Project 2 - Binary Trees and their variants

Pushkar Gupta, 13365342

May 2018

1. Analysis Summary In this project four data structures were implemented and tested on search, insert and delete operations. The test cases comprised of two cases - randomized keys and monotonically increasing keys. The data structures are - **Binary Search Trees, AVL Trees, Treaps Skip Lists**. AVL Trees are strictly balanced binary search trees, Treaps (tree + heap) are randomly balanced BSTs, and skip-lists although not trees - but are fundamentally serve the same purpose as random BSTs.

To compare the performance of these hashing schemes, worst-case and amortized run time were measured. The operations were tested as a function of the number of elements present in the structure at the time. The number of elements were incremented by +100, starting from 100 upto 1,000 maximum elements. The following are the specifications of the test-cases for all the data structures:

1) Integer objects were generated and were used as keys to test the functionality and performance of the four data structures.

2) After inserting a number of elements (integer objects) into a structure, worst-case and amortized time for another 20 operations (insert, search, delete) was measured. This run-time (for next 20 elements) has been plotted and compared for the various structures.

3) For the **Amortized case**, the keys were generated in a random order but to determine the **Worst case** performance, monotonically increasing keys were used. This was done because the worst case (insert/search/delete) across all the operations occurs when we add sorted (ascending/descending) keys sequentially. For the amortized case the average run time of the 20 operations was noted, while for the worst case the maximum run time of the 20 operations was noted as the run-time for that particular structure, as a parameter of the number of elements.

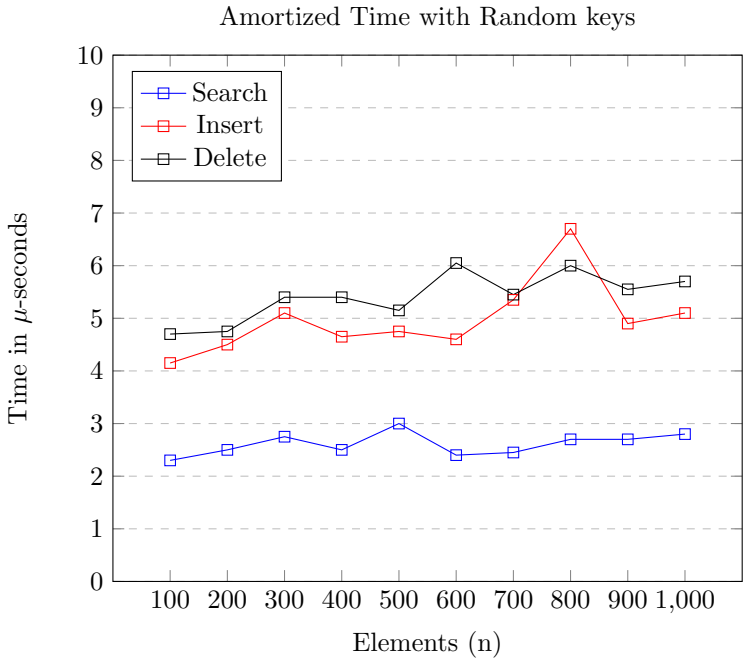
2. Data Structures

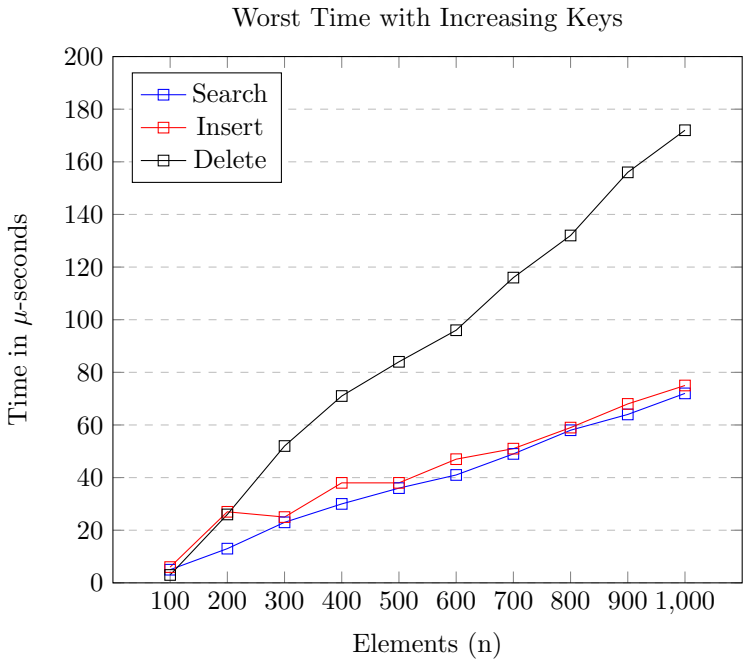
2.1 Binary Search Tree A standard binary search tree (implemented without any height balancing constraints), has the following performance characteristics. For search/insert/delete, the worst-case time is O(n) and the amortized/average case time in O($log_2 n$). The worst case can be created trivially using increasing keys, such that on every successive insert/search/delete - all the previously inserted elements have to be traversed. The randomized case has O($log_2 n$) performance (as per lecture slides). This happens because at every node, entire subtrees are skipped due to the structure of the BST.

2.1.1 Pseudo Code The pseudo-code for search/insert/delete for BST is:

```
def search(value):
    node = tree.root
    while node:
        if node.value == value:
            return True  # value found!
        elif value < node.value:
            node = node.left
        else:
            node = node.right

    return False  # default, not found!

def insert(value):
    node = tree.root
    newnode = Node(value)  # creat a new object

    if tree.root is None:
        tree.root = newnode
        return True

    while node:
        if node.value == value:
            return True  # value already exists
        elif value < node.value:
            if node.left is None:
                node.left = newnode
                return True
            else:
                node = node.left
        else:
            if node.right is None:
                node.right = newnode
                return True
            else:
                node = node.right

def delete(value):
    node = tree.objectsearch(value)  # get the node object
    if not node:
        return False  # value doesn't exist

    if node has less than 2 children:
        replace node with left or right subtree
    else:
        successor = tree.getsuccessor(node)  # get next successor
        value_to_put = successor.value
        delete(successor.value)  # recursive call
        node.value = value_to_put  # replace original value with successor
        return True
```

2.1.2 Experimental run time The plots of the empirical data for search/insert/delete operations are as follows:





2.1.3 Observation One can very clearly observe that the experimental run time for all the operations for random keys is very close the the theoretical run-time, which is O($log_2 n$), and the time for increasing keys is O(n), as expected.

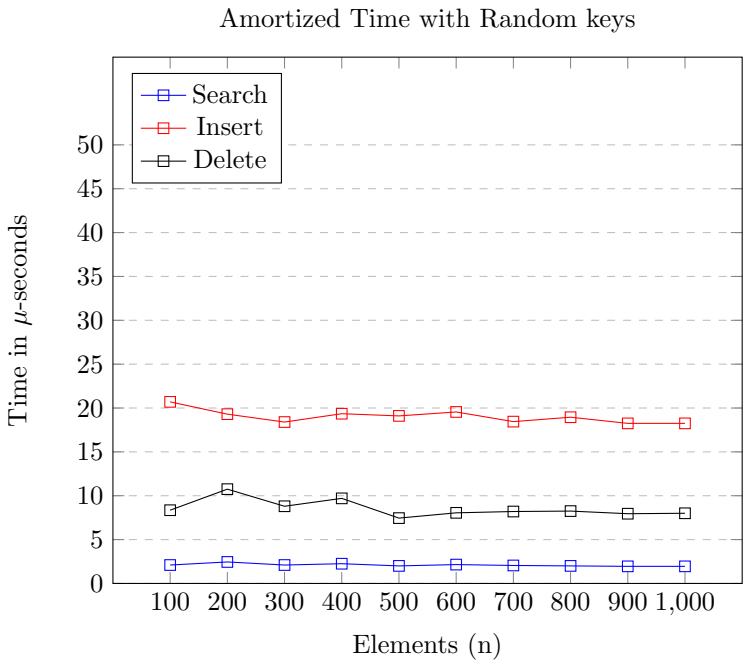
2.2 AVL Trees AVL trees are strictly balanced binary search trees. Whenever the height of the left subtree and the right subtree of a node, differ by more than 2, we rotate on the node to rebalance the tree at that point. As there are never long subtrees, search is always bound by O($log_2 n$), both in amortized and worst-case. After every insert/delete there will be at most 2 rotations (which are constant time operations). Hence, the amortized and worst case run time for both insert/ delete operations for AVL trees is O($log_2 n$). The implementation of AVL trees in this project uses node objects which store integers (keys), left and right child pointers, parent pointers, and the height of the current node.

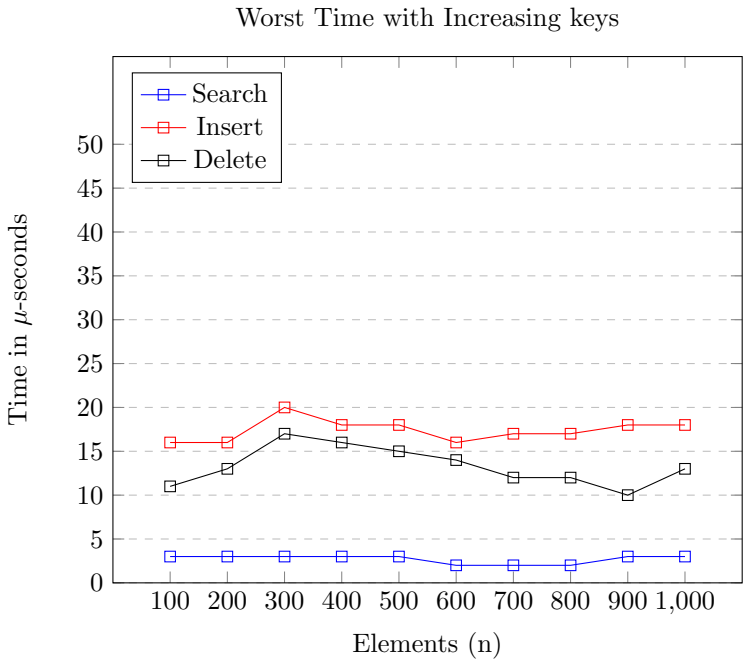
2.2.1 Pseudo-Code The pseudo-code for search/insert/delete for AVL Trees is:

```
def search(value):
    node = tree.root
    while node:
        if node.value == value:
            return True  # value found!
        elif value < node.value:
            node = node.left
        else:
            node = node.right

    return False  # default, not found!

def insert(value, entry=None):
    if entry is None:
        node = tree.root
    else:
        node = entry  # the entry point (this is a recursive funtion)
    newnode = Node(value)  # creat a new object, but this is used only once
                           # during the entire function calls stack

    if tree.root is None:
        tree.root = newnode
        return True

    while node:
        if node.value == value:
            return True  # value already exists
        elif value < node.value:
            if node.left is None:
                node.left = newnode
                break
            else:
                node = node.left
                insert(value, entry=node.left)
        else:
            if node.right is None:
                node.right = newnode
                break
            else:
                node = node.right
                insert(value, entry=node.right)

    # inserted, now rebalance
    if abs(node.left.height - node.right.height) > 2:
        rebalance(node)
    else:
        update_height(node)
    return True
    ## The rebalance step determines what side to rotate on
    ## and whether only one rotation is required or two. See source code for details.
```

```
def delete(value):
    node = tree.objectsearch(value)  # get the node object
    if not node:
        return False  # value doesn't exist

    if node has 2 children:
        successor = tree.getsuccessor(node)  # get next successor
        value_to_put = successor.value
        delete(successor.value)  # recursive call
        node.value = value_to_put  # replace original value with successor
        return True
    else:
        replace node with left or right subtree
        parent = node.parent  # now traverse up
        while parent:
            get heights and do rebalancing if required
            parent = parent.parent
        return True
```

2.2.2 Experimental run time The plots of the empirical data for search/insert/delete operations are as follows:





2.2.3 Observation One can very clearly observe that the experimental run time for all the operations for random keys is very close the the theoretical run-time, which is O($log_2 n$), and the time for increasing keys is O(n), as expected.

2.3 Treaps Treap structure is a randomized alternative to balanced binary trees. Other than maintaining the ordering structure of keys as in a BST, treaps also store a priority. Treaps are heap ordered with respect to this priority, which is a random number generated when the node object is first created. The heap-ordering constraint randomizes the tree structure by moving nodes up or down, which can increase or decrease performance. There is no hard constraint to keep the tree balanced as in AVL, but randomizing helps in evading cases like - "adding increasing keys sequentially, where a BST would fail miserably".

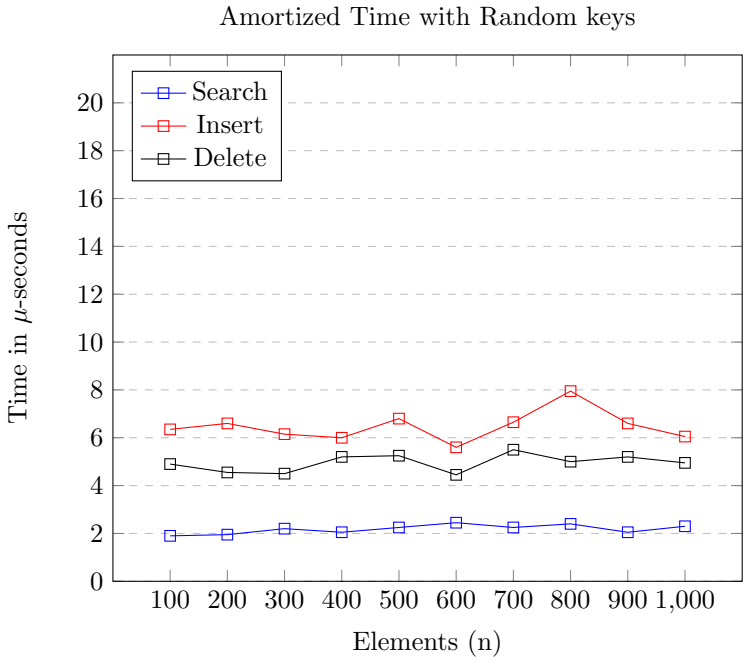
The implementation of treaps in this project uses node objects which store integers (keys), priority (float between 0 and 1), left and right child pointers, and a parent pointer. The expected time for search/insert/delete operations for treaps in $O(log_2 n)$, as per the lecture slides. The heap property is maintained using tree rotations, and the expected number of rotations per insert, is 2.

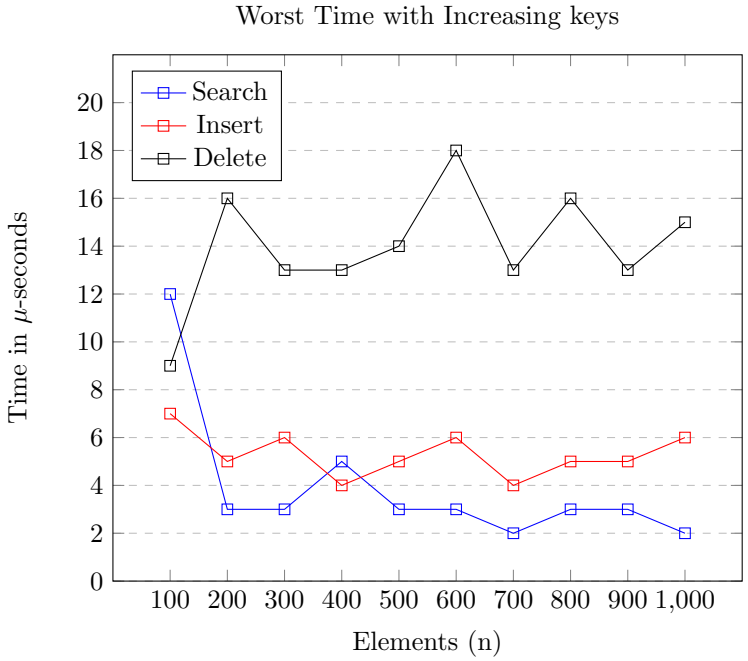
2.3.1 Pseudo-Code The pseudo-code for search/insert/delete for treaps is:

```
def search(value):
    node = tree.root
    while node:
        if node.value == value:
            return True  # value found!
        elif value < node.value:
            node = node.left
        else:
            node = node.right

    return False  # default, not found!

def insert(value):
    newnode = Node(value)  # creat a new object (with randomized priority)
    if tree.root is None:
        tree.root = newnode
        return True

    while node:
        if node.value == value:
            return True  # value already exists
        elif value < node.value:
            if node.left is None:
                node.left = newnode
                break
            else:
                node = node.left
        else:
            if node.right is None:
                node.right = newnode
                break
            else:
                node = node.right

    # now, rotate up to satisy heap-ordering
    parent = node.parent
    while parent:
        if node.priority >= parent.priority:
            return True
        else:
            rotate_up(node)  # left-or-right depending on parent & node value
            parent = node.parent
```

```
def delete(value):
    node = tree.root
    while node:  # search for the object
        if node.value == value:
            break
        elif value < node.value:
            node = node.left
        else:
            node = node.right

    if not node:
        return False  # value doesn't exist

    while node is not leaf:
        node_up = max of node.right.value, node.left.value
        rotate_up(node_up)

    del node  # now, node is a leaf node
    return True
```

2.3.2 Experimental run time The plots of the empirical data for search/insert/delete operations are as follows:





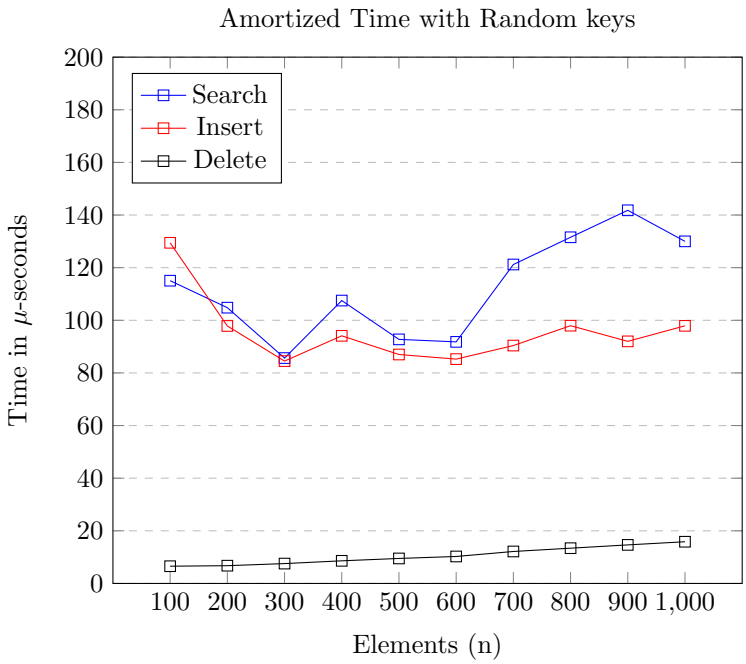
2.3.3 Observation One can observe some spikes across all the operations, for both randomized and increasing keys, but it is roughly O($log_2 n$). This is much better than the baseline O(n) of standard binary trees.

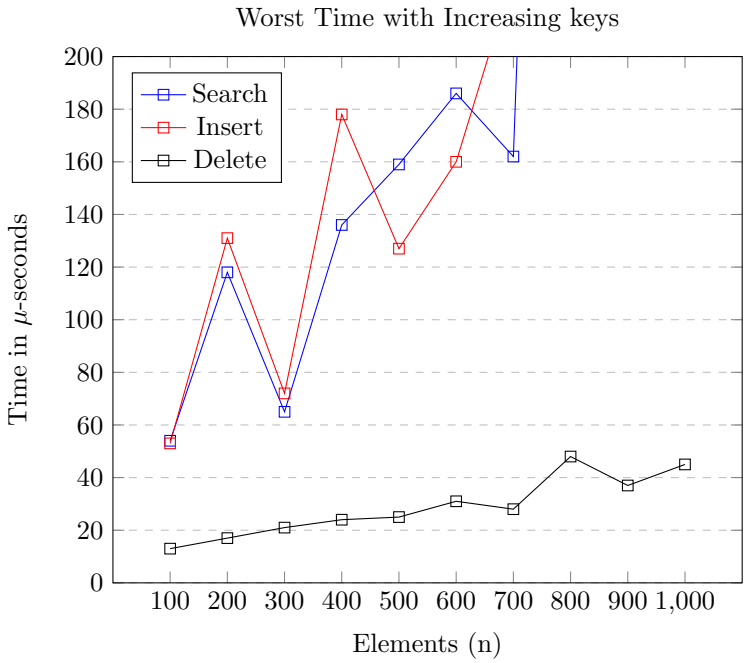
2.4 Skip Lists Skip lists are a structure of ordered linked-list which function and perform roughly like randomly balanced binary search trees. Fast search is made possible by maintaining a linked hierarchy of subsequences, with each successive subsequence skipping over fewer elements than the previous one. The skip list implemented in this project grow to the upper level both during search and insertion of new elements. The probability of a visited node growing to the upper level was set at 0.3.

2.4.1 Pseudo-Code The pseudo-code for search/insert/delete for treaps is:

```
def search(value):
    node = lists.entry  # top left element
    while node:
        if node.value == value:
            return True, node   # return True, and the SUCCESSPOINT
        if choose(1, 0, chance=0.3) == 1:  # grow on search path
            insertup(node.value, currentlevel)
        if node.next.value <= value:
            node = node.next
        else:
            node = node.down

    return False, node  # return False, and the FAILPOINT

def insert(value):
    status, failpoint = search(value)
    if status == True:
        return True  # elt already exists

    # failpoint will be on the bottom-most level
    # insert b/w failpoint and failpoint.next

    newnode = Node(value)
    newnode.next = failpoint.next
    failpoint.next = newnode
    newnode.prev = failpoint
    newnode.next.prev = newnode
    return True

def delete(value):
    status, successpoint = search(value)
    if status == False:
        return False  # elt not present

    # succespoint may be at the top of its column
    # delete every entry recursively
    while succespoint:
        del successpoint  # also fix the pointers
        successpoint = succespoint.down

    return True
```

2.4.2 Experimental run time The plots of the empirical data for search/insert/delete operations are as follows:

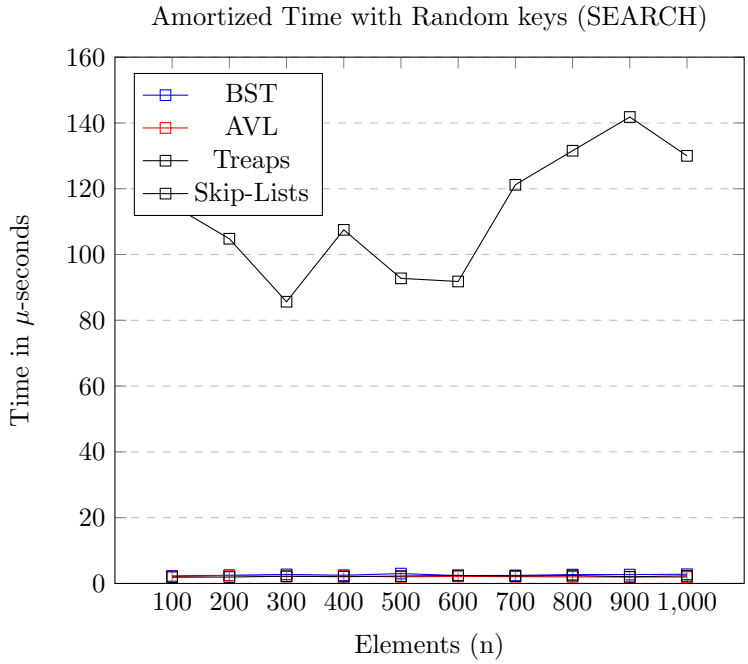


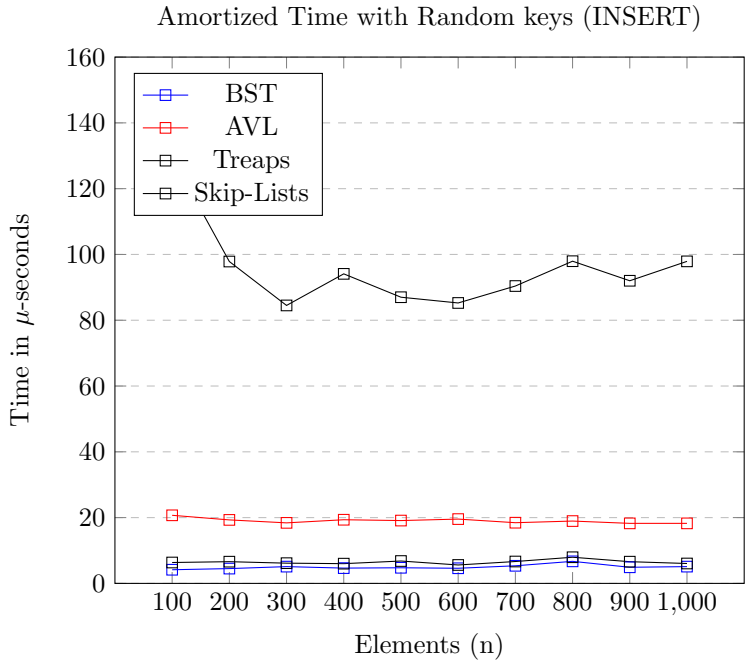


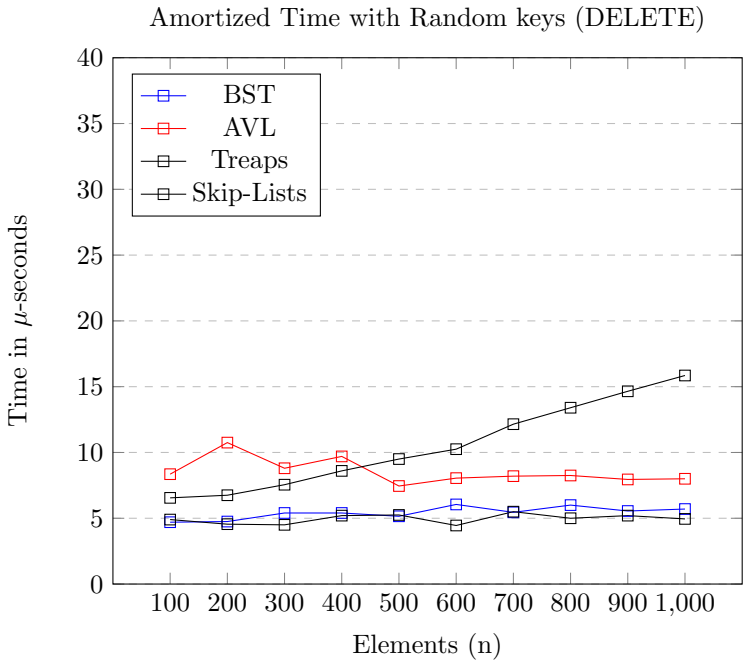
2.4.3 Observation For randomized keys, the run times are pretty much O($log_2 n$). The spikes are due to expansion of a node to the upper level. As there are no expansions during delete, the delete curve is smooth. For the case of increasing keys, search and insert become linear, which is plausible. There is very low chance of keys inserted recently, to have been expanded to the top, hence they will be at the bottom end. So, to search/insert them we have traverse all the way down to the end.

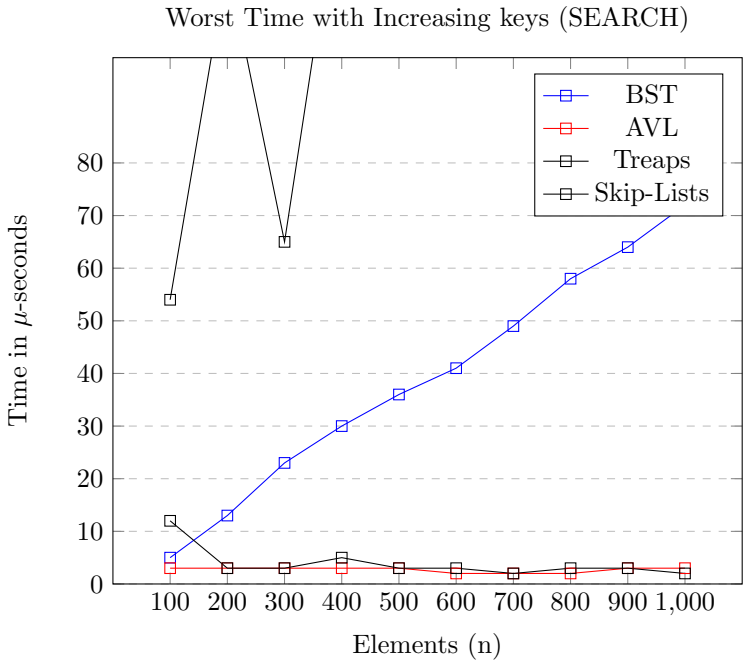
3. Comparitive analysis In the following section we compare the search/insert/delete performance for the four structures, for randomized and increasing keys.

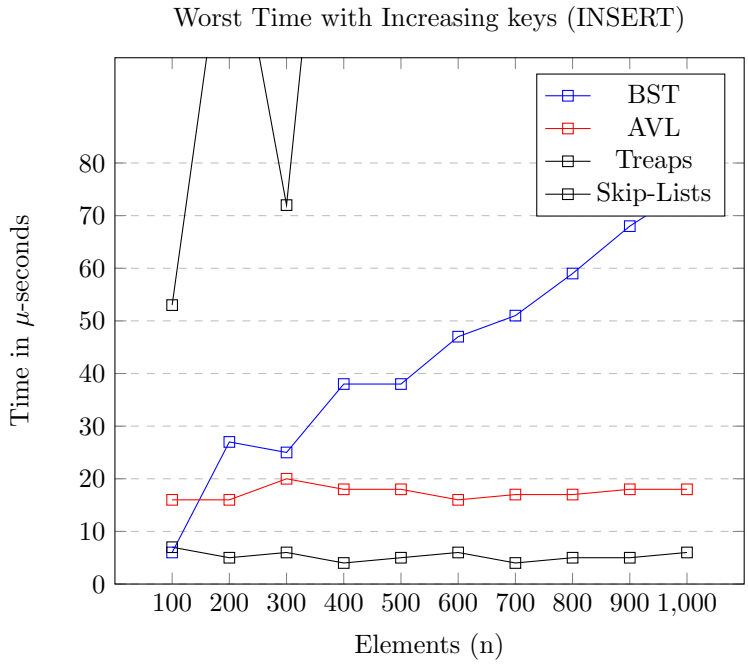
3.1 Plots

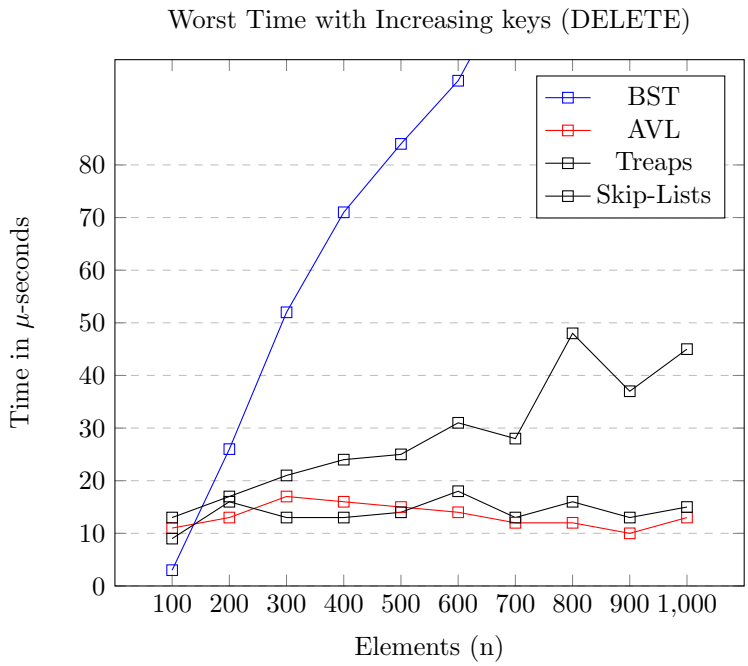





Amortized Time with Random keys (DELETE)
BST
AVL
Treaps
Skip-Lists
Time in μ-seconds
Elements (n)
0
5
10
15
20
25
30
35
40
100
200
300
400
500
600
700
800
900
1,000


Worst Time with Increasing keys (SEARCH)
BST
AVL
Treaps
Skip-Lists
Time in μ-seconds
Elements (n)
0
10
20
30
40
50
60
70
80
100
200
300
400
500
600
700
800
900
1,000





3.2 Conclusion When the keys are randomized, search and insert (with the exception of skip-lists), is O($log_2 n$). Skip lists have spikes because they expand to the upper level with a probability p=0.3, during search/insert. Delete operation is almost O($log_2 n$) for all, including skip-lists. There are no expansions during delete, hence even skip lists perform well. Observing the worst-case in the previous sections, only AVL trees have strict O($log_2 n$) performance over all operations. Standard BSTs fail miserabily to O(n), during worst case, while treaps and skip-lists are much better than BSTs but not than AVL Trees.