Basic Implementation of Map/Dictionary

	Unsorted	Sorted	Singly linked		Doubly linked	
Dictionary operation	array	array	unsorted	sorted	unsorted	sorted
Search(A, k)	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Insert(A, x)	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$
Delete(A, x) or Delete(A,k) (given location of *x*)	$O(1)^*$	$O(n)$	O(1) *	O(1) *	$O(1)$	$O(1)$
Remove(A,x) or Remove(A,k) (not given location of *x*)	O(n) requires search + delete	O(n)	O(n)	O(n)	O(n)	O(n)

A is the dictionary, *k* is a key, *x* is a key-value pair (*k*,*v*)

delete operation as defined in ADM assumes that the element is already found (known array index, pointer to the linked list node) – otherwise find operation is required first

* denotes cleverness or subtlety

Improving Map/Dictionary

- there is at least one O(n) operation for every implementation considered
  - unsorted leads to O(1) insert/delete but O(n) search for both arrays and linked lists
  - sorted leads to differences between arrays and linked lists
    - O(log n) search and O(n) delete for arrays
    - O(n) search and O(1) delete for linked lists

Can we do better?

- O(n) delete in arrays is due to shifting – can't do much about that
  - circular arrays worked for queues because insert/delete was only at the ends
- can we exploit the sorted order to improve searching in linked lists?
  - like binary search in arrays, perhaps...

Improving an Implementation – Map

Binary search exploits the sorted order – but it requires efficient random access.

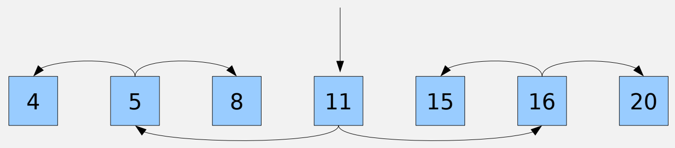
Or does it?

- the first iteration of binary search requires knowing the middle element
- successive iterations require knowing the middle element of one of the halves

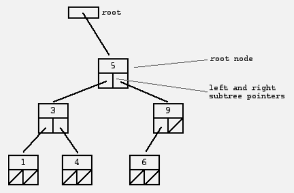
Finding the middle element is achieved in arrays by arithmetic involving array indexes, but what if we just stored the necessary info instead?

- store instead of computing…

Doing Better

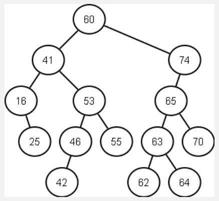


- each middle element only needs to store the location of two other middle elements → binary tree structure
- overall the elements are ordered, so the "other middle elements" are smaller and larger than the "middle element", respectively → binary search tree



Binary Search Trees

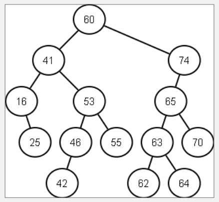
- a binary tree with an ordering property for the elements
  - for every node, all of the elements in the left subtree are less than or equal to the node's element and all of the elements in the right subtree are greater than the node's element
- operations
  - find
  - insert
  - remove
  - visit all elements (traverse) in order



(dummy leaves not shown)

Binary Search Trees

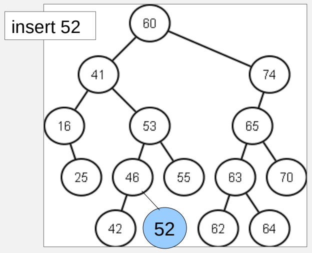
- find
  - start at root
  - go to left child or right child depending on the target value and the current node's value
  - moving down, one child pattern → loop
  - observation: if the element isn't there, search ends at a (dummy) leaf

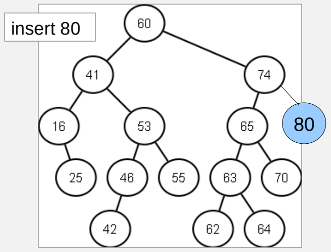


(dummy leaves not shown)

Binary Search Trees

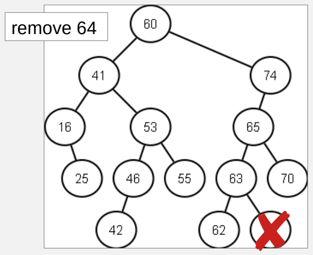
- insert
  - can only insert at a leaf
  - the correct insertion point is the leaf where an unsuccessful search for the element ends up

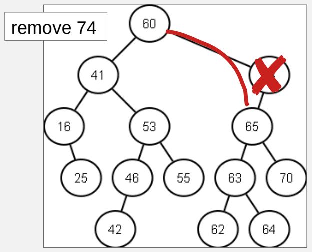




Binary Search Trees

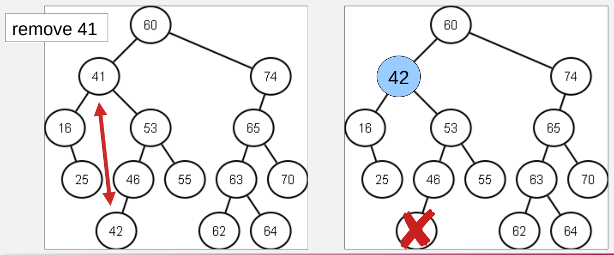
- remove
  - can only remove above a leaf





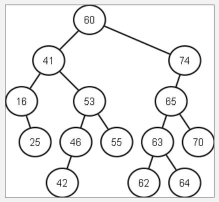
Binary Search Trees

- 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



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)

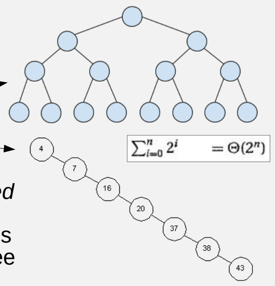
Implementing Map

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

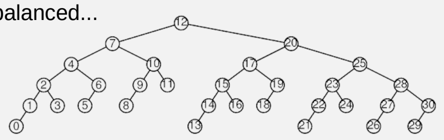
Dictionary operation	Unsorted array	Sorted array	Singly linked unsorted	Singly linked sorted	BST
$\mathrm{Search}(A, k)$	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	O(h) – root to leaf
$\mathrm{Insert}(A, x)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	O(h) – search + add node
$\mathrm{Delete}(A, x)$ or Delete(A,k) (given location of *x*)	$O(1)^*$	$O(n)$	O(1) *	O(1) *	O(h) – may need to find successor + swap, remove node
Remove(A,x) or Remove(A,k) (not given location of *x*)	O(n) requires search + delete	O(n)	O(n)	O(n)	O(h)

BST Height

- height of a binary search tree
  - best case is O(log n)
  - worst case is O(n)
- whether a BST of a given size is *balanced* (O(log n) height) or unbalanced (O(n) height) depends on the order of insertions and removals, not the elements in the tree
- can we do better?
  - try to keep the tree balanced...



$\sum_{i=0}^{n} 2^i \quad = \Theta(2^n)$


12
7
20
4
10
17
25
2
6
9
11
15
19
23
28
1
3
5
8
14
16
18
22
24
27
30
0
13
21
26
29