Ordered   Dictionaries

# Ordered Dictionaries

- Keys are ordered

- Perform usual dictionary operations (insertItem, removeItem, findElement) and maintain an order relation for the keys
  - we use an external comparator for keys

- New operations:
  - closestKeyBefore(k), closestElemBefore(k)
  - closestKeyAfter(k), closestElemAfter(k)

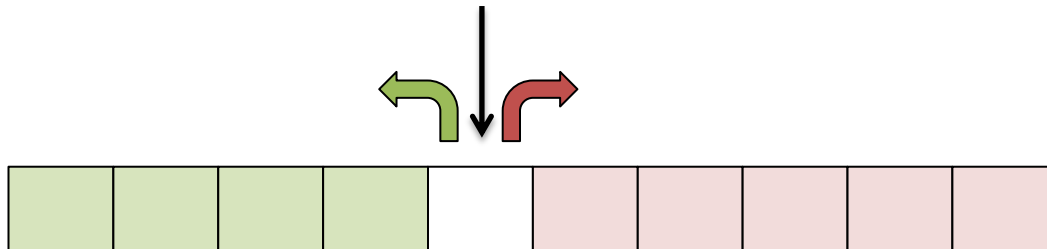- A special sentinel, NO_SUCH_KEY, is returned if no such item in the dictionary satisfies the query

Binary Search

- Items are ordered in a sorted sequence

- Find an element k

$$\leq \quad \leq \quad \leq \quad \leq \quad \leq \quad \leq \quad \leq \quad \leq \quad \leq$$
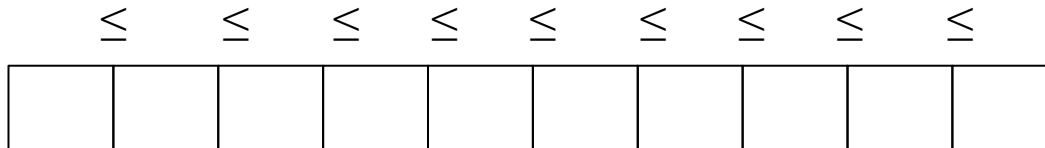
Binary Search

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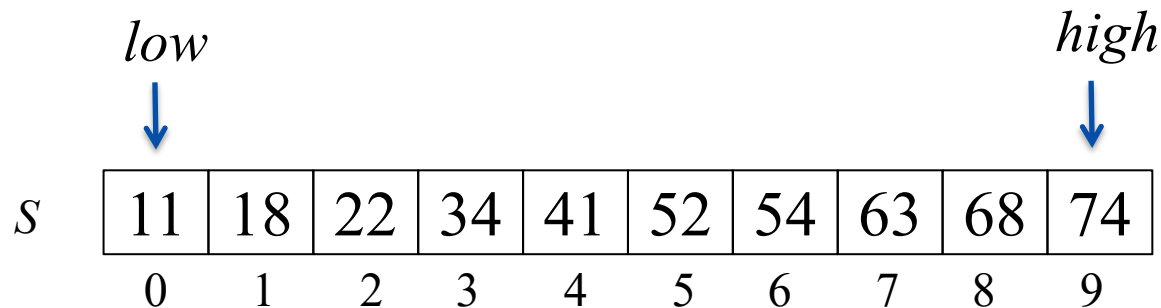
Algorithm BinarySearch(*S*, *k*, *low*, *high*):

    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)*
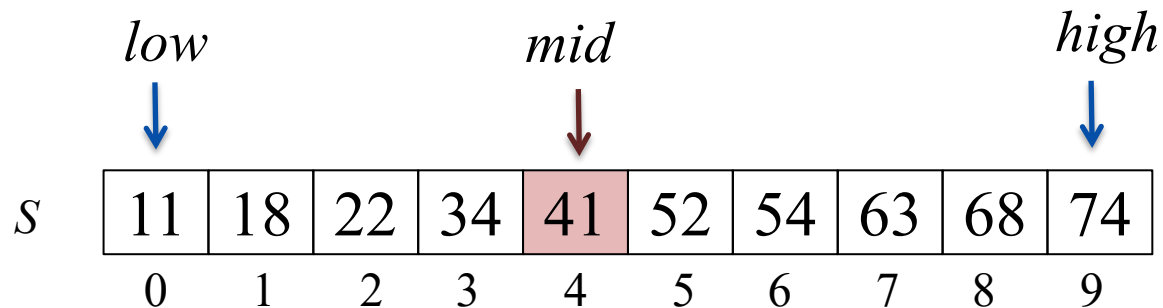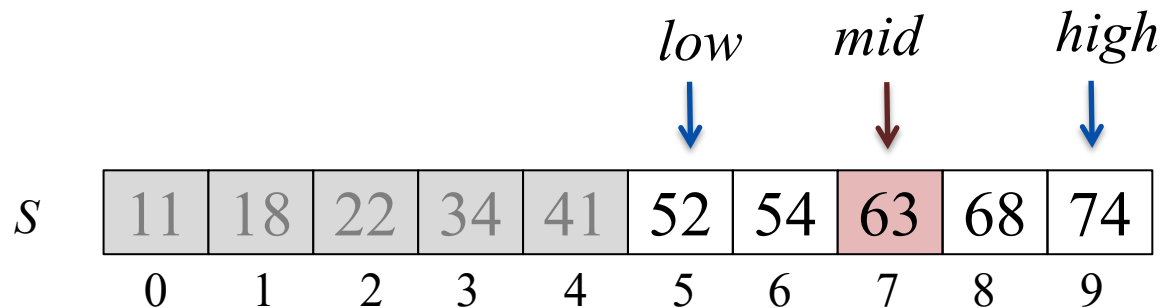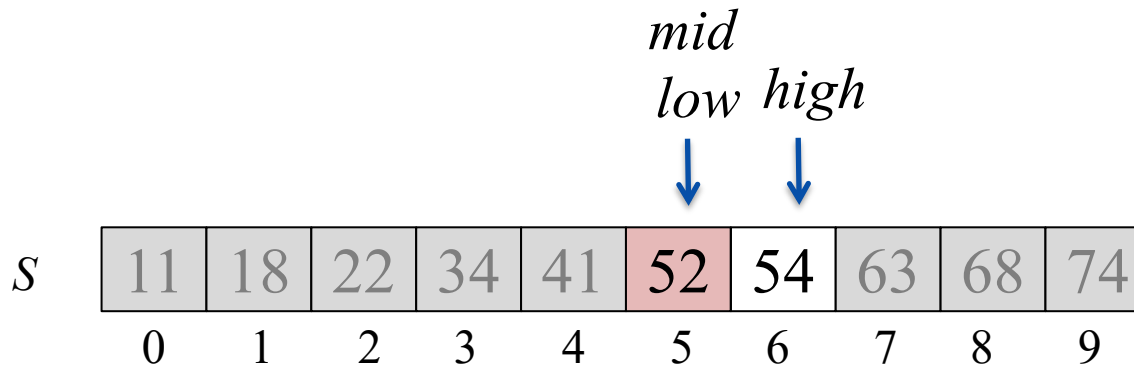
Binary Search: Find *k* = 52

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)*

	low		*mid*		*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)*

mid
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

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*)**

Lookup Table

- A dictionary implemented by means of an array-based sequence which is sorted by key
  - why use an array-based sequence rather than a linked list?

- Performance:
  - insertItem takes $O(n)$ time to make room by shifting items
  - removeItem takes $O(n)$ time to compact by shifting items
  - findElement takes $O(\log n)$ time, using binary search

- Effective only for
  - small dictionaries, or
  - when searches are the most common operations, while insertions and removals are rarely performed

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

less than 6   larger than 6
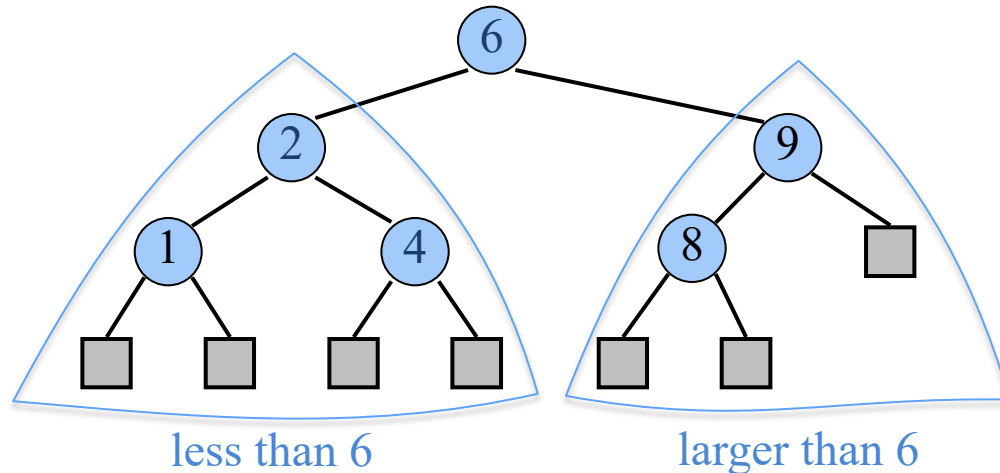
BST – Insert(k, v)

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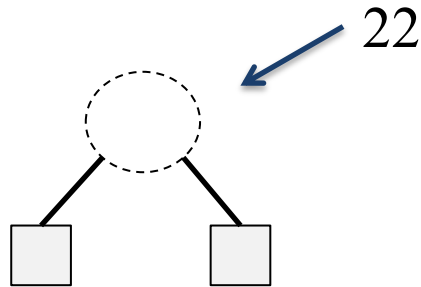
Insert the numbers 22, 80, 18, 9, 90, 20.

22

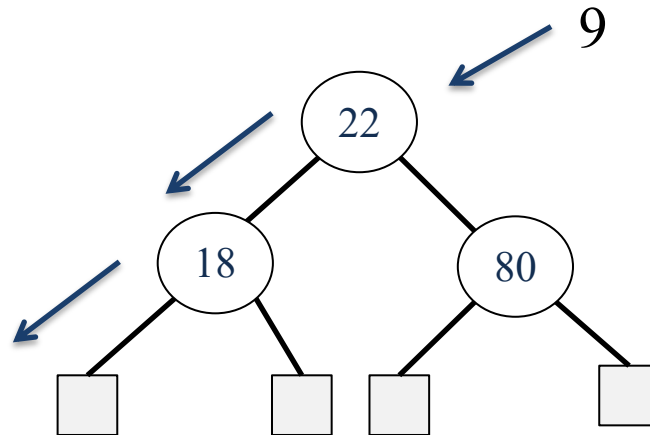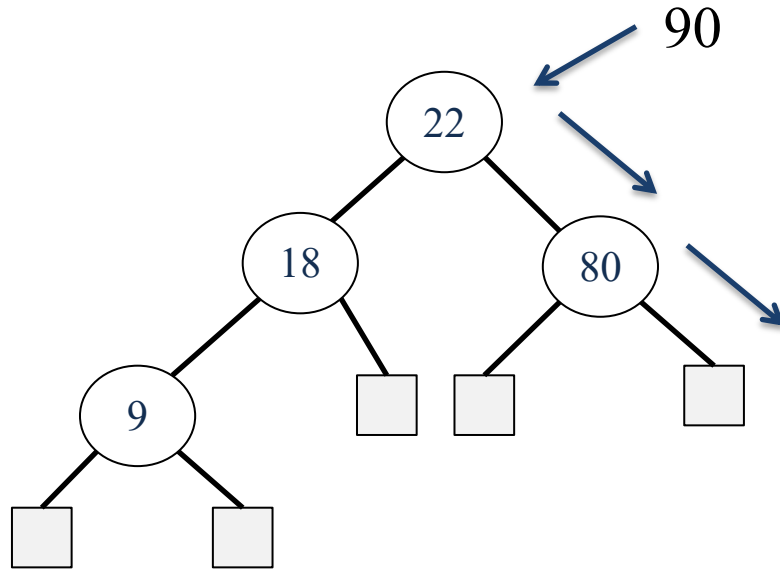# BST – Insert Example

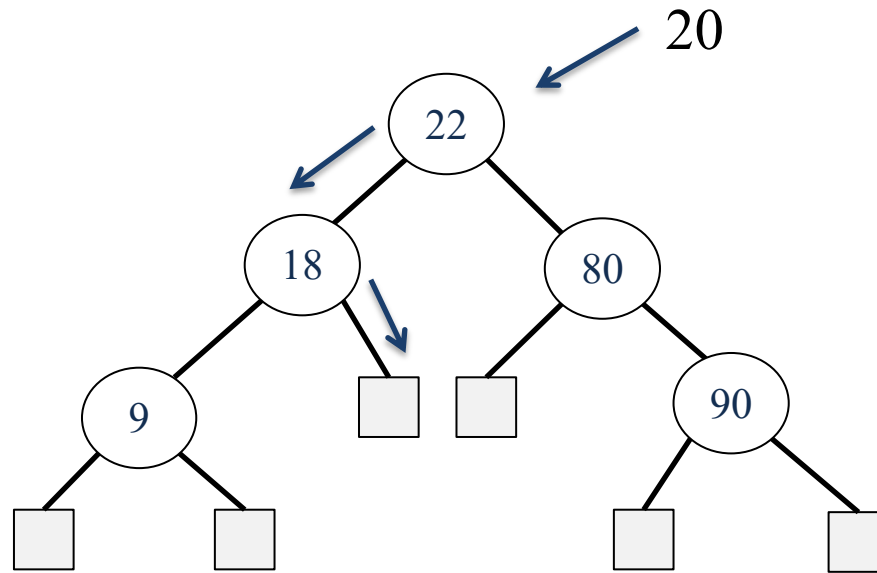Insert the numbers 22, 80, 18, 9, 90, 20.

80

22
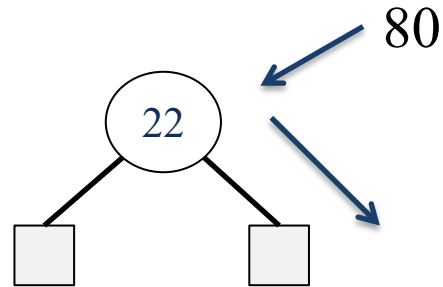
BST – Insert Example

Insert the numbers 22, 80, 18, 9, 90, 20.
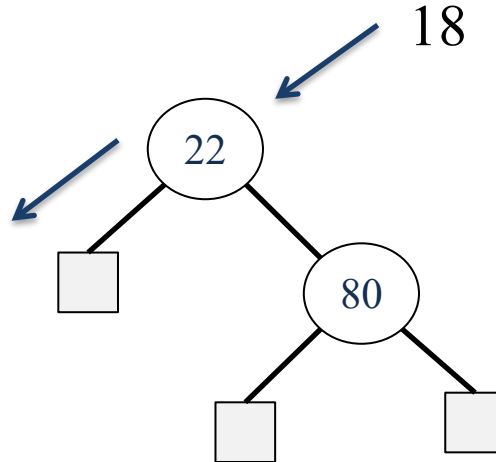
BST – Insert Example

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

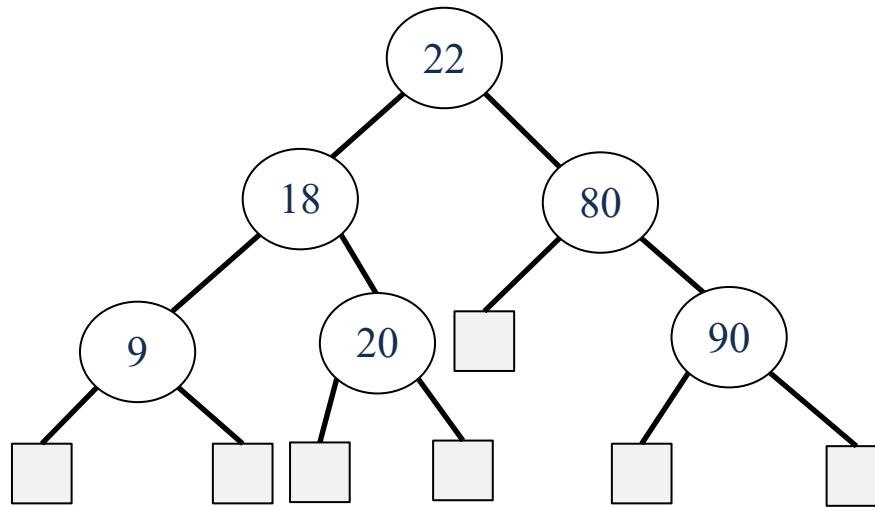# BST – Insert Example

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

BST – Insert Example

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

BST – Insert Example

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

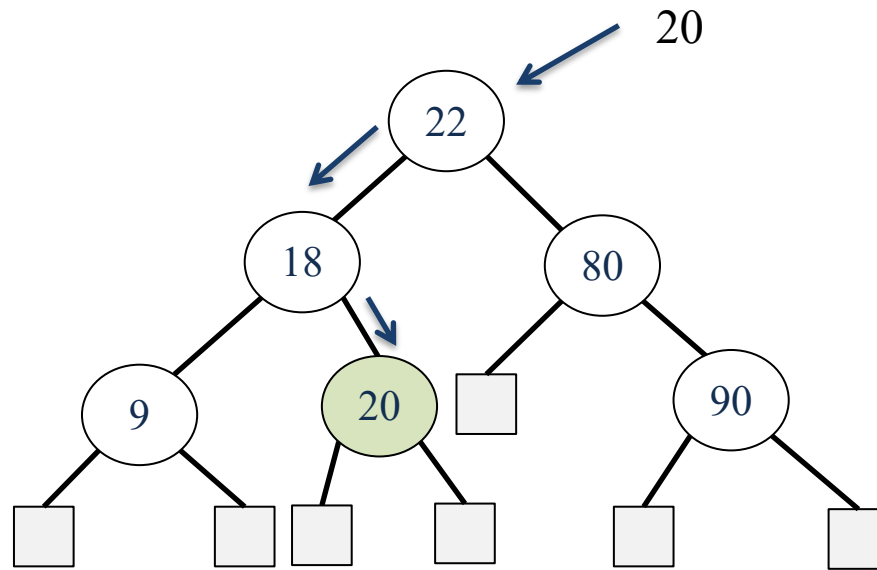# BST - Find

- Find the node with key k

- Strategy
  - start at root r
  - if $k = \text{key}(r)$, return r
  - if $k < \text{key}(r)$, continue in left subtree
  - if $k > \text{key}(r)$, 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*

- Strategy: let *n* be the position of FindElement(*k*)
  - Remove *n* without creating "holes" in the tree
  - Case 1: *n* has at least one child which is an external node
  - Case 2: *n* has two children with internal nodes

- Runtime is $O(h)$, where *h* is the height of the tree
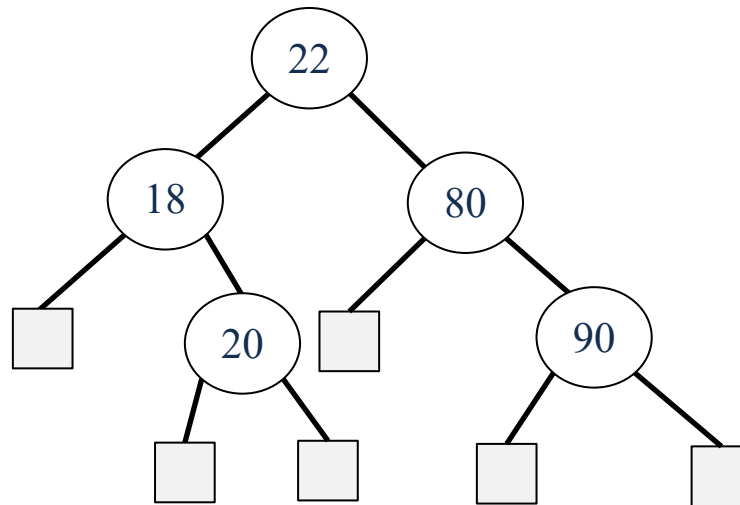
BST – Delete Example

Case 1(a): n has two children which are external nodes
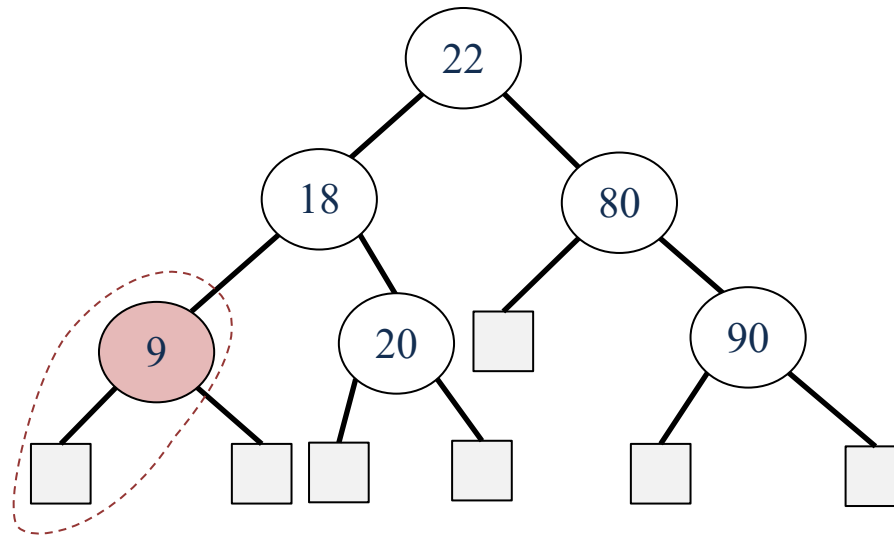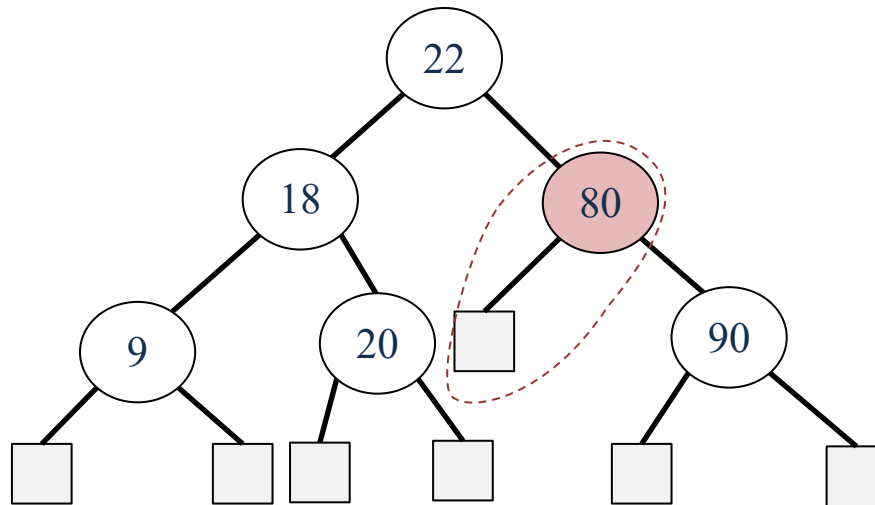

Delete 9

BST – Delete Example

Case 1(b): *n* has two children which are external nodes
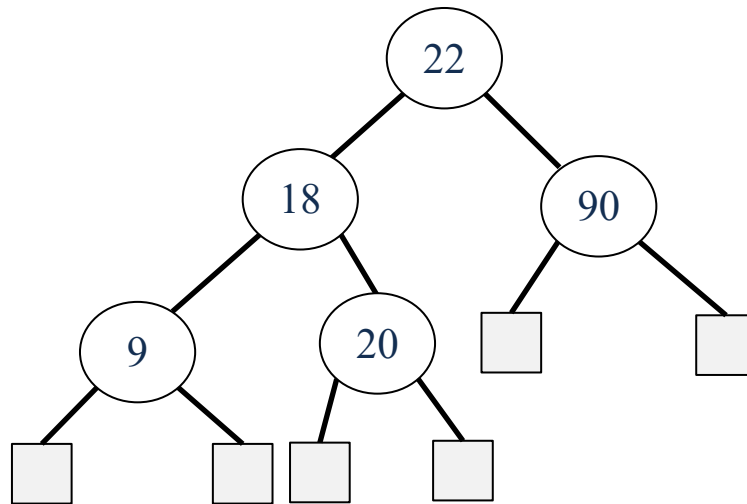

Delete 9

BST – Delete Example

Case 1: n has a child which is an internal node

Delete 80

BST – Delete Example
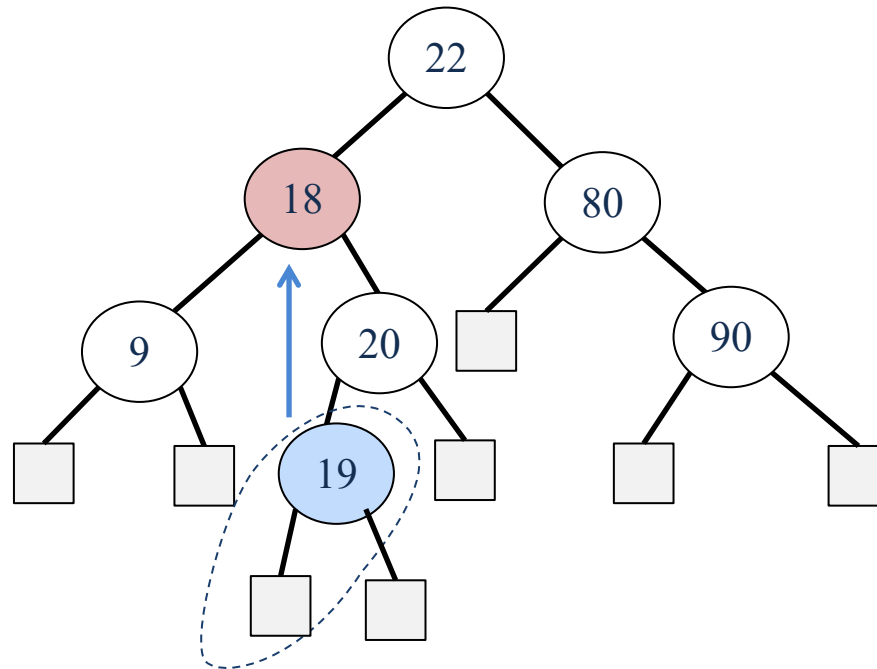
Case 1: *n* has a child which is an internal node

Delete 80

BST – Delete Example

Case 2: *n* has two children which are internal nodes

Find the first internal node *m* that follows *n* in an inorder traversal

Replace *n* with *m*

Delete 18

BST – Delete Example

Case 2: *n* has two children which are internal nodes

Find the first internal node *m* that follows *n* in an inorder traversal
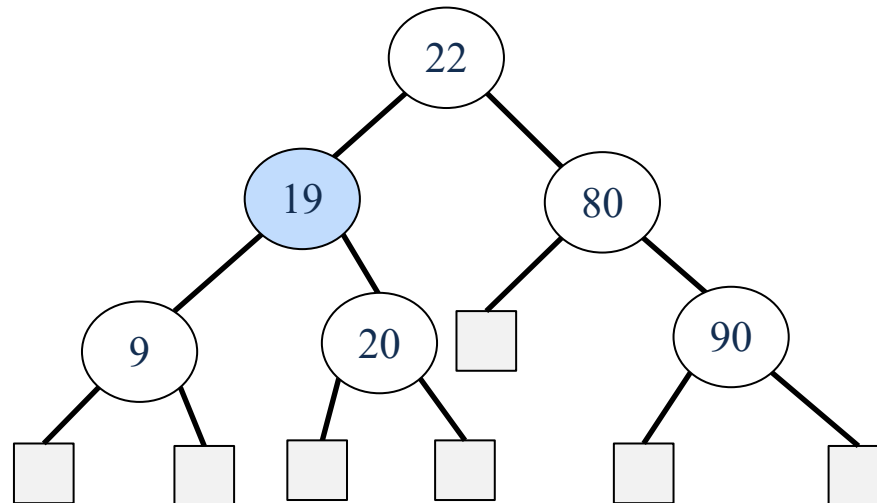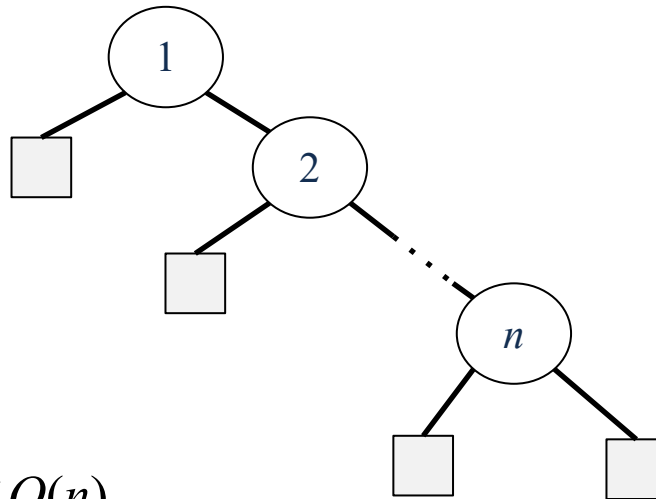
Replace *n* with *m*

Delete 18

BST Performance

Space used 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?

Dictionary: Worst-case Comparison

	Unordered		Ordered		
	Log file	**Hash table**	**Lookup table**	**Binary Search Tree**	**Balanced Trees**
size, isEmpty	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
keys, elements	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
findElement	$O(n)$	$O(n)$**	$O(\log n)$	$O(h)$	$O(\log n)$
insertItem	$O(1)$	$O(n)$**	$O(n)$	$O(h)$	$O(\log n)$
removeElement	$O(n)$	$O(n)$**	$O(n)$	$O(h)$	$O(\log n)$
closestKey closestElem	$O(n)$	$O(n)$	$O(\log n)$	$O(h)$	$O(\log n)$

** Expected running time is $O(1)$

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 $A[i] = i$.