DS PROJECT REPORT

ASM COPILOT

Teacher: Miss Mubashra Fayyaz (Theory)

Miss Erum Shaheen (Lab)

Group Members: Muhammad Moaaz bin Sajjad (20k-0154)

Saad bin Khalid (20k-0161)

Ayaan Danish (20k-0245)

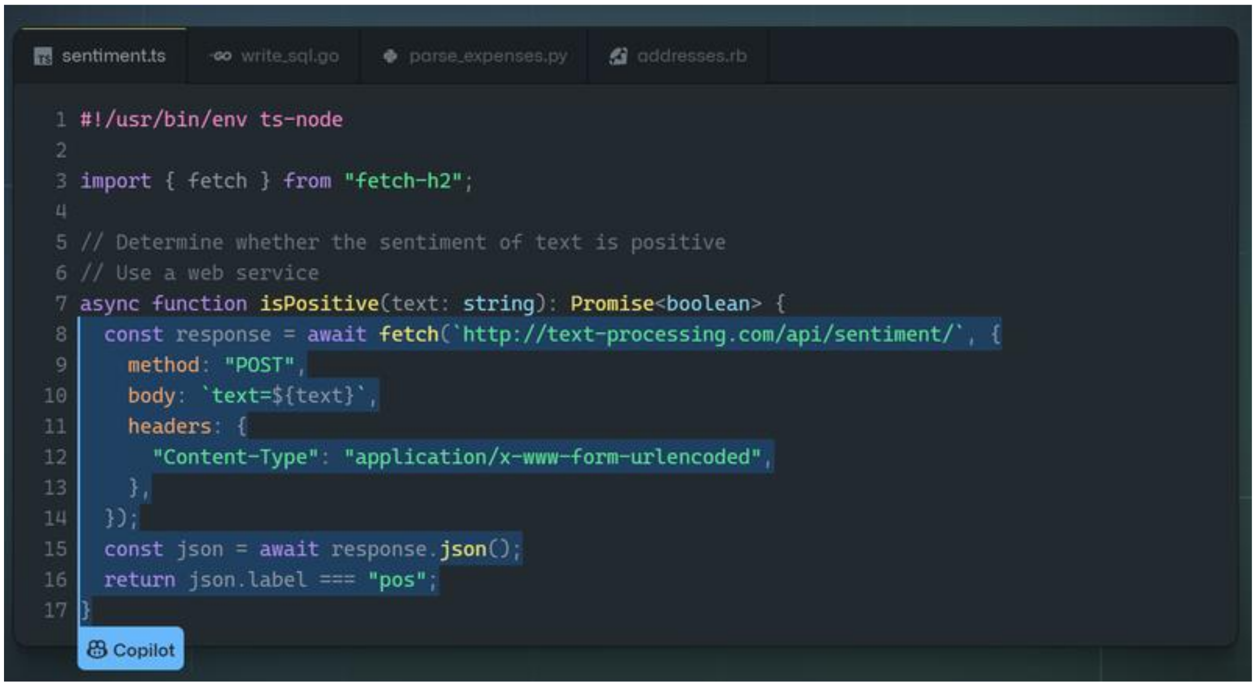
Table of Contents

Abstract

The ASM Copilot is an intelligent auto completion service which helps a programmer in autocompleting code snippets. While the programmer is still typing, the pilot calculates what the user is trying to type and suggests a set of most relevant auto completions.

Our main idea is to reduce the problem of code completion to a natural-language processing problem of predicting probabilities of sentences. We design a simple analysis that extracts sequences of keywords from a large codebase, and indexes these into a language model. We then employ the language model to find the highest ranked suggestions and use them to synthesize a code completion.

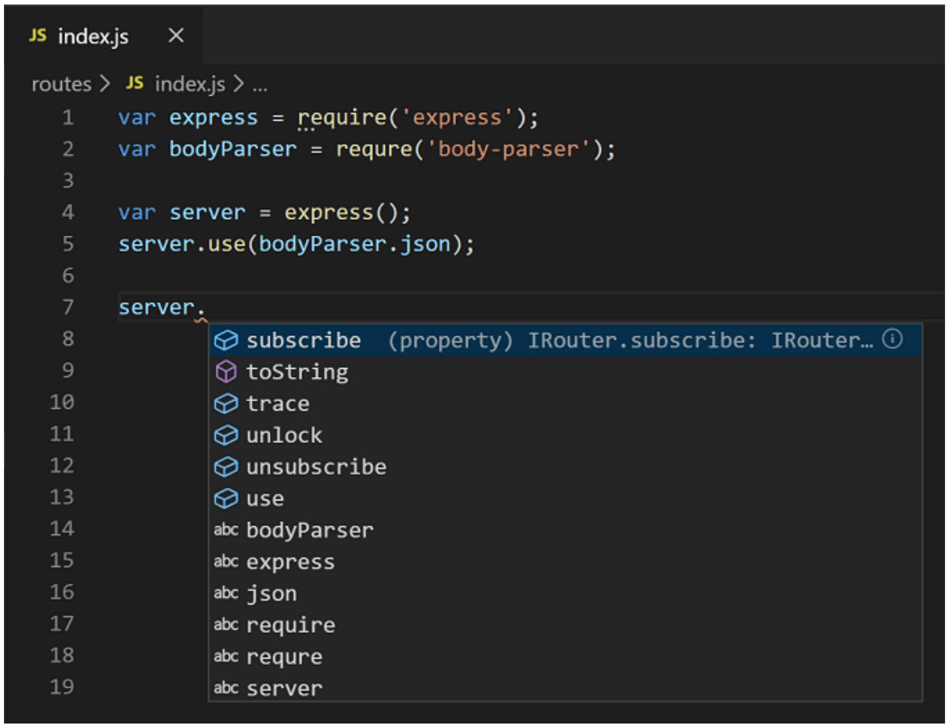
The idea of project is inspired from services like GitHub Copilot and VSCode auto completion which are illustrated in the following screenshots:



```
#!/usr/bin/env ts-node

import { fetch } from "fetch-h2";

// Determine whether the sentiment of text is positive
// Use a web service
async function isPositive(text: string): Promise<boolean> {
  const response = await fetch(`http://text-processing.com/api/sentiment/`, {
    method: "POST",
    body: `text=${text}`,
    headers: {
      "Content-Type": "application/x-www-form-urlencoded",
    },
  });
  const json = await response.json();
  return json.label === "pos";
}
```



```
var express = require('express');
var bodyParser = requre('body-parser');

var server = express();
server.use(bodyParser.json);

server.
```

Dataset

The dataset consists of sets of code snippets of projects from different GitHub repositories.

Size: 1 to 2 GB of C++ code.

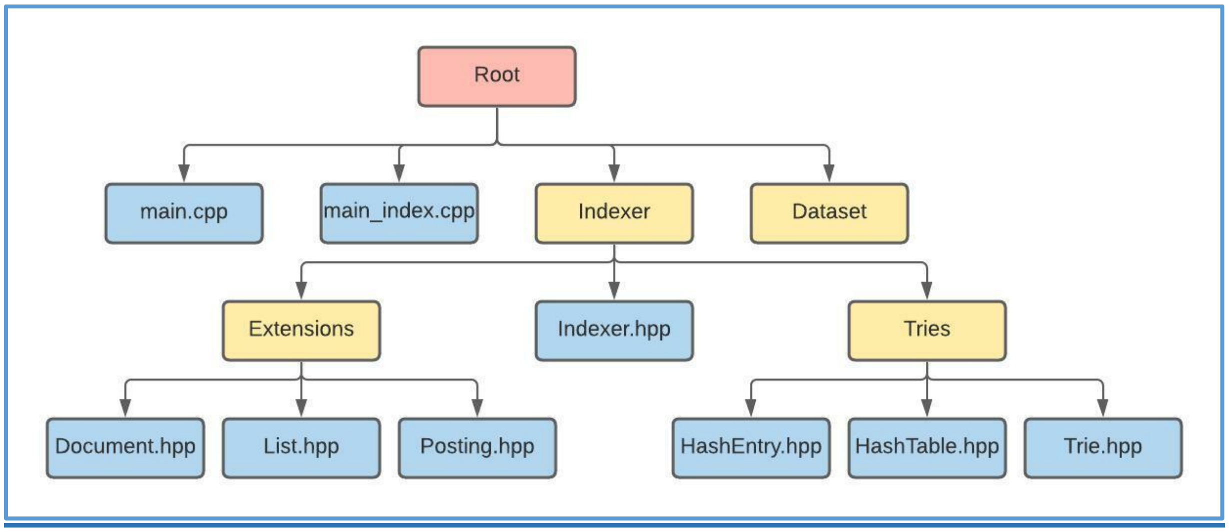
Site: https://zenodo.org/record/3472050#.YbNU1b1Bzcd.

Data Structures Used

- Linked List
- Hash Table
- Trie

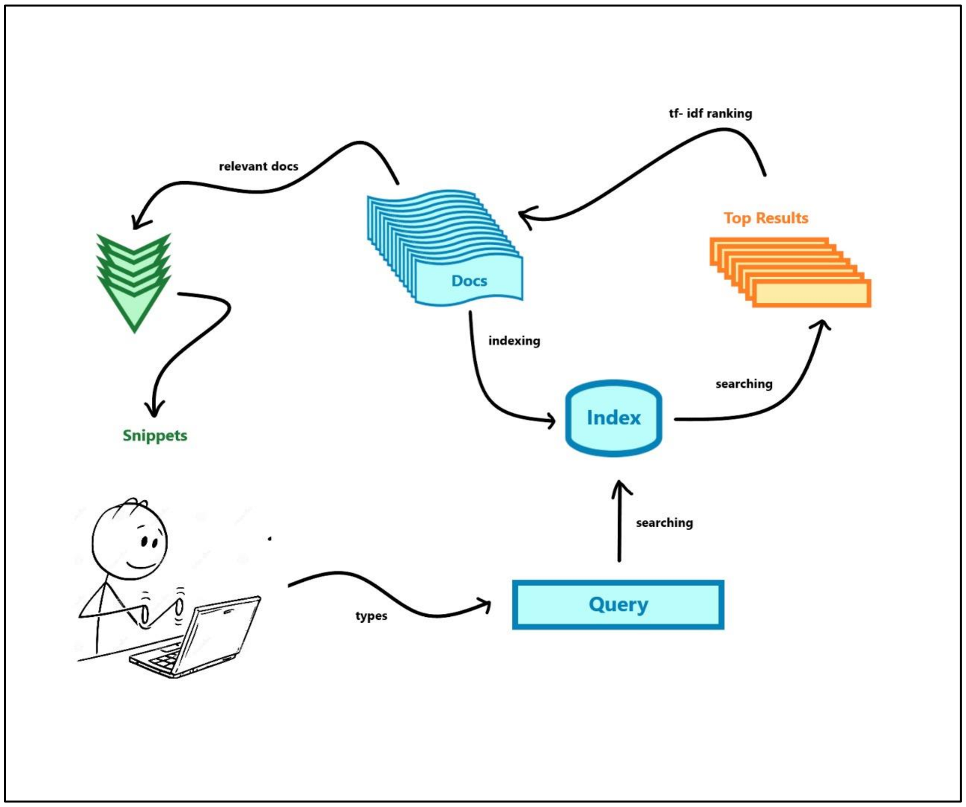
Project Structure

The project has been divided into directories and subdirectories as follows:



Methodology

The picture below illustrates the basic working of our project:

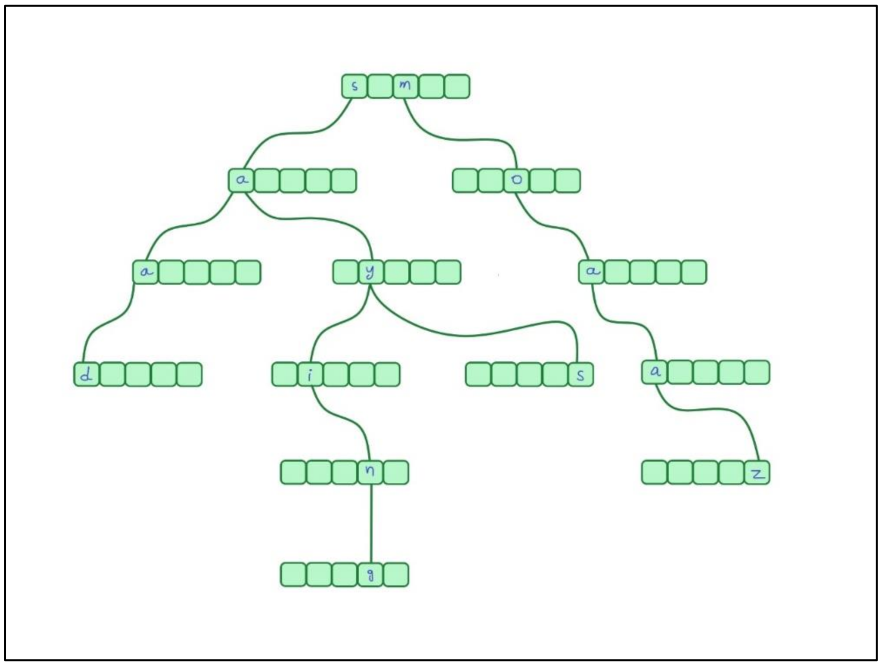


Indexing:

- We create an index of words from our dataset.
- We create two threads – one for indexing and the other for displaying percentage of indexing that has been done using the *printCount()* function.
- We iterate through all of the files in the dataset and index them one by one.
- Once all the files are done, we write the index to a new file, *index.txt,* via the *write_on()* method.
- The index consists of a dictionary and a posting list for each word.

Dictionary:

- The dictionary is a hash table.
- Hashing has been done so that we can quickly apply the hash function (a simple division hash) to know which index our character lies at.
- The first hash table stores all of the first characters.
- Each node points to a subsequent hash table which stores the next character, as shown below:



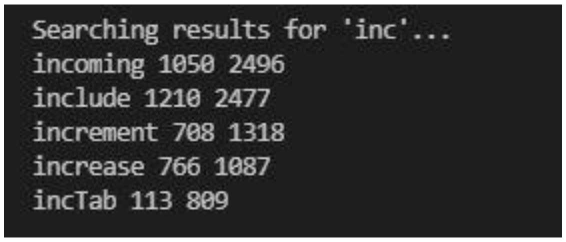
- The last node (which stores the last character of a word) points to a posting list.

Posting list:

- The posting list stores the document count (number of documents in which the word appeared), total count (total number of times the word appeared in the dataset), and a linked list of *Documents* (a struct).
- Each *Document* stores document ID, term frequency (number of times the term appeared in the document), and a linked list of line numbers on which the term appeared.

Word Auto Completion:

- We read the index from *index.txt.*
- Once the index is ready, we provide the program a partial string, and the program suggests the complete string based on the data in the indexer.
- Only the top five suggestions are displayed.
- The suggestions are filtered based on total count.
- The following picture demonstrates how this works:



```
Searching results for 'inc'...
incoming 1050 2496
include 1210 2477
increment 708 1318
increase 766 1087
incTab 113 809
```

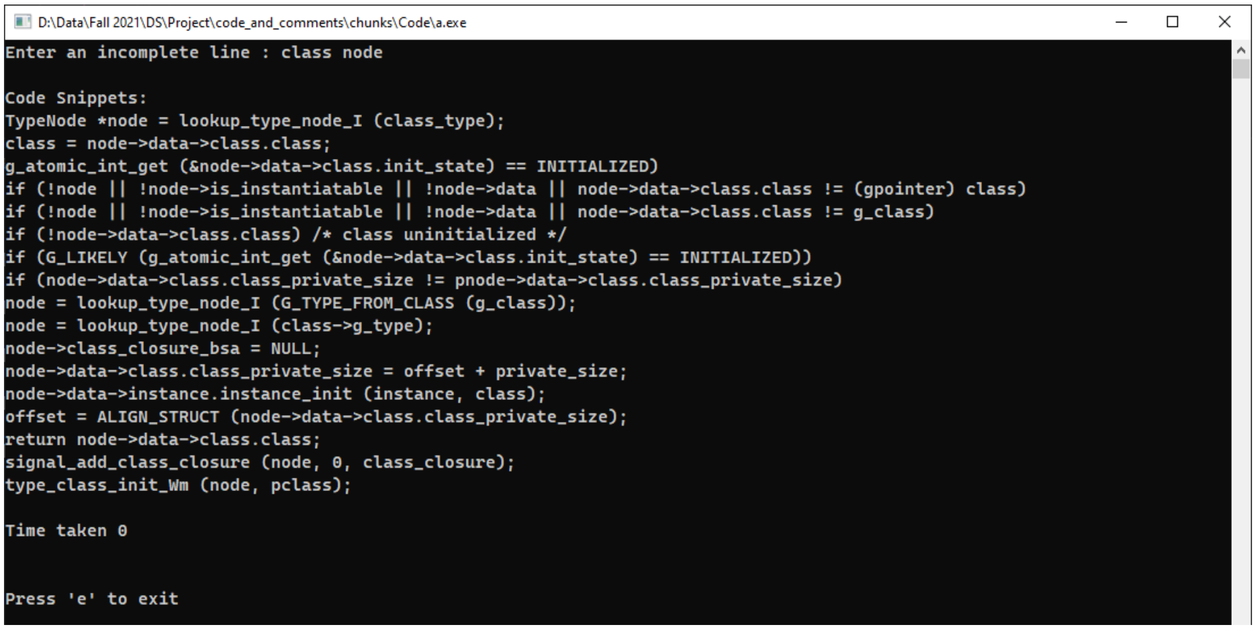
- The numbers beside the words represent the document count and total count respectively.

TF-IDF:

TF-IDF is a numerical score used in Information Retrieval systems, which can accurately represent the relevance of a search term within a large corpus of documents. The idea is that rarer words help narrow down the search more than common words, making those documents rank higher. TF-IDF is currently the best known methodology for scoring the relevance of search terms in a set of documents.

Code Snippet Auto Completion:

After successfully computing the top 5 suggestions during the Word Auto Completion phase, the TF-IDF score for these 5 words are then calculated for every single document that they occur in, producing a list of highest scoring documents. We were then able to look at the occurance of the term within these top-scoring documents and display the concurrent lines in order to provide relevant suggestions for the line being typed by the user.

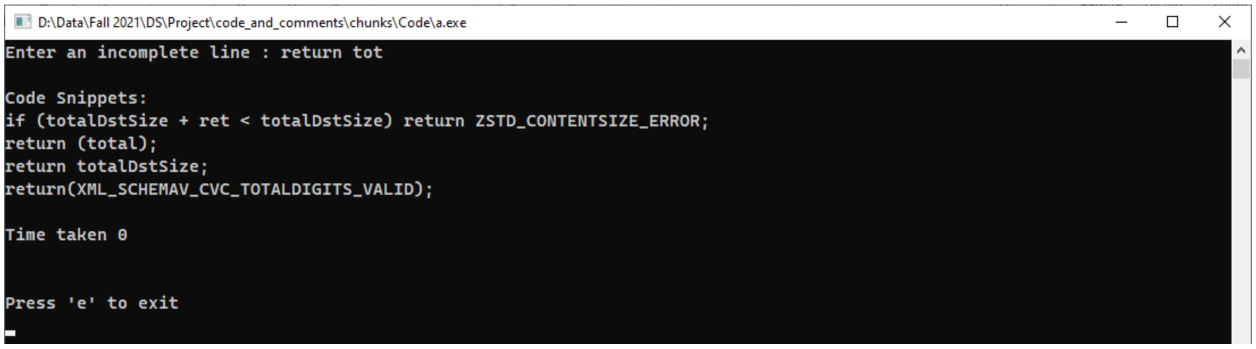

D:\Data\Fall 2021\DS\Project\code_and_comments\chunks\Code\a.exe

```
Enter an incomplete line : class node

Code Snippets:
TypeNode *node = lookup_type_node_I (class_type);
class = node->data->class.class;
g_atomic_int_get (&node->data->class.init_state) == INITIALIZED)
if (!node || !node->is_instantiatable || !node->data || node->data->class.class != (gpointer) class)
if (!node || !node->is_instantiatable || !node->data || node->data->class.class != g_class)
if (!node->data->class.class) /* class uninitialized */
if (G_LIKELY (g_atomic_int_get (&node->data->class.init_state) == INITIALIZED))
if (node->data->class.class_private_size != pnode->data->class.class_private_size)
node = lookup_type_node_I (G_TYPE_FROM_CLASS (g_class));
node = lookup_type_node_I (class->g_type);
node->class_closure_bsa = NULL;
node->data->class.class_private_size = offset + private_size;
node->data->instance.instance_init (instance, class);
offset = ALIGN_STRUCT (node->data->class.class_private_size);
return node->data->class.class;
signal_add_class_closure (node, 0, class_closure);
type_class_init_Wm (node, pclass);

Time taken 0


Press 'e' to exit
```



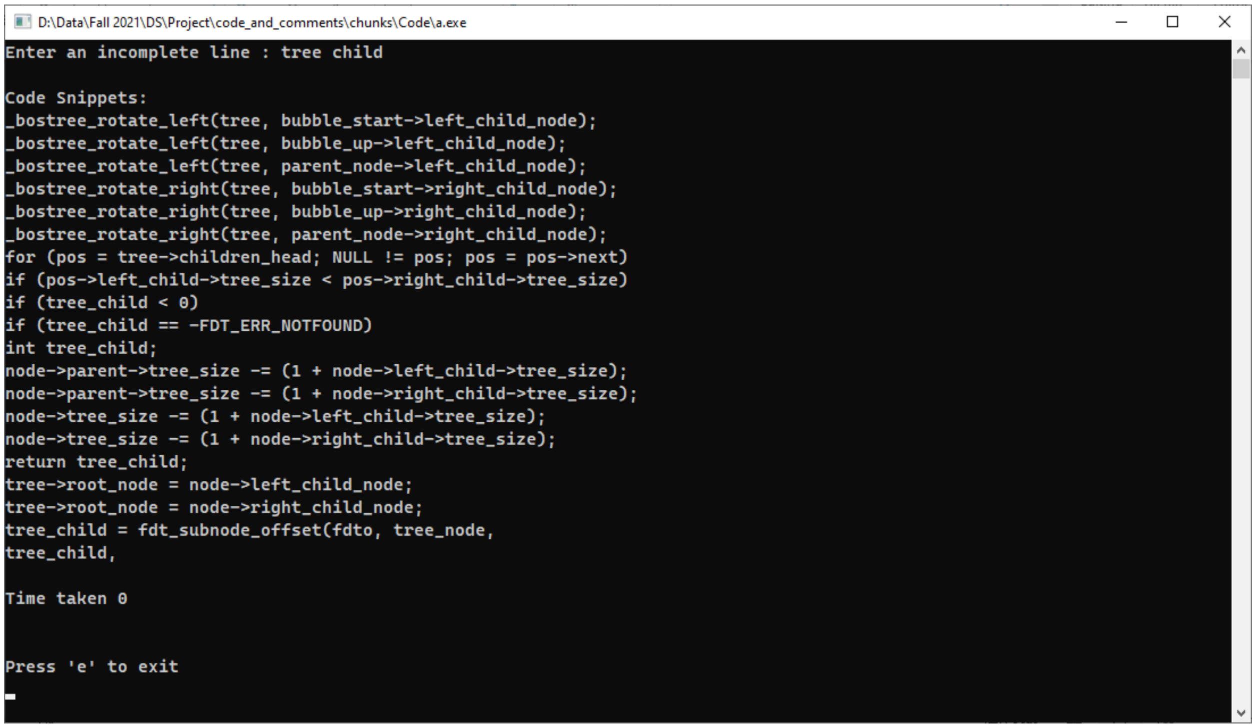
D:\Data\Fall 2021\DS\Project\code_and_comments\chunks\Code\a.exe

```
Enter an incomplete line : return tot

Code Snippets:
if (totalDstSize + ret < totalDstSize) return ZSTD_CONTENTSIZE_ERROR;
return (total);
return totalDstSize;
return(XML_SCHEMAV_CVC_TOTALDIGITS_VALID);

Time taken 0


Press 'e' to exit
```



D:\Data\Fall 2021\DS\Project\code_and_comments\chunks\Code\a.exe

```
Enter an incomplete line : tree child

Code Snippets:
_bostree_rotate_left(tree, bubble_start->left_child_node);
_bostree_rotate_left(tree, bubble_up->left_child_node);
_bostree_rotate_left(tree, parent_node->left_child_node);
_bostree_rotate_right(tree, bubble_start->right_child_node);
_bostree_rotate_right(tree, bubble_up->right_child_node);
_bostree_rotate_right(tree, parent_node->right_child_node);
for (pos = tree->children_head; NULL != pos; pos = pos->next)
if (pos->left_child->tree_size < pos->right_child->tree_size)
if (tree_child < 0)
if (tree_child == -FDT_ERR_NOTFOUND)
int tree_child;
node->parent->tree_size -= (1 + node->left_child->tree_size);
node->parent->tree_size -= (1 + node->right_child->tree_size);
node->tree_size -= (1 + node->left_child->tree_size);
node->tree_size -= (1 + node->right_child->tree_size);
return tree_child;
tree->root_node = node->left_child_node;
tree->root_node = node->right_child_node;
tree_child = fdt_subnode_offset(fdto, tree_node,
tree_child,

Time taken 0


Press 'e' to exit
```

Implementation

Functions in List.hpp:

- **Node():** Default constructor.
- **Node(const type &data, Node<type> *next):** Parameterized constructor. Initializes data (member of *Node* struct) to data (parameter) and next (member of *Node* struct) to next (parameter).
- **Node(const Node<type> &other):** Copy constructor.
- **Node<type> &operator=(const Node<type> &other):** Overloaded assignment operator.
- **List():** Default constructor.
- **List(const List<type> &other):** Copy constructor.
- **List<type> &operator=(const List<type> &other):** Overloaded assignment operator.
- **~List():** Destructor to delete the linked list.
- **void clear():** Deletes the linked list.
- **type *push_back(const type &data):** Inserts the given data in the list and returns pointer to the node in which insertion was done.
- **type *search(const type &data):** Searches the given data in the list. It returns pointer to the node in which data was found and returns null pointer if data is not found.
- **Node<type> *begin():** Constant function which returns an iterator to the start of list.
- **Node<type> *last():** Constant function which returns an iterator to the end of list.

Functions in Document.hpp:

- **Document():** Default constructor.
- **Document(const unsigned &ID):** Parameterized contructor. Initializes ID (member of *Document* struct) to ID (parameter).
- **Document(const unsigned &ID, const unsigned& line_no):** Parameterized contructor. Initializes ID (member of *Document* struct) to ID (parameter). Adds given line number to list of line numbers.
- **Document(const Document &other):** Copy constructor.

- **Document &operator=(const Document &other):** Overloaded assignment operator.
- **void update(const unsigned &line_no):** Adds given line number to list of line numbers.
- **bool operator==(const Document &other):** Returns true if ID of this document matches the ID of *other* document. Returns false otherwise.

Functions in Posting.hpp:

- **Posting():** Default constructor.
- **Posting(const unsigned &doc_ID, const unsigned &line_no):** Parameterized contructor. Adds document with the given ID to the list of documents. Adds line number to list of line numbers associated with the document.
- **Posting(const Posting &other):** Copy constructor.
- **Posting &operator=(const Posting &other):** Overloaded assignment operator.
- **void push_directly(const unsigned &doc_ID, const unsigned &line_no):** Adds document with the given ID to the list of documents. Adds line number to list of line numbers associated with the document.

Functions in HashEntry.hpp:

- **HashEntry():** Default constructor.
- **HashEntry(const char &data):** Parameterized constructor. Initializes data (member of *HashEntry* struct) to data (parameter).
- **HashEntry(const HashEntry &other):** Copy constructor.
- **HashEntry &operator=(const HashEntry &other):** Overloaded assignment operator.

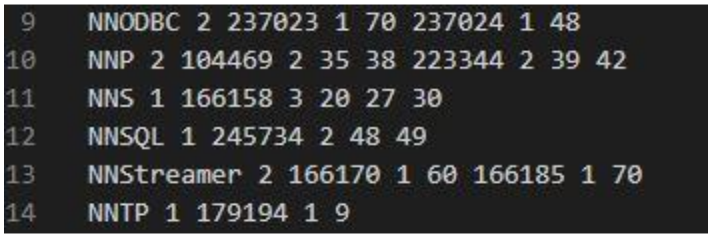
Functions in HashTable.hpp:

- **HashTable():** Default constructor.
- **HashTable(const HashTable &other):** Copy constructor.
- **HashTable &operator=(const HashTable &other):** Overloaded assignment operator.
- **~HashTable():** Destructor to delete memory allocated to *entries*.

- **HashEntry *insert(const char &symbol):** Inserts the given character into the hash table and returns pointer to the node in which the insertion is done. Division hash is used to determine where insertion should be done.
- **HashEntry *search(const char &symbol):** Divison hash is used to determine where to search. Searches for the given character in that hash entry. It returns pointer to the node in which the character is found and returns null pointer if the character is not found.

Functions in Trie.hpp:

- **Trie():** Constructor which dynamically allocates memory to *root* (member of the class – a pointer to the first hash table).
- **~Trie():** Destructor to delete the trie.
- **void deleteTrie():** Deletes the existing trie.
- **HashEntry *Trie::insert(const std::string &prefix):** Inserts the given string into the trie. Returns pointer to the hash entry in which the last character is inserted.
- **HashEntry *Trie::search(const std::string &prefix):** Searches the given string in the trie. It returns pointer to the hash entry in which the last character is present and returns null pointer if the string is not found.
- **HashEntry *Trie::search_incomplete(const std::string &str):** Same as *search()* except that it searches for a partial string. As such, *endOfWord* (member of *HashEntry* – a bool which is true if the character stored in the node is the last of the word) need not be true.
- **void write(std::ostream &buffer):** Wrapper function. Calls writeUtil(). Receives stream of the file in which we want the index to be written.
- **void Trie::writeUtil(HashTable *ptr, std::string &prefix, std::ostream &buffer):** Writes each word stored in the trie into the file whose output stream is passed as argument. Below is a snapshot of the final file:



```
NNODBC 2 237023 1 70 237024 1 48
NNP 2 104469 2 35 38 223344 2 39 42
NNS 1 166158 3 20 27 30
NNSQL 1 245734 2 48 49
NNStreamer 2 166170 1 60 166185 1 70
NNTP 1 179194 1 9
```

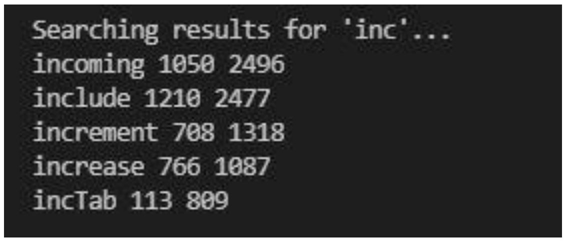
Each word is followed by its document count which is followed by document ID, term frequency, and line numbers for each document. For instance, take the word '*NNP*'. It appears in two documents. The ID of the first document is 104469 and in this document, it appears twice – on lines 35 and 38. Similarly, for the second document, the ID is 223344 and it appears twice – on lines 39 and 42.

- **void searchTopWords(std::string &partialStr, const unsigned &count, Results &results):** Receives a partial string, *partialStr,* the number of complete strings to suggest, *count,* and *results,* a vector of pairs of strings and pointers to posting lists associated with the given strings (typenamed *Results*). It calls *search_incomplete(partialStr)* to get pointer to the hash entry from which the search for complete strings is to begin. Returns if *search_incomplete()* returns null pointer; otherwise calls *writeSearches().*
- **void writeSearches(std::string &partialStr, HashTable *ptr, const unsigned &count, Results &buffer):** Receives a partial string, *partialStr*, the number of complete strings to suggest, *count*, and *buffer*, a vector of pairs of strings and pointers to posting lists associated with the given strings (typenamed *Results*). The words are filtered on the basis of total count. At the end of the function, *buffer* is filled with the top *count* suggestions for the complete string (note that the number of suggestions may be lesser).

Functions in Indexer.hpp:

- **Indexer():** Default constructor.
- **void index(const char *filename, const unsigned &doc_ID = 0):** Receives name and ID of the document we want to index. Extracts words of more than 3 characters from the document ignoring words in strings and comments, and inserts them into the trie.
- **void write_on(const char *filename):** Receives name of the file in which we want the index to be written. Opens that file and then calls *write()* to handle the rest.
- **void read(const char *filename):** Reads the index from the file whose name is passed as argument and stores it in the trie.
- **HashEntry *search(const std::string &token):** Calls *search()* to search the given string in the trie. It returns pointer to the hash entry in which the last character is present and returns null pointer if the string is not found.

- **void searchWords(std::string &partialStr, const unsigned &count):** Receives a partial string, *partialStr*, and the number of complete strings to suggest, *count.* Creates a variable of *Results* and calls *searchTopWords()* to populate it with the top *count* suggestions for the complete string (note that the number of suggestions may be lesser). The strings are then printed along with document count and total count, as shown below:



```
Searching results for 'inc'...
incoming 1050 2496
include 1210 2477
increment 708 1318
increase 766 1087
incTab 113 809
```

- **float getTfIdf(const unsigned &term_freq, const unsigned &doc_count):** Receives the frequency of a term within a document, as well as the total number of documents that the term occurs in. The formula for TF-IDF weightage is then applied and the resultant score is returned.
- **void goToLine(std::ifstream &file, unsigned line):** Receives an input file stream and a line number to jump to. Uses the ifstream::ignore function to read and discard from the file until the desired line number if reached.
- **void rankDocs(const Posting *posting, std::vector<std::pair<Document *, float>> &docs):** The TF-IDF score is calculated for every document within this posting, and the top 5 highest ranking documents are stored in the given vector.
- **void scoreWords(const Posting *posting):** Receives the posting for a term within the index.Uses rankDocs function to get top 5 relevant documents in a vector. This vector is then traversed, and for each top-scoring document the lines containing the search term, alongside a few concurrent lines, are printed to form a predicted code snippet.
- **void complete_line(std::string &context, std::string &incomplete, std::set<std::string> &dest):** An incomplete line consists of two parts: the context, and the incomplete query. This function receives both of them and searches the index for their autocompleted results. Then we find the intersecting documents in their results, and then we rank top 10 of them w.r.t the product of their individual tf-idf ratings. Hence, now we have a set of those documents which are top in relevance w.r.t both the context and

incomplete query. Finally, we look for intersections of lines and store them in the given vector.

Functions in main_index.cpp:

- **DWORD WINAPI printCount(LPVOID lpParam):** Prints the percentage of indexing that has been done, correct to one decimal place.
- **int main():** The driver function which indexes the dataset. Creates a thread which handles *printCount().* Meanwhile, it itself goes through all of the files in the dataset directory and indexes them one by one. The time taken for indexing is calculated and displayed. Finally, the index is written to *index.txt* and the created thread is closed.

Contact Us

For any queries or remarks, feel free to contact us at k200154@nu.edu.pk or k200161@nu.du.pk or k200245@nu.edu.pk