Group 5 – Project Report

1. **Simple search 1**
2. Constructing the b–tree 1
3. Original way 1
4. Improvement 2
5. Storing word count 2
6. Storing word count

and word positions 2

1. Problems and possible solutions 2
2. Searching 2
3. Single word 2
4. Minus sign (–) 3
5. Intitle 3
6. Plus sign (+) 3
7. Range 3
8. Filetype, price, and hashtags 4
9. Stopword 4
10. Synonym 4
11. **File numbering system 4**
12. Original way 4
13. Improvement 5
14. **Phrases, wildcards, and unknown words 5**
15. Preparing the suffix tree 5
16. Suffix tree 5
17. Constructing the suffix tree 5
18. Saving the suffix tree to disk 5
19. Deleting the suffix tree 6
20. Searching 6
21. Searching for exact match of

a phrase with no wildcards

or unknown words 6

1. Searching for exact match of

a phrase with wildcards

but no unknown words 6

1. Searching for exact match of

a phrase with wildcards

and unknown words 7

1. Possible improvement 7
2. **History and autocomplete 7**
3. Definition 7
4. Storage 8
5. Usage 8
6. Build tree 8
7. History and Auto–complete

recommendation 9

1. **Overall search process 9**
2. File filtering 9
3. File scoring and ranking 9
4. File highlighting 10
5. **(Reverse) Polish notation 10**
6. Definition 10
7. Usage 10
8. **Possible general improvements 11**
9. **Running time 11**

✪✪✪

1. SIMPLE SEARCH

* For most of our search functionality, we use the B–tree because it is one of the most efficient data structures to store and access data on disk and is designed to be very scalable, used in many databases and filesystems.
* Currently, the minimum degree of the B–trees of our group is **15**. This number is chosen to be larger than **11.5**, the fifth root of **200000**. The reason is that there are about **200000** words in the English language and the average word length is **5.1**. So if we use a trie (a common data structure for searching), on average we will need **5.1** file reads for each search. Because on large scale it is the number of file reads that matter, to be able to beat the trie approach we would need less than **5.1** files reads for each search, for example **5**. So roughly, the minimum degree of each node would need to be at least the fifth root of **200000**. To be safe, we pick **15**.
* There may be more sophisticated ways to pick the minimum degree.

1. **CONSTRUCTING THE B–TREE**
2. **Original way**

* We construct the B–tree based on the algorithms in “Introduction to Algorithms” by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
* The B–tree is stored and used in disk. That is, we do not reconstruct it each time we use it and write it to disk again after we are finished but instead each time we only read and write the nodes we need.
* Each node has an index that is given to it when it is created by an increasing int **curInd**.
* Originally, each node is written to a separate file, whose name is the same as the node’s index. The downside is that even if we set the minimum degree of the B–tree to 50, each node’s file will only use about 10% of the minimum file size so on average we only really need 10% of the disk space we use, which is very wasteful.

1. **Improvement**

* We improve this by putting the entire B–tree into a single binary file. We still give each node an index as before but now we use the index in a slightly different way. Because the size of each node is fixed and we are writing to a binary file so the size of each node when written to the file is fixed. Therefore from the index of the node we can find its position in the file as **sizeOfNode \* index**.
* When the key or value is a string, to make sure that the size of the node when written to file stay constant, we use a structure **fixedStr** which is just a fixed sized string. The downside of this is when we read in a word that is longer than the size of **fixedStr** which is **32** in our case. Because a word may have arbitrary length so we cannot keep increasing the size of **fixedStr**.
* Therefore, one possible solution is that we do not insert the long word into our word B–tree and when we need to search for it, we use the suffix tree. Another solution is to add more character delimiters to split the long word, such as using the hyphen – and the slash / to separate <https://www.thesun.co.uk/news/6702779/spurs-owner-yacht-joined-by-travelex-founder-river-thames/> into multiple words.

1. **Storing word count**

* We store the word counts so that we do not have to open the news files again every time we search. Instead, we only open the word count file once and use it for multiple searches.
* Originally, we store the word counts of every word in every news file in a value file. We store this information using linked lists. Each linked list will contain the word counts of a word and each node in the linked list will contain the word count of the word in a news file. In the B–tree, in the node where the word is a **key**, the corresponding value will be the position of the head of the linked list containing the word counts of the word.
* So if we want to know the word count of a word in a news file, we look for that word as **key** in the B–tree, we use the corresponding value to jump to the head of the linked list and then we traverse the linked list until we reach the node corresponding to the news file.

1. **Storing word count and word positions**

* Again, we store the word positions so that we do not have to open the news files again.
* We still use the linked lists in part (A.I.3) but now, instead of just the word count, each node in the linked list will contain the word count and the word positions of the word in the news file.
* Each node will have a field called **tempValue**. When we insert the words of a news file into the B–tree, we do it in 2 times. The first time, we set the **tempValue** of each word we encounter to point to a position in a newly created temp file, where we keep track of how many times we have seen the word. The second time, we go through all the words of the file again. Because we know how many times a word appears in the news file, the first time we meet that word we can allocate just enough space for it in the value file and when we meet that word again we can continue filling in the word positions. This way we do not have to fill in all the word positions of a word before moving on to the next word.

1. **Problems and possible solutions**

* The problem with the above design is that stopwords appear very often and in almost every file. So when we insert a stopword of the 2000th file into the B–tree, we will have to traverse through 1999 nodes in its linked lists corresponding to the 1999 files it has appeared in, which makes it very slow. Our current solution is that we do not store information related to stopwords in the B–tree and we assume that stopwords appear in every news file, which is a somewhat reasonable assumption. And when we have found a news file to highlight, we simply go through that news file for the word count and word positions of stopwords.
* Of course, this assumption is not always right. So a possible improvement is to not store the word counts and word positions of a word as a linked list but as another B–tree. So we will have 2 layers of B–tree. To search for the word count and word positions of a word in a news file, in the first layer, we search for the word as key and the corresponding value will be the position of the root node of the B–tree containing the information related to the word. We look in this second B–tree for the news file as key and the corresponding value will be the position in the value file containing the word count and word positions of the word in the news file.
* This design is more complicated but it will be significantly faster, more scalable, and we will be able to deal with stopwords more properly.

**II. SEARCHING**

1. **Single word**

* To search for the news files a word appear in, we look in the B–tree for the word as key and the corresponding value is the position of the head of the word’s linked list in the value file. We traverse the linked list where each node is a news file the word appears in.
* To get the positions of the word in a news file, we stop at the node in the linked list corresponding to that news file and read in the word positions of the word in the news file.

1. **Minus sign (–)**

* To deal with the minus sign (–), we simply search for all the news files the word appears in and put in a bool array. Then we reverse the array, if **a[i] = true** then **a[i] = false** and vice versa. This way we can find the news files the word does not appear in and we can generalize this to not just a single word after the minus sign (–) but a complicated query surrounded by parentheses.

1. **Intitle**

* To search for the news files where the word is in the title, we first construct a file containing the paragraph positions of every text files. The procedure is similar to how we construct the wordPos file in part (C.II.3b). Then when we traverse the linked list of the word, at each node we read in the first position of the word in the news file. If this position is smaller than the starting position of the second paragraph of the news file than the word is in the title (paragraphs are separated by line feeds so the title can be considered to be a paragraph).

1. **Plus sign (+)**

* Like we have mentioned in part (A.I.5), we currently do not store information related to stopwords and we assume that stopwords appear in every file. When we highlight a news file, if there is the plus sign (+) before a stopword then we go through that news file to find the positions of the stopword. For further discussion, read part (A.I.5).

1. **Range**

* To search within a range of numbers, we have a separate B–tree that only holds numbers. We search recursively. For example to search for information related to numbers between 10 and 100 we set the current node to the root node. Every key in the current node between 10 and 100 will be returned. We then go to every child node of the current node that might be within the range one at a time and set that node as the current node. A child node that cannot fit the range is a node we know contains value smaller than **x** and **x** is smaller than **10** or larger than **x** and **x** is larger than **100**. (We can know from the key–child structure of the B–tree).

1. **Filetype, price, and hashtags**

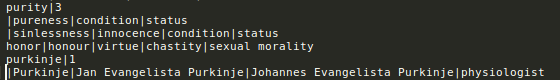
* Because we only process .txt files, we do not support the filetype feature. But one possible way is to have a separate B–tree where the news file is the key and the file type is the value.
* We do not treat price and hashtags differently. We simply consider the **$** sign and the number after to be a word and the **#** and what comes after to be a word.

1. **Stopword**
2. **Idea:** load words into a btree whose node contains only the word itself as key. To check if a word is a stopword, we check for its existence in the tree and return a boolean.
3. **Implementation:**

* Create a txt file containing stopwords.
* Alter the btree node to only contain key
* Load stopwords into the tree, one by one.
* Store the current root node id for later retrievals.

1. **Synonym**
2. **Idea:** load words into a btree whose node contains the word as key and a kind of pointer pointing to a synonym data file to retrieve an array of synonyms.
3. **Implementation:**

* Download a free synonym data file at [http://lingucomponent.openoffice.org/MyThes–1.zip](http://lingucomponent.openoffice.org/MyThes-1.zip), we are only working with the file **th\_en\_US\_new.dat** .
* Re–process the data file into the format that we want:



Which can be interpreted as:

key|n (number of lines of synonym sets)

|set 1

|set 2

…

|set n

* Alter the basic btree node to contain the word as key for searching and an integer representing the position in the file to start reading in the synonyms.
* Store the current root node id for later retrievals.
* Use the returned synonyms as keywords for new queries.

1. FILE NUMBERING SYSTEM
2. **ORIGINAL WAY**

* Originally, we only worked on the news files of our group, whose addresses are of the form rootPath + “Group05NewsXX.txt” where rootPath is an adjustable string constant holding the address of the folder containing the news files and XX is a number from 00 to 99. We can see that the only thing that varies here is XX so to store the address of a file we only need to store an integer for XX and from the integer we can reconstruct the address of the file. And so every reference to a news file will be one single number. The benefit is that it is much simpler, faster and takes much less space to store and pass an integer around rather than a string.

1. **IMPROVEMENT**

* But we now need to process the files of other groups. One possible solution is that instead of using one single integer to refer to each file, we use two integers, one for the group number and one for the file number of the group. But the problem with this method is that it is not very general, and we can only process files whose addresses are of the form rootPath + “GroupXXNewsYY.txt”.
* So the solution of our group is that we construct a file numbering system with the help of a B-tree. Each time we insert a new file, we give it a number and from then on we only refer to that file by that number. We insert the file’s address into the B-tree with the file’s number as key and the file’s address as value. Then, when we need the file’s address, we search the B-tree for the file’s number as key and the corresponding value will be the file’s address.
* So now we can process files with arbitrary addresses and file’s names while internally, we still only need one integer to refer to each file.

1. PHRASES, WILDCARDS, AND UNKNOWN WORDS

* To deal with the problem of searching for an exact match of a phrase, including wildcards and unknown words, we use the suffix tree. We use this tree because according to our research, this tree is one of the most efficient ways to accomplish these tasks.

1. **PREPARING THE SUFFIX TREE**
2. **Suffix tree**

* Basically, a suffix tree of a string **T** is a trie that holds all the suffixes of the string. Each leaf node corresponding to a suffix will hold the starting position of that suffix in the string.
* This is useful because once you have constructed the suffix tree, to determine if a string **S** of length **m** is a substring of **T**, we need only walk down the suffix tree at most **m – 1** timesfor each character in **S**. This will give the correct answer because if **S** is a substring of **T** then **S** will be the prefix of a suffix of **T**. So if we have walked down **m** characters of **S** in the suffix tree then whatever suffix of **T** we can get by traversing down the suffix tree to the end will have **S** as the prefix. Therefore, **S** is a substring of **T**. And if we cannot walk down all **m** characters of **S** then **S** is not a prefix of any suffix of **T**. That is, **S** is not a substring of **T**.

1. **Constructing the suffix tree**

* Because we only accept the **256** ASCII characters for our suffix tree (because using a larger character set would require a lot more memory and will only be useful for few use cases) and we reserve the **`** character for a special purpose in the suffix tree, we process the news files to delete non–ASCII characters and **`** before we begin.
* To construct the suffix tree we use Ukkonen’s algorithm, a very efficient algorithm that can construct the suffix tree of a string of length n in **O(n)** whereas a naïve algorithm would need **O(n2)**. Our exact implementation is based on <https://www.geeksforgeeks.org/ukkonens–suffix–tree–construction–part–6/>, with a little bit of adjustment.

1. **Saving the suffix tree to disk**

* After constructing the suffix tree, we save it to disk so that we can reuse it later without having to reconstruct it.

1. **Original way**

* Originally, we save each node of the suffix tree to a file. Each node will be given an index – which is also the name of the node’s file – and instead of writing out the pointer to a child node, we write out that child node’s index, which is equivalent. We write out the tree recursively. At each node we first go through its children, giving them indexes and writing them to file, then we write the current node to file, replacing node pointers with node indexes.
* We write the root node’s index to a **metafile**.
* The downside of this method is that for each tree there will be about a thousand files corresponding to a thousand nodes but each file will need only about **10** bytes. But because the minimum file size of Windows is **4** kilobytes, each file will still take **4** kilobytes, meaning that we only really need less than 1% of the disk space we use, which is very wasteful.

1. **First improvement**

* Instead of writing each node to a file, we write all the nodes to one file. Instead of writing out a child node’s index, we now write out that child’s node position in the file. We also write out the tree recursively. At each node we first go through its children, writing each of them to the end of the file and storing their positions. We then write the current node to the end of the file, replacing node indexes with node positions in the file. We write the root node’s position to a metafile.
* The downside of this method is that it is still not efficient enough. For **2000** news file, there will be **2000** suffix tree files and each time we search for a phrase, we will have to open and close each of these **2000** files.

1. **Second improvement**

* We now write many suffix trees to one file. There is a const **numSTreeInFile** – the number of suffix trees in one file – which we can adjust. There is not much difference compared to part (C.I.3b). Instead of creating a new file for each suffix tree, we only create a new file when the current file has reached its limit. Otherwise, we just write the current suffix tree to the end of the file and each node still holds the positions of its children. We still write the position of the root node to a **metafile**.
* Because we number the news file that we load in increasing order, we can easily find which suffix tree file the suffix tree of a news file belong to by integer dividing it by **numSTreeInFile**. For example, if **numSTreeInFile = 10** then the suffix tree of news file **97** will be in the 97 / 10 = 9th suffix file.

1. **Deleting the suffix tree**

* We delete the tree recursively, similar to how we would delete a binary tree.

1. **SEARCHING**

* We use the suffix tree directly on the disk, reading only the nodes we need, not the entire tree every time we need to use the suffix tree.

1. **Searching for exact match of a phrase with no wildcards or unknown words**

* To determine if a string **S** of length **m** is in a news file **T**, we walk down the suffix tree of the news file corresponding to the characters of **S**. If we can walk down all **m** characters then **S** is in **T**, otherwise it is not.
* To search for all appearances of **S** in **T**, after we have walked down all **m** characters and arrived at a node, we traverse the descendants of the node and each time we reach a leaf node corresponding to a suffix of **T** that **S** is a substring of, the starting position of this suffix is also a starting position of **S**.

1. **Searching for exact match of a phrase with wildcards but no unknown words**

* Each word will look like this: **in\*red\*\*le**
* We search for positions of longest substrings of the word without \*, in this case “in”, “red”, and “le”. Then we subtract from these positions the starting positions of the corresponding substring in **S**.
* For example, “in” appears in **S** at 0 so if it appears in **T** at 1 and 3, we subtract by 0 to get 1 and 3; “red” appears in **S** at 3 so if it appears in **T** at 7 and 12, we subtract by 3 to get 4 and 9. Basically, what we are doing is that because we know this substring of the word appears at this position 7 then by subtracting the position of the substring in the word, we know the word might appears at this potential position 4.
* We look for the positions in **T** that qualifies for each substring of the word (by using an int array to count how many times a position is considered “potential” and choosing the positions that have the count equal to the number of substrings in the word).

1. **Searching for exact match of a phrase with wildcards and unknown words**
2. **Searching**

* The phrase essentially looks like this: **word1 word2 \* word3 \* word4…** where word1, word2,… might contain wildcards.
* We expand on the idea in part (C.II.2):
* We first differentiate between the two types of \*, the \* that stand between two delimiters stands for a word with varying length (the 1st \*), the other \* stands for a single character in a word (the 2nd \*). We search for positions of longest substrings of the word without the 1st \*, in this case word1 word2, word3, word4,… by using the method in part (C.II.2).
* Then, we change the positions from character position to word position.
* For example, the substring **word1** **word2** appears at the 50th character in the news file, corresponding to the 10th word in the news file. Then, like in part (C.II.2), we subtract by the word position of the substring in **S**. For example, the word position of **word1 word2** in **S** is 0 so we subtracts the word positions of **word1 word2** in T by 0, the word position of **word3** in **S** is 3 so we subtracts the word positions of **word3** in **T** by 3,… Then like in part (C.II.2), we look for the word positions in **T** that qualifies for each substring of the phrase. Then, we change these word positions back to character positions.

1. **Word position**

* To switch between character position and word position quickly, for each file we store the starting position of every word.
* Originally, we store the word positions of each news file in a separate file, with **– 1** at the end.
* The first improvement is to switch from text file to binary file with a word count of the file at the beginning. This way we can read in the information with two reads, one to read in the word count and one to read in the rest.
* The problem is that to search for a phrase in 2000 news files, we will have to open 2000 **wordPos** files. So the second improvement is to store the word position information of every news file into a single **wordPos** file. Each time we add a news file, we just write the word count and word positions of this news file at the end of the **wordPos** file. We will use a B–tree to find where the word positions of a news files is in the **wordPos** file, the key is the number of the news file and the value is the position in the **wordPos** file.

1. **POSSIBLE IMPROVEMENT**

* The suffix tree can be expanded so that \* does not mean just one but possibly multiple characters or multiple words. This will be more complicated and more likely to give incorrect results but probably closer to what the user needs. To make sure that it does not give too wrong results, we can limit the number of characters or words that \* means or give lower scores if \* contains too many characters or words.

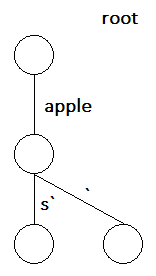
1. HISTORY AND AUTOCOMPLETE

* In our CS163 final project, we use Radix Tree to store History and print on the screen the autocomplete–recommendation.

1. **DEFINITION**

* According to Wikipedia: “In computer science, a **radix tree** (also **radix trie** or **compact prefix tree**) is a data structure that represents a space–optimized trie in which each node that is the only child is merged with its parent.”

1. **STORAGE**

* A radix tree can have at most **r** children, where **r** in our project is **256** since it is the total ASCII code.
* Unlike in regular tries, edges can be labeled with **sequences of elements** as well as **single elements**. This makes radix trees much more efficient for small sets (especially if the strings are long) and for sets of strings that share long prefixes.
* Each Radix Tree node will consist of 3 elements:
* its **r** children pointers.
* a string containing the path from its parent.
* a boolean type parameter to indicate whether it is a “leaf node” or not
* remember that the word ‘leaf’ doesn’t indicate that this node doesn’t have any children but rather means it is the end of a string it stores. (But in our implementation, we add ` to the end of each word when we insert so no word will be a prefix of another word and a leaf node doesn’t have any children)
* Example: if you input “apple” and “apples”, our function will 1st rewrite the queries into “apple`” and “apples`” then insert them into the tree.

Here in the picture beside, you can see that whichever node contains **`** is the leaf node.

* One more important thing to remember is that the path of the root node is empty.

1. **USAGE**

(In the following comparisons, it is assumed that the queries are of length **k** and the data structure contains **n** members)

* Unlike balanced trees, radix trees permit lookup, insertion, and deletion in **O(k)** time rather than **O(log n)**. Even though this looks like a disadvantage since **k** ≥ **log n**, the worst–case for comparison of radix tree is only **O(k)** in total whereas that of other types of balances tree is **O(k\*log n)**.
* Even though the disadvantage of radix tree is that it can only be applied to strings of elements, it is suitable for our group’s storing purpose.
* Although we need to check whether the input word has already in the tree or not, we can merge the two functions SEARCH and INSERT into one function.

1. **Build tree**
2. **Original way**

* We will 1st check if the root is NULL; if it equals NULL, we will create a new node. Next, we will declare two variables **cur** – equals root – and **curPos­** – equals 0 to track the current letter in the string. Then, we will jump to a **while** loop that only ends when it complete the insertion or the **query** has already existed.
* In the **while** loop:
* If **curPos** equals the length of query – which means we has reached the end of the query and the query has already been inserted – we will escape the insertion function. Otherwise we 1st declared an integer **ind** that store the ASCII code of the character at **curPos** in the query.
* Next, if the child in **ind** is empty, we will declare that child a new node and store the substring in the child’s path. The substring goes from **curPos** to **query.length() – curPos**.
* If the child is not empty, we will store that child’s path into a string **tempStr**. We declared an integer **i** with value zero to track the current position of in **tempStr**. The **while** loop will go on, and the two integers **i** and **curPos** will increase until either reaches the end of one of the strings or the character within the two strings aren’t the same.
* If **i** equals **tempStr.length()**, we will continue the big **while** loop with **cur** now being its **ind** child. If **i** doesn’t equal **tempStr.length()**, we will declare **temp** as a temporary node to hold **cur–>child[ind]** and **cur->child[ind]** now is not that same node but rather a new node whose path stores a substring of **tempStr** from 0 to **i**. And path of **temp** which is the old **cur->child[ind]** now stores a substring of **tempStr** from **i** to the end.
* For better understanding, read the void **insertTree** function in Tree.cpp, line 662.

1. **File storage**

* The idea of storing the radix tree in one file is the same as storing the suffix tree.

1. **History and Auto–complete recommendation**

* We continue to check the path of each node until we reaches the end of the **query** or reaches the end of an actual ‘leaf’ node (means that this node has no children).
* If we reach the end of the **query**, we will call another function that traverses through all the remains children nodes and print to console when reaches a leaf node.
* If we reach the end of an actual ‘leaf’ node while still in the middle of **query** which means there is no string that satisfies and no recommendation is being printed.
* If the user chooses to search a query from history then we can instantly print out the results instead of calculating everything again.

1. OVERALL SEARCH PROCESS

* If the query does not exist in history then once we have entered the searching phase, we divide our overall searching process into 3 steps:

1. **FILE FILTERING**

* In the first step, we use the reverse Polish notation in combination with our functionalities to choose the news files that qualify for our query.

1. **FILE SCORING AND RANKING**

* In the second step, we score the news files that qualify. Our group’s current implementation is that the score of a paragraph in a file is equal to the square of the number of keywords in that paragraph divide by the length of the paragraph. That is, **((numKeyWord / paragraphLength) ^ 2).** The score of the news file will be the sum of the scores of the paragraphs.
* By dividing the number of keywords in a paragraph by the length of that paragraph, we are avoiding the cases where the paragraph is very long and therefore more likely to contain many keywords but the concentration is not very high. By squaring this quotient, we are focusing on the case where many keywords concentrate in one paragraph rather than spread out evenly over the entire news file, making it more likely that the keywords are related.
* We can adjust the formula further, changing the powers of the components. For example, the score of a paragraph can be **((numKeyWord ^ 2) / paragraphLength) ^ 3)**, depending on what we need.
* A weakness of the current design is that we are not taking the order of the keywords into account. For example, “big green apple” and “apple green big” is treated the same although they may mean different things. We can adjust the algorithm to take this into account. However, the current design is simple, and if we prioritize the number of keywords enough (raise to high enough power), our group believes the results will be about the same. The reason is that for normal queries, if a paragraph has a high enough concentration of the keywords then it will likely be relevant to the query, regardless of the order of the keywords.
* One possible improvement is to prioritize the paragraphs that contain a high number of different keywords instead of the same keyword repeated many times. Another possible improvement is to prioritize the news files that contain the keywords in the title, using intitle.
* After scoring the paragraphs, we pick out the news files with the highest scores to show to the users and the number of news files to show can be adjusted. The purpose of the history functionality is to instantly give us the news files to show to the users without calculating everything again.

1. **FILE HIGHLIGHTING**

* When the user chooses a file to highlight, we score it again and pick the paragraphs with the highest scores to highlight and print. Again the number of paragraphs to highlight and print can be adjusted.

1. (REVERSE) POLISH NOTATION

* In order to process the query, we use reverse Polish notation.

1. **DEFINITION**

* Reverse Polish notation (RPN) is a mathematical notation in which operators follow their operands, in contrast to Polish notation in which operators precede their operands. It does not need any parentheses as long as each operator has a fixed number of operands.
* The expression for adding the numbers 1 and 2 is written in reverse Polish notation as **1 2 +** (suf-fix), rather than as **1 + 2** (in-fix). In more complex expressions, the operators still follow their operands, but the operands may themselves be expressions including again operators and their operands. For instance, the expression that would be written in conventional infix notation as: **(5 − 6) × 7** can be written in reverse Polish notation as **5 6 – 7 x**.

1. **USAGE**

* In Computer Science, this notation is convenient for people to calculate the result of an expression by using Stack data structure. This is our implementation step-by-step:
* For example: **“Loc oppa” OR Chipu NOT TungMTP**.
* We get the sub step by step:

+ “Loc oppa” → expression = “Loc oppa”

+ OR → put OR into the stack

+ “Chipu” → expression = “Loc oppa” “Chipu”

+ NOT → take OR out of the stack, put NOT into the stack.

   Expression = “Loc oppa” “Chipu” OR

+ “TungMTP” → expression = “Loc oppa” “Chipu” OR “TungMTP”

+ Finally, the stack hasn’t been empty yet. Pop\_back + update expression until stack is empty

→ expression = “Loc oppa” “Chipu” OR “TungMTP” NOT

* By using an array index from 0 to 2500 (for this example we assume there are 2501 news files numbered from 0 to 2500), we highlight the files which satisfy our expression.
* “Loc oppa” → put an **array\_1[0..2500]**, with **array\_1[i] = 1**

if file i contains “Loc oppa” into the stack

* “Chipu” → put an **array\_2[0..2500]**, with **array\_2[i] = 1**

if file **i** contains “Loc” into the stack

* OR → take **array\_2** out of the stack. Update the **array\_1**

if **array\_1[i] == 1 || array\_2[i] ==1 then array\_1[i] = 1**

* “TungMTP” → put an **array\_3[0..2500]**, with **array\_3[i] = 1**

if file **i** contains “TungMTP” into the stack

* NOT → take array\_3 out of the stack. Update the array\_1

if  **!(array\_1[i] == 1 && array\_3[i] == 0) then array\_1[i] = 0**

* Finally, we use a loop from 0 to 2500 to check if **array\_1[i] == 1** (the files **i** that satisfy our expression)
* In our project, the operand is not limited to words or phrases but can also include number range, synonym, intitle,…

1. POSSIBLE GENERAL IMPROVEMENTS

* If on average the user will search several queries each time, we can load the first few levels of our trees into memory to speed up the process or load more and more as the number of queries increases.
* In general, the user does not care about upper/lowercase (Google is case–insensitive) so we should process everything in lowercase.
* Adjust number processing so that we can support searching for real numbers or numbers formatted with commas (1,000).

1. RUNNING TIME

* Here is the code we use to calculate the running time:

|  |
| --- |
| auto started = std::chrono::high\_resolution\_clock::now();  **// Do something here**  auto done = std::chrono::high\_resolution\_clock::now();  std::cout << std::chrono::duration\_cast<std::chrono::milliseconds>(done - started).count(); |

* After running our project, here is the running time for each case:

+ Search 1000 queries: **9302** ms

+ Search 1 query: **6** ms

+ Insert 1 group’s files: **8707** ms

+ Insert 1 file txt: **708** ms