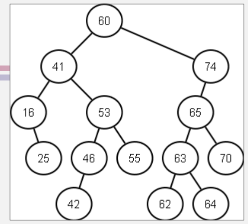
Binary Search Trees

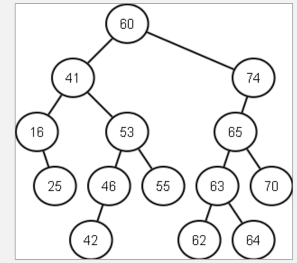


(dummy leaves not shown)

- lookup
  - moving down, 1-finger (only go to one child) pattern → loop
  - search ends when element is found or a leaf is reached (element not found)
- insert
  - can only insert at a leaf
  - the correct insertion point is the leaf where an unsuccessful search for the element ends up
- remove
  - can only remove above a leaf
  - if the element to remove does not have at least one leaf child, swap it with a safe element which does has at least one leaf child
    - i.e. the next element larger or smaller than the one to remove

Self-Test

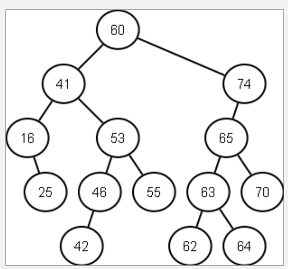
insert the elements 30, 61, 80, 50 into the binary search tree shown – draw the tree after each insertion



(dummy leaves not shown)

Self-Test

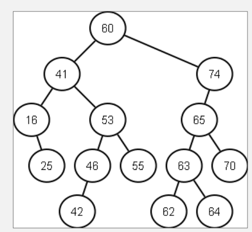
remove the elements 42, 16, 65, 60 from the binary search tree shown – draw the tree after each removal



(dummy leaves not shown)

Binary Search Trees

- visit all elements in order
  - moving down, both children pattern → recursion
  - need to visit smaller elements before the current node's element before the larger elements → inorder traversal



(dummy leaves not shown)

Implementing Map/Set

	unsorted array or linked list	sorted array	sorted linked list
map: insert(key,value) set: add(elt)			
map: remove(key) set: remove(elt)			
map: get(key) set: contains(elt)			

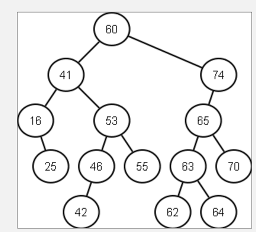
Implementing Map/Set

	unsorted array or linked list	sorted array	sorted linked list
	the element can be put anywhere so insert is fast, but find requires sequential search of the entire array/list	can utilize binary search to quick locate element or its insertion point, but must shift on both insert and remove	no need to shift, but requires sequential search to find element or its insertion point
map: insert(key,value) set: add(elt)	Θ(1) – put at head (linked list) or end (array)	O(n) – O(log n) to find correct insertion point but O(n) to shift	O(n) – to find correct insertion point, then Θ(1) to insert
map: remove(key) set: remove(elt)	Θ(n) – to find, then Θ(1) to remove	O(n) – O(log n) to find but O(n) to shift	O(n) – to find, then Θ(1) to remove
map: get(key) set: contains(elt)	Θ(n) – to find	O(log n) – binary search	O(n) – to find

Implementing Map/Set

- can store (key,value) pairs in a binary search tree ordered by key
  - let *h* be the height of the tree
  - lookup, insert, remove are all O(*h*) as it may be necessary to go from the root all the way down to a leaf
    - the loop may repeat up to *h* times

height h = 5



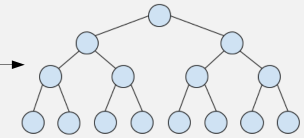
(only keys are shown; dummy leaves not shown)

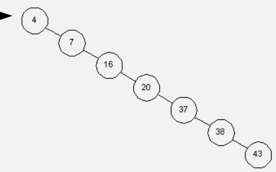
Implementing Map/Set

	unsorted array or linked list	sorted array	sorted linked list	binary search tree
	the element can be put anywhere so insert is fast, but find requires sequential search of the entire array/list	can utilize binary search to quick locate element or its insertion point, but must shift on both insert and remove	no need to shift, but requires sequential search to find element or its insertion point	loop from root to leaf
map: insert(key,value) set: add(elt)	Θ(1) – put at head (linked list) or end (array)	O(n) – O(log n) to find correct insertion point but O(n) to shift	O(n) – to find correct insertion point, then Θ(1) to insert	O(h) – to find correct insertion point, then Θ(1) to insert
map: remove(key) set: remove(elt)	Θ(n) – to find, then Θ(1) to remove	O(n) – O(log n) to find but O(n) to shift	O(n) – to find, then Θ(1) to remove	O(h) – to find and then find element to swap with, then then Θ(1) to swap and remove
map: get(key) set: contains(elt)	Θ(n) – to find	O(log n) – binary search	O(n) – to find	O(h) – to find

BST Height

- height of a binary search tree containing *n* elements
  - shortest possible tree has height O(log n)
    - this means that doubling the number of nodes only increases the height of the tree by 1
  - tallest possible tree has height O(n)
    - one element per level of the tree

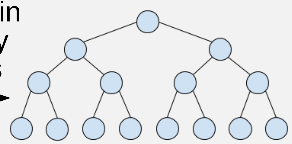


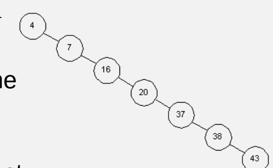


(dummy leaves not shown)

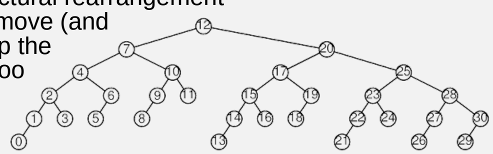
Balanced BSTs

- using a BST to store the elements in a Map or Set is faster than an array or linked list for most operations as long as the tree is more like this than this





- whether a BST with a given number of elements is shorter or taller depends on the order of insertions and removals, not the elements in the tree
- can do some extra structural rearrangement as part of insert and remove (and possibly lookup) to keep the height from becoming too large → *balanced* BST
  - O(log n) lookup, insert, remove



Implementing Map/Set

	unsorted array or linked list	sorted linked list	sorted array	balanced BST
	the element can be put anywhere so insert is fast, but find requires sequential search of the entire array/list	can utilize binary search to quick locate element or its insertion point, but must shift on both insert and remove	no need to shift, but requires sequential search to find element or its insertion point	loop from root to leaf + rebalancing as needed after operation
map: insert(key,value) set: add(elt)	Θ(1) – put at head (linked list) or end (array)	O(n) – O(log n) to find correct insertion point but O(n) to shift	O(n) – to find correct insertion point, then Θ(1) to insert	Θ(log n)
map: remove(key) set: remove(elt)	Θ(n) – to find, then Θ(1) to remove	O(n) – O(log n) to find but O(n) to shift	O(n) – to find, then Θ(1) to remove	Θ(log n)
map: get(key) set: contains(elt)	Θ(n) – to find	O(log n) – binary search	O(n) – to find	Θ(log n)

Implementing PriorityQueue

Consider using a binary search tree to store the elements in a priority queue –

- how would you carry out the insert, remove smallest, and retrieve smallest operations?
- in terms of efficiency, how does using a BST compare to using a sorted or unsorted array or linked list?

	unsorted array or linked list	sorted linked list	sorted array	balanced BST
insert				
remove smallest				
retrieve smallest				

Implementing PriorityQueue

Consider using a binary search tree to store the elements in a priority queue –

- how would you carry out the insert, remove smallest, and retrieve smallest operations?
- in terms of efficiency, how does using a BST compare to using a sorted or unsorted array or linked list?

	unsorted array or linked list	sorted linked list	sorted array	balanced BST
insert	Θ(1) – put at head (linked list) or end (array)	O(n) – to find correct insertion point, then Θ(1) to insert	O(n) – O(log n) to find correct insertion point but O(n) to shift	Θ(log n)
remove smallest	Θ(n) – to find, then Θ(1) to remove	Θ(1) – at head	Θ(1) – with a circular array so that shifting can be avoided	Θ(log n) – leftmost internal node
retrieve smallest	Θ(n) – to find	Θ(1)	Θ(1)	Θ(log n) – leftmost internal node

Implementing Lookup

- with an unsorted array, linked list, or tree, we might have to look at all of the elements – O(n)
  - we do have to look at all of them if our element isn't present
- with a sorted linked list, we might have to look at all of the elements – O(n)
  - but we can potentially stop early if the element isn't present
- with a sorted array or (balanced) binary search tree, we only have to look at O(log n) elements
- can we only have to look at O(1) elements…?