Binary Search Trees

CLRS 12.1 – 12.3

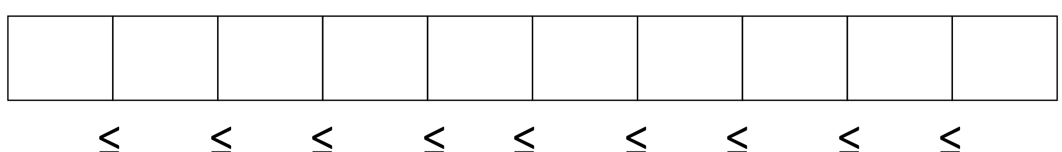
(+ some supplemental material)

(Supplemental): What is *Binary Search*?

- ”Binary Search” vs. “Binary Search Tree (BST)”
- To understand a BST, let’s talk first about what a **binary search** is

<u>Binary Search – occurs on an **array** of sorted items</u>

- Find an element *k*
- After checking a key *j* in the sequence, we can tell if item with key *k* will come before or after it

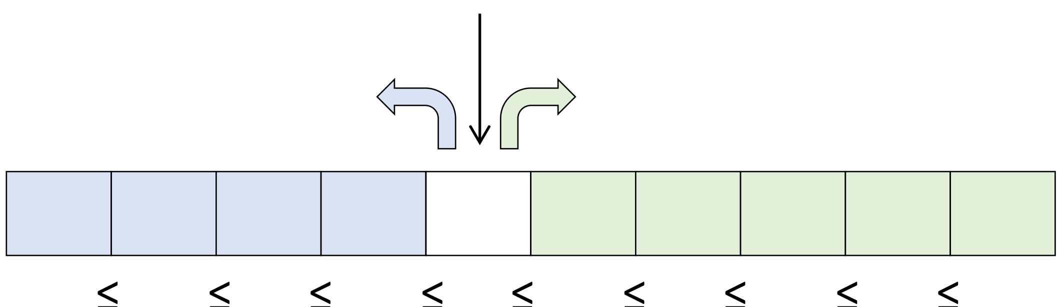


(Supplemental): What is *Binary Search*?

- ”Binary Search” vs. “Binary Search Tree (BST)”
- To understand a BST, let’s talk first about what a **binary search** is

<u>Binary Search – occurs on an **array** of sorted items</u>

- Find an element *k*
- After checking a key *j* in the sequence, we can tell if item with key *k* will come before or after it
- Which item should we compare against first?
  - The middle!



Ex. Binary Search: Find k = 52

Algorithm BinarySearch(*S*, *k*, *low*, *high*):

if *low* > *high* **then return** *NO_SUCH_KEY*

mid $\leftarrow$ $\lfloor(low + high) / 2\rfloor$

if *key(mid)* = *k* **then return** *elem(mid)*

if *key(mid)* < *k* **then return** *BinarySearch(S, k, mid + 1, high)*

if *key(mid)* > *k* **then return** *BinarySearch(S, k, low, mid -1)*

low ↓ (index 0) *high* ↓ (index 9)

S										
	11	18	22	34	41	52	54	63	68	74
	0	1	2	3	4	5	6	7	8	9

Ex. Binary Search: Find k = 52

Algorithm BinarySearch(S, k, low, $high$):

  if $low > high$ **then return** *NO_SUCH_KEY*

  mid $\leftarrow \lfloor (low + high) / 2 \rfloor$

  if *key(mid) = k* **then return** *elem(mid)*

  if *key(mid) < k* **then return** *BinarySearch(S, k, mid + 1, high)*

  if *key(mid) > k* **then return** *BinarySearch(S, k, low, mid -1)*

	low				*mid*					*high*
S	11	18	22	34	41	52	54	63	68	74
	0	1	2	3	4	5	6	7	8	9

Ex. Binary Search: Find k = 52

Algorithm BinarySearch(*S*, *k*, *low*, *high*):

    if *low* > *high* **then return** *NO_SUCH_KEY*

    mid ← $\lfloor$(*low* + *high*) / 2$\rfloor$

    if *key(mid) = k* **then return** *elem(mid)*

    if *key(mid) < k* **then return** *BinarySearch(S, k, mid + 1, high)*

    if *key(mid) > k* **then return** *BinarySearch(S, k, low, mid -1)*

						low		*mid*		*high*
S	11	18	22	34	41	52	54	63	68	74
	0	1	2	3	4	5	6	7	8	9

Ex. Binary Search: Find *k* = 52

Algorithm BinarySearch(*S*, *k*, *low*, *high*):

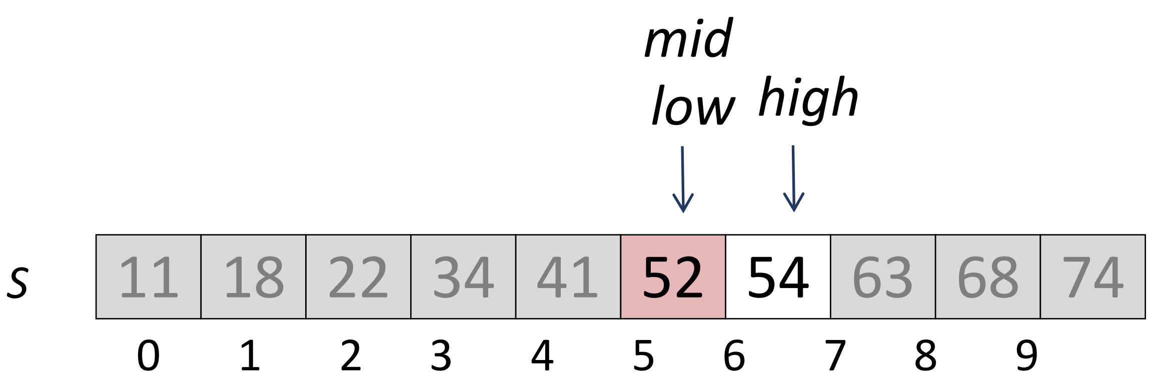
  if *low* > *high* **then return** *NO_SUCH_KEY*

  mid $\leftarrow \lfloor (low + high) / 2 \rfloor$

  if *key(mid) = k* **then return** *elem(mid)*

  if *key(mid)* < *k* **then return** *BinarySearch(S, k, mid + 1, high)*

  if *key(mid)* > *k* **then return** *BinarySearch(S, k, low, mid -1)*



Binary Search

Algorithm BinarySearch(*S*, *k*, *low*, *high*):

if *low* > *high* **then return** *NO_SUCH_KEY*

mid ← $\lfloor(low + high) / 2\rfloor$

if *key(mid) = k* **then return** *elem(mid)*

if *key(mid) < k* **then return** *BinarySearch(S, k, mid + 1, high)*

if *key(mid) > k* **then return** *BinarySearch(S, k, low, mid -1)*

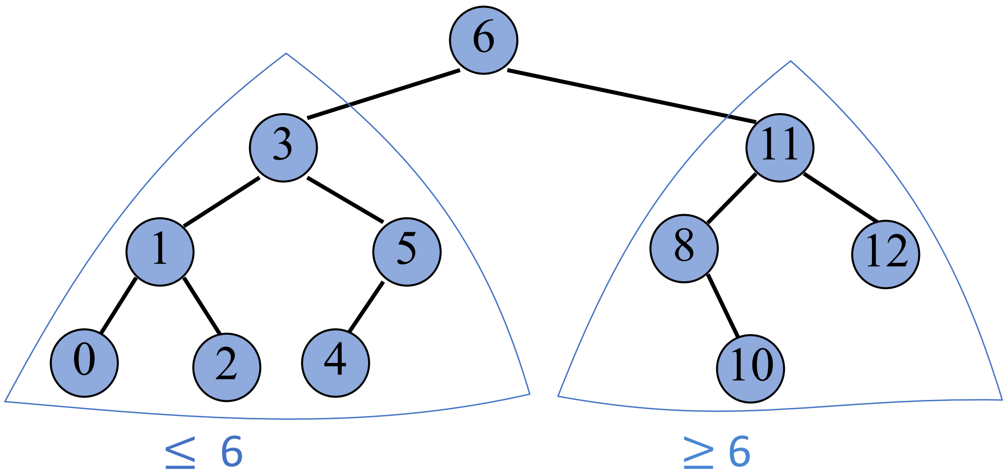
Each successive call to BinarySearch halves the input, so the running time is ***O*****(log*n*)**

Now … Binary Search Trees (BSTs)

They are trees! Not arrays.

Binary Search Tree (BST)

- An implementation of an **ordered dictionary**
  - We can search for an item based on its key
  - Keys have some inherent order to them
- A **binary search tree** is a binary tree where each internal node stores a (key, element)-pair, and
  - each element in the left subtree is smaller than or equal to the root
  - each element in the right subtree is larger than or equal to the root
  - the left and right subtrees are binary search trees
- An inorder traversal visits items in ascending order



BST – Tree-Insert(T, k)

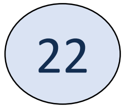
- Idea: find a free spot in the tree and add a node which stores that item *k*
- Strategy
  - start at root *r*
  - if $k < \text{key}(r)$, continue in left subtree
  - otherwise, continue in right subtree
- Runtime is $O(\boldsymbol{h})$, where $\boldsymbol{h}$ is the height of the tree

- Ex: Insert the numbers 22, 80, 18, 9, 90, 20.

BST – Tree-Insert(T, k)

- Idea: find a free spot in the tree and add a node which stores that item *k*
- Strategy
  - start at root *r*
  - if *k* < key(*r*), continue in left subtree
  - otherwise, continue in right subtree
- Runtime is *O*(***h***), where ***h*** is the height of the tree

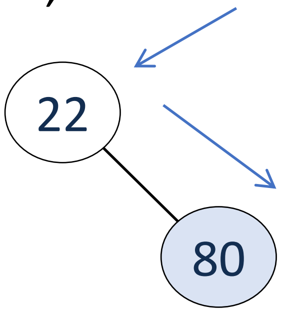
- Ex: Insert the numbers 22, 80, 18, 9, 90, 20.



BST – Tree-Insert(T, k)

- Idea: find a free spot in the tree and add a node which stores that item *k*
- Strategy
    - start at root *r*
    - if *k* < key(*r*), continue in left subtree
    - otherwise, continue in right subtree
- Runtime is *O*(***h***), where ***h*** is the height of the tree

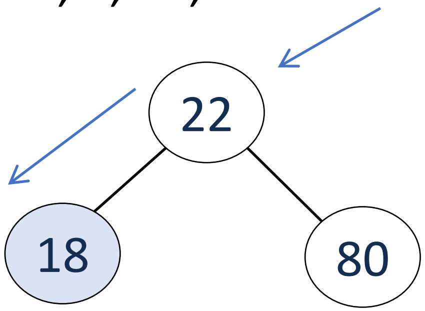
- Ex: Insert the numbers 22, 80, 18, 9, 90, 20.



BST – Tree-Insert(T, k)

- Idea: find a free spot in the tree and add a node which stores that item *k*
- Strategy
  - start at root *r*
  - if *k* < key(*r*), continue in left subtree
  - otherwise, continue in right subtree
- Runtime is *O*(***h***), where ***h*** is the height of the tree

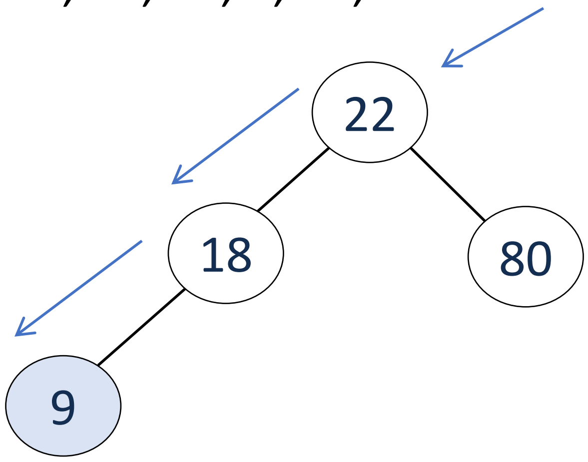
- Ex: Insert the numbers 22, 80, 18, 9, 90, 20.



BST – Tree-Insert(T, k)

- Idea: find a free spot in the tree and add a node which stores that item *k*
- Strategy
  - start at root *r*
  - if *k* < key(*r*), continue in left subtree
  - otherwise, continue in right subtree
- Runtime is *O*(***h***), where ***h*** is the height of the tree

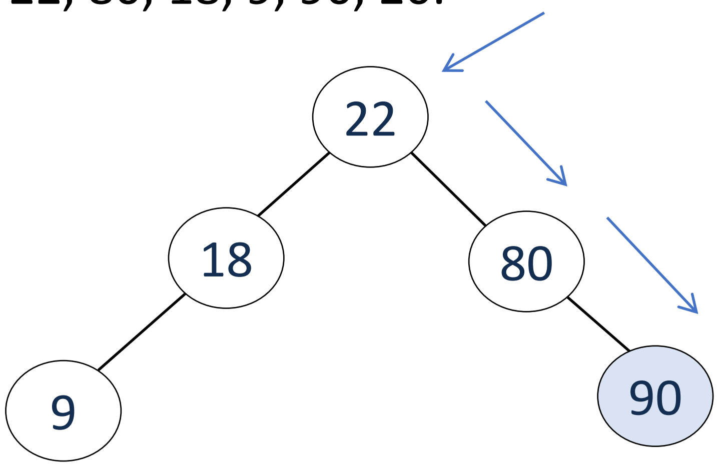
- Ex: Insert the numbers 22, 80, 18, 9, 90, 20.



BST – Tree-Insert(T, k)

- Idea: find a free spot in the tree and add a node which stores that item *k*
- Strategy
  - start at root *r*
  - if *k* < key(*r*), continue in left subtree
  - otherwise, continue in right subtree
- Runtime is *O*(***h***), where ***h*** is the height of the tree

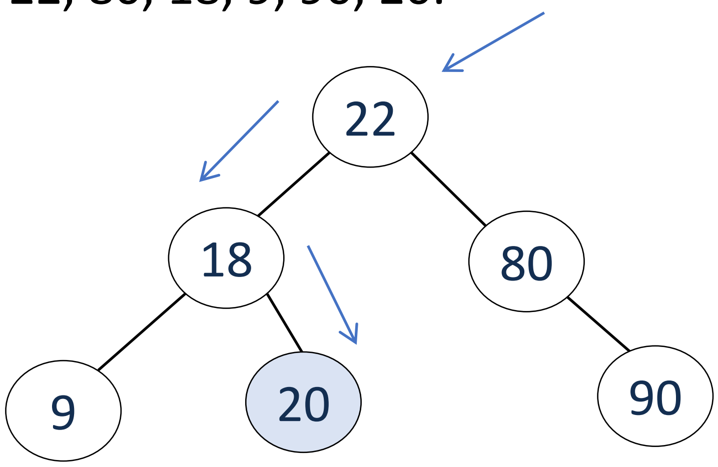
- Ex: Insert the numbers 22, 80, 18, 9, 90, 20.



BST – Tree-Insert(T, k)

- Idea: find a free spot in the tree and add a node which stores that item *k*
- Strategy
  - start at root *r*
  - if *k* < key(*r*), continue in left subtree
  - otherwise, continue in right subtree
- Runtime is *O*(***h***), where ***h*** is the height of the tree

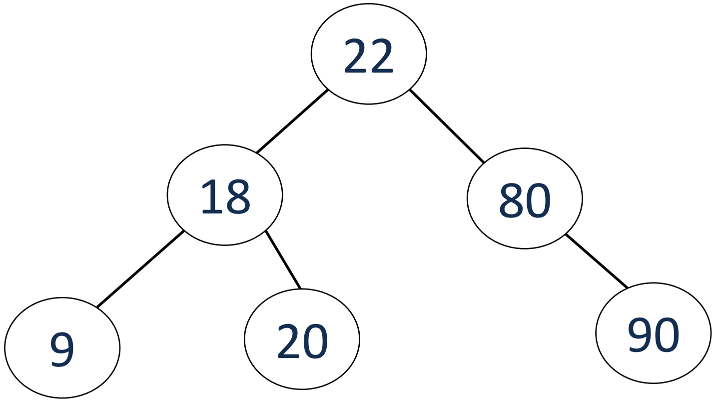
- Ex: Insert the numbers 22, 80, 18, 9, 90, 20.



BST – Tree-Insert(T, k)

- Idea: find a free spot in the tree and add a node which stores that item *k*
- Strategy
  - start at root *r*
  - if *k* < key(*r*), continue in left subtree
  - otherwise, continue in right subtree
- Runtime is *O*(***h***), where ***h*** is the height of the tree

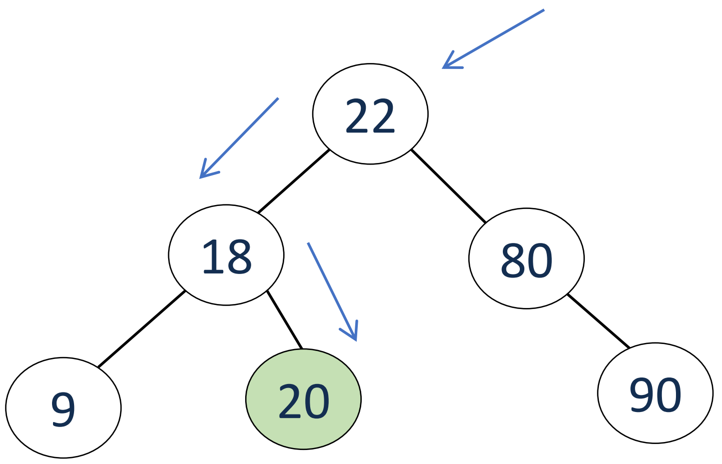
- Ex: Insert the numbers 22, 80, 18, 9, 90, 20.



BST – Tree-Search(T, k)

- Idea: find item *k*
- Strategy
  - start at root *r*
  - if *k* = key(*r*), return *r*
  - if *k* < key(*r*), continue in left subtree
  - if *k* > key(*r*), continue in right subtree
- Runtime is $O(\boldsymbol{h})$, where ***h*** is the height of the tree

- Ex: Find 20.



BST – Tree-Delete(T, k)

- Idea: remove item *k*

- Strategy: let *z* be the position of Tree-Search(T, *k*). Remove *z* without creating "holes" in the tree

  - **Case 1: *z* has at most one child** (easier: removing z creates easily filled hole)
    - Replace z with subtree rooted at child

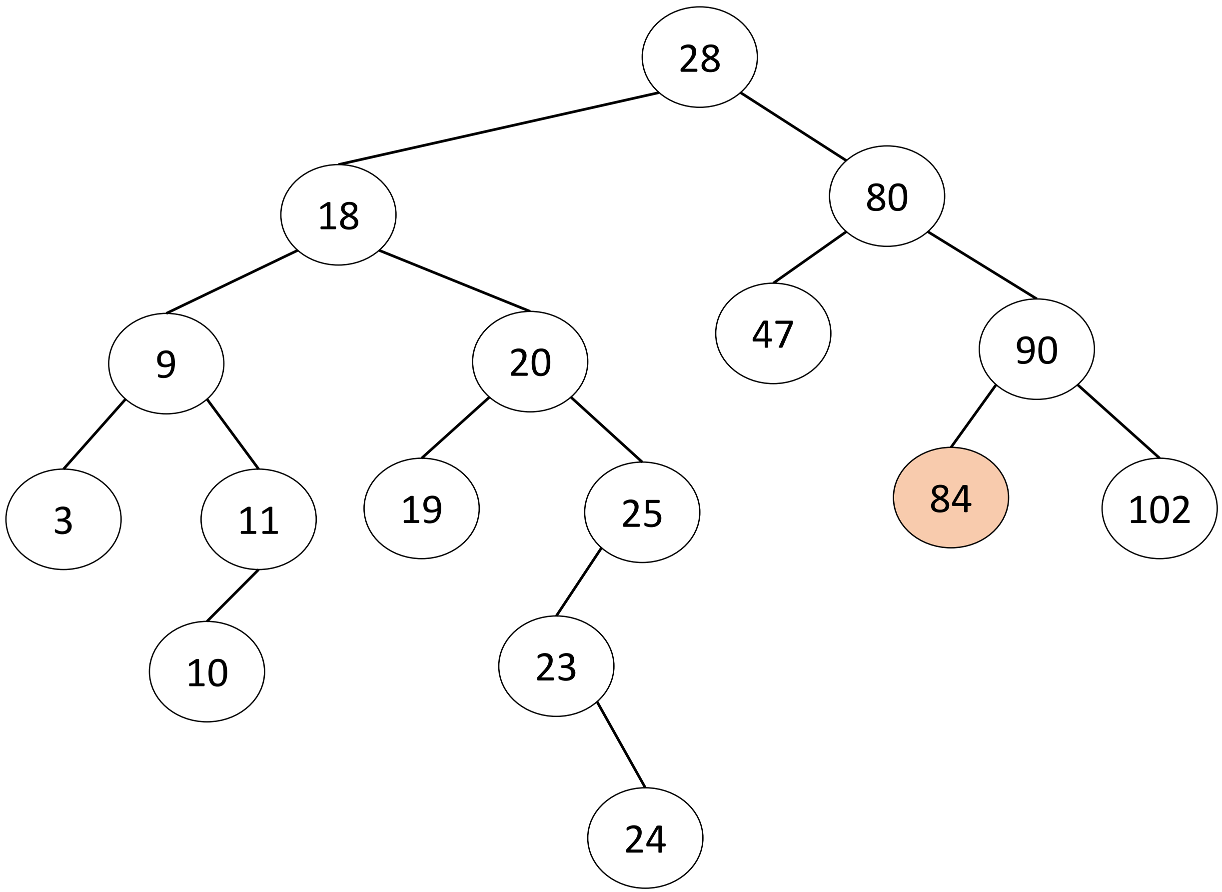
  - **Case 2: *z* has two children** (harder: removing z creates holes)
    - Let y be the next node that follows in an inorder traversal
    - y is guaranteed to be a leaf node (it is the leftmost node in the right subtree of z)
    - Swap z and y
    - Remove z

- Runtime is *O*(***h***), where ***h*** is the height of the tree

BST – Tree-Delete(T, k)

Case 1(a): *z* has no children

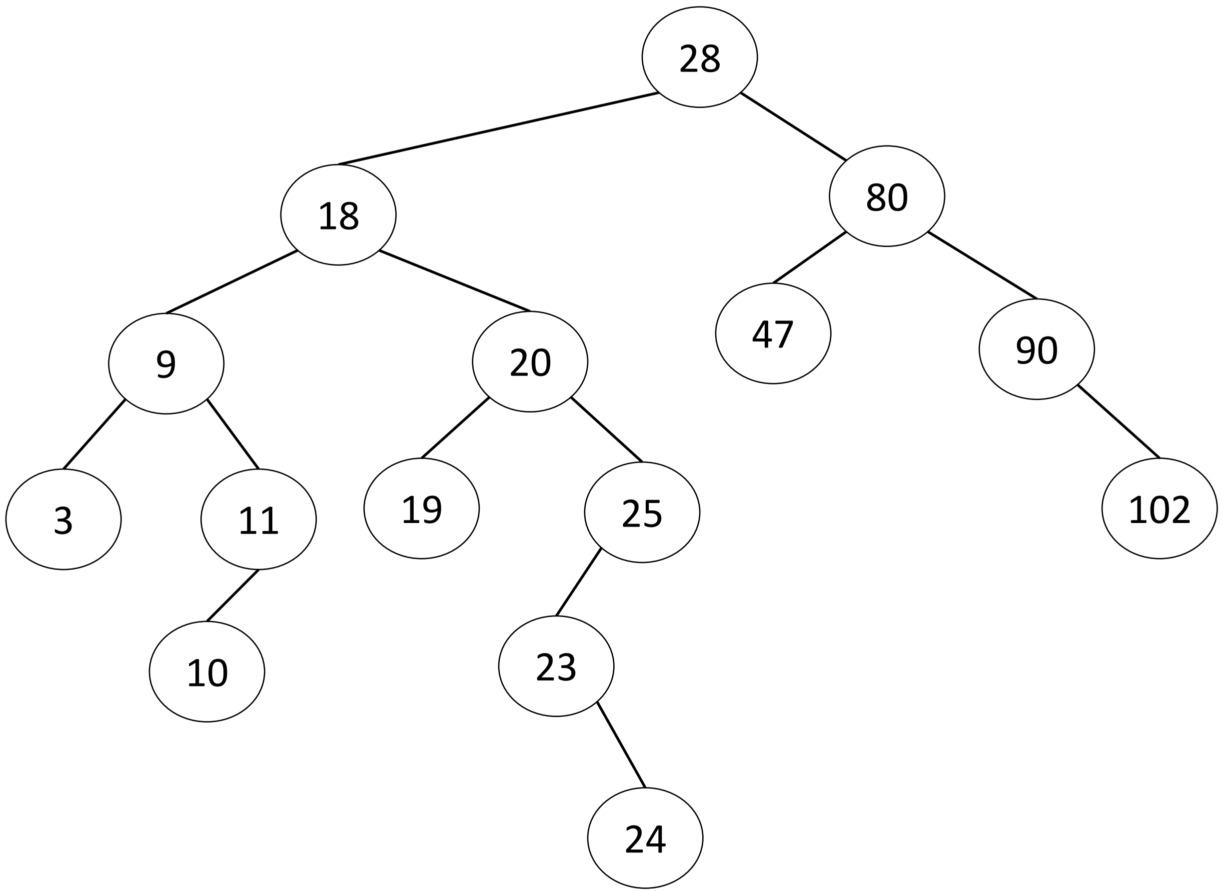
Ex: Delete 84



BST – Tree-Delete(T, k)

Case 1(a): *z* has no children

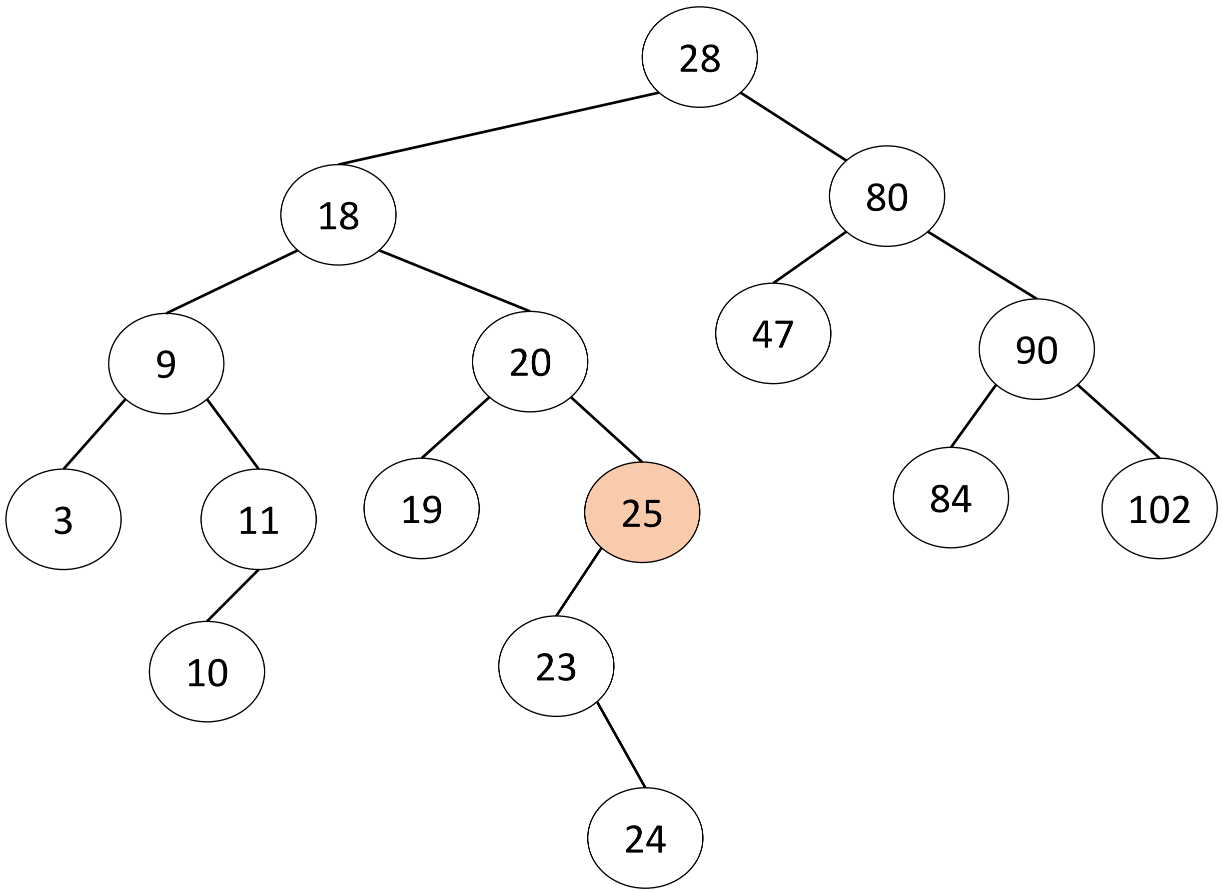
Ex: Delete 84



BST – Tree-Delete(T, k)

Case 1(b): *z* has one child

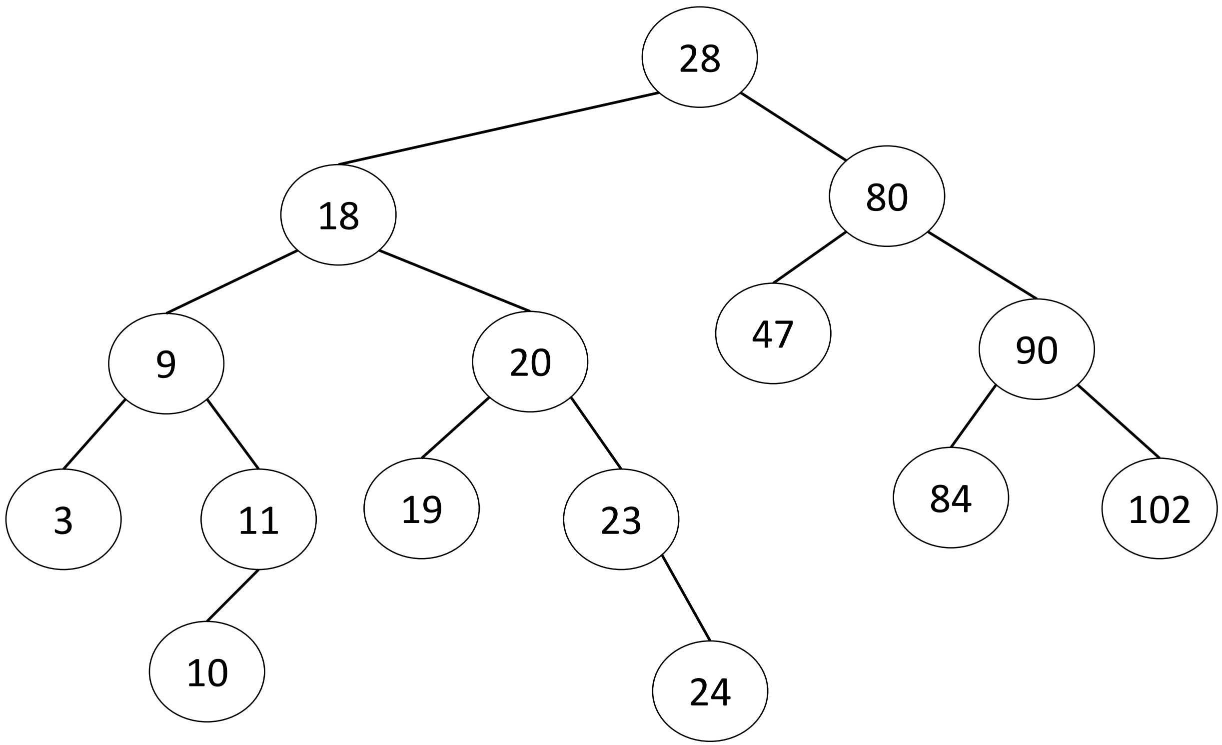
Ex: Delete 25



BST – Tree-Delete(T, k)

Case 1(b): ***z*** **has one child**

Ex: Delete 25



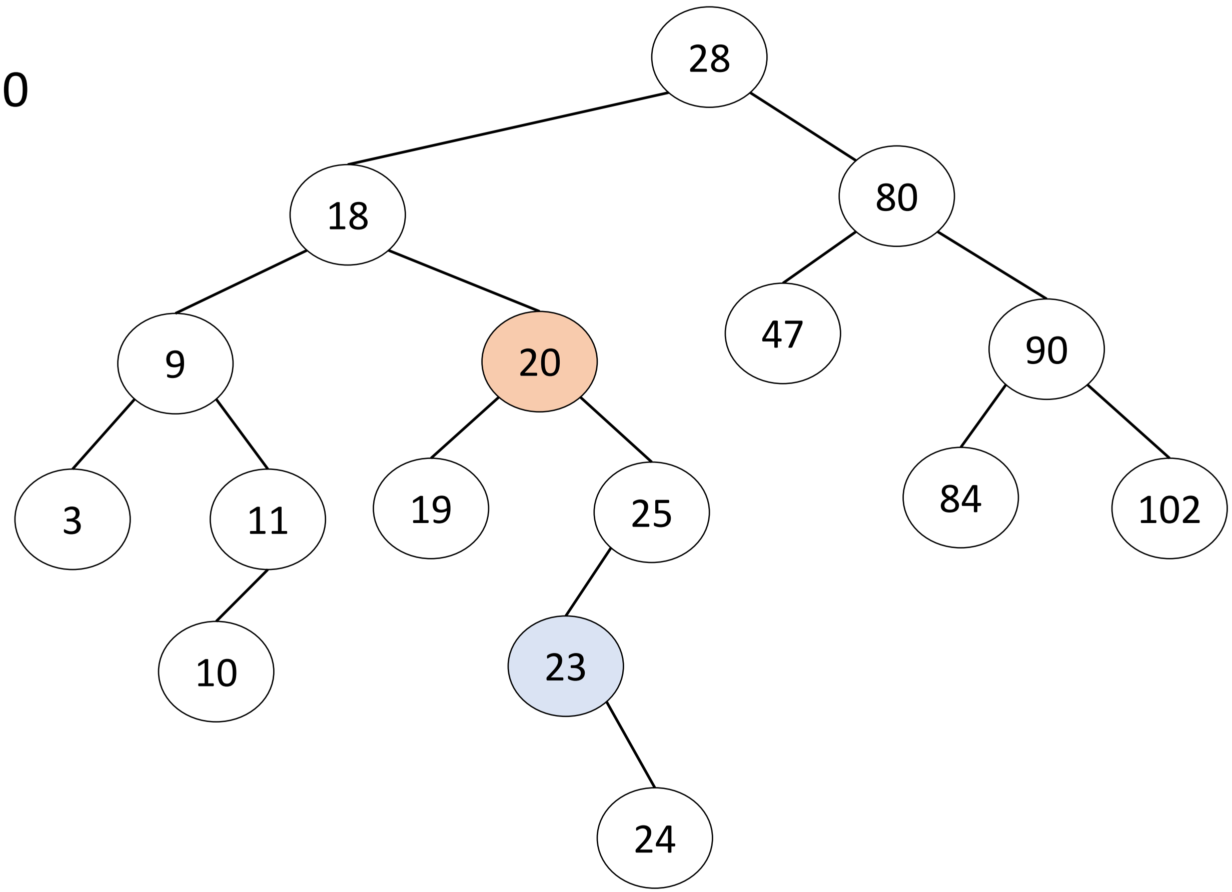
BST – Tree-Delete(T, k)

Case 2: *z* has two children

Find the first internal node *y* that follows *z* in an inorder traversal

Swap *z* and *y*; Remove z

Ex: Delete 20



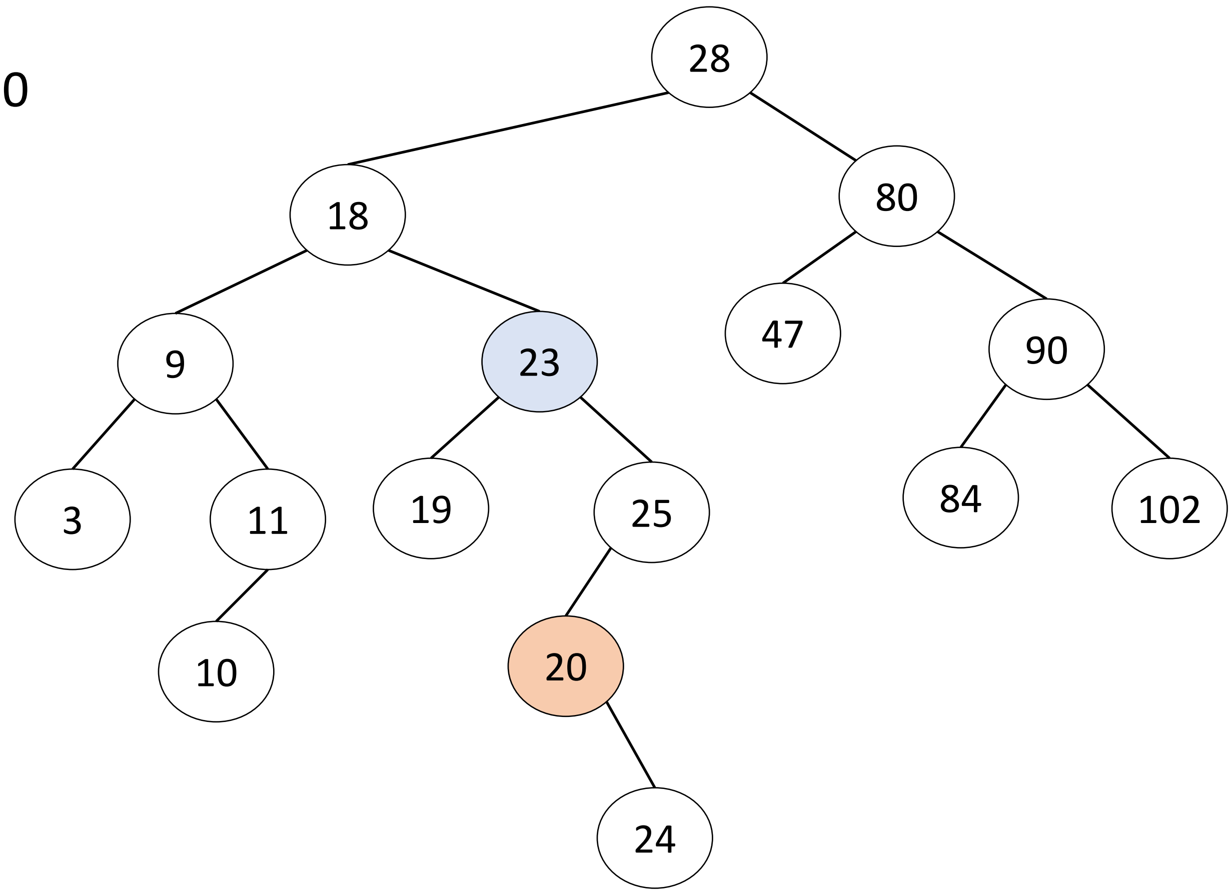
BST – Tree-Delete(T, k)

Case 2: *z* has two children

Find the first internal node *y* that follows *z* in an inorder traversal

Swap *z* and *y*; Remove z

Ex: Delete 20



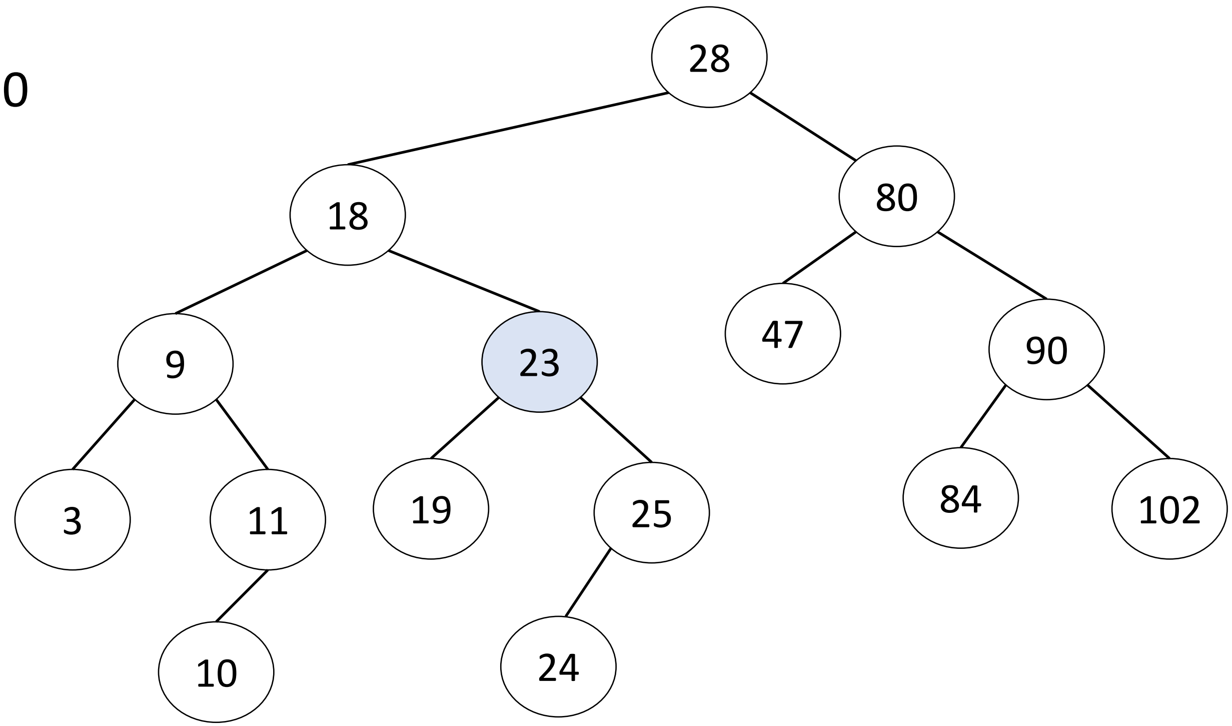
BST – Tree-Delete(T, k)

Case 2: *z* has two children

Find the first internal node *y* that follows *z* in an inorder traversal

Swap *z* and *y*; Remove z

Ex: Delete 20

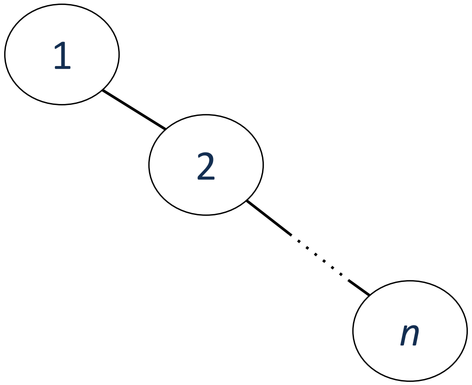


Performance of BST operations

- Space used for BST is O(n)
- Runtime of all operations is $O(h)$

What is h in the worst case?

- Consider inserting the sequence 1, 2, …, $n-1$, n
- Worst case height $h \in O(n)$.



How do we keep the tree balanced?

Other

- You are given two sorted integer arrays *A* and *B* such that no integer is contained twice in the same array. *A* and *B* are nearly identical. However, *B* is missing exactly one number. Find the missing number in *B.*

- You are given a sorted array *A* of distinct integers. Determine whether there exists an index *i* such that A[*i*] = *i*.