DATA STRUCTUTRES PROJECT '13

B+- v/s Skip Lists

Kalpit Thakkar :201201071
Jigar Thakkar:201201072

SKIP LISTS

Skip lists are probabilistic data structures that can be used in place of balanced trees. Skip lists use probabilistoc balancing rather than strictly enforced balancing and as a result the algorigthms for insertion and deletion in skip lists are much simpler and significantly faster than equivalent algorithms for balanced trees.

Balancing a data structure probabilistically is easier than explicitly maintaining balance. For many applications, skip lists are a more natural representation than trees, also leading to simpler algorithms. The simplicity of skip list algorithms makes them easier to implement and provides significant constant factor speed improvements over balanced tree and self-adjusting tree algorithms. Skip lists are also very space efficient. They can easily be configured to require an average of 1.33 pointers per element (or even less) and do not require balance or priority information to be stored with each node.

INITIALIZATION

An element NIL is allocated and given a key greater than any legal key. All levels of all skip lists are terminated with NIL. A new list is initialized so that the the level of the list is equal to 1 and all forward pointers of the list's header point to NIL.

SEARCH ALGORITHMS

We search for an element by traversing forward pointers that do not overshoot the node containing the element being searched for. When no more progress can be made at the current level of forward pointers, the search moves down to the next level. When we can make no more progress at level 1, w e must be immediately in front of the node that contains the desired element (if it is in the list).

PSEUDOCODE:

```
Search(list, searchKey)
    x := list→header
    -- loop invariant: x→key < searchKey
    for i := list→level downto 1 do
        while x→forward[i]→key < searchKey do
            x := x→forward[i]
    -- x→key < searchKey ≤ x→forward[1] →key
    x := x→forward[1]
    if x→key = searchKey then return x→value
        else return failure
```

INSERT AND DELETE ALGORITHMS

To insert or delete a node, we simply search and splice. A vector update is maintained so that when the search is complete (and we are ready to per form the splice), update[i] contains a pointer to the rightmost node of level i or higher that is to the left of the location of the insertion/deletion. If an insertion generates a node with a level greater than the previous maximum level of the list, we update the maximum level of the list and initialize the appropriate portions of the update vector. After each deletion, we check if we have deleted the maximum element of the list and if so, decrease the maximum level of the list.

INSERT PSEUDOCODE

```
Insert(list, searchKey, newValue)
    local update[1..MaxLevel]
    x := list→header
    for i := list→level downto 1 do
        while x→forward[i]→key < searchKey do
            x := x→forward[i]
        -- x→key < searchKey ≤ x→forward[i]→key
        update[i] := x
    x := x→forward[1]
    if
        x→key = searchKey then x→value := newValue
    else
        lvl := randomLevel()
        if lvl > list→level then
            for i := list→level + 1 to lvl do
                update[i] := list→header
            list→level := lvl
        x := makeNode(lvl, searchKey, value)
        for i := 1 to level do
            x→forward[i] := update[i]→forward[i]
            update[i]→forward[i] := x
```

DELETE PSEUDOCODE

```
Delete(list, searchKey)
    local update[1..MaxLevel]
    x := list→header
    for i := list→level downto 1 do
        while x→forward[i]→key < searchKey do
                x := x→forward[i]
        update[i] := x
    x := x→forward[1]
    if x→key = searchKey then
        for i := 1 to list→level do
            if update[i]→forward[i] ≠ x then break
            update[i]→forward[i] := x→forward[i]
        free(x)
        while list→level > 1 and
list→header→forward[list→level] = NIL
        do
            list→level := list→level – 1
```

ANALYSIS OF SEARCH (EXPECTED TIME)

In a list of n element if we have to climb from level 1 to any certain level L(n).
We use this analysis go up to level L(n) and use a different analysis technique for the rest of the journey. The number of leftward movements remaining is bounded by the number of elements of level L(n) or higher in the entire list, which has an expected value of 1/p.
We also move upwards from level L(n) to the maximum level in the list. The probability that the maximum level of the

list is a greater than k is equal to 1–(1–pk)n , which is at most npk. We can calculate the expected maximum level is at most L(n) + 1/(1–p). Putting our results together
Thus,

Total expected cost to climb out of a list of n elements
$\leq$ L(n)/p + 1/(1–p) which is O(log n).

B+ TREES

The regular B-tree is a disk-based data structure for indexing records based on an ordered key set. The B^+-tree is a variant of the original B-tree in which all records are stored in the leaves and all leaves are linked with the other ones. The B+-tree is very useful in the Relational Database Management Systems.

B+-tree considers all the keys in nodes except the leaves as dummies. All keys are duplicated in the leaves. This has the advantage that is all the leaves are linked together sequentially, the entire tree may be scanned without visiting the higher nodes at all.

STRUCTURE OF A B+ TREE AND SOME IMP. TERMS

• A B + -Tree consists of one or more blocks of data, called *node*s, linked together by pointers. The B + -Tree is a tree structure. The tree has a single node at the top, called the *root nod*e. The root node points to two or more blocks , called *child node*s. Each child nodes points to further child nodes and so on.

• The B + -Tree consists of two types of (1) *internal nodes* and (2) *leaf node*s:

(1)Internal Nodes

An **internal node** in a B + -Tree consists of a set of **key values** and **pointers.**
The set of keys and values are ordered so that a pointer is followed by a
key value. The last key value is followed by one pointer.

(2)Leaf Nodes

A **leaf node** in a B + -Tree consists of a set of *key values*and **data pointers.**
Each key value has one data pointer. The key values and data pointers are ordered by the key values.

• Internal nodes point to other nodes in the tree.

• Leaf nodes point to data in the database using *data pointer*s. Leaf nodes also contain an additional pointer, called the *sibling pointe*r, which is used to improve the efficiency of certain types of search.

• All the nodes in a B + -Tree must be at least half full except the root node
which may contain a minimum of two entries. The algorithms that allow
data to be inserted into and deleted from a B + -Tree guarantee that each
node in the tree will be at least half full.

• Both internal and leaf nodes contain *key values* that are used to guide the
search for entries in the index.

• The B + -Tree is called a *balanced tree* because every path from the root
node to a leaf node is the same length. A balanced tree means that all searches for individual values require the same number of nodes to be
read from the disc.

Order of a B + -Tree

The *order* of a B + -Tree is the number of keys and pointers that an internal
node can contain. An order size of *m* means that an internal node can contain *m-1* keys and *m* pointers.

PSEUDOCODE FOR SEARCH

```
s = Key value to be found
n = Root node
o = Order of B+-Tree
WHILE n is not a leaf node
    i = 1
    found = FALSE
    WHILE i <= (o-1) AND NOT found
        IF s <= nk[i] THEN
            n = np[i]
            found = TRUE
        ELSE
            i = i + 1
        END
    END
    IF NOT found THEN
```

```
        n = np[i]
    END
END
```

PSEUDOCODE FOR INSERT

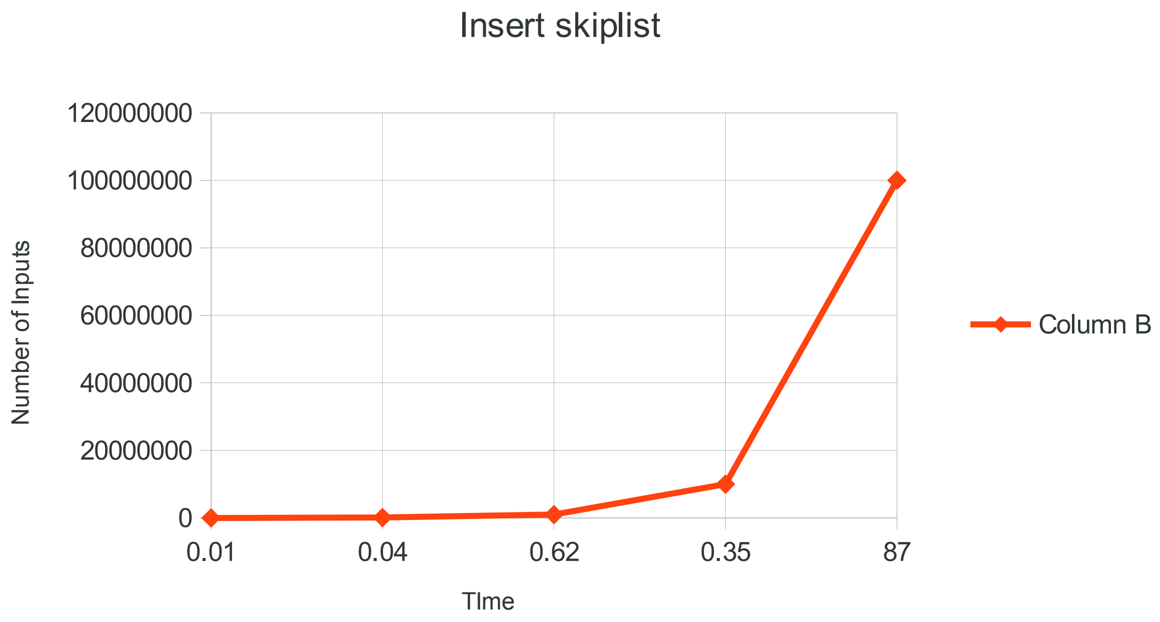
Insert Algorithm

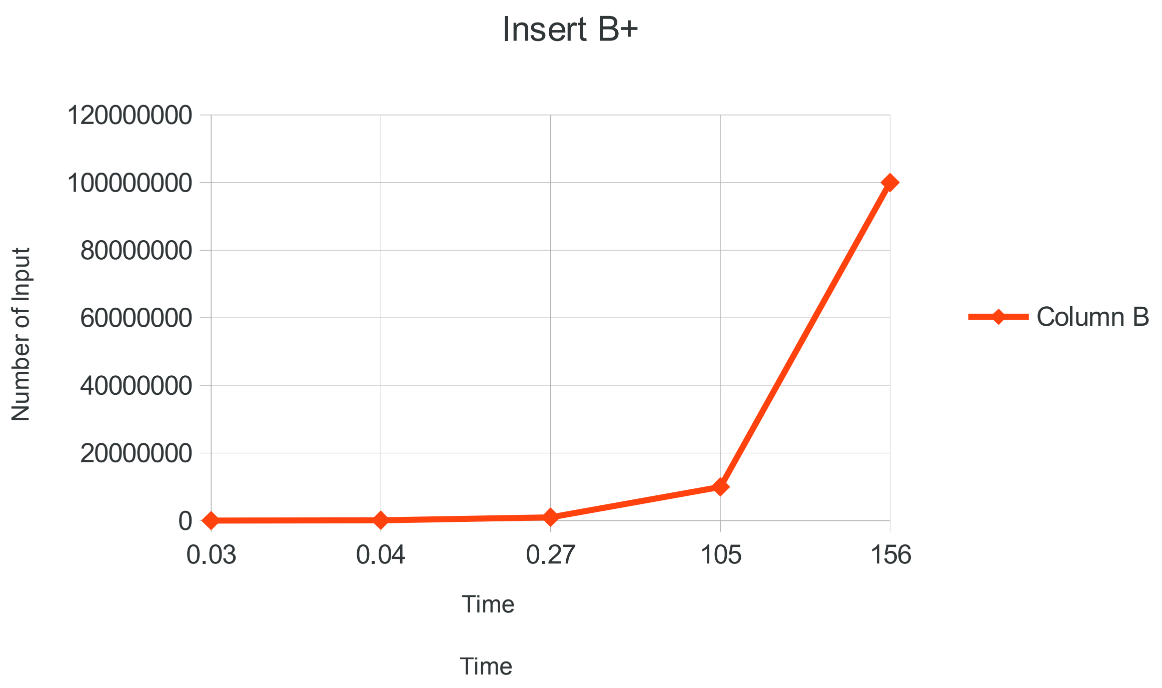
```
s = Key value to be inserted
Search tree for node n containing key s with path in stack p
from root(bottom) to parent of node n(top).
IF found THEN
    STOP
ELSE
    IF n is not full THEN
        Insert s into n
    ELSE
        Insert s in n (* assume n can hold s temporarily *)
        j = number of keys in n / 2
        Split n to give n and n1
        Put first j keys from n in n
        Put remaining keys from n in n1
        (k,p) = (nk[j],"pointer to n1")
    REPEAT
    IF p is empty THEN
        Create internal node n2
        Put (k,p) in n2
        finished = TRUE
    ELSE
        n = POP p
    IF n is not full THEN
```

```
            Put (k,p) in n
            finished = TRUE
        ELSE
            j = number of keys in n / 2
            Split n into n and n1
            Put first j keys and pointers in n into n
            Put remaining keys and pointers in n into (k,p) =
(nk[j],"pointer to n1")
        END
     END
  UNTIL finished
END
```

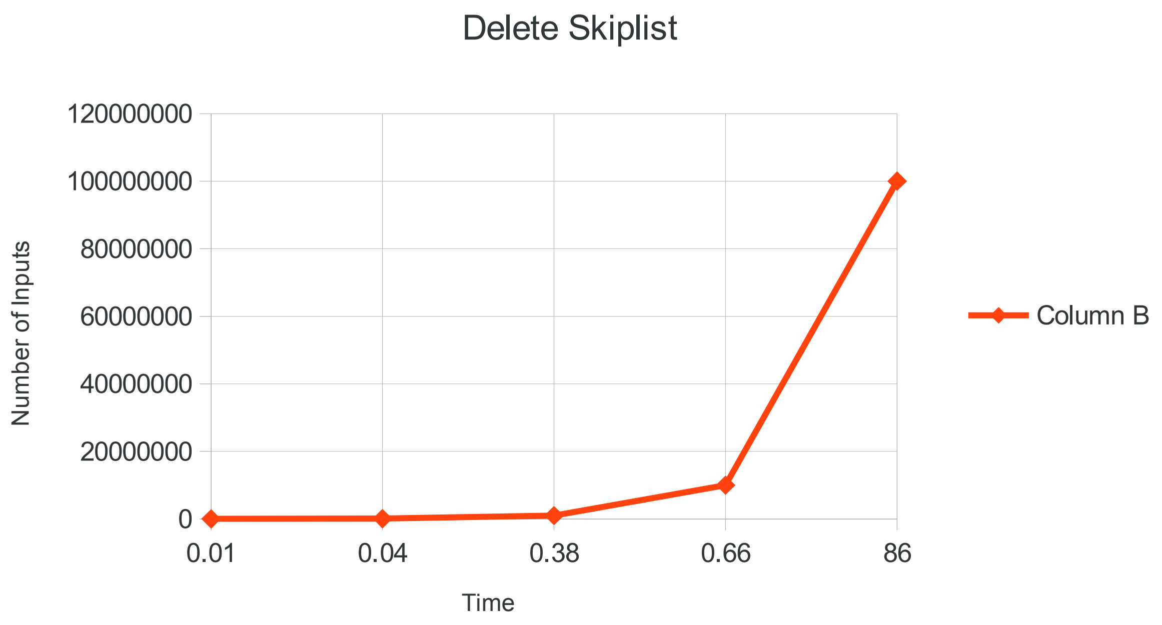
COMPARISON

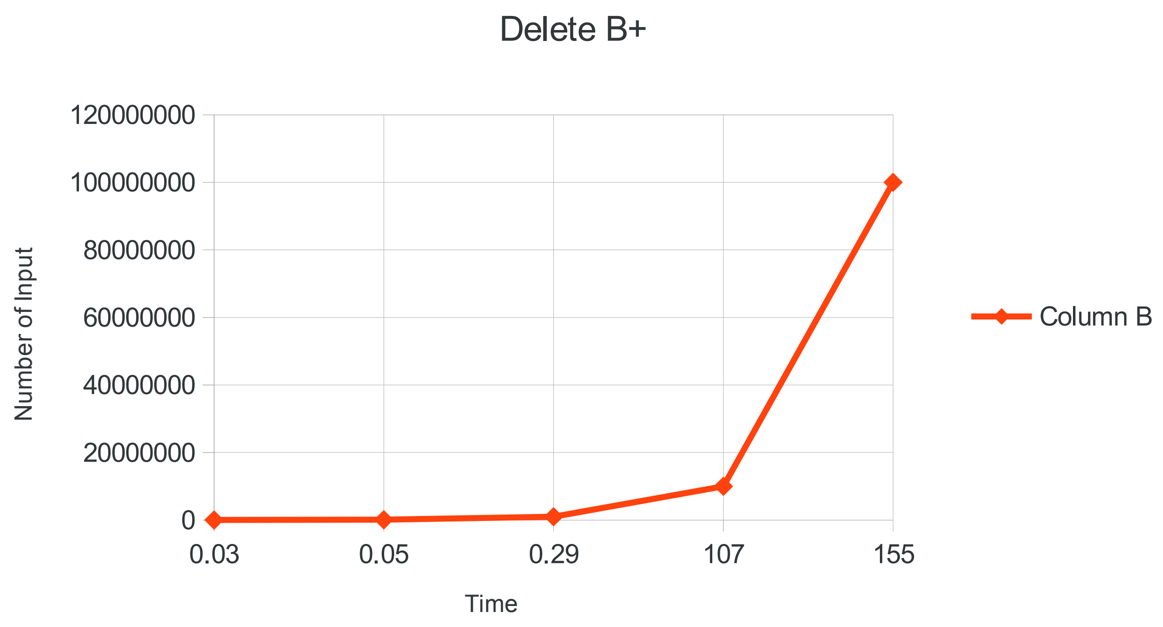
INSERT (Skip List v/s B+ Tree)



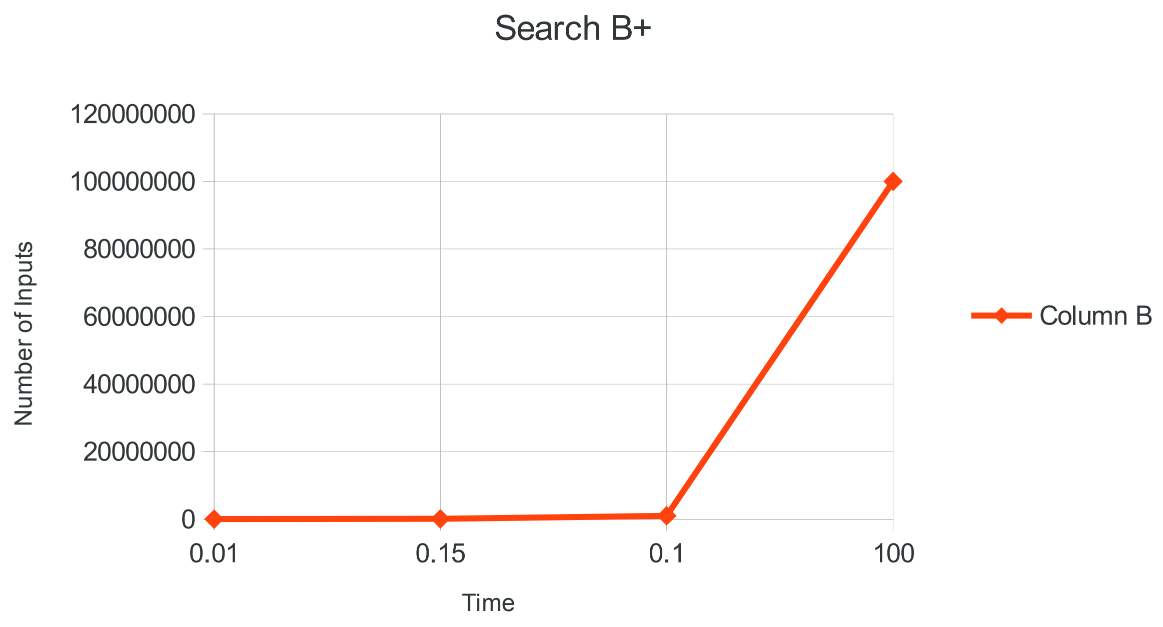

Insert B+
120000000
100000000
80000000
60000000
40000000
20000000
0
Number of Input
0.03
0.04
0.27
105
156
Time
Time
Column B

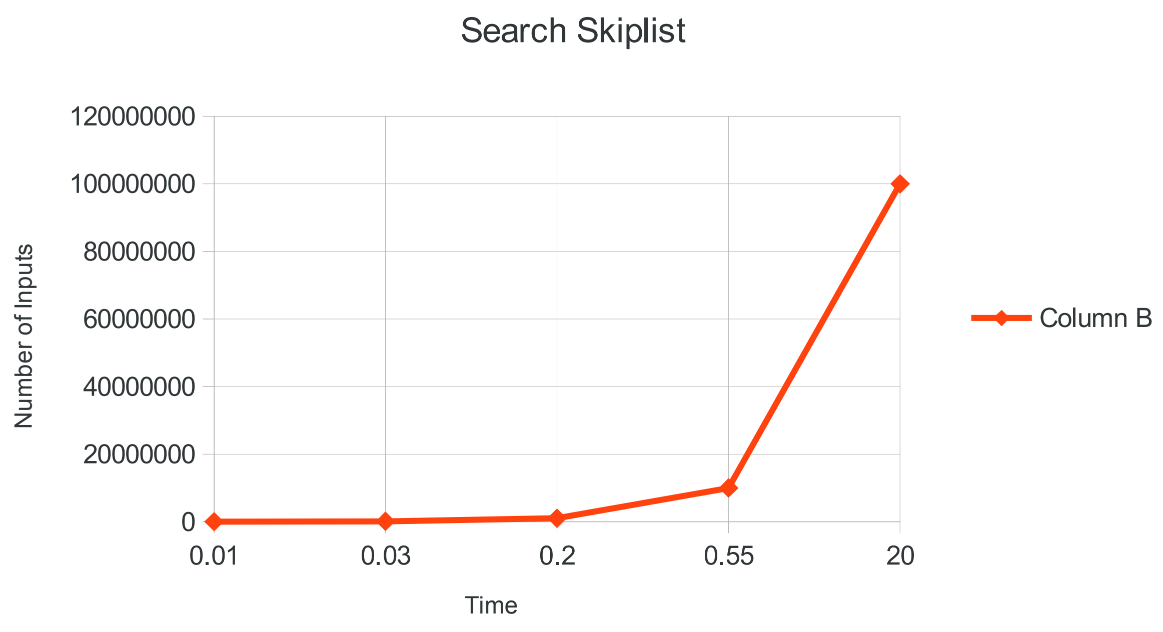
DELETE


Delete Skiplist
120000000
100000000
80000000
60000000
40000000
20000000
0
Number of Inputs
0.01
0.04
0.38
0.66
86
Time
Column B


Delete B+
Number of Input
120000000
100000000
80000000
60000000
40000000
20000000
0
0.03
0.05
0.29
107
155
Time
Column B

SEARCH


Search B+
Number of Inputs
120000000
100000000
80000000
60000000
40000000
20000000
0
0.01
0.15
0.1
100
Time
Column B



Time complexity of insert for B+ trees is TlogN(base T) ,where T is the time needed for linear seach in the node.

Time complexity for Skip lists if log N to the base 1/p where p is the probability (for insert and search).

Thus we see that skip lists have less time complexity than B+trees.

Skip Lists are thus a better way to implement (somewhat) search tree type of structures but B+trees are very important for reading data from the disk.

Thus we conclude that both skip lists and +trees are useful in their own ways but basic functionalities of skip lists are faster than B+trees.