California State University, San Bernardino

**Department of Computer Science And Engineering**

# {AlgorithmA}; 2010

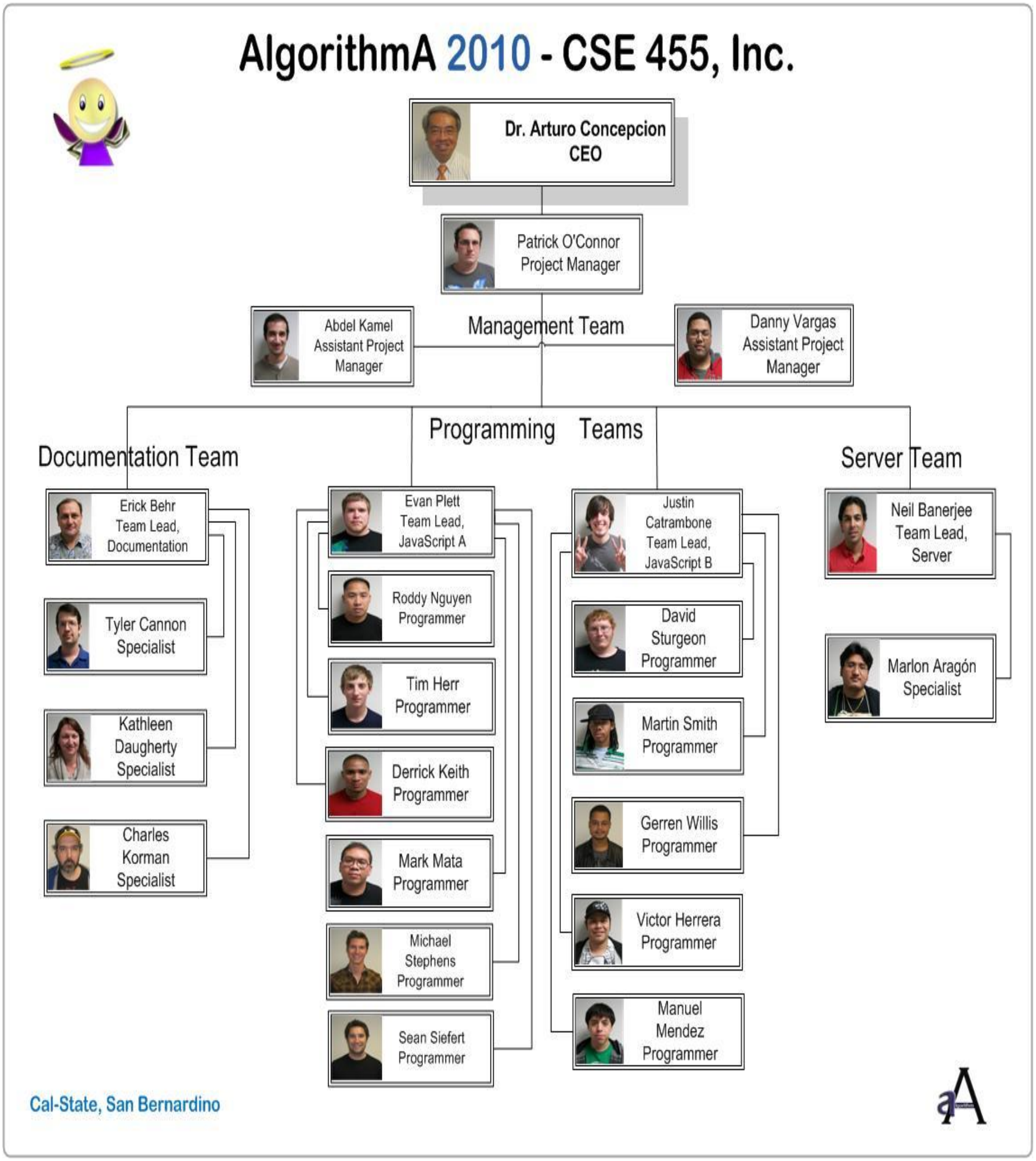
# User's Manual

**CSE 455, Inc.**

CEO: Dr. Concepcion
Project Manager: Patrick O'Connor
Assistant Managers: Danny Vargas and Abdelrahman Kamel
Documentation Team: Erick Behr, Tyler Cannon,
Charles Korma, Kathleen Daugherty

**Published: March 22, 2010**

## Table of Contents


AlgorithmA 2010 - CSE 455, Inc.
Dr. Arturo Concepcion CEO
Patrick O'Connor Project Manager
Management Team
Abdel Kamel Assistant Project Manager
Danny Vargas Assistant Project Manager
Programming Teams
Documentation Team
Server Team
Erick Behr Team Lead, Documentation
Tyler Cannon Specialist
Kathleen Daugherty Specialist
Charles Korman Specialist
Evan Plett Team Lead, JavaScript A
Roddy Nguyen Programmer
Tim Herr Programmer
Derrick Keith Programmer
Mark Mata Programmer
Michael Stephens Programmer
Sean Siefert Programmer
Justin Catrambone Team Lead, JavaScript B
David Sturgeon Programmer
Martin Smith Programmer
Gerren Willis Programmer
Victor Herrera Programmer
Manuel Mendez Programmer
Neil Banerjee Team Lead, Server
Marlon Aragón Specialist
Cal-State, San Bernardino

CSE 455, Inc.

# 1. Introduction

## 1.1 Who is CSE 455, Inc.?

CSE 455, Inc is a fictitious software development company focused to develop educational software for the academic community in Computer Science. This software company is managed by a CEO, a Program Manager, two Assistant Managers, four Team Leads and a group of competent programmers highly qualified in several high-level computer languages.

## 1.2 What is AlgorithmA 2010?

AlgorithmA 2010 is a dedicated website for students of Computer Science to learn the foundation of programming, mathematical algorithms, and data structures. This framework is taught in the first two classes of the Computer Science BS and BA degree programs. AlgorithmA 2010 is a learning tool for these students to help them understand how these mechanisms work in software. AlgorithmA is structured in a way to give its user a detailed account of how an algorithm or data structure works. The walkthroughs go line-by-line and show, using a graph, what happens when each line is executed. This design shows what the algorithm actually does and in what sequence. AlgorithmA 2010 is structured in a way to give its user a detailed account of how an Algorithm or a Data Structure works. Using a graph, each line of pseudo code is concurrently highlighted as it is executed. Parts of the graph are highlighted as the code is executed as well. This design explicitly illustrates what the algorithm actually does and in what sequence.

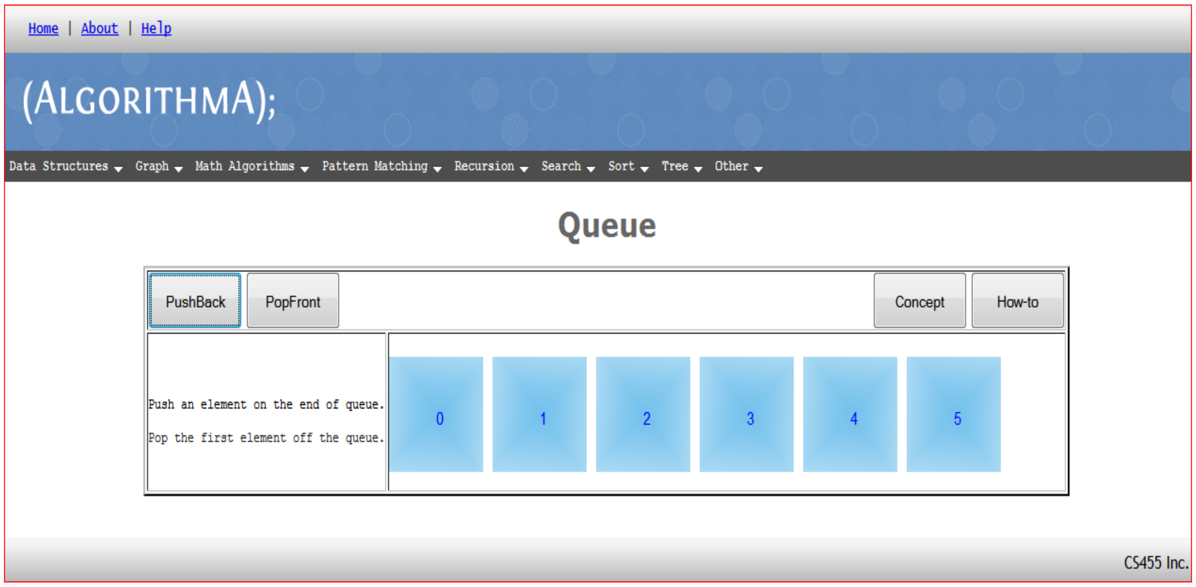
AlgorithmA 2010 is a continuation of the CSE 455 class at California State University, San Bernardino. AlgorithmA first started in 1991. It has gone through yearly iterations that have included updating, adding new algorithms, and reengineering code to more current software design. For 2010 AlgorithmA is being updated from Java to JavaScript. At least 20 animations are being redesigned including the sorting, searching and data structures. The ultimate goal is to have republished the entire project on the open source.

# 2. Data Structures

## 2.1 Queue

### 2.1.1 Overview

A queue is a particular kind of collection in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position and removal of entities from the front terminal position. This makes the queue a First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that whenever an element is added, all elements that were added before have to be removed before the new element can be invoked. A queue is an example of a linear data structure.



Push an element on the end of queue.

Pop the first element off the queue.

**Queue Animation**

### 2.1.2 Layout

The user will be presented with buttons detailing the concept and the how-to of the function and the data structure algorithm to be used in creating a list. To the left of the animation work area will be a pseudo-code walkthrough that is highlighted as instructions are being executed.

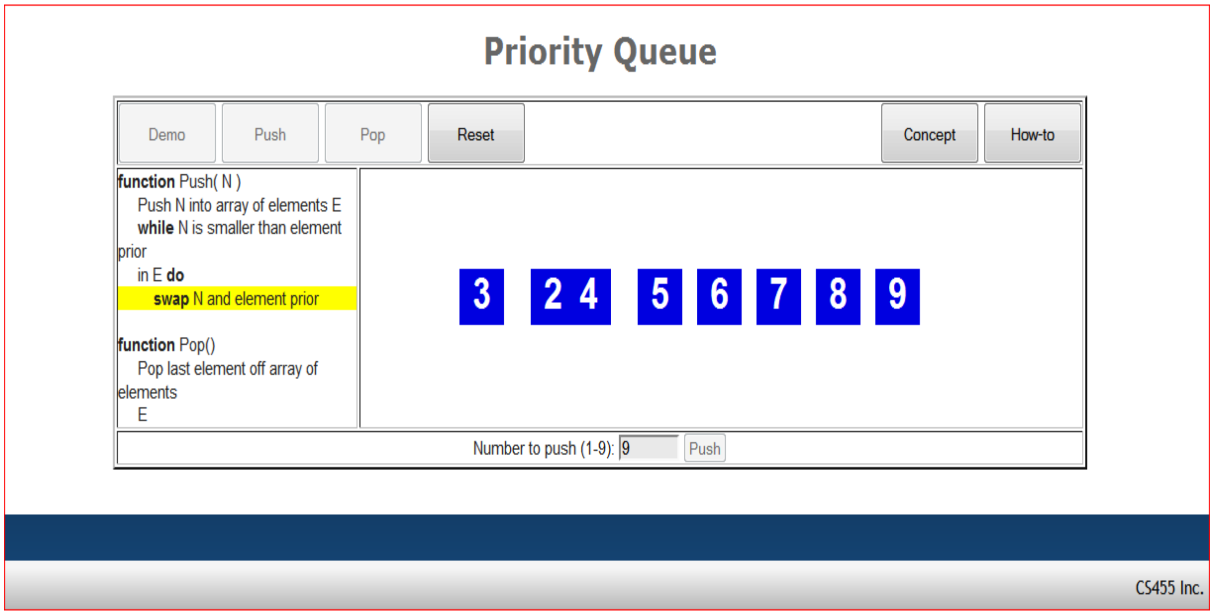
### 2.1.3 Functionality

- Push Back – Push a random element to the structure to the back of the list
- Pop Front – Remove an element from the front of the list

## 2.2 Priority Queue

### 2.2.1 Overview

Priority queue is an abstract data type which efficiently finds the item with the highest priority across a series of operations. One can imagine a priority queue as a modified queue, but when one would get the next element off the queue, the highest-priority one is retrieved first. Stacks and queues may be modeled as particular kinds of priority queues. In a stack, the priority of each inserted element is monotonically increasing; thus, the last element inserted is always the first retrieved. In a queue, the priority of each inserted element is monotonically decreasing; hence, the first element inserted is always the first retrieved.



```
function Push( N )
    Push N into array of elements E
    while N is smaller than element prior
    in E do
        swap N and element prior

function Pop()
    Pop last element off array of elements
    E
```

Number to push (1-9): 9

Priority Queue Animation

### 2.2.2 Layout

The user will be presented with buttons detailing the concept and the how-to of the function and the data structure algorithm to be used in creating a list. To the left of the animation work area will be a pseudo-code walkthrough that is highlighted as actions are performed.

### 2.2.3 Functionality

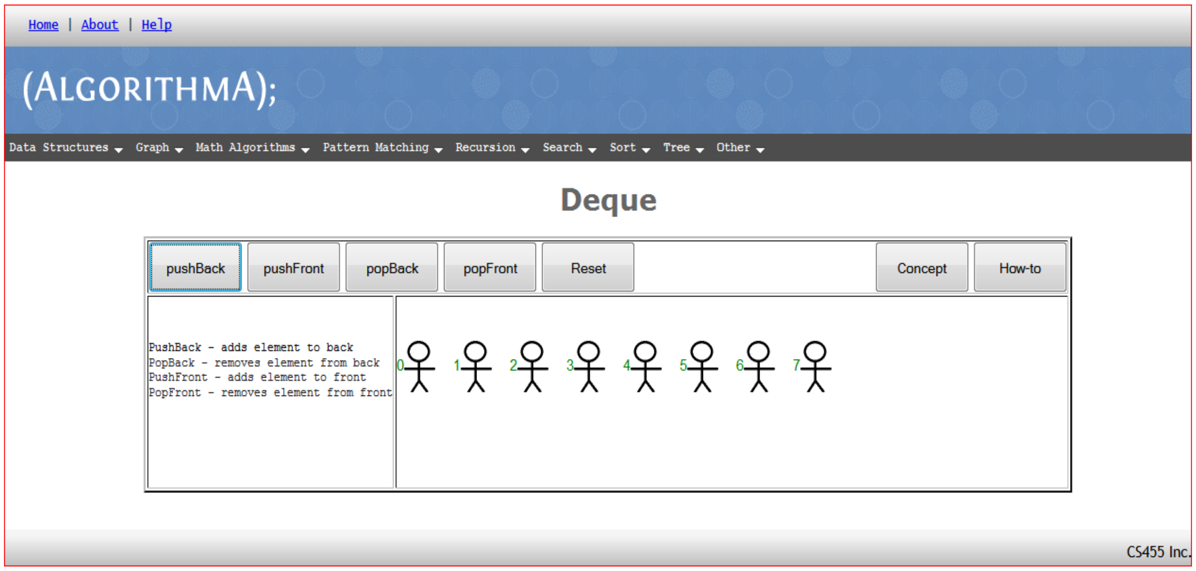
- Demo – Starts the priority queue animation
- Push – Push an element onto the structure of the list
- Sort – Activates the sorting feature
- Prioritize – Organizes the list in ascending order
- Pop – Remove an element from the list
- Reset – Reset to the initial layout when first loaded.

## 2.3 Deque

### 2.3.1 Overview

Deque (double-ended queue) is an abstract data structure that implements a queue for which elements can only be added to or removed from the front (head) or back (tail).

In a doubly-linked list implementation, the time complexity of all Deque operations is O(1). Additionally, the time complexity of insertion or deletion in the middle, given an iterator, is O(1); however, the time complexity of random access by index is O(n).



```
PushBack - adds element to back
PopBack - removes element from back
PushFront - adds element to front
PopFront - removes element from front
```

Deque Animation

### 2.3.2 Layout

The user will be presented with buttons detailing the concept and the how-to of the function and the data structure algorithm to be used in creating a list. To the left of the animation work area will be a pseudo-code walkthrough that is highlighted as actions are performed.

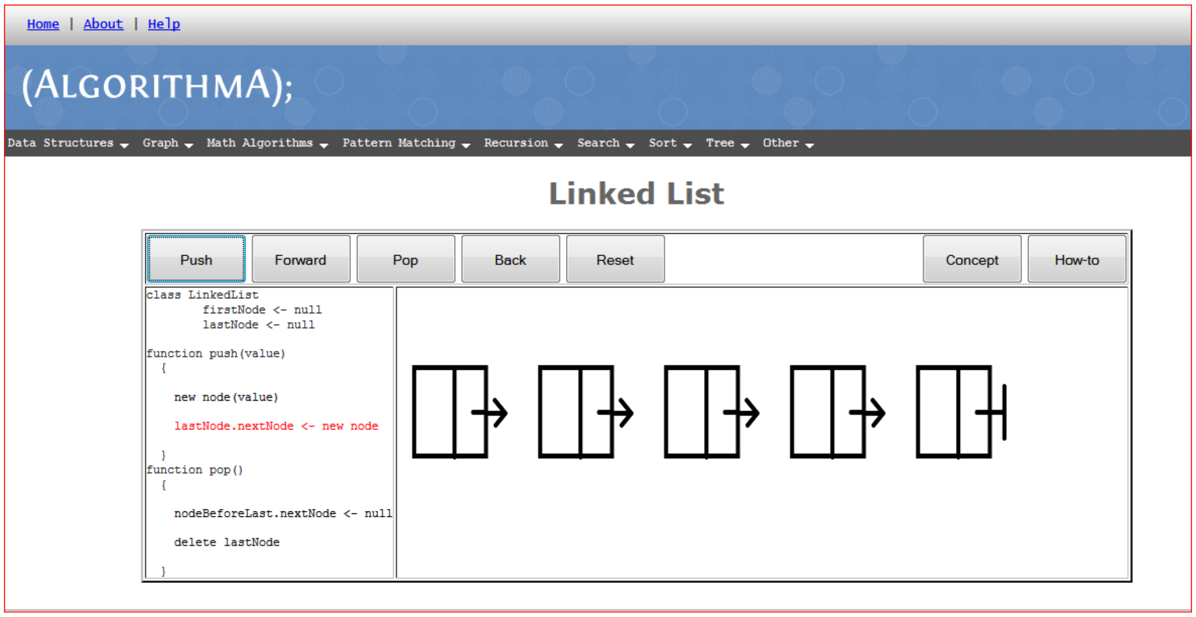
### 2.3.3 Functionality

- Push Front – Push a random element to the structure to the front of the list
- Push Back – Push a random element to the structure to the back of the list
- Pop Front – Remove an element from the front of the list
- Pop Back – Remove an element from the back of the list
- Reset – Reset to the initial layout when first loaded.

## 2.4 Linked List

### 2.4.1 Overview

Linked List is a data structure that consists of a sequence of data records such that in each record there is a field that contains a reference to the next record in the sequence. Linked Lists are among the simplest and most common data structures, and are used to implement many important abstract data structures such as stacks, queues, hash tables, symbolic expressions, skip lists, and many more. Each record of a linked list is often called an element or node. The field of each node that contains the address of the next node is usually called the next link or next pointer. The remaining fields are known as the data, information, value, or payload fields. The head of a list is its first node, and the tail is the list minus that node. In LISP and some derived languages, the tail may be called the CDR of the list, while the payload of the head node may be called the CAR.



```
class LinkedList
        firstNode <- null
        lastNode <- null

function push(value)
  {

    new node(value)

    lastNode.nextNode <- new node

  }
function pop()
  {

    nodeBeforeLast.nextNode <- null

    delete lastNode

  }
```

Linked List Animation

### 2.4.2 Layout

The user will be presented with buttons detailing the concept and the how-to of the function and the data structure algorithm to be used in creating a list. To the left of the animation work area will be a pseudo-code walkthrough that is highlighted as actions are performed.

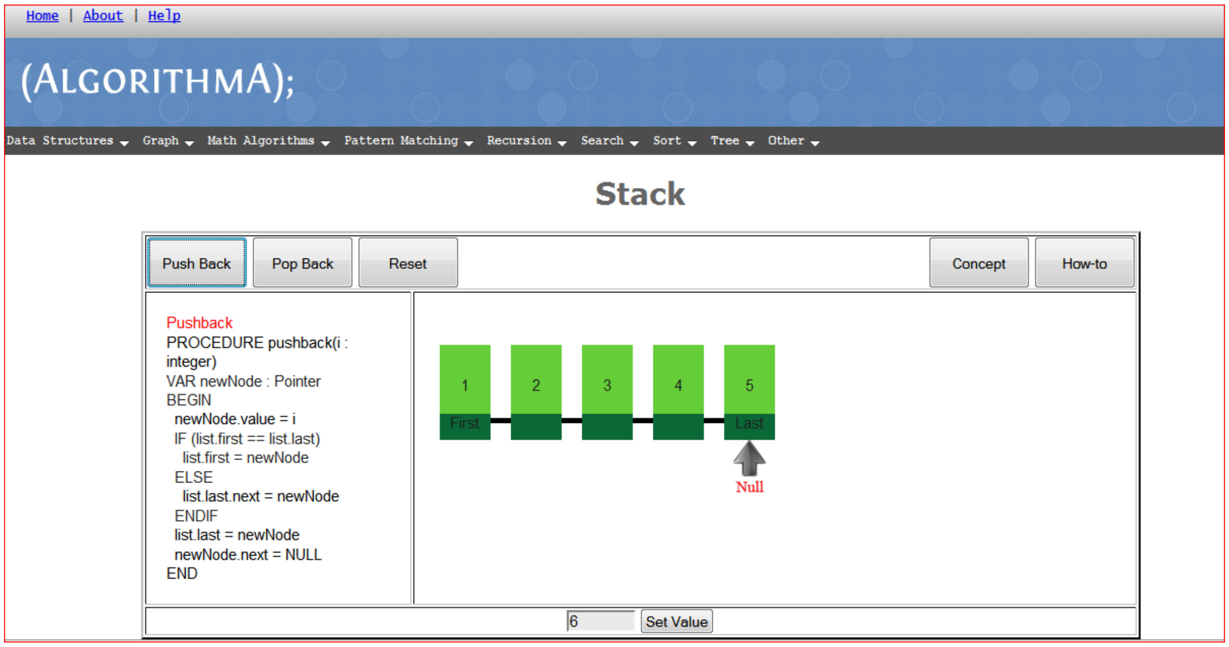
### 2.4.3 Functionality

- Push – Push an element onto the structure of the list
- Forward – Steps through the pseudo code one line at a time in a forward direction
- Pop – Remove an element from the list
- Back – Steps through the pseudo code one line at a time in a backward direction
- Reset – Reset to the initial layout when first loaded.

## 2.5 Stack

### 2.5.1 Overview

Stack is a last in, first out (LIFO) abstract data type and data structure. A stack can have any abstract data type as an element, but is characterized by only two fundamental operations: push and pop. The push operation adds to the top of the list, hiding any items already on the stack, or initializing the stack if it is empty. The pop operation removes an item from the top of the list, and returns this value to the caller. A pop either reveals previously concealed items, or results in an empty list.  A stack is a restricted data structure, because only a small number of operations are performed on it. The nature of the pop and push operations also mean that stack elements have a natural order. Elements are removed from the stack in the reverse order to the order of their addition: therefore, the lower elements are typically those that have been in the list the longest.



Pushback

```
PROCEDURE pushback(i : integer)
VAR newNode : Pointer
BEGIN
  newNode.value = i
  IF (list.first == list.last)
    list.first = newNode
  ELSE
    list.last.next = newNode
  ENDIF
  list.last = newNode
  newNode.next = NULL
END
```

Stack Animation

### 2.5.2 Layout

The user will be presented with buttons detailing the concept and the how-to of the function and the data structure algorithm to be used in creating a list. To the left of the animation work area will be a pseudo-code walkthrough that is highlighted as actions are performed.

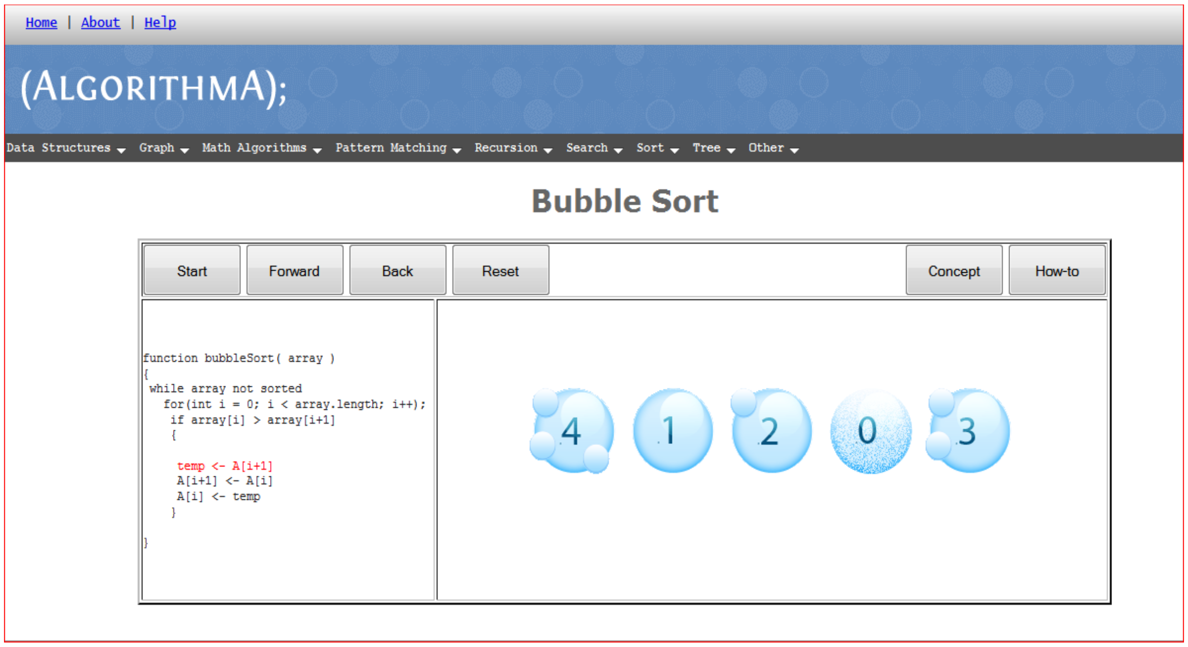
### 2.5.3 Functionality

- Push Back – Push an element onto the structure to the back of the list
- Pop Back – Remove an element from the back of the list
- Reset – Reset to the initial layout when first loaded.

# 3. Sorting Algorithms

## 3.1 Bubble Sort - Overview

The bubble sort is the oldest and simplest sort in use. Unfortunately, it's also the slowest. The bubble sort works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items (in other words, all items are in the correct order). This causes larger values to "bubble" to the end of the list while smaller values "sink" towards the beginning of the list. It is generally considered to be the most inefficient sorting algorithm in common usage. Under best-case conditions (the list is already sorted), the bubble sort can approach a constant O ( n ) level of complexity. General-case is an abysmal O ( $n^2$ ).



```
function bubbleSort( array )
{
 while array not sorted
   for(int i = 0; i < array.length; i++);
    if array[i] > array[i+1]
    {

     temp <- A[i+1]
     A[i+1] <- A[i]
     A[i] <- temp
    }

}
```

Bubble Sort Animation

### 3.1.1 Layout

Bubble sort will be classified under Sort. When Bubble sort is selected from sort the basic layout will be displayed. The animation will be displayed by default with an option to select viewable source code. Start, Reset, Pause, Forward, Step check box, Animation check box and Speed setting bar will be displayed.

### 3.1.2 Functionality

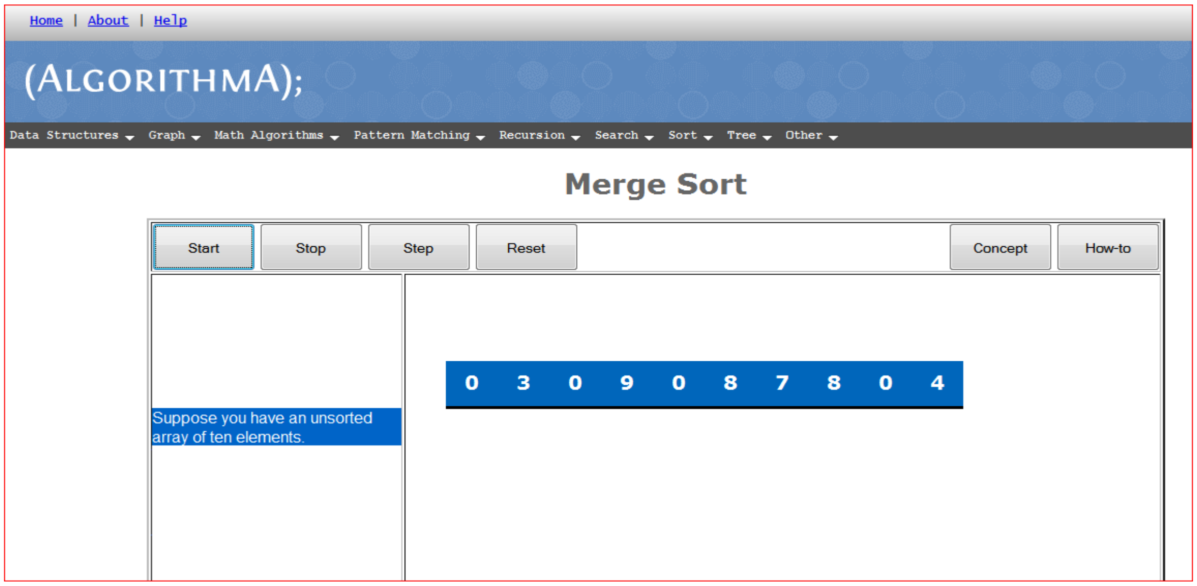
- Start – Starts the bubble sort animation.
- Forward – Allows the user to step through the sort one-step at a time.
- Back – Steps through the pseudo code one line at a time in a backward direction
- Reset – Resets the bubble sort animation to the beginning.

## 3.2 Merge Sort

### 3.2.1 Overview

The Merge Sort algorithm is a sorting algorithm that is similar to Quick Sort. Merge Sort has a complexity of O(n log(n)) and is therefore an optimal sort. The Merge Sort is based on a "divide and conquer" strategy. First, the sequence of data to be sorted is divided into two approximately equal halves. Each half is recursively sorted and then the two halves are merged into a sorted sequence.

In AlgorithmA 2010, the animation and walkthrough views are merged into one. The animation view is the default. To initiate the animation, the user must click on the "Start" button. Once initiated, the animation will proceed until the sort is complete. To view the walkthrough, the user must press the "Step" button and then the "Forward" button to proceed through the walkthrough.



Suppose you have an unsorted array of ten elements.

Merge Sort Animation

### 3.2.2 Layout

The Merge Sort algorithm is already classified under Sort. When selected, the user will be given a short explanation of how the algorithm works. The explanation of the algorithm is taken from a web page at Western Kentucky University, http://linux.wku.edu/~lamonml/algor/sort/merge.html, and may not be expressed in a way that the expected user (first-year Computer Science majors) will understand. The explanation will be altered to better represent this audience.

### 3.2.3 Functionality

The Merge Sort interface will contain the following functions:

- Start – Starts either the animation or walkthrough depending on whether the Step button is checked or not. See the Step button functionality for more detail.

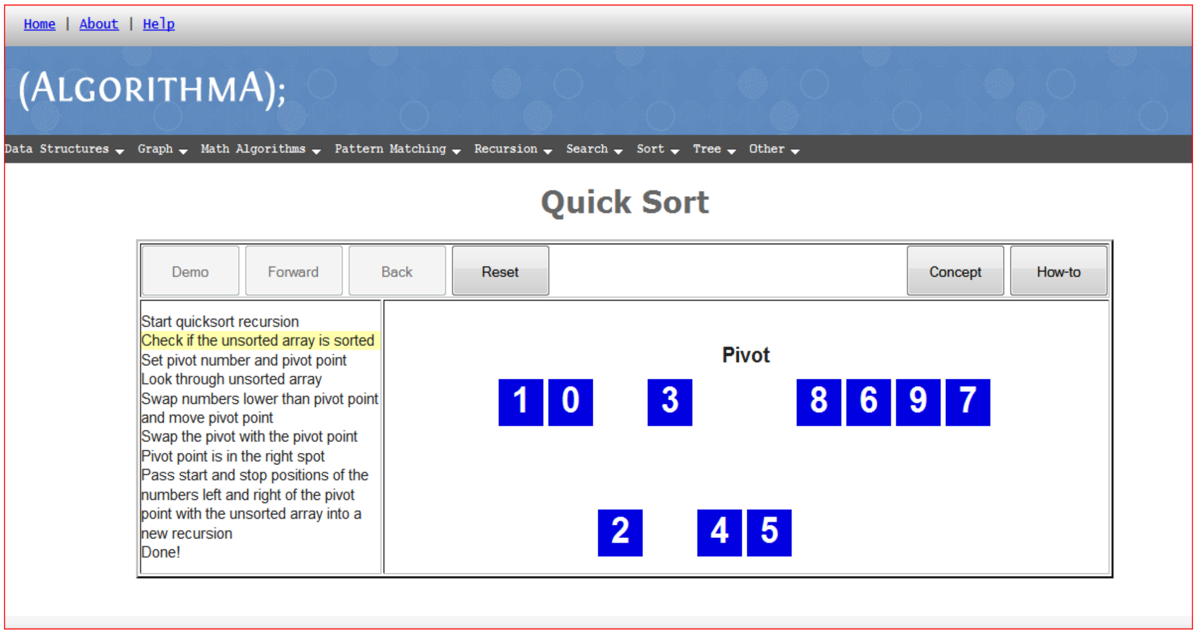
- Stop – Halts the execution of the animation if the current state of the animation is in execution mode.
- Step – When this is checked, the view is that of the walkthrough. When not checked, the view is that of the animation.
- Reset – Stops the animation or walkthrough and initializes the data again.

## 3.3 Quick Sort

### 3.3.1 Overview

The Quick Sort algorithm is a sorting algorithm whose complexity can range from $O(nlog_n)$ to $O(n^2)$. It first chooses a pivot element, and creates three lists. The first list is all elements that are less than the pivot value. The second list contains all elements that are equal to that value. The third list contains all elements that are greater than that value. The algorithm is recursively called onto the first and last list, and continues until there is only one element in each list. Doing so will give the desired result.

For AlgorithmA 2010, the animation and walkthrough components are merged into one. By default, the animation is set to automatic. Upon pressing the start button, the sorting algorithm will continue to sort without any interaction. If the user wants to see a walkthrough, the "step" button can be pressed, which will stop the next frame of animation until the user desires.



Start quicksort recursion
Check if the unsorted array is sorted
Set pivot number and pivot point
Look through unsorted array
Swap numbers lower than pivot point and move pivot point
Swap the pivot with the pivot point
Pivot point is in the right spot
Pass start and stop positions of the numbers left and right of the pivot point with the unsorted array into a new recursion
Done!

Quick Sort Animation

### 3.3.2 Layout

The Quick Sort algorithm is already classified under Sort. When selected, the user will be given a quick explanation of the algorithm. The explanation is insufficient from previous iterations and will be improved to provide more detail and a thorough explanation of how the algorithm works.

### 3.3.3 Functionality

The Quick Sort interface will contain the following functions:

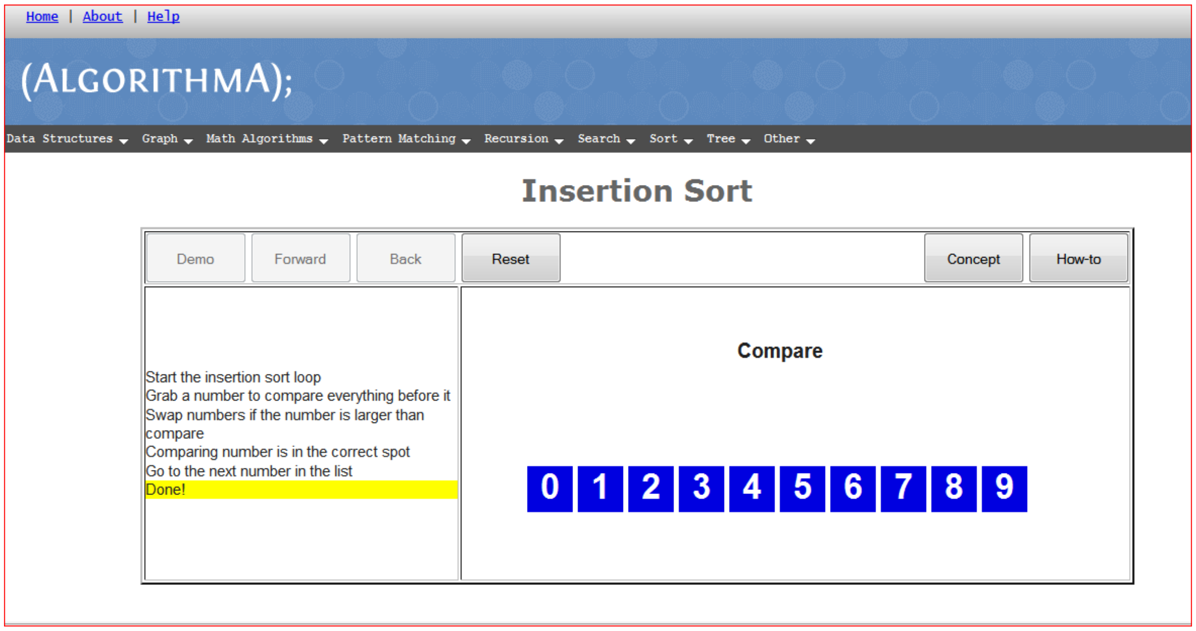
- Demo – Starts the Quick Sort animation
- Forward – Only activated during walkthrough mode. The next line of code will not be executed until this button has been pressed.
- Back – Steps through the pseudo code one line at a time in a backward direction
- Reset – Cancels the animation or walkthrough and reinitializes data seen on screen.

## 3.4 Insertion Sort

### 3.4.1 Overview

The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list. The insertion sort works by taking the values one by one and inserting each one into a new list that it constructs, constantly maintaining the condition that the elements of the new list are in the desired order with respect to one another. Clearly, this condition will not be maintained if each element is added to the new list at the beginning, instead, the insertion sort adds each element at a carefully selected position within the new list, placing the new element after each previously placed element that precedes it according to the given precedence rule, but before every such element that it precedes. The simplest implementation of this requires two list structures - the source list and the list into which sorted items are inserted. To save memory, most implementations use an in-place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceding item until it is in place.

Like the bubble sort, the insertion sort has a complexity of O (n 2). Although it has the same complexity, the insertion sort is a little over twice as efficient as the bubble sort.



Start the insertion sort loop
Grab a number to compare everything before it
Swap numbers if the number is larger than compare
Comparing number is in the correct spot
Go to the next number in the list
Done!

Insertion Sort Animation

### 3.4.2 Layout

Insertion sort will be classified under Sort. When Insertion sort is selected from sort the basic layout will be displayed. The animation will be displayed by default with an option to select viewable source code. Start, Reset, Pause, Forward, Step check box, Animation check box and Speed setting bar will be displayed.

**Functionality**

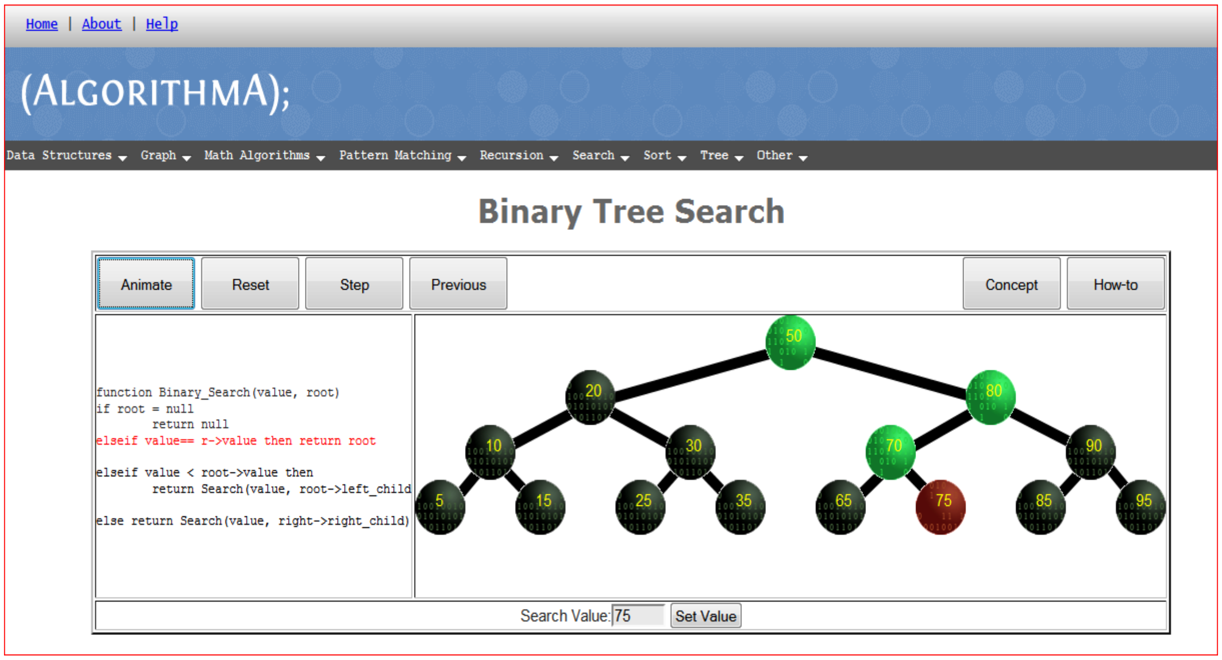
- Demo – Starts the Insertion sort animation.
- Forward – Only activated during walkthrough mode. The next line of code will not be executed until this button has been pressed.
- Back – Steps through the pseudo code one line at a time in a backward direction
- Reset – Cancels the animation or walkthrough and reinitializes data seen on screen.

## 4. Search algorithms

### 4.1 Binary Search Tree

### 4.1.1 Overview

A binary tree is a tree data structure in which each node has at most two children. Typically the first node is known as the parent and the child nodes are called left and right. Binary trees are commonly used to implement binary search trees and binary heaps. Every left node has a value less than or equal to its parent node and every right node has a value greater than or equal to its parent. A new node is always added as a leaf, following the specified rule above.



```
function Binary_Search(value, root)
if root = null
        return null
elseif value== r->value then return root

elseif value < root->value then
        return Search(value, root->left_child

else return Search(value, right->right_child)
```

Search Value: 75

Binary Search Tree Animation

### 4.1.2 Layout

The user will be presented with buttons detailing the concept and the how-to of the function and a basic tree to perform a search on. To the left of the animation work area will be a pseudo-code walkthrough that is highlighted as actions are performed.

### 4.1.3 Functionality

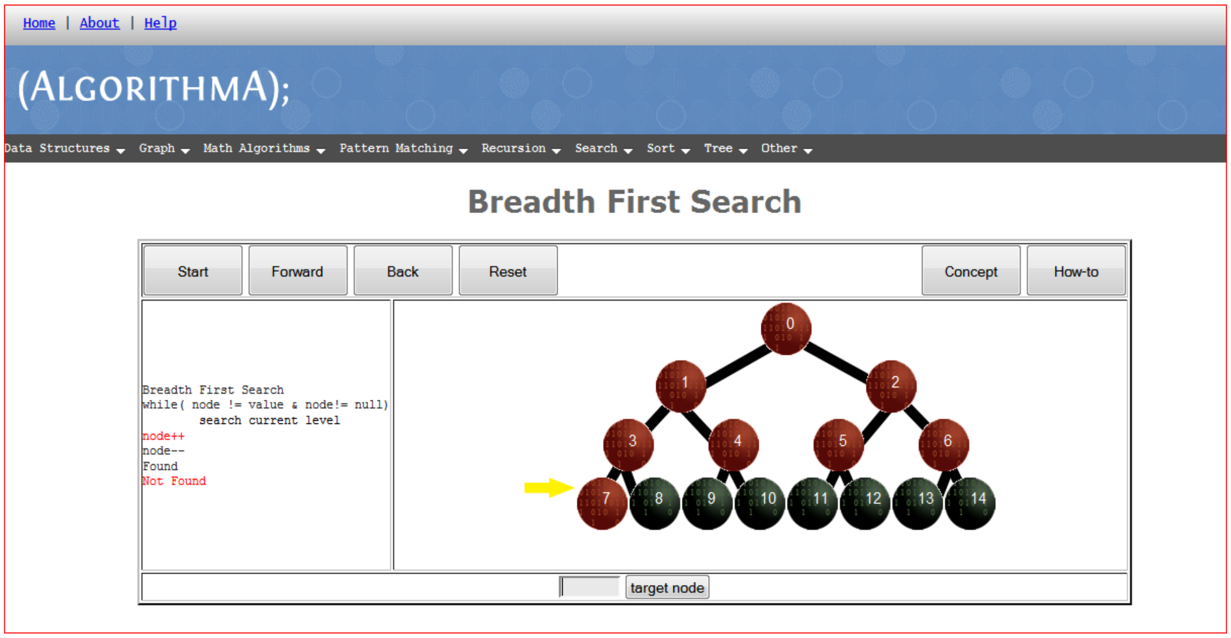
- Search Value/Set Value – User will enter a value to be set. This value will then be searched in the tree.
- Animate – Will perform the full search after a search value has been entered.
- Reset – Reset to the initial layout when first loaded.
- Step – Will allow user to run through animation one step at a time.
- Previous – Will allow user to back up through the animation one step at a time.

## 4.2 Search – Breadth First Search

### 4.2.1 Overview

Breadth-First Search (BFS) is a search that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal. BFS is an uninformed search method that aims to expand and examine all nodes of a graph or combination of sequences by systematically searching through every solution. In other words, it exhaustively searches the entire graph or sequence without considering the goal until it finds it. It does not use a heuristic algorithm.

From the standpoint of the algorithm, all child nodes obtained by expanding a node are added to a FIFO queue. In typical implementations, nodes that have not yet been examined for their neighbors are placed in some container (such as a queue or linked list) called "open" and then once examined are placed in the container "closed".



```
Breadth First Search
while( node != value & node!= null)
        search current level
node++
node--
Found
Not Found
```

Breadth First Search Animation

### 4.2.2 Layout

The user will be presented with buttons detailing the concept and the how-to of the function and a basic tree to perform a search on. To the left of the animation work area will be a pseudo-code walkthrough that is highlighted as actions are performed.

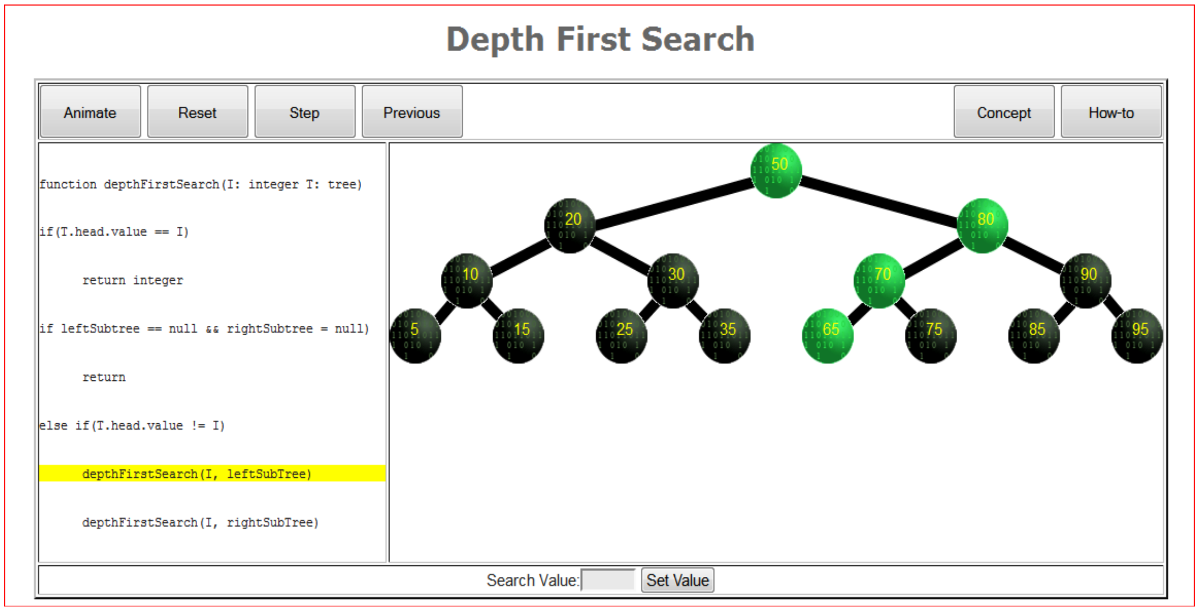
### 4.2.3 Functionality

- Target Node – User will enter a value to be searched for.
- Start – Will perform the full search after a search value has been entered.
- Forward – Will allow user to step through animation.
- Back – Will allow user to step through animation backward one step at a time.
- Reset – Reset to the initial layout when first loaded.

## 4.3 Search – Depth First Search

### 4.3.1 Overview

Depth-First Search (DFS) is an algorithm for traversing or searching a tree, tree structure, or graph. One starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking. DFS is an uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hasn't finished exploring. In a non-recursive implementation, all freshly expanded nodes are added to a stack for exploration.



```
function depthFirstSearch(I: integer T: tree)
if(T.head.value == I)
      return integer
if leftSubtree == null && rightSubtree = null)
      return
else if(T.head.value != I)
      depthFirstSearch(I, leftSubTree)
      depthFirstSearch(I, rightSubTree)
```

Depth First Search Animation

### 4.3.2 Layout

The user will be presented with buttons detailing the concept and the how-to of the function and a basic tree to perform a search on. To the left of the animation work area will be a pseudo-code walkthrough that is highlighted as actions are performed.

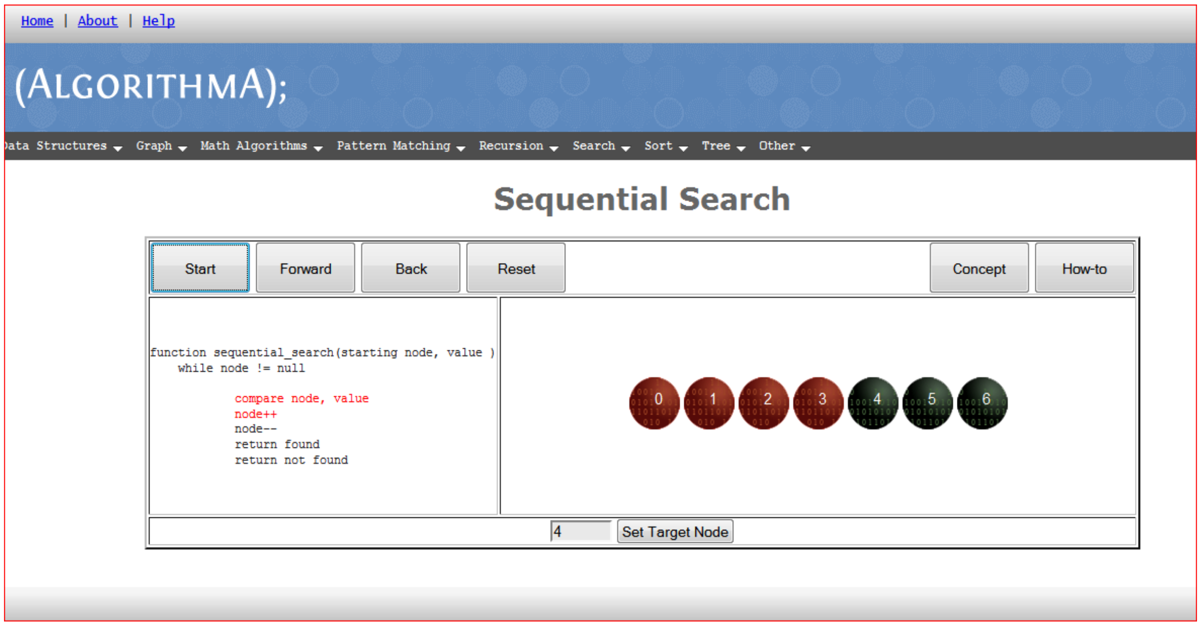
### 4.3.3 Functionality

- Search value – User will enter a value to be searched for.
- Animate – Will perform the full search after a search value has been entered.
- Step – Will allow user to pause through animation.
- Previous – Will allow user to step through animation backward one step at a time.
- Reset – Reset to the initial layout when first loaded.

## 4.4. Search – Sequential Search

### 4.4.1 Overview

Sequential search, also called linear search, is a method for finding a particular value in a list that consists in checking every one of its elements, one at a time and in sequence, until the desired one is found. Sequential search is the simplest search algorithm; it is a special case of brute-force search. Its worst case cost is proportional to the number of elements in the list; and so is its expected cost, if all list elements are equally likely to be searched for. Therefore, if the list has more than a few elements, other methods (such as binary search or hashing) may be much more efficient.



```
function sequential_search(starting node, value )
    while node != null

            compare node, value
            node++
            node--
            return found
            return not found
```

Sequential Search Animation

### 4.4.2 Layout

The user will be presented with buttons detailing the concept and the how-to of the function and the basic list to perform a search on. To the left of the animation work area will be a pseudo-code walkthrough that is highlighted as actions are performed.

### 4.4.3 Functionality

Sequential Search should only contain the bare minimum. This includes:

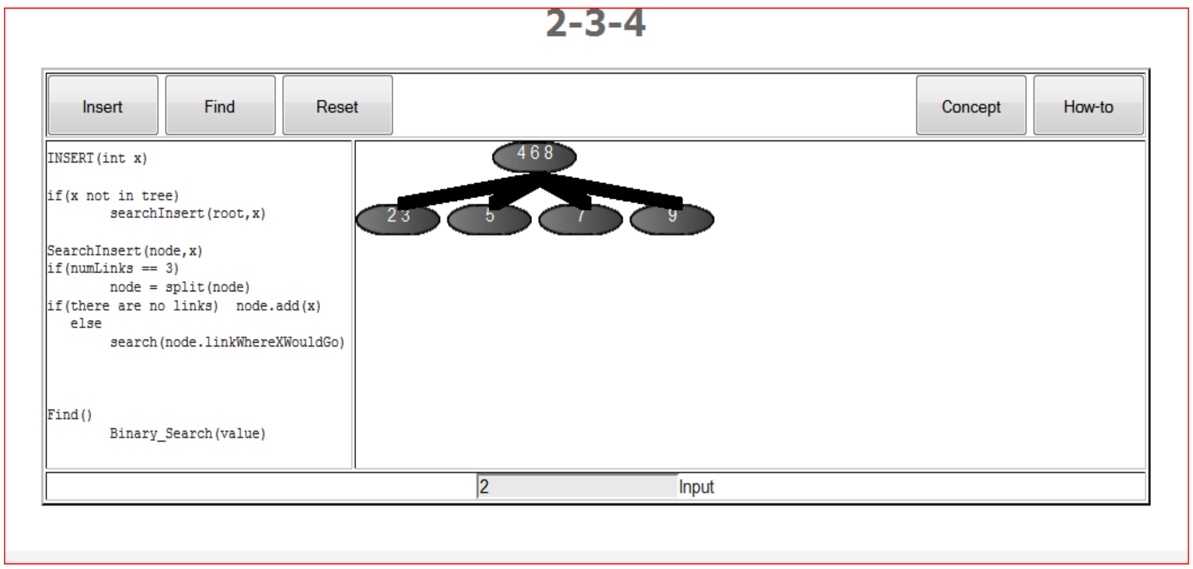
- Set Target Node – Will allow user to enter and set a value to be searched for.
- Start – Will perform the search either with or without a target value set.
- Forward – Will allow user to step through the animation one step at a time.
- Back – Will allow user to step backward through animation one step at a time.
- Reset – Reset to the initial layout when first loaded.

## 5. Trees

### 5.1 2-3-4 Tree

### 5.1.1 Overview

A 2-3-4 tree (also called a 2-4 tree) is a self-balancing data structure that is commonly used to implement dictionaries. 2-3-4 trees are B-trees of order 4, and can search, insert and delete in O(log *n*) time. One property of a 2-3-4 tree is that all external nodes are at the same depth. 2-3-4 trees can be difficult to implement in most programming languages because of the large number of special cases involved in operations on the tree.



```
INSERT(int x)

if(x not in tree)
        searchInsert(root,x)

SearchInsert(node,x)
if(numLinks == 3)
        node = split(node)
if(there are no links)  node.add(x)
   else
        search(node.linkWhereXWouldGo)



Find()
        Binary_Search(value)
```

### 5.1.2 Layout

The user will be presented with a brief introduction of the function and the basic already "useful" tree to perform a search on. To the left of the animation work area will be a pseudo-code walkthrough that is highlighted as actions are performed.

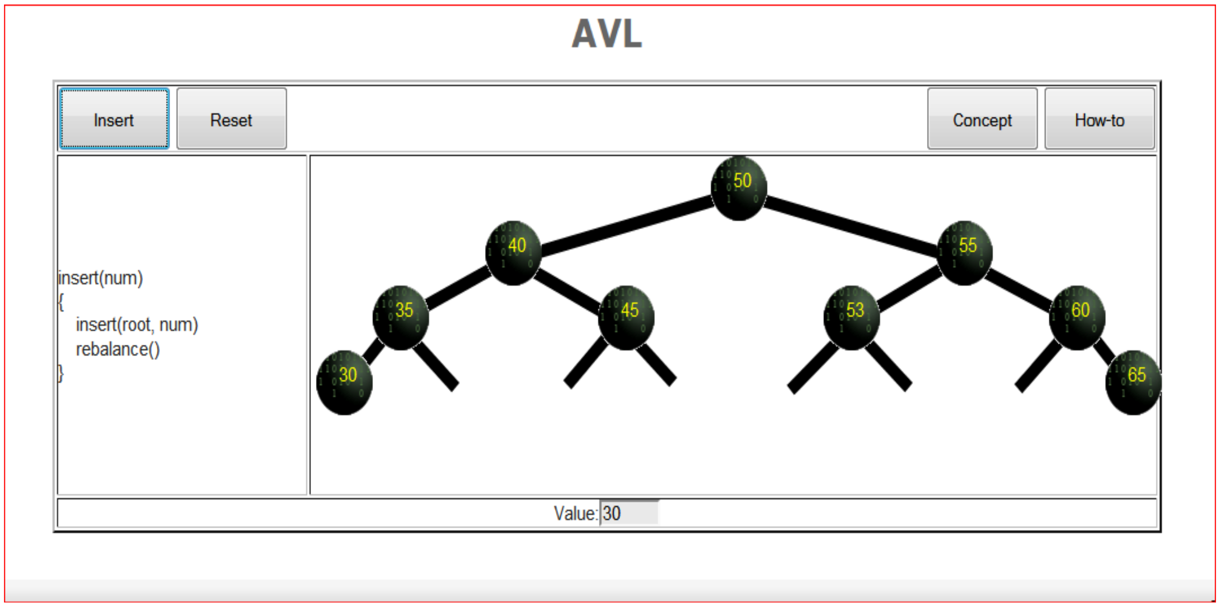
### 5.1.3 Functionality

- Input – Allows entering a whole numeric value.
- Insert – Will insert nodes with an input numeric value.
- Find – After a value in input, find will attempt to locate it.
- Reset – Reset to the initial layout when first loaded.

## 5.2 AVL Tree

### 5.2.1 Overview

An AVL tree is a self-balancing binary search tree. In an AVL tree, the heights of the two child subtrees of any node differ by at most one; therefore, it is also said to be height-balanced. Lookup, insertion, and deletion all take O(log *n*) time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations. The balance factor of a node is the height of its right subtree minus the height of its left subtree, and a node with balance factor 1, 0, or −1 is considered balanced. A node with any other balance factor is considered unbalanced and requires rebalancing the tree. The balance factor is either stored directly at each node or computed from the heights of the subtrees.



```
insert(num)
{
    insert(root, num)
    rebalance()
}
```

### 5.2.2 Layout

The user will be presented with a brief introduction of the function and the basic already "useful" tree to perform a search on. To the left of the animation work area will be a pseudo-code walkthrough that is highlighted as actions are performed.

### 5.2.3 Functionality

- Input – Allows entering a whole numeric value.
- Insert – Will insert nodes with an input numeric value.
- Reset – Reset to the initial layout when first loaded.

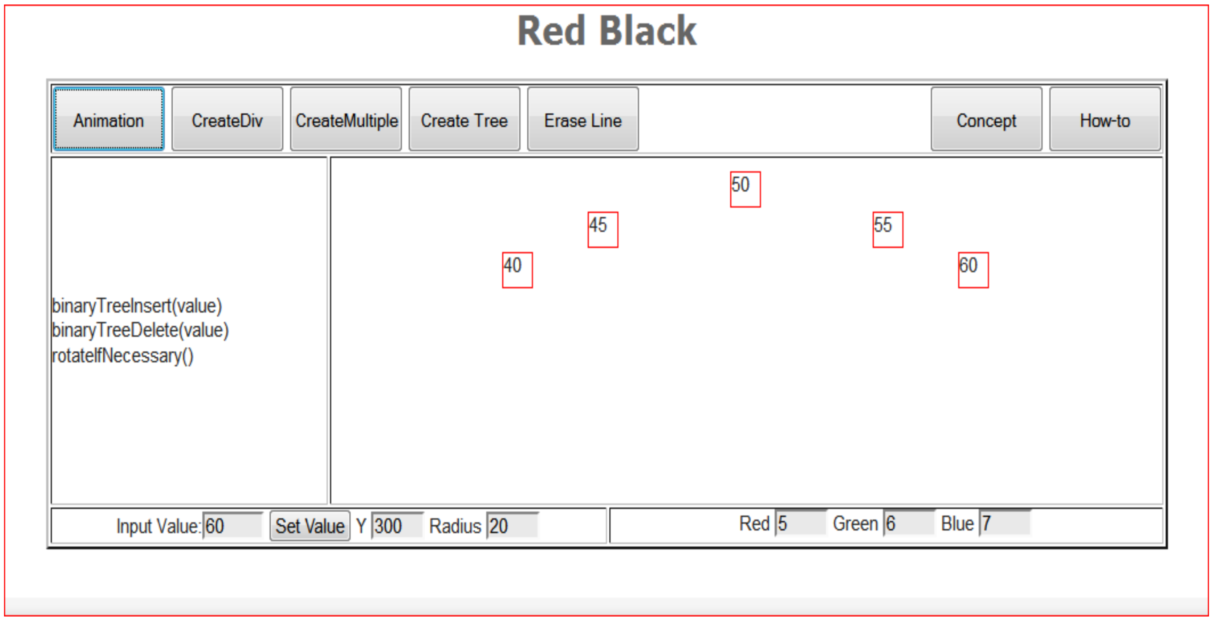
## 5.3 Red-Black Tree

### 5.3.1 Overview

A red-black tree is a type of self-balancing binary search tree, typically used to implement associative arrays. It is complex, but has good worst-case running time for its operations and is efficient in practice: it can search, insert, and delete in O(log *n*) time, where *n* is total number of elements in the tree. Red-black trees, like all binary search trees, allow efficient in-order traversal of their elements. The search-time results from the traversal from root to leaf, and therefore a balanced tree, having the least possible tree height, results in O(log *n*) search time. A red-black tree is a binary search tree where each node has a *color* attribute, the value of which is either *red* or *black*. In addition to the ordinary requirements imposed on binary search trees, the following additional requirements apply to red-black trees:

1. A node is either red or black.
2. The root is black. All leaves are black.
3. Both children of every red node are black.
4. Every simple path from a given node to any of its descendant leaves contains the same number of black nodes.

These constraints enforce a critical property of red-black trees: that the longest path from the root to any leaf is no more than twice as long as the shortest path from the root to any other leaf in that tree. The result is that the tree is roughly balanced. Since operations such as inserting, deleting, and finding values require worst-case time proportional to the height of the tree, this theoretical upper bound on the height allows red-black trees to be efficient in the worst-case, unlike ordinary binary search trees.



binaryTreeInsert(value)
binaryTreeDelete(value)
rotateIfNecessary()

Input Value: 60 Y 300 Radius 20 Red 5 Green 6 Blue 7

### 5.3.2 Layout

The user will be presented with a brief introduction of the function and the basic already "useful" tree to perform a search on. To the left of the animation work area will be a pseudo-code walkthrough that is highlighted as actions are performed.

### 5.3.3 Functionality

- Animation – A single button that will start the animation.
- CreateDiv – It allows to generate a division
- CreateMultiple – It allows to generate multiple divisions
- Create Tree– Reset to the initial layout when first loaded
- Erase Line - Reset to the initial layout when first loaded
- Input value – Allows to enter a whole numeric value. Set Value must be pressed after entering a number.

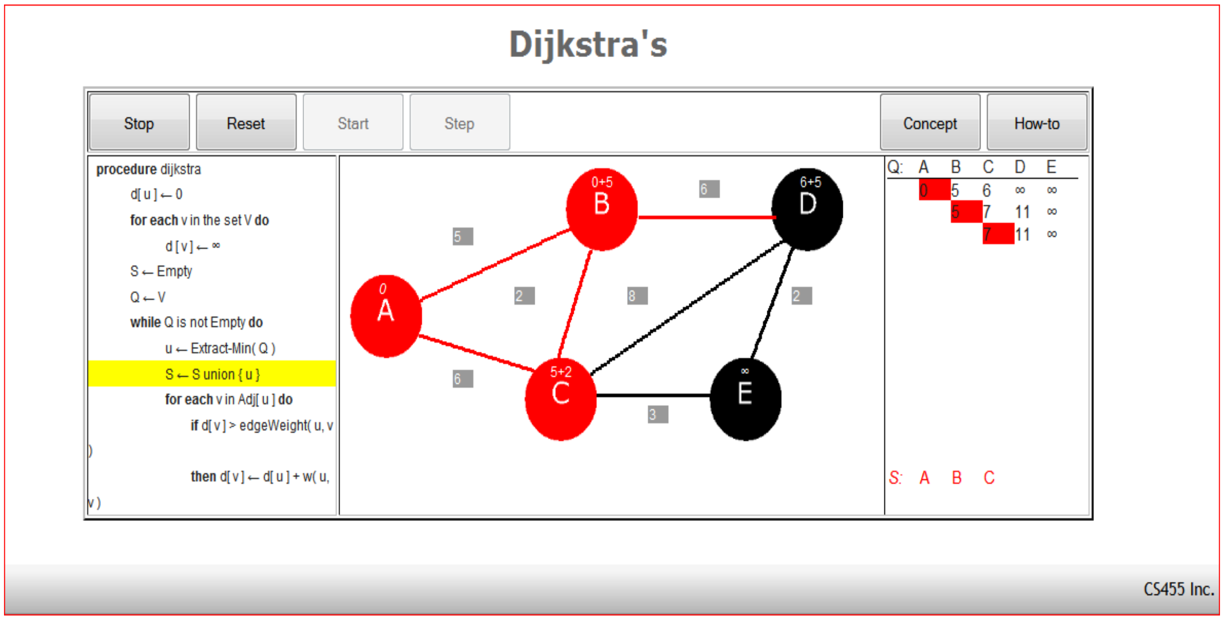
# 6. Graphs

## 6.1 Dijkstra's Shortest Path

### 6.1.1 Overview

Dijkstra's algorithm is a graph search algorithm that solves the single-source shortest path problem for a graph with nonnegative edge path costs, producing a shortest path tree. This algorithm is often used in routing.

For a given source vertex (node) in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined. For example, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. As a result, the shortest path first is widely used in network routing protocols, most notably IS-IS and OSPF (Open Shortest Path First).



```
procedure dijkstra
    d[ u ] ← 0
    for each v in the set V do
        d [ v ] ← ∞
    S ← Empty
    Q ← V
    while Q is not Empty do
        u ← Extract-Min( Q )
        S ← S union { u }
        for each v in Adj[ u ] do
            if d[ v ] > edgeWeight( u, v
)
            then d[ v ] ← d[ u ] + w( u,
v )
```

| Q: | A | B | C | D | E |
|---|---|---|---|---|---|
| | 0 | 5 | 6 | ∞ | ∞ |
| | | 5 | 7 | 11 | ∞ |
| | | | 7 | 11 | ∞ |

S: A B C

### 6.1.2 Layout

The user will be presented with a brief introduction of the function and the basic already "useful" graph to perform a search on. To the left of the animation work area will be a pseudo-code walkthrough that is highlighted as actions are performed.

### 6.1.3 Functionality

Dijkstra's Shortest Path should only contain the bare minimum. This includes:

- Pause – Allows the user to stop the animation and restart it when pressed again.
- Reset – Reset to the initial layout when first loaded.
- Start – Allows the user to run the animation.
- Step – Allows the user to step through the animation one step at a time.

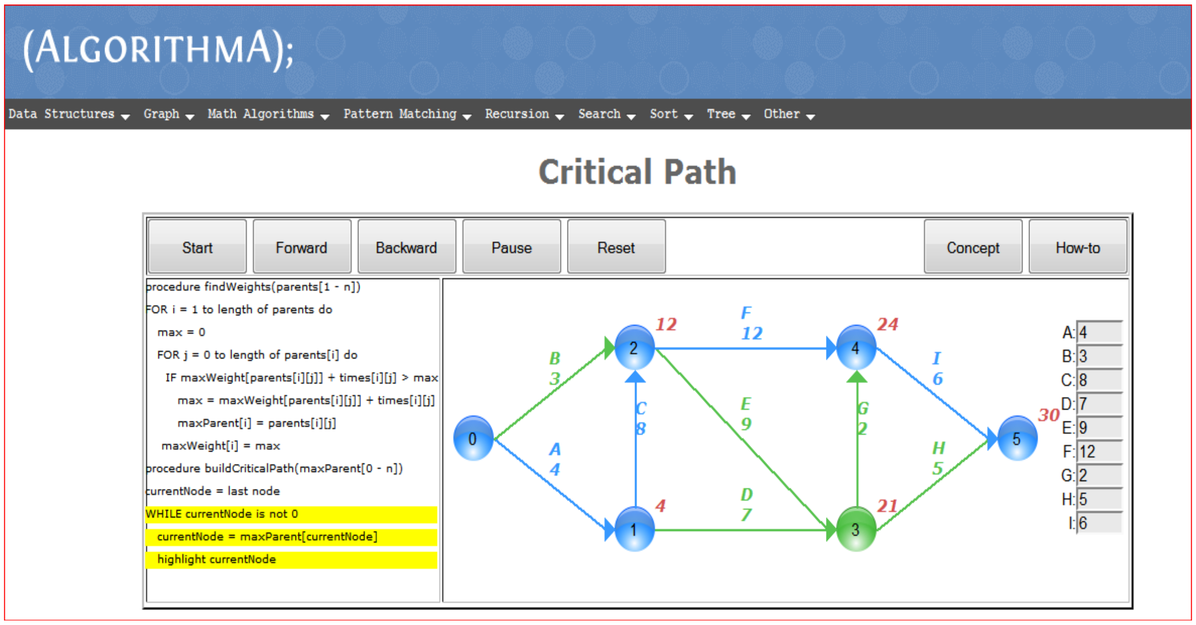
## 6.2 Critical Path

### 6.2.1 Overview

The critical path method (CPM) or critical path analysis is a mathematically based algorithm for scheduling a set of project activities. It is an important tool for effective project management. The essential technique for using critical path is to construct a model of the project that includes the following:

1. A list of all activities required to complete the project (typically categorized within a work breakdown structure),
2. The time (duration) that each activity will take to completion, and
3. The dependencies between the activities

Using these values, critical path calculates the longest path of planned activities to the end of the project, and the earliest and latest that each activity can start and finish without making the project longer. This process determines which activities are "critical" (i.e., on the longest path) and which have "total float" (i.e., can be delayed without making the project longer). This determines the shortest time possible to complete the project. A project can have several, parallel, near critical paths. An additional parallel path through the network with the total durations shorter than the critical path is called a sub-critical or non-critical path. A GANTT chart is uses the critical path algorithm to schedule how long a project should take.



```
procedure findWeights(parents[1 - n])
FOR i = 1 to length of parents do
  max = 0
  FOR j = 0 to length of parents[i] do
    IF maxWeight[parents[i][j]] + times[i][j] > max
      max = maxWeight[parents[i][j]] + times[i][j]
      maxParent[i] = parents[i][j]
   maxWeight[i] = max
procedure buildCriticalPath(maxParent[0 - n])
currentNode = last node
WHILE currentNode is not 0
  currentNode = maxParent[currentNode]
  highlight currentNode
```

| | |
|---|---|
| A: | 4 |
| B: | 3 |
| C: | 8 |
| D: | 7 |
| E: | 9 |
| F: | 12 |
| G: | 2 |
| H: | 5 |
| I: | 6 |

Critical Path Animation

### 6.2.2 Layout

The user will be presented with a brief introduction of the function and the basic already "useful" graph to perform a search on. To the left of the animation work area will be a pseudo-code walkthrough that is highlighted as actions are performed.

### 6.2.3 Functionality

- Start – Starts the bubble sort animation.
- Forward – Allows the user to step through the sort one step at a time.
- Backward – Steps through the pseudo code one line at a time in a backward direction
- Pause – Interactively allows the user to stop and continue the animation
- Reset – Resets the bubble sort animation to the beginning.

## 6.3 Kruskal's Minimum Tree

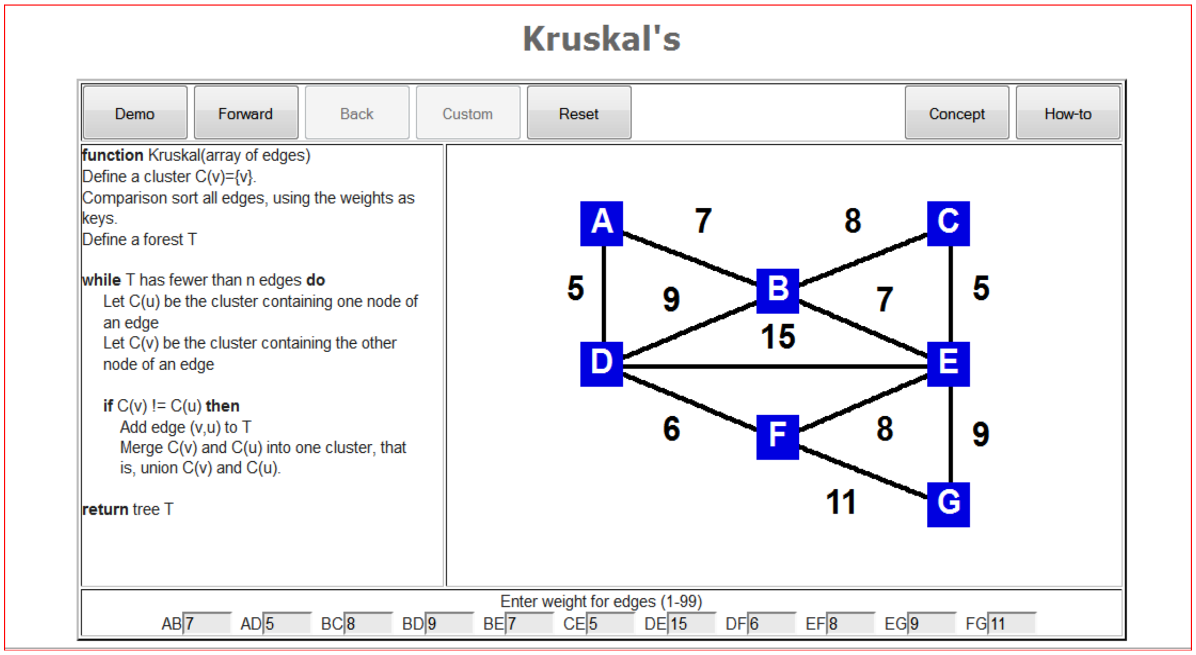
### 6.3.1 Overview

Kruskal's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a *minimum spanning forest* (a minimum spanning tree for each connected component). Kruskal's algorithm is an example of a greedy algorithm.

It works as follows:

- create a forest *F* (a set of trees), where each vertex in the graph is a separate tree
- create a set *S* containing all the edges in the graph
- while *S* is nonempty and F is not yet spanning
- remove an edge with minimum weight from *S*
- if that edge connects two different trees, then add it to the forest, combining two trees into a single tree
- Otherwise discard that edge.

At the termination of the algorithm, the forest has only one component and forms a minimum spanning tree of the graph.



```
function Kruskal(array of edges)
Define a cluster C(v)={v}.
Comparison sort all edges, using the weights as keys.
Define a forest T

while T has fewer than n edges do
    Let C(u) be the cluster containing one node of an edge
    Let C(v) be the cluster containing the other node of an edge

    if C(v) != C(u) then
        Add edge (v,u) to T
        Merge C(v) and C(u) into one cluster, that is, union C(v) and C(u).

return tree T
```

Enter weight for edges (1-99)

| AB | AD | BC | BD | BE | CE | DE | DF | EF | EG | FG |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 5 | 8 | 9 | 7 | 5 | 15 | 6 | 8 | 9 | 11 |

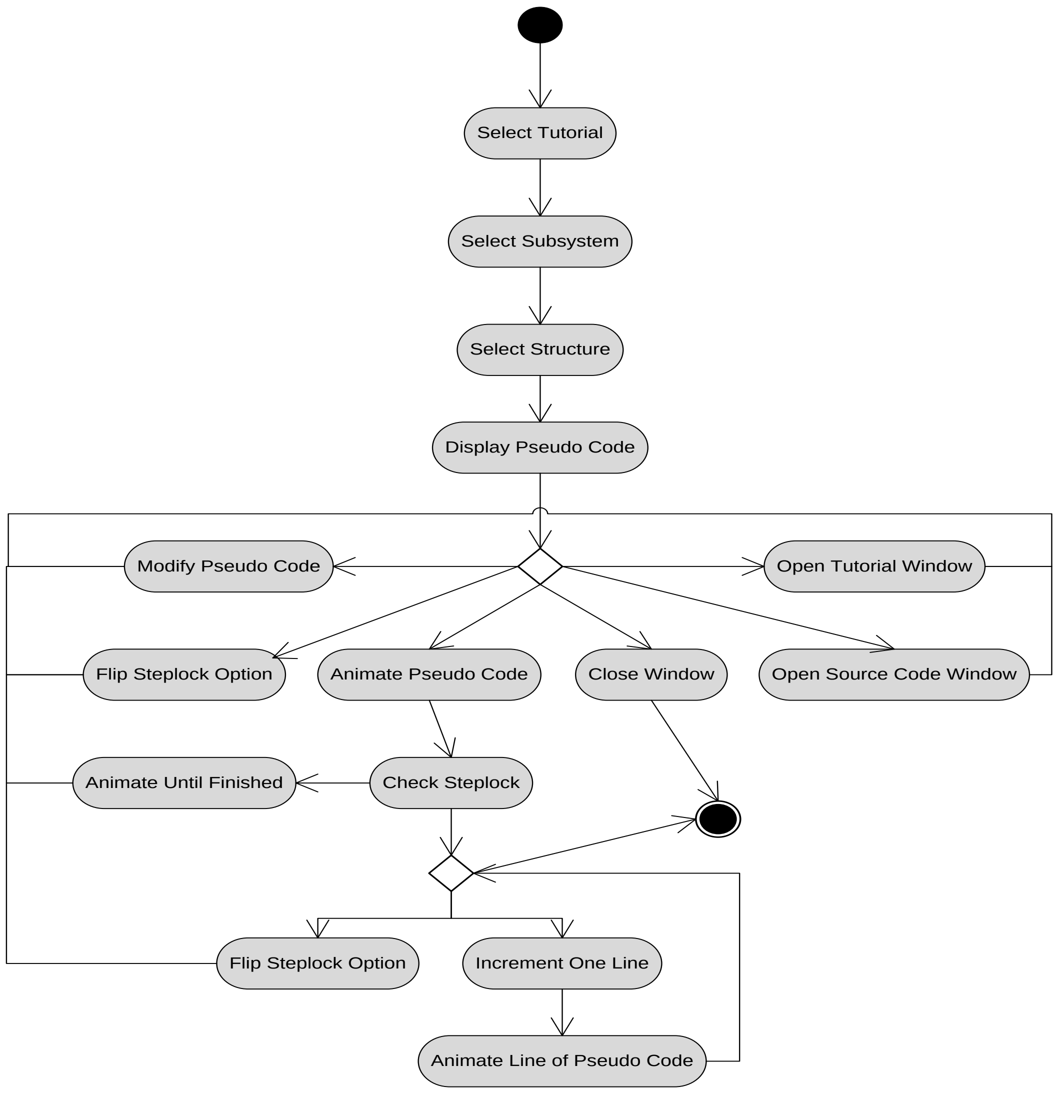
Kruskal's Animation

### 6.3.2 Layout

The user will be presented with a brief introduction of the function and the basic already "useful" graph to perform a search on. To the left of the animation work area will be a pseudo-code walkthrough that is highlighted as actions are performed.

### 6.3.3 Functionality

- Demo – Allows user to run a demo of the algorithm.
- Forward – Will allow user to step through the animation one step at a time.
- Back – Will allow user to step backward through animation one step at a time.
- Custom – Allows the user to create an animation of the algorithm from scratch.
- Reset – Reset to the initial layout when first loaded.



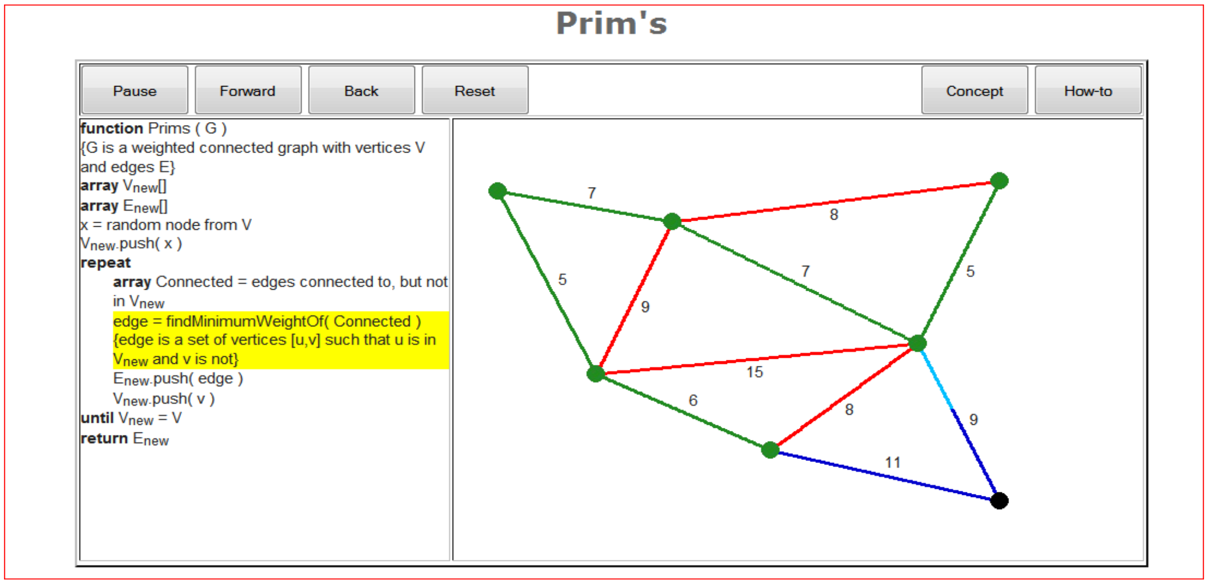
Kruskal's Algorithm
Activity Diagram

### 6.4 Prim's Minimum Tree

#### 6.4.1 Overview

Prim's algorithm closely resembles Dijkstra's algorithm because they both rely on a similar approach of finding the "next closest" vertex. Prim's algorithm slowly grows a minimum spanning tree, starting from a single vertex and adding in new edges that link the partial tree to a new vertex outside of the tree. Using the cut property, we know that if we have a partial tree, then to add a new vertex to the tree, all we actually need to do is find the vertex that is closest to the tree (the elements in the tree and the elements outside the tree are the two sets in the cut property, so we just choose the smallest edge bridging the two).

A significant difference between Prim's algorithm and Dijkstra's algorithm is that in Dijkstra's algorithm, the distance being checked is to a source node whereas in Prim's algorithm, the distance is only to the minimum spanning tree. The running time, too, is similar to Dijkstra's algorithm. When a heap is used to pick the next vertex to consider, the running time is O(|E| * log|V|), whereas when an unsorted list of vertices is searched in linear time, the running time is O(|V|^2 + |E|). V stands for number of vertices and E is the number of edges.



```
function Prims ( G )
{G is a weighted connected graph with vertices V
and edges E}
array Vnew[]
array Enew[]
x = random node from V
Vnew.push( x )
repeat
    array Connected = edges connected to, but not
    in Vnew
    edge = findMinimumWeightOf( Connected )
    {edge is a set of vertices [u,v] such that u is in
    Vnew and v is not}
    Enew.push( edge )
    Vnew.push( v )
until Vnew = V
return Enew
```

#### 6.4.2 Layout

The user will be presented with a brief introduction of the function and the basic already "useful" graph to perform a search on. To the left of the animation work area will be a pseudo-code walkthrough that is highlighted as actions are performed.

#### 6.4.3 Functionality

- Pause – Interactively allows the user to stop and continue the animation
- Forward – Will allow user to step through the animation one step at a time.
- Back – Will allow user to step backward through animation one step at a time.
- Reset – Reset to the initial layout when first loaded.

## 7. Glossary

**AlgorithmA**

PattE's sister project, which stands for "Algorithm Animation."

**Animation**

A visual display that depicts selected algorithms being studied. The display is based on the Cartesian x-y coordinate system.

**Computer Science**

Study of information and computation.

**CSE**

Acronym for Computer Science and Engineering.

**Deployment Diagram**

Deployment diagrams serve to model the hardware used in system implementations and the associations between those components.

**Fault**

An abnormal condition or defect at the component, equipment, or subsystem level, which may lead to failure. It is informally linked with "bug."

**Graphical User Interface**

A GUI is a method of interacting with a computer through a metaphor of direct manipulation of graphical images and widgets in addition to text.

**HTML**

Acronym for Hypertext Markup Language, the authoring language used to create documents on the World Wide Web.

**HTTP**

Acronym for Hypertext Transfer Protocol. It is the underlying protocol used by the World Wide Web. HTTP defines how messages are formatted and transmitted, and what actions Web Servers and browsers should take in response to various commands.

**IDE**

Acronym for Integrated Development Environment. IDEs assist computer programmers in developing software.

**Interface**

The communication boundary between two entities such as software and its users.

**Iteration**

The repetition of a process.

**Java**

A high-level programming language developed by Sun Microsystems.

**Model-View-Controller**

A software architecture that separates an application's data model, user interface, and control logic into three distinct components so that modification to one component can be made with minimal impact to the others.

**Mozilla Firefox**

A free, cross-platform, graphical web browser that complies with many of today's standards on the World Wide Web. The most important standards for *AlgorithmA 2010* will be the W3C web standards.

**Open Source Software**

Software whose source code is published and made available to the public, enabling anyone to copy, modify and redistribute the source code without paying royalties or fees. Open source code evolves through community cooperation.

**PHP**

Self-referential acronym for PHP: Hypertext Preprocessor, an open source, server-side, HTML embedded scripting language used to create dynamic Web pages. In an HTML document, PHP script (similar syntax to that of Perl or C) is enclosed within special PHP tags.

**Software Requirements Specification**

An SRS is used to describe all the tasks that go into the instigation, scoping, and definition of a new or altered computer system.

**SRS**

An acronym for Software Requirements Specification.

**W3C**

An acronym for the World Wide Web Consortium.

**World Wide Web Consortium**

An international organization that works to define standards for the World Wide Web.

## References

Wikipedia, http://en.wikipedia.org/

Bruegge, Bernd, and Allen H. Dutoit. Object-Oriented Software Engineering. 3rd Ed. New Jersey: Pearson Prentice Hall, 2010.

W3schools, http://www.w3schools.com/jsref/default.asp

"AlgorithmA 2008 SRS Prototype 2." CS455 Inc. Management Team. February 4, 2008.

Fowler, Martin. UML Distilled Third Edition. A Brief Guide to the Standard Object Modeling Language. Boston: Pearson Education, Inc. 2009.

IEEE SRS Std. 830-1998.

http://ieeexplore.ieee.org/Xplore/login.jsp?url=http%3A%2F%2Fieeexplore.ieee.org%2Fiel4%2F5841%2F15571%2F00720574.pdf&authDecision=-203