CSC 212: Data Structures and Abstractions

15: Binary Search Trees I

Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Fall 2017

k-ary Trees

- In a **k-ary tree**, every node has between 0 and k children

k-ary Trees

- In a **k-ary tree**, every node has between 0 and k children
- In a **full (proper)** k-ary tree, every node has exactly 0 or k children

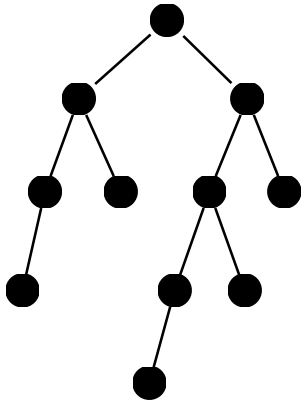
k-ary Trees

- In a **k-ary tree**, every node has between 0 and k children
- In a **full (proper)** k-ary tree, every node has exactly 0 or k children
- In a **complete** k-ary tree, every level is entirely filled, except possibly the deepest, where all nodes are as far left as possible

k-ary Trees

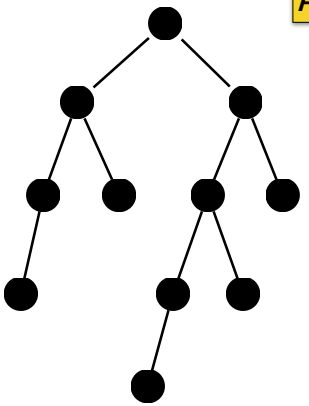
- In a **k-ary tree**, every node has between 0 and k children
- In a **full (proper)** k-ary tree, every node has exactly 0 or k children
- In a **complete** k-ary tree, every level is entirely filled, except possibly the deepest, where all nodes are as far left as possible
- In a **perfect** k-ary tree, every leaf has the same depth and the tree is full

Binary Tree



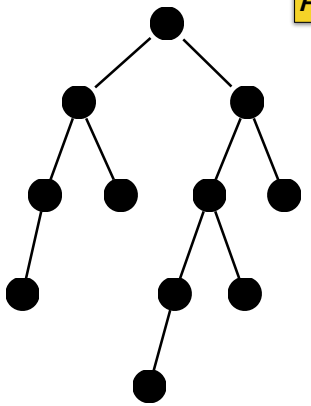
Binary Tree

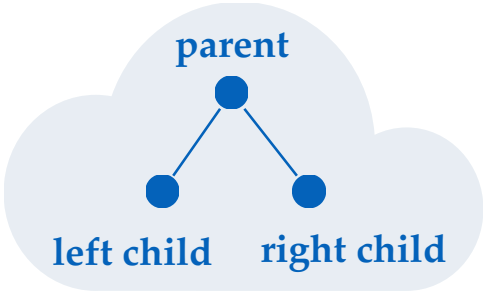
A k-ary tree where k = 2



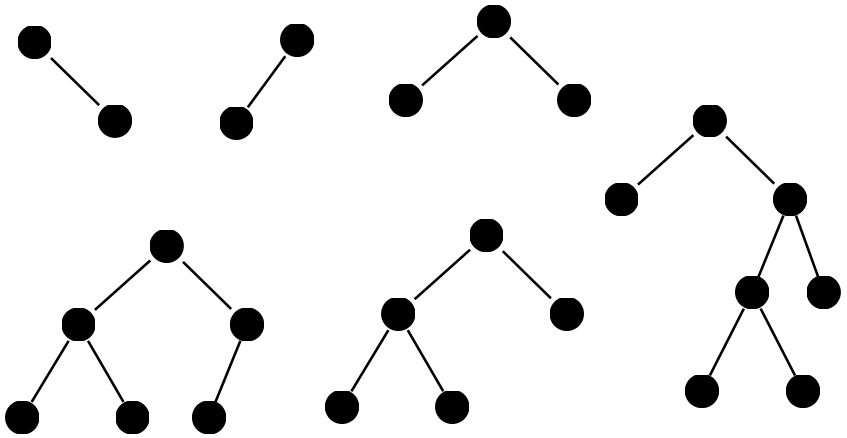
Binary Tree

A k-ary tree where k = 2





Quiz (binary trees)



Full? Complete? Perfect?

How to implement binary trees?

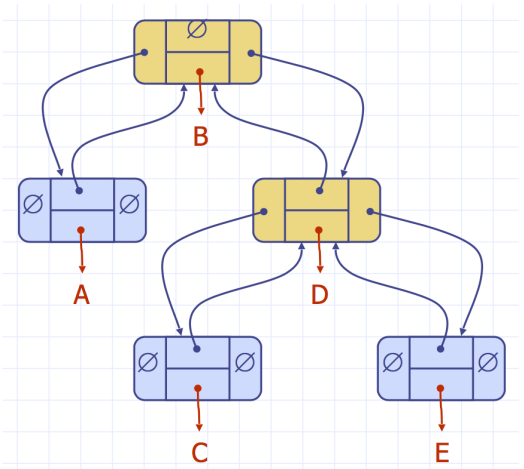
Node:

data
parent
left child
right child

How to implement binary trees?

Node:

data
parent
left child
right child



From Algorithm Design and Applications, Goodrich & Tamassia

Collections/Dictionaries as arrays

	What?		
`search`	search for a key		
`insert`	insert a key		
`delete`	delete a key		
`min/max`	smallest/largest key		
`floor/ ceiling`	predecessor/successor		
`rank`	number of keys less than key		

Collections/Dictionaries as arrays

	What?	Sequential Search (unordered sequence)	
`search`	search for a key	`O(n)`	
`insert`	insert a key	`O(n)`	
`delete`	delete a key	`O(n)`	
`min/max`	smallest/largest key	`O(n)`	
`floor/ ceiling`	predecessor/successor	`O(n)`	
`rank`	number of keys less than key	`O(n)`	

Collections/Dictionaries as arrays

	What?	Sequential Search (unordered sequence)	Binary Search (ordered sequence)
`search`	search for a key	`O(n)`	`O(log n)`
`insert`	insert a key	`O(n)`	`O(n)`
`delete`	delete a key	`O(n)`	`O(n)`
`min/max`	smallest/largest key	`O(n)`	`O(1)`
`floor/ ceiling`	predecessor/successor	`O(n)`	`O(log n)`
`rank`	number of keys less than key	`O(n)`	`O(log n)`

Binary Search Trees

Binary Search Tree

- A BST is a binary tree

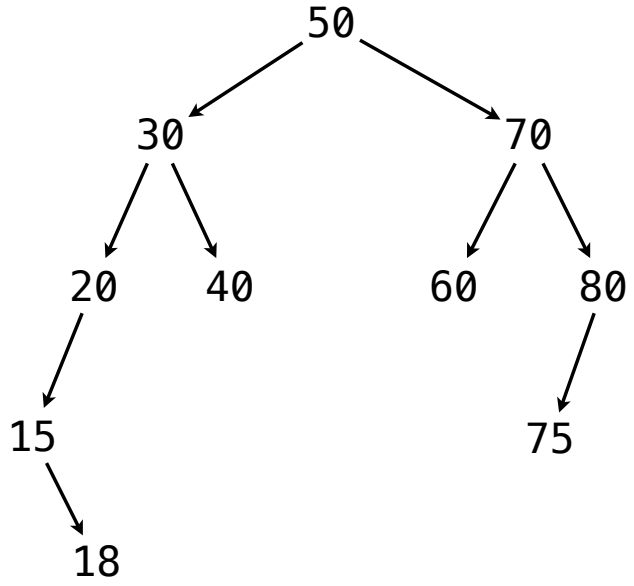
Binary Search Tree

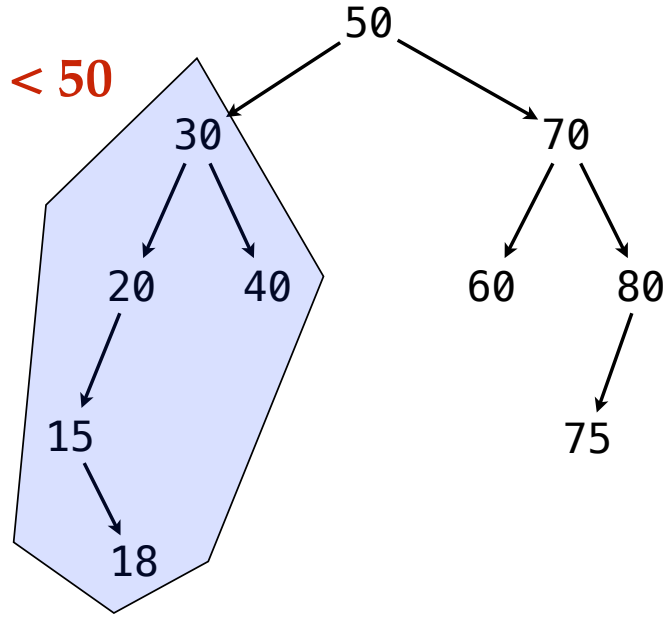
- A BST is a binary tree
- A BST has symmetric order
  - each node **x** in a BST has a key **key(x)**
  - for all nodes **y** in the left subtree of **x**, **key(y) < key(x)** **
  - for all nodes **y** in the right subtree of **x**, **key(y) > key(x)** **

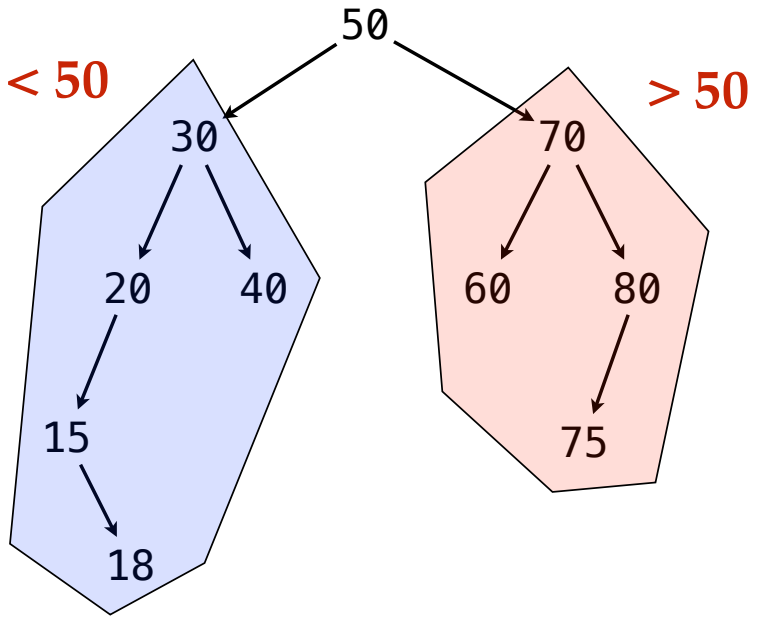
Binary Search Tree

- A BST is a binary tree
- A BST has symmetric order
  - each node **x** in a BST has a key **key(x)**
  - for all nodes **y** in the left subtree of **x**, **key(y) < key(x)** **
  - for all nodes **y** in the right subtree of **x**, **key(y) > key(x)** **

(**) assume that the keys of a BST are pairwise distinct








```
 1 class BSTNode {
 2
 3     private:
 4         int data;
 5         BSTNode *left;
 6         BSTNode *right;
 7
 8     public:
 9         BSTNode(int d);
10         ~BSTNode();
11
12     friend class BSTree;
13
14 };
```



```
 1 class BSTree {
 2
 3     private:
 4         unsigned int size;
 5         BSTNode *root;
 6         void destroy(BSTNode *p);
 7
 8     public:
 9         BSTree();
10         ~BSTree();
11         void insert(int d);
12         void remove(int d);
13         BSTNode *search(int d);
14
15 };
```


Search

- Start at root node

Search

- Start at root node
- If the search key matches the current node's key then **found**

Search

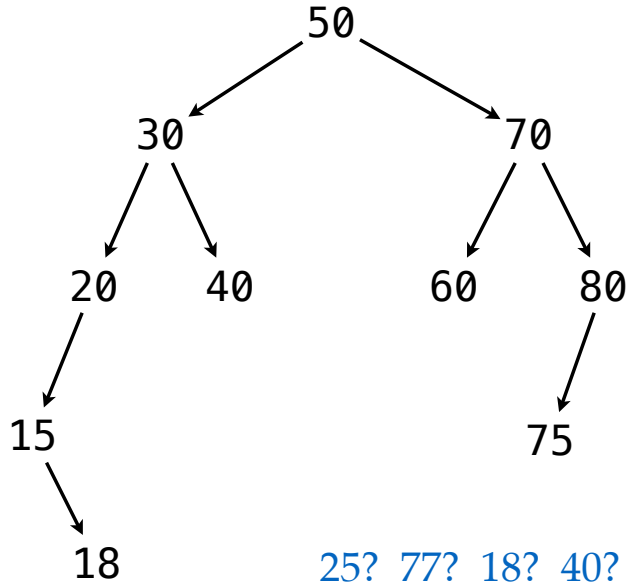
- Start at root node
- If the search key matches the current node's key then **found**
- If search key is greater than current node's key
  - search **recursively** on right child

Search

- Start at root node
- If the search key matches the current node's key then **found**
- If search key is greater than current node's key
  - search **recursively** on right child
- If search key is less than current node's
  - key search **recursively** on left child

Search

- Start at root node
- If the search key matches the current node's key then **found**
- If search key is greater than current node's key
  - search **recursively** on right child
- If search key is less than current node's
  - key search **recursively** on left child
- Stop recursion when current node is NULL (**not found**)



25? 77? 18? 40? 91?

Search: Iterative Algorithm

Search: Recursive Algorithm

Insert

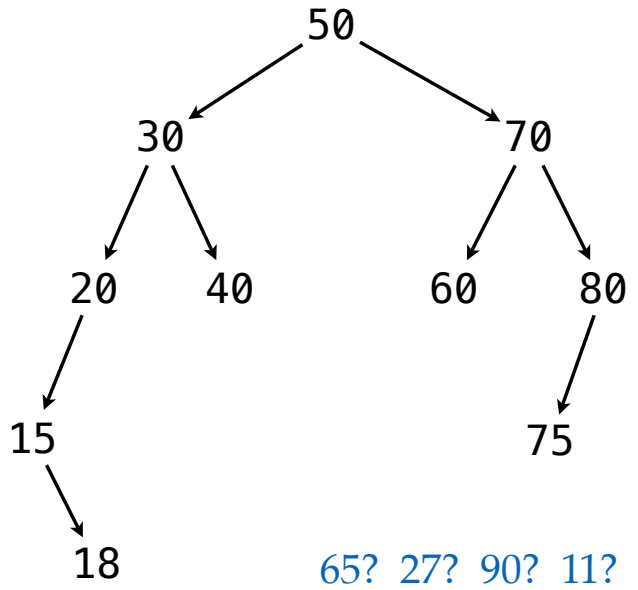
- Perform a Search operation

Insert

- Perform a Search operation
- If **found**, no need to insert (may increase counter)

Insert

- Perform a Search operation
- If **found**, no need to insert (may increase counter)
- If **not found**, insert node where Search stopped



65? 27? 90? 11? 51?

Insert: Iterative Algorithm

Insert: Recursive Algorithm