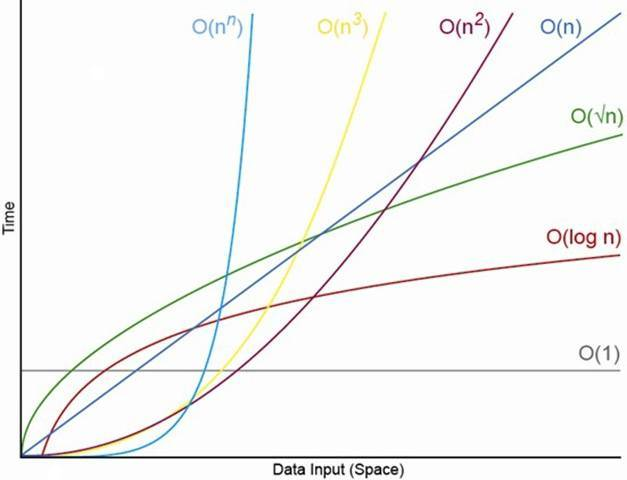
Beck 345 midterm study guide

Big ‘oof’ notation



* Coefficients and lower order terms become increasingly less relevant as n increases.
* Big Omega is the inverse of Big O thus, Omega(f(x)). Big omega represents the fastest possible running time.
* Big Theta means that one function is Big Theta of another if and only if that function is both Big O and Big Omega of it. Big Theta describes a tight bound and represents the strongest asymptotic statement we can make.

Growth Functions

* Optimize efficiency
  + Time complexity (CPU time)
  + Space complexity (Memory Space)
  + CPU time is general focus
* Growth function shows the relationship between the size of the problem (n) and the value optimized (time).
* Faster processors do not make efficient algorithms less important, a faster CPU helps but is not relative to the dominant term.

Asymptotic Complexity

* Asymptotic Complexity is based on the dominant term of the growth function – the term that increase more quickly as n increases.

Analyzing Nested Loops

* When loops are nested, we multiply the complexity of the outer loop by the complexity of the inner loop.
  + For (int count = 0; count < n; count++)
    - For (int count2 = 0; count2 < n; count2++) {//Some sequence of 0(1) steps}
  + Both inner and outer loop have complexity of O(n)
  + The overall efficiency is O(n^2)

Collections

* A collection is an object that holds and organizes other objects.
  + Provides operations for accessing and managing its elements
  + Some are linear in nature and others are non-linear
* Linear
  + It traverses data elements sequentially in which only one element can directly be reached
    - Arrays (elements can be accessed directly by their position)
    - Strings
    - Stack
    - Queue
    - Lists (LinkedList can be accessed using pointer Variable)
* Non-linear
  + The data items are not arranged sequentially
    - Tree
    - Graph
* Stack
  + A stack is a linear collection whose elements are added in a last in, first out (LIFO) manner
  + You cannot reach the middle of the stack
  + You can only add and remove from the top
  + Operations:
    - push(); //adds at the top
    - pop(); //removes from the top
    - peek(); // shows what’s at the top
    - isEmpty(); //Boolean condition true if empty, false if not
    - size(); //returns size of stack
* Queue
  + Collection whose elements are added on one end and removed from the other
  + Labeled as First In, First Out (FIFO)
  + Elements are removed the same order they arrive
  + Can access elements in the middle as well
  + Idea is that of a waiting line eg. Checkout line, cars at light, assembly line
  + Operations:
    - Enqueue //Adds an element to the rear
    - Dequeue //Removes an element from the front of the queue
    - First //Examines the element at the front of queue
    - IsEmpty //Boolean condition true if empty, false if not
    - Size //returns size of queue
    - toString //Returns string representation of queue

Linked Structures

* Uses object references to create links between objects
  + An object reference variable holds the address of an object
* Links could also be used to form more complicated, non-linear structures
* Linked Lists
  + There are no index values built into linked lists
  + Must follow the reference from one node to the next
  + To insert a node at the front of the list, first point the new node to the front, then reassign the front reference
  + To delete the first node, reassign the front reference
  + If the deleted node is need elsewhere, a reference to it must be established before reassigning the front pointer

Abstraction

* An abstraction hides details to make a concept easier to manage
* All objects are abstractions in that they provide well-defined operations (the interface)
* They hide (encapsulate) the object’s data and the implementation of the operations
* Object is good mechanism for implementing a collection

Abstract Data Type

* A data type is a group of values and the operations defined on those values
* An abstract data type is a data type that isn’t pre-defined in the programming language
* A data structure is the set of programming constructs and techniques used to implement a collection
* The distinction between the terms ADT, data structure and collection is sometimes blurred in casual use.

Generics

* (His definition) define collections to hold data of a generic type
* Holds a reference of an object, it’s a placeholder
* Define a class that is based on a generic type
  + Type is referred to generically in the class, specific type is specified only when an object of that class is created.
* Placeholder is specified in angle brackets eg. <T> or <E> are standard
* When the object of the class is created it is instantiated with a specific class instead of the generic type T
* Generics provide better type management control at compile-time and simplify the use of collection classes

Implementing a Queue with an Array

* We treat the array as circular
* A circular array is a regular array that is treated as if it loops back around on itself
* The last index is thought to precede index 0
* We use two integers to keep track of where the front and rear of the queue are at any given time
* At some point, the elements of the queue may straddle the end of the array
* Both the front and rear index values are incremented, wrapping back to 0

List Collections

* A list is a linear collection, like stacks and queues, but is more flexible
* Adding and removing elements in list can occur at either end or anywhere in the middle
* List collections:
  + Ordered lists
    - Elements in an ordered list are ordered by some inherent characteristic of the elements
      * Alphabetical order
      * Scores in ascending order
    - Elements themselves determine where they are stored in the list
  + Unordered lists
    - There is an order to the elements in an unordered list, but that order is not based on element characteristics
    - User determines the order of the elements in the list
    - A new element can be put on the front, rear of the list, or it can be inserted after a particular element already in the list
    - Operations:
      * addToFront //adds element to front of list
      * addToRear //adds element to the rear of the list
      * addAfter //adds an element after a particular element already in the list
  + Indexed lists
    - Elements are referenced by their numeric position in the list [index]
    - There is no inherent relationship among the elements
      * User can determine the order
    - Everytime the list changes, the indexes are updated.
    - Index starts at 0
* List Operations:
  + add(element) //adds element to end of list
    - Ordered list uses this one
  + add(index, element) //inserts element at specific index
  + get(index) //returns element at the specified index
  + remove(index) //removes element at the specified index
  + remove(object) //removes the first occurrence of the specified object
  + set(index, element) //Replaces the element at the specified index
  + size() // returns the number of elements in the list
* Implementing List operations for all types of lists:
  + removeFirst //removes first element
  + removeLast //removes last element
  + remove //removes particular element
  + first //examines the element at the front
  + last //examines the element at the rear
  + contains //determines if list contains particular element
  + isEmpty //checks if list is empty
  + size //determines number of elements in list

Recursion

* Programming technique in which a method can call itself to fulfill its purpose
* A recursive definition is one which uses the word or concept being defined in the definition itself
* Example:
  + Pete and Repeat are on a wall, pete jumps off, who is left, Repeat, thus repeat the statement again and again, etc.
* Application:
  + Solving a maze
  + Towers of Hanoi
    - Using recursion is simple and elegant, but inefficient
    - If we used an iterative solution, it would be much more complex to define and program.

Recursive Programming

* A method in java can invoke itself; if set up that way, it is called a recursive method
* The code of a recursive method must handle both the base case and the recursive case
* Each call sets up a new execution environment, with new parameters and new local variables
* As always, when method completes, control returns to the method that invoked it (which may be another instance of itself)
* Two Cases: Base case, recursive case (can have multiple of the two)

Recursive vs. Iteration

* Recursion shouldn’t be used for all problems
* Ex. not use recursion to solve the sum of 1 to N
  + The iterative version is easier to understand
* Every recursive solution has a corresponding iterative solution
  + A recursive solution may be less efficient
    - Recursion has the overhead of multiple method invocations
  + For some problems recursive solutions are often more simple and elegant to express

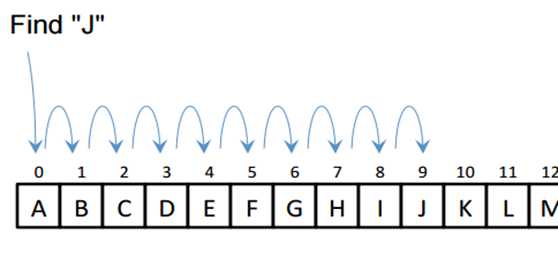
Infinite Recursion

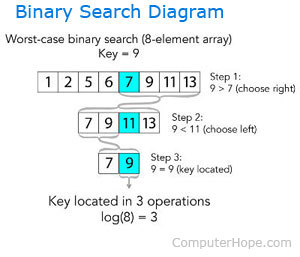
* All recursive definitions must have a non-recursive part
  + If they don’t, there is no way to terminate the recursive path
  + A definition without a non-recursive part cause infinite recursion
* Non-recursive part is called the base case

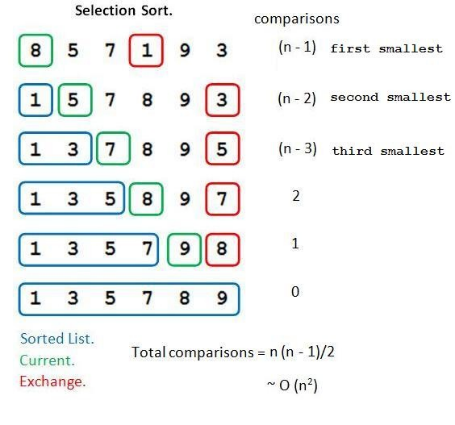
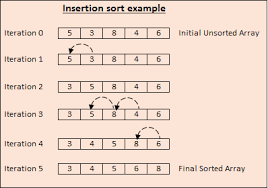
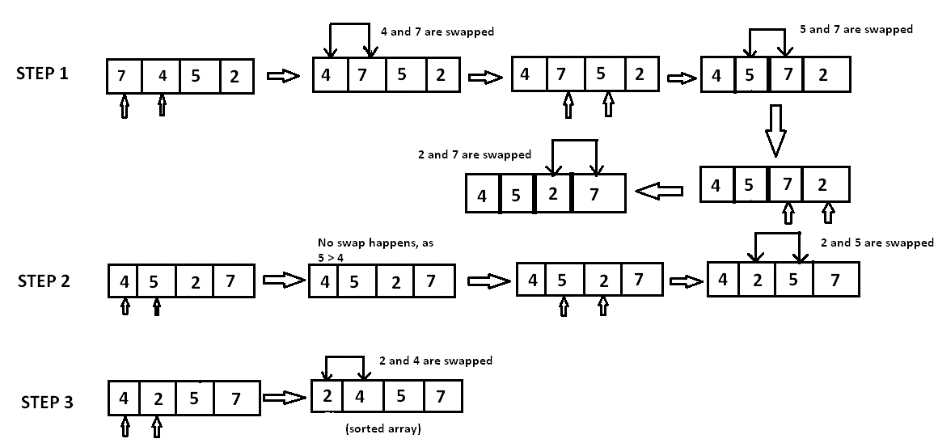
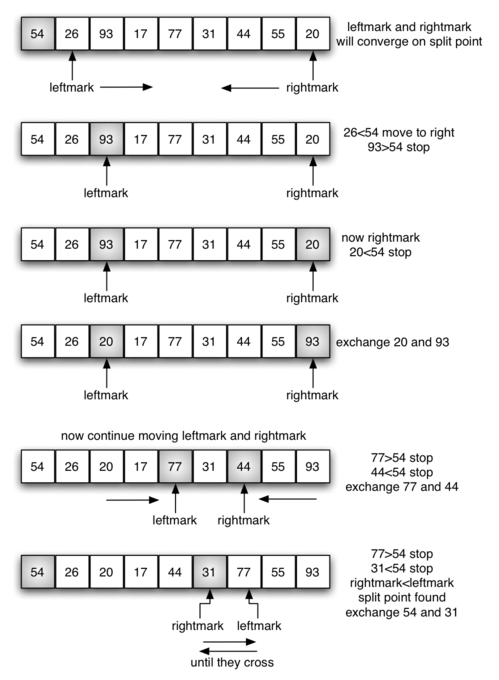
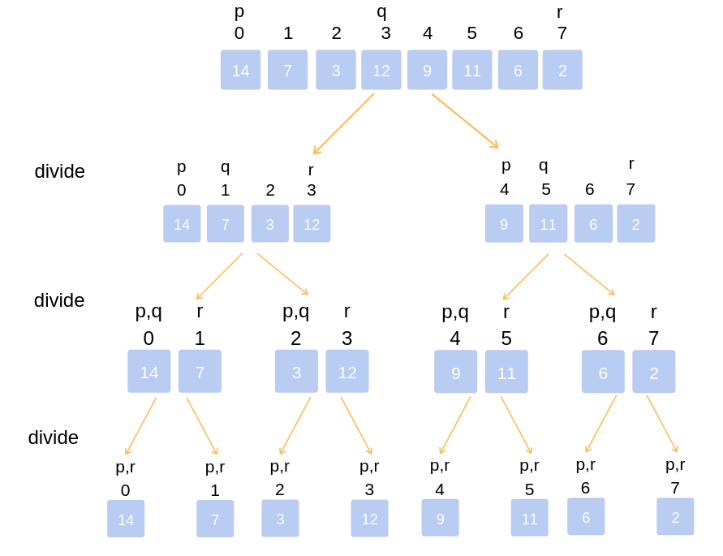
Direct vs. Indirect Recursion

* A method invoking itself is considered to be direct recursion
* A method could invoke another method, which invokes another, etc. until eventually the original method is invoked again
  + M1 could invoke m2, which invokes m3, which invokes m1 again
    - This is called indirect recursion
    - Much more difficult to trace and debug

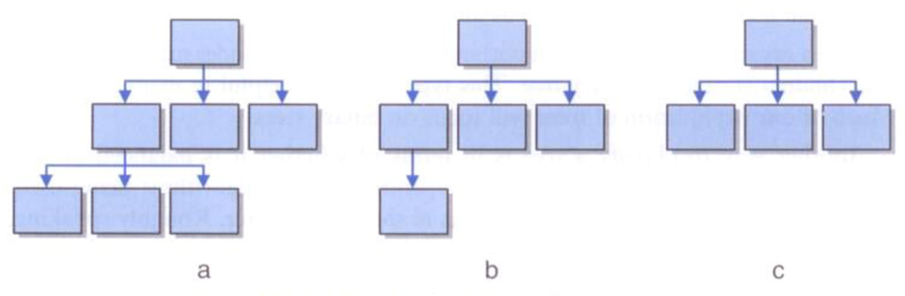
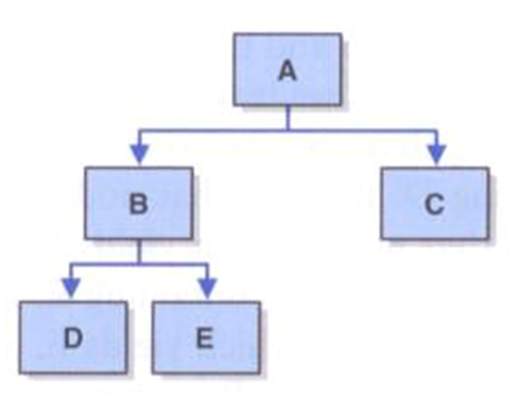
Searching and sorting Algorithms

* Searching
  + is the process of finding a target element among a group of items, the search pool, or determining that it isn’t there.
  + This requires us to repetitively comparing the target to candidates in the search pool
  + We define the algorithms such that they can search any set of objects, we will search objects that implement the COMPARABLE interface
    - We use the COMPARETO method, that returns an integer that specifies the relationship between two objects
      * Object1.compareTo(object2);
      * This call returns a number less than, equal to, or greater that 0 if object1 is less than, equal to, or greater than object2
  + Comparing Search algorithms
    - The expected case for finding an element with a linear search is n/2, which is O(n) 🡨 (constant)
      * Worst case is O(n)
      * Worst case for binary search is (logV2n)/2 comparisons
        + Which makes binary search O(log n)
        + For a binary search to work, the elements must already be sorted
* Sorting
  + The process of arranging a group of items into a defined order based on certain criteria
    - Many have been designed
      * Sequential Sorts – require approximately n^2 comparisons to sort n elements
      * Logarithmic sorts – typically require nlogV2n comparisons to sort n elements
      * With searching, we must be able to compare one element to another
* Searching Algorithms:
  + Linear Search
    - A linear search simply examines each item in the search pool, one at a time, until either the target is found or until the pool is exhausted
      * This approach does not assume the items in the search pool are in any particular order
  + Binary Search
    - A binary search eliminates large parts of the search pool with each comparison
      * For the search to work, the elements must already be sorted
      * Instead of starting the search at one end, we begin in the middle
      * If the target isn’t found, we know that if it is in the pool at all, it is in one half or the other
        + We can then jump to the middle of the halves and search similarly
      * Each comparison in a binary search eliminates half of the viable candidates that remain in the search pool
      * A binary search algorithm is often implemented recursively, each recursive call searches a smaller portion of the search pool
        + The base case is when there a no more viable candidates



* Sorting Algorithms
  + Selection Sort
    - Orders a list of values by repetitively putting a particular value into its final position
      * Specifically:
        + Find the smallest value in the list
        + Switch it with the value in the first position
        + Find the next smallest value in the list
        + Switch it with the value in the second position
        + Repeat until all values are in their proper places
  + Insertion Sort
    - Orders a set of values by repetitively inserting a particular value into a sorted subset of the list
      * Specifically:
        + Consider the first term to be sorted sublist of length one
        + Insert the second item into the sorted sublist, shifting the first item if needed
        + Insert the third item into the sorted sublist, shifting the other items as needed
        + Repeat until all values have been inserted into their proper positions
  + Bubble Sort
    - Orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
      * Specifically:
        + Scan the list, exchanging adjacent elements if they are not in relative order; this bubbles the highest value to the top
        + Scan the list again, bubbling up the second highest value
        + Repeat until all elements have been placed in their proper order
  + Quick Sort
    - Orders values by partitioning the list around one element, then sorting each partition
      * Specifically:
        + Choose one element in the list to be the partition element
        + Organize the elements so that all elements less than the partition element are to the left and all greater are to the right
        + Apply the quick sort algorithm (recursively) to both partitions
  + Merge Sort
    - Orders values by recursively dividing the list in half until each sub-list has one element, then recombining
      * Specifically:
        + Divide the list into two roughly equal parts
        + Recursively divide each part in half, continuing until a part contains only one element
        + Merge the two parts into one sorted list
        + Continue to merge parts as the recursion unfolds
* Comparing the sorts
  + Selection, insertion, and bubble use different techniques but are all O(n^2)
    - They are all based in a nested loop approach
    - For quick sort: if the partition element divides the element in half, each recursive call operates on about half the data
      * The act of partitioning the element at each levels is O(n)
      * The Effect to sort the entire list is O(n log n)
      * If the partition element is written poorly is could go to O(n^2)
    - For Merge sort: the list divides repeatedly in half, which results in the O( log n)
      * The act of merging is O(n)
      * The efficiency of the sort is thus O(n log n)
  + Selection, insertion, and bubble sort are called quadratic sorts
  + Quick Sort and merge sort are called logarithmic sorts

Trees

* A tree is a non-linear structure in which elements are organized into a hierarchy
  + It is comprised of a set of nodes in which elements are stored and edges connect one node to another
  + Each node is on a certain level
  + There is only one root node
  + Nodes at the lower level of a tree are the children of nodes at the previous level
    - A node can only have one parent, but may have multiple children
      * Nodes with the same parent are siblings
  + The root node is the only node that doesn’t have a parent
  + A node that has no children is a leaf node
  + A node this is not the root and has atleast one child is an internal node
  + A sub tree is a tree structure that makes up part of another tree
  + A node is an ancestor of another node if it is above it on the path from the root
  + Order of the tree is classified by the max amount of child nodes it can have
  + General trees have no limit to the number of children a node may have
  + A tree that limit each node to no more than n children is referred to as an n-ary tree
* Balanced Trees:
  + Trees in which may have at most two children are called binary 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 will have a height of logVn m
  + A balanced binary tree with n nodes has a height of logV2 n
  + A 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-to0last level with all leaves at the bottom level on the left side of the tree
    - * A and B are complete trees
      * Only C is a full tree
* Tree traversal
  + For linear structures, the process of iterating through the elements is easy (forward and backwards)
  + For non-linear structures like a tree, traversal is done 4 different ways;
    - Preorder:
      * Visit the root, then traverse the subtree from left to right
    - Inorder:
      * Traverse the left subtree, then visit the root, then travers the 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
      * Level-order is more complicated, it requires the use of extra structures (such as queues and/or lists) to create the necessary order
      * 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
* Binary Tree functions
  + getRoot() 🡪 Returns a reference to the root of the binary tree
  + isEmpty() 🡪 Determines whether the tree is empty
  + size() 🡪 returns the number of elements in the tree
  + contains() 🡪 Determines whether the specified target is in the tree
  + find() 🡪 returns a reference to the specified target element if it is found
  + toString() 🡪 Returns a string representation of the tree
  + iteratorInOrder 🡪 Returns an iterator for an inorder traversal of the tree
  + iteratorPreOrder 🡪 returns an iterator for a preorder traversal of the tree
  + iteratorPostOrder 🡪 Returns an iterator for a postorder traversal of the tree
  + iteratorLevelOrder 🡪 Returns an iterator for a level-order traversald of the tree

Binary Search Trees

* A search tree is a tree whose elements are organized to facilitate finding a particular element when needed
* A BST 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
* The particular shape of a binary search tree depends on the order in which the elements are added
* The shape may also be dependent on any additional processing performed on the tree to reshape it
  + Process of adding an element is similar to finding an element
* 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
* New elements are added as leaf nodes
* We start at the root, follow the path dictated by existing elements until you find no child in the desired direction
  + Then we add the new element
* Binary Search Tree Operations
  + addElement() 🡪 Add an element to the tree
  + removeElement() 🡪 Remove an element from the tree
  + removeAllOccurances() 🡪 Remove all occurrences of element from the tree
  + removeMin() 🡪 Remove the minimum element in the tree
  + removeMax() 🡪 Remove the maximum element in the tree
  + findMin() 🡪 Returns a reference to the minimum element in the tree
  + findMax() 🡪 Returns a reference to the maximum element in the tree

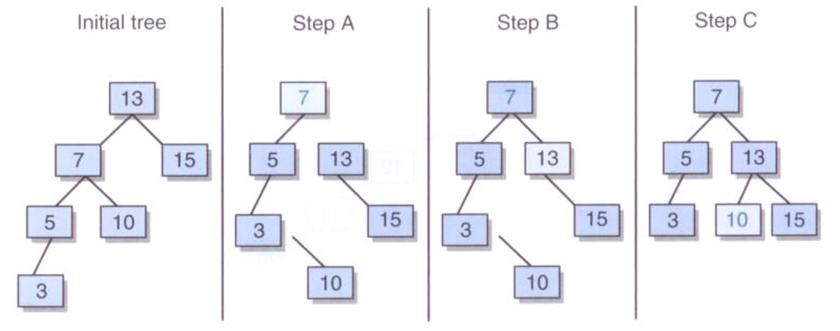
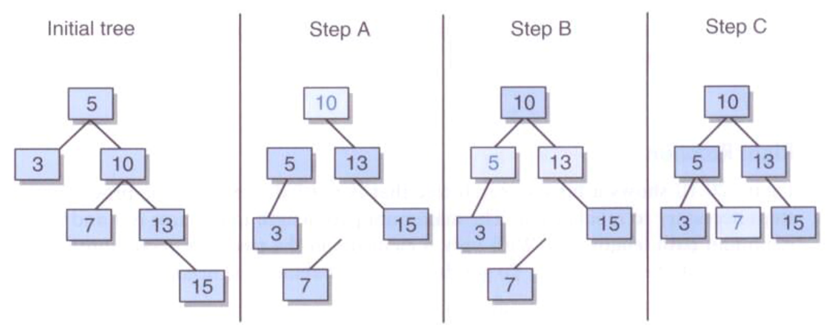
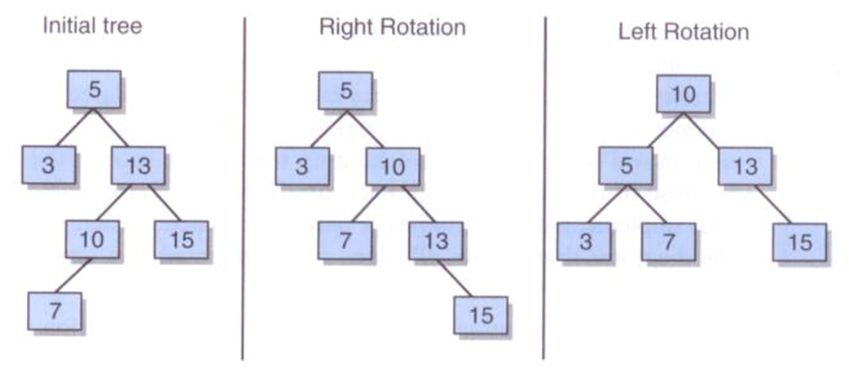
Binary Search Tree Element Removal

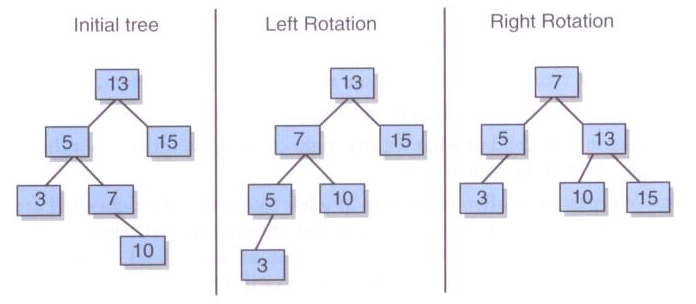
* Dealing with the situations
  + Node is a leaf: It can simply be deleted
  + Node has one child: the deleted node is replaced by the child
  + Node has two children: an appropriate node is found lower in the tree and use the replace the node
    - Good Choice: Inorder successor (node that would follow the removed node in an inorder traversal)
    - This 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 (its leaf or has one child)

Using BST’s to implement Ordered lists

* Operations with both list and BST
  + removeFirst() 🡪 removes the first element from the list
  + removeLast() 🡪 removes the last element from the list
  + remove() 🡪 removes a particular element from the list
  + first() 🡪 Examines the first element at the front of the list
  + last() 🡪 Examines the element at the rear of the list
  + contains() 🡪 Determines if the list contains a particular element
  + isEmpty() 🡪 Determines if the list is empty
  + size() 🡪 determines the number of elements on the list
  + add() 🡪(mostly for order lists) adds an element to the list

Balancing BST’s

* As operations are performed on a BST, it could become highly unbalanced (a degenerate tree) (linear tree, more of line)
* The way we implement our tree does not ensure the BST stays balanced
  + Other approaches do, such as AVL trees and red/black trees
* Rotations assist in the process of keeping the tree balanced
  + They don’t solve all the problems created by unbalanced trees, but instead show the basic algorithmic process that are used to manipulate trees
* 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 node
  + To correct the imbalance:
    - Make the left child element of the root the new root element
    - Make the former root element the right child element of the new root
    - Make the right child of what was the left child of the former root the new left child of the former root
* 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:
    - Make the right child element of the root the new root element
    - Make the former root element the left child element of the new root
    - Make the left child of what was the right child of the former root the new right child of the former root
* If the imbalance is cause by a long path in the left subtree of the right child of the root, we can address it by performing a ‘rightleft rotation’
  + Performing a right rotation around the offending subtree
    - And then performing a left rotation around the root
* If the imbalanced is cause by a long path in the right subtree of the left child of the root we can address it by performing a ‘leftright rotation’
  + Performing a left rotation around the offending subtree
    - And then performing a right rotation around the root

Graphs

* Like trees, graphs are made up of nodes and the connections between those nodes
  + In graph terminology we refer to the nodes as VERTICES and refer to the connections among them as EDGES
  + Vertices are typically reference by a name or label
  + Edges are referenced by a pairing of the vertices (A, B) that they connect
* An UNDIRECTED GRAPH is a graph where the pairings representing the edges are unordered
  + An edge in an undirected graph can be traversed in either direction
  + Two vertices are said to be adjacent if there is an edge connecting them
  + Adjacent vertices are sometimes referred to as neighbors
  + An edge of a graph that connects a vertex to itself is called a self-loop or sling
  + An undirected graph is considered COMPLETE if it has a maximum number of edges connecting vertices
  + A PATH is a sequence of edges that connects two vertices in a graph
    - The LENGTH of a path is in the number of edges in the path (or the number of vertices minus 1)
    - AN undirected graph is considered connected if for any two vertices in the graph there is a path between them
* A CYCLE is a path in which the first and last vertices are the same and none of the edges are repeated
  + A graph that has no cycles is called acyclic
* A DIRECTED GRAPH, sometimes referred to as a digraph, is a graph where the edges are order pairs of vertices
  + This means that the edges (A, B) and (B, A) are separate, directional Edges in a directed graph
  + A path in a directed graph is a sequence of directed edges that connects two vertices in a graph
  + A directed graph is connected if for any two vertices in the graph there is a path between them
  + If a directed graph has no cycles, it is possible to arrange the vertices such that vertex A precedes vertex B if an edges exists from A to B
* A WEIGHTED GRAPH, sometimes called a network, is a graph with weights ( or costs ) associated with each edge
  + The weight of a path in a weighted graph is the sum of the weights of the edges in the path
  + Weighted graphs may be either undirected or directed
  + For weighted graphs, we represent each edge with a triple including the starting vertex, ending vertex and the weight (A, B, 120)
  + We could use an undirected network to represent flights between cities, the weights are the cost
  + A directed version of the graph could show different costs depending on the direction

Common Graph Algorithms

* There are several common algorithms that may apply to undirect, directed, and/or weighted graphs
  + These include:
    - Various Traversal algorithms
    - Algorithms for finding the shortest path
    - Algorithms for finding the least costly path in a network
    - Algorithms for answering simple questions (such as connectivity)
* Graph Traversals
  + There are two main types of traversals for graphs
    - Breadth-first: behaves much like level-order traversal of a tree
      * We can construct this traversal using a queue and an iterator
      * We use the queue to manage the traversal and the iterator to build the result
        + Enqueue the starting vertex, and mark it as “visited”
        + Loop until queue is empty

Dequeue first vertex and add it to iterator

Enqueue each unmarked vertex adjacent to the dequeued vertex

Mark each vertex enqueued as “visited”

* + - Depth-first: behaves much like the preorder traversal of a tree
      * We use the same approach as the breadth-first traversal, but we replace the queue with a stack
      * We do not mark a vertex as visited until it has been added to the iterator
  + Difference:
    - There is no root node present in a graph
    - Graph traversals may start at any vertex in the graph
  + Connectivity:
    - A graph is connected if and only if for each vertex v in a graph containing n vertices, the size of the result of a breadth-first traversal at v is n

Spanning Trees

* A spanning tree is a tree that includes all of the vertices of a graph and some, but possibly not all of the edges
* Since trees are also graphs, for some graphs, the graph itself will be a spanning tree, and thus the only spanning tree

Minimum Spanning Tree

* A minimum spanning tree is a spanning tree where the sum of the weights of the edges is less than or equal to the sum of the weights for any other spanning tree for the same graph
* Algorithm overview:
  + Pick an arbitrary starting vertex and add it to our MST
  + Add all of the edges that include our starting vertex to a minheap ordered by weight
  + Remove the minimum edge from the minheap and add the edge and the new vertex to our MST
  + Add to the minheap all of the edges that include this new vertex and whose other vertex is not already in the MST
  + Continue until the minheap is empty or all vertices are in the MST