* **Design issues:**

Our tasks:

Design and implement an algorithm that searches a collection of documents and a set of sample queries:

* We will process the documents and store their content (i.e. words / tokens) in the data structures that we selected (in information retrieval, this phase is called indexing).
* Next, for every input query, we will process the query and search its keywords in the documents; using the previously implemented data structures and an algorithm of our choice (the phase is called retrieval). For each such query, we will have to display the documents that satisfy the query.

After discussing, we chose Trie as the data structures and algorithms that we consider to be more efficient for this task as the reasons below:

* **Data structures - Reason for decision:**

Trie: In computer science, a trie, also called digital tree and sometimes radix tree or prefix tree (as they can be searched by prefixes), is an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. Values are normally not associated with every node, only with leaves and some inner nodes that correspond to keys of interest. For the space-optimized presentation of prefix tree, see compact prefix tree.

In the example shown, keys are listed in the nodes and values below them. Each complete English word has an arbitrary integer value associated with it. A trie can be seen as a deterministic finite automaton without loops. Each finite language is generated by a trie automaton, and each trie can be compressed into a DAFSA.

It is not necessary for keys to be explicitly stored in nodes. (In the figure, words are shown only to illustrate how the trie works.)

Though tries are most commonly keyed by character strings, they don't need to be. The same algorithms can easily be adapted to serve similar functions of ordered lists of any construct, e.g., permutations on a list of digits or shapes. In particular, a bitwise trie is keyed on the individual bits making up a short, fixed size of bits such as an integer number or memory address.

A Trie (so called a prefix tree), could be a good data structure for building a memory-efficient dictionary with fast lookups, auto completion.

A trie has a number of advantages over binary search trees. A trie can also be used to replace a hash table, over which it has the following advantages:

* Looking up data in a trie is faster in the worst case, O(N) time (where m is the length of a search string), compared to an imperfect hash table. An imperfect hash table can have key collisions. A key collision is the hash function mapping of different keys to the same position in a hash table. The worst-case lookup speed in an imperfect hash table is O(N) time, but far more typically is O(1), with O(m) time spent evaluating the hash.
* There are no collisions of different keys in a trie.
* Buckets in a trie, which are analogous to hash table buckets that store key collisions, are necessary only if a single key is associated with more than one value.
* There is no need to provide a hash function or to change hash functions as more keys are added to a trie.
* A trie can provide an alphabetical ordering of the entries by key.

*For more briefly specific keys:*

Advantages of Trie over Binary Search Tree:

* Looking up of keys is faster. Worst case scenario a key of length m takes O(m) time. BST performs search by doing O(log(n)) comparisons. In worst case BST takes O(m log n) time. Simple operations that tries use during lookup such as array indexing using a character are fast on real machines
* Tries are more space efficient when they contain a large number of short keys, because nodes are shared between keys with common initial sub-sequences.
* Tries facilitating longest-prefix matching, helping to find the key sharing the longest possible prefix of characters all unique.
* The number of internal nodes from root to leaf equals the length of the key. Balancing the tree is therefore no concern.

Advantages of Trie over Hash Tables:

* Tries support ordered iteration, whereas iteration over a hash table will result in a pseudorandom order given by the hash function (also, the order of hash collisions is implementation defined), which is usually meaningless
* Tries facilitate longest-prefix matching, but hashing does not, as a consequence of the above. Performing such a "closest fit" find can, depending on implementation, be as quick as an exact find.
* Tries tend to be faster on average at insertion than hash tables because hash tables must rebuild their index when it becomes full - a very expensive operation. Tries therefore have much better bounded worst case time costs, which is important for latency sensitive programs.
* By avoiding the hash function, tries are generally faster than hash tables for small keys like integers and pointers.

Nevertheless, tries do have some drawbacks/ cons as well:

* Tries can be slower in some cases than hash tables for looking up data, especially if the data is directly accessed on a hard disk drive or some other secondary storage device where the random-access time is high compared to main memory.
* Some keys, such as floating point numbers, can lead to long chains and prefixes that are not particularly meaningful. Nevertheless a bitwise trie can handle standard IEEE single and double format floating point numbers.
* Some tries can require more space than a hash table, as memory may be allocated for each character in the search string, rather than a single chunk of memory for the whole entry, as in most hash tables.
* **Algorithms we employ:**

Storing system:

Search engine: we develop step by step: search by 2 words, by 3 words; AND query, OR query and so on.

For more specific:

Create an overall search function - this will check if the word exists in the whole trie:

bool \* Tree::Search\_Node(string target,int length, int count);

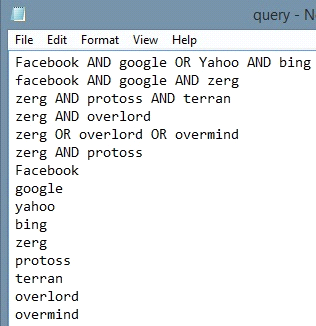
This bool function will return an array of bool, which will contain the result of that word by having the containing file number mark as true and default as false in an array of bools.

To optimize this further, we don't want to check an empty array full of false results so if the array has no slot marked as true the function will return NULL. The main reason behind this decision it's that it gives us an array of result which will benefit the following function which is reading query and process it.

Consequently, the “queries” function will take advantage of the result of the bool function; if it is true, the result is an array of bool in which there are the numbers of files containing the word.

Into the bargain, the following steps are our remarks on design and implementation of this project.

At the outset, we were facing the problem of complexity of the code of the searching process due to our thought that we had to divide into as many cases as possible. For instance:



1st case: A AND B

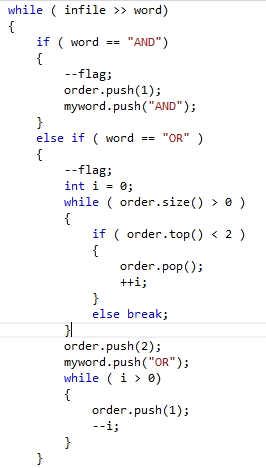
2nd case: A OR B

3rd case: A AND B OR C

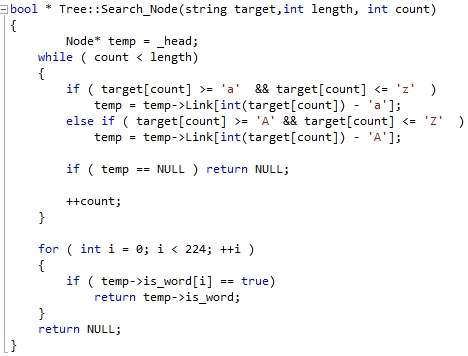
And so on …

This had led to the complexity in our code. Howbeit, we did try to manage to find another way to solve this problem. As a result, coming up with an ideal, we made a breakthrough to simplify our program.

Applying the theory of stack and queue enables us to “push”, “pop” … the queries regardless of the order of coming words.

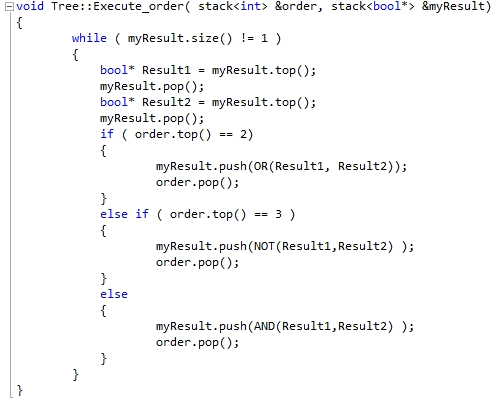


Basically, we will put the words into a stack, and the key searching (“AND”, “OR”, “NOT”) into another one in the process of reading. Thereafter, the program will “pop” them out and start searching in the files.

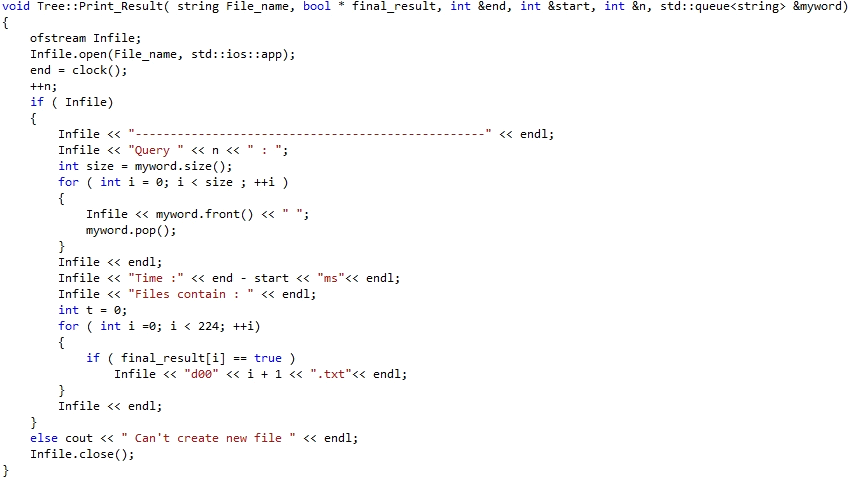


The idea may not be perfect but it's an idea after all. By putting the orders ("AND", "OR", "NOT") and the results of the searching's function in 2 different stacks. We can somewhat apply the Polish notation but not exactly. The way it works is execute all the "AND" on top and the stack will execute the remaining "OR" on top follow by "NOT" order. The "OR" and "NOT" orders will be execute AFTER all the "AND" in the query is gone.

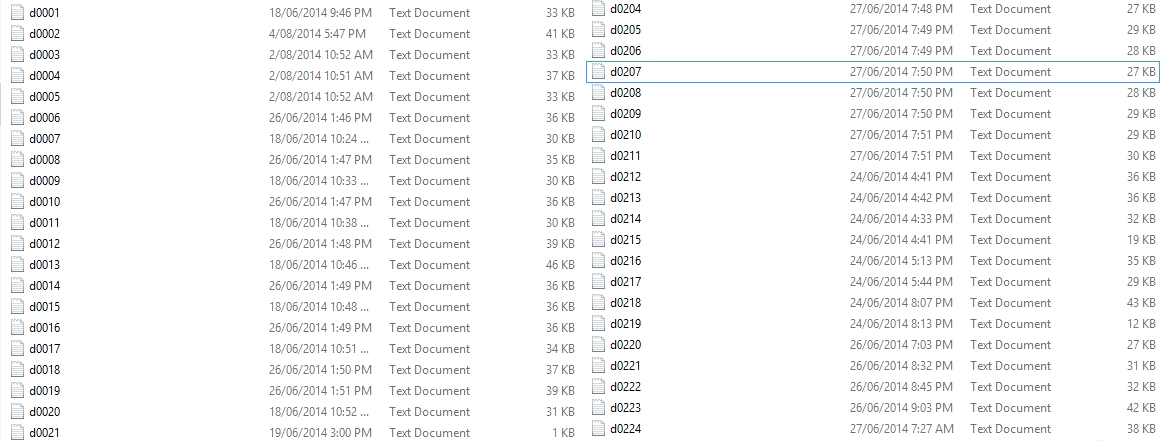
Moving forward to the executing process, the program will compare the result of the bool function above:



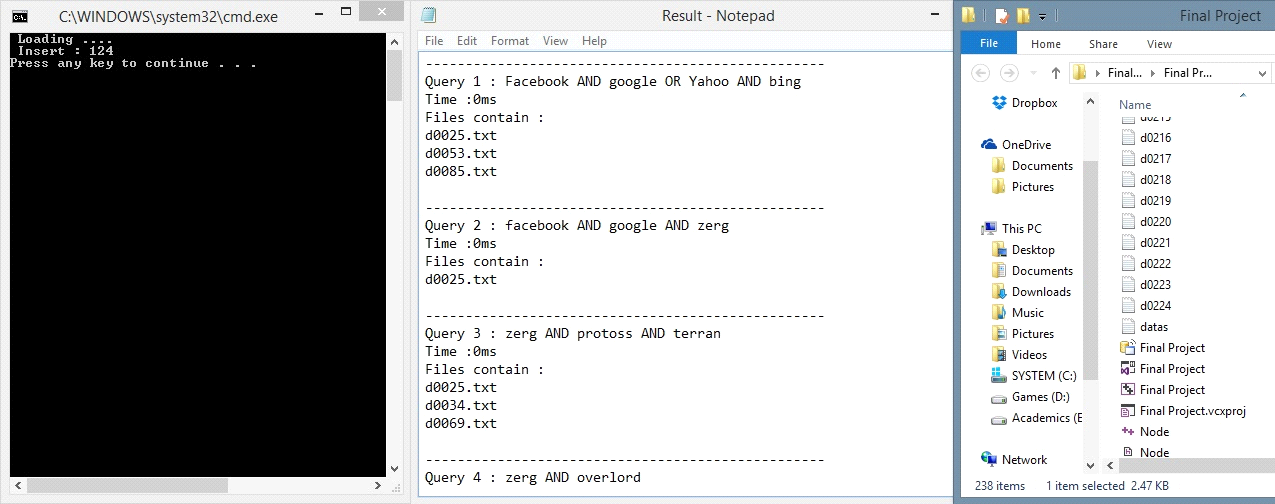
Eventually, depending on the result of the executing process, it will print out to the file:



This is our input file:



This is the result after compiling the whole program:



Inserting Time: 124 ms.

Searching Time: 0 ms each query.

* **Optimization:**
  + Insertion: We are not quite satisfied with how with read the files. We basically used binary reading and store everything in an array. We would like to optimize how we process that array because our way might not be perfect.
  + Searching: We haven't done partial search, so what we would like to improve on this specific subject is to implement half-match searching.
  + Reading queries: This part we would like to look in further but because we ran out of time. Our reading technique is flawed because we read word after word and if there're 2 words next to each other it will process everything before the 2nd word which lead to the fact that problems may occur if the form of queries isn't correct.
* **Scalability:**
* For a very large text file so far, we haven't had any problem running 224 text files and the pre-processing time is quite fast.
* Additionally, the preparation has to be perfect. You need to create a datas.txt file that stores all the names of the files which are to read and I have implement the code to increase the bool size for the result's array. We consider this as a tremendous weakness which may need to be enhanced.
* All and all if the preparation is in place the program seem to run quite nice and yes we do delete after the program so there shouldn't be any memory leaking or that sort of stuff.