3. SYSTEM DESIGN AND ANALYSIS

3.1 STACK DESIGN:

Stack is a collection of items accessed as last in first out order. It is hence referred to as last in first out (LIFO) list. It can be implemented using either an array or a linked list. A stack is a list of elements in which only the last element is accessible. i.e., the operations like storing or retrieval, which is common to all data structures, are performed on the element added last to the list.

The primitive operations on a stack include:

- Push(S, x) - to insert an element x into the stack S
- Pop(S) - to delete an element from stack S
- Top(S) - to return the element at the top of the stack

The proposed system demonstrates the algorithms of the primitive stack operations like push and pop. All these are implemented graphically using Java.

ALGORITHM ANALYSIS

Push (S, element)

Push algorithm should do the following steps:

1. Get space for the element(this depends on the implementation)

2. If step 1 cannot allocate space (due to non availability)
   a. then
      return error
   b. else
      i. store element in the space allocated in step1
      ii. update Top so that it points to the currently added element
      iii. return
3. end

Pop (S)

Following steps are performed in Pop function:

1. Check whether there is atleast an element that can be popped out (i.e., Check for nonempty stack)
2. If the stack is not empty
   then
   a. copy the value stored in a temporary variable to return at the end
   b. bring down Top to point to the next element.
3. Return.

Pop function should first check for the non empty condition. In case if an attempt is made to pop an element from the empty stack, an error must be indicated. This condition check is referred to as ‘Underflow Condition’.

3.2. QUEUE DESIGN:

Queue refers to a collection of elements which are accessed in a first-in-first-out (FIFO) order. Elements are added to one end (rear end) and deleted from the other end (front) of the queue. The basic operations on a queue include insertion and deletion of elements. The insertion operation puts the element to the end of the queue and the deletion operation removes the element at the front of the queue. There are various forms of queue such as circular queue, dequeue, and priority queue, based on how the elements are accessed.

There are two forms of implementation for queues, similar to stacks. One is array-based implementation and the other is linked list-based implementation. Since both ends of the queue are used (insertion at rear end and deletion at front end), any implementation should keep track of both ends of the queue.

Insertion

Insertion requires incrementing the rear variable and storing the value.

Algorithm to insert an element into a queue

Insert (Q, front, rear, item)

1. If rear>=SIZE
   then
   a. Write “QUEUE OVERFLOW”
   b. Exit
   else
   a. rear ← rear+1
   b. Q[rear] ← item;
   c. if front=0
   then
   front ← 1;
2. End

This algorithm takes as input the queue, its front and rear pointer indices and the item to be placed. As in push operation of the stack, we need to check for the overflow condition during insertion. This condition checks whether there is enough room for the item.

Deletion

The first element of the queue is at the index front. Hence deletion requires extracting the element and incrementing the front variable, so that it points to the next element of the queue. The condition 'Queue underflow' is to check whether the queue has at least one element to delete. If the delete algorithm deletes the only element of the queue, it should reinitialize front and rear pointers (to mark an empty queue).

Algorithm Delete (Q, front, rear)

1. If front=0
   then
   write "Queue underflow"
   Exit
2. item←Q[front]
3. If(front==rear)
   then
   front ←rear←0;
   else
   front =front+1
4. Return.

3.3. BINARY SEARCH TREE DESIGN:

A binary tree where the value at root is greater than those in its left subtree and less than those in its right subtree is called a Binary Search Tree (BST). Both right and left subtrees also have this property of BST. Normally BSTs do not have duplicate values. If duplicate values are stored, they are referred as BST with duplicates. Our proposed system supports BST with duplicates. An indexed BST is one where every node has an index associated with it. Index represents the number of elements in its left subtree. This could be used to identify its position in the sequence of elements in its subtree.

The searching in BST is O(h), where h is the height of the tree. If the tree is balanced in the form of complete binary trees, then searching will be of O(log n), where n is the number of nodes in the tree.

Inserting an element into a BST

Algorithm InsertBST(int elt, NODE *T)

```
1. If tree is empty
   then
      create a one-node tree and return
  else
      if elt is less than the key in root
      then
        insertBST( elt, T->lchild)
     else
       if elt is greater than the key in root
       then
         insertBST(elt,T->rchild)
2. return T
3. End
```

Searching an element in BST

Searching an element in BST is similar to insertion operation, but they only return the pointer to the node that contains the key value or if element is null, a NULL is returned.

Searching starts from root of the tree. If the search key value is less than that in root, then the possible position of the element is in the left subtree. If the search key value is greater than that in root, then it should be in the right subtree based on the property of binary search tree. This searching should continue till the node with the search key value or a null pointer is reached. In case a null pointer is reached, it is an indication of the absence of the node.

Algorithm SearchBST(int elt,NODE *Tree)

```
1. If Tree is null
```

then

return NULL

2. If elt is less than key in root

then

return SearchBST(elt, Tree-> lchild)

else

if elt is greater than the key in root

then

return SearchBST(elt, Tree-> rchild)

else

return Tree

3.End

Deleting an element in a BST

The node to be deleted can fall into any one of the following categories:

- Node may not have any children(i.e. it is a leaf node)
- Node may have only one child(either left /right child)
- Node nay have two children(both left and right)

The procedure to delete a node in each of these cases differs.

Algorithm BSTDelete(int elt, NODE *Tree)

1. If Tree is null

then

begin

print ”Element not found”

end

2. If elt is less than info(Tree)

then

locate element in left subtree and delete it

```
    else
          If elt is greater than info(Tree)
     then
          locate element in right subtree and delete it
           else
           begin
                If(both left and right child are not NULL)
      then
           begin
                Locate minimum element  in right subtree
                 Replace  elt  by this value
                 Delete min element in right subtree and move the
                 remaining  tree as its right child
            end
else
  if left subtree is Null
   then
        replace node by its rchild
   else
        if right subtree is Null
        then
          replace node by its left child
end
Return Tree
End
```

3.4 SORTING ALGORITHMS

A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms

(such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output.

3.4.1BUBBLE SORT

Bubble sort is a simple sorting algorithm. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. This algorithm's average and worst case performance is O(n^2), so it is rarely used to sort large, unordered, data sets. Bubble sort can be used to sort a small number of items (where its inefficiency is not a high penalty). Bubble sort may also be efficiently used on a list that is already sorted except for a very small number of elements. For example, if only one element is not in order, bubble sort will take only *2n* time. If two elements are not in order, bubble sort will take only at most *3n* time.

Algorithm Bubblesort(data[ ],n)

1. For pass=1 to n-1
   a. For i=0 to(n-pass)
      If data[i]>data[i+1]
         Then[Interchange elements]
      i. Data[i]←→data[i+1]

2.End

3.4.2 SELECTION SORT

Table 3

Selection sort is an in-place comparison sort. It has O(n^2) complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations.

The algorithm finds the minimum value, swaps it with the value in the first position, and repeats these steps for the remainder of the list. It does no more than *n* swaps, and thus is useful where swapping is very expensive.

Algorithm Selection Sort(data[],n)

1. For pass=n-1 down to 1
   a. Select largest among data[0]…data[pass]
   b. Interchange largest and data[pass]
2. End

3.4.3 INSERTION SORT

Table 4

Insertion sort is a simple sorting algorithm that is relatively efficient for small lists and mostly sorted lists, and often is used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one. However insertion sort provides several advantages:

- Simple implementation
- Efficient for (quite) small data sets
- Adaptive (i.e., efficient) for data sets that are already substantially sorted: the time complexity is O(n + d), where d is the number of inversions
- More efficient in practice than most other simple quadratic (i.e., O(n^2)) algorithms such as selection sort or bubble sort; the best case (nearly sorted input) is O(n)
- Stable; i.e., does not change the relative order of elements with equal keys
- In-place; i.e., only requires a constant amount O(1) of additional memory space
- Online; i.e., can sort a list as it receives it

Algorithm InsertionSort(data[ ],n)

1. For pass=1 to n-1
   a . index=pass
   b. place data[index]in correct position among data[0]..data[pass]
2. End

4. SYSTEM REQUIREMENTS

4.1 HARDWARE REQUIREMENTS

Processor	:Intel Pentium 4 or higher
Hard disk	:20 GB or higher
RAM	:512 MB or higher
Input and Output	:Standard

4.2 SOFTWARE REQUIREMENTS

Operating System	:Microsoft Windows XP or higher, Linux
Software	:JDK
IDE	:Netbeans 6.9.1

5. IMPLEMENTATION DETAILS

To begin the interactive data structure visualizations, select one of the visualizations from the menu at the top of the screen. There are several groups of visualizations including stack, queue, binary search tree, and sorting algorithms. Once you have selected a topic the panel related to that topic will appear in the window. If you wish to view the visualization, click the appropriate button at the top of the screen. This application also provides options to adjust the speed of animation, to skip and pause animation.

In this software package visualization of stack and queue using arrays and linked lists are available. In the case of implementation of stack using arrays and linked list, options are provided for pushing and popping elements

from stack. Similarly in the case of queue, options are provided to enqueue and dequeue elements from queue.

A binary search tree is one method for implementing a keyed table. The advantage to this approach is that it combines the advantage of an array--the ability to do a binary search with the advantage of a linked list--its dynamic size. The efficiency of all of the operations is O(log n), only if the tree is reasonably height-balanced. Each operation involves examining at most one branch of the tree. In a balanced tree, the height of the tree is logarithmically related to the number of nodes.

There are three fundamental operations, find, insert, and delete. The find operation examines the value of the current node. If it matches the value being searched for, it stops. It goes left if the value being searched for is less than the value in the current node, and right if it is greater. If it reaches a leaf

node, the value is not in the tree. As you watch the animation you will notice that the red circle traces the path of the search.

The insert operation first performs a find operation. As you watch the animation, notice that the red circle begins at the root and moves down the tree searching for the specified value. If it finds the value, insertion is performed at the right child of the node. If it does not find the value, it performs the insertion on the leaf node, where the search ended. Because of the bounded screen size, a node will not be added when the length of any branch reaches its maximum length.

Like the insert operation, the deletion also begins with a find operation. In this case, the operation fails if the value is not found. If found, there are three distinct cases: 1) the node to be deleted is a leaf node, 2) it has one child, 3) it has two children. The first case is the simplest. The node is simply removed from its branch of the tree. The second case is similar to deleting from a singly linked list. The node is removed and its parent is connected to its only child. Notice that in the animation the subtree is then moved up. This movement is not a part of the actual algorithm and is done to keep the tree looking uniform. The two child case is the most complicated. The node containing the value to be deleted is not removed. Instead, the value of the node containing its inorder successor is moved to that node and the former is deleted. Use the radio button to select the deletion value. In that way you will have control over what node is deleted.

Sorting algorithms are fundamental to computer science. Sorted data has the important advantage that it can then be efficiently searched, using a binary search. The quadratic sorting algorithms, those whose efficiency is $O(n^2)$, are important because, although they are not the most efficient algorithms, they are conceptually the simplest.

Three sorting algorithms are implemented in this segment. The swap operation is fundamental to both the bubble sort and the selection sort. In the bubble sort, pairs of adjacent elements are swapped, if they are out of order. After one sweep of the data from left to right, the largest element is in the right

most position. Subsequent sweeps move the remaining values to their proper position. In the selection sort, the smallest element is swapped with the element in the first position. Then, the smallest of the remaining elements is swapped with the second element and so on.

The fundamental operation of the insertion sort is a right rotation. In the insertion sort each element is examined in turn and inserted into its proper position among the already sorted elements. The insertion sort is one we commonly use, to sort checks or to assemble a hand of cards in a card game.

The primary goals of this project were two-fold. The first was to build a visual, graphical piece of software that beginning computer science students can play with in order to learn the data structures and algorithms. This software should be intuitive, non-obtrusive, and widely distributable and available. The second was to make this software an enabling teaching tool for instructors desirous of using a more visual, interactive aid in teaching common algorithms. This tool would not be a replacement, but a supplement to traditional teaching methods and text book material.

The objectives for this project were simple and included the following...

- Present a clean, simple interface to the user
- Maintain a high degree of platform independence
- Make the software visually appealing
- Have the software present real working algorithm code

From these objectives, the *Java* language was chosen for the task. It's object orientation, programming ease, readability, modularity, and platform independence made it the ideal language to code this visualization tool.

To meet the goals and objectives presented above, the following requirements were decided upon...

🕐 **Have visualization graphics occur in real time with motion and animation**

Animation provides for a more interesting display. Animation that is

appealing, and visually non disruptive provokes interest in the algorithm and its inner workings. Obviously this will motivate the students to play with the applet, retain their attention, facilitate learning the algorithm.

- **Make visualization action completely user controllable**

If the software is under the control of the student, or user, that user can learn the concepts presented in their own pace. This also allows the user to dissect the events that are occurring within the algorithm in as small or as large chunks that the user desires. It also provides instructors the ability to stop the action for further explanation and questioning regarding particular portions of the algorithm before continuing further.

The two requirements dictated two main functional units within the application: a visualization display; and a utility or interface bar. These two units had to be functional to fully support their respective role, and cohesive so as to integrate well with the other units in support of the applets fundamental goals.

Visualization display

The visualization displays role is to display the following in support of the first requirement...

- The state of the current data structure or the array to be sorted.
- The main events that occur to the data structure, mainly comparisons and swaps.

Displaying these elements provides all the visual information that pertains to the data structures and algorithm and grasping its inner workings. Providing too many visual cues can be possibly distracting. Consequently, these display elements provide the right amount of visual information to observe the algorithm's workings at an abstract level.

Since arrays are usually presented left to right, horizontally in text books, and most languages are read from left to right, it was decided that the display would present the visual elements horizontally from left to right. With this approach, the array is presented visually horizontally as a series of adjacent blocks.

Different panels were created to display the windows corresponding to each of the data structures and sorting algorithms. The panels for BST, queue, stack and sorting algorithms inherited functions from the base panel DSPanel. DSPanel included abstract functions which were overridden in the corresponding classes.

Utility bar

The utility bar's role is to provide the following interface functionality in support of the second requirement...

- The ability to stop the visualization at any point.
- The ability to resume the application
- The ability to control how little, or how much of the algorithm that runs on execution.

The first piece of functionality was easily served with a pause button. This button had to have the functionality to immediately stop algorithm execution as well as visual animation. The second piece of functionality is likewise served by a run or execute button. Execute is a slightly better terminology since this is an actual working program. The third piece of functionality could be handled by either a speed control mechanism, a set of choices for execution chunks or granularities. Having granularity choices was the better design decision because execution precision is superior to having to pause at appropriate moments within the algorithm's execution.

The algorithms were implemented in the View classes. DSViewBST included functions for animation of insert, delete and find operations. Similarly each data structure has corresponding View classes where algorithms are implemented.

6. TESTING

The main philosophy behind testing is to discover errors and improve existing system. We used following software engineering methods for testing our application.

6.1 CODE TESTING

Code testing means checking correctness of code fragment by examining its logic using test cases.

Following are some of the test cases that we have used:

Case 1: 34,56,21,67

Case 2: ; , ^ , 7 , $

Case 3: a, b, c, d

These three test cases were given as input to stack, queue and BST. Case 1 did not cause any trouble when given to stack, queue and BST. When case 2 was given as input to stack, it caused the application to die. This was rectified by changing data type to string. Case 3 also caused the same problem and was corrected by the above step. In the case of BST, decision to do lexicographic sorting solved problem of comparison to some extent.

6.2 UNIT TESTING

In Unit testing, we concentrated on individual modules. Each of the modules like panels, stack, queue, BST and sorting algorithms were individually tested and errors were corrected.

6.3 INTEGRATION TESTING

In integration testing we tested our application by combining the different modules. Initially compilation error regarding missing packages were shown which was corrected by giving correct class path names.

6.4 SYSTEM TESTING

In this phase we tested our application in different platforms, for example Windows and Linux. Our application was found to work correctly in both platforms.

7. CONCLUSION

In this project, we have presented a visualization tool designed to aid computer science students learn Data Structures and Algorithms. This tool lets students visualize the commonly used data structures. We believe this tool will be an effective supplement to traditional instruction. Because of the time limitation, only the most commonly used data structures are implemented in this version of the software package, which include arrays, stacks, queues, binary search tree and sorting algorithms like insertion sort, selection sort and bubble sort.

8. FUTURE SCOPE

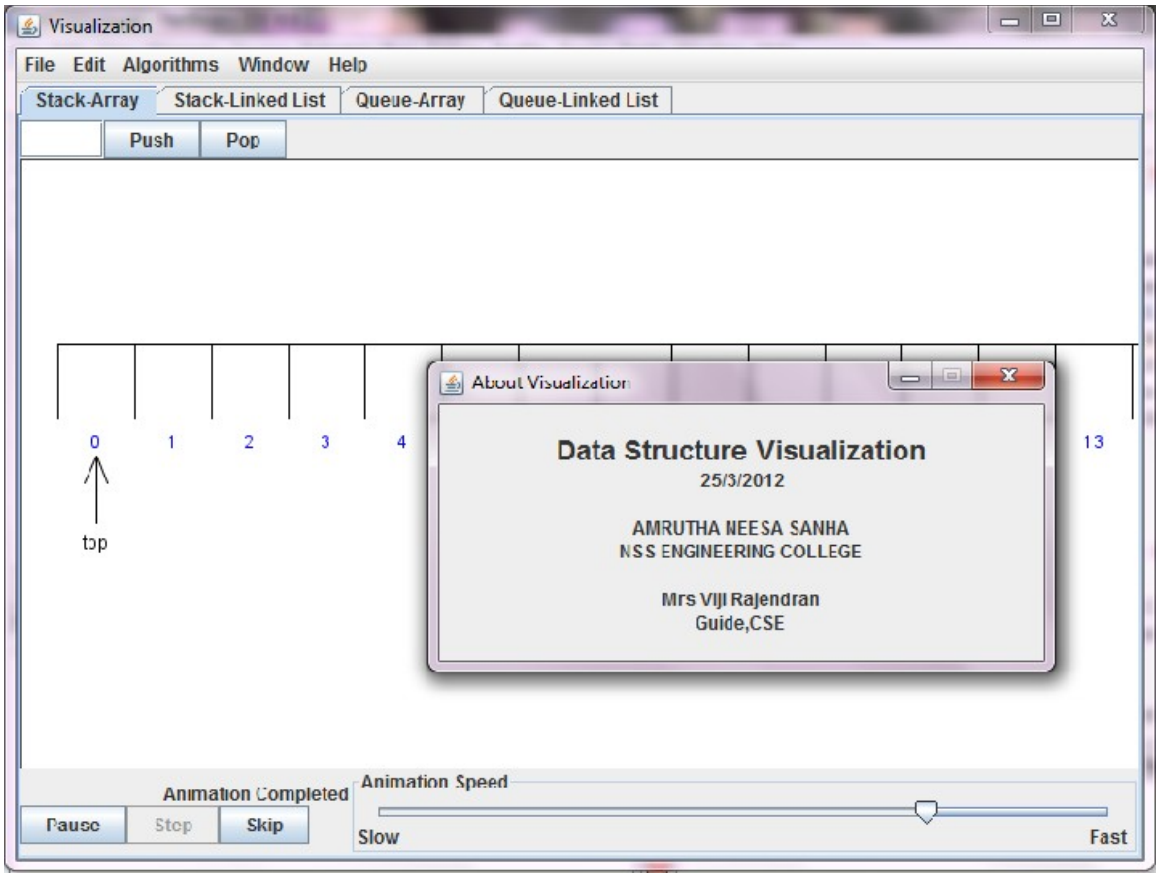
This application can be expanded by including more data structures and algorithms. This can also be included with visualization of user defined algorithms. There are two ways to add more observable data structures to this software such as directed graph, weighted graph, AVL tree, Red Black Tree, AA-tree, splay tree, hash table, etc. One way is to implement these data structures in the software. Another approach would be to develop and implement a mechanism for the software package to recognize the user-defined observable data structures, and leave the implementation to the user. This approach will allow users to use their own observable data structures, hence add more flexibility to the software. Another possible future enhancement for the software is to highlight the executing command line of the user-defined algorithm file. This would help the user to better follow the execution of the algorithm. This software package can be included with time and space complexities as well as algorithms.

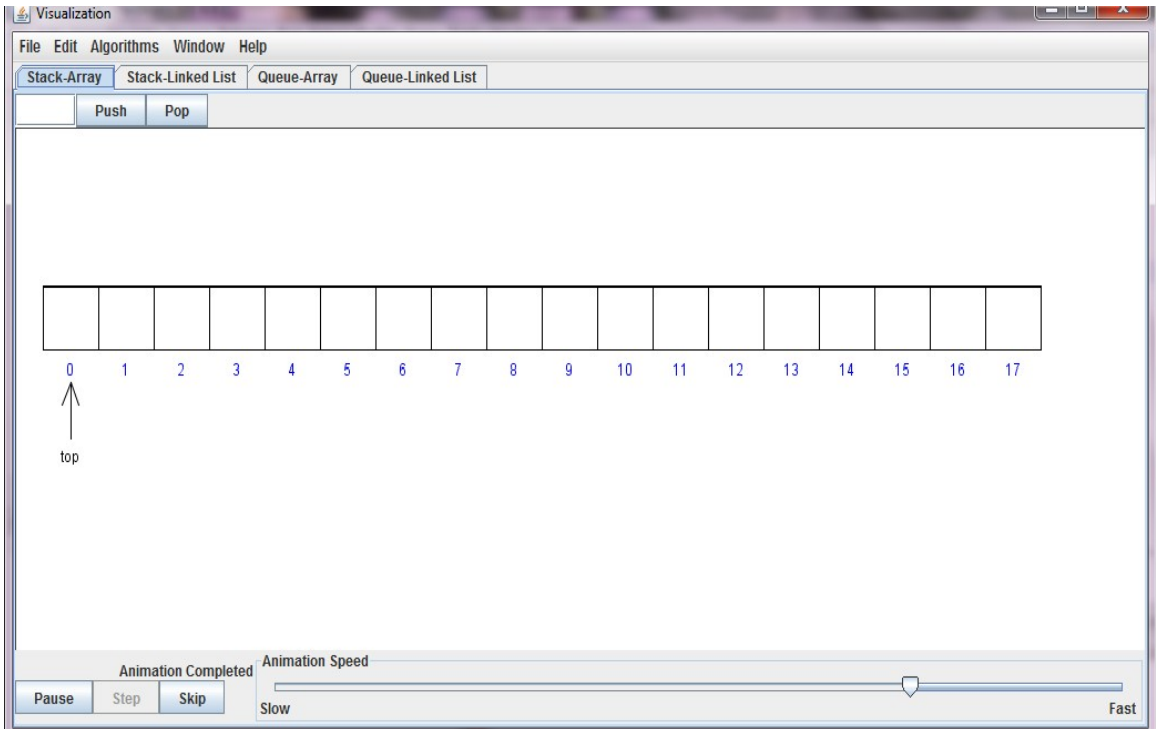
9. BIBLIOGRAPHY

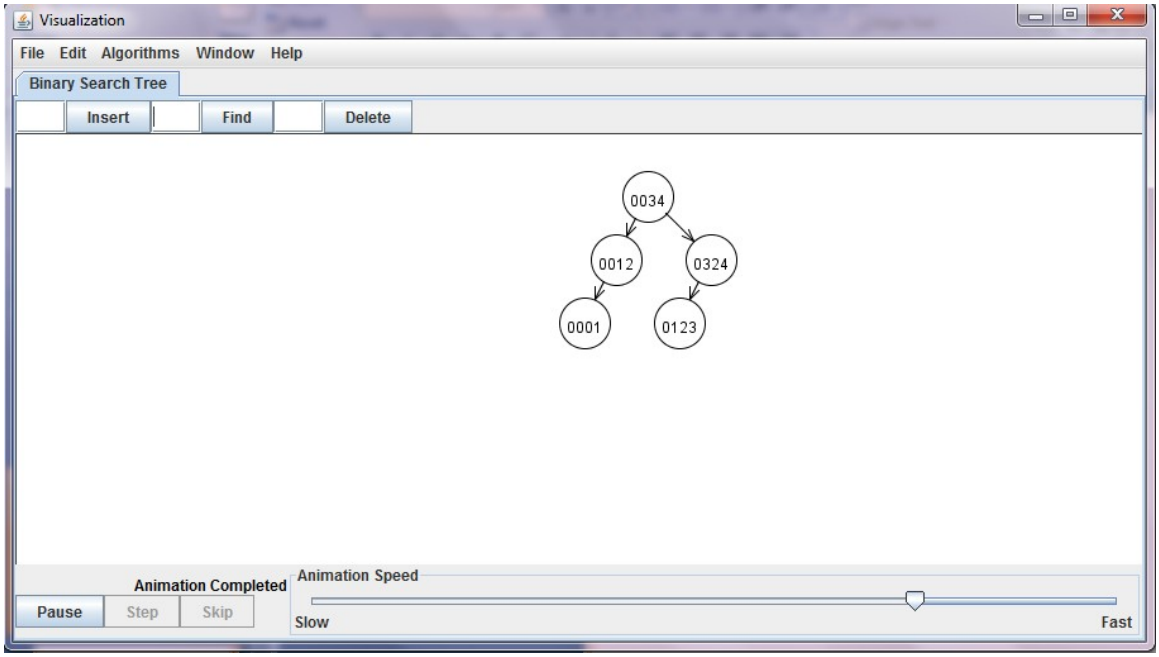
- http://www.student.seas.gwu.edu/~idsv/idsv.html
- http://wiki.algoviz.org/AlgovizWiki/LinearStructures
- A TOOL FOR DATA STRUCTURE VISUALIZATION AND USER-DEFINED ALGORITHM ANIMATION *Tao Chen', Tarek Sobh'* 3 1 ASEE/IEEE Frontiers in Education Conference
- Goodrich, Michael T. and Tamassia, Roberto, “Data Structures and Algorithms in Java”, httu://www.cs.brown.cdu/courses/cs9 16;booki’

Appendix

Visualization of Data Structures






Visualization
File Edit Algorithms Window Help
Binary Search Tree
Insert
Find
Delete
0034
0012
0324
0001
0123
Animation Completed
Pause
Step
Skip
Animation Speed
Slow
Fast

Visualization of Data Structures