DATA MANAGEMENT BY FILTERS AND TREES

Advanced Data Structures and Algorithms

This report includes the descriptions, operations, applications and comparisons of four data structures in the context of data mangement.

GROUP 16

106119014 – Anurag Goyal

106119058 – Kartikey Agarwal

106119046 – Hari Hara Sudhan

DATA MANAGEMENT BY FILTERS AND TREES

Advanced Data Structures and Algorithms

1. INTRODUCTION

- We have used two categories of data structures and implemented their use for the purpose of data management.
    - Probabilistic Filters
        - Bloom Filter
        - Cuckoo Filter
            - Low load factor
            - High load factor
    - Multiway Trees
        - B Tree (RAM based)
        - B+ Tree (both RAM based and Disk based)

    (All implementations are done in C++ for speed benefit)

- Probabilistic filters have been employed to determine the existence of an element in a set with high memory and time efficiency.

  Such filters provide a way to know about the existence of an element in a set but with an uncertainty in case of affirmative results.
- Multiway trees have been employed to maintain data in a structured manner. There are two implementations which store data in main memory and secondary memory respectively.

  They provide the benefit of reduced disk operations as compared to conventional binary trees.

PROBABILISTIC FILTERS

2. BLOOM FILTER

2.1 DESCRIPTION

- A Bloom filter is a memory efficient probabilistic data structure. It is used for determining the existence of an element in a set.
- It can only tell whether an element is "strictly not" or "maybe there" in the set.
- It maintains a bit array , i.e, an array of boolean type of a predetermined size.
- Element insertion is done by using a hash function to determine the locations of interest in the array.
- Searching in a Bloom filter is faster than most searching algorithms because of constant time taken by the hashing operation.
- The boolean values corresponding to the hash determined indexes are set to True.
- The false positive probability in determining set membership depends on the size of the bit array and the number of elements that have been inserted.
- The more elements are added, the greater the probability of false positives becomes.
- Its memory efficiency primarily comes from the fact that it doesn't need to store the actual elements.

2.2 REAL LIFE APPLICATIONS

- **Distributed Storage Systems**
    - Such systems organize their data into a number of tables that reside on the disk and are structured as key-value maps.
    - When users query for data, the problem is to locate which table contains the desired data.
    - For locating the right table without checking explicitly on the disk, a Bloom filter is maintained in the RAM for each table.
    - This way, each query triggers a lookup for each of the Bloom filters and when a filter reports the query data as present, we check in the table corresponding to that filter.
    - If it is a false alarm (the probability of which is kept very low by design) then we continue checking until another filter report the query data as present.

- **Web Caches**
    - Web caches exist around the globe to cache and serve web content with greater performance and reliability.
    - A majority of web content accessed around the globe is accessed only once and never again for a long time. Such content is not to be cached.
    - Bloom filters are used to determine which web content is to be cached. They are used to keep track of all the URLs accessed by users.
    - A web object is cached only when it is being accessed a second time.
    - The result is a significant reduction in disk write workload and saving of cache space which increases the cache hit rates.

2.3 LIMITATIONS

- The major limitation is that the existing elements can't be removed without rebuilding the entire filter
- Despite being memory efficient, Bloom filter come with an uncertainity built into them when giving positive results.
- This false positive probability increases as more elements are inserted, so care must be taken to not overload the filter because a Bloom filter can accommodate even an infinite number of insertions.
- Choice of hashing function matters. The hashing function must be able to convert the concerned type of element into a positive integer.
- A poor choice of hashing function would result in a greater chance of hash collisions. These collisions also produce false positives.
- The hashing function may also need performace optimizations based on the type of hardware it is being run on.

2.4 OPERATIONS

By definition, Bloom filters support only two kinds of operations on them :

1. Element insertion
2. Element lookup

Deletion of elements is not a wise choice because there is a reasonable chance that it can wrongly remove the trace of existence of multiple other elements from the filter.

PREREQUISITES

Before doing any operation, we need to establish K number of hash functions which will be used to generate K number of hashes which will act as indices to the bit array.

The optimal number of hash functions is calculated in terms of the size of bit array (M) and the rough upper limit of the number of elements to be inserted (N).

$$K = ceil\left(\frac{M}{N}\ln 2\right)$$

We can either use K different hash functions or the same hash function with K different seed values.

The false positive probability can be calculated in terms of the size of bit array (M) and the number of elements that have been inserted (C).

$$P = 2^{-\frac{M}{C}\ln 2}$$

2.4.1 ELEMENT INSERTION : O(1)

Let the element to be inserted be X.

```
for i ranging from 0 to K :
     hash = hash_function(X, i) % M
     bit_array[hash] = true
```

Example : M = 10, K = 3

X = “data” => hashes = {7, 0, 6}

1	0	0	0	0	0	1	1	0	0

X = “structure” => hashes = {6, 8, 1}

1	1	0	0	0	0	1	1	1	0

2.4.2 ELEMENT LOOKUP : O(1)

Let the element to be looked up be X.

```
for i ranging from 0 to K :
     hash = hash_function(X, i) % M
     if (bit_array[hash] == false):
          return false
return true
```

Example: M = 10, K = 3

X = “structure” => hashes = {6, 8, 1} **PRESENT**

1	1	0	0	0	0	1	1	1	0

X = “hello” => hashes = {8, 3, 6} **ABSENT**

1	1	0	0	0	0	1	1	1	0

X = “hearang” => hashes = {1, 0, 7} **FALSE POSITIVE**

1	1	0	0	0	0	1	1	1	0

3. CUCKOO FILTER

3.1 DESCRIPTION

- A Cuckoo filter is also a memory efficient probabilistic data structure like Bloom filter. It is used for determining the existence of an element in a set.
- It can only tell whether an element is "strictly not" or "maybe there" in the set.
- It maintains multiple arrays of the same size called as 'buckets'.
- Element insertion is done by using a hash function to determine the locations of interest in a bucket by allowing open addressing.
- Amortized searching time in a Cuckoo filter is as fast as Bloom filter.
- The false positive probability in determining set membership depends only on the choice of fingerprinting technique.
- The more elements are added, the greater the probability of false positives becomes.
- Its memory efficiency primarily comes from the fact that it doesn't need to store the actual elements, instead it stores the fingerprints of those elements.
- We have implemented two variations of Cuckoo filters :
    - Low load factor Cuckoo filter
    - High load factor Cuckoo filter

3.2 REAL LIFE APPLICATIONS

- **Remote REST Services**

  Cuckoo filters are employed in remote REST API services which have a cost per API call made.

  Any API call with a binary answer such as “this address is invalid” uses a cuckoo filter and eliminates over 90% of duplicate queries.

- **Firewalls in Virtual Network Functions**

  A firewall needs to maintain a record of rules to be applied on a set of IP addresses.

  Cuckoo filters allow writing of rule sets in the form of hash tables that can store an extensive collection of rule sets and effective match network packets with those rules.

  The hash table in Cuckoo filter can be trained for the required parameters and the parameters in the incoming packets can be match with high speed and accuracy.

- **Loop Detection in Software Defined Networking**

  IP routing for multicast connection requires solving an NP-Hard graph matching problem.

  The underlying graph hosting the mutipath connections has to be analyzed for possible paths and forward loops.

  Cuckoo filters have a unique property that detects if the filter is full and relocated keys. If the keys get displaced for the entire round that means a forwarding loop has been detected efficiently.

3.3 LIMITATIONS

- Unlike Bloom filters, Cuckoo filters allow deletion of elements. But deletion can only be done for elements which are known to have been inserted.
- If the fingerprints for two elements collide then deleting the wrong element will end up removing the originally inserted element from the filter.
- Insertion can sometimes trigger a fingerprint relocation chain which results in greater time needed for insertion, but the amortized insertion complexity is still remains to be O(1).
- If the relocation chain reaches a predetermined relocation threshold then insertion is considered as a failure and the entire cuckoo hash table needs rehashing.
- Choice of the hashing and fingerprinting functions matters because the probability of getting a false positive depends on how prone the fingerprinting function is in producing collisions.
- In the low load factor implementation of Cuckoo filter, a certain step (XOR) in insertion operation is prone to produce hashes which overflow the hash table range limit. To prevent this, one must choose the hashing function carefully depending on the type of data being processed by the filter. That is why the ratio of number of elements to the size of buckets
- In the high load factor implementation of Cuckoo filter, the lookup and deletion operations can also trigger a fingerprint relocation chain just like the insertion operation, but their amortized time complexity still remains to be O(1).

3.4 OPERATIONS

The following operations can be performed on Cuckoo filters :

1. Element insertion (low and high load)
2. Element lookup (low load)
3. Element deletion (low load)
4. Element lookup (high load)
5. Element deletion (high load)

PREREQUISITES

Number of buckets in the filter	:	2
Size of each bucket	:	M
Hashing function	:	H
Fingerprinting function	:	F
Relocation threshold	:	T
Element to be operated upon	:	X

3.4.1 ELEMENT INSERTION (LOW AND HIGH LOAD) : O(1) AMORTIZED

- Calculate the fingerprint and hash of X as :
  - `fingerprint = F(X)`
  - `hash = H(X) % M`
- If bucket[0][hash] is empty then store the fingerprint there.
- Otherwise go to the alternative location in the next bucket :
  - `alt_hash = hash XOR (H(fingerprint) % M)` (Low Load -> overflow risk)
  - `alt_hash = (hash XOR H(fingerprint)) % M` (High Load -> no such risk)
- If that location is also occupied then randomly choose any location among **hash** and **alt_hash** and relocate the fingerprint present there to its alternative location calculated by the above method.
- Keep doing this until an empty location is reached in either of the buckets but halt the process once relocation has happened more than T times.

Example : M = 10

X = “cuckoo”	=> hash = 0 ;	fingerprint = 93497
X = “filter”	=> hash = 1 ;	fingerprint = 98649
X = “data”	=> hash = 2 ;	fingerprint = 16542

93497	93497	16542	0	0
0	0	0	0	0

X = “structure”	=> hash = 1 ;	fingerprint = 26375

Now, 93497 is relocated to index 1 XOR (H(94397) % 5) = 0 in second bucket

93497	26375	16542	0	0
98649	0	0	0	0

3.4.2 ELEMENT LOOKUP (LOW LOAD) : O(1)

```
fingerprint = F(X)
hash = H(X) % M

if (bucket[0][hash] == fingerprint):
    return true
else :
    alt_hash = hash XOR(H(fingerprint) % M)
    if (bucket[1][alt_hash] == fp):
          return true
return false
```

Example : M = 5

X = “filter” => hash = 1 ; fingerprint = 98649 **PRESENT**

93497	26375	16542	0	0
98649	0	0	0	0

alternative hash = 1 XOR (H(98649) % 5) = 0 in the second bucket

X = “hello” => hash = 3 ; fingerprint = 93332 **ABSENT**

93497	26375	16542	0	0
98649	0	0	0	0

alternative hash = 3 XOR (H(93332) % 5) = 2 in the second bucket

3.4.3 ELEMENT DELETION (LOW LOAD FACTOR) : O(1)

Deletion is similar to searching, we just have to set the array value to 0 if found.

```
fingerprint = F(X)
hash = H(X) % M

if (bucket[0][hash] == fingerprint):
    bucket[0][hash] = 0
    return true
else :
    alt_hash = hash XOR (H(fingerprint) % M)
    if (bucket[1][alt_hash] == fp):
         bucket[1][alt_hash] = 0
         return true
return false
```

Example : M = 5

X = "lemming" => hash = 2 ; fingerprint = 14405 **NOT DELETED**

93497	26375	16542	0	0
98649	0	0	0	0

alternative hash = 2 XOR (H(98649) % 5) = 3 in the second bucket

X = "filter" => hash = 1 ; fingerprint = 98649 **DELETED**

93497	26375	16542	0	0
98649 → 0	0	0	0	0

alternative hash = 1 XOR (H(98649) % 5) = 0 in the second bucket

3.4.4 ELEMENT LOOKUP (HIGH LOAD) : O(1) AMORTIZED

- If bucket[0][hash] == fingerprint then return true.
- Else calculate the alternative hash as :
    - `alt_hash = (hash XOR H(fingerprint)) % M`
- If bucket[1][alt_hash] == fingerprint then return true.
- Else calculate a new alt_hash in the above way and start a recursive call by searching in the first bucket again.
- Keep repeating this process until the target fingerprint is found or the relocation threshold is reached and return false if threshold is reached.

Example : M = 5

93497	26375	16542	89461	10199
98649	37423	62138	82915	48134

X = “roncsdcw” => hash = 1 ; fingerprint = 10199 **PRESENT**

93497	26375	16542	89461	10199
98649	37423	62138	82915	48134

alternative hash = (1 XOR H(26375)) % 5 = 3 in the second bucket

93497	26375	16542	89461	10199
98649	37423	62138	82915	48134

alternative hash = (3 XOR H(82915)) % 5 = 4 in the first bucket

3.4.5 ELEMENT DELETION (HIGH LOAD) : O(1) AMORTIZED

- If bucket[0][hash] == fingerprint then set that value to 0 and return true.
- Else calculate the alternative hash as :
    - `alt_hash = (hash XOR H(fingerprint)) % M`
- If bucket[1][alt_hash] == fingerprint then set that value to 0 and return true.
- Else calculate a new alt_hash in the above way and start a recursive call by searching in the first bucket again.
- Keep repeating this process until the target fingerprint is found or the relocation threshold is reached and return false if threshold is reached.

Example : M = 5

93497	26375	16542	89461	10199
98649	37423	62138	82915	48134

X = "vanhelsing" => hash = 2 ; fingerprint = 37423 **DELETED**

93497	26375	16542	89461	10199
98649	37423	62138	82915	48134

alternative hash = (2 XOR H(16542)) % 5 = 4 in the second bucket

93497	26375	16542	89461	10199
98649	37423	62138	82915	48134

alternative hash = (4 XOR H(48134)) % 5 = 0 in the first bucket
alternative hash = (1 XOR H(93497)) % 5 = 1 in the second bucket

MULTIWAY TREES

4. B TREE

4.1 DESCRIPTION

B Tree is a self-balancing search tree. In most of the other self-balancing search trees like AVL and Red-Black trees, the assumption is that all data resides in the main memory. To understand the use of B Trees, we must think of the huge amount of data that cannot fit in the main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to the main memory access time. The main idea of using B Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min … etc.) require O(h) number of disk accesses where h is the height of the tree. The height of B Tree is kept low by putting maximum possible keys in a B Tree node. Generally, the B Tree node size is kept equal to the disk block size. Since the height of B Tree is low so total disk accesses for operations are reduced significantly compared to balanced binary search trees like AVL and Red-Black trees.

Properties of B Tree:

- All leaf nodes are at the same level.
- A B Tree is defined by the term minimum degree 't' whose value depends on the disk block size.
- Every node except the root node must contain at least $ceil\left(\frac{t-1}{2}\right)$ keys. The root node can contain a minimum of 1 key.
- All nodes (including the root node) can contain no more than $(2t-1)$ keys.
- Number of children of a node is 1 more than the number of keys in it.
- All keys of a node are sorted in increasing order. The child node between keys k1 and k2 contains all keys in the range (k1, k2).
- B Tree grows and shrinks from the root unlike BST.
- Like other balanced binary search trees, time complexity of insertion, searching and deletion in a B Tree is also O(log n).

4.2 REAL LIFE APPLICATIONS

- **Filesystems**

  B Tree is used in filesystems to allow quick random access to an arbitary block in a particular file.

  B Tree algorithms are efficient for accessing blocks of stored information which is then copied into main memory for processing. In the worst case, they are designed to do dynamic set operations in O(log N) time because of their high 'branching factor'.

  This branching factor makes B Trees so efficient for block accessing, since a large branching factor greatly reduces the height of the tree and thus the number of disk accesses needed to find any key.

- **Database Management Systems**

  Large databases are kept on the disk drives. The time taken to read a record on a disk drive far exceeds the time needed to compare keys once the record is available.

  B Tree is used to index the data and provides faster access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process.

  Searching an un-indexed and unsorted database containing N key values needs O(N) running time in worst case. However, if we use B Tree to index this database then it will be searched in O(log N) time in worst case.

4.3 LIMITATIONS

- B Tree searching is more complicated than linear searching and is not efficient for a small number of elements.
- It works properly only on lists that are sorted and are kept sorted. That is not always feasible especially if elements are constantly being added to the list.
- B Trees work only on those types of elements for which it is possible to establish a less-than type of relationship.
- There is a great loss of efficiency if the list does not support random access. If you need to immediately jump to the middle of the list, i.e, simple arrays work prove to be better than linked lists.
- Depending on the cost of the comparison operation, the cost of traversing a non-random-access list can reduce the cost of those comparisons.
- There are even faster search methods available, such as hash lookups. However, a hash lookup requires the elements to be organized in a much more complicated data structure (hash tables).

4.4 OPERATIONS

We can perform three kinds of operations on B Trees :

1. Element insertion
2. Element searching
3. Element deletion

4.4.1 ELEMENT INSERTION : O(log N)

A new key is always inserted at the leaf node. Let the key to be inserted be K. Like BST we start from the root and traverse down till we reach a leaf node. Once we reach a leaf node, we insert the key in that leaf node. Unlike BST, we have a predefined range on the number of keys that a node can contain. So before inserting a key to the node, we make sure that the node has extra space.

If a leaf node is full then we split it into two nodes to make space and a key is moved up into the parent node and that is why B Trees grow upwards rather than downwards.

A) Child y of parent node x is split into y and x

```
b_tree_split_child (x, i):
  z = allocate_node()
  y = x.child[i]
  z.leaf = y.leaf
  z.n = t - 1
  for j = 1 to (t - 1):
      z.key[j] = y.key[j + t]
  if not y.leaf:
      for j = 1 to t:
           z.child[j] = y.child[j + t]
  y.n = t - 1
  for j = x.n + 1 downto i + 1:
      x.child[j + 1] = x.child[j]
```

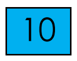
```
    x.child[i + 1] = z
  for j = x.n downto i:
    x.key[j + 1] = x.key[t]
  x.key[i] = y.key[t]
  x.n += 1
  disk_write(y)
  disk_write(z)
  disk_write(x)
```

B)
```
b_tree_insert_non_full (x, k):
  i = x.n
  if x.leaf:
      while i >= 1 and k < x.key[i]:
           x.key[i + 1] = x.key[i]
           i -= 1
      x.key[i + 1] = k
      x.n += 1
      disk_write(x)
  else:
      while i >= 1 and k < x.key[i]:
           i -= 1
      i += 1
      disk_read(x.child[i])
      if x.child[i].n == 2*t  - 1:
           b_tree_split_child(x, i)
           if k > x.key[i]:
                i += 1
      b_tree_insert_nonfull(x.child[i], k)
```

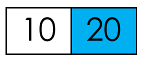
C)
```
b_tree_insert(T, k):
  r = T.root
  if r.n == 2*t - 1:
      s = allocate_node()
      T.root = s
      s.leaf = false
      s.n = 0
      s.c[1] = r
      b_tree_split_child(s, 1)
      b_tree_insert_non_full(s, k)
  else:
      b_tree_insert_non_full(r, k)
```

Example : Insert 10, 20, 30, 40, 50 and 60 in a B Tree of min. degree 3

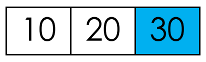
- Insert 10



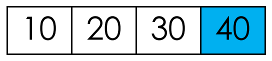
- Insert 20



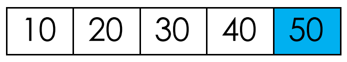
- Insert 30



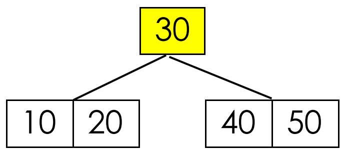
- Insert 40



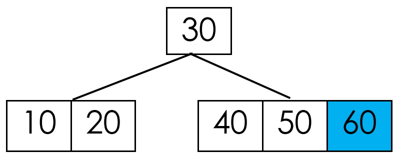
- Insert 50



- Insert 60



(node is splitted)

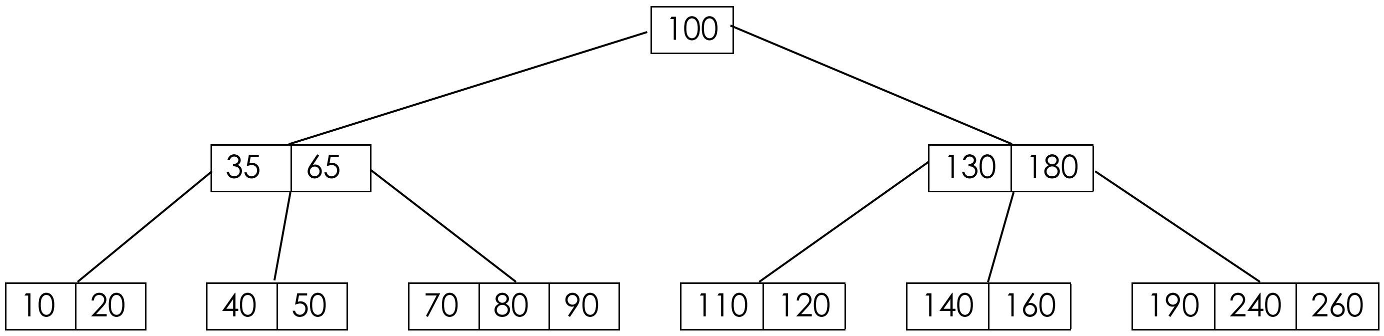


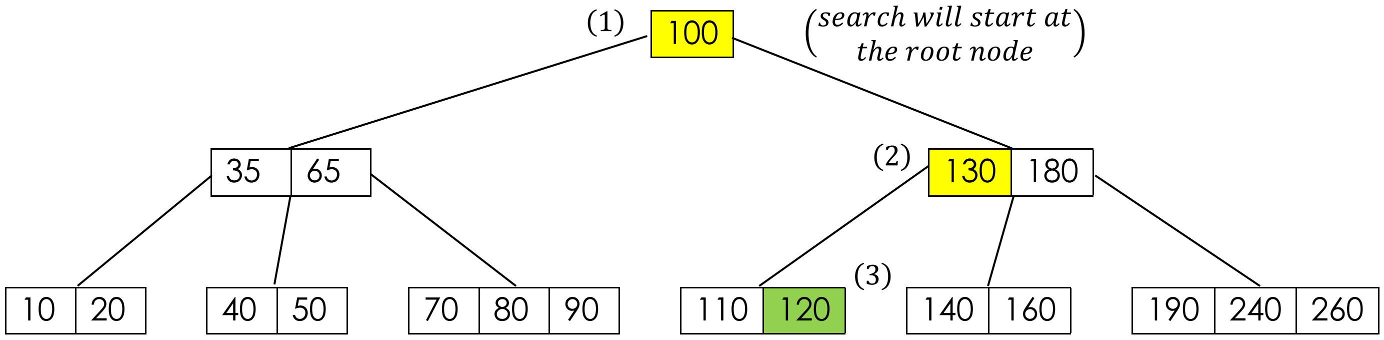
4.4.2 ELEMENT SEARCHING : O(log N)

Searching in a B Tree is similar to searching in BST. Let the key to be searched be K. We start from the root and rercursively traverse down. For every visited non-leaf node, if the node has the key, we simply return the node. Otherwise, we recur down to the appropriate child node which is just before the first greater key. If we reach a leaf node and don't find K in the leaf node, we return NULL.

```
b_tree_search(x, k):
       i = 1
       while i <= x.n and k > x.key[i]:
           i += 1
       if i <= x.n and k == x.key[i]:
           return (x, i)
       else if x.leaf:
           return NULL
       else:
           disk_read(x.child[i])
           return b_tree_search(x.child[i], k)
```

Example : Search for 120 in the following B Tree.





4.4.3 DELETION : O(log N)

Deletion is more complicated than insertion because we can delete a key from any node (not just leaf) and when we delete a key from an internal node, we have to rearrange that node's children.

As in insertion, we must make sure that deletion does not violate the B Tree properties. Just as we had to ensure that a node doesn't get too big due to insertion, we must ensure that a node doesn't get too small during deletion (except the root node).

A simple approach to deletion might have to back up if a node along the path has the minimum number of keys. This guarantees that whenever the deletion function calls itself recursively on a node X, the number of keys in X is at least the minimum degree t. This condition requires one more key than the minimum required by the usual B Tree conditions, so that sometimes a key may have to be moved into a child node before recursion descends into that child.

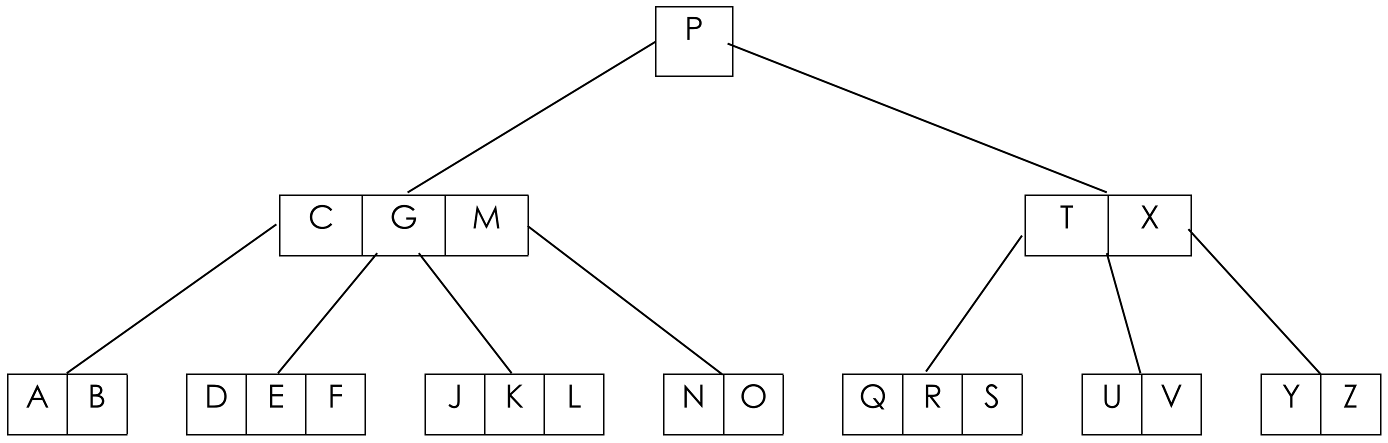
This strengthened condition allows us to delete a key from the tree in one downward pass without having to back up. If the root node X ever becomes an internal node having no keys then we delete X and X's only child becomes the new root of the tree, decreasing the height of the tree by one level and preserving the property that the root node must contain at least one key.

1. If the key is in node X and X is a leaf node then delete the key from X.
2. If the key K is in node X and X is an internal node then do the following :
    a. If the child Y that precedes K in node X has at least t keys, then find the predecessor K_0 of K in the sub-tree rooted at Y. Recursively delete K_0 and replace K by K_0 in X.
    b. If Y has fewer than t keys then symmetrically examine the child Z that follows K in node X. If Z has at least t keys then find the successor K_0 of K in the tree subrooted at Z. Recursively delete K_0 and replace K by K_0 in X.
    c. Otherwise, if both Y and Z have only t keys, merge K and all of Z into Y, so that X loses both K and the pointer to Z and Y now contains $2t - 1$ keys. Now free Z and delete K from Y.

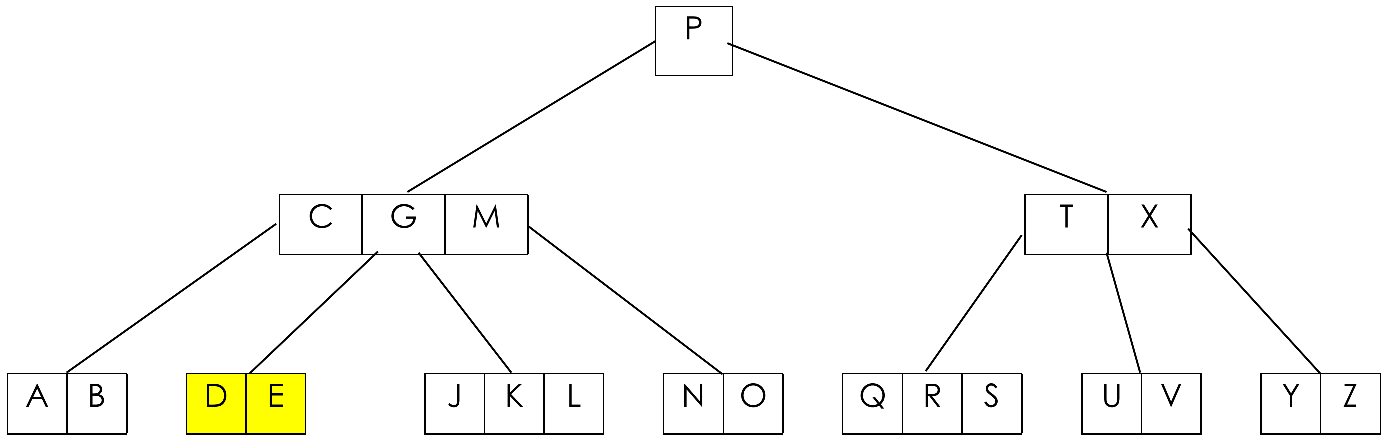
3. If the key K is not present in the internal node X, determine the root X.child[i] of the appropriate subtree that must contain K, if K is in the tree at all. If X.child[i] has only $t - 1$ keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Finally finish by recursing on the appropriate child of X.
   a. If X.child[i] has only $t - 1$ keys but has an immediate sibling with at least t keys, give X.child[i] an extra key by moving a key from X down into X.child[i], moving a key X.child[i]'s immediate left or right sibling up into X and moving the appropriate child pointer from the sibling into X.child[i].
   b. If X.child[i] and both of its immediate siblings have $t - 1$ keys then merge X.child[i] with one sibling which involves moving a key from X down into the new merged node to become the median key for that node.

Since most of the keys in a B Tree are in the leaves, deletion operations are most often used to delete keys from leaves. The recursive delete operation then acts in one downward pass through the tree, without having to back up. When deleting a key in an internal node, however, the procedure makes a downward pass through the tree but may have to return to the node from which the key was deleted to replace the key with its predecessor or successor.

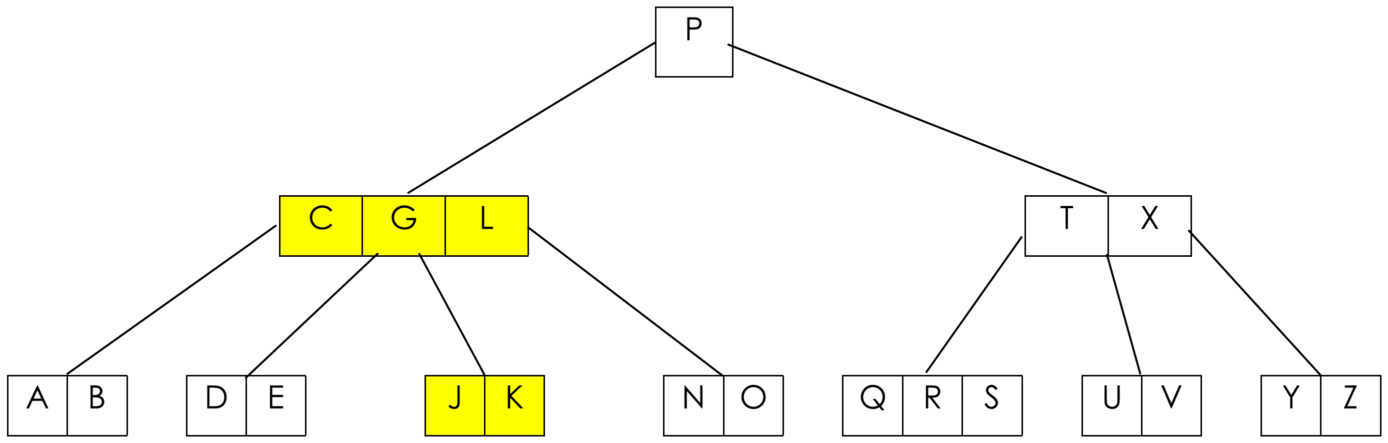
Example : Consider the given B Tree



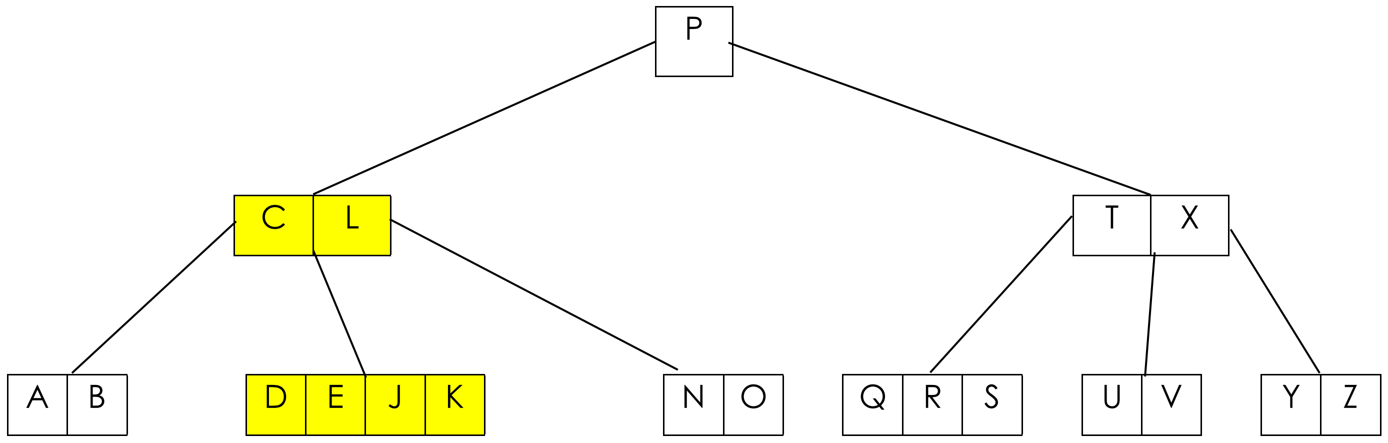
- Delete F



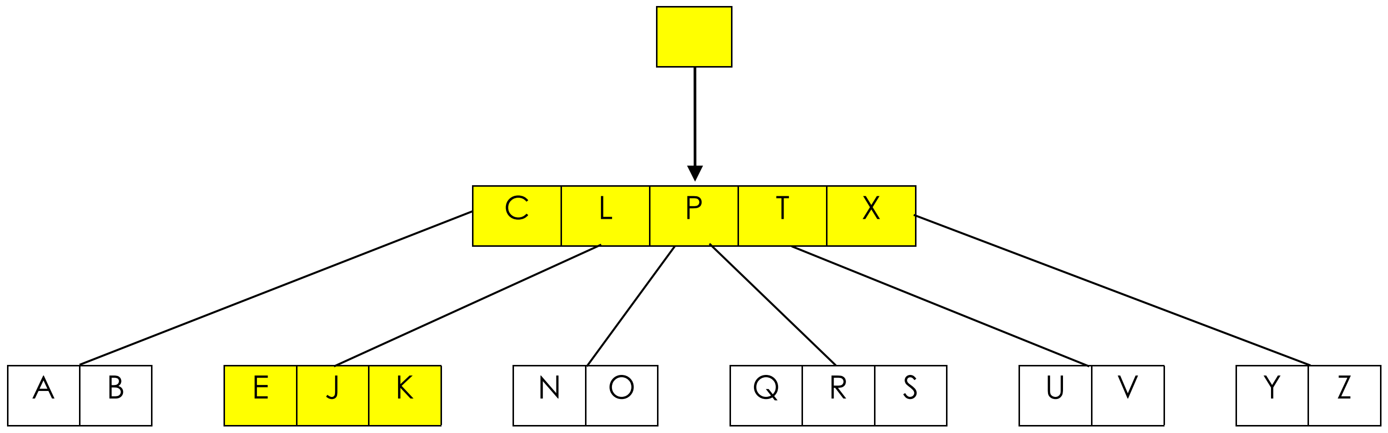
- Delete M

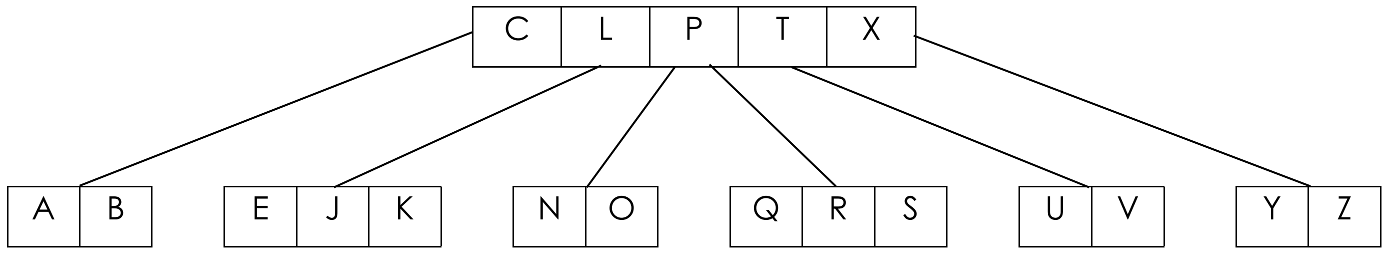


- Delete G

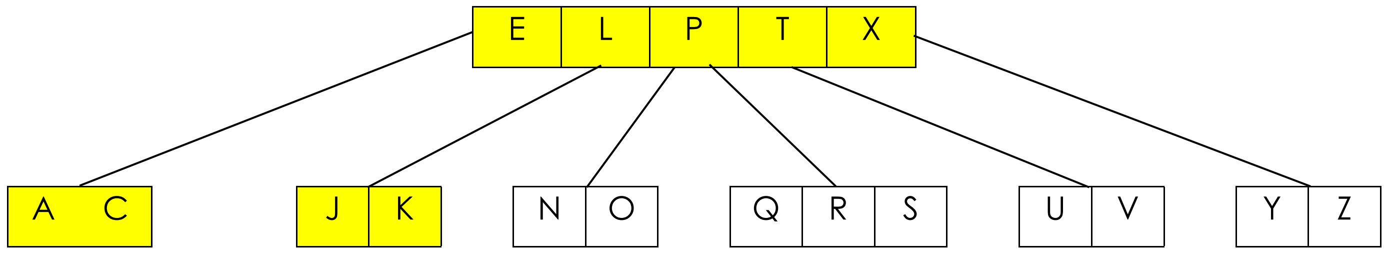


- Delete D





- Delete B



5. B+ TREE

5.1 DESCRIPTION

A B+ Tree is a m-ary tree with a variable but often large number of children per node. A B+ Tree consists of a root node, internal nodes and leaves. The root may be either a leaf node with two or more children.

A B+ Tree can be viewed as a B Tree in which each node contains only keys (not key-value pairs) and to which additional level is added at the bottom with linked leaves. The difference is that in B+ Trees only leaf nodes contain the actual key values. The non-leaf nodes of the B+ Tree contain router values. Routers are entities of the same type as the key values, but they are not the keys stored in the search structure. They are only used to guide the search in the tree. In classical B Trees, the key values are stored in both leaf and non-leaf nodes of the tree.

STRUCTURE

A B+ Tree can consists of nodes. One of the nodes is a special node, i.e, the root node. If a node X is a non-leaf node then it has the following fields :

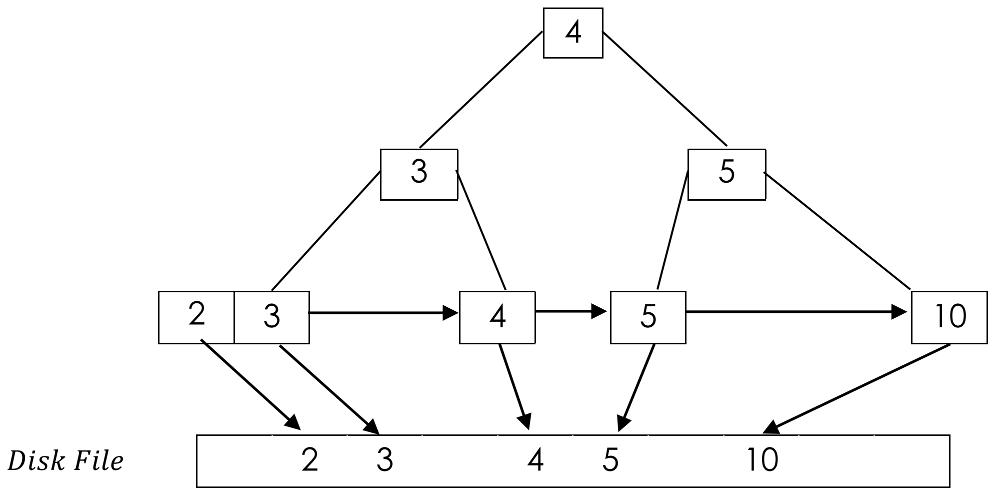
- X.n, the number of router values currently stored in the node X.
- The router values are stored in increasing order.
- X.leaf, a boolean field whose value tells whether the node is a leaf node.
- X.n + 1 pointers X.child[0] to X.child[X.n] to the child nodes of X.

If the node X is a leaf node then it has the following fields :

- X.n, the number of key values currently stored in the node X.
- The key values are stored in increasing order.
- X.leaf, a boolean filed whose value is true.

The order or branching factor, b of a B+ Tree measures the capacity of internal nodes. The actual number of children, m is constrained for internal nodes such that $\frac{b}{2} \leq m \leq b$. The root node is an exception and is allowed to have as few as two children.

Node Type	**Children Type**	**Minimum Children**	**Maximum Children**
Root Node (alone)	Records	0	$b - 1$
Root Node	Internal / Leaf Nodes	2	b
Internal Node	Internal / Leaf Nodes	$ceil\ (b/2)$	b
Leaf Node	Records	$ceil\ (b/2)$	b



5.2 REAL LIFE APPLICATIONS

- **Large Scale Data Storage**

  The primary value of a B+ Tree is in storing data for efficient retrieval in a block-oriented storage context, in particular, filesystems. This is primarily because unlike binary search trees, B+ Trees have very high fanout which reduces the number of I/O operations required to find an element in the tree.

- **Metadata Indexing**

  The ReiserFS, NSS, XFS, JFS, ReFS and BFS filesystems all use B+ Trees for metadata indexing. BFS uses B+ Trees for storing directories also. NTFS uses B+ Trees for directory and security-related metadata indexing. EXT4 uses extent trees (a modified form of B+ Tree) for file extent indexing. APFS uses B+ Trees to store mappings from filesystems object IDs to their locations on disk, and to store filesystem records.

  RDBMS systems such as IBM DB2, Informix, Microsoft SQL Server, Oracle 8, Sysbase ASE and SQLite support this type of tree for table indices. Key-value database management systems such as CouchDB and Tokyo Cabinet use B+ Trees for accessing data from the disk.

5.3 LIMITATIONS

- In indexed sequential files, performance degrades as the file's size grows, since many overflow blocks get created. Periodic reorganization of entire file is required.
- However, B+ Trees automatically reorganize themselves with small and local changes in the face of insertions and deletions.
- Reorganization of the entire file is not required to maintain performance. Due to this extra insertion and deletion overhead, space overhead is present but advantages of using B+ Trees outweigh these disadvantages and they are used extensively.
- B+ Trees also need extra memory overheads for insertion and deletion.

5.4 OPERATIONS

We can perform three kinds of operations on B+ Trees :

1. Element insertion
2. Element searching
3. Element deletion

PREREQUISITES

1. **Node Splitting**

   When number of search-key values exceeds n-1 :
   - If it is a leaf node :
     - Split into two nodes
     - 1st node contains $ceil\left(\frac{n-1}{2}\right)$ values.
     - 2nd node contains the remaining values.
     - Copy the smallest search-key value from the 2nd node to parent node.
   - If it is a non-leaf node :
     - Split into two nodes.
     - 1st node contains $\frac{n}{2} - 1$ values.
     - 2nd node contains the remaining values.

2. **Node Merging**

- Move all values and pointers to the left node.
- Remove the middle value element in the parent node.
- If non-leaf node :
  - Redistribute to sibling :
    - Through parent's values.
    - Right node has no less elements than left node.
  - Merge (if nodes contain too few entries) :
    - Bring the parent node down.
    - Move all values and pointers to the left node.
    - Delete the right node and delete the pointers in the parent node.

5.4.1 ELEMENT INSERTION : O(log N)

The insertion of element K into a B+ Tree is started by searching for the leaf node Y which is going to contain K. If there is enough space in Y to insert the key K, then simply insert K and no other actions are needed.

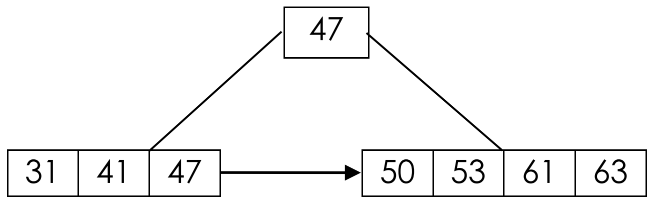
If the node Y is full before insertion then it must be split as follows $(t = b/2)$:

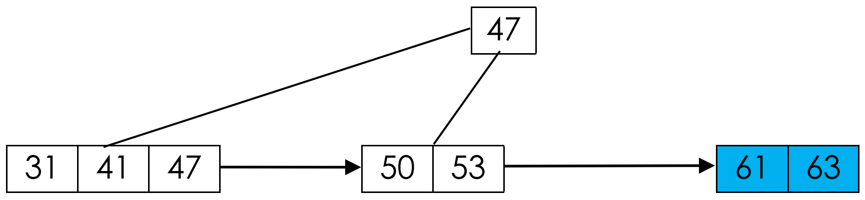
- Allocate memory for a new node Z.
- The first t keys of node Y are left in node Y.
- The last t keys if node Y are transferred to the new node Z.
- The key K is inserted to either node Y or node Z according to the value of K.
- A pointer to the new node Z and a new router value are inserted to node X which is the parent of both Y and Z. A good choice for the new router value is the last key value in Y.
- If there is not enough space for a new pointer and router value in node X then X must be split. In the worst case, all nodes in the path from leaf to the root must be split.

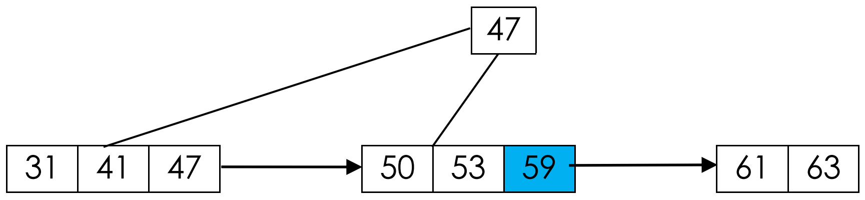
ALGORITHM :

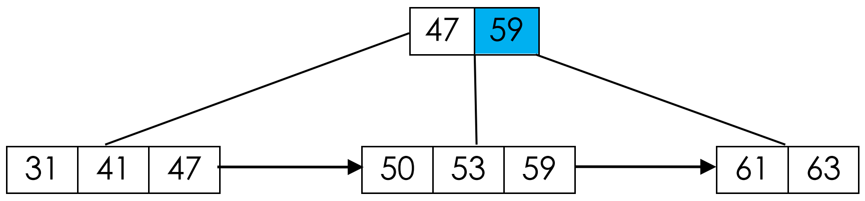
1. Insert the key in its leaf in sorted order.
2. If the leaf ends up with $m + 1$ items then split the leaf node.
    a. Split the leaf into two nodes :
        i. Original one with $(m + 1)/2$ items.
        ii. New one with $(m + 1)/2$ items.
    b. Add the new child to the parent node.
    c. If the parent ends up with $m + 1$ children then split the parent node.
3. If an internal node ends up with $m + 1$ children then split that node :
    a. Split the node into two nodes :
        i. Original one with $(m + 1)/2$ items.
        ii. New one with $(m + 1)/2$ items.
    b. Add the new child to the parent node.
    c. If the parent ends up with $m + 1$ children then split the parent node.
4. Split an overflowed root node in two and assign the new nodes as children of a new root node.
5. Propagate the keys up the tree.

EXAMPLE : Insert 59 into the given B+ Tree.









5.4.2 ELEMENT SEARCHING : O(log N)

The root of a B+ Tree represents the whole range of values in the tree, where every internal node is a sub-interval. We are looking for a value K in the tree. Starting from the root, we are looking for the leaf which may contain the value K. At each node, we figure out which internal pointer we should follow. An internal B+ Tree node has at most b children where each one of them represents a different sub-interval. We select the corresponding node by searching on the key values of the node.

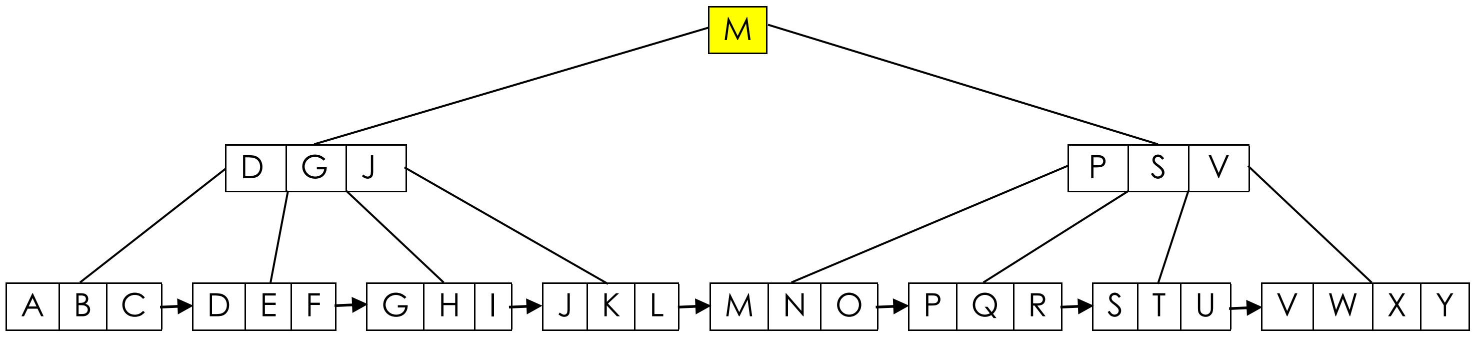
The search operation is started in the root node and it proceeds in every node X as :

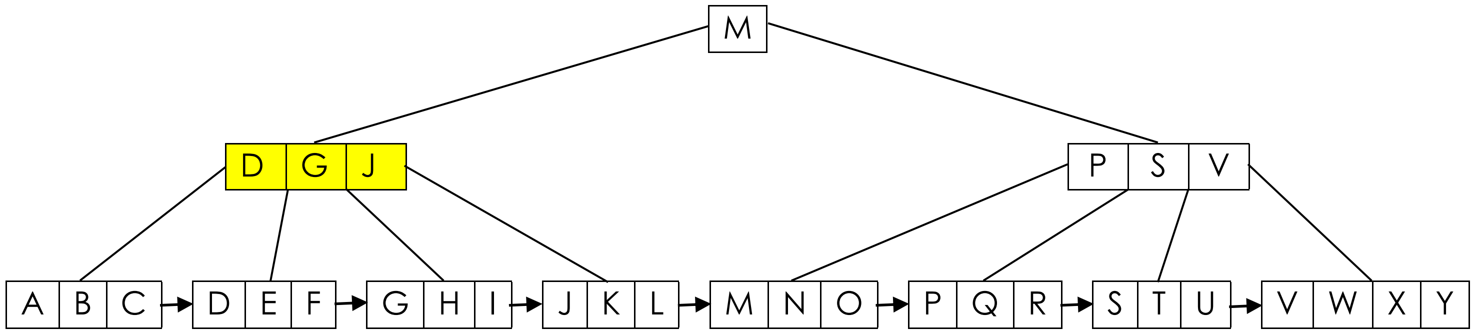
- If X is a non-leaf node, we seek for the first router value X.router[i] which is greater than or equal to the key K. After that, the search continues in the node pointed to by X.child[i].
- If all router values in node X are smaller than the key K searched for, we continue in the node pointed to by the last pointer in the X.child array.
- If X is a leaf node, we inspect whether X is stored in this node.

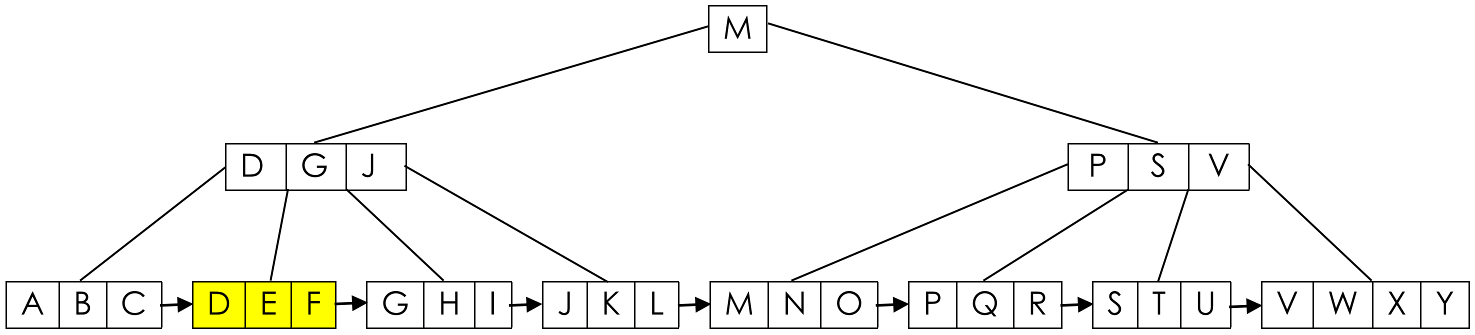
ALGORITHM :

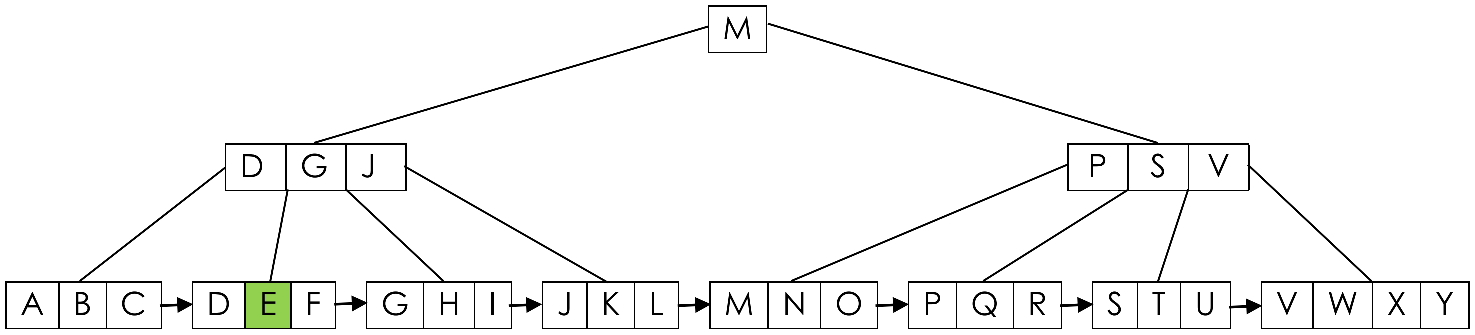
```
x = tree.root
while (x.leaf == false):
   i = 1
   while (i <= x.n and k > x.router[i]):
         i += 1
   x = x.child[i]
   disk_read(x)
i = 1
while (i <= x.n and k > x.key[i]):
   i += 1
if (i <= x.n and k == x.key[i]):
   return (x, i)
else:
   return NULL
```

EXAMPLE : Search for E in the given B+ Tree.









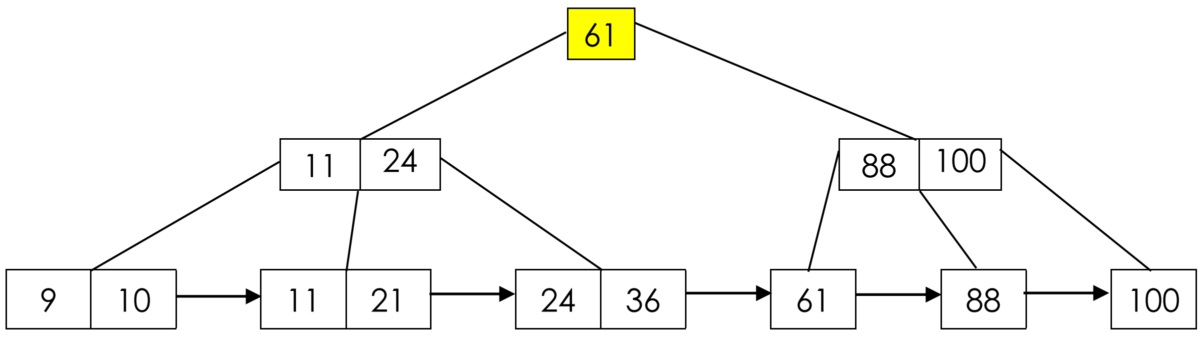
5.4.3 ELEMENT DELETION : O(log N)

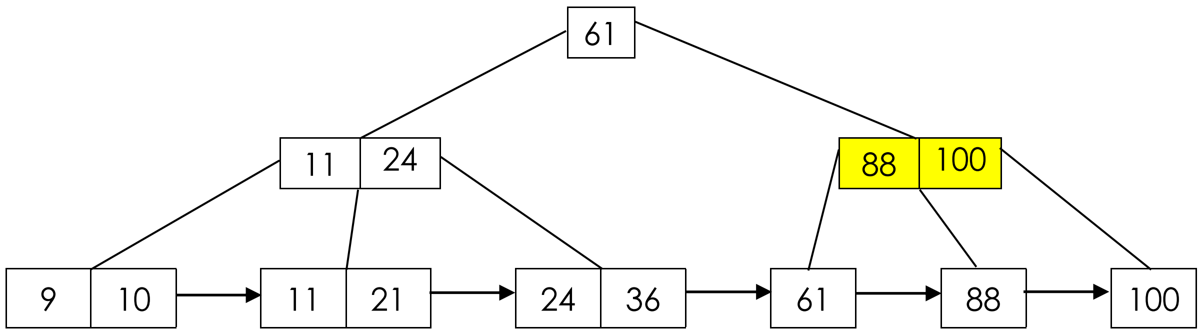
We start the deletion by searching for the leaf node containing the key to be deleted. Notice that we do not delete the same value from the non-leaf node although a non-leaf node would contain this value as a router value. Because every leaf node contains more than one key values, we do not delete the node itself. We just remove the key value from the node. If a leaf node contains at least t keys after the deletion, we are ready and no other actions are needed. If the number of keys in the leaf node is $t - 1$ after deletion then rebalancing becomes necessary. $(t = b/2)$

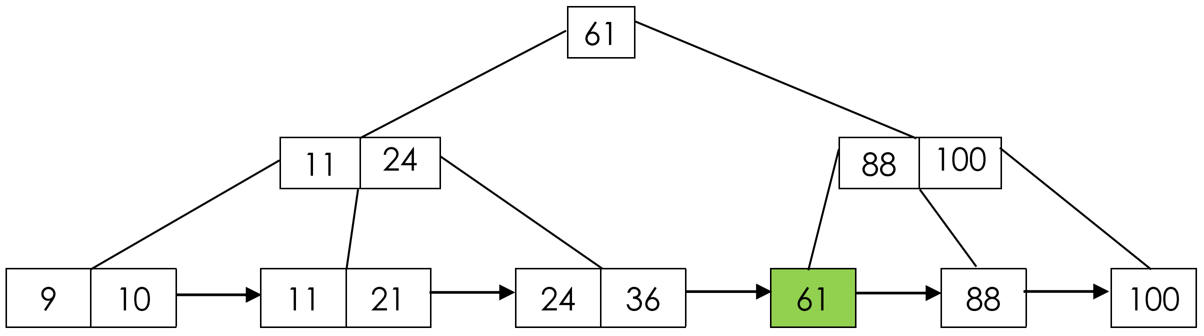
ALGORITHM :

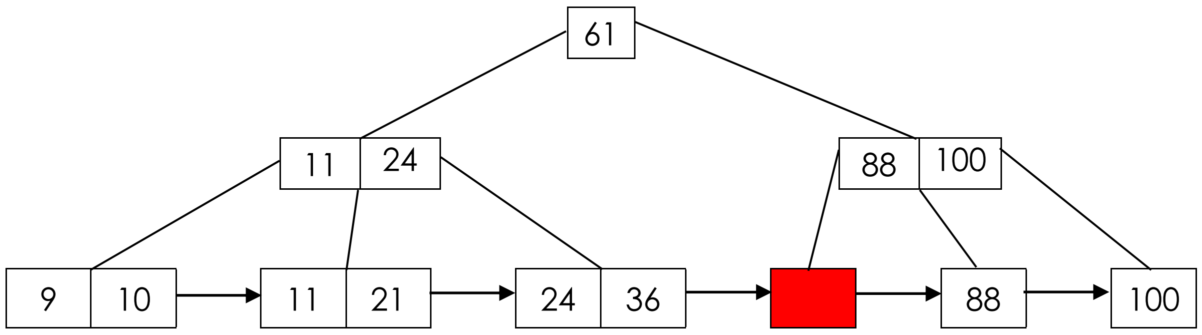
1. Remove the key from its leaf node.
2. If the leaf ends up with fewer than t items then :
    a. Borrow data from a neighbouring node and update the parent node.
    b. If borrowing is not possible, merge the node with a neighbouring node.
    c. If the parent ends up with fewer than t children then follow step 3.
3. If a non-leaf node ends up with fewer than t children then :
    a. Borrow data from a neighbouring node and update the parent node.
    b. If borrowing is not possible, merge the node with a neighbouring node.
    c. If the parent ends up with fewer than than t children then repeat step 3 for the parent node.
4. If the root node ends up with only one child then make the child node as the new root node of the tree.

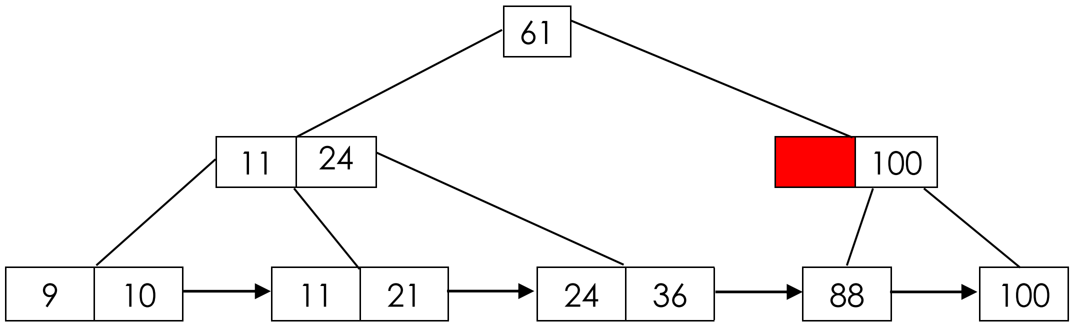
EXAMPLE : Delete 61 from the given B+ Tree.

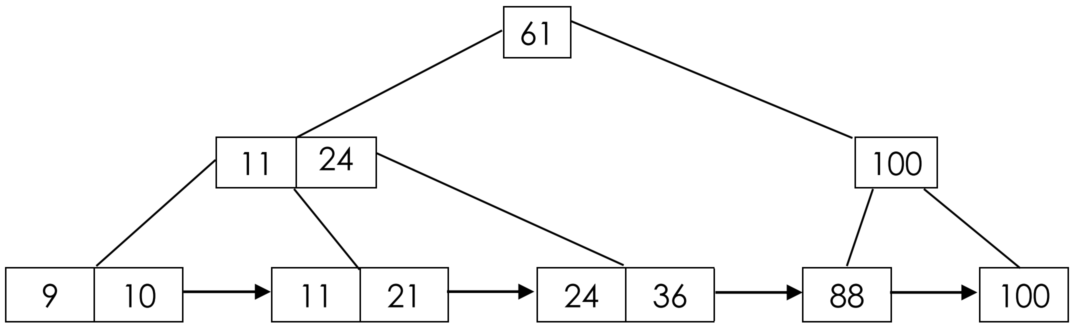







61
11
24
88
100
9
10
11
21
24
36
88
100


61
11
24
100
9
10
11
21
24
36
88
100


61
11
24
100
9
10
11
21
24
36
88
100

6. SUMMARY

S. No.	Data Structure	Applications	Limitations	Insertion	Searching	Deletion
1	Bloom Filter	Distributed Storage Systems, Web Caches	False positive probability increases if filter is overloaded	**O(1)** Faster than Cuckoo Filter	**O(1)** Faster than high load Cuckoo Fiter	Doesn't allow deletion
2	Cuckoo Filter (Low Load)	Virtual Networks, Software Defined Networking	Large memory overhead because of XOR step in insertion	**O(1) Amortized**	**O(1)** Faster than high load Cuckoo Fiter	**O(1)** Faster than high load Cuckoo Fiter
3	Cuckoo Filter (High Load)	Computer Network Security	Operations can take longer time if relocation chain starts	**O(1) Amortized**	**O(1) Amortized**	**O(1) Amortized**
4	B Tree	Filesystems, Database Management Systems	Random lookups need more disk seeks if tree doesn't accommodate on RAM	**O(log N)** Faster than B+ Tree in the long run	**O(log N)**	**O(log N)**
5	B+ Tree	Large Scale Data Storage, Metadata Indexing	Require extra memory overhead and needs periodic reorganization of growing files	**O(log N)**	**O(log N)** Faster than B Tree	**O(log N)** Faster than B Tree

7. PERFORMANCE COMPARISON

By definition, probabilistic filters and multiway trees serve different purposes.

Bloom filters and Cuckoo filters are probabilistic data structures and can only give probabilistic answers to the question "Is an element is a member of a given set?". There is no order to the data stored in such data structures.

On the other hand, multiway trees are deterministic data structures and serve the purpose of storing data in a structured manner and minimizing the number of disk accesses in order to locate a block of data on the disk.

So naturally, there is no solid ground for comparison between probabilistic filters and multiway trees.

Probabilistic filters are supplementary to multiway trees rather than being complementary to them just as explained in the **Distributed Storage Systems** example in Section 2.2.

Therefore, the following performance comparisons are as :

1. Bloom Filter vs Cuckoo Filters
2. B Tree vs B+ Tree

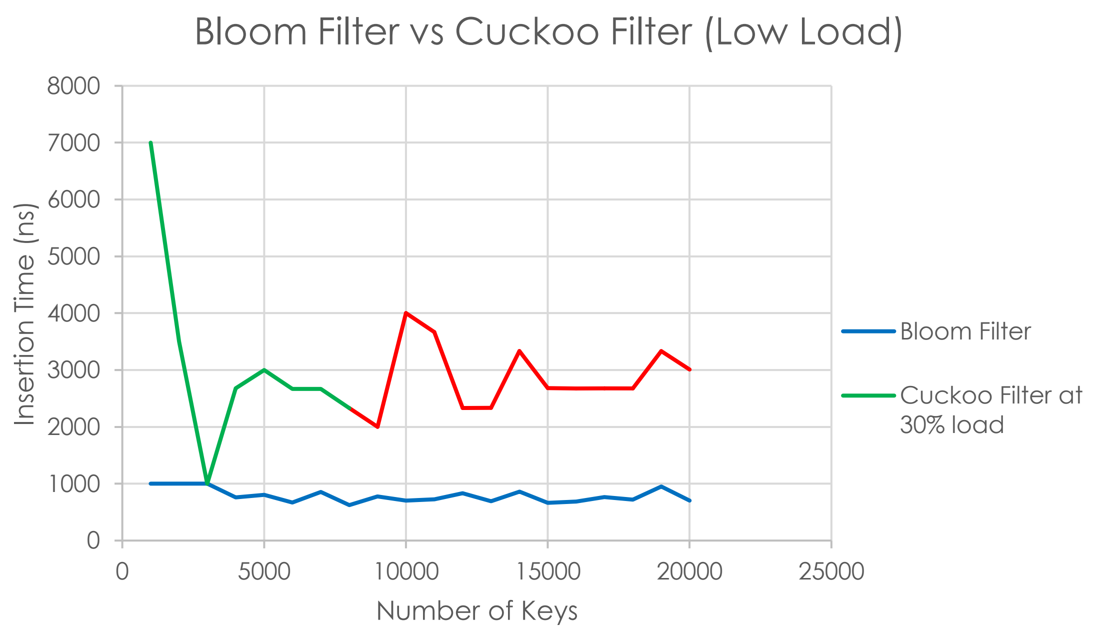
PROBABILISTIC FILTERS

7.1 ELEMENT INSERTION

Insertion operation in Bloom Filters is pretty simple. We just have to hash a particular input a certain number of times with different seed values and set the values as true in the bit array at those hash indices. All that happens is basic overwriting of data.

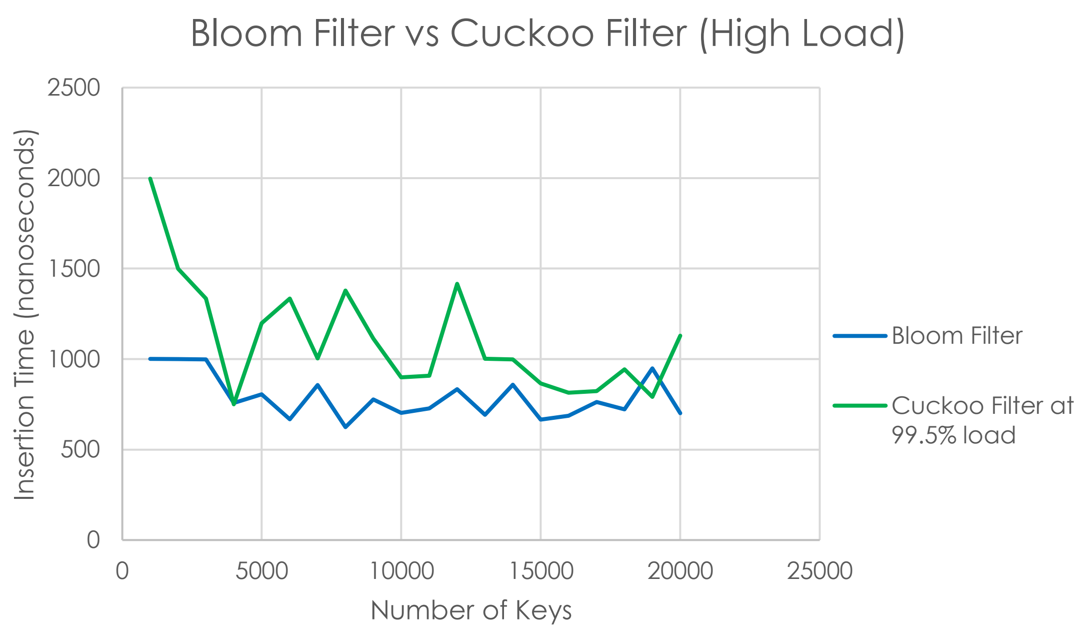
Whereas in Cuckoo Filters insertion can undeterministically take longer time because it doesn't overwrite data in a preoccupied bucket index, instead it tries to relocate it to its alternate location which can sometimes trigger a relocation chain. This accounts for the generally longer insertion time seen in case of Cuckoo Filters.

The number of keys that can be inserted in a low load factor Cuckoo Filter is highly limited because of XOR overflow stated in Section 3.3 which causes segmentation faults if not taken care of.

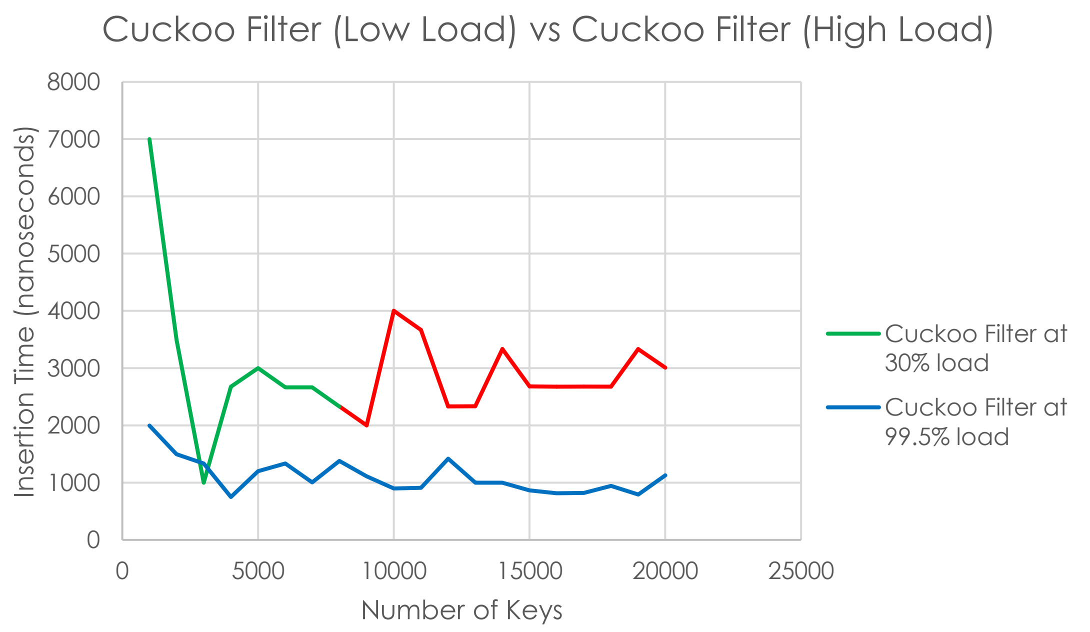


INFERENCE : Insertion in Bloom Filter is faster than in Cuckoo Filter (Low Load).

(Red part means that the hardware couldn't allocate that much memory to the filter so the data was collected by running the filter on the maximum memory allotted)



INFERENCE : Insertion is faster for Bloom Filters than for high load Cuckoo Filters.



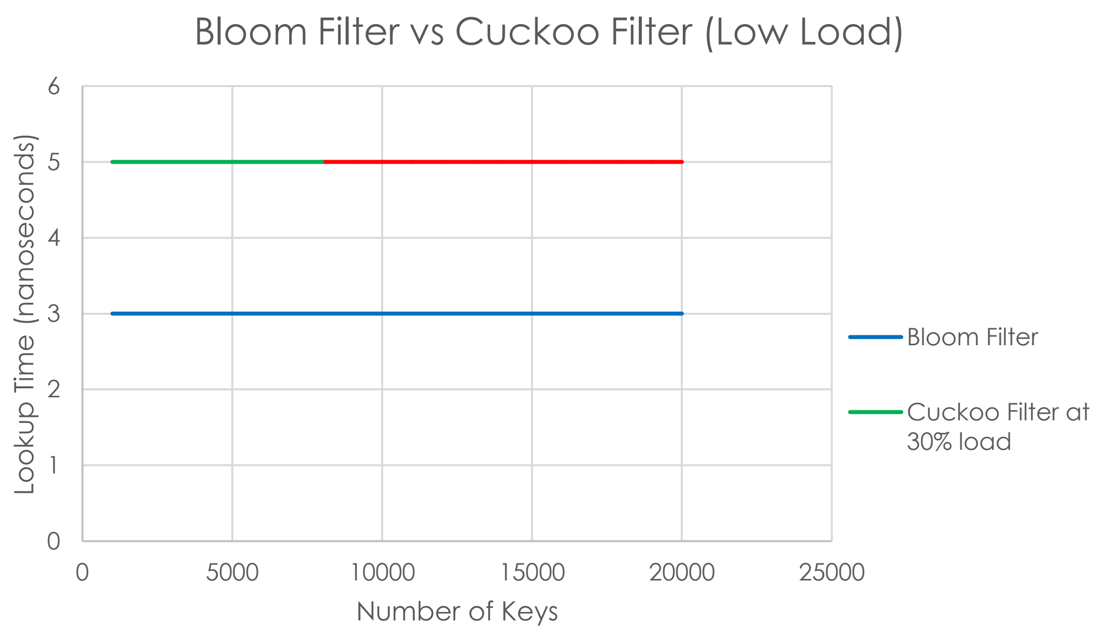
INFERENCE : Insertion is faster for high load Cuckoo Filters than for low load Cuckoo Filters.

7.2 ELEMENT LOOKUP

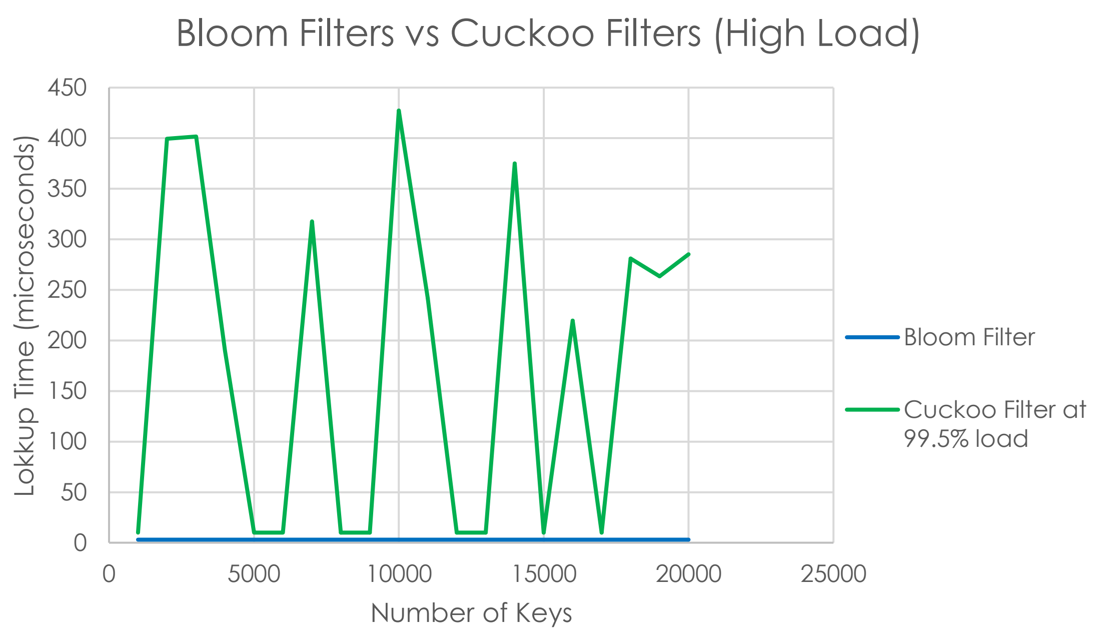
Lookup operation in Bloom Filters and Cuckoo Filters (Low Load) is equally straightforward.

Whereas, in the case of Cuckoo Filters (High Load), lookup can take underterministically high time because it will keep on searching in modified alternate indexes until it finds the element or reaches the relocation threshold.

Results are collected by taking the average of lookup times for five long random strings at each step.



INFERENCE : Lookup time is constant for Bloom Filters and low load Cuckoo Filters.



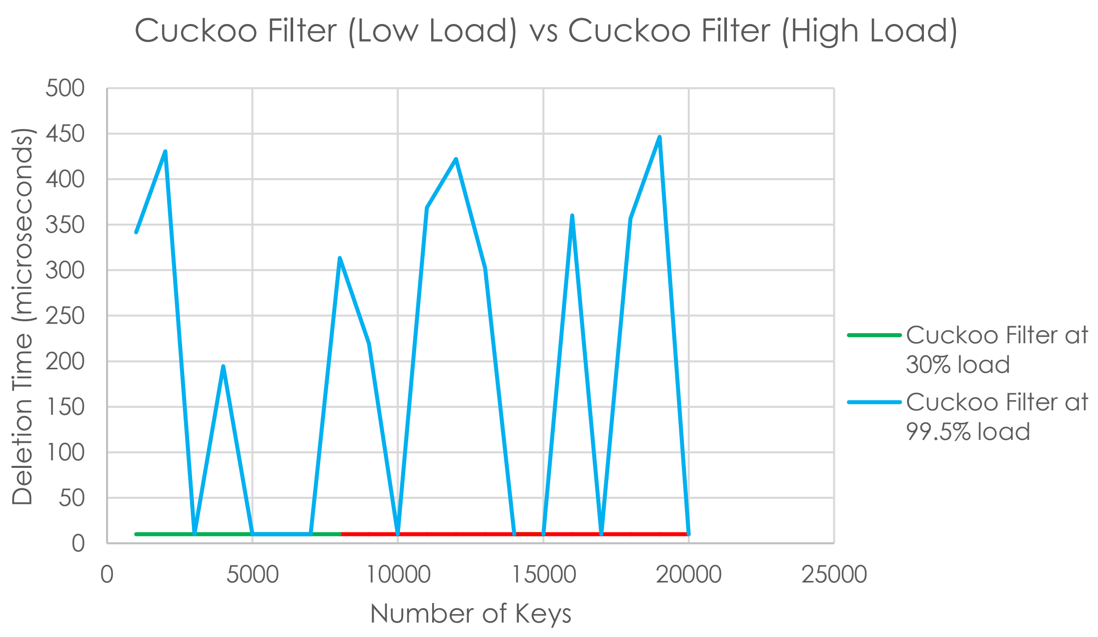
INFRENCE : Lookup time is constant for Bloom Filter but it can reach undeterministically high for some inputs but the average lookup time remains O(1) for high load Cuckoo Filters.

7.3 ELEMENT DELETION

Bloom Filters don't allow deleting any element from them so the comparison is between low load and high load Cuckoo Filters.

Deletion in low load Cuckoo Filters always takes constant time because it searches at two locations at the very most.

While high load Cuckoo Filters can take underterministically high time because it will keep on searching in modified alternate indexes until it finds the element or reaches the relocation threshold.

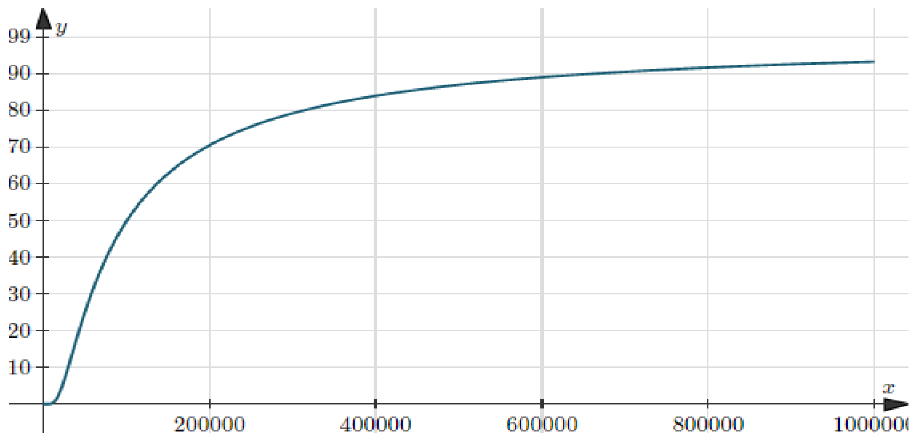


INSERTION : Deletion time is constant for low load Cuckoo Filter but it can reach undeterministically high for some inputs but the average deletion time remains O(1) for high load Cuckoo Filters.

7.4 FALSE POSITIVE PROBABILITY

The false positive probability of a Bloom Filter depends on the size of its bit array (M) and the number of elements that have been inserted into it (C) as :

$$P = 2^{-\frac{M}{C}\ln 2}$$

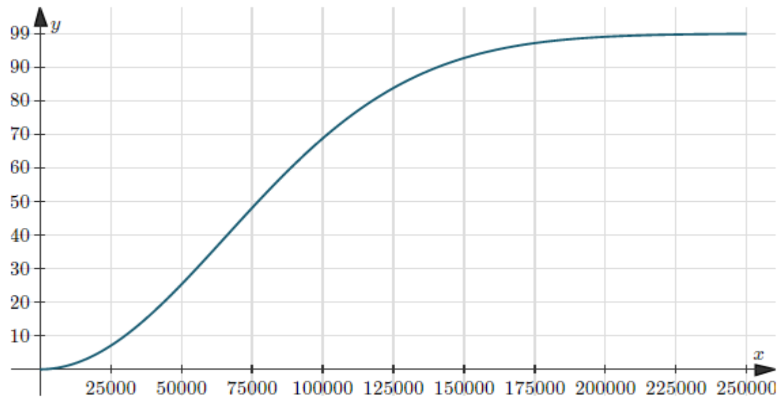


$M = 150000$

On the other hand, for a Cuckoo Filter, the false positive probabiity is same as the collision probability of its underlying fingerprinting function and is given by :

$$P = 1 - e^{\frac{-k(k-1)}{2N}}$$

where, N is the number of possible hash values that the function can generate and k is the number of times the function is called with distinct inputs.



$N = 2^{32}$

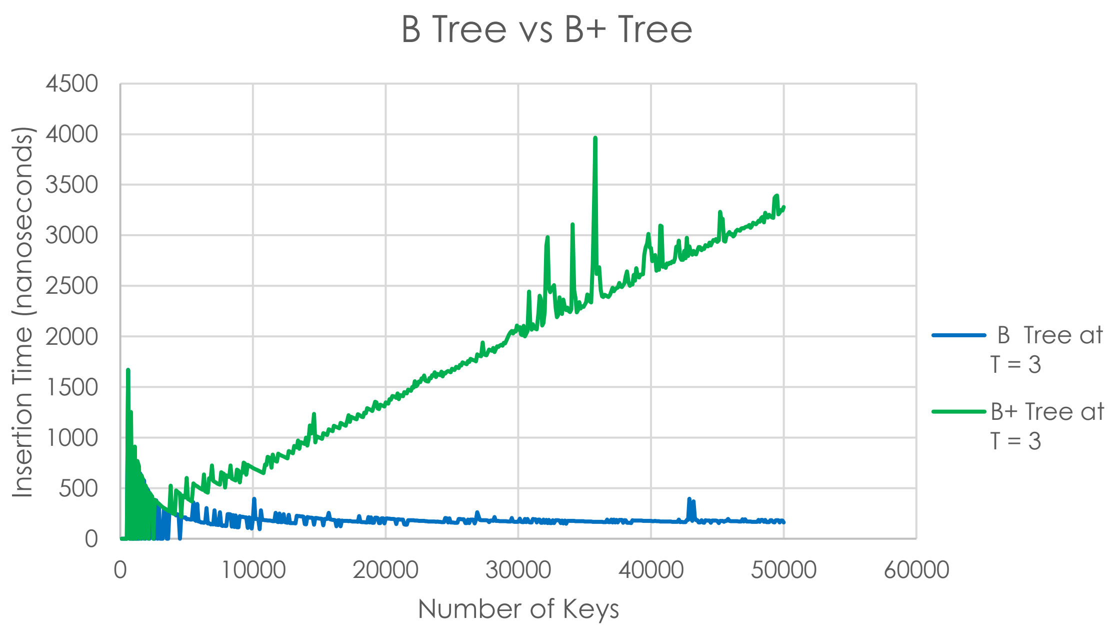
INFERENCE : The false positive probability of Bloom Filters can be reduced by not letting the occupancy ratio go very high, but for Cuckoo Filters it is purely dependent on the choice of fingerprinting function.

MULTIWAY TREES

7.5 ELEMENT INSERTION

Insertion happens at leaf nodes in both B Trees and B+ Trees. The nodes can accommodate a maximum of $2t - 1$ keys and $2t$ child nodes where t is the minimum degree of the tree.

If a node grows beyond that then it is splitted into two nodes and a key is moved upward into the parent node and that is how multiway trees grow upwards.



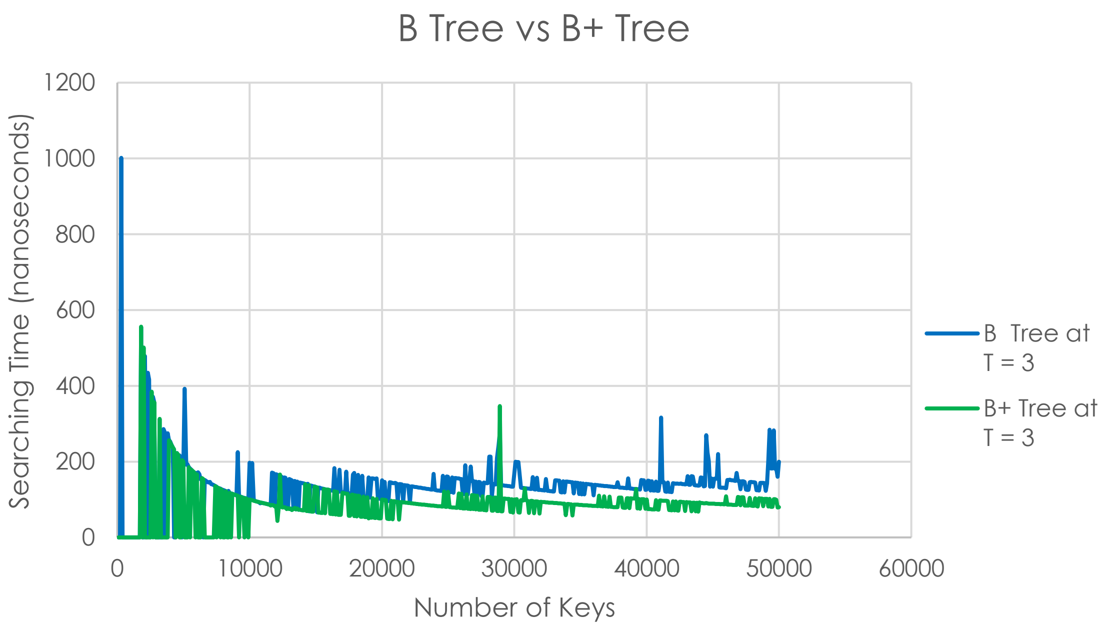
INFERENCE : Insertion gets slower in B+ Trees as number of element increases but for B Trees it shows little deviation. One factor is that the constant factor for B Tree in its time complexity equation is very small in comparison to B+ Trees.

7.6 ELEMENT SEARCHING

Searching in mutiway trees is somewhat similar to the searching pattern in BST. But here we have to search in all the keys of a node before descending down to a child node.

In B Trees, the element can be present in an internal node or a leaf node but in B+ Trees all elements are guaranteed to be present in the leaf nodes and all leaf nodes are linearly connected like linked lists.

Searching time is calculated by taking the average of the searching time for all inserted elements in tree.

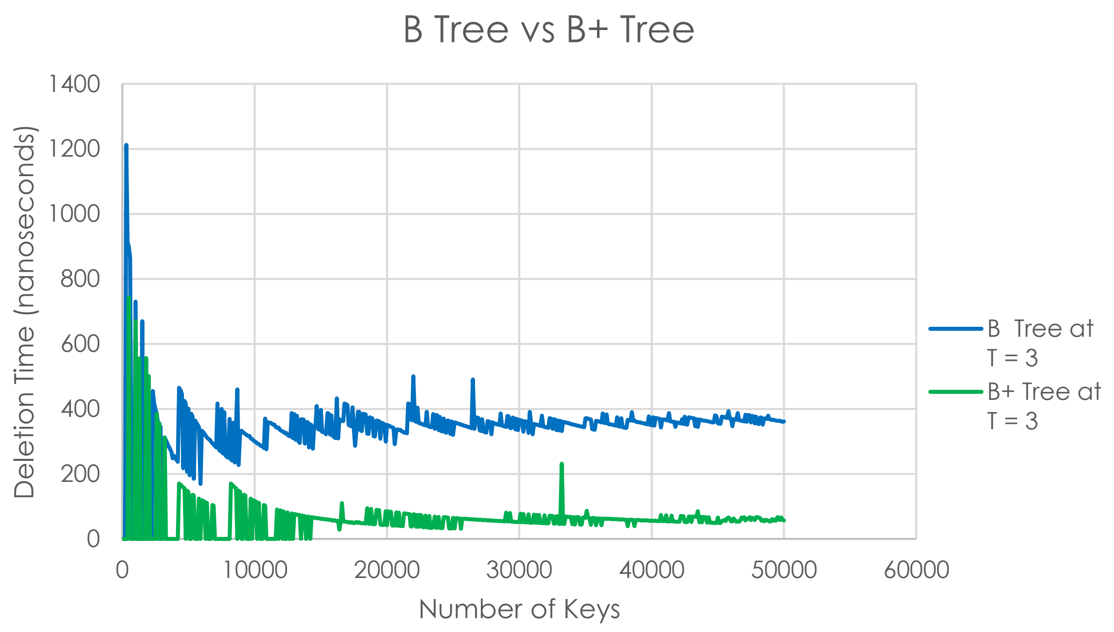


INFERENCE : Searching time follows the same trend for B Trees and B+ Trees but it remains lower for B+ Trees by a little amount because we directly descend to the leaf node to search for the element.

7.7 ELEMENT DELETION

Deletion in B Trees happends by searching for the element to be deleted and checking that whether deletion would cause the number of keys in the node to go below the minimum degree of the tree. If no then deletion is simply carried out, otherwise one key is borrowed from either of the siblings. If the siblings cannot afford to lend keys then merging of nodes happens to merge two nodes into one and finally the key is deleted.

For B+ Trees, we search for the element in the leaf nodes. If the leaf node wouldn't contain the minimum amount of keys after deletion then rebalancing is necessary.



INFERENCE : Deletion time follows the same trend for B Trees and B+ Trees but it remains lower for B+ Trees by a little amount because we look for the element directly in the leaf nodes. For a large number of keys, deletion time doesn't differ by too much for both trees.

8. CONCLUSIONS

PROBABLISTIC FILTERS

- Bloom Filters and Cuckoo Filters are good for quick set membership tests.
- False positive probability of Bloom Filters can be kept under control by not letting their occupancy ratio get too high. While for Cuckoo Filters it can be done by choosing a fingerprinting function according to the use case.
- Insertion and searching in Bloom Filters is faster than in high load Cuckoo Filters.

Overall, Cuckoo Filters are a better choice than Bloom Filters because they allow us to remove an element which doesn't let them grow obsolete with their continued use.

MULTIWAY TREES

- B Trees and B+ Trees are an excellent choice for data management on disk and reducing disk I/O operations.
- The number of disk accesses can further be reduced by tweaking the branching factor of B Trees and B+ Trees according to the block size of disk.
- Insertion is faster in B Trees while searching and deletion is faster in B+ Trees.

B+ Trees are better than B Trees when it comes to reducing disk accesses. Since they don't have data associated with interior nodes, more keys can fit on a page of memory. Therefore, it will require fewer cache misses in order to access data that is on a leaf node.

The combined scenario for Cuckoo Filters and B+ Trees would be a data storage system which stores data in multiple tables on the disk and each table has an associated B+ Tree for disk access operations. Each table also maintains a Cuckoo Filter which can face a data query before the B+ Tree and tell whether that data is present in the underlying table or not. Such a combined use of both data structures saves a huge amount of lookup time and disk operations for the data storage system.