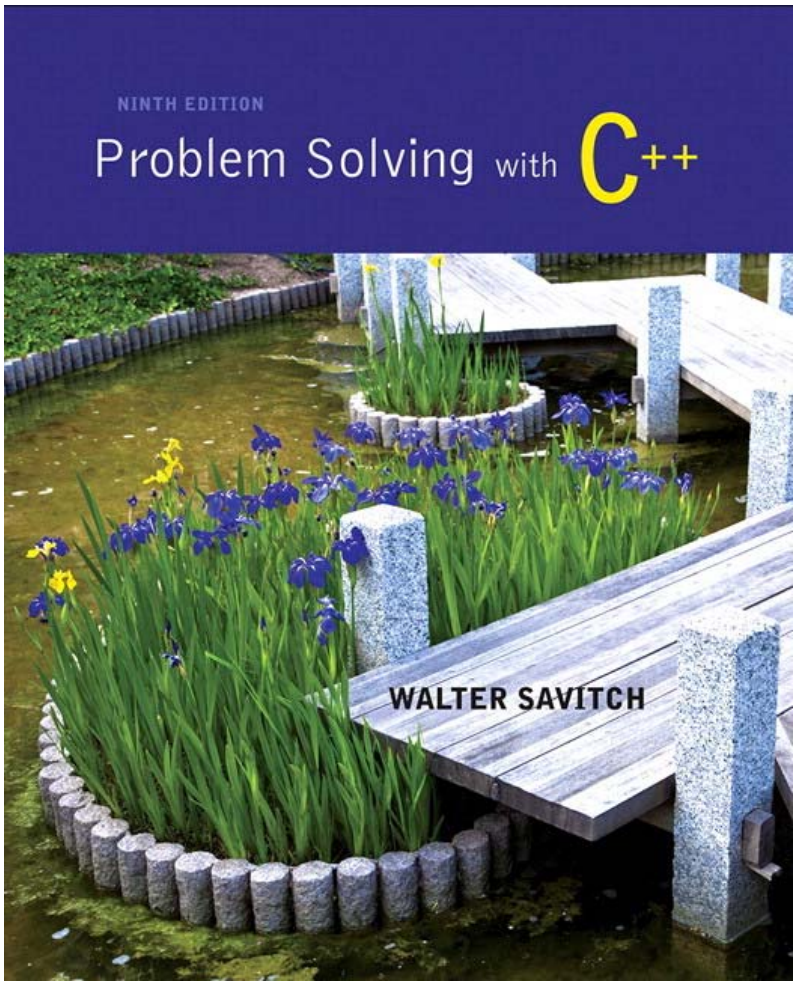
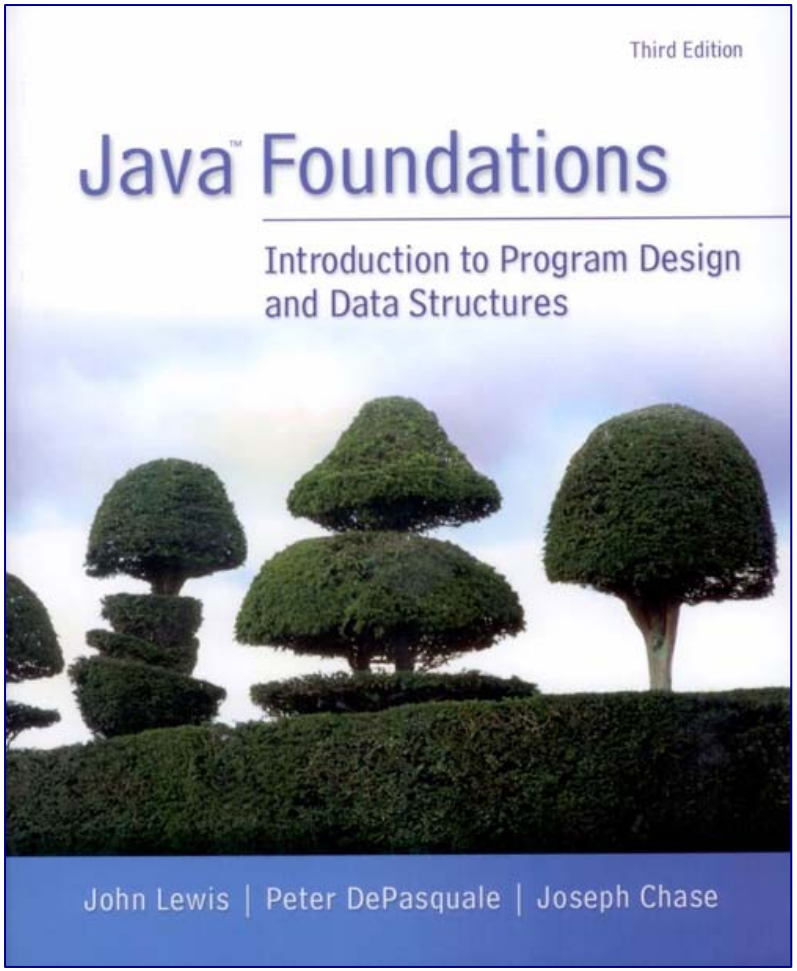
Chapter 13.1 – Binary Trees
Chapter 19,20 – Trees and Binary Search Trees (Java Foundations)

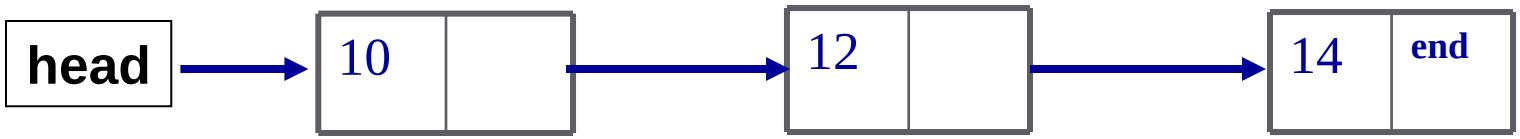




Recall: Nodes and Linked Lists

A Linked List:

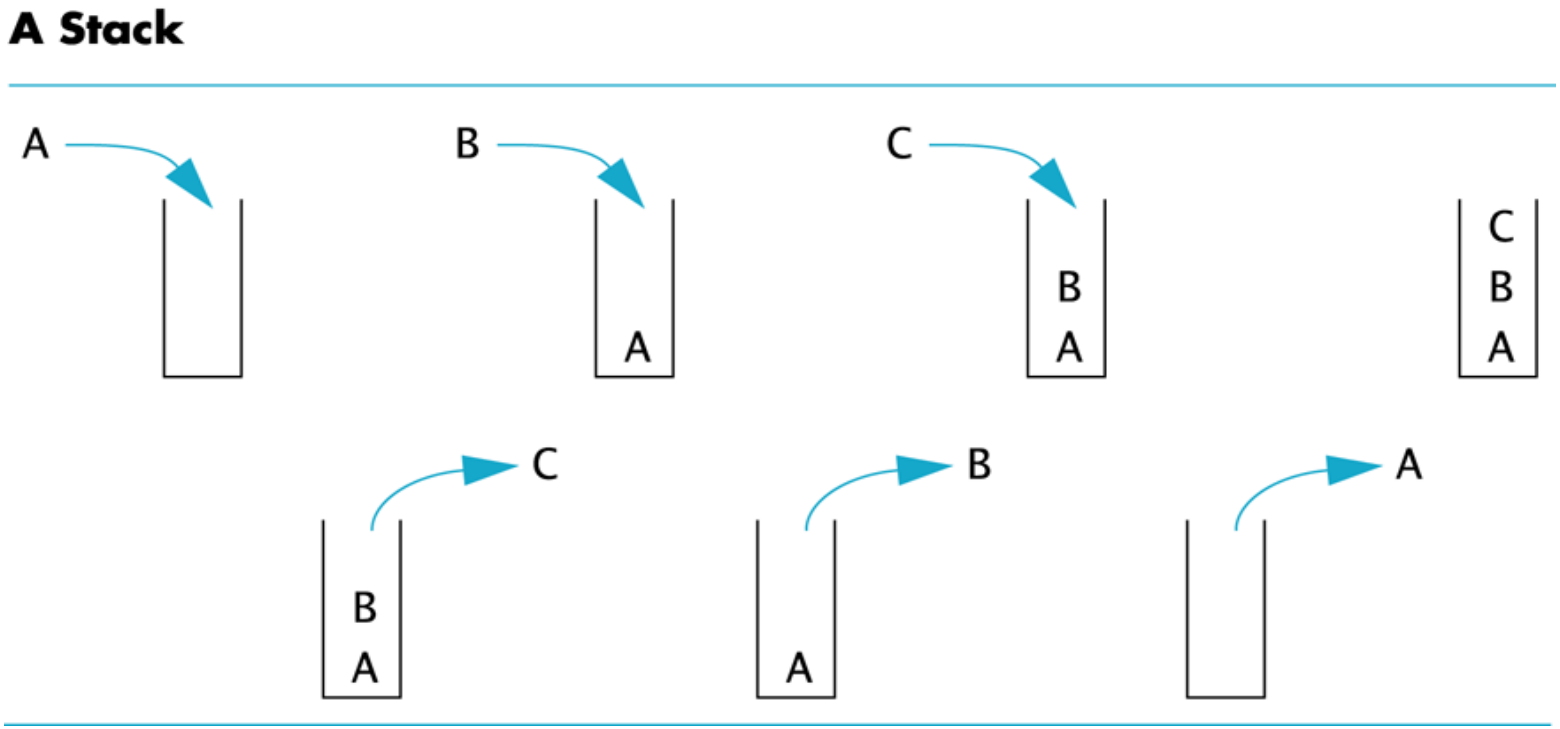
- can grow and shrink while the program is running
- is constructed using pointers
- often consists of structs or classes that contain pointers connecting each other
- Advantages over Arrays and Vectors
  - Easy to add and remove elements, slow random access
  - Search: O(n)
  - Insert at Head O(1), Insert in order or at End: O(n)



13.2 Stacks

A stack is a last-in/first-out data structure like the stack of plates in a cafeteria; adding a plate pushes down the stack and the top plate is the first one removed

- Used in our PostFix and PostFixRational Expression Evaluators



A Stack Class

Create a stack class to store characters

- Adding an item to a stack is pushing onto the stack
- Member function push will perform this task
- Removing an item is popping th item off the stack
- Member function pop will perfor this task

Interface File for a Stack Class

```
//This is the header file stack.h. This is the interface for the class Stack,
//which is a class for a stack of symbols.
#ifndef STACK_H
#define STACK_H
namespace stacksavitch
{
    struct StackFrame
    {
        char data;
        StackFrame *link;
    };

    typedef StackFrame* StackFramePtr;

    class Stack
    {
    public:
        Stack( );
        //Initializes the object to an empty stack.

        Stack(const Stack& a_stack);
        //Copy constructor.

        ~Stack( );
        //Destroys the stack and returns all the memory to the freestore.

        void push(char the_symbol);
        //Postcondition: the_symbol has been added to the stack.

        char pop( );
        //Precondition: The stack is not empty.
        //Returns the top symbol on the stack and removes that
        //top symbol from the stack.

        bool empty( ) const;
        //Returns true if the stack is empty. Returns false otherwise.
    private:
        StackFramePtr top;
    };
}//stacksavitch

#endif //STACK_H
```

Function push

The push function adds an item to the stack

- It uses a parameter of the type stored in the stack

```
void push(char   the_symbol);
```

- The same head_insert of the linked list
- For a stack, a pointer named top is used instead of a pointer named head

Function pop

The pop function returns the item that was at the top of the stack

char pop( );

- Before popping an item from a stack, pop checks that the stack is not empty
- pop stores the top item in a local variable `result`, and the item is "popped" by: top = top->link;
- A temporary pointer must point to the old top item so it can be "deleted" to prevent a memory leak
- pop then returns variable `result`

Empty Stack

An empty stack is identified by setting the top pointer to NULL or nullptr

```
top = nullptr;
```

What about memory leaks?
What about the nodes that top pointed to?

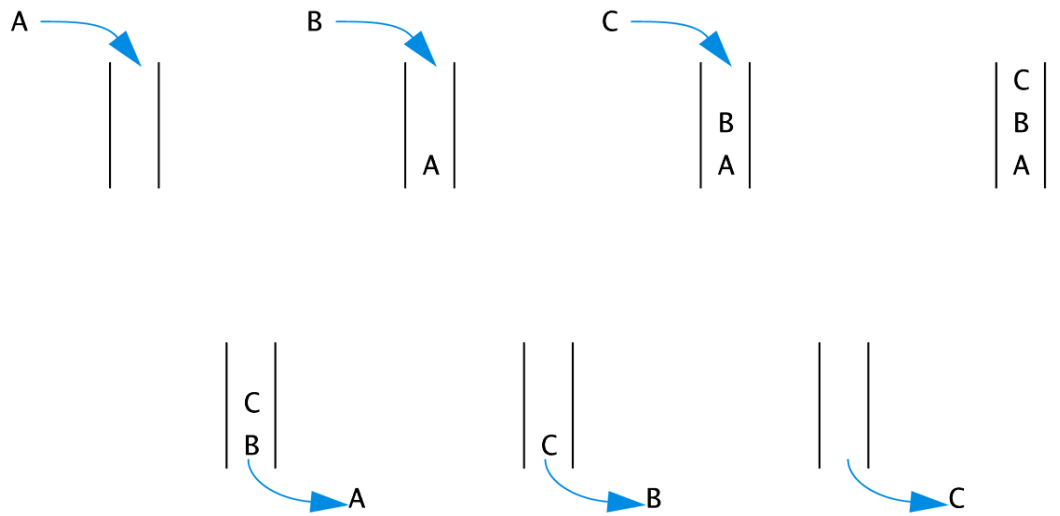
Chapter 13.2 Queues

A queue is a data structure that retrieves data in the same order the data was stored

- If 'A', 'B', and then 'C' are placed in a queue, they will be removed in the order ‘A', 'B', and then ‘C'

A queue is a first-in/first-out data structure like the checkout line in a supermarket

DISPLAY 13.20 A Queue



Section 13.2 Conclusion

Can you?

- Give the definition of stack member function push()?
- Know how to tell if a stack is empty?
- Know when to use a queue vs a stack?

Scope

Trees:

- How Google finds websites
- Trees as data structures
- Tree terminology
- Tree implementations
- Analyzing tree efficiency
- Tree traversals
- Expression trees

Binary Search Trees

A *search tree* is a tree whose elements are organized to facilitate finding a particular element when needed

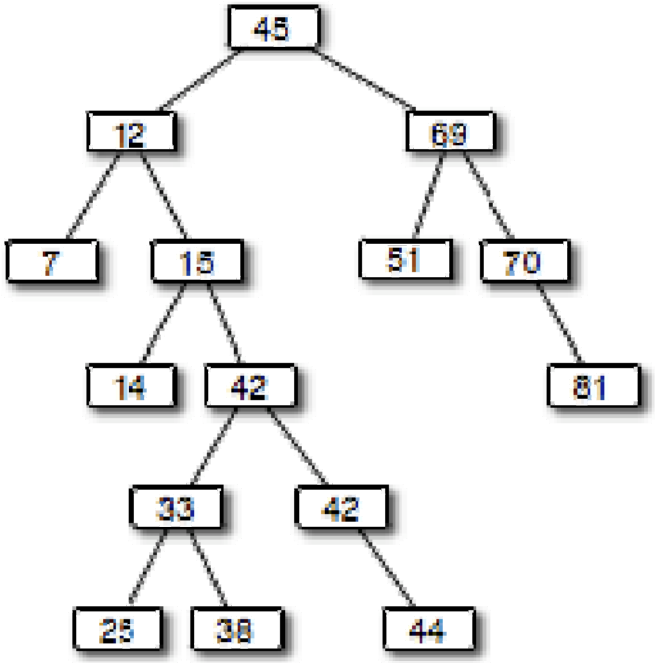
A *binary search tree* is a binary tree that, for each node *n*

- the left subtree of *n* contains elements less than the element stored in *n*
- the right subtree of *n* contains elements greater than or equal to the element stored in *n*

This is *how*

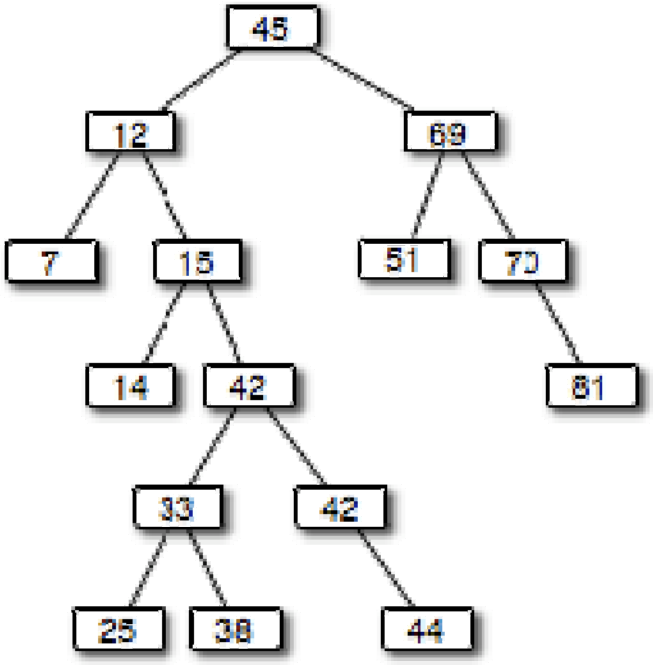
Google

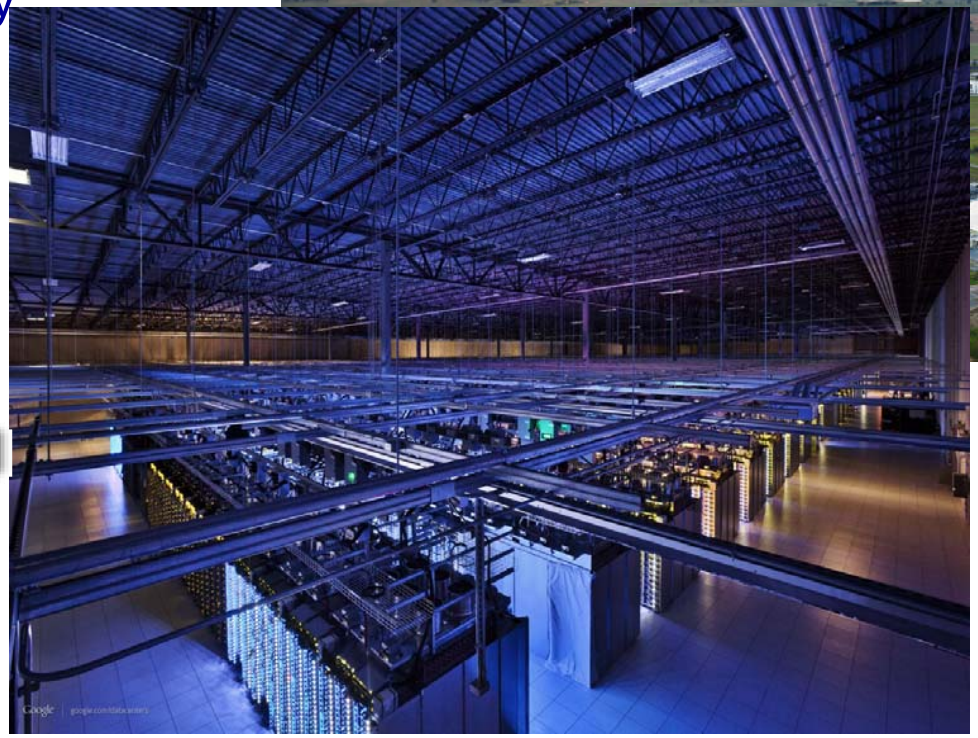
finds your websites!



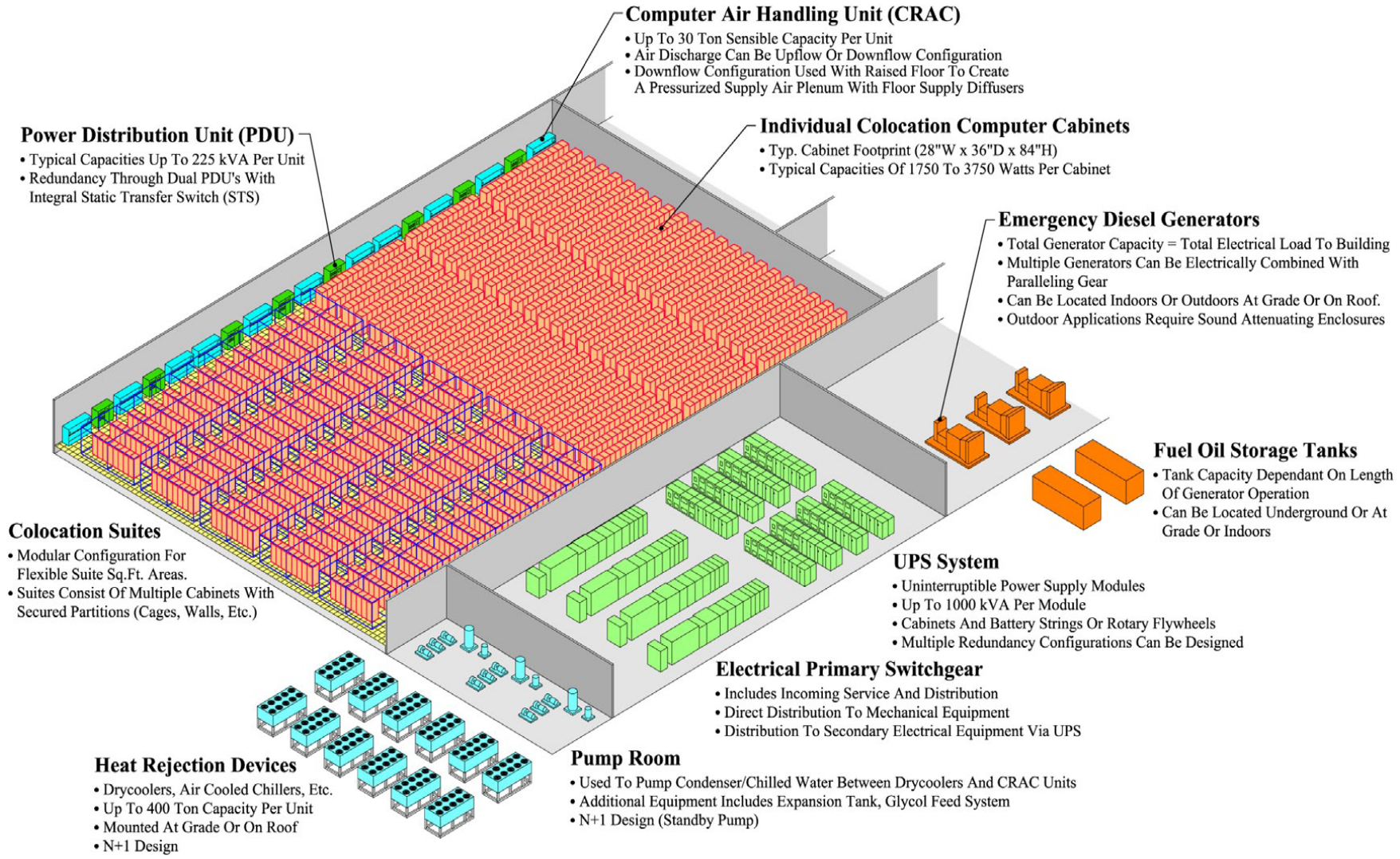
How Google finds Websites

Web Crawlers search the Internet for new websites
Read every webpage and every word
HUGE files of data – Petabytes!
Data Centers process this data
And Update Google Search Trees
Every webpage, every day

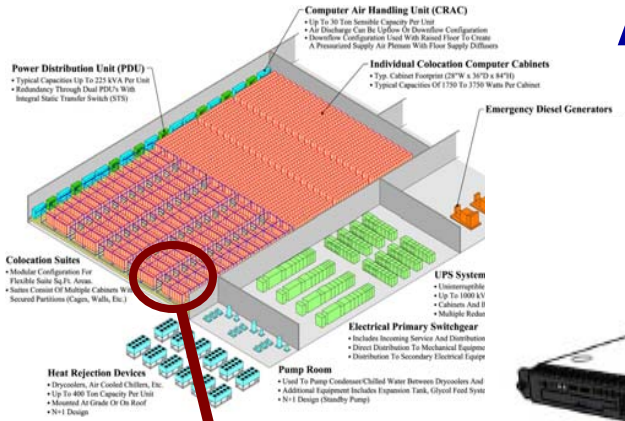




Google and Amazon Data Centers

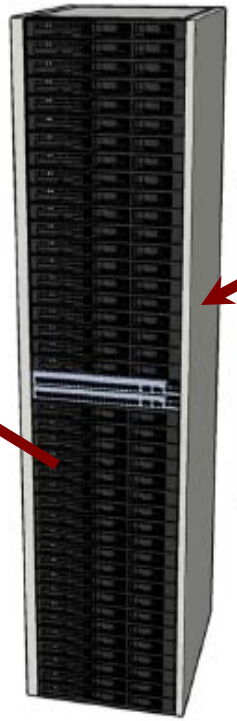


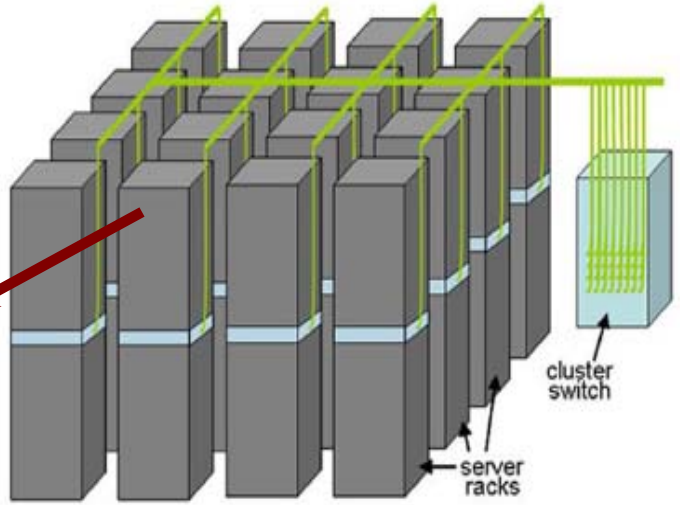
Data Centers – A Closer Look

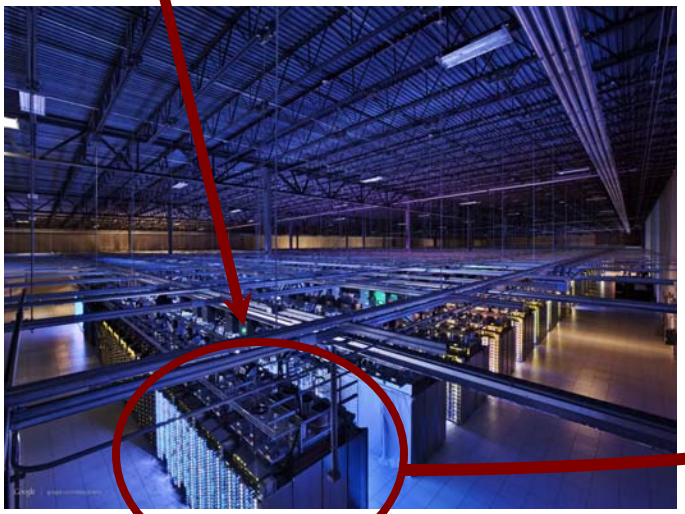


All this just to update some search trees

Some VERY BIG search trees!







For more info:
Dr. Mohamed Hefeeda
Big-Data and Multimedia

Binary Search Trees

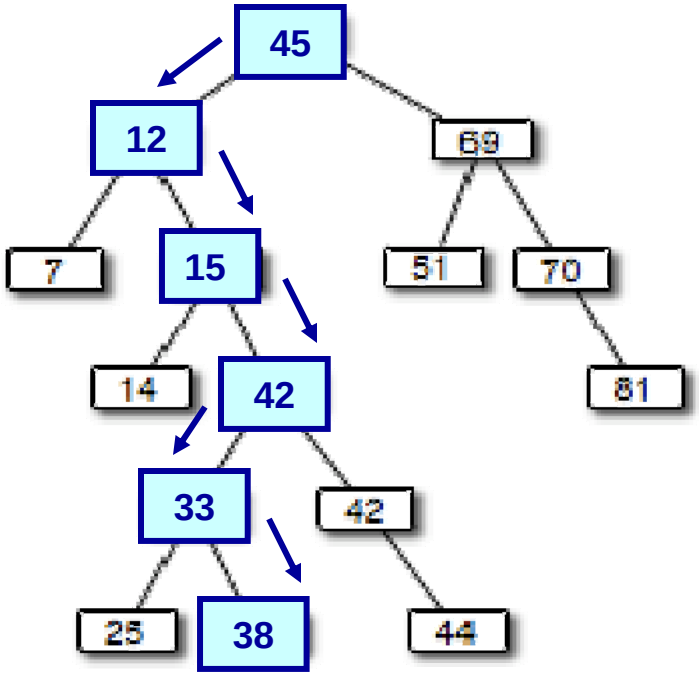
To determine if a particular value exists in a tree

- start at the root
- compare target to element at current node
- move left from current node if target is less than element in the current node
- move right from current node if target is greater than element in the current node

We eventually find the target or hit the end of a path (target is not found)

How to find node with *key* value 38?

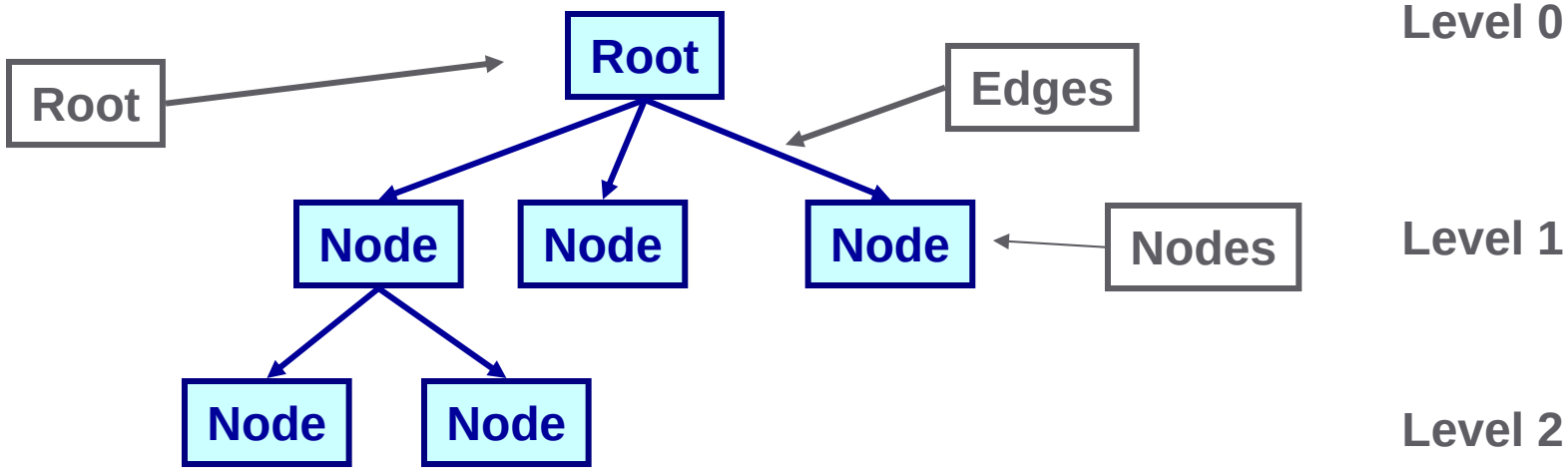
- **Start at Root and compare 38 to 45**
- **38 < 45 so go to left subtree**
- **38 > 12 so go to the right**
- **38 > 15 so go to the right again**
- **38 < 42 so go left**
- **38 > 33 so go right**
- **38 found!**
- **Return Object stored at this node**



Trees

A *Tree* looks like an upside-down tree with the root at the top

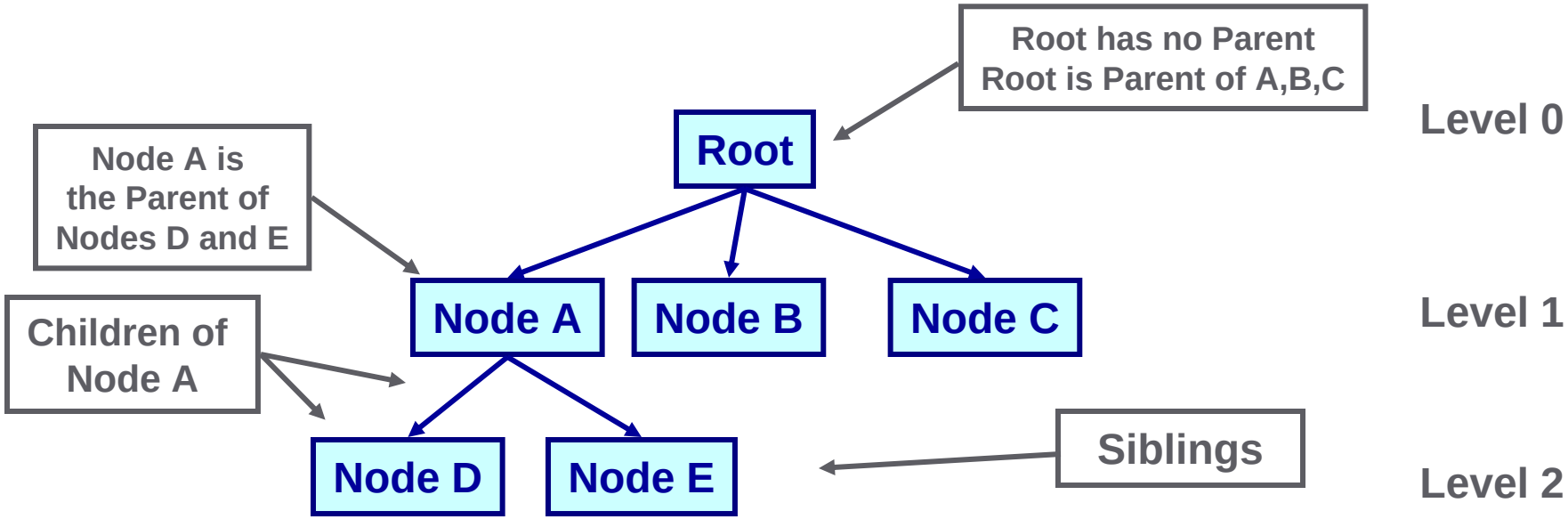
- elements organized into a hierarchy
- a set of *Nodes* and *Edges* connecting those nodes
- Data elements are stored within the nodes
- Each node is located on a particular *level*
- There is only one *root* node in the tree
- No Cycles permitted



Trees

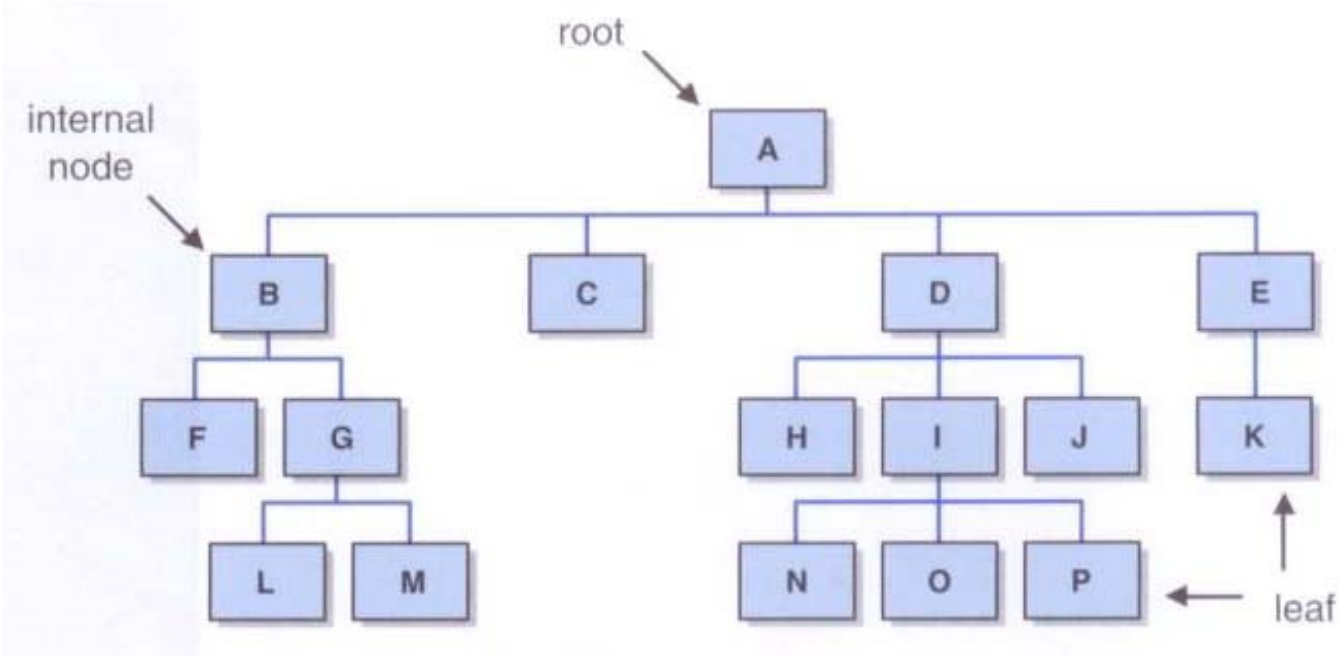
Nodes at the lower level of a tree are the *children* of nodes at the previous level

- Nodes can have only one *parent*, but multiple children
- Nodes that have the same parent are *siblings*
- The root is the only node which has no parent



Tree Terminology

A node that has no children is a *leaf* node
A node that is not the root and has at least one child is an *internal node*
A *subtree* is a tree structure that makes up part of another tree
We can follow a *path* through a tree from parent to child, starting at the root
A node is an *ancestor* of a node if it is above it on the path from the root.



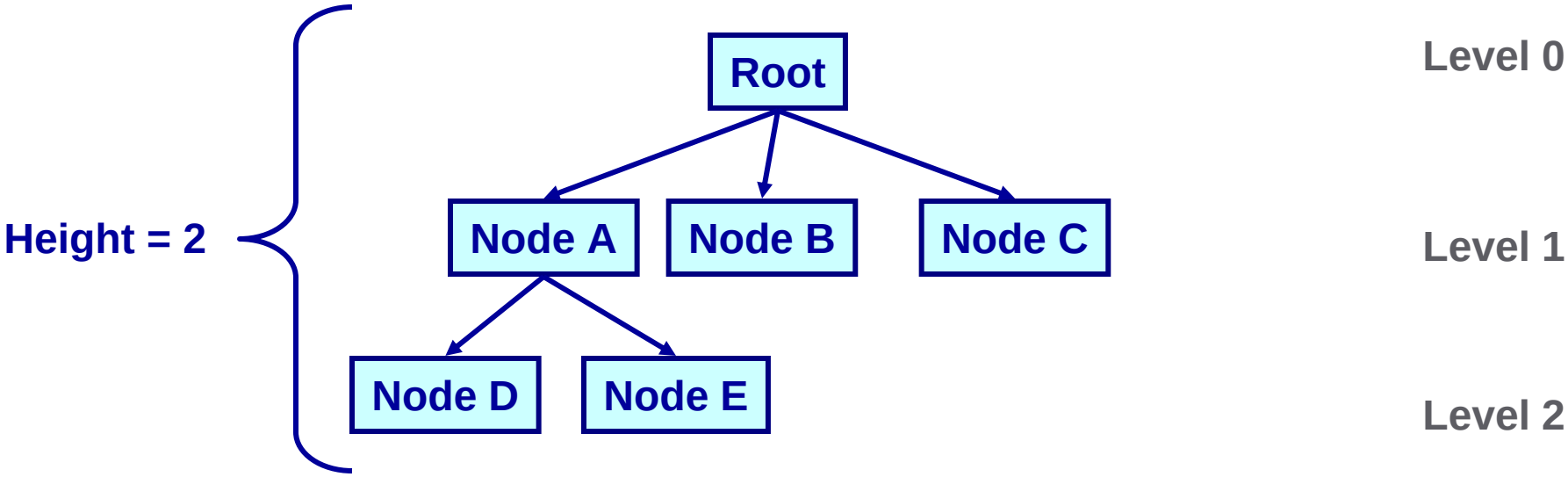
Trees Terminology

Nodes that can be reached by following a path from a particular node are the *descendants* of that node

The *level* of a node is the length of the path from the root to the node

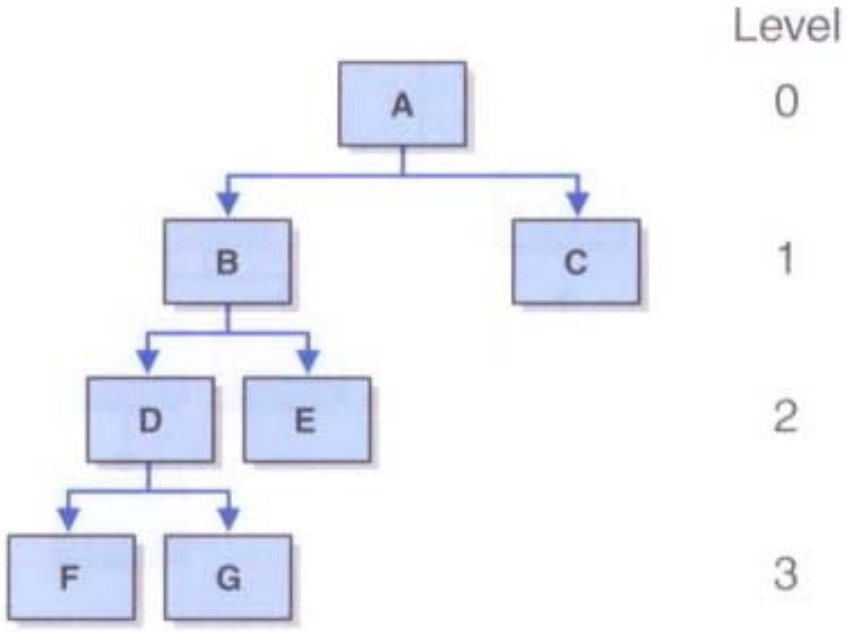
The *path length* is the number of edges to get from the root to the node

The *height* of a tree is the length of the longest path from the root to a leaf



Trees – Quiz

What are the *descendents* of node B?
What is the *level* of node E?
What is the *path length* to get from the root to node G?
What is the *height* of this tree?



Classifying Trees

Trees can be classified in many ways

One important criterion is the maximum number of children any node in the tree may have

This may be referred to as the ***order of the tree***

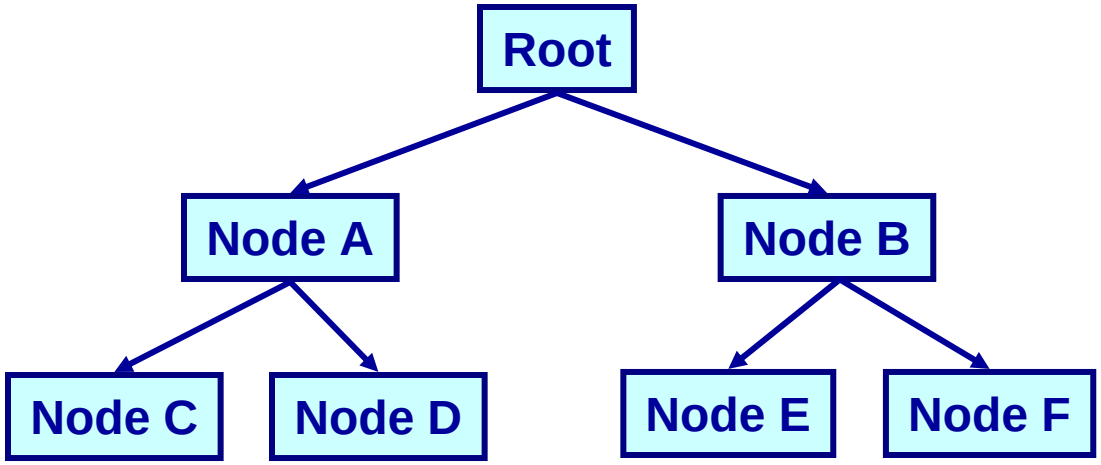
General trees have no limit to the number of children a node may have

A tree that limits each node to no more than *n* children is referred to as an *n-ary* tree

The tree for a TicTacToe game is an 9-ary tree

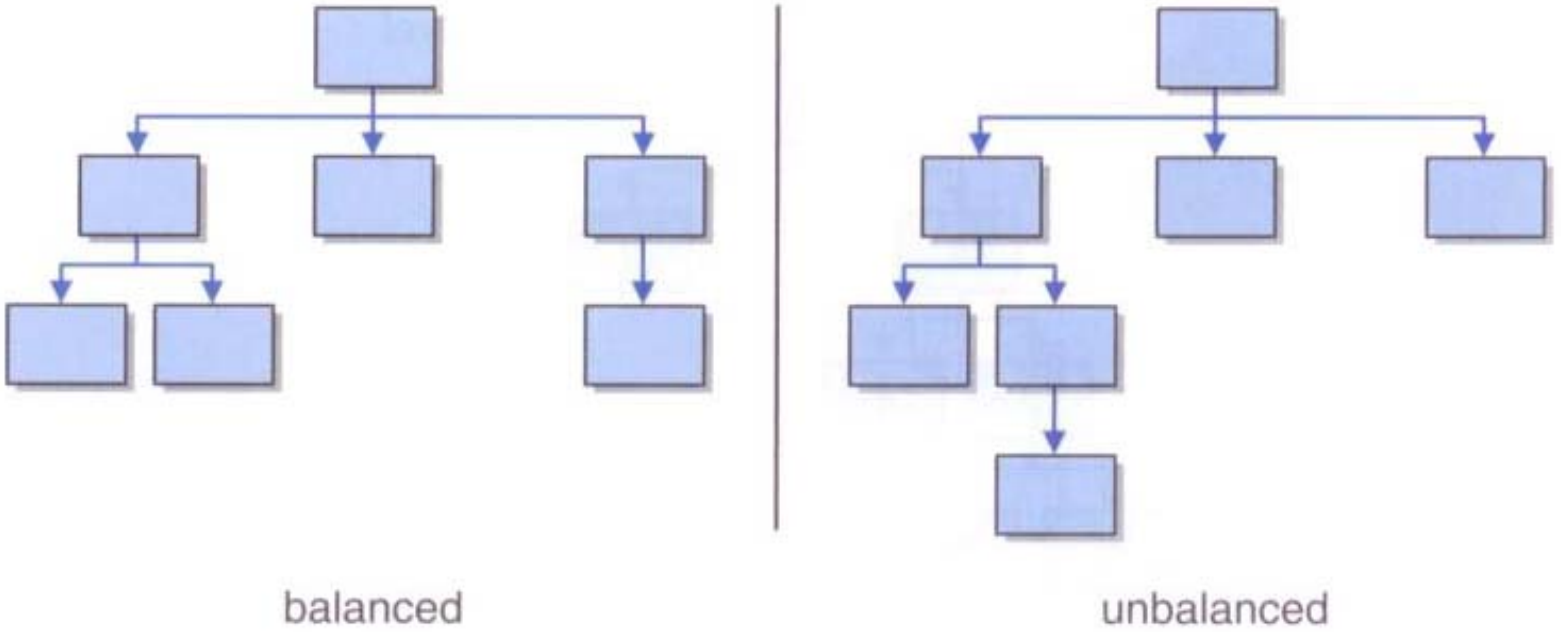
Binary Trees

Trees in which nodes may have at most two children are called *binary trees*



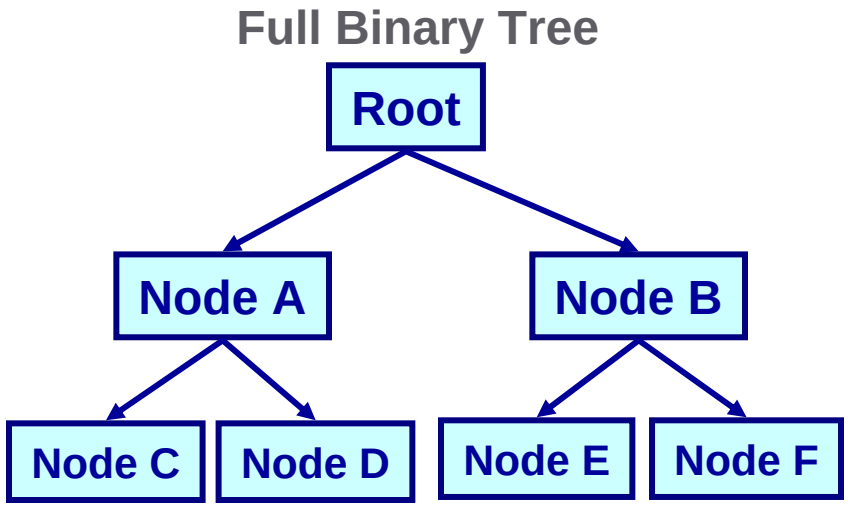
Balanced Trees

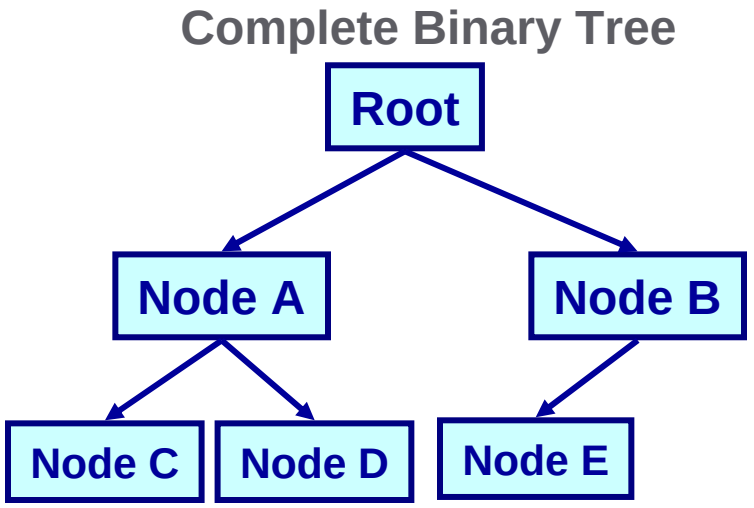
A tree is *balanced* if all of the leaves of the tree are on the same level or within one level of each other



Full and Complete Trees

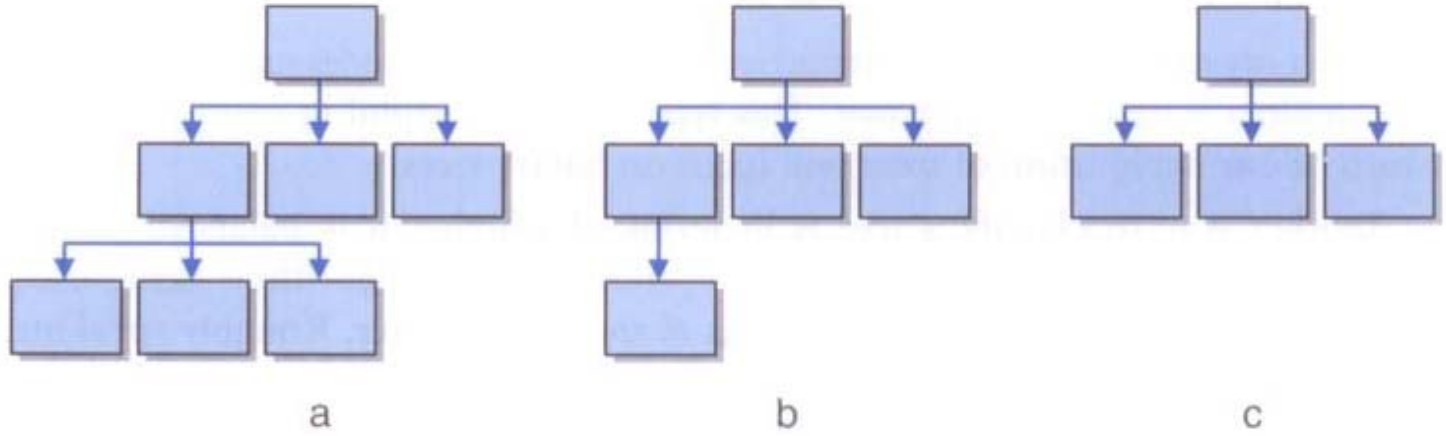
A balanced n-ary tree with m elements has a height of $\log_n m$
A balanced binary tree with n nodes has a height of $\log_2 n$
An n-ary tree is *full* if all leaves of the tree are at the same height and every non-leaf node has exactly n children
A tree is *complete* if it is full, or full to the next-to-last level with all leaves at the bottom level on the left side of the tree





Full and Complete Trees

Three complete trees:



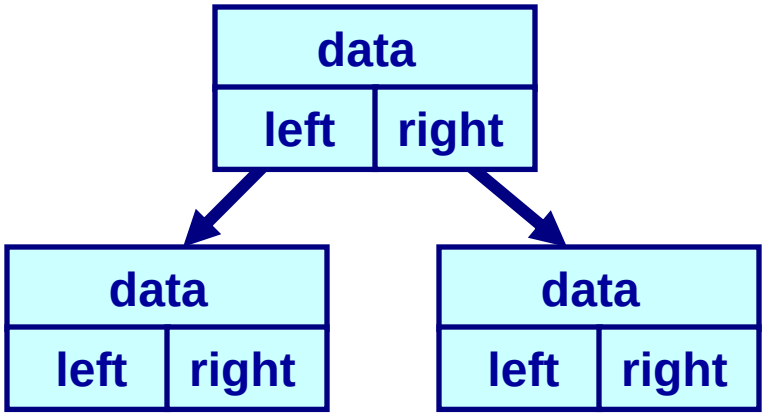
Which trees are full?

Only tree c is full

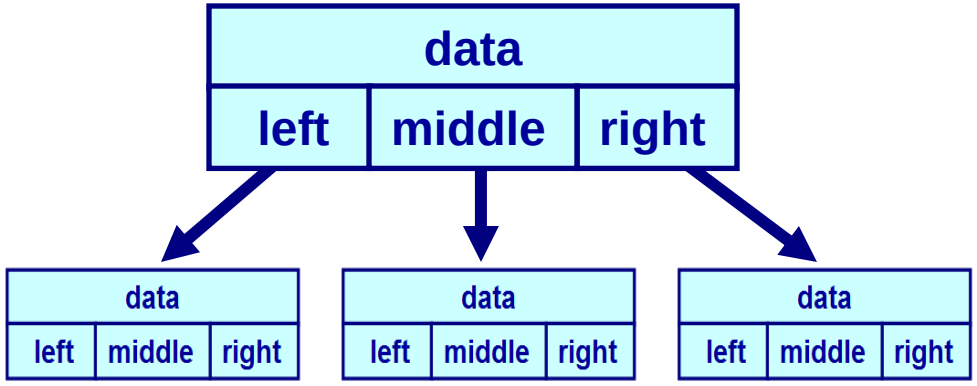
Implementing Trees

An obvious choice for implementing trees is a linked structure

```
struct BinaryTreeNode
{
 int data;
 BinaryTreeNode *left;
 BinaryTreeNode *right;
};
```




```
struct TertiaryTreeNode
{
 int data;
 TertiaryTreeNode *left;
 TertiaryTreeNode *middle;
 TertiaryTreeNode *right;
};
```



Tree Traversals

For linear structures, the process of iterating through the elements is fairly obvious (forwards or backwards)

For non-linear structures like a tree, it is more interesting

Let's look at four classic ways of *traversing* the nodes of a tree

All traversals start at the root of the tree

Each node can be thought of as the root of a subtree

- A tree is a recursive data structure
- Traversing the tree will require a recursive algorithm

Tree Traversals

Preorder :
visit the root, then traverse the subtrees from left to right

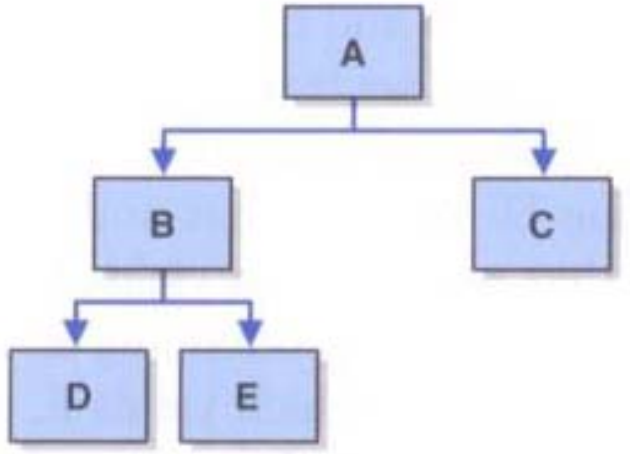
Inorder :
traverse left subtree, then the root, then traverse right subtree

Postorder :
traverse the subtrees from left to right, then visit the root

Level-order :
visit each node at each level of the tree from top (root) to bottom and left to right

Tree Traversals

Preorder: A B D E C
Inorder: D B E A C
Postorder: D E B C A
Level-Order: A B C D E



Tree Traversals

Recursion simplifies the implementation of tree traversals

Preorder:

```
Visit node
Traverse (left child)
Traverse (right child)
```

Inorder:

```
Traverse (left child)
Visit node
Traverse (right child)
```

Postorder:

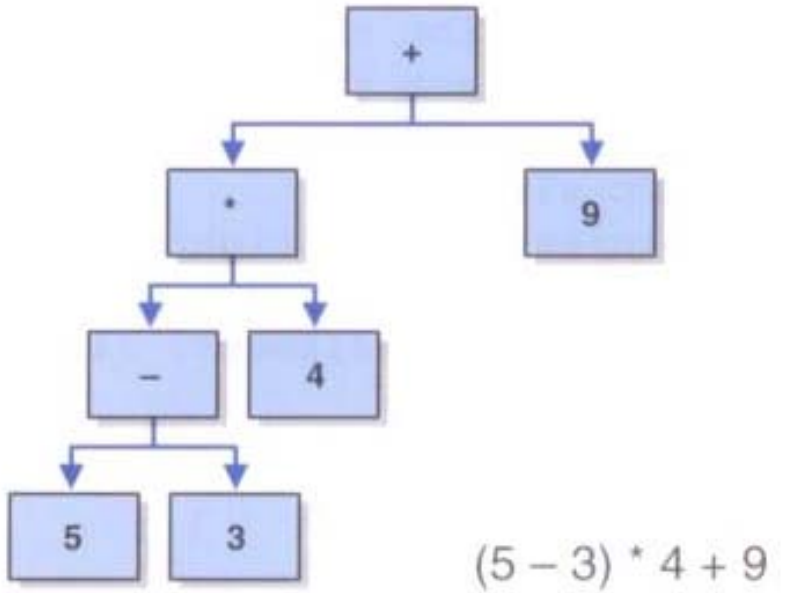
```
Traverse (left child)
Traverse (right child)
Visit node
```

Level-order traversal is more complicated and requires queues

Expression Trees

An *expression tree* is a tree that shows the relationships among operators and operands in an expression

An expression tree is evaluated from the bottom up



Decision Trees

A *decision tree* is a tree whose nodes represent decision points, and whose children represent the options available

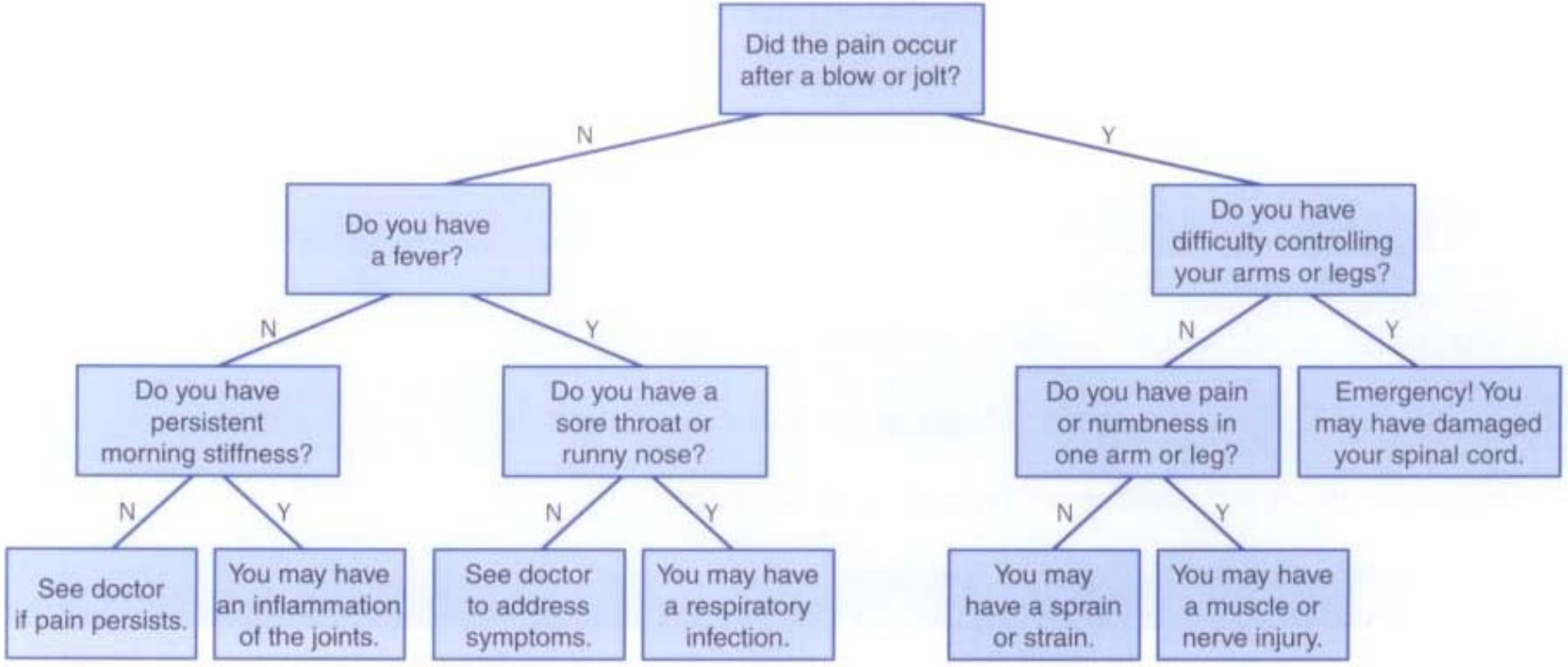
The leaves of a decision tree represent the possible conclusions that might be drawn

A simple decision tree, with yes/no questions, can be modeled by a binary tree

Decision trees are useful in diagnostic situations (medical, car repair, etc.)

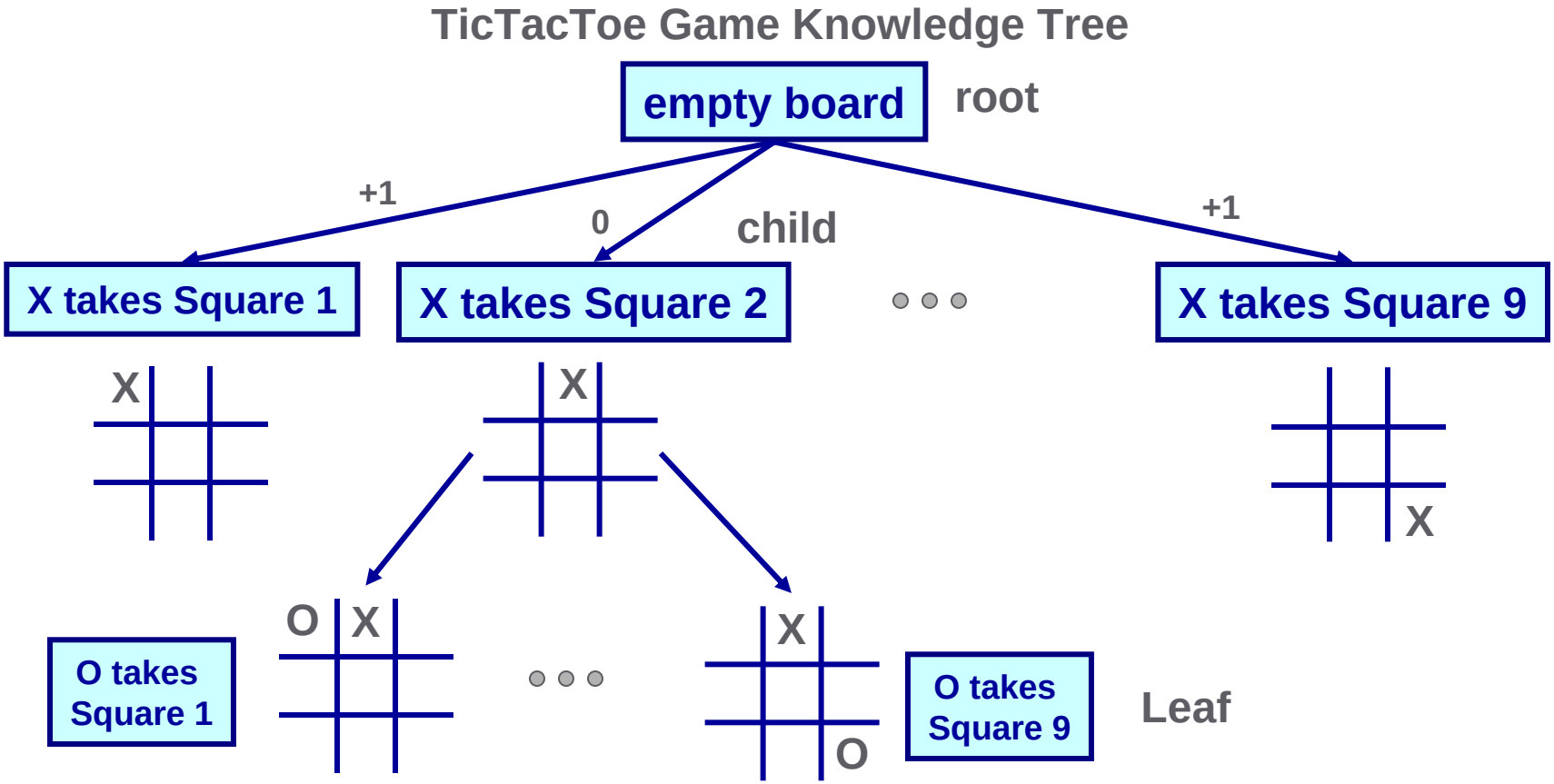
Decision Trees

A simplified decision tree for diagnosing back pain:



Using Trees to Represent Knowledge - TicTacToe

As was seen with LearningPlayer in TicTacToe,
Trees can be used to represent and capture knowledge.



Key Things to take away:

Trees:

- A tree is a nonlinear structure whose elements form a hierarchy
- Can be stored in an array using simulated link strategy
- Trees can be balanced or non-balanced
- A balanced X-ary tree with n elements has height $\log_x n$
- Four basic tree traversal methods: preorder, inorder, postorder, level order
- Preorder: visit the node first, then its children left to right
- Inorder: visit the left child, then the node, then its right child
- Postorder: visit the children left to right, then visit the node itself
- Level-Order: visit all nodes in a level left to right, starting with the root
- Decision Trees can be read from files and used to create an expert system
- Execution Trees can be used to Describe Recursive program flow
- Trees can be used to store and update knowledge for training AI programs

Binary Search Trees

Binary search tree processing
Using BSTs to solve problems
BST implementations
Strategies for balancing BSTs

Binary Search Trees

A *search tree* is a tree whose elements are organized to facilitate finding a particular element when needed

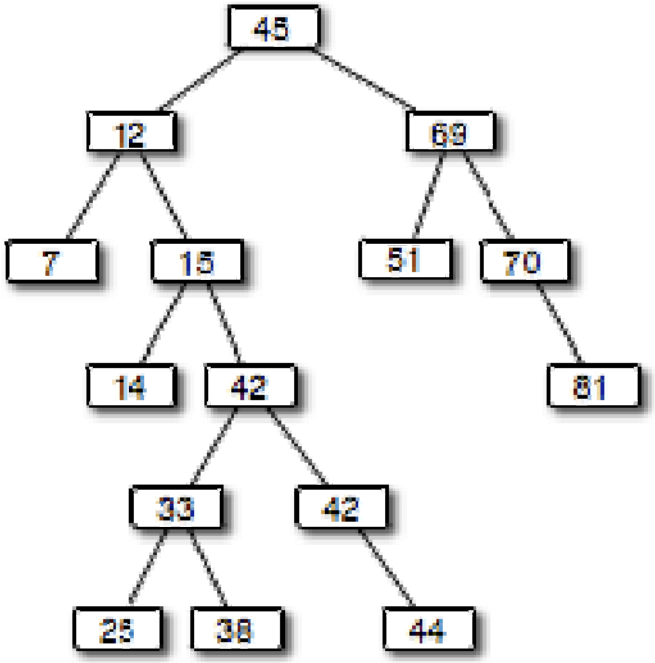
A *binary search tree* is a binary tree that, for each node *n*

- the left subtree of *n* contains elements less than the element stored in *n*
- the right subtree of *n* contains elements greater than or equal to the element stored in *n*

This is *how*

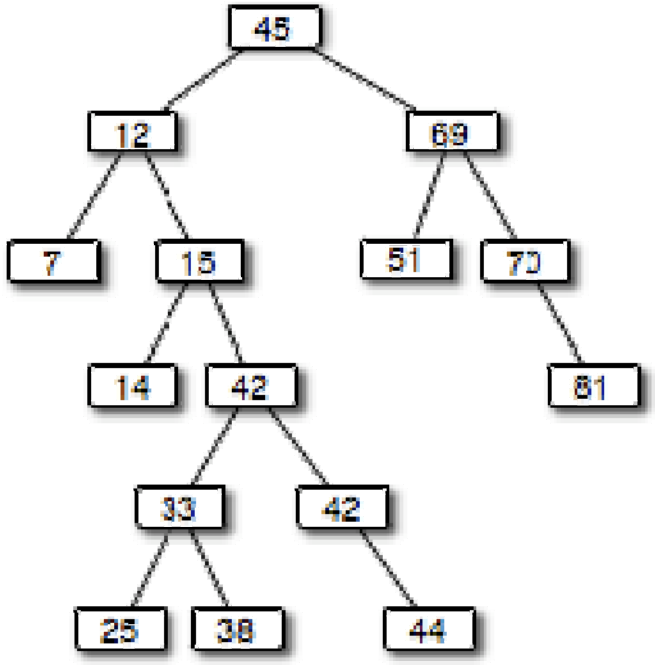
Google

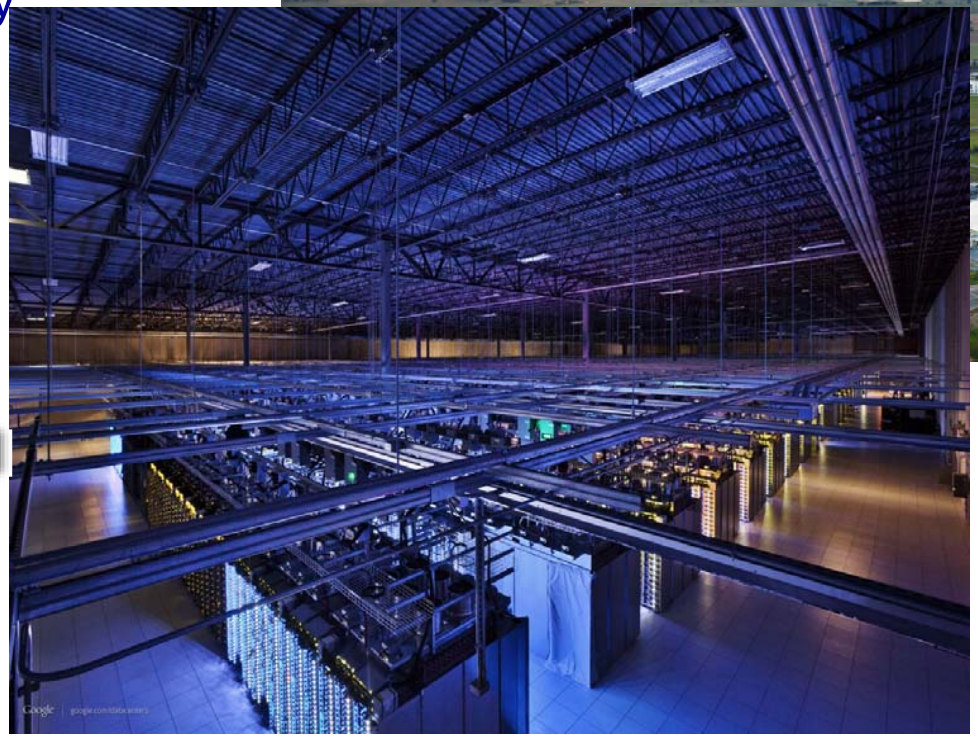
finds your websites!



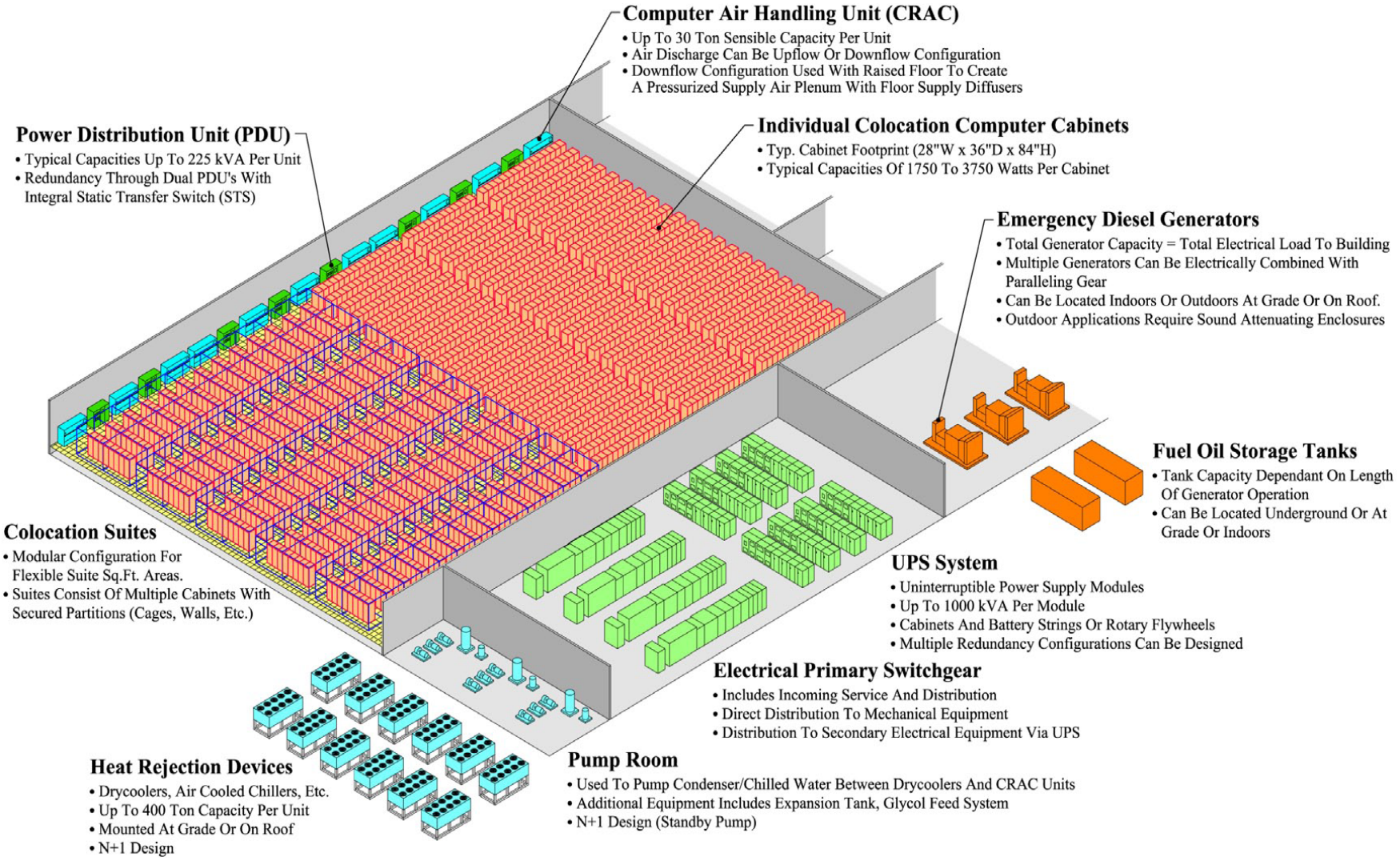
How Google finds Websites

Web Crawlers search the Internet for new websites
Read every webpage and every word
HUGE files of data – Petabytes!
Data Centers process this data
And Update Google Search Trees
Every webpage, every day





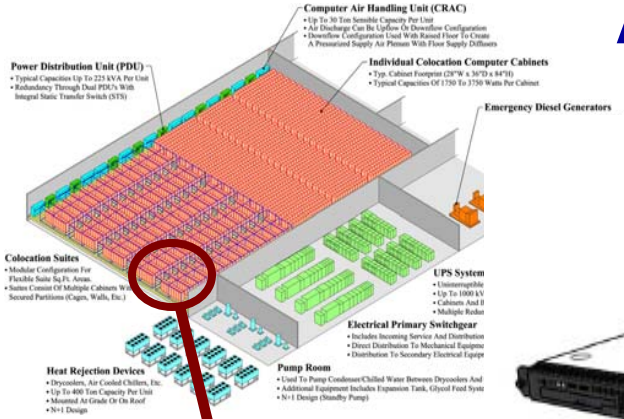
Google and Amazon Data Centers

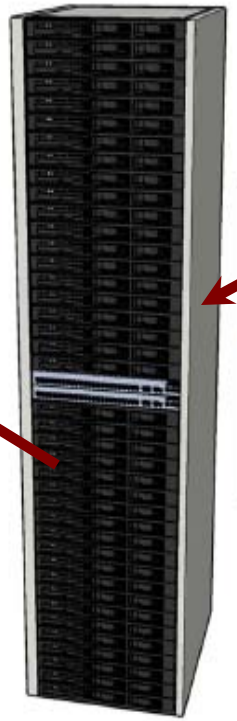


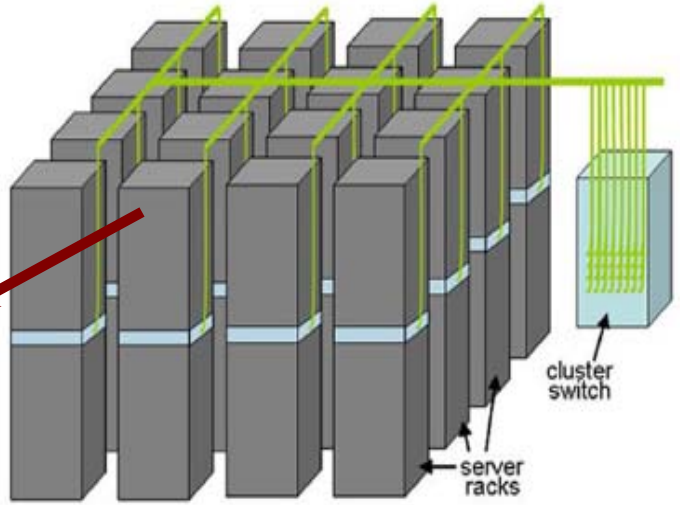
Data Centers – A Closer Look

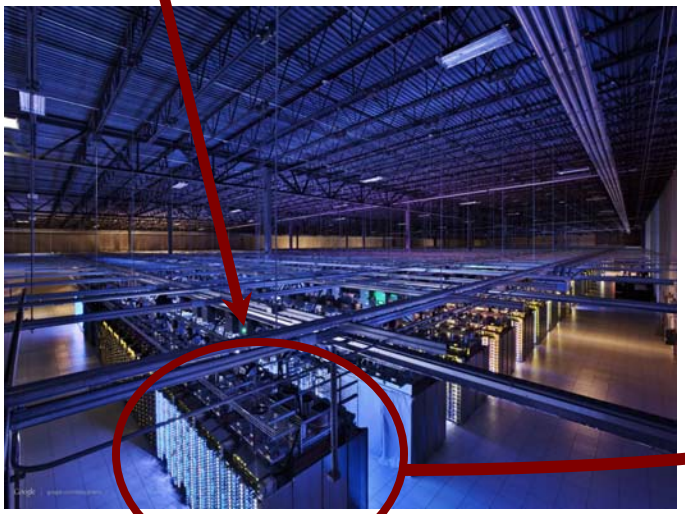
All this just to update some search trees

Some VERY BIG search trees!









For more info:
Dr. Mohamed Hefeeda
Big-Data and Multimedia

Binary Search Trees

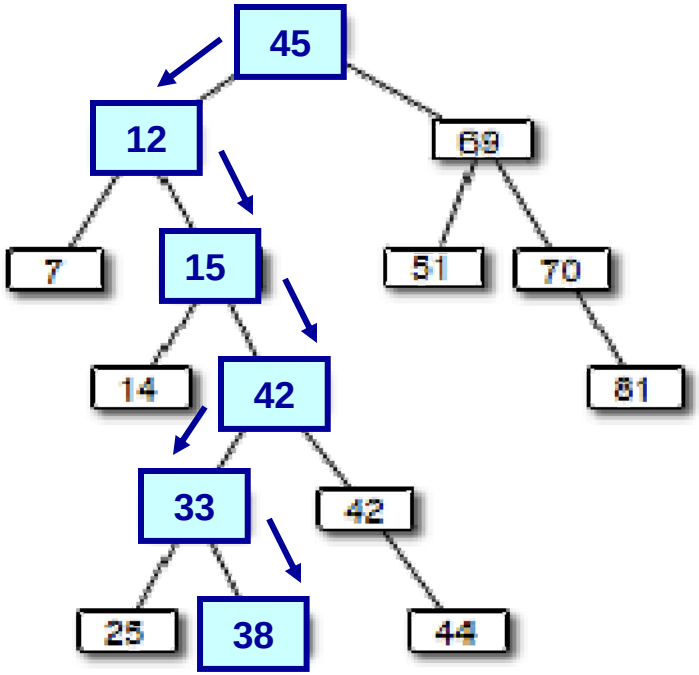
To determine if a particular value exists in a tree

- start at the root
- compare target to element at current node
- move left from current node if target is less than element in the current node
- move right from current node if target is greater than element in the current node

We eventually find the target or hit the end of a path (target is not found)

How to find node with *key* value 38?

- **Start at Root and compare 38 to 45**
- **38 < 45 so go to left subtree**
- **38 > 12 so go to the right**
- **38 > 15 so go to the right again**
- **38 < 42 so go left**
- **38 > 33 so go right**
- **38 found!**
- **Return Object stored at this node**



Binary Search Trees

The particular shape of a binary search tree depends on the order in which the elements are added

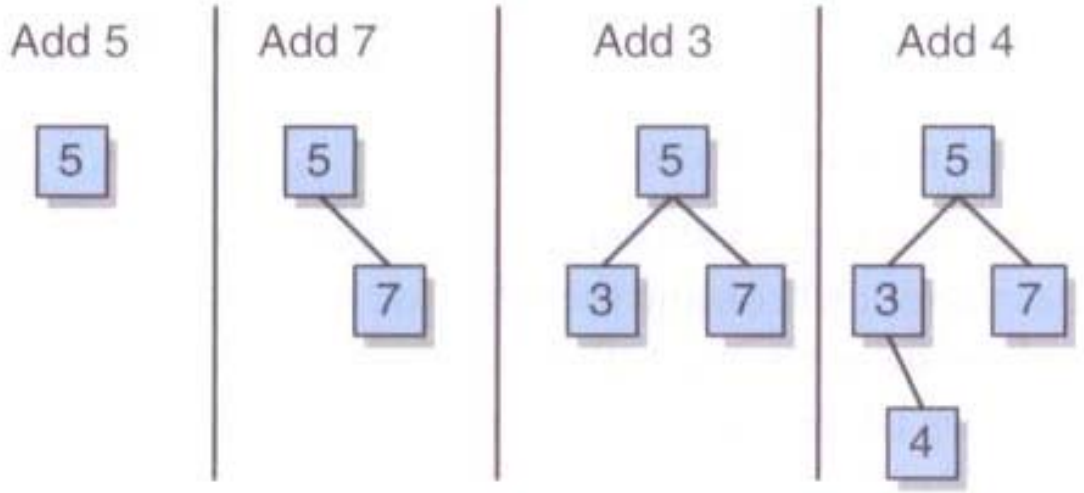
The shape may also be dependant on any additional processing performed on the tree to reshape it

Binary search trees can hold any type of data, so long as we have a way to determine relative ordering

Objects implementing the `Comparable` interface provide such capability

Binary Search Trees

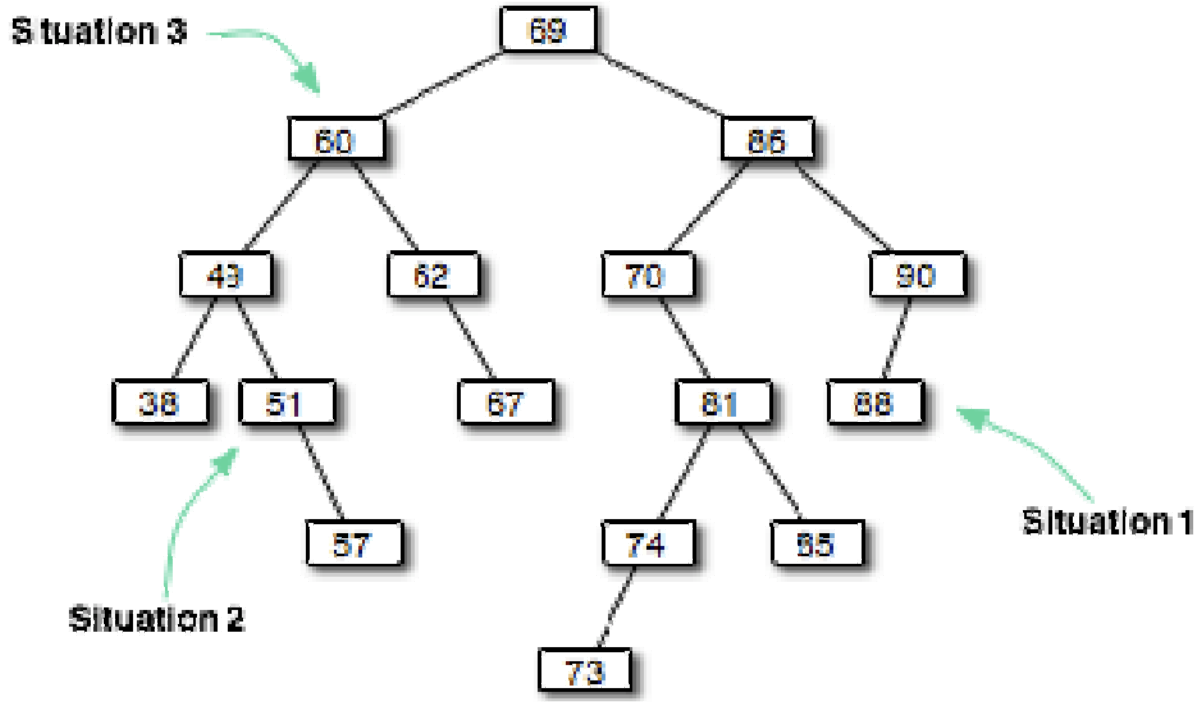
Process of adding an element is similar to finding an element
New elements are added as leaf nodes
Start at the root, follow path dictated by existing elements until you find no child in the desired direction
Then add the new element



BST Element Removal

Removing a target in a BST is not as simple as that for linear data structures
After removing the element, the resulting tree must still be valid
Three distinct situations must be considered when removing an element

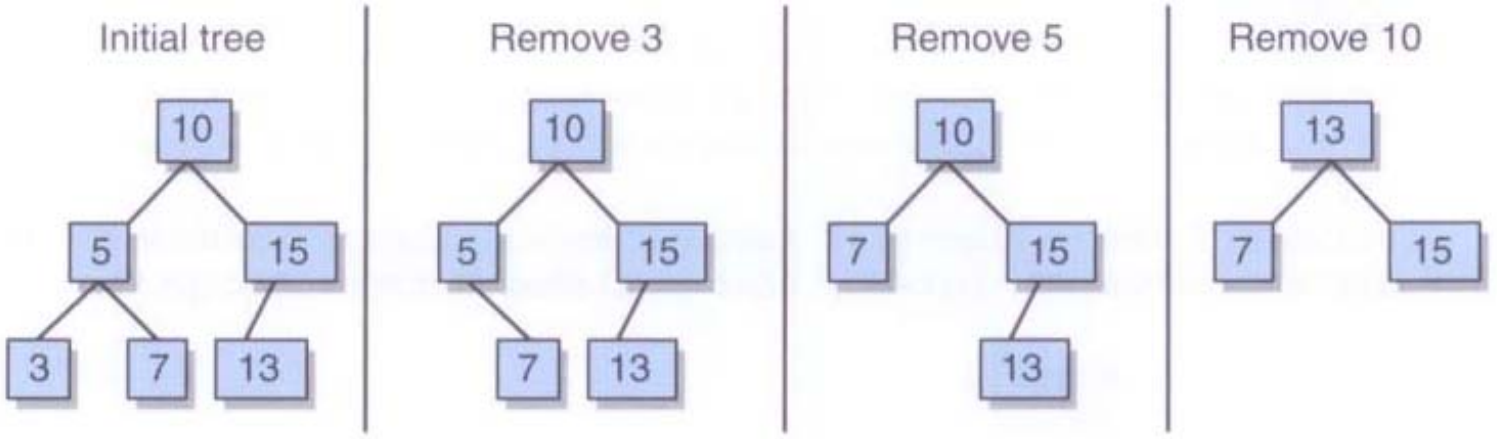
1. Node to remove is a leaf
2. Node to remove has one child
3. Node to remove has two children



BST Element Removal

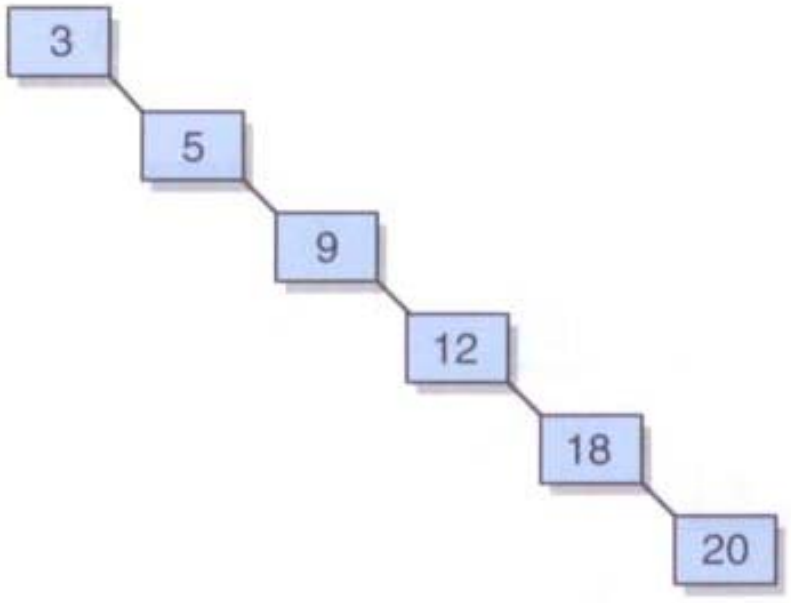
Dealing with the situations

- Node is a leaf: it can simply be deleted
- Node has one child: the deleted node is replaced by the child
- Node has two children: an appropriate node is found lower in the tree and used to replace the node
  - Good choice: *inorder successor* (node that follows in inorder traversal)
  - The inorder successor is guaranteed not to have a left child
  - Thus, removing the inorder successor to replace the deleted node will result in one of the first two situations (it's a leaf or has one child)



Balancing BSTs

As operations are performed on a BST, it could become highly unbalanced (a *degenerate tree*)



Balancing BSTs

AVL trees and red/black trees ensure the BST stays balanced

We will explore *rotations* – operations on binary search trees to assist in the process of keeping a tree balanced
Rotations do not solve all problems created by unbalanced trees, but show the basic algorithmic processes that are used to manipulate trees

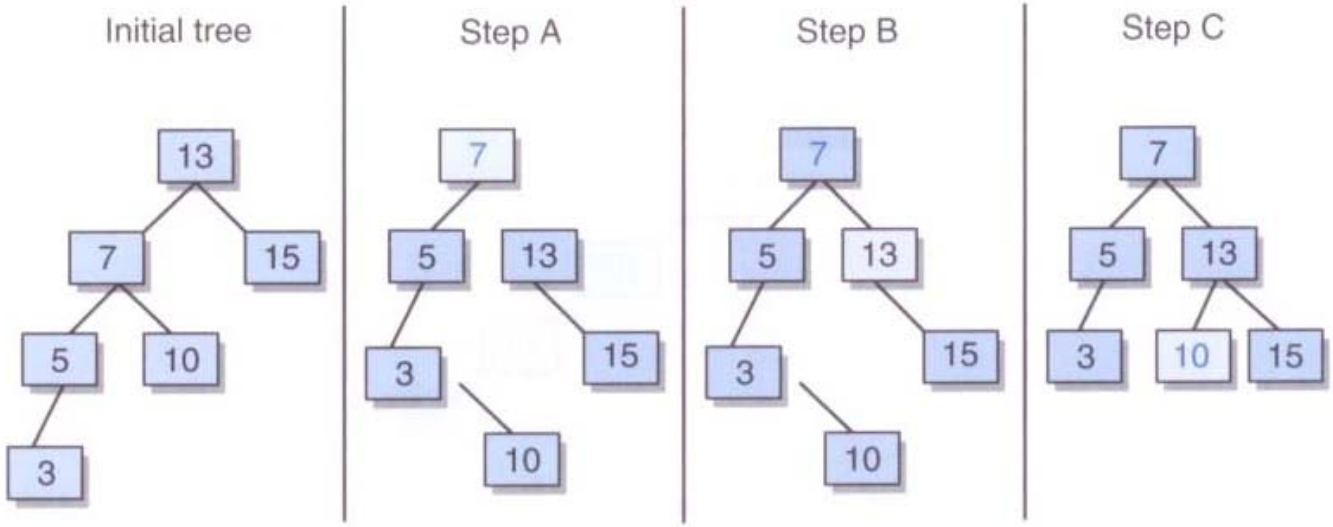
Balancing BSTs

A *right rotation* can be performed at any level of a tree, around the root of any subtree

Corrects an imbalance caused by a long path in the left subtree of the left child of the root

To correct the imbalance

- A: Make the left child element of the root the new root element
- B: Make the former root element the right child element of the new root
- C: Make the right child of what was the left child of the former root, the new left child of the former root



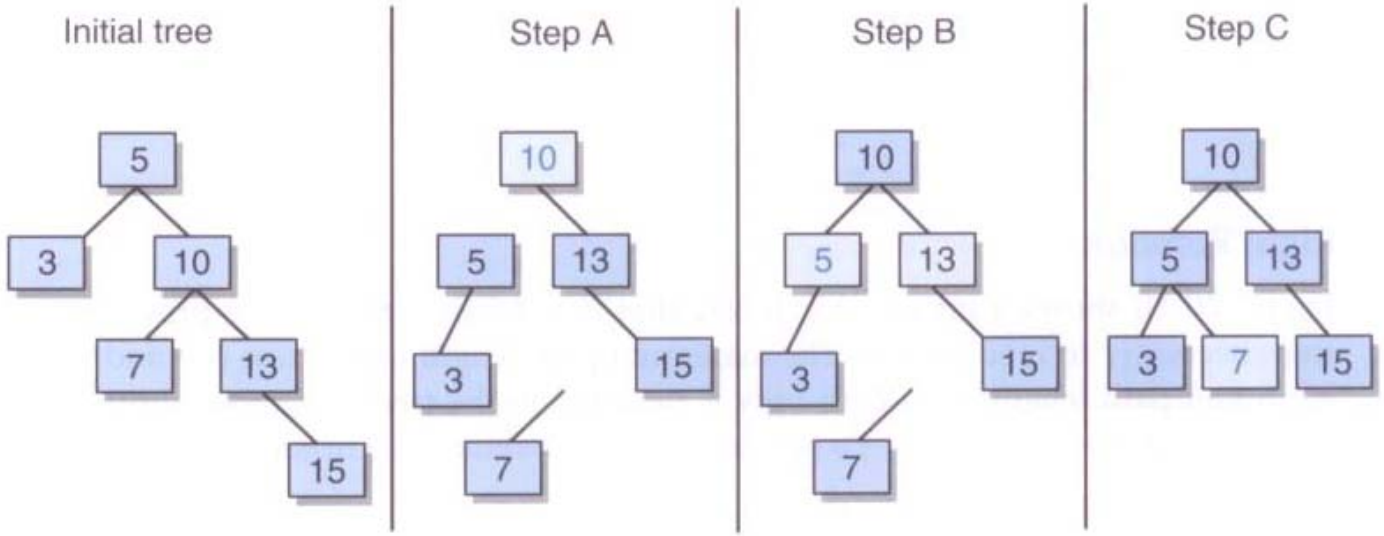
Balancing BSTs

A *left rotation* can be performed at any level of a tree, around the root of any subtree

Corrects an imbalance caused by a long path in the right subtree of the left child of the root

To correct the imbalance

- A: Make the right child element of the root the new root element
- B: Make the former root element the left child element of the new root
- C: Make the left child of what was the right child of the former root, the new right child of the former root



Key Things to take away:

Trees – Part 2:

- CMPT 135 introduces trees and tree algorithms
- Will go into more depth in CMPT225 and beyond
- A binary search tree is a binary tree with the added properties that:
    - the left child's key is ***less*** than the parent's key value
    - the right child's key is ***more*** than the parent's
    - this is a recursive definition which results in maintaining sorted order and allows for $O(\log_x N)$ searches
- Trees are used to provide efficient implementations for other collections
- Trees are critical to making the Internet Searching work

References:

1. J. Lewis, P. DePasquale, and J. Chase., *Java Foundations: Introduction to Program Design & Data Structures*. Addison-Wesley, Boston, Massachusetts, 3rd edition, 2014, ISBN 978-0-13-337046-1