1. Background

In Assignment 1 you implemented a dictionary using a linked list. Many applications of dictionaries cannot assume that the user will query in the correct format. Take Google search as an example. If someone is so excited to check the recent Olympics updates, they may accidentally search something similar to this:

图形用户界面, 文本, 应用程序

描述已自动生成

Thankfully, there is enough context in the original query that the intended search result is recommended back to the user. In Assignment 2 you will be extending your implementation to account for misspelled keys when querying your dictionary.

2. Your Task

**Assignment:**

Overall, you will create a partial error-handling dictionary (spellchecker) using a radix tree.

* You will be using the **same dataset** as Assignment 1.
* Users will be able to **query the radix tree** and will get either the expected key, or the closest recommended key.
* You will then **write a report** to analyse the time and memory complexity of your Assignment 1 linked list compared to your radix tree implementation.

**C Implementation:**

* Your programs will build the dictionary by reading data from a file. They will **insert each suburb into the dictionary (either the linked list (Stage 3) or radix tree (Stage 4))**.
* Your programs will **handle the search for keys**. There are three situations that your programs must handle:
  + Handle exact matches: output **all** records that match the key (Stage 3 and 4).
  + Handle similar matches: if there are no exact matches, find **the most similar** key and output its associated records (**Stage 4 only**).
  + Handle no matches being found: if neither exact nor similar matches are found, indicate that there is no match or recommendation (Stage 3 and 4).
* Your programs should also be able to **populate and query** the dictionary with the Assignment 1 linked list approach and the radix tree approach.
* Your programs should also provide enough information to compare the **efficiency** of the linked list with the radix tree with a operation-based measure that ignores less important run-time factors (e.g. comparison count).

It is recommended to use your own solution of Assignment 1, and extend it for Assignment 2. But, we will also provide a solution for Assignment 1 if you prefer to work with our code.

3. An Introduction to Tries and Radix Trees

First, it is important to establish the **difference** between a **Trie** and a **Tree**. This is best illustrated with an example. One example of a tree is a binary search tree (BST), where each node in the tree stores an entire string, as illustrated below. The nodes are ordered and allow easy searching. When searching in the BST, we compare the query with the entire string at each node, deciding whether to switch to the left or right subtree or stop (if the subtree is empty) based on the result of the comparison.

图示

描述已自动生成

A *trie* is slightly different. It has multiple names, such as *retrieval tree* or *prefix tree*. **In a trie, the traversal of the tree determines the corresponding key**. For the same strings as above with one letter per node, it would look like:

图示

描述已自动生成

Tries allow for quick lookup of strings. Querying this trie with the key "hey" would find no valid path after the "e" node. Therefore, you can determine that the key "hey" does not exist in this trie.

A **radix trie** is a subtype of *trie*, sometimes called a *compressed trie*. Radix trees do not store a single letter at each node, but compress multiple letters into a single node to save space. At the character level, it would look like this:

图示

描述已自动生成

Radix tries can again be broken down into different types depending on *how many bits are used to define the branching factor*. In this case, we are using a radix (r) of 2, which means every node has 2 children. This is accomplished by using 1 bit of precision, so each branch would be either a 0 or 1. This type of *radix trie (with r=2)* is called a **Patricia trie.** Another example of a radix trie with r=4 would have 4 children, determined by the binary numbers 00, 01, 10, 11. The radix is related to the binary precision by r = 2x where x is the number of bits used for branching.

Radix trees benefit from less memory usage and quicker search times when compared to a regular trie.

While these visual representations are at the character level, a **Patricia** **trie is implemented using the binary of each character**. Each node in the trie stores the binary prefix at the current node, rather than the character prefix. For example, we can insert 3 binary numbers into a PATRICIA trie: 0000, 0010 and 0011.

图示

描述已自动生成

Combining the previous worded example with the binary representation gives us a patricia tree of the form:

图示

描述已自动生成

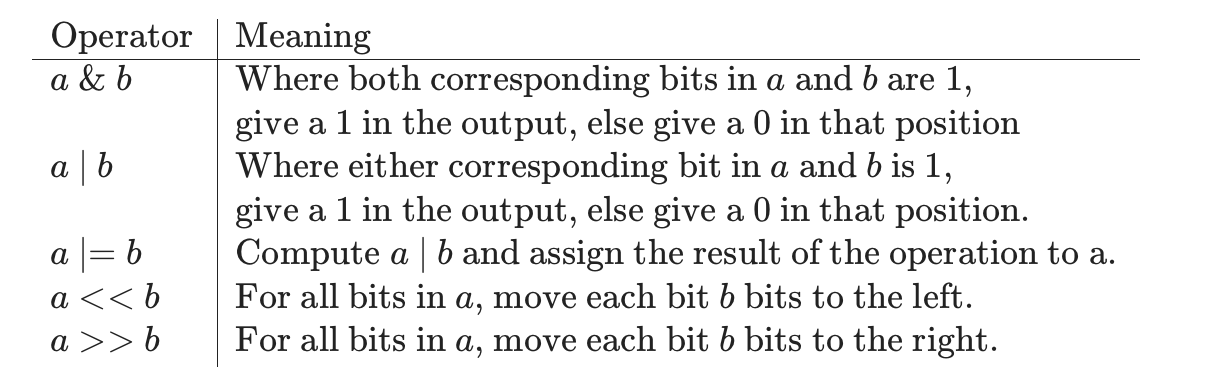
You should trace along each path and validate that the stored strings are the same as the example above. Each character is represented over 8 bits, and the last character is followed by an end of string 00000000 8 bit character, i.e. NULL.

It is important to note that these representations only show the prefix at each node. An actual implementation will require more information within a node struct. To see this, look at the "extra insertion example" slide.

Implementation Tips

To implement the Radix Tree, a recommended approach is to use the functions provided below.

1. Get a particular bit from a given character getBit(key, n) and
2. Extract a stem from a given key splitStem(key, start, length) which extracts a certain number of bits from a given key.

If you want to understand the code provided, you need to understand the following bit operators over numbers *a* and *b*: 

Use The Following Functions

The two recommended functions, getBit and splitStem are provided here as they can typically be used without making major changes to their structure. Diagrams explaining their algorithmic structure and example outputs are also provided.

getBit

This function works out the byte which the bit we're trying to extract lies in and then works out which bit it will be in. The offset is not simply the remainder of the division because the bits are filled and split from the left (highest value bit) rather than the right.

/\* Number of bits in a single character. \*/

#define BITS\_PER\_BYTE 8

/\* Helper function. Gets the bit at bitIndex from the string s. \*/

static int getBit(char \*s, unsigned int bitIndex);

static int getBit(char \*s, unsigned int bitIndex){

    assert(s && bitIndex >= 0);

    unsigned int byte = bitIndex / BITS\_PER\_BYTE;

    unsigned int indexFromLeft = bitIndex % BITS\_PER\_BYTE;

    /\*

        Since we split from the highest order bit first, the bit we are interested

        will be the highest order bit, rather than a bit that occurs at the end of the

        number.

\*/

unsigned int offset = (BITS\_PER\_BYTE - (indexFromLeft) - 1) % BITS\_PER\_BYTE;

unsigned char byteOfInterest = s[byte];

unsigned int offsetMask = (1 << offset);

unsigned int maskedByte = (byteOfInterest & offsetMask);

/\*

The masked byte will still have the bit in its original position, to return

either 0 or 1, we need to move the bit to the lowest order bit in the number.

\*/

unsigned int bitOnly = maskedByte >> offset;

return bitOnly；

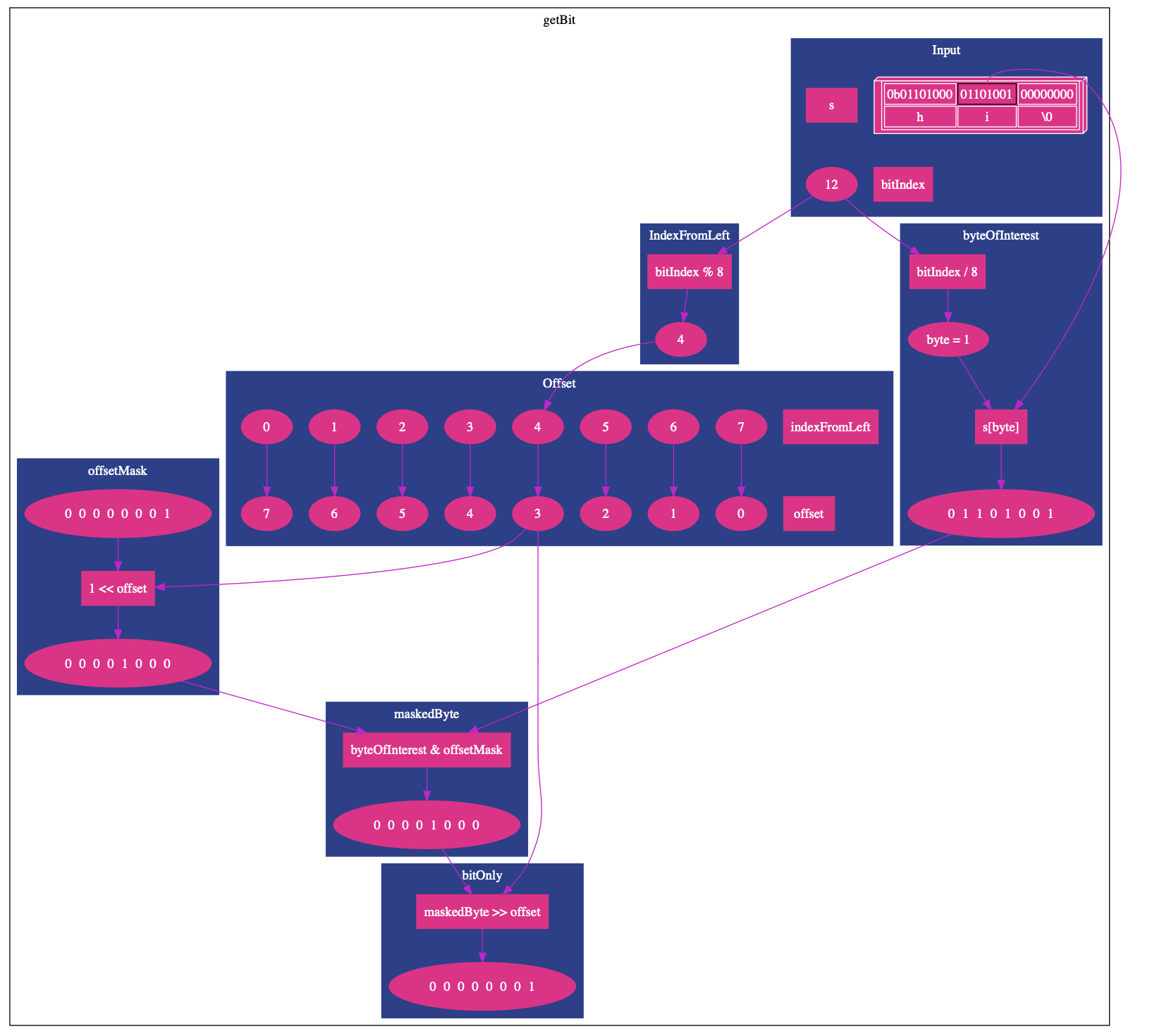
}

An example diagram of what the calculation of the offset does from each possible value of indexFromLeft is shown here:

图片包含 背景图案

描述已自动生成

An example for getting bit number 12 from the string "hi" could be called using getBit("hi", 12). An example of how the calculation occurs is shown below:



createStem

This function creates a new collection of bytes from an existing collection of bytes. Conceptually this is extracting a substring from a string and returning the new substring, but the returned value will *not* necessarily be NULL terminated and may not start from the beginning of a character - so may not be printable as a string for multiple reasons.

/\* Allocates new memory to hold the numBits specified and fills the allocated

    memory with the numBits specified starting from the startBit of the oldKey

    array of bytes. \*/

char \*createStem(char \*oldKey, unsigned int startBit, unsigned int numBits);

char \*createStem(char \*oldKey, unsigned int startBit, unsigned int numBits){

    assert(numBits > 0 && startBit >= 0 && oldKey);

    int extraBytes = 0;

    if((numBits % BITS\_PER\_BYTE) > 0){

        extraBytes = 1;

    }

    int totalBytes = (numBits / BITS\_PER\_BYTE) + extraBytes

    char \*newStem = malloc(sizeof(char) \* totalBytes);

    assert(newStem);

    for(unsigned int i = 0; i < totalBytes; i++){

newStem[i] = 0;

}

for(unsigned int i = 0; i < numBits; i++){

unsigned int indexFromLeft = i % BITS\_PER\_BYTE;

unsigned int offset = (BITS\_PER\_BYTE - indexFromLeft - 1) % BITS\_PER\_BYTE;

unsigned int bitMaskForPosition = 1 << offset;

unsigned int bitValueAtPosition = getBit(oldKey, startBit + i);

unsigned int byteInNewStem = i / BITS\_PER\_BYTE;

newStem[byteInNewStem] |= bitMaskForPosition \* bitValueAtPosition;

}

return newStem;

}

If the number of bits to extract matches exactly on a character border (i.e. if it is a multiple of 8) an extra character is not needed, but otherwise, an extra character is needed - with the remainder of the value being filled with 0's.

It works by fetching each bit from the given byte array and setting it in the new byte array.

An example of createStem("hello", 0, 12) is shown here:

图形用户界面, 网站

描述已自动生成

An example of createStem("hello", 12, 15) is shown here:

图形用户界面, 网站

描述已自动生成

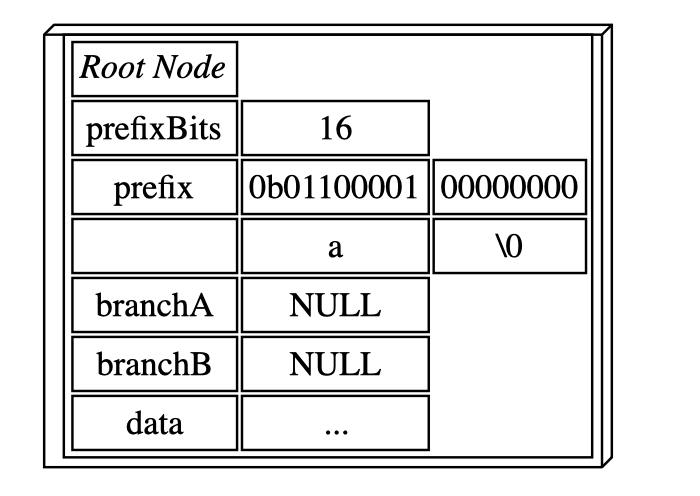
**Note**: The result of createStem is **not** a string, strlen will not reliably count its length and printing it as a string may lead to unexpected results.

4. Extra Insertion Example and Insertion Process

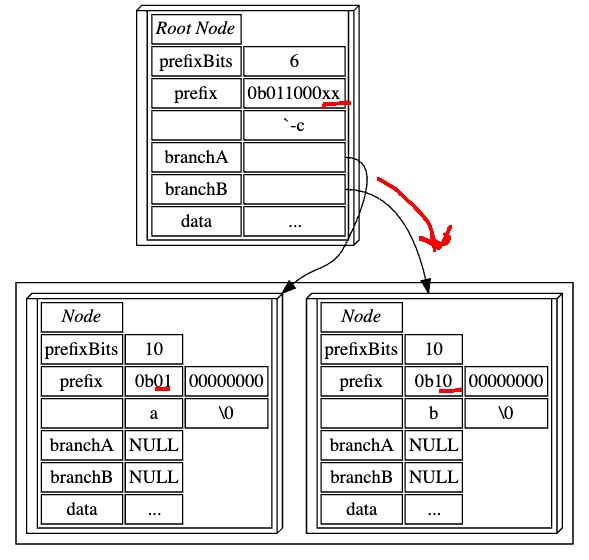
This extra slide shows an additional example for inserting data into a prefix tree. This shows the process of inserting the seven strings:

* "a" 01100001 00000000
* "b" 01100010 00000000
* "c" 01100011 00000000
* "d" 01100100 00000000
* "e" 01100101 00000000
* "f" 01100110 00000000
* "g" 01100111 00000000

The first inserted string ("a" - 1100001 00000000) has no splits, so the prefix is the whole string:



The second inserted string ("b" - 1100010 00000000) first differs in the second last bit of the first character, so the tree splits at that point. The root node's prefix only determines the first 6 bits of the string - narrowing all strings in the the tree to starting with a character which has the first six bits 011000. Therefore, the root node can lead to any character from ` to c. [Check this ASCII-Binary table (numbers 96-99)](https://www.gcsecs.com/ascii1.html).



The third inserted string ("c" - 01100011 00000000) produces a new difference in the last bit (of the first character) in the node completing the string "b". After the split, the right branch of the second level node can lead the (first) character to be 'b' and 'c'.

图示

描述已自动生成

The fourth inserted string ("d" - 01100100 00000000) produces a difference in the root node, as the (first) character differs in its third last bit - causing a split in the root node after 5 bits. After the split, the root node only narrows the letters in the first character of all items in the tree from the character '`' to the character 'g'.

图示

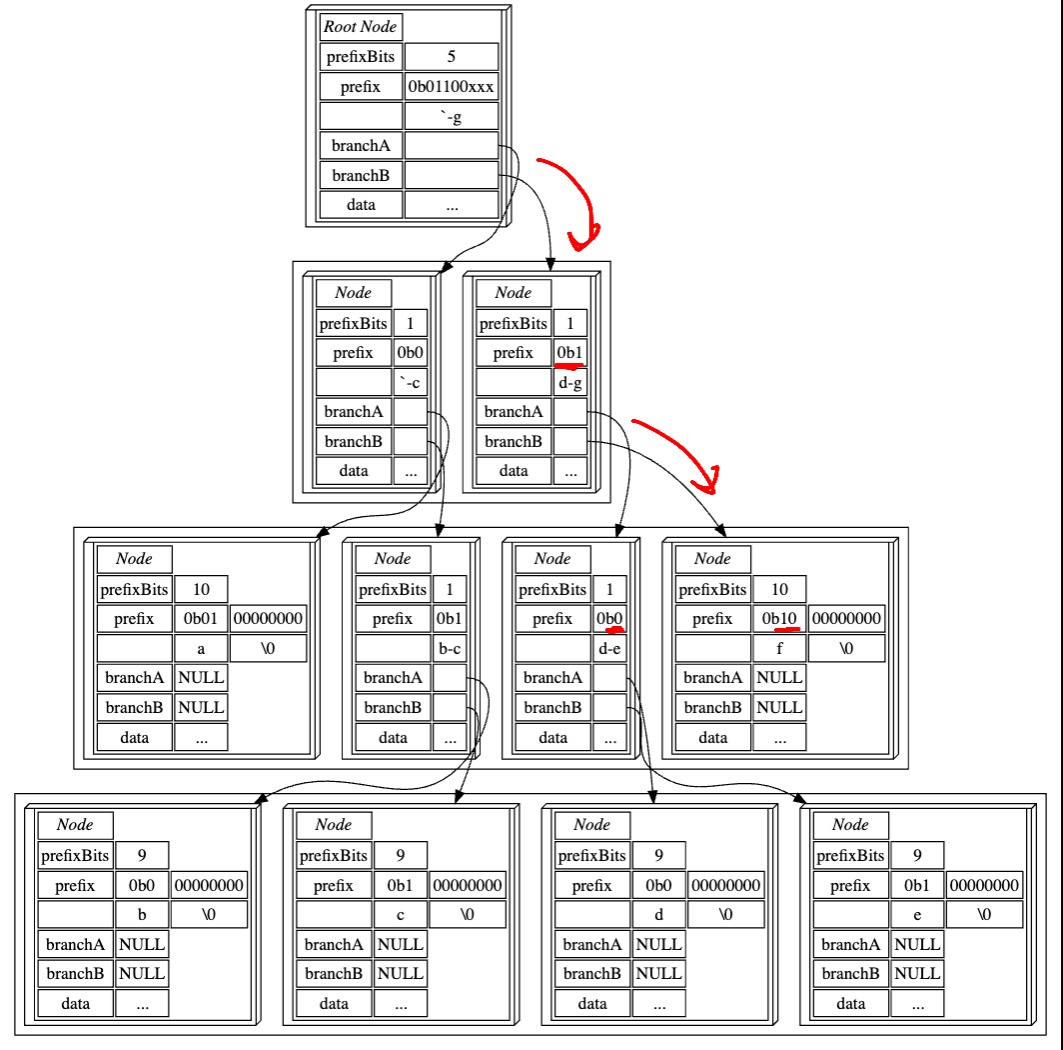
描述已自动生成

The fifth inserted string ("e"- 01100101 00000000) produces a new difference in the node completing the string "d" in the last bit of the (first) character in the string. This causes a split, the third last and second last bits for 'd' and 'e' are still the same, so the node retains these two bits.

图示

描述已自动生成

The sixth inserted string ("f" - 01100110 00000000) produces a split in the node deciding between the (first) character being 'd' or 'e', with the second last bit of the character (the second bit in the node) being different.



Then the final inserted string ("g" - 01100111 00000000) produces a split in the node completing the string "f", with the last bit of the first character differing.

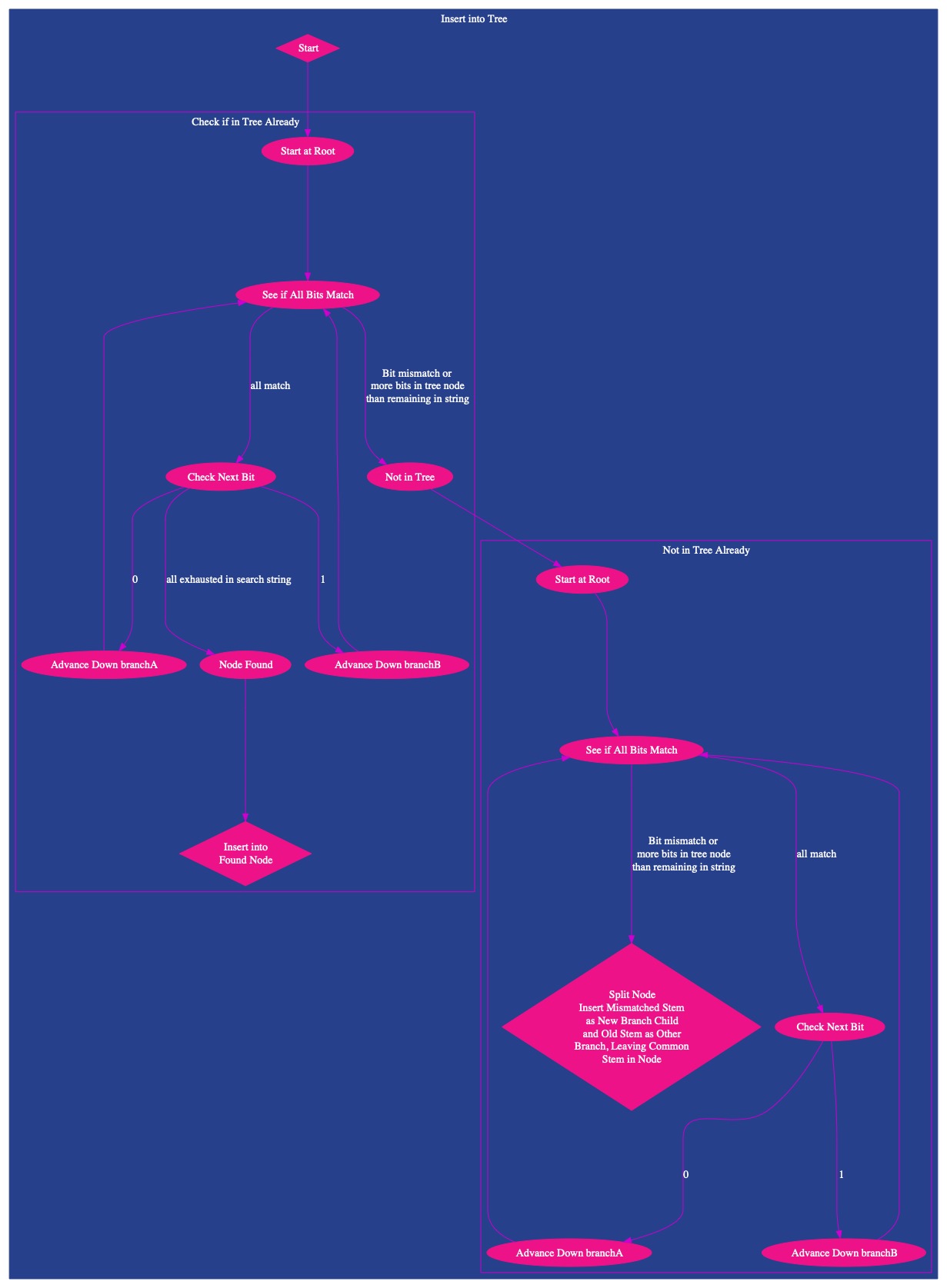
图示

描述已自动生成

Insertion

In addition to the two functions provided to work over bit representations of strings, you will likely have some kind of insertion operation to handle the example above.

The code logic details will likely vary depending on your data structure and implementation of Assignment 1. Therefore, this diagram is provided as an overarching diagram to help you through the task of inserting a new string.



Inserting into the found node simply adds to the list of data in the node and the split node step will split at the common bit position, leaving the common stem in the node, placing the stem from the inserted string in a newly created node down the appropriate branch (A if a 0 is the following bit in the string, and B if it is a 1) and the stem from the existing string down the other branch in a newly created branch. After this split - the insertion is complete.

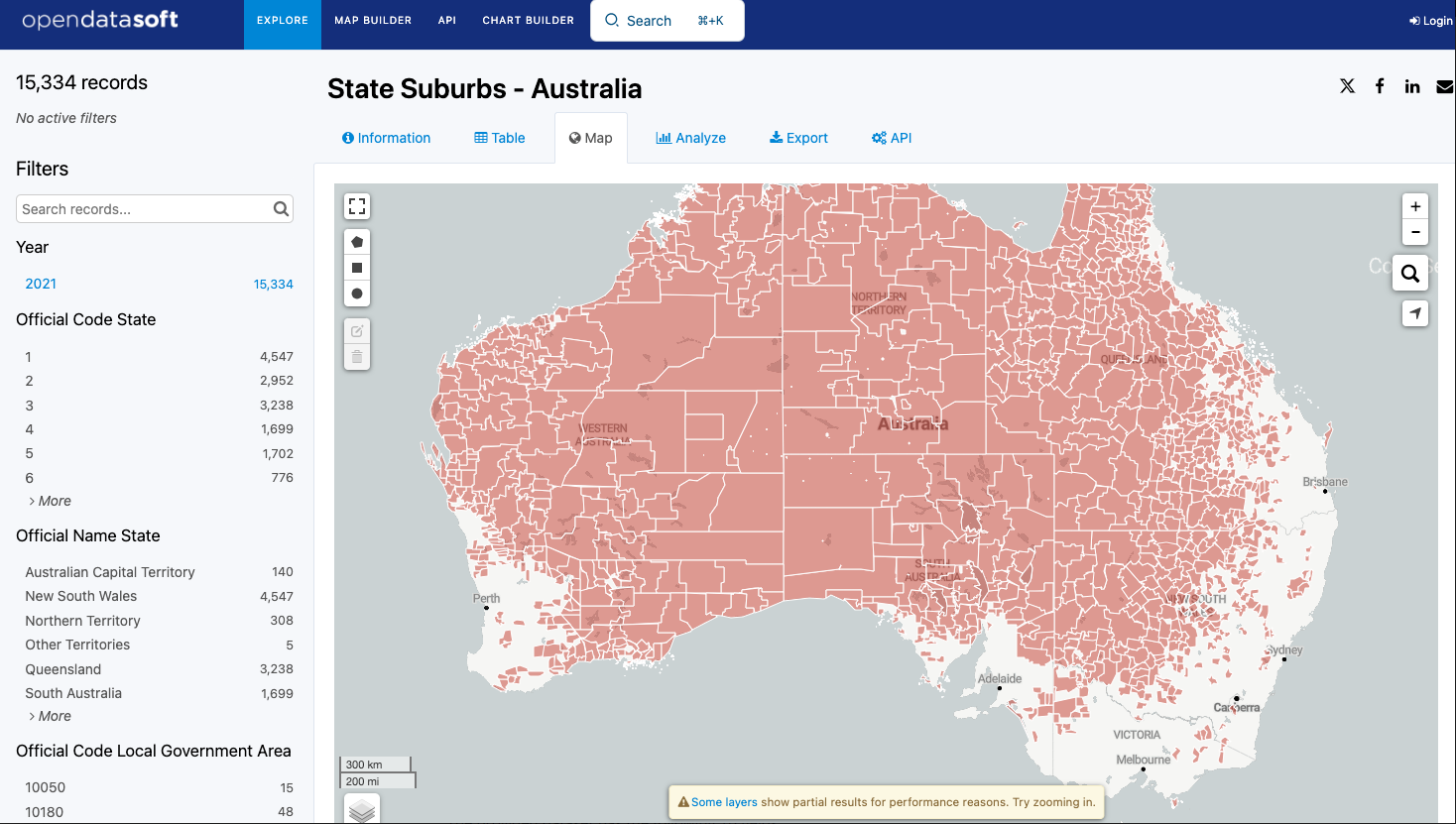
You may find it useful to set the data in the existing node to NULL to avoid the same data existing in multiple places in your tree.

By storing the \0 character, we do not need to worry about the case where the string continues after an existing node is exhausted - we will always reach a mismatch or the string will match, simplifying both the search and insertion logic

5. Dataset and Assumptions

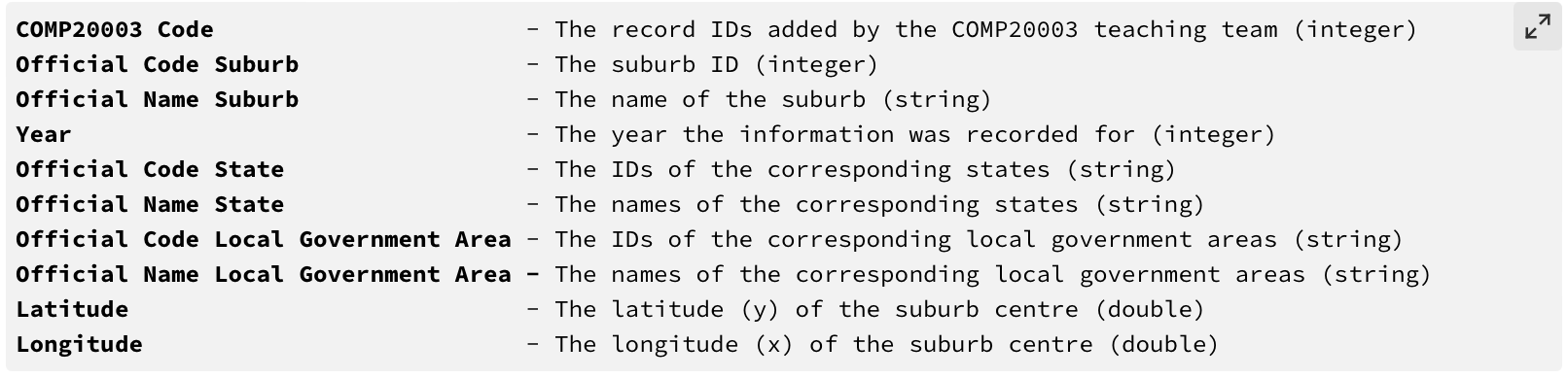
**Note: this slide is identical to the Assignment 1 dataset and assumptions.**

The [opendatasoft](https://data.opendatasoft.com/pages/home/) website provides numerous databases, enabling organisations and individuals to seamlessly access and share data. The dataset used in this assignment is a simplified version derived from the "[State Suburbs - Australia](https://data.opendatasoft.com/explore/dataset/georef-australia-state-suburb@public/information/?disjunctive.ste_code&disjunctive.ste_name&disjunctive.lga_code&disjunctive.lga_name&disjunctive.scc_code&disjunctive.scc_name)" database on that website which originates from [ABS data](https://www.abs.gov.au/AUSSTATS/abs@.nsf/DetailsPage/1270.0.55.001July 2016?OpenDocument). You can browse or visualize the original database using the link above. The whole simplified dataset, as well as smaller samples, can be found in the ***Dataset Download*** slide.



The processed dataset can be found in the ***Dataset Download*** slide. You aren't expected to perform this processing yourself.

The provided dataset has the following 10 fields:

The provided text gives a detailed description of the data format and assumptions for the assignment. Here's a summary of the key points:

**Fields and Data Types:**

* COMP20003 Code, Official Code Suburb, Year: integers
* Longitude, Latitude: doubles
* all other fields: strings (may contain spaces and commas)

*Special Cases:*

* Official Code State, Official Code Local Government Area: comma-separated lists of integers (treated as strings for this assignment).
* comma-containing strings**:** enclosed in double quotes ("") within the CSV file, the quotes are removed when stored in your program according to standard CSV rules.

**CSV File Assumptions:**

Note that normally a suburb lies entirely inside a single state, as well as a single local government area, but this is not a universal rule. Suburbs might be in multiple local governments areas and multiple states. This is why the fields Official Code State and Official Code Local Government Area are not classified as integers. Each of these fields is actually a comma-separated list of integers. For the purposes of this assignment, it is fully sufficient to consider this as a string.

This data is in CSV format, with each field separated by a comma. For the purposes of the assignment, you may assume that:

* the input data is well-formatted,
* input files are not empty,
* input files include a header line that contains the filled headers in the CSV format ,
* any field containing a comma will begin with a double quotation mark (") and end with a quotation mark ("),
* each field contains at least one character and at most 127 characters,
* fields always occur in the order given above, and that
* the maximum length of an input record (a single full line of the CSV) is 511 characters.

In this dataset, CSV conventions are followed for strings containing commas. These strings are enclosed in double quotation marks (") at the beginning and end. However, when processing the data and storing it in your dictionary, these quotation marks are removed. You can safely assume there won't be any double quotes within the actual string values stored in the dictionary so your program does not have to handle this case.

**Processing Assumptions:**

* The column headers (such as COMP20003 Code and Official Name Suburb) can be assumed to match the names expected and can be used directly in the output as the labels for each field.
* The number of columns (10 in this assignment), as well as the order, text and data type of each column are assumed to be known. While it's possible to automatically determine this information, it's beyond the scope of this assignment.

6. Implementation Details

Assignment 2 will involve roughly **three** new stages.

* **Stage 3** will extend your Assignment 1 solution to **count the number of comparisons** it takes to find a given key, i.e. a search query.
* **Stage 4** will implement the **lookup** of data by a given key (Official Name Suburb) **in a Patricia tree**.
* **Stage 5** is a **report** about the **complexity** of a linked list compared to the Patricia tree.

Stage 3: Linked List Search Comparisons

In this stage, you will extend your Assignment 1 solution to count the number of binary and string comparisons and node accesses used when searching for a given key.

Your Makefile will produce an executable called dict3. The command line arguments are identical to assignment 1 but with the stage being 3 instead.

* The first argument will be the *stage*, for this part, the value will always be 3 (for linked list lookup with comparison count added).
* The second argument will be the name of the input *CSV* *data file*.
* The third argument will be the name of the *output* *file*.

Your dict3 program should function exactly the same as assignment 1's stage 1. You will add the functionality to count comparisons when searching for a key which will be added to the stdout output. For this stage, and this stage only, you may assume:

* Each character compared adds exactly 8 bits to the bit comparison count.
* The node access is incremented once per accessed linked list node.
* Each string comparison, even if a mismatch occurs on the first character, is 1 string comparison.

You should create the functionality to store comparisons with stage 4 in mind. You will also be recording comparisons in the Patricia tree implementation, so your code should be easily applied to both. For this reason, you may want to extend your search function to include a pointer to a struct that holds information about the query and results.

**Important Notes**:

* You do *not* have to implement any spellchecking in the linked list. Your only task at this stage is to add functionality to count comparisons.
* You do *not* need to add this functionality to the deletion of nodes. Only the search will be assessed. You should, however, be able to recognize how to implement this in your deletion code.

Example Execution

make -B dict3

./dict3 3 tests/dataset\_1.csv output.txt < tests/test1.in

Example Output

The expected output to the *output* file would look like:

Carlton -->

COMP20003 Code: 9773, Official Code Suburb: 20495, Official Name Suburb: Carlton, Year: 2021, Official Code State: 2, Official Name State: Victoria, Official Code Local Government Area: 24600, Official Name Local Government Area: Melbourne, Latitude: -37.8004392, Longitude: 144.9680900

South Melbourne -->

With the following printed to stdout:

Carlton --> 1 records - comparisons: b64 n1 s1

South Melbourne --> NOTFOUND

Note: the bit comparisons are 8 times the character comparisons here

Stage 4: Patricia Tree Spellchecker

In Stage 4, you will implement the another dictionary allowing the lookup of data by key (Suburb Name).

Your Makefile should produce an executable program called dict4. This program should take three command line arguments:

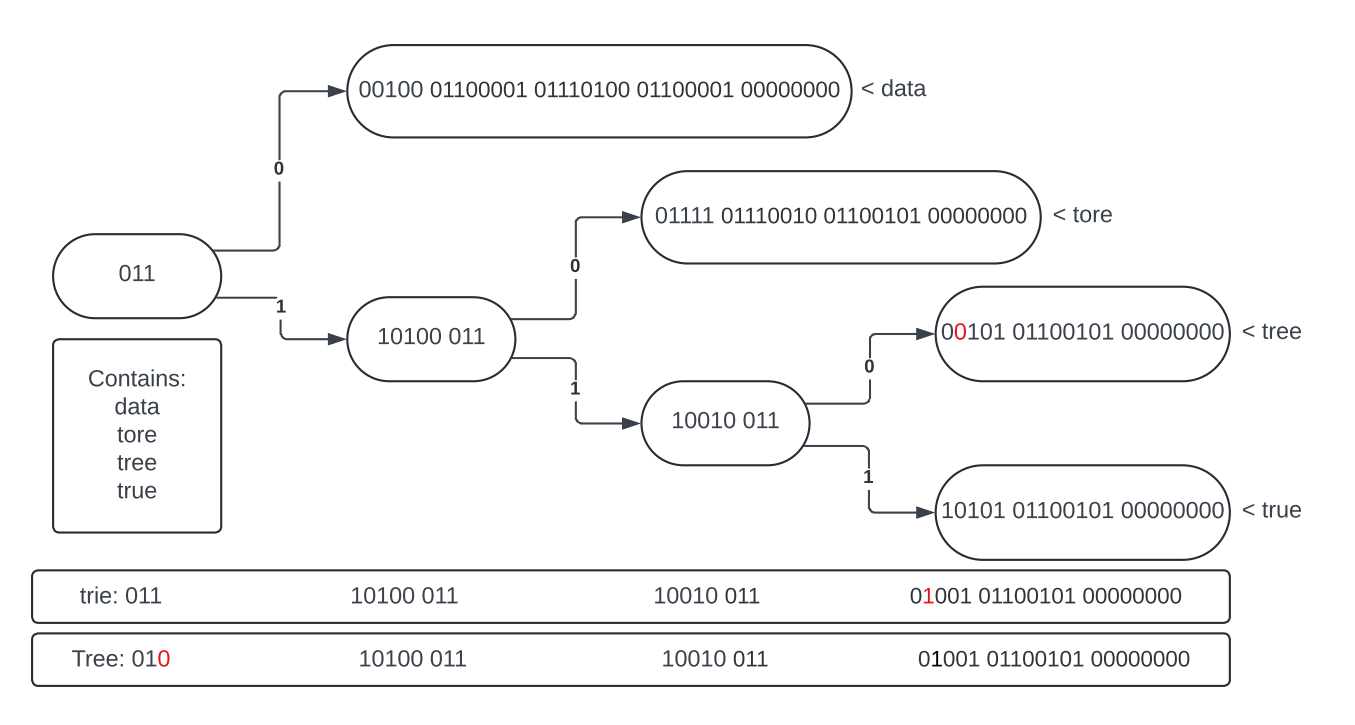
* The first argument will be the *stage*, for this part, the value will always be 4 (for Patricia tree lookup and comparison counting).
* The second argument will be the name of the input *CSV* *data file*.
* The third argument will be the name of the *output* *file*.

Your dict4 program should:

* Read the data in the same format as assignment 1 and stage 1, and save each entry in a Patricia tree using the Official Name Suburb as the key.
* The program should:
  + Accept Official Name Suburbs from stdin, and search the tree for the matching key. Since your program should act as a simple spellchecker, an exact match may not be found. Follow the process "mismatch in key" as outlined below to implement spellchecking logic.
  + You must output all matching records to the *output file*, where a matching record may be an exact match, or the closest match determined by the mismatch key algorithm. If no matches for the query are found (i.e. the trie is empty), your program should output NOTFOUND. When outputting data records, you should follow the guidelines described in the slide ***Dataset and Assumptions***.
  + In addition to outputting the record(s) to the output file, the number of records found and the comparisons and node accesses when querying should be output to stdout .

**Mismatch in Key**

Since a Patricia tree is a type of prefix tree, any mismatch that occurs will contain the same prefix. For example, say we are searching for "trie", but our dictionary contains "tree". Here, a mismatch occurs at some binary bit in the third character. This mismatch will only occur once we are past the node in our tree that contains all prefixes with "tr". Therefore, when the mismatch occurs, we can get all descendants at the mismatched node and calculate the most likely candidate using the edit distance of the query and a key. This way, if we query "trie", we should return "tree". This is explained with the following graphic:



The patricia tree contains 4 strings as listed. If we query "trie", the mismatch happens on the bit highlighted in red. At this node, the only descendant data is "tree", and is therefore determined to be the closest match.

If we query "Tree", the mismatch happens on the third bit. This means that all descendant data is taken to be possible candidates, so all 4 keys have their edit distance against the query calculated. "tree" is determined to be the closest match with an edit distance of 1, and is returned.

**Tips:**

* The results struct you created to hold comparisons can hold matching data
* A node access count should be incremented each time a new node of the trie is looked at.
* If two strings have an equal edit distance, return all results for the first suburb name encountered with that edit distance (this will be the alphabetically earliest result).

Edit Distance

In Stage 4, you will need to calculate the edit distance of multiple strings to determine the closest match. You are welcome to use the code here to calculate the edit distance - you do not have to worry about its complexity:

int editDistance(char \*str1, char \*str2, int n, int m);

int int min(int a, int b, int c);

/\* Returns min of 3 integers

reference: https://www.geeksforgeeks.org/edit-distance-in-c/ \*/

int min(int a, int b, int c) {

if (a < b) {

if(a < c) {

return a;

} else {

return c;

}

} else {

if(b < c) {

return b;

} else {

return c;

}

}

}

/\* Returns the edit distance of two strings

reference: https://www.geeksforgeeks.org/edit-distance-in-c/ \*/

int editDistance(char \*str1, char \*str2, int n, int m){

assert(m >= 0 && n >= 0 && (str1 || m == 0) && (str2 || n == 0));

// Declare a 2D array to store the dynamic programming

// table

int dp[n + 1][m + 1];

// Initialize the dp table

for (int i = 0; i <= n; i++) {

for (int j = 0; j <= m; j++) {

// If the first string is empty, the only option

// is to insert all characters of the second

// string

if (i == 0) {

dp[i][j] = j;

}

// If the second string is empty, the only

// option is to remove all characters of the

// first string

else if (j == 0) {

dp[i][j] = i;

}

// If the last characters are the same, no

// modification is necessary to the string.

else if (str1[i - 1] == str2[j - 1]) {

dp[i][j] = min(1 + dp[i - 1][j], 1 + dp[i][j - 1],

dp[i - 1][j - 1]);

}

// If the last characters are different,

// consider all three operations and find the

// minimum

else {

dp[i][j] = 1 + min(dp[i - 1][j], dp[i][j - 1],

dp[i - 1][j - 1]);

}

}

}

// Return the result from the dynamic programming table

return dp[n][m];

}

Example Execution

make -B dict4

./dict4 4 tests/dataset\_15.csv output.txt < tests/test1.in

Example Output

Output file:

Carlton -->

COMP20003 Code: 9773, Official Code Suburb: 20495, Official Name Suburb: Carlton, Year: 2021, Official Code State: 2, Official Name State: Victoria, Official Code Local Government Area: 24600, Official Name Local Government Area: Melbourne, Latitude: -37.8004392, Longitude: 144.9680900

South Melbourne -->

COMP20003 Code: 390, Official Code Suburb: 22313, Official Name Suburb: South Wharf, Year: 2021, Official Code State: 2, Official Name State: Victoria, Official Code Local Government Area: 24600, Official Name Local Government Area: Melbourne, Latitude: -37.8252526, Longitude: 144.9518185

stdout:

Carlton --> 1 records - comparisons: b58 n4 s1

South Melbourne --> 1 records - comparisons: b52 n5 s1

Stage 5: Complexity Report

The final deliverable for this assignment is a small report analyzing the complexity of both dictionaries. You will run various test cases through your program to test its correctness and also to examine the number of key comparisons used when searching keys across different dictionaries. You will report on the number of comparisons used by your linked list and patricia tree implementations for different data file inputs and key prefixes. You will compare these results with what you expect based on theory (Big-O).

Your approach should be systematic, varying the size of the files you use and their characteristics (e.g. sorted/unsorted, duplicates, range of key space, length of prefix, etc.), and observing how the number of key comparisons varies given different sizes of key prefixes. Looking at statistical measures about the results of your tests may help.

UNIX programs such as sort (in particular -R), head, tail, cat and awk may be valuable. You may also find small C programs (or additional "Stages") useful, the test data you produce and use does not have to be restricted to the dataset provided.

Additionally, though your C code should run on Ed, you may like to run offline tests or other work outside of Ed and are welcome to do so.

You will write up your findings and submit them along with your Assignment Submission in the Assignment Submission Tab.

Make sure to click the "Mark" button after uploading your report. It is often the case for results to be released and more than a handful of students to find an unexpected 0 mark for the report due to not technically submitting it for marking.

You are encouraged to present the information you've collected in appropriate tables and graphs. Select informative data, more is not always better.

In particular, you should present your findings clearly, in light of what you know about the data structures used in your programs and in light of their known computational complexity. You may find that your results are what you expected, based on theory. Alternatively, you may find your results do not agree with theory. In either case, you should state what you expected from the theory, and if there is a discrepancy you should suggest possible reasons. You might want to discuss space-time trade-offs, if this is appropriate to your code and data.

It is recommended to look at extreme cases where each option will do better or worse - e.g. think about dense data where the Patricia tree will save many bit comparisons and character sparse data where string comparisons would terminate quickly in both data structures. It is also highly recommended to look at the theory covered so far and how you can match it - i.e. if a particular complexity is expected, that it appears at large input sizes and represents the *growth* behaviour - not just single point comparisons. Similarly, pay particular attention to the meaning of variables, character comparisons, string comparisons and bit comparisons may all have different behaviours and certain investigations might expect different relations on these.

You are not constrained to any particular structure in this report, but a well laid out plan with a strong structure tends to lead to the best results. A more general and helpful guiding structure is shown in the Additional Support slide, but you may find it valuable to keep in mind a general flow:

* Introduction - Summary of data structures and inputs.
* Linked list and Patricia Tree:  
  - Data (data file, number of keys, characteristics, summary statistics for relevant metrics)  
  - Comparison with theory
* Discussion

Indicatively, around 4 pages is typical for the report. Though a hard page limit is not imposed. But please consider that markers will need to read your report so aim to make it reasonably concise if you can.

7. Implementation Requirements

The implementation requirements are broadly the same as for Assignment 1. Though the data structure requirements differ slightly. The comparison count matching requirements are also slightly varied to reflect differences in counting that may be equally sensible to the recommended counting method.

The following implementation requirements must be adhered to:

* *Programming Language:* You must write your code in the C programming language.
* *Record Structure:* Each data record (representing a suburb) must be stored in a custom structure (struct) that reflects the previously mentioned data types. This struct will have member variables for each field (Year, Suburb Code, etc.) with the appropriate data types (int, double, string).
* *Linked List Implementation:* You must use a linked list to implement the dictionary in Stage 3. The order in which data records appear in the input file must be preserved within the linked list.
* *Patricia Tree Implementation:* You must use a Patricia tree to implement the dictionary in Stage 4. The order in which data records appear in the input file must be preserved within the Patricia tree (i.e. duplicates should appear in the same order as the relative order in the original file).
* *Modular Design:* You must write your code in a modular way. This means your dictionary operations should be kept together in a separate .c file, with its own header .h file, separate from the main program. Other distinct modules should similarly be separated into their own sections, requiring as little knowledge of each other's internal details as possible. This facilitates easier use in other programs without extensive rewriting or copying.
* *Multiple Dictionary Support:* Your code should be easily extensible to handle multiple dictionaries. The functions for interacting with your dictionary should take arguments that include not only the values required for the operation but also a pointer to a specific dictionary (e.g., search(dictionary, value)).
* *Single File Read:* Your implementation must read the input file only once.
* *Space-Efficient Strings:* Your program should store strings in a space-efficient manner. If you use malloc() to allocate space for a string, remember to allocate enough space for the final end-of-string character (\0).
* *Exact Output Matching:* Your outputs for sample tests should be exactly identical to the expected output, including the order in which the data appears. This means your program's results must match the provided test cases character-for-character, preserving the order of elements within the output. Your comparison counts **are not required** to match the provided test cases, but you should understand and note why these do not match by noting how you count these if you do not match the provided test cases.
* Makefile**:** A full Makefile is not provided. The Makefile should direct the compilation of your program. To use it, ensure it's in the same directory as your code. Type make dict3 to build the first program and make dict4 to make the second program. You must submit your Makefile with your assignment.

Hints:  
• If you haven’t used make before, try it on simple programs first. If it doesn’t work, read the error messages carefully. A common problem in compiling multifile executables is in the included header files. Note also that the whitespace before the command is a tab, and not multiple spaces.  
• It is not a good idea to code your program as a single file and then try to break it down into multiple files. Start by using multiple files, with minimal content, and make sure they are communicating with each other before starting more serious coding.

10. Assessment

There are a total of 15 marks given for this assignment.

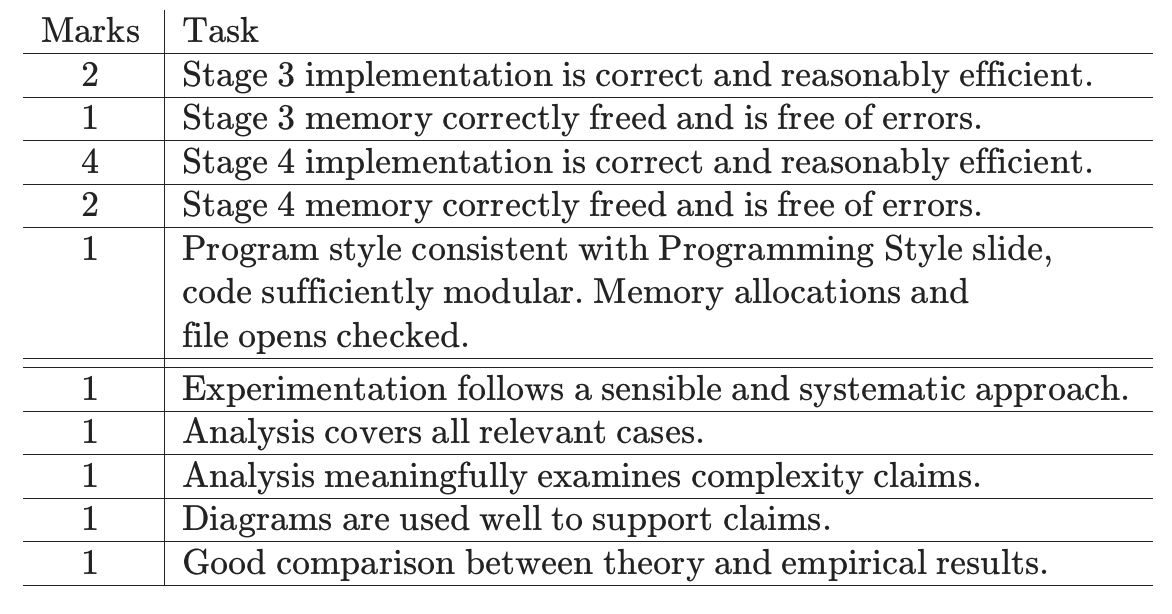
Your C program will be marked on the basis of accuracy, readability, and good C programming structure, safety and style, including documentation (1 mark). Safety refers to checking whether opening a file returns something, whether malloc()s do their job, etc. The documentation should explain all major design decisions, and should be formatted so that it does not interfere with reading the code. As much as possible, try to make your code self-documenting by choosing descriptive variable names.

It is common to lose marks for modularity and efficiency because the ***Requirements*** slide was not adequately read, make sure to double check requirements at intervals during the writing of your program and to check Ed regularly.

The remainder of the code-related marks will be based on the correct functioning of your submission.

The second component of the rubric is about the associated analysis which compares the performance between the two approaches (Stage 3 and Stage 4) - this analysis may not look too difficult at first glance, but is non-trivial and will likely involve revisiting the lectures from earlier weeks to successfully prepare.

Note that unlike the first assignment, passing test cases will not guarantee you marks - for example, your approach *should be reasonably time efficient* (such as sorting and uniqueness checks). Passing test cases is approximately aligned with demonstrated functionality but does not directly evaluate efficiency of code or careful adherence to the specification's Implementation Details, Requirements or Task requirements.



Ungrading

In Assignment 2, we will continue the ungrading experiment. The experimentation has historically been an area where many marks were lost, in grading your work, you are encouraged to mark to reflect further work done to engage with the problem using understanding gained in the subject and in your own research. Since common pitfalls are easy to fall into, you might like to highlight what work has gone into making sure your results are both convincing and as robust as possible.

Note: Getting perfect marks on Assignment 2 is often time consuming - consider taking advantage of ungrading if you are spending far too long on part of the assignment. As mentioned in Assignment 1, you may find a partial solution where you have learned a lot is still valuable. However, make sure you give each area sufficient time - e.g. the report is allocated one third of the marks, not completing it because you spent too long on the other parts is not likely to justify receiving marks for it. The indicative work time is 15 hours, so if you are nearing this, it is worthwhile planning your time to ensure you learn in all areas.

13. Dataset Download

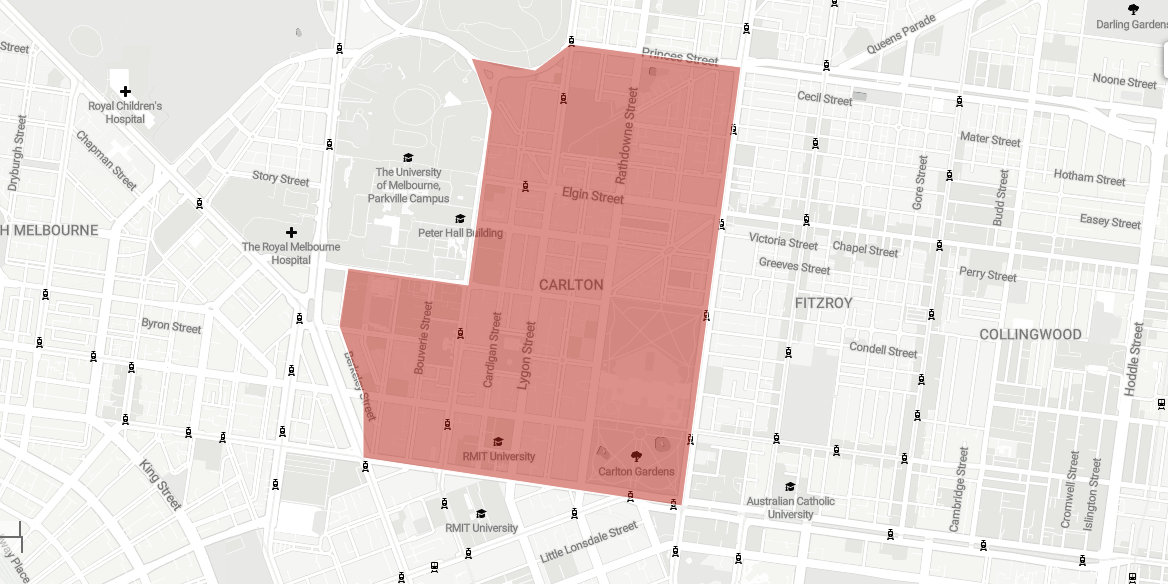
The original dataset (dataset\_full.csv) contains the following suburbs (each represented as a white-edged polygon):

地图

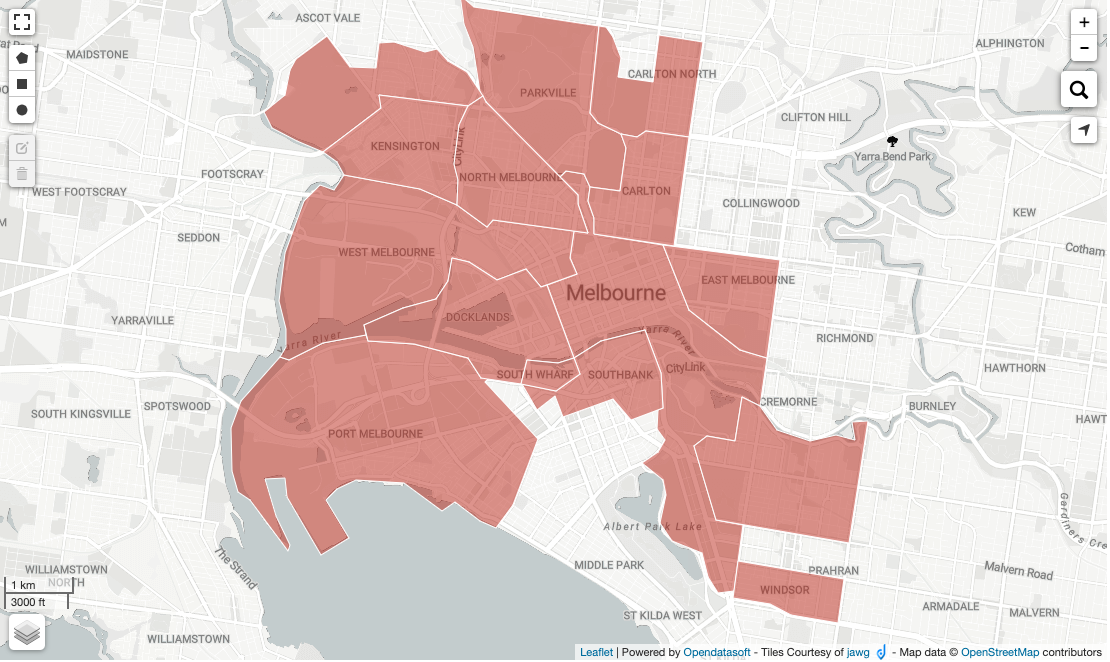
描述已自动生成

This dataset is then sampled for four more datasets of increasing scale. Early datasets are likely to reveal issues in your approach, while the larger datasets are more likely to reveal issues with your memory allocation, etc.

* dataset\_1.csv - 1 suburb



* dataset\_15.csv - 15 suburbs



* dataset\_100.csv - 100 suburbs (no map provided)

**Note** that in Assignment 2 - dataset\_100 has all duplicates removed, removing 51 suburbs with COMP20003 Codes:  
- 1986, 2856, 3201, 3670, 4278, 4338, 4794, 5549, 5926, 6139, 6165, 6206, 6466, 6499, 6636, 7035, 7057, 7302, 7720, 7750, 7903, 8161, 8205, 8540, 8552, 8972, 9032, 9130, 9272, 9414, 9602, 9993, 10486, 10671, 11249, 11480, 11627, 12364, 12417, 12570, 12786, 12852, 14412, 14446, 14507, 14539, 14551, 14736  
This affects the suburbs:  
- Cedar Creek, Springfield, Seddon, Windsor, Kensington, Kingswood, Williamstown, Ascot, Red Hill, Richmond, Ascot, Glenroy, Greenlands, Carlton, Back Creek, Buckingham

* dataset\_1000.csv - 1000 suburbs (no map provided)

14. Assignment Submission

**Code Submission:**

* **File Upload:** Upload your C code files (including your Makefile and report and any other files required to run your code) here. Your programs must compile and run correctly on Ed.
* **Cross-Environment Compatibility:** While you may have developed your program in another environment, it still needs to compile and run successfully on Ed at the time of submission. Due to potential compiler differences, it's recommended to upload and test your code on Ed regularly, especially if you're working in a different environment.
* **Missing Files:** A common reason for compilation errors is accidentally omitting a file from the submission. Please double-check your submission and resubmit all necessary files if required.

**Testing:**

* **Test Cases:** Test cases for the first 9 marks are available here. The remaining 6 marks will be assessed manually.
* **Character Limit:** Marking feedback is limited to 50,000 characters, so some output may be truncated.
* **Sample Usage:** A set of tests is provided for your reference. Here's an example of how to use them:

./dict3 stage tests/dataset\_1.csv output.out < tests/test1.in > output.stdout.out

This command should execute your first program (dict3) with the following arguments:

* stage: an argument specifying the stage, must be 3 (if the executable is dict3) or 4 (if the executable is dict4) for this assignment.
* tests/dataset\_1.csv: The input data file
* output.out: The file where your program's output will be written
* Redirects input (<) from tests/test1.in
* Redirects standard output (>) to output.stdout.out

The expected outputs in tests/test1.out and tests/test1.stdout.out should then match the corresponding files you created (output.out and output.stdout.out).

* **Remaking Your Code:** You can remake your code from scratch (the -B flag forces it to ignore any existing up-to-date elements from the dependencies) using

make -B dict3 dict4

The full suite of tests are:

./dict3 3 tests/dataset\_1.csv output.out < tests/test1.in > output.stdout.out

./dict3 3 tests/dataset\_15.csv output.out < tests/test15.in > output.stdout.out

./dict3 3 tests/dataset\_100.csv output.out < tests/test100.in > output.stdout.out

./dict3 3 tests/dataset\_1000.csv output.out < tests/test1000.in > output.stdout.out

./dict3 3 tests/dataset\_full.csv output.out < tests/testfull.in > output.stdout.out

./dict4 4 tests/dataset\_1.csv output.out < tests/test1.in > output.stdout.out

./dict4 4 tests/dataset\_15.csv output.out < tests/test15.in > output.stdout.out

./dict4 4 tests/dataset\_100.csv output.out < tests/test100.in > output.stdout.out

./dict4 4 tests/dataset\_1000.csv output.out < tests/test1000.in > output.stdout.out

./dict4 4 tests/dataset\_full.csv output.out < tests/testfull.in > output.stdout.out

Testing with valgrind will typically follow a pattern similar to:

valgrind ./dict3 3 tests/dataset\_1000.csv output.out < tests/test1000.in > output.stdout.out

or

valgrind --track-origins=yes --leak-check=full ./dict3 3 tests/dataset\_1000.csv output.out < tests/test1000.in > output.stdout.out

Note that testing using the Mark button may take a while to run as the tests are heavier duty than the tests in the workshop questions.