First coursework of Advanced Topics in Algorithms:

Dynamic Set Data Structures

Implementation and analysis of AVL Trees and Bloom Filters.

By: Marta Gómez (202003750) and Yosbi Saenz (201801653)

Faculdade de Ciencias
Universidade do Porto
Portugal
May 9, 2021

AVL Trees

- **What it is?:** An AVL tree is a binary search tree that guarantees that for each node, the heights of the left and the right subtrees differ by at most one unit (height invariant). Even inserting and removing nodes, the height remains invariant despite changing the tree structure.
- **Implementation:** For the implementation of the AVL tree we have created a data type *node* where the information of each node will be stored. Here we will find the value of the node, a pointer pointing to the left child and another pointing to the right child, as well as the height. In our tree implementation we will also take into account a pointer to the *rootnode*.
- **The math:**
  1. **Time complexity:** Binary search trees guarantee $O(h)$ worst-case complexity for search, insert and delete operations, where h is the height of the tree. Unless care is taken, however, the height h may be as bad as N, the number of nodes. AVL trees require the heights of the subtrees of any node to differ by no more than one level, which ensures that the height is $O(logN)$. The height of an AVL tree is bounded by roughly $1.44 * log2N$ and if height of AVL tree is h, maximum number of nodes can be $2h + 1–1$.
     (a) **Search:** The number of comparisons required for successful search is limited by the height h and for unsuccessful search is very close to h, so both are in $O(logn)$.
     (b) **Insert:** The time required is $O(logn)$ for lookup, plus a maximum of $O(logn)$ retracing levels ($O(1)$ on average) on the way back to the root, so the operation can be completed in $O(logn)$ time
     (c) **Delete:** Same reasoning as for inserting, $O(logn)$.
  2. **Space complexity:** We have $O(1)$ space for each node, and we have exactly n nodes, so the space complexity will be $O(n)$.
- **Empirical results:** Our results effectively show the efficiency of an AVL tree in different tests. Among them we find for example that for a 1.000.000 element insertion with an 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz processor it only takes about 1.78s (test data4.txt).

Bloom Filter

- **What it is?:** A bloom filter is a probabilistic data structure that is considered as space-efficient as possible and is designed to inform the user, with a certain degree of certainty, whether an item is in a set or not.

  It is important to note that this data structure allows false positives but not false negatives. So if the algorithm says that a piece of data does not exist in the set, it is because it doesn't really exist in the set. But if it says that it does exist, it probably exists with a certain degree of confidence.

- **Implementation:** The implementation of the data structure can support any data type that is serialized in memory, for example an array of chars, single integers, chars or floats, structs, etc. We can divide the implementation in three parts for the sake of simplicity, in the first part the data structure, in the second part the hashing, and in the third part stats.

  1. **Data structure:** The main data structure we used is a dynamically allocated array of integers which we will later access by setting and testing bits in a bitwise manner. We do this, instead of using an array of booleans because, in the implementations of C++ we tested (Apple clang version 12.0.5), each boolean has a byte size (8bits) instead of 1 bit. So it is in fact more optimal in memory space to use an array of integers and access them bitwise instead of having an array of booleans.

     To give the power to the user, we implemented two constructors, one in which we allow the user to manually define the size of the bloom filter (in bits) and the number of hashing functions he wants to use. With this option, the user has the responsibility to handle the rate of false positives. On the other constructor, the user must specify the number of expected items in the bloom filter, and the false positives probability, and we handle automatically the size and the number of hashing functions that will be optimal given these parameters. With this last option, the user must be aware that with more items and a lower ratio of false positives, the algorithm will decide to use more memory and/or use more hashing functions to assure that there is enough space and randomness in the hashing to be able to fit the expected data.
  2. **Hashing:** To help avoid collisions, in the first two hashings we use two different functions, murmur3 and Jenkins. In the subsequent n - 2 functions we use murmur3 with a different seed (the seed we actually use is the ith number of iteration).

     By using two different hash functions, instead of using only a murmur function (for example) with the seed as the ith iteration, we can avoid the case in where the murmur function gives the same result, for example, for "foo" and "bar" in every iteration.
  3. **Stats:** In any moment we can get stats of the data structure such as the size (in bits) of the filter, the number of hashes used, number of objects inserted, the probability of false positives and the number of collisions (hashes repeated) we have. We can also print the data structure (the array of bits) to give a visual aid of what's going on below the hood (this option is only recommended on small filters such as m <= 1024)

- **The math:**

  1. **Time complexity:** We have that the complexity of this algorithm for insertions and for checking whether a data is dominated by the number of hashes that we have to make, so the complexity is O(k) being k = number of hashes. Since in a determined instance of a filter k is a constant we can reduce that the complexity is O(1).
  2. **Space complexity:** To get the space complexity of the algorithm we can think of an ideal hash that does not produce any collisions ever, so in this case k = 1. So to save

n elements inside the data structure we will need n bits, so the space complexity will be O(n). But, as we cannot guarantee this hash function exists, we should use more than one function (k > 1). For this reason we will need m bits to store all the n elements, so the space complexity will be O(m).

3. **False positive probability:** We can estimate the false positive probability of a item to really be inside the bloom filter with the formula: $p = (1 - e^{-kn/m})^k$ in where k is the number of hashing functions, n the number of items inserted into the structure, and m is the size in bits o the structure.
4. **Filter size:** Given the number of objects expected to insert (n),and the expected false positive probability(p), we can calculate the filter size by applying the formula: $m = -(n * log(p))/pow(log(2), 2)$
5. **Number of hashes:** Given the filter size (m) and the number of expected objects to be inserted (n), we can calculate an optimal number of hash function following this formula: $k = (m/n) * log(2)$

- **Empirical results:** Empirically our bloom filter offers an amazing performance storing and searching a huge amount of data very fast.

  For example, for storing 1.000.000.000 objects we will need a filter of size 3.847.396.112 bits and using 2 hashes with a false positive rate of 0.02 (2%). The amount of memory required is around 3,59GB.

  Our test computer, with 16GB of ram and a processor Intel(R) Core(TM) i7-4770HQ CPU @ 2.20GHz, took around 123 seconds to sequentially insert 1.000.000.000 objects and around the same amount of time to make 1.000.000.000 sequential reads.

  (This benchmark is given as an option in the main menu)

Comparison between AVL Trees and Bloom Filters

- **Storage of objects:**
  - AVL Trees stores the actual data inside the node or a pointer to the position in memory of the data
  - Bloom Filters does not store the data, it only suggests whether the data is present or not.
- **Space utilization:**
  - AVL Trees are not space-efficient as they store directly all the data in the node or a pointer to the actual data.
  - Bloom filter is space-efficient as they store only an array of bits depending on the number of hashes, the probability of false positives intended, and also the size of the data.
- **Deletion operation:**
  - On AVL Trees the deletion operation is possible.
  - On the Bloom filters, the deletion operation is not possible or recommended, because of the possible collisions between hash functions. If we set a bit to 0 that was referenced by other hash functions while inserting other objects in the filter, we are deleting all those objects too without knowing which ones.
- **Accuracy of results:**
  - AVL Trees gives accurate results
  - Bloom filters have a false positive probability, but this probability can be manipulated to be very small by instantiating a more large filter and/or adding more hash functions.
- **Collision handling:**
  - On AVL Trees there are no collisions.
  - Bloom filters use multiple hashing functions to reduce the probability of collisions
- **Time and space complexity:**
  - On AVL Trees for the operations of insertion, deletion and search the time complexity is O(log n), and the space complexity is O(n)
  - On Bloom filters for the operations of insertion and search, the time complexity is O(k), but as K is constant in any instance we can reduce it to O(1). On the other hand, the space complexity is O(m) in where m is the number of bits used.
- **Uses:**
  - AVL Trees are used when we need to search efficiently for data in memory and we need to use that data for something, and we are not doing many inserts or deletes. The real-world applications will be when we need to load at the beginning of a service a database, thus making an on memory database. Also, they can be used as the base data structure for hashmaps in some implementation of a programming language.
  - The most common use cases of Bloom Filters are when you need to replicate a large amount of data between different machines and we don't need the data per se but to know the existence or absence of that data. In this case, sending a stream of bits is more cost-effective than sending all the data. The real-world applications are in network routers, in web browsers (to detect the malicious URLs), in existing username checkers, in password checkers, etc.

References

- AVL Trees: Rotations, Insertion, Deletion with C++ Example. Accessed on: May 3st 2021, in https://www.guru99.com/avl-tree.html
- AVL Tree. Accessed on: March 24st 2021, in https://www.programiz.com/dsa/avl-tree
- AVL Trees. Accesed on: March 24st 2021, in http://pages.cs.wisc.edu/~ealexand/cs367/NOTES/AVL-Trees/index.html
- AVL tree. Accessed on: April 10st 2021, in https://en.wikipedia.org/wiki/AVL_tree
- Random numbers. Accessed on: May 9st 2021, in https://www.ugr.es/~jsalinas/Aleatorios.htm
- Bloom Filters by Example. Acessed on: April 4st 2021, in https://llimllib.github.io/bloomfilter-tutorial/
- Bloom filter. Acessed on: April 4st 2021, in https://en.wikipedia.org/wiki/Bloom_filter
- Array of bits. Acessed on: April 4st 2021, in http://www.mathcs.emory.edu/~cheung/Courses/255/Syllabus/1-C-intro/bit-array.html
- Jenkins hash function. Acessed on: April 4st 2021, in https://en.wikipedia.org/wiki/Jenkins_hash_function
- PeterScott / murmur3. Acessed on: April 4st 2021, in https://github.com/PeterScott/murmur3/blob/master/murmur3.c
- Bloom filter calculator. Acessed on: April 10st 2021, in https://hur.st/bloomfilter/?n=12&p=&m=12&k=2
- Bloom Filters - the math. Acessed on: April 10st 2021, in http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html