QUAID-E-AZAM UNIVERSITY, ISLAMABAD

# Data structures Assignment:

# Analysis of ADTs

**SUBMITTED TO:**

SIR RABEEH AYYAZ ABBASI

**SUBMITTED BY:**

AMNA MUZAFFAR

04072013003

**DEPARTMENT OF COMPUTER SCIENCE, QUAID-E-AZAM UNIVERSITY, ISLAMABAD**

# DATA STRUCTURES AND ALGORITHMS

## (CS-211)

In this Course, we studied different general purpose and special purpose ADTs (Abstract Data Types). Also, some types of **sorting algorithms** (Selection Sort, Bubble Sort, Insertion Sort).

In **Special Purpose ADTs**, we cover Stacks, Queues and Priority Queues using Heap. Whether in **General Purpose ADTs**, we have covered following ADTs:

1. LinkedList (Unsorted)
2. LinkedList (Sorted)
3. STL List(**list**)
   (We have also implemented Array based List.)
4. Binary Search Tree (Recursive/Iterative)
5. STL Binary Search Tree (**map**)
6. Hash Tables with chaining and division method
7. Hash Tables with chaining and multiplication method
8. Hash Tables with open addressing and linear probing
9. Hash Tables with open addressing and quadratic probing
10. Hash Tables with open addressing and double hashing probing
11. STL Hash Tables (**unordered map**)

In this Report, I will evaluate the performance of above-mentioned ADTs if we wish to execute them for large number of values (10,0000). Functions I will Evaluate are:

- Insert
- Search (random 10,000 values that exist in array)
- Search (random 10,000 values that does not exist in array)
- Delete (random 10,000 values that exist in array)

I have implemented these 11ADTs for 50 thousand (50,000) randomly generated values that were stored in array and have made observations 10 times. And calculated execution time of each ADT for each function and stored it in **".csv"** file.

## STATISTICAL ANYLYSIS:

1. **Linked List (Unsorted):**

In Linked list (unsorted), we simply insert values at the end without checking whether it is smaller or greater than previously added value. Also searching and deleting is done with scanning list from beginning to end.

**Time Complexities:**

- **INSERT:**
  Worst-Case=O (1)
  Best-Case=O (1)
- **DELETE:**
  Worst-Case=O (n)
  Best-Case=Ω (1)
- **SEARCH:**
  Worst-Case=O (n)
  Best-Case= Ω (1)

//n is number of values in list

2. **Linked List (Sorted):**

In Linked list (Sorted), we insert values in ascending/descending order. So, at the end, we get list in sorted form. Searching and deleting is same as in linked list(unsorted).

**Time Complexities:**

- **INSERT:**
  Worst-Case =O (n)
  Best-Case= Ω (1)

- **DELETE:**
  Worst-Case=O (n)
  Best-Case=Ω (1)
- **SEARCH:**
  Worst-Case=O (n)
  Best-Case=Ω (1)

//n is number of values in list

3. **List-STL (list)**

In STL-based list, all functionalities are performed by using iterators.

**<u>Time Complexities:</u>**

- **INSERT:**
  Worst-Case=O (1)
  Best-Case=Ω (1)
- **DELETE:**
  Worst-Case=O (n)
  Best-Case= Ω (1)
- **SEARCH:**
  Worst-Case=O (n)
  Best-Case= Ω (1)

//n is number of values in list

4. **Binary Search Tree (Iterative):**

In BST, the main concept is that our whole work is divided into two parts at each step. In Insert, Delete and Search, we scan from root and if value is smaller than root, we ignore right side and went to left side and if value is greater than root, we consider right side. Same process follows every time.

I have implemented BST iteratively as recursive method is not efficient for large number of values.

**<u>Time Complexities:</u>**

- **INSERT:**
  Worst-Case/Best-Case=O (log(n))
- **DELETE:**
  Worst-Case/Best-Case= O (log(n))
- **SEARCH:**
  Worst-Case/Best-Case= O (log(n))

//n is number of values in list

5. **Binary Search Tree STL based (map):**

In STL based BST "map", there are two parts of each node. It is in form of pair one having key and other having value.

**Time Complexities:**

- **INSERT:**
  Worst-Case/Best-Case=O (log(n))
- **DELETE:**
  Worst-Case/Best-Case= O (log(n))
- **SEARCH:**
  Worst-Case/Best-Case= O (log(n))

//n is number of values in list

## 6. Hash Tables (chaining through multiplication)

In Hash tables, our main aim is to reduce time complexity from O(log(n)) to O (1). It is only possible that we insert some value and it is directly inserted at that place also in search and delete same thing happens. In chaining method, we create an array and each index of array consists of list in order to avoid collision. We calculate index by using formula:

m*(k*a)-int(k*a))

**Time Complexities:**

- **INSERT:**
  Worst-Case=O(n)
  Best-Case=Ω (1)
  Average-Case=θ (1)
- **DELETE:**
  Worst-Case=O(n)
  Best-Case=Ω (1)
  Average-Case=θ (1)

- **SEARCH:**
  Worst-Case=O(n)
  Best-Case=Ω (1)
  Average-Case=θ (1)

## 7. Hash Tables (chaining through division)

It is same as Hash tables with multiplication with only change in formula:

K%m;

**Time Complexities:**

- **INSERT:**
  Worst-Case=O(n)
  Best-Case=Ω (1)
  Average-Case=θ (1)
- **DELETE:**
  Worst-Case=O(n)
  Best-Case=Ω (1)
  Average-Case=θ (1)

- **SEARCH:**
  Worst-Case=O(n)
  Best-Case=Ω (1)
  Average-Case=θ (1)

## 8. Hash Tables (open addressing through linear probing)

In open addressing, we directly insert values on indexes by creating probing sequences,

Formula for linear probing is: `idxx = (idx + i) % m;`

**Time Complexities:**

- **INSERT:**
  Worst-Case=O(n)
  Best-Case=Ω (1)
  Average-Case=θ (1)
- **DELETE:**
  Worst-Case=O(n)
  Best-Case=Ω (1)
  Average-Case=θ (1)

- **SEARCH:**

Worst-Case=O(n)
Best-Case=Ω (1)
Average-Case=θ (1)

## 9. Hash Tables (open addressing through quadratic probing)

In open addressing, we directly insert values on indexes by creating probing sequences,

Formula for linear probing is: `idxx = (idx + c1 * i + c2 * i * i) % m;`

**Time Complexities:**

- **INSERT:**
  Worst-Case=O(n)
  Best-Case=Ω (1)
  Average-Case=θ (1)
- **DELETE:**
  Worst-Case=O(n)
  Best-Case=Ω (1)
  Average-Case=θ (1)
- **SEARCH:**
  Worst-Case=O(n)
  Best-Case=Ω (1)
  Average-Case=θ (1)

## 10.Hash Tables (open addressing through double hashing)

In open addressing, we directly insert values on indexes by creating probing sequences,

Formula for linear probing is: `idxx = (idx + i * (1 + h2)) % m;`

**Time Complexities:**

- **INSERT:**
  Worst-Case=O(n)
  Best-Case=Ω (1)
  Average-Case=θ (1)

- **DELETE:**

  Worst-Case=O(n)
  Best-Case=Ω (1)
  Average-Case=θ (1)

- **SEARCH:**

  Worst-Case=O(n)
  Best-Case=Ω (1)
  Average-Case=θ (1)

**11.Hash Tables STL based (unordered map)**

It is same as map stl but in unordered_map, duplicate values can be stored.

**Time Complexities:**

- **INSERT:**

  Worst-Case=O(n)
  Best-Case=Ω (1)
  Average-Case=θ (1)

- **DELETE:**

  Worst-Case=O(n)
  Best-Case=Ω (1)
  Average-Case=θ (1)

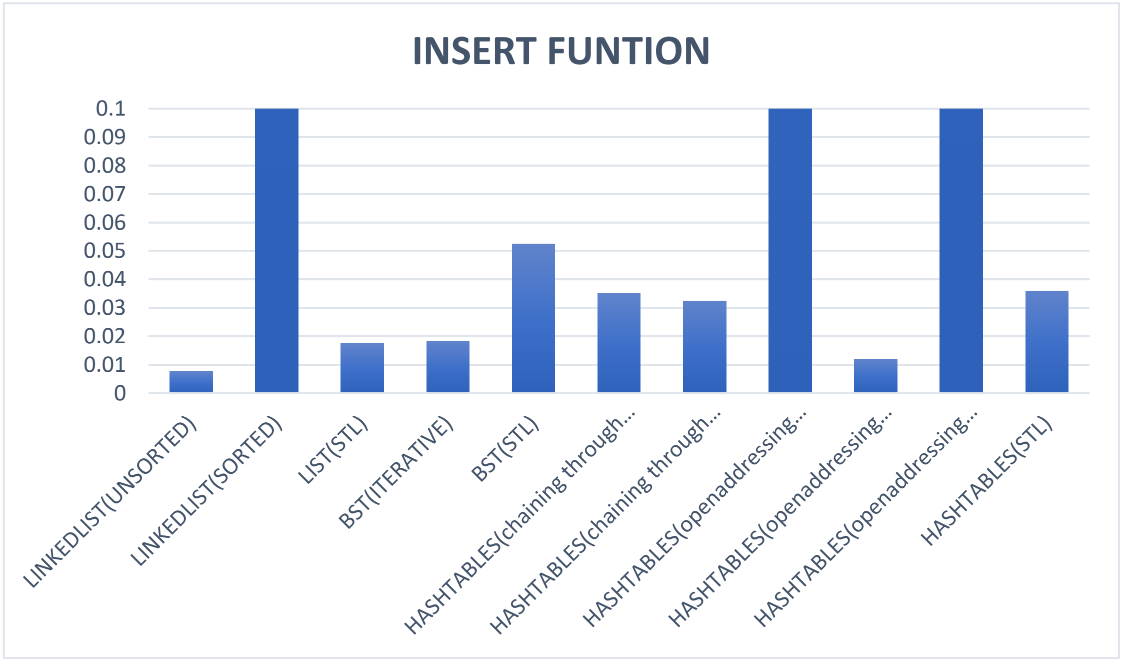
- **SEARCH:**

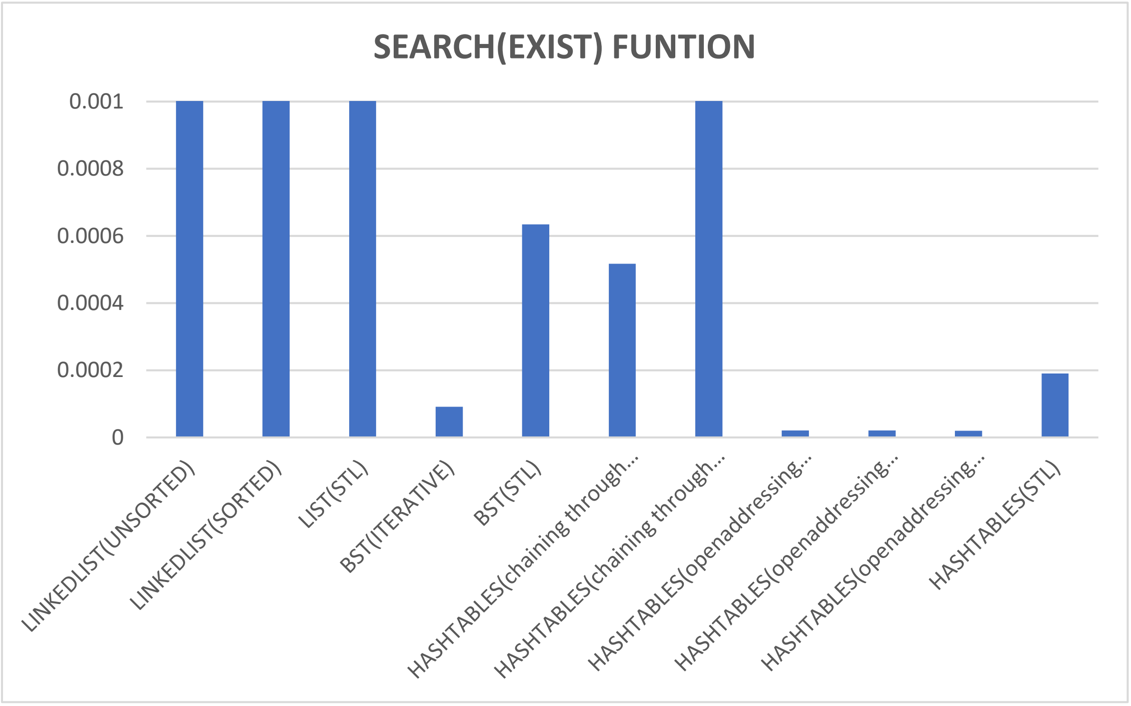
  Worst-Case=O(n)
  Best-Case=Ω (1)
  Average-Case=θ (1)

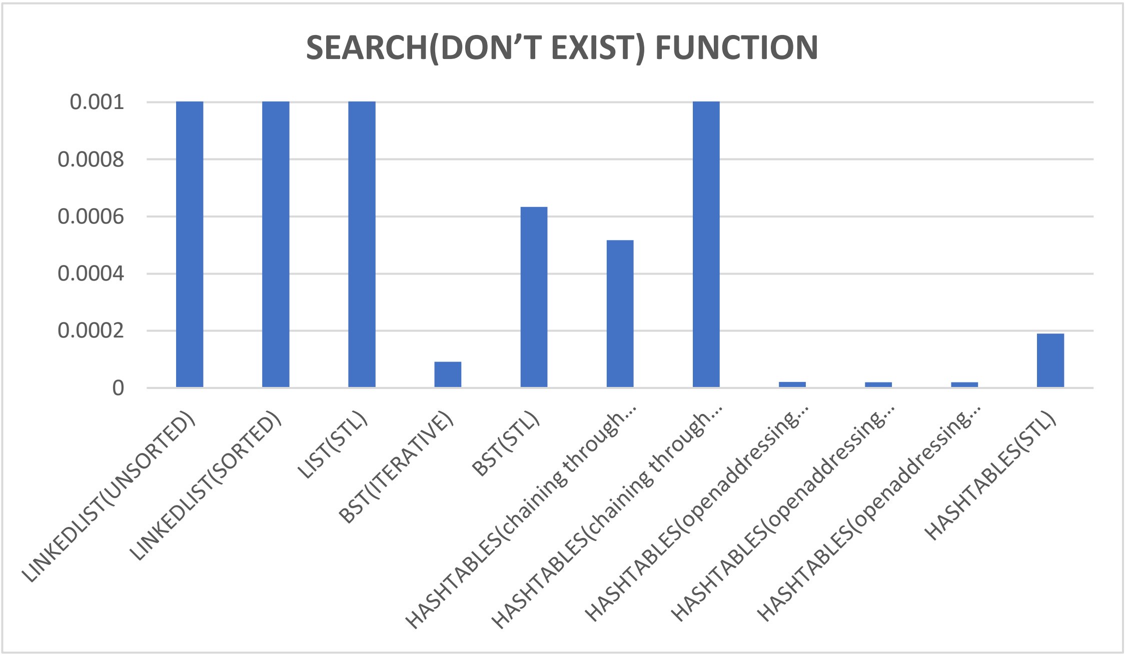
## OBSERVATIONS:

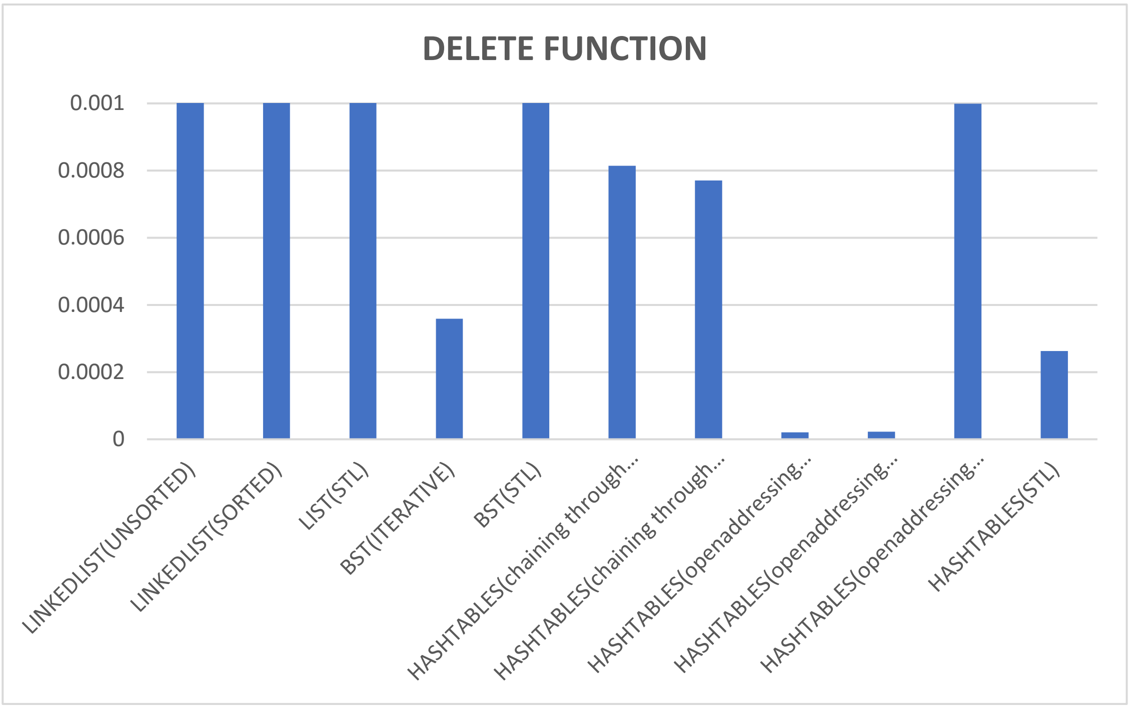
| AVERAGES OF FUNCTIONS: | | | | |
|---|---|---|---|---|
| | INSERT | SEARCH(EXIST) | SEARCH(DONT EXIST) | DELETE |
| LINKEDLIST(UNSORTED) | 0.00769587 | 0.278316 | 0.584527 | 0.252316 |
| LINKEDLIST(SORTED) | 10.1032 | 0.474773 | 1.06852 | 0.456472 |
| LIST(STL) | 0.0173905 | 10.0778 | 20.5135 | 0.496026 |
| BST(ITERATIVE) | 0.018317 | 9.16E-05 | 3.26E-05 | 0.00035881 |
| BST(STL) | 0.0525369 | 0.00063384 | 0.00053232 | 0.00134278 |
| HASHTABLES(chaining through Division) | 0.0350882 | 0.00051675 | 0.00253656 | 0.00081383 |
| HASHTABLES(chaining through multipication) | 0.0324359 | 0.00954405 | 0.230311 | 0.00077021 |
| HASHTABLES(openaddressing through linear probing) | 1.71929 | 2.11E-05 | 0.211781 | 2.10E-05 |
| HASHTABLES(openaddressing through Quadratic probing) | 0.012091 | 2.03E-05 | 0.322435 | 2.21E-05 |
| HASHTABLES(openaddressing through Double-Hashing) | 11.321 | 2.01E-05 | 0.215 | 0.00099938 |
| HASHTABLES(STL) | 0.0360021 | 0.00018999 | 0.00019722 | 0.00026297 |

## GRAPHS:




SEARCH(EXIST) FUNTION
0.001
0.0008
0.0006
0.0004
0.0002
0
LINKEDLIST(UNSORTED)
LINKEDLIST(SORTED)
LIST(STL)
BST(ITERATIVE)
BST(STL)
HASHTABLES(chaining through...
HASHTABLES(chaining through...
HASHTABLES(openaddressing...
HASHTABLES(openaddressing...
HASHTABLES(openaddressing...
HASHTABLES(STL)


SEARCH(DON'T EXIST) FUNCTION
0.001
0.0008
0.0006
0.0004
0.0002
0
LINKEDLIST(UNSORTED)
LINKEDLIST(SORTED)
LIST(STL)
BST(ITERATIVE)
BST(STL)
HASHTABLES(chaining through...
HASHTABLES(chaining through...
HASHTABLES(openaddressing...
HASHTABLES(openaddressing...
HASHTABLES(openaddressing...
HASHTABLES(STL)



## <u>CONCLUSION:</u>

Now if we look at table and graphs, we conclude that for **INSERTION**, all ADTS work in similar manner except sorted linked list which takes n steps and it is very time consuming and BST which takes log(n) steps but it will help us in searching.

In **SEARCHING** and **DELETION**, we conclude that BST and Hash Tables are working efficiently as BST is in O(log(n)) and Hash Tables in O (1).

**So overall, Hash Tables are more effective if we want to store large number of values and wants quick search but one drawback is that space is wasted if not used properly.**