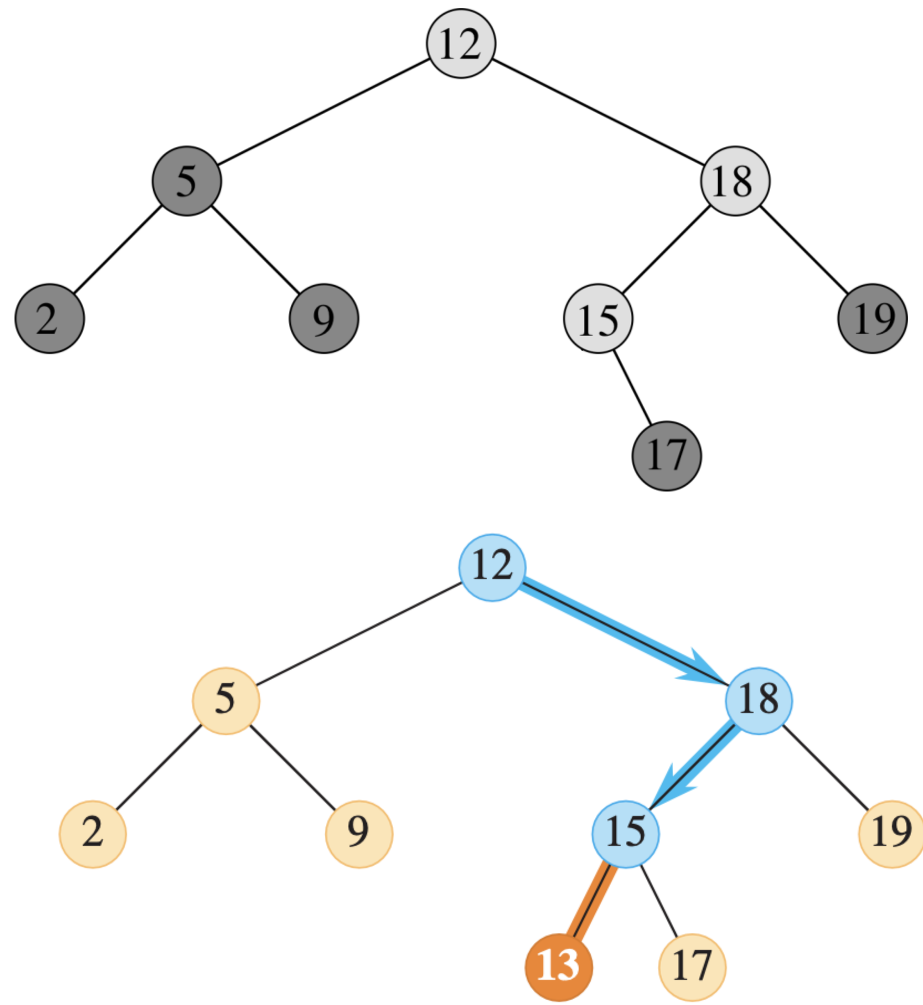
Searching in a BST: Worst case runtime



- BSTs can be **imbalanced** and even degenerate to a single path!
- Height can be as bad as *n-1,* e.g. when the input is sorted.
- So the **worst-case runtime is $\Theta(n)$.**
- If keys are inserted in uniform random order, the expected height is $O(\log n)$.
- Can we rely on our data being random? Such inputs might be very unlikely.
- We'll see **balanced trees** later on, guaranteeing a height of $O(\log n)$.

▶Insert(T,*z*)

Example: Insert(T,13)



Idea

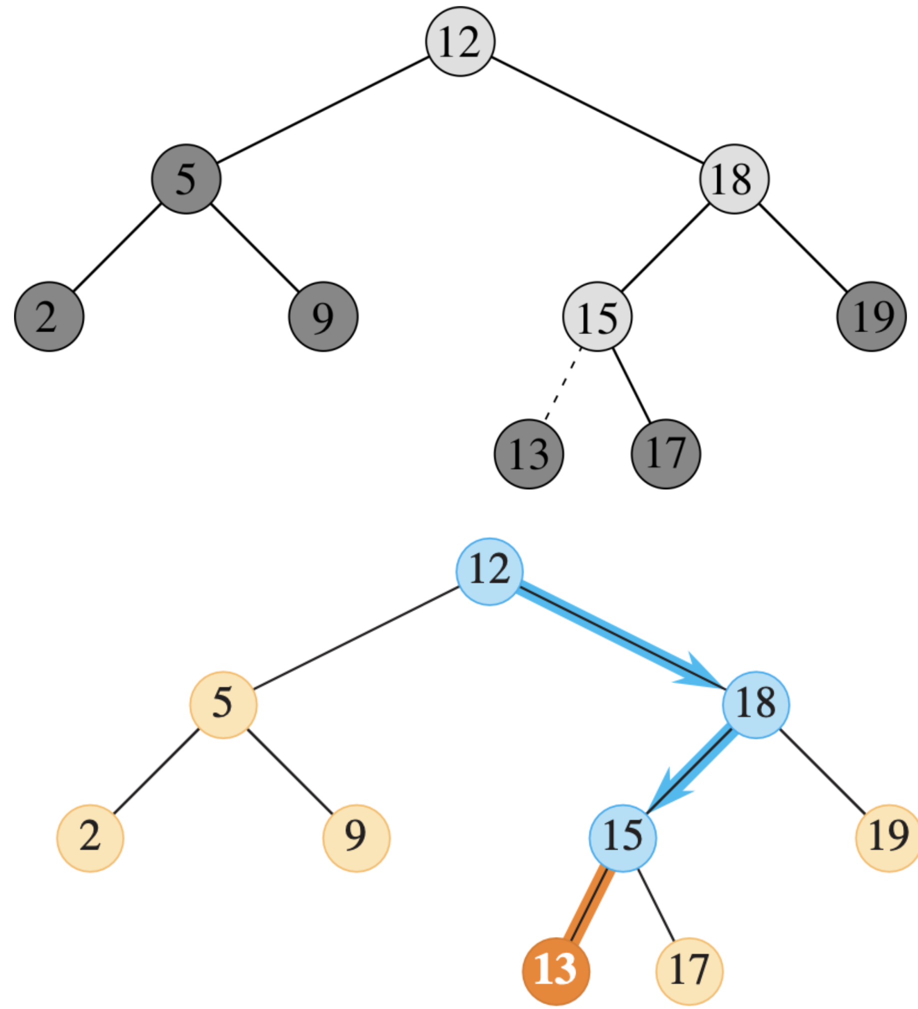
Go down the tree like in Search to find where the new element needs to go.

1. The search will end in NIL, hence we record the **search path** (e.g. 12, 18, 15, NIL).
2. Add the element as a left or right subtree to last non-NIL node.

▶ Insert(T,*z*)

```
TREE-INSERT(T, z)
x = T.root
y = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

Example: Insert(T,13)

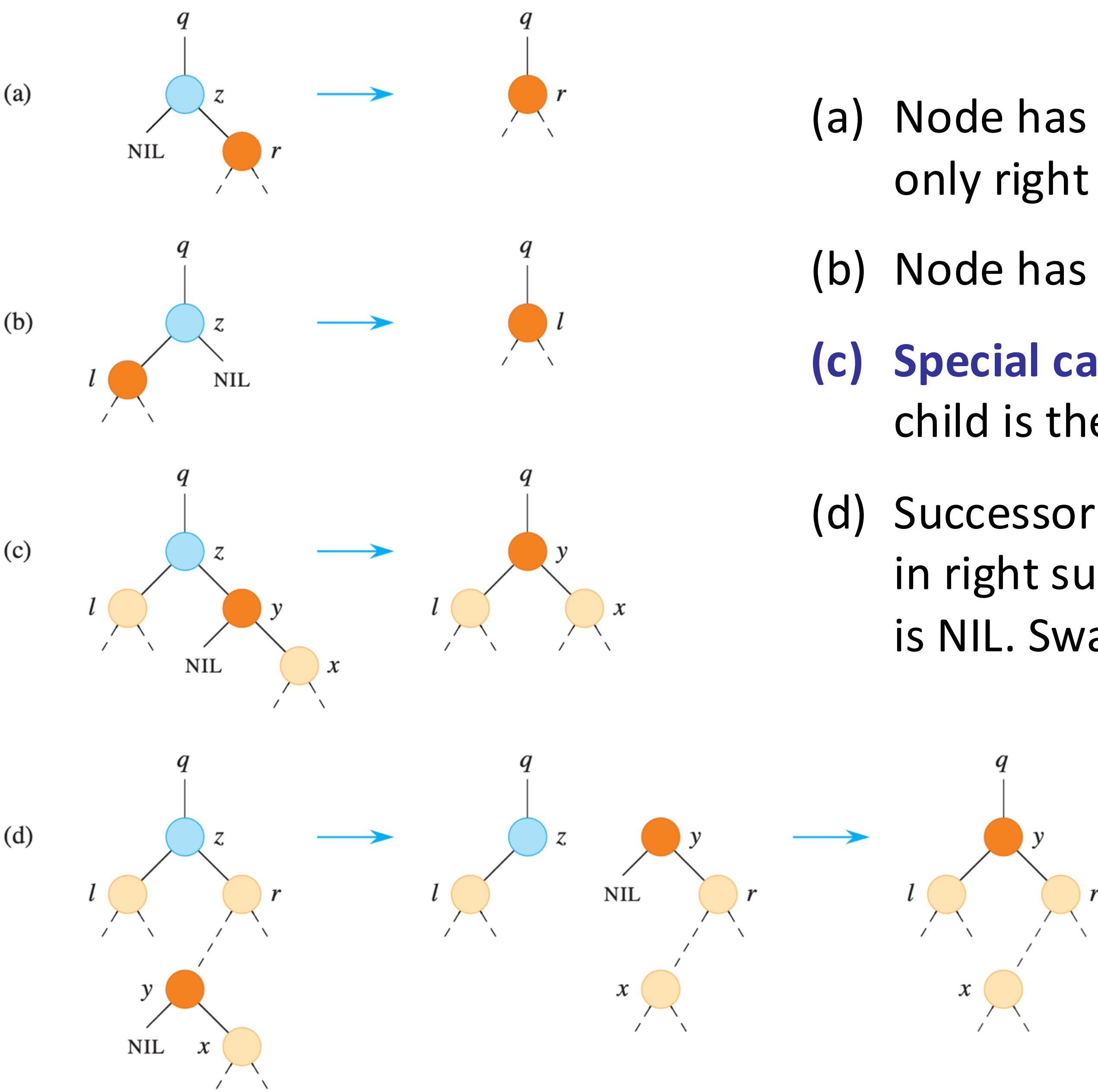


Runtime: *O(h), h* the height of the tree

▶Delete(T,z)

- **Idea:** Three cases

1. Easy when z is a **leaf** (delete z).
2. If **z has one child**, have the child replace z.
3. Otherwise, if **z has two children**, we can't leave a hole in the tree!
   - Solution: replace z with its **successor**.
   - z's successor is the minimum in the right subtree (this subtree exists since z has two children).
   - z's successor has no left child.
   - Hence we can swap it with z and then delete z.



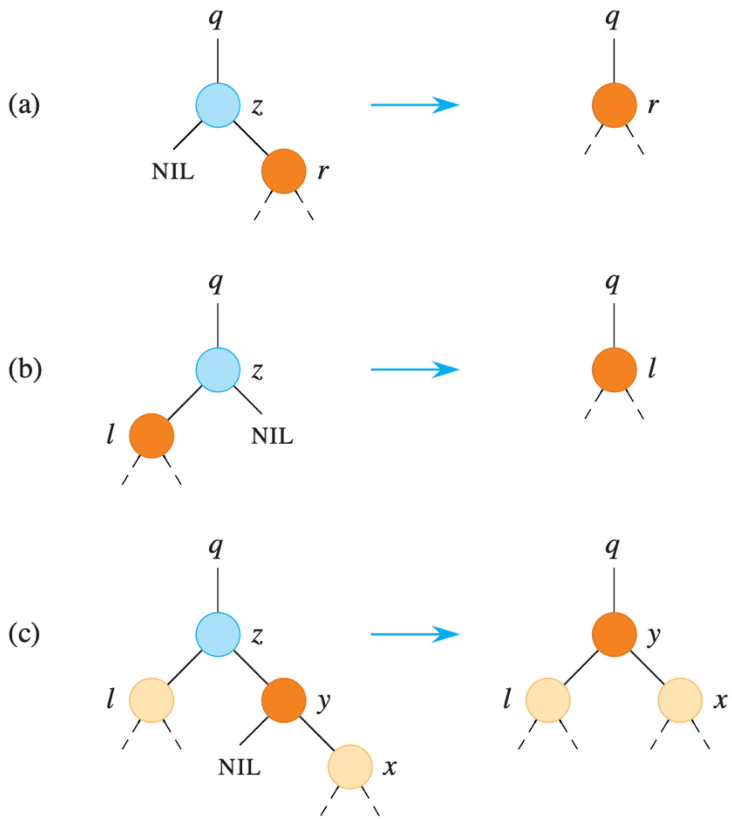
(a) Node has no children or only right child

(b) Node has only left child

(c) **Special case** where right child is the successor.

(d) Successor *y* is the minimum in right subtree; *y*'s left child is NIL. Swapping z and y.

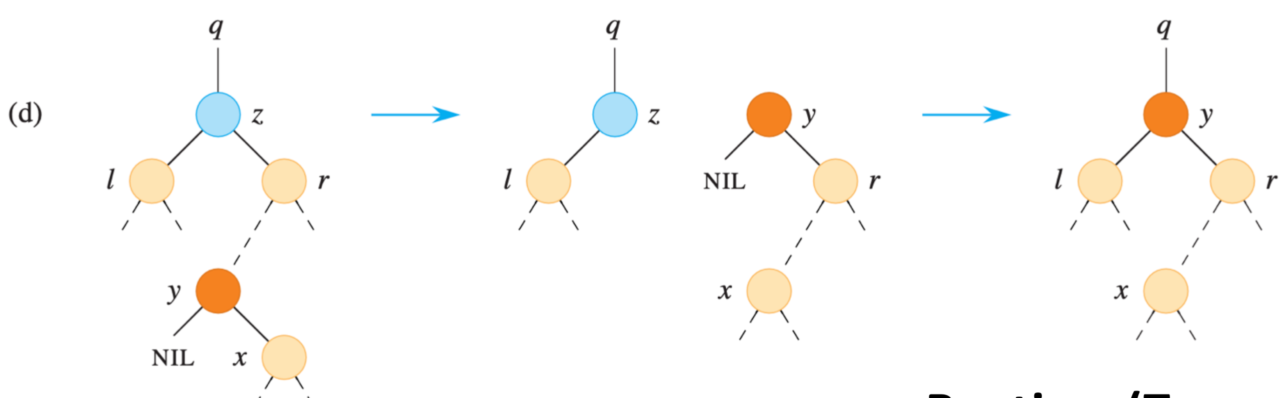
▶Transplant(T,u,v)




```
TRANSPLANT(T, u, v)
  if u.p == NIL
        T.root = v
  elseif u == u.p.left
        u.p.left = v
  else u.p.right = v
  if v ≠ NIL
        v.p = u.p
```

Replaces subtree rooted in u with a subtree rooted in v

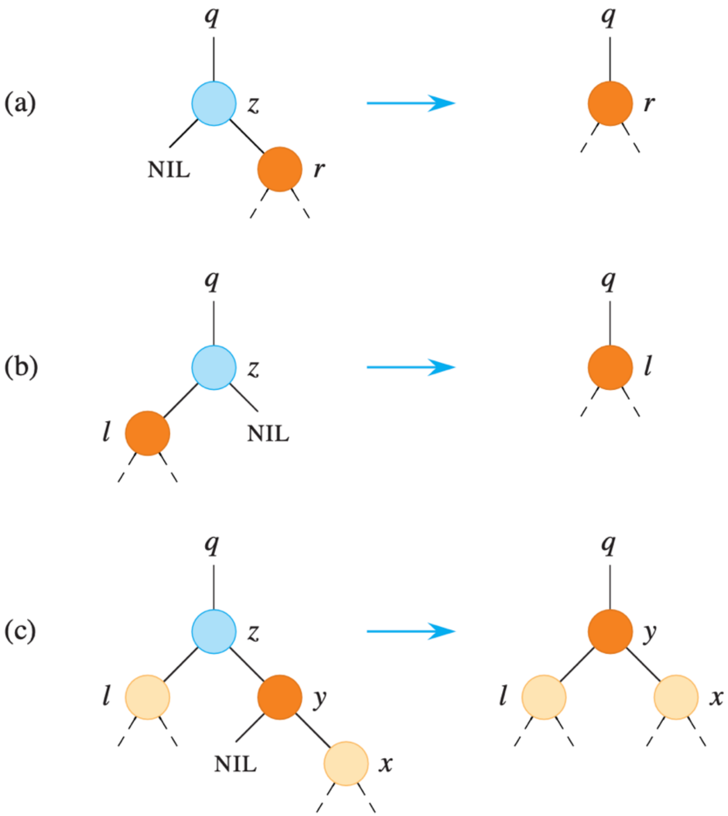
Transplant does not update $v.left$ and $v.right$: this is the responsibility of the caller of ***Transplant!!***



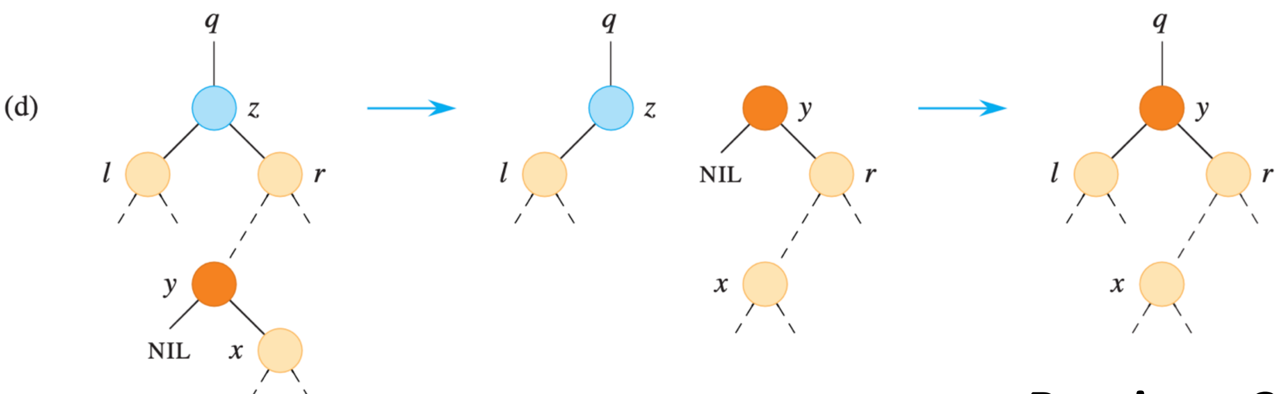
Runtime (Transplant) ?

O(1)

Delete(T,z)




```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)          // replace z by its right child
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)           // replace z by its left child
else y = TREE-MINIMUM(z.right)         // y is z's successor
    if y ≠ z.right                     // is y farther down the tree?
        TRANSPLANT(T, y, y.right)      // replace y by its right child
        y.right = z.right              // z's right child becomes
        y.right.p = y                  //      y's right child
    TRANSPLANT(T, z, y)                // replace z by its successor y
    y.left = z.left                    // and give z's left child to y,
    y.left.p = y                       //      which had no left child
```



Runtime: *O(h), h* the height of the tree

Tree walks

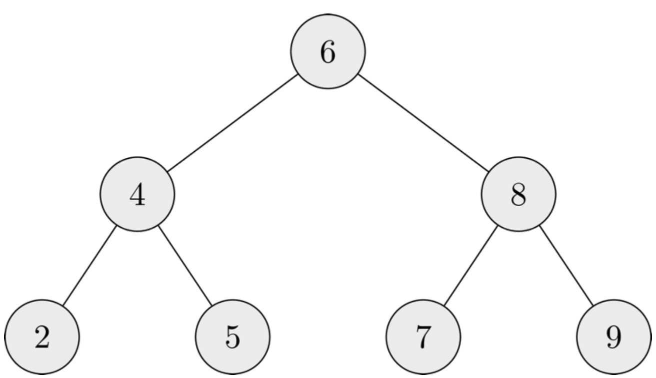
- We can print out the keys of a BST by a **tree walk**:

```
PREORDER(x)
if x ≠ NIL then
    print x.key
    PREORDER(x.left)
    PREORDER(x.right)
```

```
INORDER(x)
if x ≠ NIL then
    INORDER(x.left)
    print x.key
    INORDER(x.right)
```

```
POSTORDER(x)
if x ≠ NIL then
    POSTORDER(x.left)
    POSTORDER(x.right)
    print x.key
```

- Inorder tree walk outputs sorted sequence.



Inorder: 2, 4, 5, 6, 7, 8, 9
Preorder: 6, 4, 2, 5, 8, 7, 9
Postorder: 2, 5, 4, 7, 9, 8, 6

Tree walks: runtime

- **Theorem**: Inorder (Preorder/Postorder) tree walk of the root of an *n*-node tree takes time $\Theta(n)$.

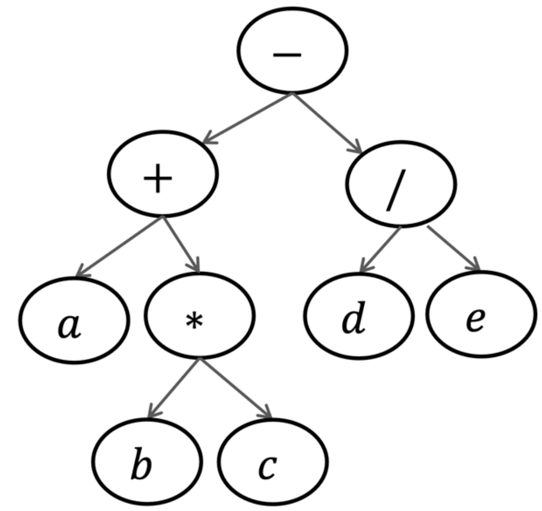
```
PREORDER(x)
if x ≠ NIL then
    print x.key
    PREORDER(x.left)
    PREORDER(x.right)
```

```
INORDER(x)
if x ≠ NIL then
    INORDER(x.left)
    print x.key
    INORDER(x.right)
```

```
POSTORDER(x)
if x ≠ NIL then
    POSTORDER(x.left)
    POSTORDER(x.right)
    print x.key
```

- Book gives a rather dull proof based on recurrences.
- A simpler proof:
  - Assign costs (time) for operations made at x to node *x*.
  - Cost at each node is $\Theta(1)$, and all costs are accounted for.
  - Sum of costs = runtime is $n \cdot \Theta(1) = \Theta(n)$.
- NB: This kind of argument is called **accounting method**.

Algebraic Expressions



- Algebraic expression with binary operators
- \+ - * /
- We can use a binary tree to represent it because the operations are binary
- Internal nodes: **operators**
- Leaves: **operands**
- (a+(b*c)) – (d/e)

- Inorder tree walk? **((a+(b*c))-(d/e))** Print **(** before left visit and **)** after right subtree visit
- Postorder tree walk? **a b c * + d e / -**
- Preorder tree walk? **- (+(a,*(b,c)),/(d,e))** add commas **,** after left visit

Inorder: infix expression; **Postorder:** postfix expression (stack);
Preorder: functional programming notation

Summary

- Binary trees have at most 2 children and can be defined recursively:
  - A tree is either empty or it contains a root and two subtrees (=trees).
  - Very useful for inductive proofs for trees.
- Binary search trees store data such that smaller keys are in the left subtree and larger keys are in the right subtree.
- BSTs of height *h* execute the following operations in time $O(h)$
  - **Searching, Minimum, Maximum, Successor**
  - **Insertion**
  - **Deletion**
- Binary search trees can be **imbalanced:** trees can degenerate to height $h = \Theta(n)$ **and worst-case time $\Theta(n)$** for many operations.
- Inorder/preorder/postorder walks output all elements in time $\Theta(n)$.