BIG DATA W/ DATA STRUCTURES

Daniel Ordonez[1*]

*Corresponding author
[1]Address (Daniel Ordonez)
E-mails: daniel.ordonez210@gmail.com

ABSTRACT Data Structures is an import topic that is learned in computer science. These data structure are used to organize, process, and retrieve data. All of the data structures have pros and cons to the each of them. Either the pros and cons be how fast they can search for an item, how fast they can insert new data, and how fast they can delete data. As the amount of data increases, the amount of time to complete certain task also increases. The results obtained shows that certain data structures are faster than others.

Keywords: Data Structures, Search, Delete, Insert

1 Introduction

The purpose of this document is to outline the efficiency **OR** the speed of data structure functions and sorting algorithms. While many data structures and algorithms exist in the realm of computer science, this document will focus on a few selected data structures and sorting algorithms. To begin with, all of the data structure functions and sorting algorithms efficiency will be measured by the duration of each. In order to measure the time of when an event begins and ends accurately, the <**chrono**> STL library will be used to measure the time being passed.

All of the recorded data will be measured in **milliseconds (ms)**.

2 Hardware & Software Specifications

When running a program or software, every computer will behave differently when matched with the same program or software. Some computers will have dual core processors, a single core processors, or even a quad core processors. The number of core can significantly impact performance on a running process. The following will provide specifications of hardware being used to perform all programs:

CPU: Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz

RAM: 16.0 GB DDR4 @ 3600 MHz

OS: Windows 10 Pro

3 Project Outline

While many data structures exist in the world of computer science, this document will only focus on three data structures. This includes, **Stack, Binary Search Tree, & Vector**. Each of these data structures will use a template for the implementation and all related functions will be also implemented (e.g Vector will use array, Stack will use STL queue). Implementation will be also shown in every section. The data size that will be used will be sets of: **10k , 50k, & 100k** items. This document will analyze the speed and will compared to each other of the following functions:

- Insertion
- Deletion
- Searching
- Sorting

4 Analysis: Insertion

Listings 1, 2, and 3 shows the insertion implementation of StackQ<T>, VectorT<T>, & BinarySearchTree<T>. StackQ<T> is implemented using the STL queue for its implementation of Stack. VectorT<T> is implemented using STL arrays for the implementation of Vector. BinarySearchTree<T> is implemented using nodes for the implementation of Binary Search Tree.

Listing 1: push() Implementation w/ STL queue

```
template<typename T>
void Stack<T>::push(T data) {

    secondQ.push(data);

    while (!(firstQ.empty())) {
            secondQ.push(peek());
            firstQ.pop();
    }

    firstQ = secondQ;

    std::queue<T> emptyQ;

    secondQ = emptyQ;
}
```

Listing 2: insert() Implementation w/ array

```
template <typename T>
void VectorT<T>::insert(T data) {

        if (!(currIndex >= size)) {
            arr[currIndex] = data;
            currIndex++;
            return;
    }

    resizeArr();
    arr[currIndex] = data;
    currIndex++;
    return;
}
```

Listing 3: append() w/ Node

```
template <typename T>
BinarySearchTree<T>*
BinarySearchTree<T>::append(BinarySearchTree* root, T data) {

    if (root == NULL) {return new BinarySearchTree<T>(data);}

    (data < root->data) ? root->left = append(root->left, data)
    : root->right = append(root->right, data);

    return root;

}
```

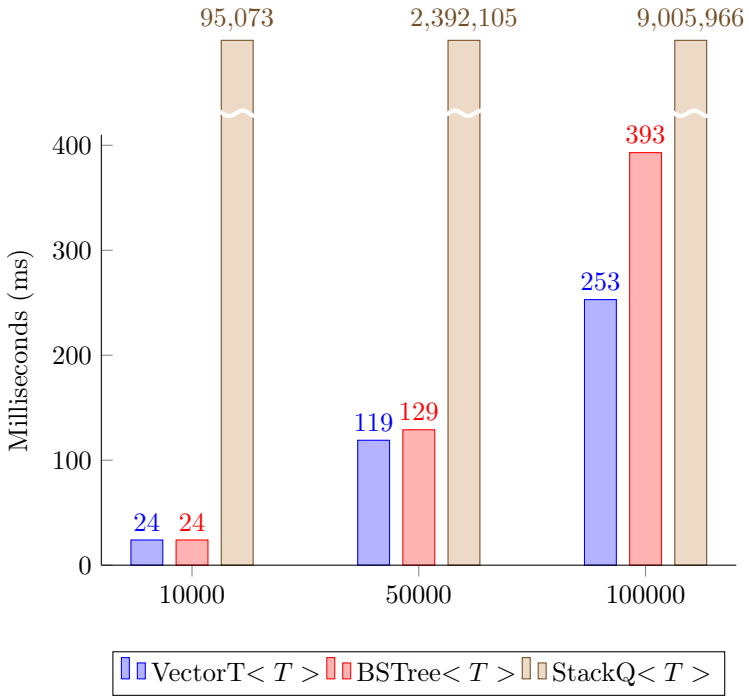

Figure 1: Running time of insert function of 10,000 , 50,000, & 100,000 items

Insertion Running Time			
Input Size	**StackQ<T>**	**BST<T>**	**VectorT<T>**
10,000	95,073 ms	24 ms	24 ms
50,000	2,392,105 ms	129 ms	119 ms
100,000	9,005,966 ms	393 ms	253 ms

4.1 Running Time Analysis

The above graph represents the insertion of 10,000 items, 50,000 items, & 100,000 items, which is measured in milliseconds (ms). As seen in the Figure 1, the running time of each the data structure vary. StackQ<T> has the longest running time of all three data structures. In the worst case, push() from StackQ<T> data structure must complete O(n) traversals. In the best case, push() is O(1).

BinarySearchTree<T> has one of shortest running time of the three data structures. In the worst case, append() from the BinarySearchTree<T> data structure must complete O(n) travesals, in the case where left/right skewed tree exist. In the best case, append() can perform at O(1) time complexity.

VectorT<T> has shortest running time of all data structures. In the worst case, insert() from the VectorT<T> runs at O(1), where adding an element is done at the end of the array. When VectorT<T> size reached maximum size, the array is doubled which is a O(n) operation of initializing a new array of (size * 2) size and inserting the old array values into the new initialized array with (size * 2) space. In the best case, insert() performs at O(1) time complexity.

5 Analysis: Deletion

Listings 4, 5, and 6 shows the deletion implementation of StackQ<T>, VectorT<T>, & BinarySearchTree<T>.As mentioned before in this document, the StackQ<T> is implemented with STL queue, VectorT<T> is implemented with array, BinarySearchTree<T> is implemented using Node. The time complexity of StackQ<T> and VectorT<T> are mostly constant time, O(1). BinarySearchTree<T> has a general time complexity of O(n), where it must traverse of n height.

Listing 4: qpop() Implementation w/ STL queue

```
template<typename T>
void Stack<T>::qpop() {

    if (firstQ.empty()) { std::cout << "Stack is empty" << "\n"; }
    firstQ.pop();

}
```

Listing 5: remove() Implementation w/ Array

```
template <typename T>
void VectorT<T>::remove() {

            if(currIndex == 0) {
                    return;
            }
            if (currIndex <= (size / 2) && (size != 10)) {

                    T* newArr = new T[(size / 2)];

                    for (auto i = 0; i <= currIndex - 1; i++) {
                            newArr[i] = arr[i];
                    }

                    size /= 2;
                    delete[] arr;
```

```
                arr = newArr;
                currIndex--;
                return;
            }


    currIndex--;
    arr[currIndex] = 0;
}
```

Listing 6: remove() w/ Node

```
template <typename T>
BinarySearchTree<T>*
BinarySearchTree<T>::remove(BinarySearchTree* root, T data) {

    if (root == NULL) { return root; }

    if (data > root->data) { root->right = remove(root->right, data); }

    else if (data < root->data) { root->left = remove(root->left, data); }

    else {

        if (root->left == NULL &&
                root->right == NULL) {
                return NULL;
        }


        if (root->left == NULL) {
                BinarySearchTree* tree = root->right;
                free(root);
                return tree;
        }

        if (root->right == NULL) {
                BinarySearchTree* tree = root->left;
                free(root);
                return tree;
        }

        BinarySearchTree* bst = minVal(root->right);

        root->data = bst->data;

        root->right = remove(root->right, bst->data);
```

```
        }
        return root;

    }
```

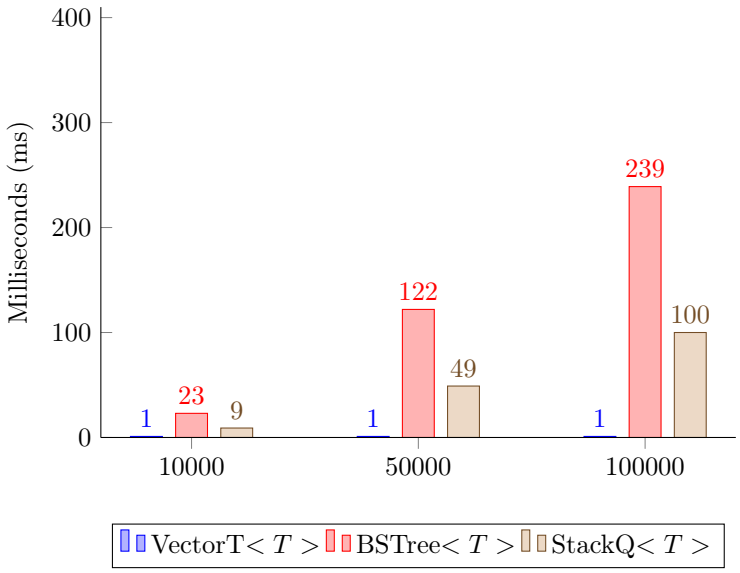

Figure 2: Running time of deletion function of 10,000 , 50,000, & 100,000 items

Deletion Running Time			
Input Size	**StackQ<T>**	**BST<T>**	**VectorT<T>**
10,000	9 ms	1̃ ms	23 ms
50,000	49 ms	1̃ ms	122 ms
100,000	100 ms	1̃ ms	239 ms

5.1 Running Time Analysis

The above graph represents the deletion of 10,000 items, 50,000 items, & 100,000 items, which is measured in milliseconds (ms). As seen in the Figure 2, the running time of each the data structure vary. With certain linear data structures, deletion is much easier to accomplish and more costly with non-linear data structures.

The VectorT<T> data structure deletion is of constant time complexity, O(1), where downsizing of the array is not required. When downsizing is required, the operation must make a new array with a size of (size / 2) and update the new array with all old values of the old array.This is return takes linear time O(n). Downsizing in the VectorT<T> is done in order to remove any extra space that may exist when deleting elements to further minimize memory usage.

StackQ<T> uses the STL queue for its implementation of the Stack data structure. With StackQ<T> using queue, deletion of the StackT<T> is constant time, O(1), where the

top of the stack is removed or popped. Before removing the top of the stack, the stack must be also checked if it is empty in order.

BinarySearchTree<T> deletion general time complexity is of O(n), where it must traverse height and search item to delete. After found, deletion of the node begins where the Binary Search Tree rules must not be broken.

6 Analysis: Searching

Listings 7, 8, and 9 shows the search implementation of StackQ<T>, VectorT<T>, & BinarySearchTree<T>. StackQ<T> is implemented using the STL queue for its implementation of Stack. VectorT<T> is implemented using STL arrays for the implementation of Vector. BinarySearchTree<T> is implemented using nodes for the implementation of Binary Search Tree.

Listing 7: searchStack() Implementation w/ STL queue

```
template<typename T>
int Stack<T>::searchStack(T data) {

    while (!(firstQ.empty())) {

            if (peek() == data) {
                    std::cout << "Found" << "\n";
                    return 1;
            }

            firstQ.pop();
    }

    return -1;
}
```

Listing 8: vectorSearch() Implementation w/ Array

```
template <typename T>
int VectorT<T>::vectorSearch(T data) {

    for (auto i = 0; i < currIndex; i++) {
            if (arr[i] == data) {
                    return 1;
            }
    }
    return -1;
```

```
}
```

Listing 9: search() w/ Node

```
template <typename T>
BinarySearchTree<T>*
BinarySearchTree<T>::search(BinarySearchTree* root, T target) {

    if (root == NULL || root->data == target) { return root; }

    if(root->data > target) { return search(root->left, target);}

    return search(root->right, target);
}
```

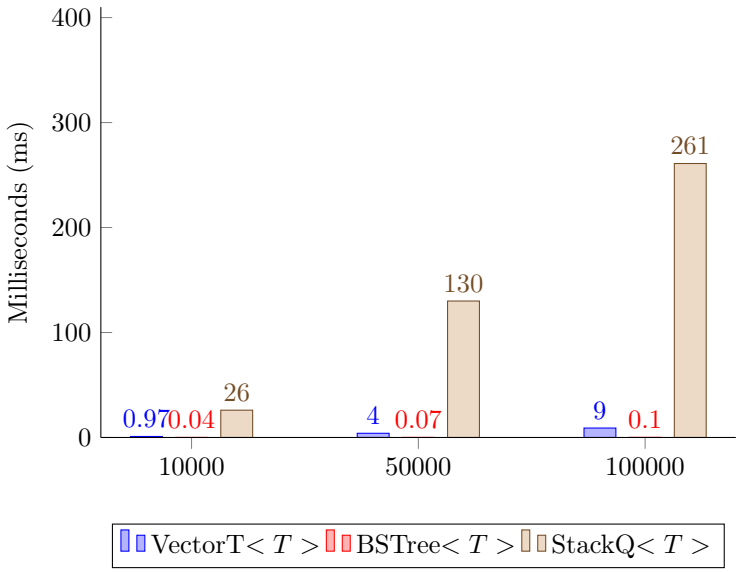

Figure 3: (Worst Case) Running time of search function of 10,000 , 50,000, & 100,000 string items

(Worst Case) Searching Running Time			
Input Size	**StackQ<T>**	**BST<T>**	**VectorT<T>**
10,000	26 ms	0.9681 ms	0.0417 ms
50,000	130 ms	4 ms	0.0665 ms
100,000	261 ms	9 ms	0.0974 ms

6.1 (Worst Case) Running Time Analysis

The above graph represents the searching of 10,000 items, 50,000 items, & 100,000 items, which is measured in milliseconds (ms). As seen in the Figure 3, the running time of each the data structure vary.

In the scenario of searching a specified item, the worst case exist when the specified item is the last element of the data structure, in which the the user must traverse through n elements to find the correct value.

VectorT<T> implements a linear search algorithm where its time complexity is of O(n), where it must traverse N elements to find the target value. For the worst case, the value is located at the end of the array and must traverse through all array elements to find the target value.

StackQ<T> uses the same approach of the VectorT<T> where it must compare the target value to the top of the stack and remove if the target value does not match the value. The time complexity of this algorithm is of O(n). In the worst case the target value is located at the bottom of the stack.

BinarySearchTree<T> implementation of search is among the more efficient searching algorithms, where is is of O(log n) in the best case. In the worst case, a left/right skewed tree exist where a long chain of consecutive N elements must be traversed to search the target value.

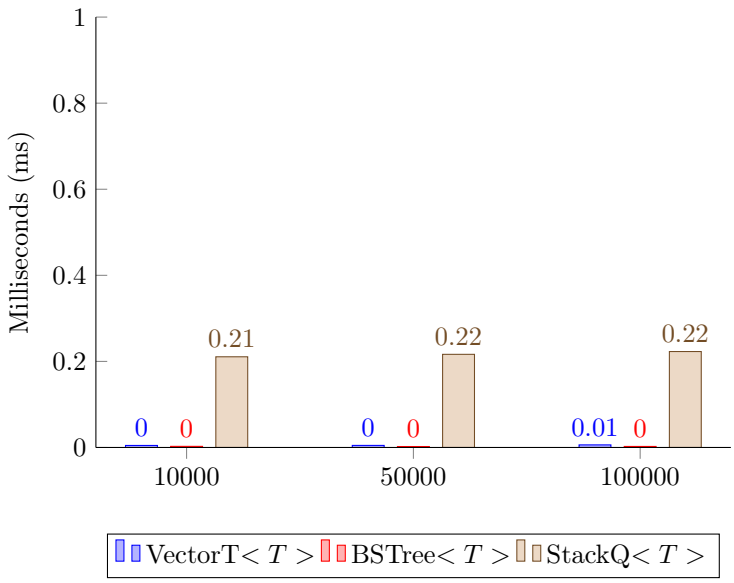

Figure 4: (Best Case) Running time of search function of 10,000 , 50,000, & 100,000 string items

(Best Case) Searching Running Time			
Input Size	**StackQ<T>**	**BST<T>**	**VectorT<T>**
10,000	0.2107 ms	0.0047 ms	0.0026 ms
50,000	0.2164 ms	0.0048 ms	0.0022 ms
100,000	0.2229 ms	0.006 ms	0.0024 ms

6.2 (Best Case) Running Time Analysis

The above graph represents the searching of 10,000 items, 50,000 items, & 100,000 items, which is measured in milliseconds (ms). As seen in the Figure 3, the running time of each the data structure vary.

In the scenario of searching a specified item, the best case exist when the target value exist at the beginning of the data structure.

VectorT<T> implements a linear search algorithm where its time complexity is of O(n). In the best case, the target value exist at the beginning of the array where it must only compare one value. The time complexity of this operation being O(1).

StackQ<T> also implements a linear search in which it must compare the target value to the top of the stack to see if a match occurs and removes the top of the stack if a match does not occur. In the best case, the target value exist at the top of the stack where its operation is of O(1).

BinarySearchTree<T> traverses through all nodes and searches for the target value where it must traverse through all N height to find the target value. In the best case, when beginning the search, the root is the target value which its operation is constant time of O(1).

7 Analysis: Sorting

Listings 10, 11, and 12 shows the insertion implementation of StackQ<T>, VectorT<T>, & BinarySearchTree<T>. StackQ<T> is implemented using the STL queue for its implementation of Stack. VectorT<T> is implemented using STL arrays for the implementation of Vector. BinarySearchTree<T> is implemented using nodes for the implementation of Binary Search Tree.

Listing 10: stackSort() Implementation w/ STL queue

```
template<typename T>
void Stack<T>::stackSort() {

    Stack<T> stackTemp;
    Stack<T> stack;

    stack.firstQ = firstQ;

    if (stack.empty()) { return; }

    while (!(stack.empty())) {
```

```
            T data = stack.peek();
            stack.qpop();

            while (!(stackTemp.empty())
                    && stackTemp.peek() < data) {

                stack.push(stackTemp.peek());
                stackTemp.qpop();
            }

            stackTemp.push(data);

    }

    firstQ = stackTemp.firstQ;
}
```

Listing 11: qSortHelper() Implementation w/ Array

```
template <typename T>
void VectorT<T>::qSortHelper(T arr[], int low, int high) {

    if (low < high) {

            int partionVal = partion(arr, low, high);

            qSortHelper(arr, low, partionVal - 1);
            qSortHelper(arr, partionVal + 1, high);

    }

}
```

Listing 12: inorder() w/ Node

```
template <typename T>
void BinarySearchTree<T>::inorder(BinarySearchTree* root) {

    if (root == NULL) { return;  }

    inorder(root->left);

    inorder(root->right);
}
```

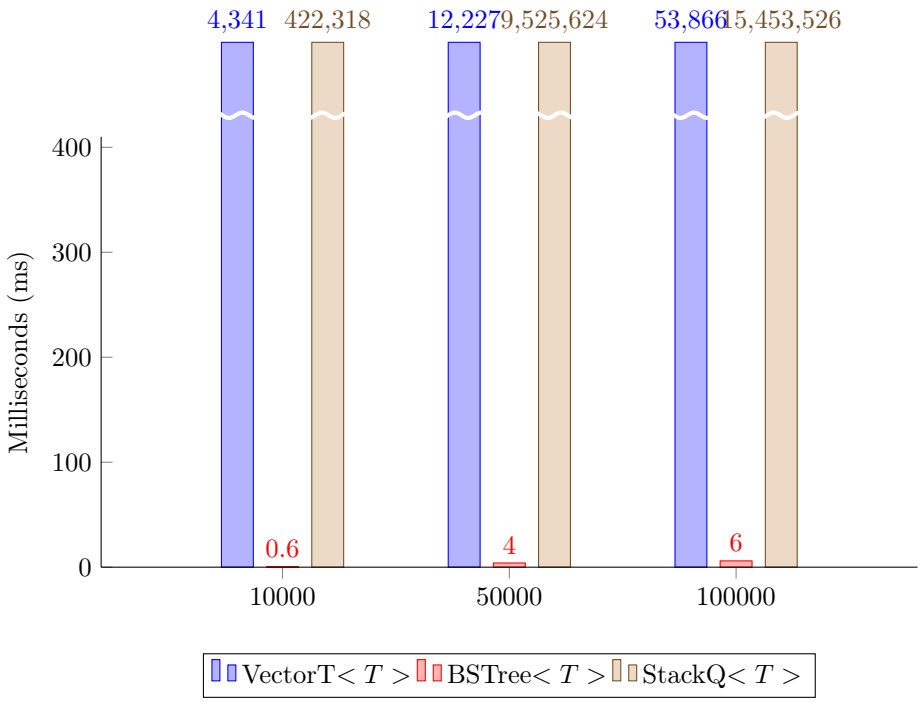

Figure 5: (Worst Case) Running time of sorting function of 10,000 , 50,000, & 100,000 string items

(Worst Case) Sorting Running Time			
Input Size	**StackQ<T>**	**BST<T>**	**VectorT<T>**
10,000	422,318 ms	0.6 ms	4341 ms
50,000	9,525,624 ms	4 ms	12227ms
100,000	15,453,526 ms	6 ms	53866 ms

7.1 (Worst Case) Running Time Analysis

The above graph represents the sorting of 10,000 items, 50,000 items, & 100,000 items, which is measured in milliseconds (ms). As seen in the Figure 5, the running time of the sorting these different data structures vary.

In sorting, most of the values in the data structures are at random and have no form of order when sorting. When looking at the worst case of sorting, when inputting an array to sort the array will in a descending form, where the elements go from largest to smallest. This means that in the process of sorting, it must revere the whole order of the array and take maximum time to sort.

VectorT<T> uses the implementation of quick sort where a pivot is chosen at random or median, depending on the situation. After the pivot is chosen, the algorithm forms to sub arrays and swaps the smaller value depending on the pivot that is chosen. In the worst case, the array inputted is in ascending/descending order where it must choose a pivot at random or median. This operation is of O(n^2).

StackQ<T> uses two temporary to sort the stack. The stack sort begins with pushing into the second stack which is empty. At this point, the next value that is pushed in comparesitself with the top of the stack in the second stack and pops if the value is smaller than the top of the stack an inserts the value into the correct position where the value compared is greater than the value below it in the stack. In the worst case, all elements in the stack are inputted in the descending order where it then must remove N elements in the stack and replace the bottom of the stack with the correct value. After this operation, it must push all values again into the stack. In the worst case, this operation is of O(n^2).

BinarySearchTree<T> does not have any sorting algorithms related to binary search trees. With this in mind, this document will use the in order traversal of the binary search tree to mimic the sorting occurring. When an in order traversal is done on a binary search tree, it will result in a sorted list of values. The traversal of all nodes in the binary search tree results in the time complexity of O(n).

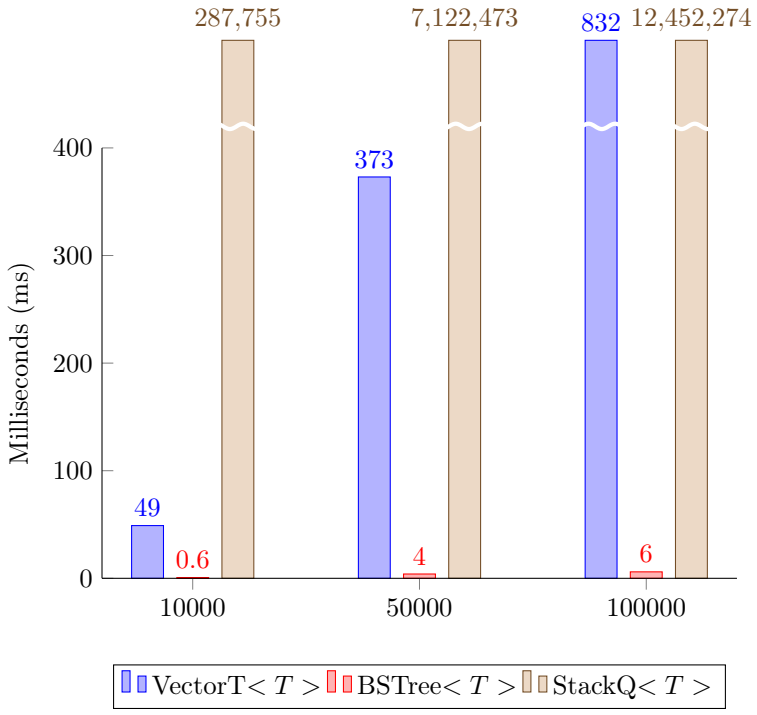

Figure 6: (Best Case) Running time of sorting function of 10,000 , 50,000, & 100,000 string items

(Best Case) Sorting Running Time			
Input Size	**StackQ<T>**	**BST<T>**	**VectorT<T>**
10,000	287,755 ms	49 ms	0.601 ms
50,000	7,122,473 ms	373 ms	4 ms
100,000	12,452,274 ms	832 ms	6 ms

7.2 (Best Case) Running Time Analysis

The above graph represents the sorting of 10,000 items, 50,000 items, & 100,000 items, which is measured in milliseconds (ms). As seen in the Figure 6, the running time of each the data structure vary.

As mentioned before in this document, the worst case when sorting a data set is to sort the inputted data in descending order where all elements are from the larger to smallest values. In the best case for sorting algorithms, the data set inputted is already sorted where the the sorting algorithm must not do so many comparison to the cases of worst case data set.

The VectorT<T> uses a quick sort implementation where its worst case is of O(n^2), in which all data set inputted are either sorted or in descending form. In the best case, quicksort performs O(n*log n), where the inputted the data set is not sorted or in descending form, but rather in random order.

The StackQ<T> uses the stackSort() where at the worst case is O(n^2). In the best case the stackSort() is of O(n), where the data set inputted is already sorted which allows for the stack to push N values into the second stack. This operation will then run at O(n) due to the stack just popping and pushing until it is empty.

BinarySearchTree<T> does not have any sorting algorithms related to binary search trees. With this in mind, this document will use the in order traversal of the binary search tree to mimic the sorting occurring. When an in order traversal is done on a binary search tree, it will result in a sorted list of values. The traversal of all nodes in the binary search tree results in the time complexity of O(n).

8 Conclusions

This document discusses the efficiency of the different data structure functions. This includes: insertion, deletion, searching, & sorting. It analyzes the performance of these different functions and compares them to each other. While one of the data structure was more efficient for one job, it may be slower at other jobs. As seen with deletion stack & binary search tree deletion, the difference is significant in terms of time. With the grow of data, the time it takes to complete a task increases, and with more less efficient data structures it can be the difference between minutes and hours.