Symbol Tables & Searching

Section 2.1

Binary Search

Cutting the problem in half… many times

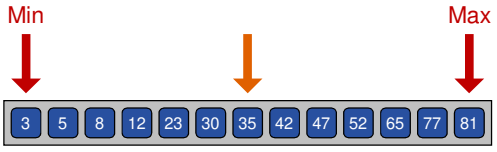
Binary Searching

- A *binary search* is an fast and efficient way to search an array
- Algorithm works like the classic "secret number game"
- Requires that the array is sorted before the search

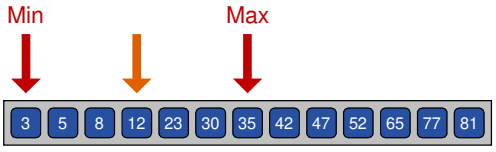
How it Works

- Starts knowing the max & min values
  - in the case of arrays, this is the min and max index
  - in the number game, it is the min and max value
- Algorithm continues
  - it looks at the midpoint between the first and last
  - if the value > target, the max is set to the midpoint
  - if the value < target, the min is set to the midpoint
  - *this eliminates half of the numbers each iteration*

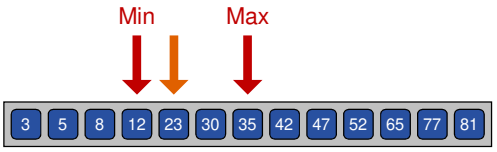
Binary Example: Find 30



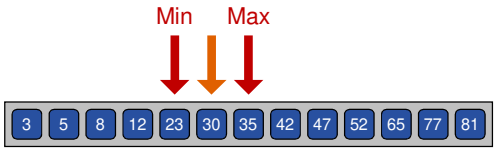
Binary Example: Find 30



Binary Example: Find 30



Binary Example: Find 30



Benefits

- The binary search is incredibly efficient and absolutely necessary for large arrays
- Any item can be found only $\log_2(n)$ searches! It is O(log n)
- However, since array must be sorted, sorting algorithms are equally vital

Maximum # of Searches

Array Size	Sequential	Binary
10	10	4
100	100	7
1,000	1,000	10
10,000	10,000	14
100,000	100,000	17
1,000,000	1,000,000	20
10,000,000	10,000,000	24
100,000,000	100,000,000	27
1,000,000,000	1,000,000,000	30

Dictionaries

the "ADT"… not in a human one!

Moving Past Arrays....

- A *collection* is general term for an group of data items
- So, this can include arrays, linked lists, stacks, queues, and much more
- So far, we have just used arrays – which are indexed by an integer

Moving Past Arrays....

- Are there are other ways to index data?
- Yes.
  - any object can be used as an index
  - e.g. strings, integers, pictures, etc...

Dictionaries

- Collections of objects indexed by other objects are called *dictionaries*
- They have a few alternative names...
  - *keyed tables*
  - *symbol tables*
  - *maps*

Dictionary Terminology

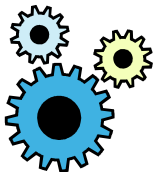
- The objects that are used for indices are called *keys*
- The objects that are accessed using the key are called *values*

Databases vs. Dictionaries

- Dictionaries...
  - have a single key
  - that key is the only way to access data
  - key returns a single value
- Databases...
  - may have multiple keys (e.g. SSN, name, age, etc…)
  - may return multiple objects – e.g. all the students taking CSc 130

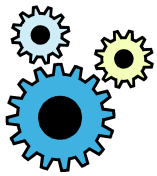
Implementing Dictionaries

- There are numerous approaches to implementing dictionaries
- Typically, it uses a *keyed-value* structure
  - a class stores a key object and data object
  - this can be stored in any data structure we have covered



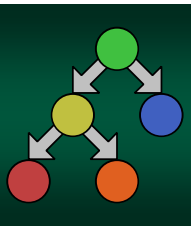
Implementing Dictionaries

- Using a linked list
  - adding takes O(1)
  - access is O(n)
- Unsorted array
  - add is O(n) – have to resize
  - access is O(n)
- Sorted array
  - add is O(n) – have to resize
  - access is O(log n)



This Ain't So Good

- So, adding in to an array is O(n)!
- Arrays seem like a poor approach
- Is there a better way to store dictionary data? Keeping adding close to O(1)?
- … and keep access at O(log n)
- Perhaps, we will learn that soon….

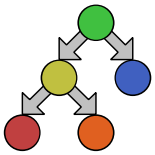


Binary Search Trees

Climbing a tree has never been so fun

Binary Search Trees

- *Binary Search Tree (BST)* is a special type of binary tree that sorts nodes by value
- Basically,
  - all the nodes on the left branch are less than the current node
  - all the nodes on the right branch are greater than the current node



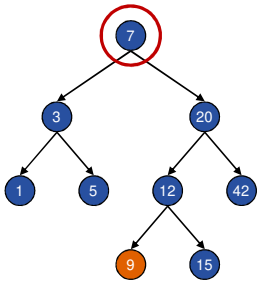
Searching the Tree

- Since the tree divides the problem progressively by two, the time complexity is only O(log n)
- Which gives that all the benefits of a sorted array
- Worst case is O(n) – if the tree is a list-like chain

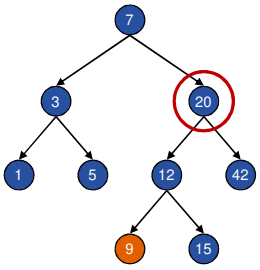
Search logic (looking for *S*)

- If *S* is equal to the current node, you found it
- If *S* is smaller than the current node, take the left branch
- If *S* is bigger than the current node, take the right branch
- If there are no branches, *S* was not found

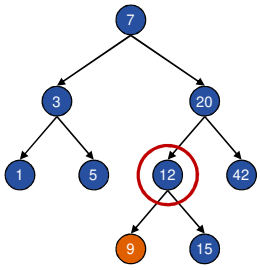
Searching for 9



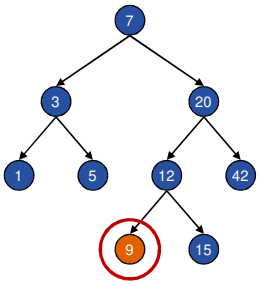
Searching for 9



Searching for 9



Searching for 9



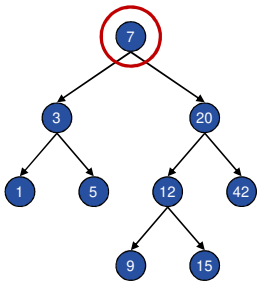
Binary Search Trees vs. Arrays

- When data is inserted in a Binary Search Tree
  - traverses down the tree until it finds the correct location
  - it then add itself there and restructures the tree
  - hence, the tree remains sorted as new data is added
  - it requires only O(log n)
- When data is inserted in an Array
  - it must be expanded when new elements are added
  - …and compacted when elements are removed
  - these requires O(n)

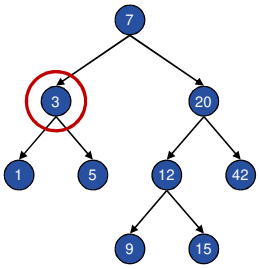
Inserting into the Tree

- Inserting is handled exactly like a search
- The only difference is that if the item is not found, the node is added
- If the item is not found,
  - we are already at the max-depth of the tree
  - we are at the node that needs to be changed
  - so, add a left or right node (based on value)
  - … wow, this is easy!

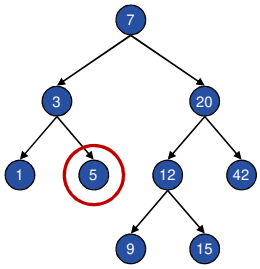
Inserting 4



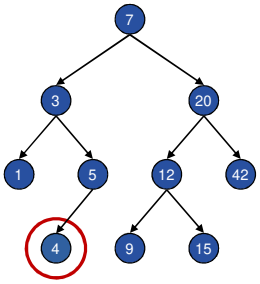
Inserting 4



Inserting 4



Inserting 4



Deleting From a BST

- Deleting from a BST is a tad more tricky
- When the node is deleted, the tree needs to be re-linked to *still preserve the ordering*

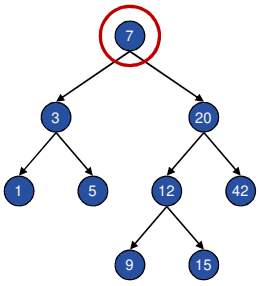
Deleting From a BST

- Fortunately, while the logic might seem a tad hard, it is fairly straight forward
- There are 3 different scenarios that must be taken into account based on the node being deleted

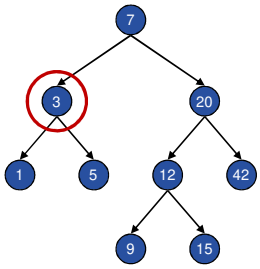
Case 1: The Node is a Leaf

- If the node (to be deleted) is a leaf, it can simply be removed from the parent
- This is due to the fact that nothing is linked from the node

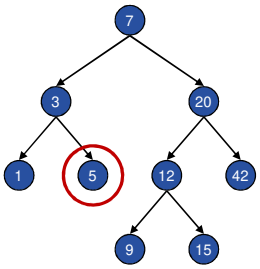
Case 1: Deleting 5



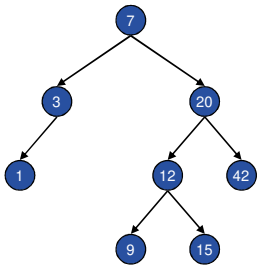
Case 1: Deleting 5



Case 1: Deleting 5



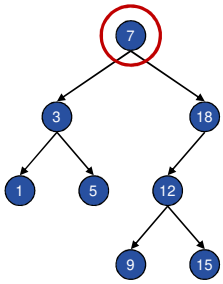
Case 1: Deleting 5



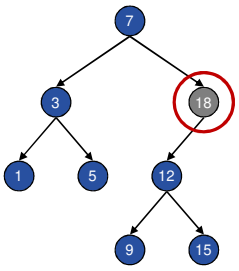
Case 2: One Children

- If the delete has a single child, then that child can be *"promoted"*
- The deleted node's child can be moved into its position

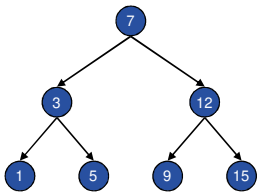
Case 2: Deleting 18



Case 2: Deleting 18



Case 2: Deleting 18



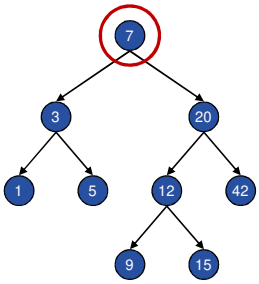
Case 3: Two Children

- However, if the deleted node has two children, then things get a tad more complex
- We need to find a node to *"promote"* to the deleted node's position

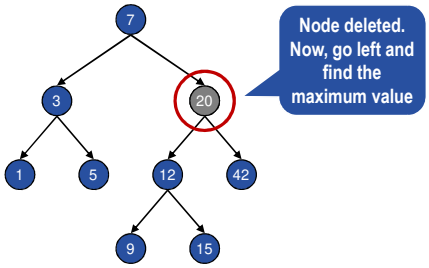
Case 3: Two Children

- We need to find a value that is mathematically next to the deleted value
- There are two equally valid options
- Choose one:
  - go down the left side and find the maximum node
  - … or go down the right side and find the minimum node

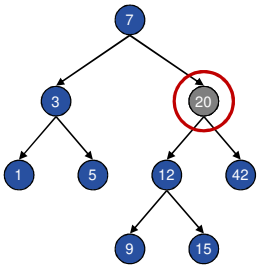
Case 3: Deleting 20 (Going Left)



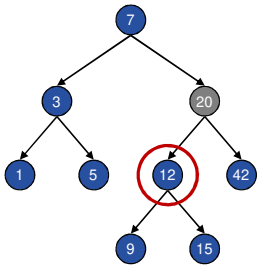
Case 3: Deleting 20 (Going Left)



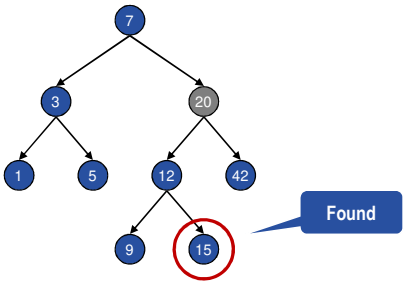
Case 3: Deleting 20 (Going Left)



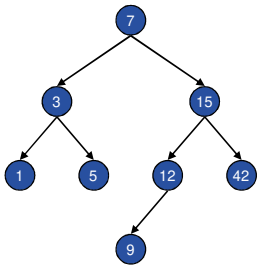
Case 3: Deleting 20 (Going Left)



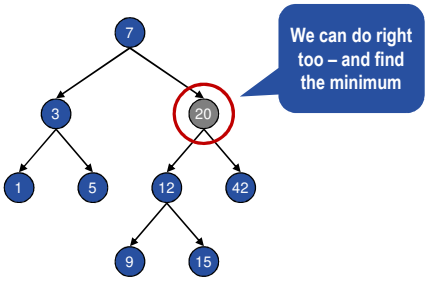
Case 3: Deleting 20 (Going Left)



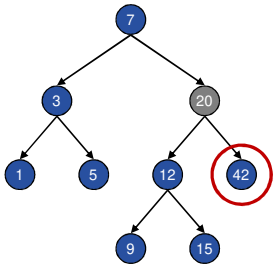
Case 3: Deleting 20 (Going Left)



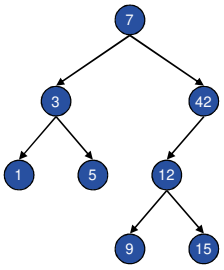
Deleting 20 : Going Right

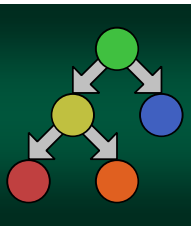


Case 3: Deleting 20 (Going Right)



Case 3: Deleting 20 (Going Right)



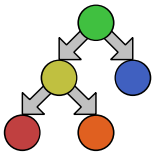


The Problem with BSTs

Climbing a tree has never been so fun

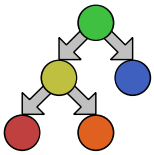
The Problem with BSTs

- In a BST, internal nodes never change – and have a profound affect on the rest of the tree
- There are cases where the tree is unbalanced – one particular path contains most of the data

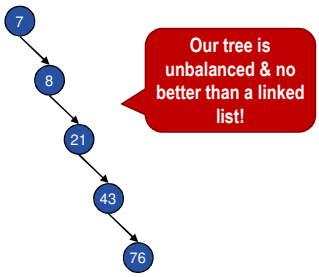


The Problem with BSTs

- This can easily occur of the data is not truly random (which is generally the case)
- When this happens, **the time complexity slowly deteriorates to O(n)**



This Might Get Crazy



Sort in a Binary Search Tree?

- Unfortunately, while this might seem like a good idea, this is not a great solution
- Binary Search Trees can deteriorate into linked lists
- So…
  - O(log n) search can quickly deteriorate to O(n)
  - …and O(n log n) sort can deteriorate to $O(n^2)$