CPSC 4660 - Database Management

Systems Course Project

Final Report

-

Rylan Bueckert and Siebrand Soule

B+ Trees

Summary:

B+ tree indexing is a common way to index records in a database. In a B+ tree, all data is kept at the leaf nodes, and all the nodes above are for giving directions to the desired information. As the leaf nodes fill up, more leaf nodes are created to accommodate new records. This requires more internal nodes to connect them. If a B+ tree has n nodes, and is of order m, the height of the tree is approximately $\log_m n$. For this project, we have implemented a B+ tree data structure and have implemented its key functions. Our B+ tree supports insert, find and remove.

Implementation Details:

The B+ tree is implemented as a C++ class. For testing purposes, it can only work with the `int` data type. To use this class, you must `#include <BPTree.h>`, then you can create a B+ tree like this:

```
BPTree myTree();
```

You may optionally specify the order of the tree as a parameter like this:

```
BPTree myTree(6);
```

The specified order must be at least 3. If no order is specified, it will be 4 by default.

Available Functions:

`bool insert(int key)`	`Inserts a key into the B+ Tree`
	`Parameters:`
	`key - the number to be inserted into the B+ Tree`
	`Returns:`
	`True if the key was inserted`
	`False if the key already exists in the tree`

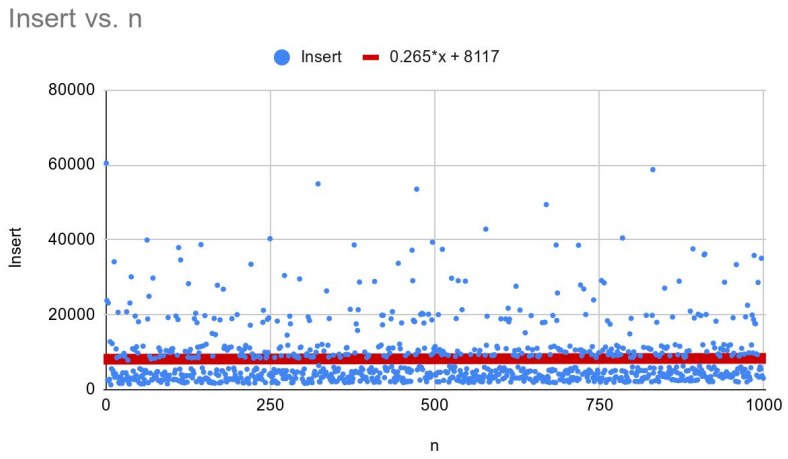
`bool find(int key)`	`Checks if a key exists in the B+ Tree`
	`Parameters:`
	`key - the number to be searched for`
	`Returns:`
	`True if the key was found`
	`False if the key was not found`

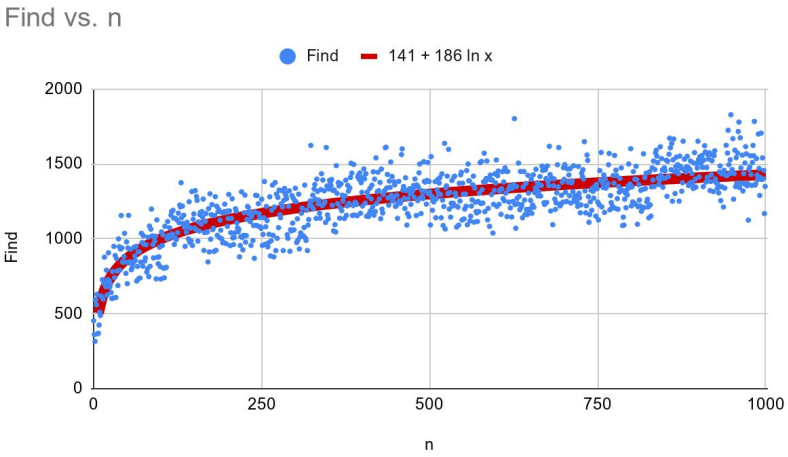
`bool remove(int key)`	`Deletes a key into the B+ Tree`
	`Parameters:`
	`key - the number to be deleted from the B+ Tree`
	`Returns:`
	`True if the key was deleted`
	`False if the key didn't exist in the tree`

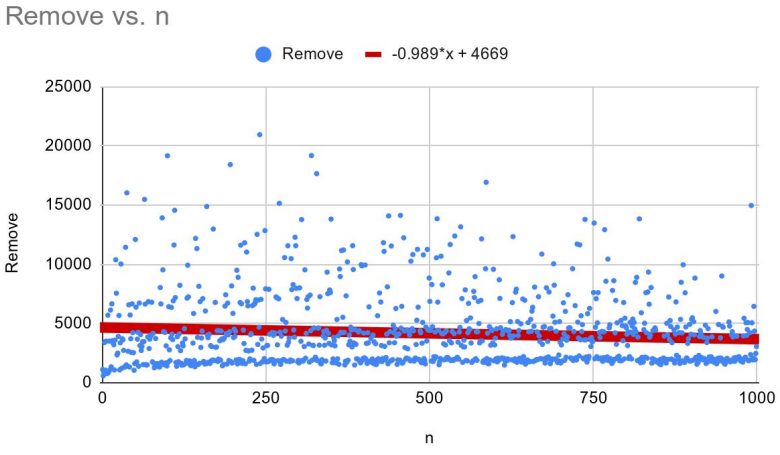
Evaluation Strategy:

The B+ tree was evaluated by timing the execution time of these 3 operations of the tree at different sizes. Timing was done in nanoseconds using the <chrono> library. The numbers 0-999 were inserted in a random order. The random order was determined using std::shuffle and std::default_random_engine with a seed of 0. After every insert, find was executed on the number just inserted. At the end, all the numbers were removed from the B+ tree in a different random order. The execution time of each operation was recorded and displayed at the end so that it could be exported to a spreadsheet and graphed. For this test, the tree was of order 3.

Results:


Insert vs. n
Insert
0.265*x + 8117
80000
60000
40000
20000
0
Insert
0
250
500
750
1000
n


Find vs. n
Find
141 + 186 ln x
2000
1500
1000
500
0
Find
0
250
500
750
1000
n


Remove vs. n
Remove
-0.989*x + 4669
25000
20000
15000
10000
5000
0
Remove
0
250
500
750
1000
n

For inserting, we expected a logarithmic execution time with respect to n because of the traversal of the tree. What we observed was that time to insert grew very slowly. This is likely because the traversal of the tree is dwarfed by the time it takes for splits and redistribution.

Finding values in the tree was very consistent with what was expected. The execution time has a very logarithmic shape, as find is mostly just a traversal of the tree, then a linear search through a node.

Remove was very similar to insert, in that it had a very close to constant execution time. Once again, this is probably because the traversal of the tree is dwarfed by the time it takes for concatenation. There does appear to be a logarithmic relation for the fastest cases, as those cases don't require any concatenation.

Conclusion:

The B+ tree seems to be an effective indexing method for data, and due to the near constant times for insert and remove, it would work well for a database. The paper that was used as a reference for the insert algorithm skipped some details in a few places, which meant that the algorithm was slightly altered. We would have liked to have compared the impact of changing the order of the tree as well, but the extendible hashing took the rest of our time. We are satisfied with the results, and they did not deviate too far from what we expected them to be.

Extendible Hashing Tables

Summary:

Extendible hashing tables are a variation of hashing tables, where they dynamically resize to fit the data stored. Global depth is added or decreased in the address space for insertions and deletions respectively. Address space contain pointers to buckets. Buckets hold data that has a local depth which reflects the bits in the address space that point to it. The global depth and local depth are determined by the hash function and we bitshift to get the amount of significant bits needed. We have implemented insert, find and delete. Extendible hash tables resize by spitting buckets and increase global depth during insertions and merge from deletions and shrink the global depth. Insert, find, delete, split, and growing works whereas we ran into complications and time constraints to get merging and shrinking working properly.

Implementation Details:

Extendible hashing tables proved to be more difficult than we initially thought. Thus we ran into problems with split and merge when larger amounts of data is stored. Extendible hashing tables merge buckets together when two buckets have the same local depth and bucket space to account for the keys. The address shrinks(halves) when there is half the buckets as address space. Similarly, for the splitting of bucket, except the address space doubles. The extendible hashing tables were implemented in C++, the table has a set hash function of: h(x) = `key mod 64`, then we bit shift to get the required amount of bits to perform operations on the key. An extendible hash table is created by:

```
Address myTable(x, y);  // Where x is the initial global depth usually
                        // 0, and y is the number of buckets per
                        // address space.
```

Available Functions:

`bool search(int key)`	`Checks if a key exists in the table`
	`Parameters:`
	`key - the number to be searched for`
	`Returns:`
	`True if the key was found`
	`False if the key was not found`

`void insert(int key)`	`Inserts a key into the hash table, checks if grow or split need to be done`
	`Parameters:`
	`key - the number to be inserted into the hash table`

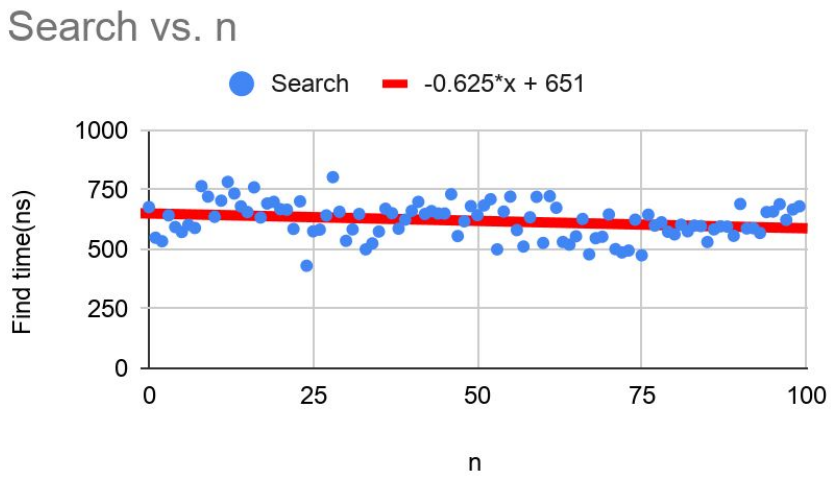
`void deleteKey(int key)`
 	
Note: Does not work with merging and shrinking address space.
	`Deletes a key into the table, checks if shrink or merge need to be done`
	`Parameters:`
	`key - the number to be deleted from the hash table`

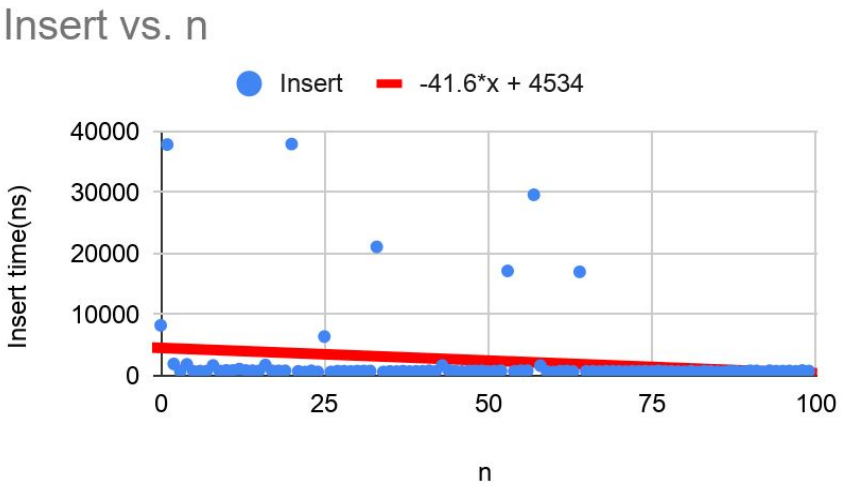
Evaluation Strategy:

Due to stated complications our evaluation had to be altered from the proposed criteria as our table will not perform as expected. The <chrono> timing library was also used for timing the extendible hash table operations. We recorded the execution times of extendible hashing tables the same way as B+ trees, except we used 100 keys instead of 1000. We were able to test 100 random insertions between 0 and 99 while performing a search for the key right after. We displayed the execution time and recorded it to display using graphs. We were unable to test delete because merge and

shrink do not work correctly. We ran into bugs with split when the table got to a certain size and splitting would cause the program to stop working. A beginning global depth of 0 and bucket size of 20 was used during the tests. We initially desired to use a bucket size of 2 but we could only insert 20 keys before it would break. Thus we used a bucket size of 20 to enable to input 100 keys.

Results:





The search operation functions very quickly because only 2 block transfers will determine the place of the key. The first block transfer is determined by the hash function giving the address in which the bucket that the key is a member of. Although a linear search through the buckets may slow down the process but this would only significantly affect the execution time if extremely large bucket sizes were used. Therefore we expected an approximately constant runtime and this is what we observed in our data.

For insertions we expected a relatively constant graph trendline with several outlying data points. Those points reflect a split operation or a split and expansion on the address space. As the hash table gets larger fewer outlying points will result as the table will not expand as often or have the need for as many splits.

We expect 100 random deletion to function very similar to insertions because merge and shrinking address space is the opposite to split and grow address space. Although we were not able to test delete with merging and shrinking properly we still believe that if it was complete the hash table would reflect similar results to the insert graph as delete.

Conclusion:

Extendible hashing is overall an efficient database storing method, where access times are minimal, insertions and deletions are relatively fast apart from the occasional split or merge as well as dynamically resizing the table accordingly. Some bugs in the program set up roadblocks that made splitting and merging not work properly. It would have been ideal to have 1000 keys to perform operations on to make comparing

extendible hashing to B+ trees on the same scale. We feel 100 operations enables us to accurately predict results and the differences of extendible hashing to B+ trees.