Searching Methods in Data Structure

Search Types

- What is Searching?
  - Searching is the process of finding a given value position in a list of values.
  - It decides whether a search key is present in the data or not.

- **To search an element in a given array, it can be done in following ways:**
  - (Linear) Sequential Search
  - Binary Search
- **To search in an element in a binary tree we can use:**
  - Binary Search Tree

Linear Search

- In a linear search, the search key is compared with each element of the array linearly. If there is a match, it returns the index of the array between otherwise, it returns -1
- Linear search has complexity of O(n).

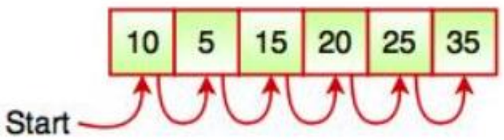
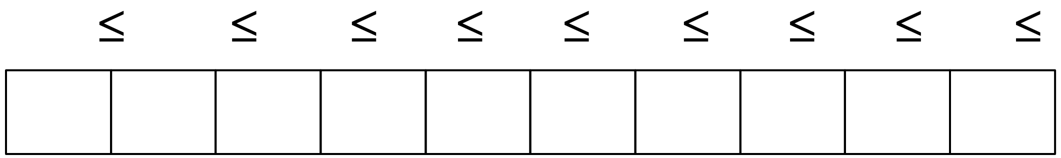


Fig. Sequential Search

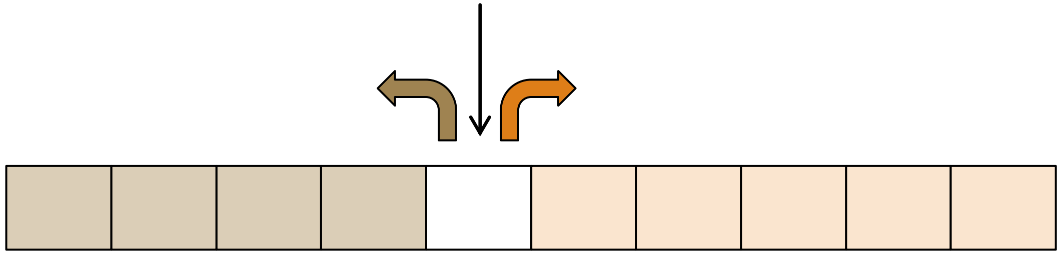
Binary Search

- Items are ordered in a sorted sequence
- Find an element *k*



Binary Search

- Items are ordered in a sorted sequence
- 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

Binary Search: Find *k* = 52

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

- **if** *low* > *high* **then return** *NO_SUCH_KEY*
- *mid* ← ⌊(*low* + *high*) / 2⌋
- **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 ↓									*high* ↓
S	11	18	22	34	41	52	54	63	68	74
	0	1	2	3	4	5	6	7	8	9

Binary Search: Find *k* = 52

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

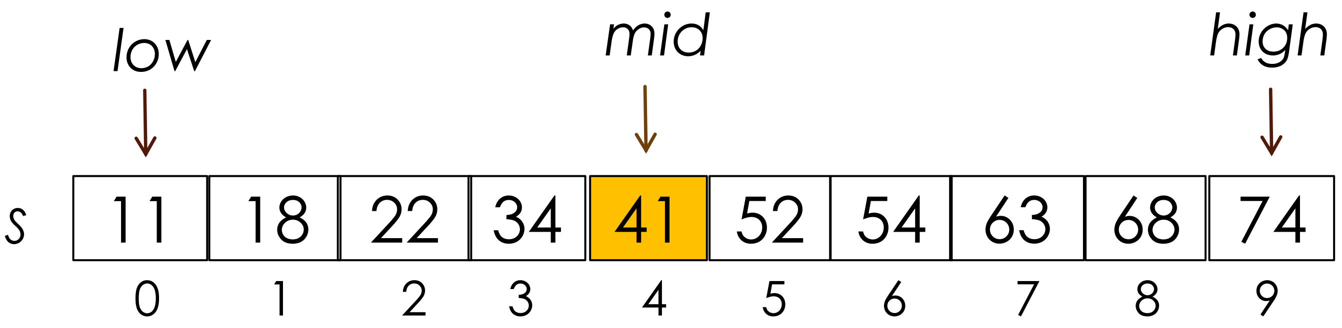
if *low* > *high* **then return** *NO_SUCH_KEY*

mid ← ⌊(*low* + *high*) / 2⌋

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

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

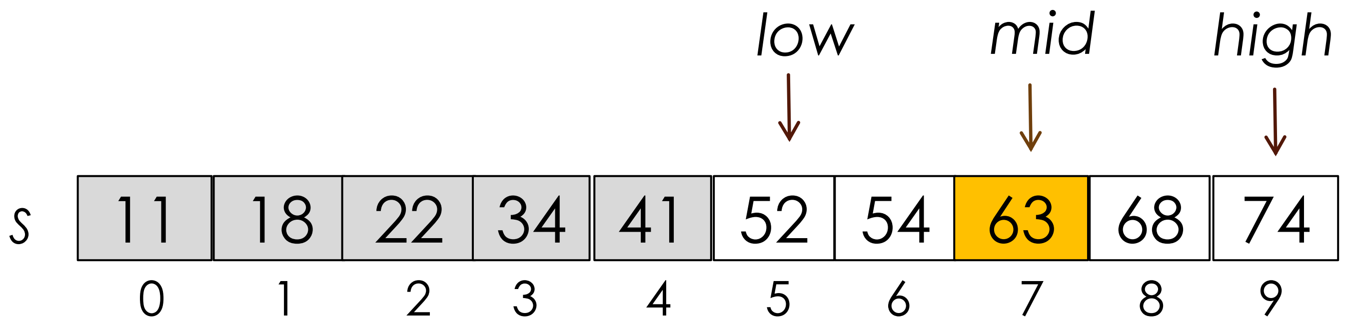
if *key(mid)* > *k* **then return** *BinarySearch(S, k, low, mid -1)*



Binary Search: Find *k* = 52

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

- **if** *low* > *high* **then return** *NO_SUCH_KEY*
- *mid* ← ⌊(*low* + *high*) / 2⌋
- **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: Find k = 52

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

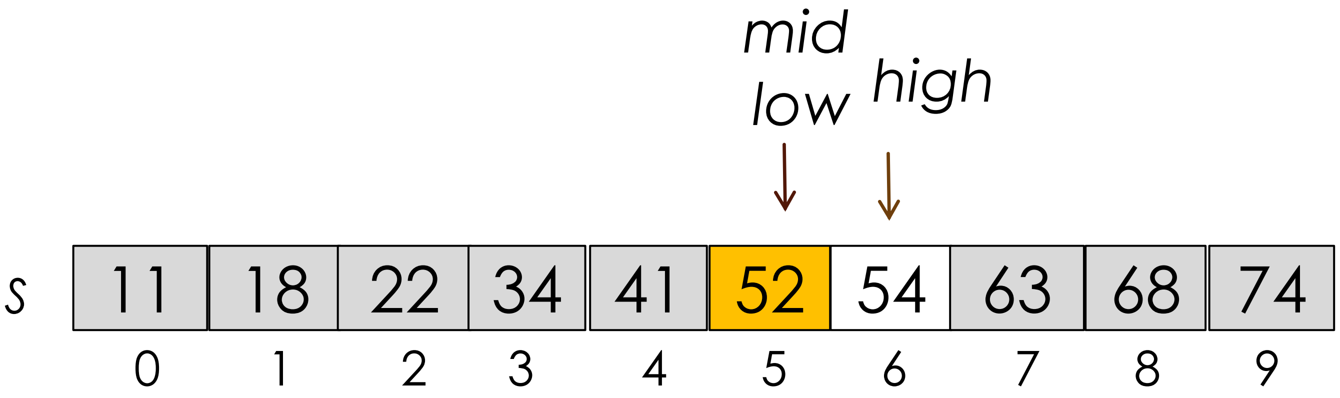
if *low* > *high* **then return** *NO_SUCH_KEY*

mid ← ⌊(*low* + *high*) / 2⌋

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 ← ⌊(*low* + *high*) / 2⌋

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*)**

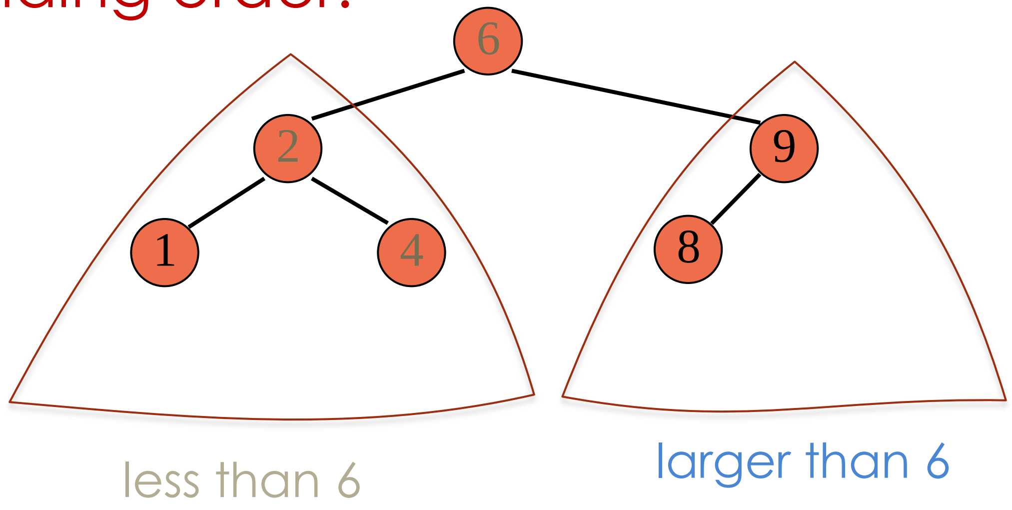
Exercises

- **Write the iterative** Implementation (Pseudocode) of binary search?

Binary Search Tree

Binary Search Tree

- 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 the root
  - each element in the right subtree is larger than the root
  - the left and right subtrees are binary search trees
- An inorder traversal visits items in ascending order.

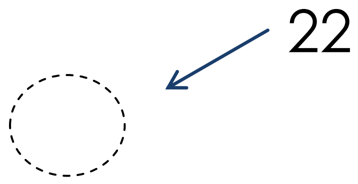


BST – Insert(k, v)

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

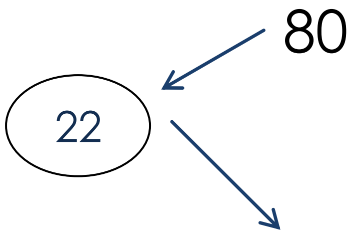
BST – Insert Example

Insert the numbers 22, 80, 18, 9, 90, 20.



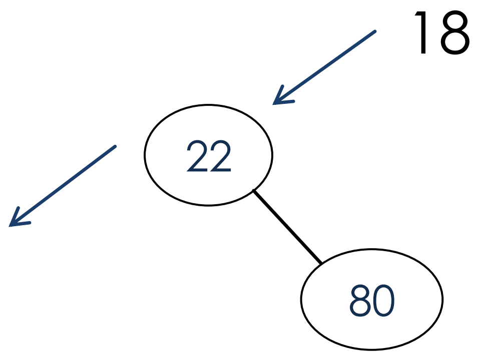
BST – Insert Example

Insert the numbers 22, 80, 18, 9, 90, 20,19.



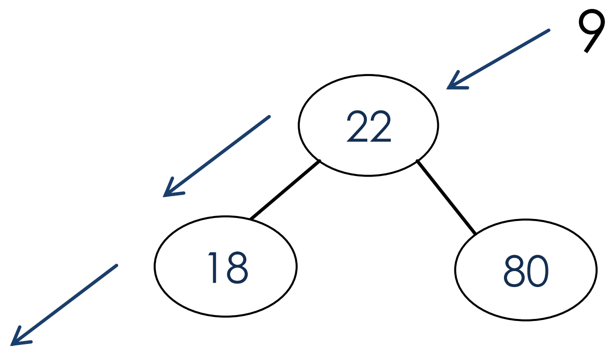
BST – Insert Example

Insert the numbers 22, 80, 18, 9, 90, 20, 19.



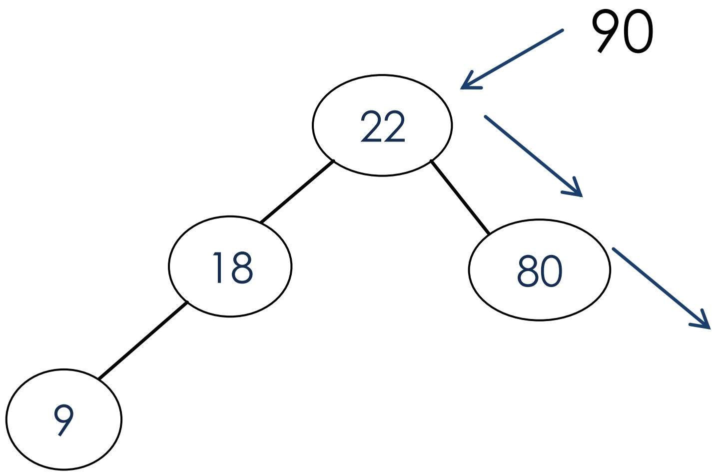
BST – Insert Example

Insert the numbers 22, 80, 18, 9, 90, 20, 19.



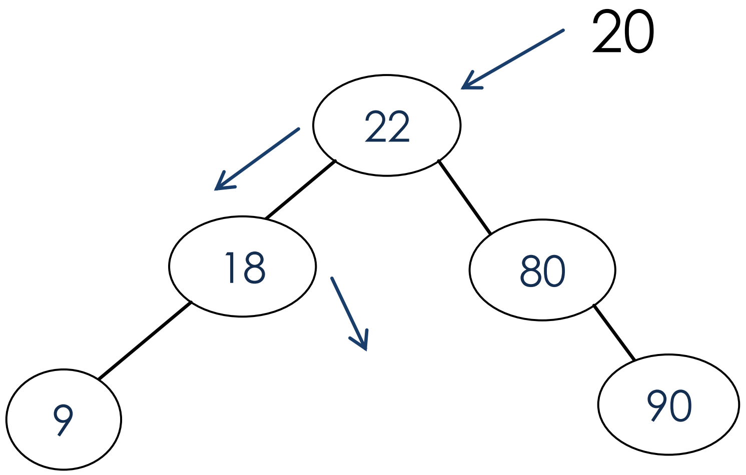
BST – Insert Example

Insert the numbers 22, 80, 18, 9, 90, 20, 19.



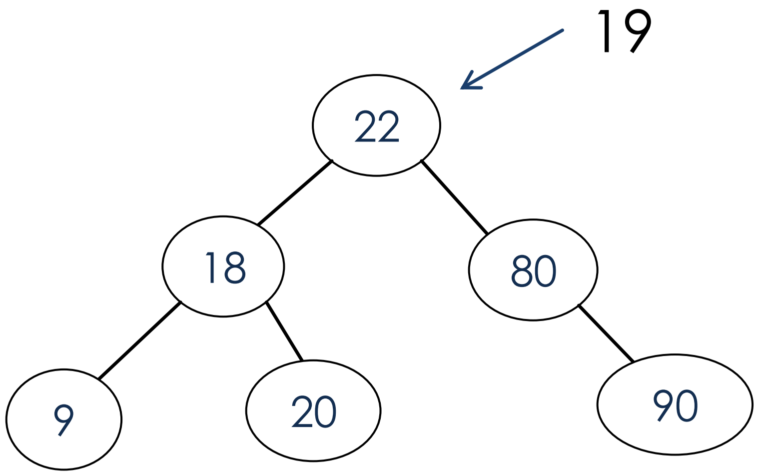
BST – Insert Example

Insert the numbers 22, 80, 18, 9, 90, 20, 19.



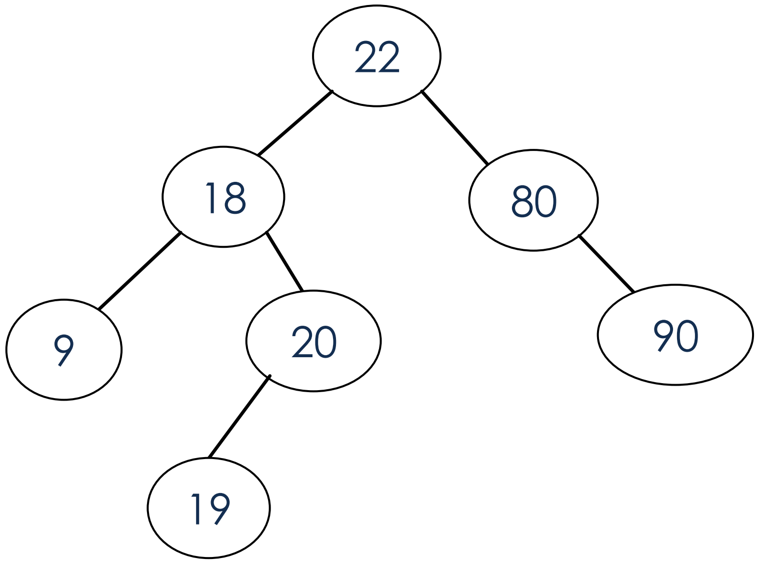
BST – Insert Example

Insert the numbers 22, 80, 18, 9, 90, 20, 19.



BST – Insert Example

Insert the numbers 22, 80, 18, 9, 90, 20, 19.



BST – Search (Find)

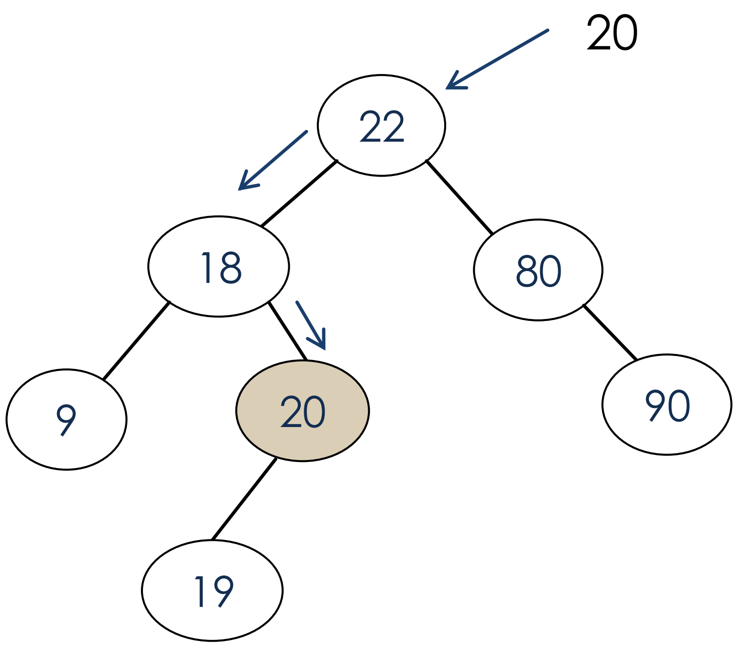
- Find the node with key *k*

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

BST – Find Example

Find the number 20



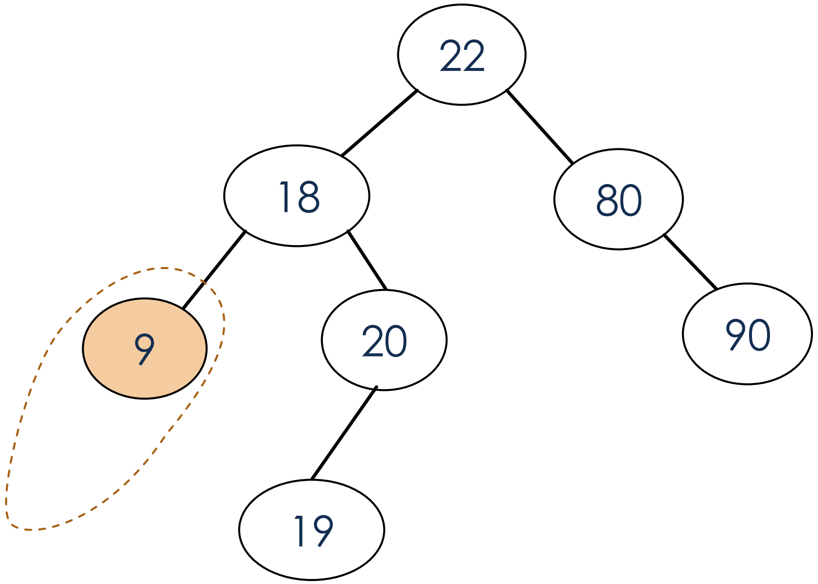
BST - Delete

- Delete the node with key k

- Three cases to remove a node z:
  - Case 0: z has no children.
  - Case 1: z has a one child.
  - Case 2: z has two children.
- Runtime is $O(h)$, where h is the height of the tree

BST – Delete Example

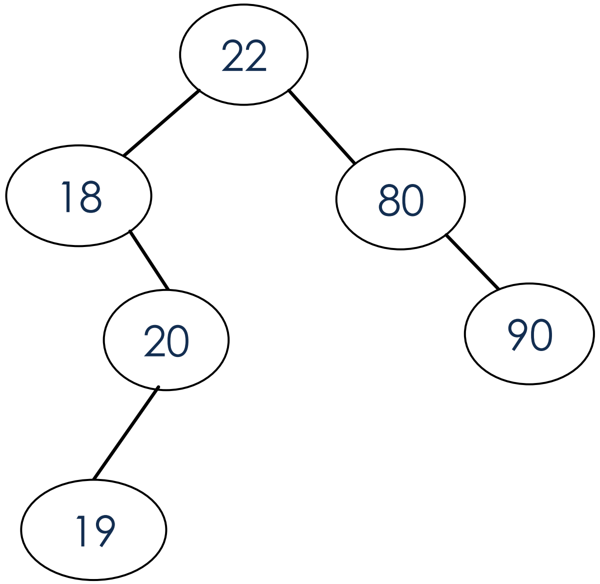
- **Case 0:** z has no children.
  - Simply just remove it.



For Example: Delete 9

BST – Delete Example

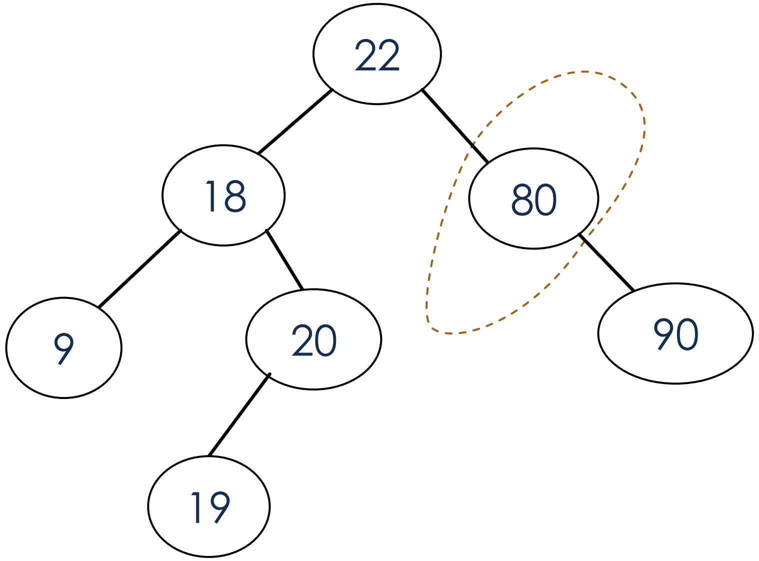
- **Case 0:** z has no children.
  - Simply just remove it.



For Example: Delete 9

BST – Delete Example

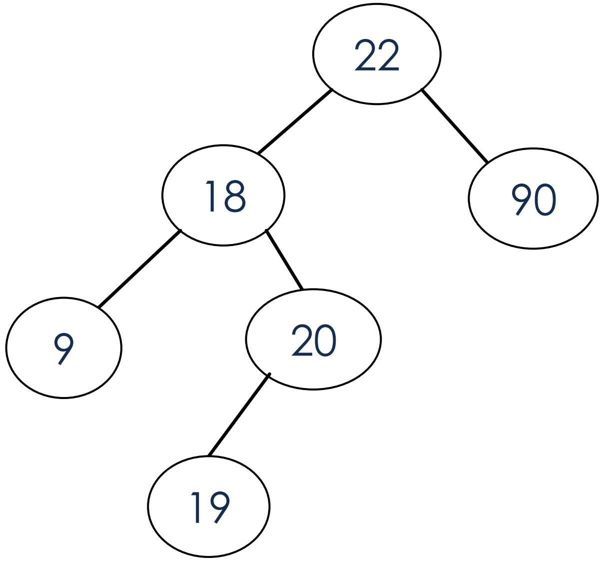
- **Case 1:** z has a one child.
  - If z has just one child, then we elevate that child to take z 's position in the tree by modifying z's parent to replace z by z's child.



For example: Delete 80 or Delete 20

BST – Delete Example

- **Case 1:** z has a one child.

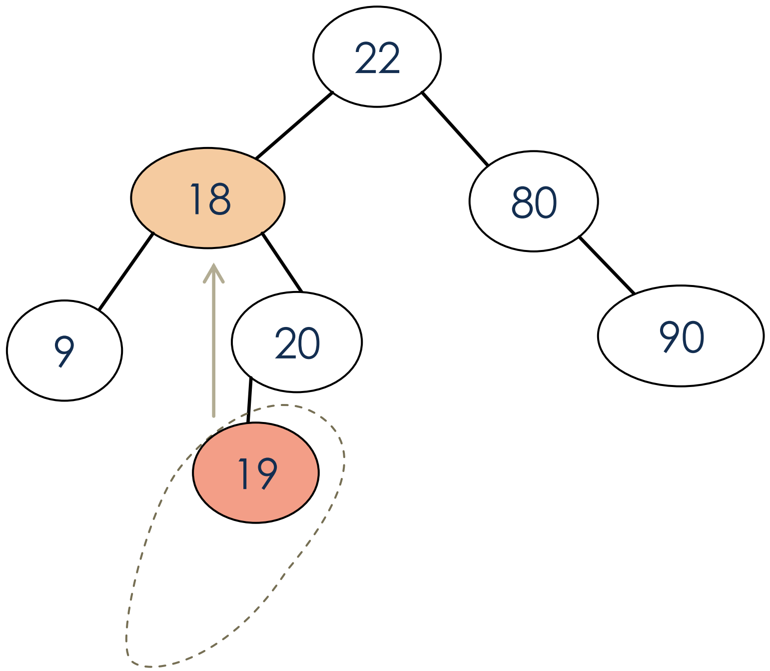


For example: Delete 80

BST – Delete Example

Case 2: z has two children

- Find the first node y that follows z in an inorder traversal.
- Replace z with y.



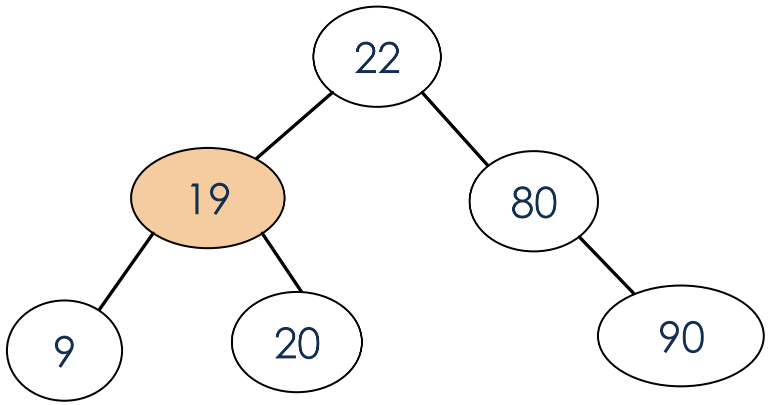
Inorder: 9 18 19 20 22 80 90

For example: Delete 18

BST – Delete Example

Case 2: z has two children

- Find the first node y that follows z in an inorder traversal.
- Replace z with y.



For example: Delete 18

Exercises

- 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*.
- Insert items with the following keys (in the given order) into an initially empty binary search tree: 30, 40, 24, 58, 48, 26, 1 1 , 1 3 . Draw the tree after each insertion.
- Given a binary search tree, which traversal type would print the values in the nodes in sorted order?

  a) Preorder

  b) Postorder

  c) Inorder

  d) None of the above