DS project-B+ tree vs Hashing

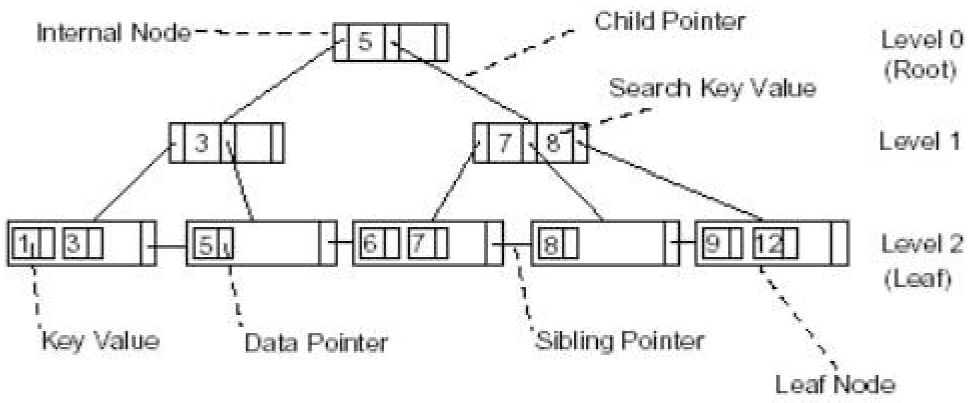
20120206 - Ayush Khandelwal
201202100 - Ashish Kumar

April 21, 2013

Contents

1 B+ trees

The B-tree is the classic disk-based data structure for indexing records based on an ordered key set. The B+-tree (sometimes written B+-tree, B+tree, or just B-tree) is a variant of the original B-tree in which all records are stored in the leaves and all leaves are linked sequentially. The B+-tree is used as a (dynamic) indexing method in relational database management systems.

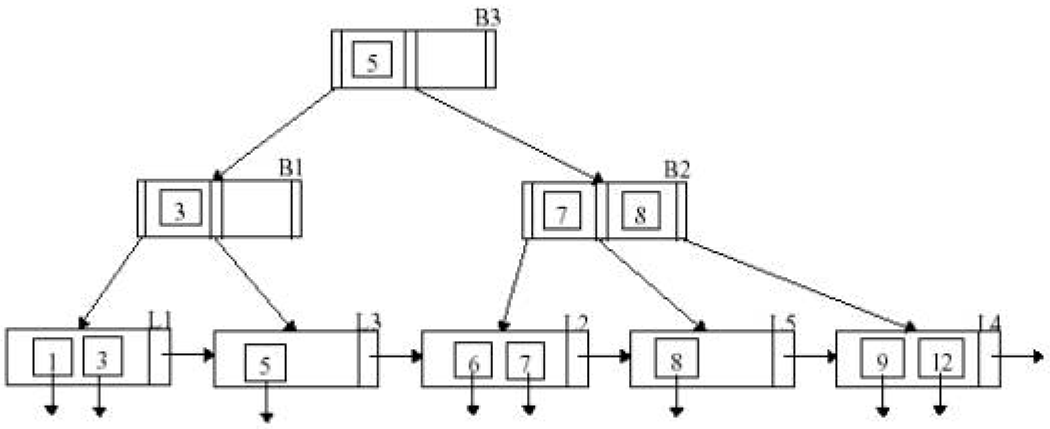


B+-tree considers all the keys in nodes except the leaves as dummies. All keys are duplicated in the leaves. This has the advantage that is all the leaves are linked together sequentially, the entire tree may be scanned without visiting the higher nodes at all. The B + -Tree consists of two types of (1) internal nodes and (2) leaf nodes:

1. Internal nodes point to other nodes in the tree.
2. Leaf nodes point to data in the database using data pointers. Leaf nodes also contain an additional pointer, called the sibling pointer, which is used to improve the efficiency of certain types of search.

1.1 Searching in B+ trees

Searching a B+-Tree for a key value always starts at the root node and descends down the tree. A search for a single key value in a B+-Tree consisting of unique values will always follow one path from the root node to a leaf node.



Searching for Key Value 6

1. Read block B3 from disc. read the root node
2. Is B3 a leaf node? No its not a leaf node so the search continues
3. Is 6 ¡= 5? No step through each value in B3
4. Read block B2. when all else fails follow the infinity pointer
5. Is B2 a leaf node? No B2 is not a leaf node, continue the search
6. Is 6 ¡= 7? Yes 6 is less than or equal to 7, follow pointer
7. Read block L2. read node L2 which is pointed to by 7 in B2
8. Is L2 a leaf node? Yes L2 is a leaf node
9. Search L2 for the key value 6. if 6 is in the index it must be in L2

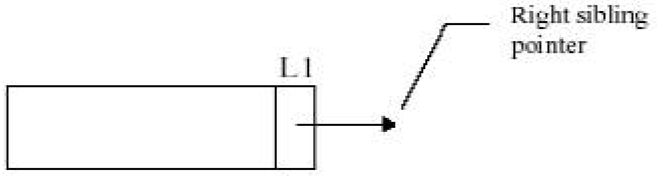
Searching for Key Value 5

1. Read block B3 from disc. read the root node
2. Is B3 a leaf node? No its not a leaf node so the search continues
3. Is 5 ¡= 5? Yes step through each value in B3
4. Read block B1. read node B1 which is pointed to by 5 in B3
5. Is B1 a leaf node? No B1 is not a leaf node, continue the search
6. Is 5 ¡= 3? No step through each value in B1
7. Read block L3. when all else fails follow the infinity pointer
8. Is L3 a leaf node? Yes L3 is a leaf node
9. Search L3 for the key value 5. if 5 is in the index it must be in L3

1.2 Inserting in B+ trees

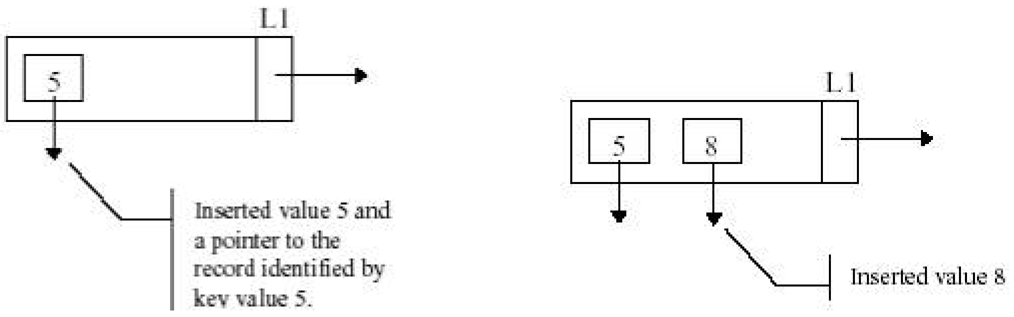
Simple example of insertion in a B+ tree is shown with the following insertion sequence.

Insert sequence : 5, 8, 1, 7, 3, 12, 9, 6 **Empty Tree**



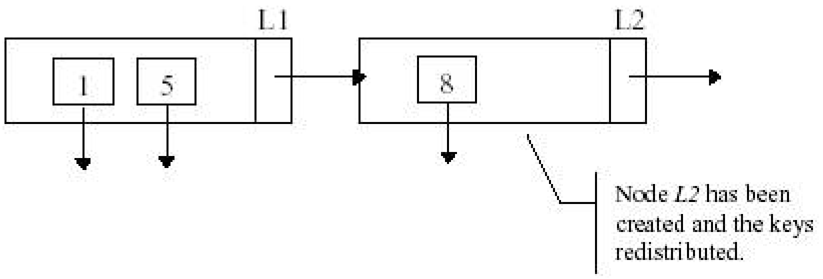
A leaf node consists of one or more data pointers and a pointer to its right sibling.

Inserting Key Value 5 and 8

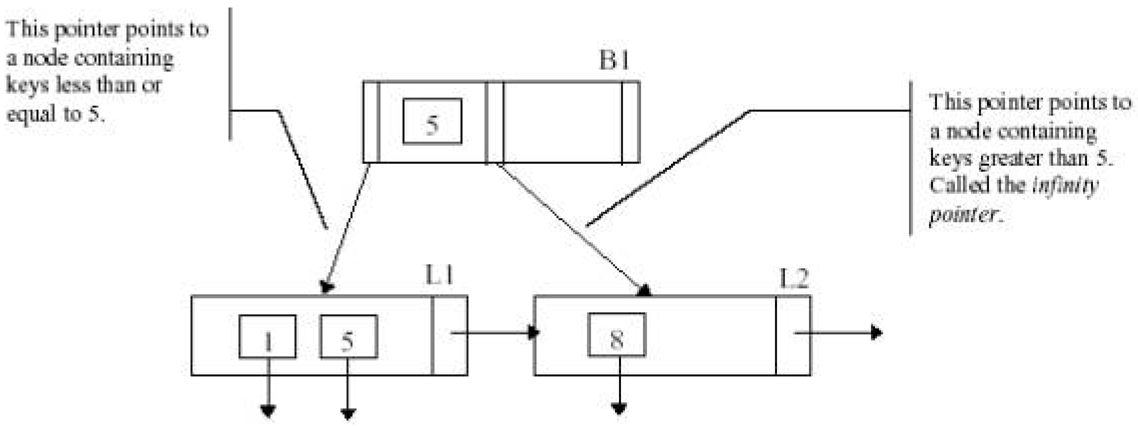

L1
5
Inserted value 5 and a pointer to the record identified by key value 5.
L1
5
8
Inserted value 8

To insert a key search for the location where the key would be expected to occur. In our example the B+-Tree consists of a single leaf node, L1, which is empty. Hence, the key value 5 must be placed in leaf node L1.
Again, search for the location where key value 8 is expected to be found. This is in leaf node L1.
There is room in L1 so insert the new key.

Inserting Key Value 1



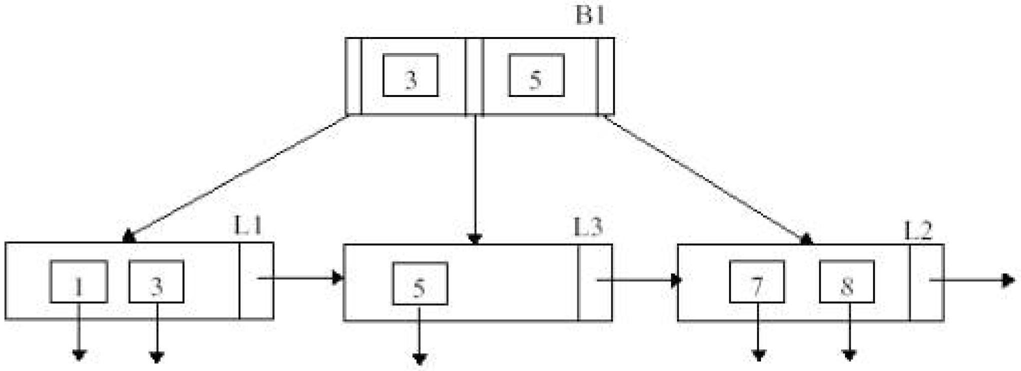
L1 must be split into two nodes. The first node will contain the first half of the keys and the second node will contain the second half of the keys.


This pointer points to a node containing keys less than or equal to 5.
B1
5
This pointer points to a node containing keys greater than 5. Called the *infinity pointer*.
L1
1
5
L2
8

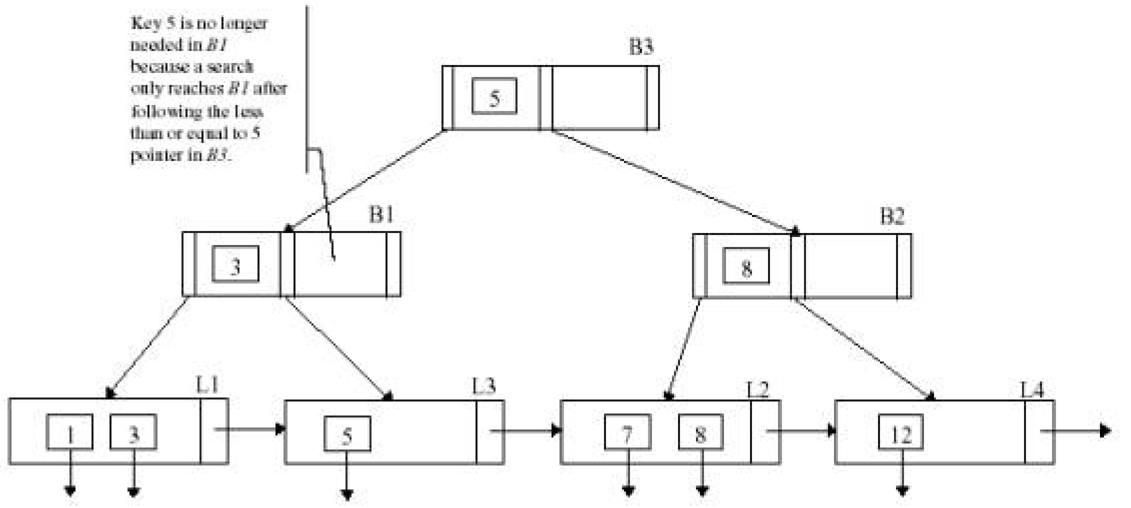
We create a new root node and promote the rightmost key from node L1.

Insert Key Value 7 and 3

Search for the location where key 3 is expected to be found results in reading L1. But, L1 is full and must be split.

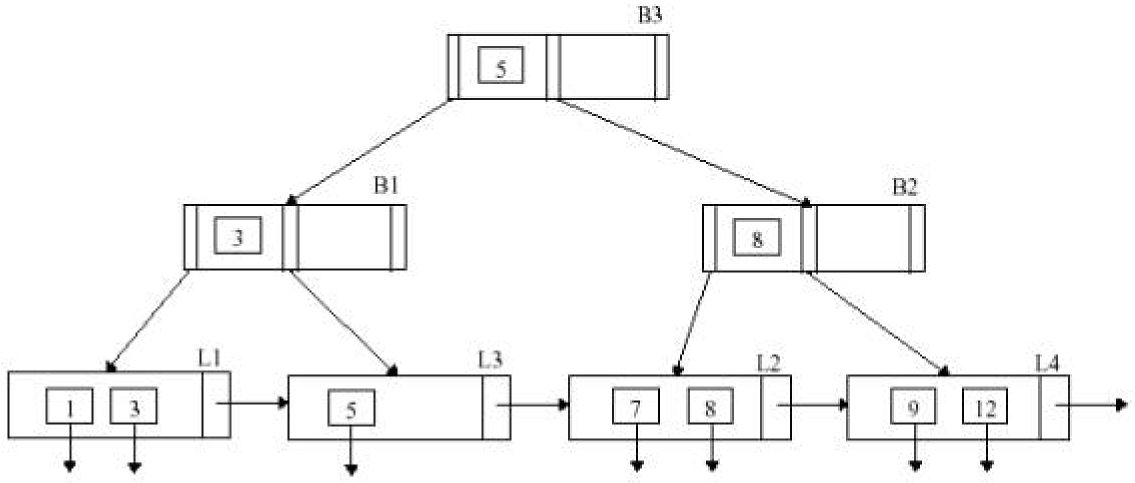


L1 was pointed to by key 5 in B1. Therefore, all the key values in B1 to the right of and including key 5 are moved to the right one place.
Insert Key Value 12
Search for the location where key 12 is expected to be found, L2. Try to insert 12 into L2. Because L2 is full it must be split.
As before, we must promote the rightmost value of L2 but B1 is full and so it must be split.


Key 5 is no longer needed in *B1* because a search only reaches *B1* after following the less than or equal to 5 pointer in *B3*.
B3
5
B1
3
B2
8
L1
1
3
L3
5
L2
7
8
L4
12

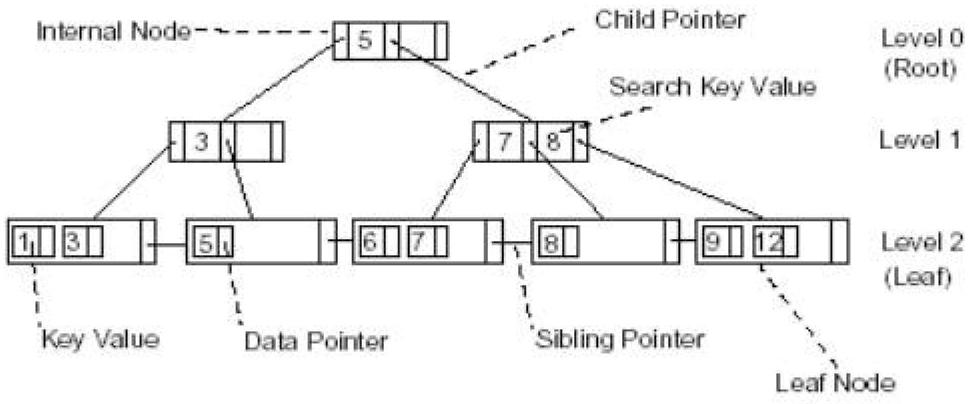
Now the tree requires a new root node, so we promote the rightmost value of B1 into a new node.

Insert Key Value 9



Search for the location where key value 9 would be expected to be found, L4.
Insert key 9 into L4.
As before, we must promote the rightmost value of L2 but B1 is full and so it must be split.
Now the tree requires a new root node, so we promote the rightmost value of B1 into a new node.
The tree is still balanced, that is, all paths from the root node, B3, to a leaf node are of equal length.

Insert Key Value 6

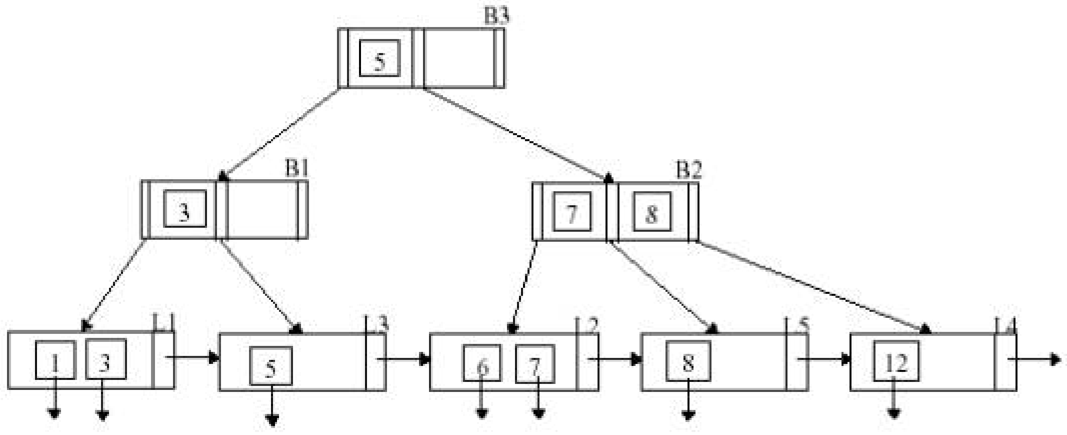


Key value 6 should be inserted into L2 but it is full. Therefore, split it and promote the appropriate key value.
Leaf block L2 has split and the middle key, 7, has been promoted into B2.

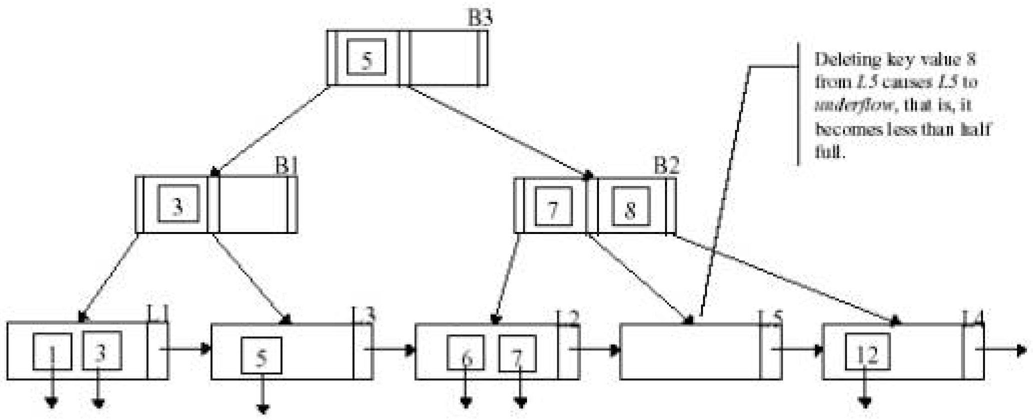
1.3 Deleting in B+ trees

Deleting entries from a B+-Tree may require some redistribution of the key values to guarantee a wellbalanced tree.
Deletion sequence: 9, 8, 12.

Delete Key Value 9



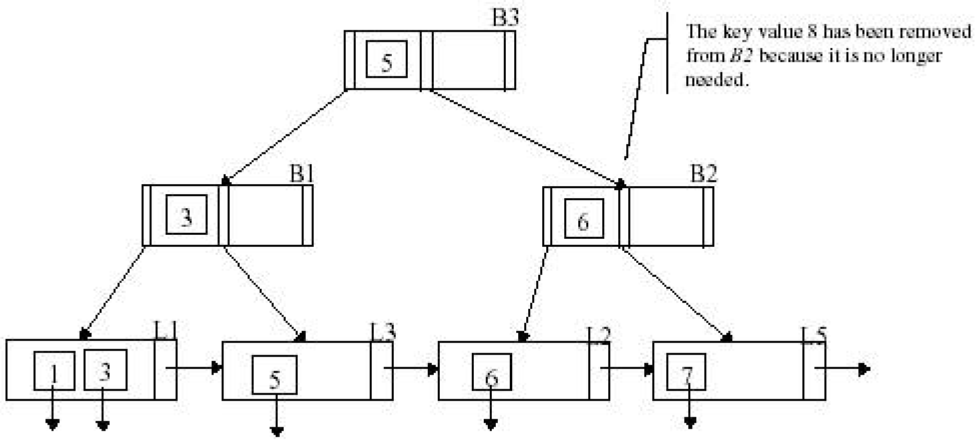
First, search for the location of key value 9, L4. Delete 9 from L4. L4 is not less than half full and the tree is correct.
Delete Key Value 8


B3
5
B1
3
B2
7
8
L1
1
3
L3
5
L2
6
7
L5
L4
12
Deleting key value 8 from *L5* causes *L5* to *underflow*, that is, it becomes less than half full.

Search for key value 8, L5. Deleting 8 from L5 causes L5 to underflow, that is, it becomes less than half full.
We will now redistribute some of the values from L2. This is possible because L2 is full and half its contents can be placed in L5. As some entries have been removed from L2, its parent B2 must be adjusted to reflect the change.

Deleting Key Value 12

Deleting key value 12 from L4 causes L4 to underflow. However, because L5 is already half full we cannot redistribute keys between the nodes. L4 must be deleted from the index and B2 adjusted to reflect the change.


B3
5
B1
3
B2
6
L1
1
3
L3
5
L2
6
L5
7
The key value 8 has been removed from *B2* because it is no longer needed.

The tree is still balanced and all nodes are at least half full. However, to guarantee this property it is sometimes necessary to perform a more extensive redistribution of the data.

2 Hashing

Hashing is based on array of linked lists. It is called a hash table. Consider an array T of size $|T|$. Consider a function h that maps elements in U to the set $0, 1, ..., |T| - 1$.
T is called a hash table
h is called the hash function.
The function h can be used to map indices to the array. Now U can be any set, not just integers. The function h can map its input to an integer in the appropriate range. We will still however use integers for our setting.

2.1 Insert

1. Let $U = 1, 2, ..., 100$.
2. Let $K = 34, 65, 22, 76, 97$.
3. Let $h(k) = kmod10.So, |T| = 10$.
4. Key 22 to be stored in cell h(22) = 22 mod 10 = 2.

Key 76 to be stored in cell h(76) = 76 mod 10 = 6. Suppose $U = 1, 2, ..., 100$ as earlier.
Suppose h(k) = k mod 10, as earlier, with $|T| = 10$.
Suppose $K = 25, 76, 82, 91, 65$.
Notice that 65 is different from 25. So should store both. But, each cell of the array T can store only one element of U. The situation is termed as a **collision**. Notice that h maps elements of U to a range of size $|T|$. If $|U| > |T|$, we cannot always avoid collisions. We should have a way to handle them properly. Such techniques are called collision resolution techniques. Can treat each cell of the table T as a pointer to a list. The list at cell k contains all those elements that have a hash value of k. The above technique is called chaining. Names comes from the fact that elements with the same hash value are chained together in a linked list.

2.2 Search and delete

Searching in a hash table is easy and fast. We have used binary search tree to get better runtime for search and delete.

1. Apply the hash function h on the search query and it gives the index of the hash table were the query is located.
2. Go to that index and traverse the tree connected to it to search for the requested query.

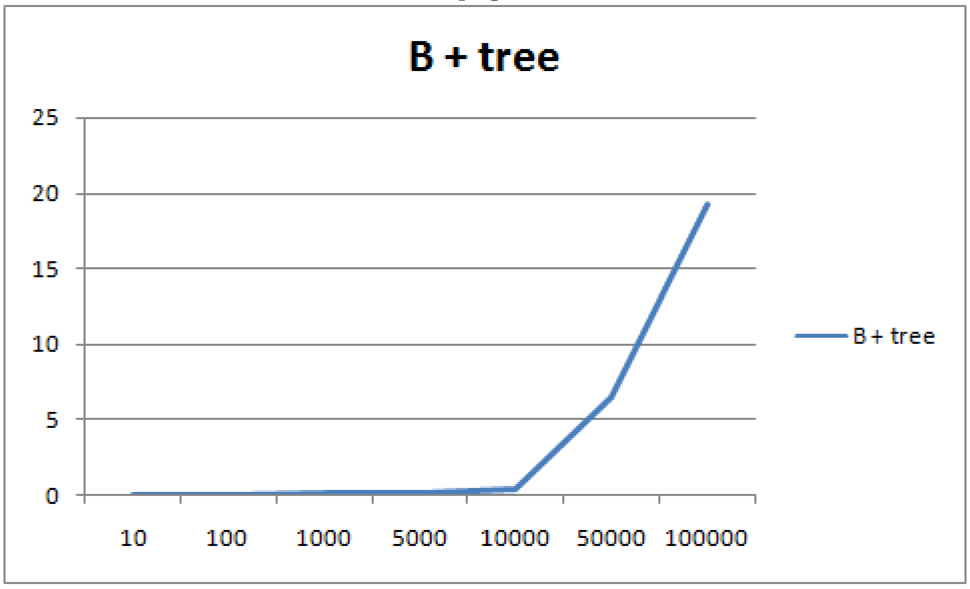
Therefore searching can be approximated to be of order O(1). Searching can be improved by implementing a binary search tree at every node of the hash table.

Delete is similar to searching with the addition that we actually need to delete the node from the chained linked list.

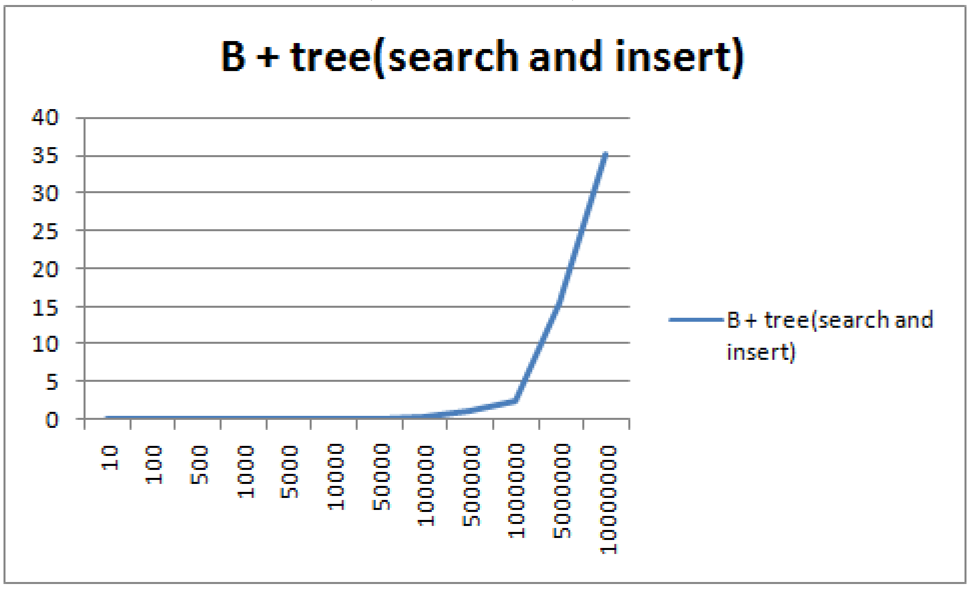
3 Analysis

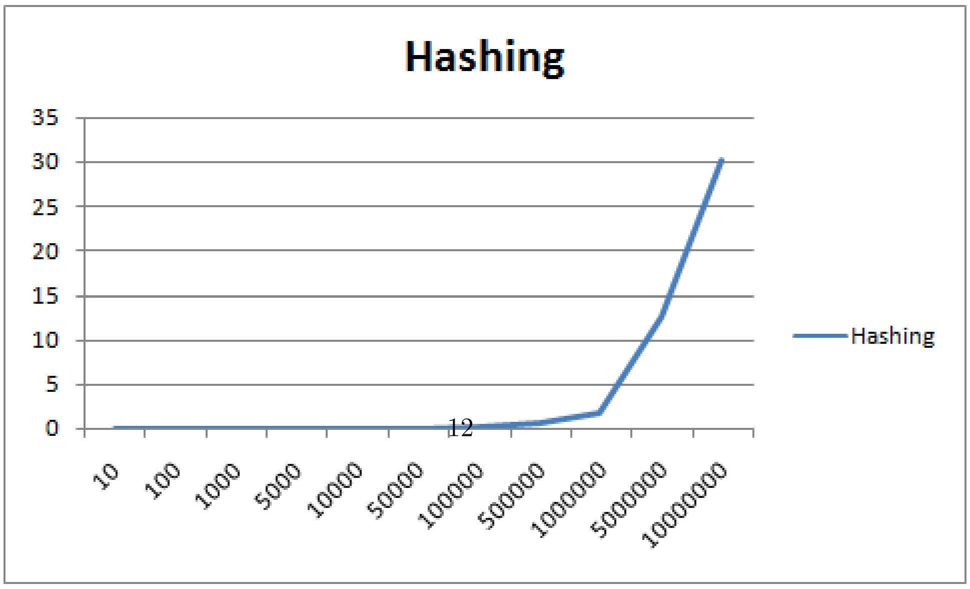
Time complexity of insert, search and delete in $O(t * log_t(n))$ for B+ tree.
The number of disk operations needed is O(logt n). This is the dominating factor of the time demand in practice.
Time complexity for searching and insertion is O((logn)) where n is the number of numbers at every node.

trr.png



+ tree(search and insert).png


B + tree(search and insert)
B + tree(search and insert)


Hashing
Hashing