8-2013

Experimental Studies of Android APP Development for Smart Chess Board System

Srujan Gopu
Western Kentucky University, srujan.gopu505@topper.wku.edu

EXPERIMENTAL STUDIES OF ANDROID APP DEVELOPMENT FOR SMART CHESS BOARD SYSTEM

A Thesis
Presented to
The Faculty of the Department of Computer Science
Western Kentucky University
Bowling Green, Kentucky

In Partial Fulfillment
Of the Requirements for the Degree
Master of Science

By
Srujan Gopu

August 2013

EXPERIMENTAL STUDIES OF ANDROID APP DEVELOPMENT FOR SMART CHESS BOARD SYSTEM

Date Recommended 8/8/13

Dr. Guangming Xing, Director of Thesis

Dr. Qi Li

Dr. Zhonghang Xia

8-9-13

Dean, Graduate Studies and Research Date

ACKNOWLEDGMENTS

It was a great pleasure working under my graduate advisor, Dr. Guangming Xing, who provided me with everything I need to succeed. His inspiration and guidance at each and every step made this Master of Science degree so rewarding and satisfactory. He always encouraged my work in every possible way and also gave me the freedom to express and implement my ideas without any restrictions. I feel very fortunate and proud to have been his student and really think the experience which I gained working under him is invaluable. I would like to whole heartedly thank Dr. Xing for the immense trust and patience he has over me. He constantly supported and directed me in each and every step. Without him this thesis would not have been so successful.

I would like to thank Dr. Qi Li and Dr. Zhonghang Xia for their valuable time and suggestions that helped me improve this Thesis. I would like to thank all my friends at Western Kentucky University, my friends in India and my family for the never-ending support. I would like to specially thank my parents for their everlasting love and encouragement for me to succeed.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

EXPERIMENTAL STUDIES OF ANDROID APP DEVELOPMENT FOR SMART CHESS BOARD SYSTEM

Srujan Gopu August 2013 50 Pages

Directed by: Guangming Xing, Qi Li, and Zhonghang Xia

Department of Computer Science Western Kentucky University

Playing chess on a smart phone has gained popularity in the last few years, offering the convenience of correspondence play, automatic recording of a game, etc. Although a good number of players love playing chess on a tablet/smart phone, it doesn't come close to the experience of playing over the traditional board. The feel and pleasure are more real when playing face down with the opponent sitting across each other rather than playing in mobile devices. This is especially true during chess tournaments. It would be ideal to enhance the experience of playing chess on board with the features of chess playing on smart phones. Based on the design of a roll able smart chess board, an android app has been implemented to interact with the board. It reads signals from the smart chess board and maps the movements of the chess pieces to the phone. The recorded play would be used as input for game analysis. The design and implementation of a server for playing and reviewing a game online have also been studied in this thesis.

CHAPTER 1: INTRODUCTION

1.1 Background

Technology has changed a lot of things around us and the way we play games is one of the many. We can now play a lot of games virtually with the help of different gaming devices and more recently with the advancement in the technology of mobile apps. Chess has always been one of the most interesting games from the pristine ages. It appeals to a wide variety of enthusiasts. Undoubtedly, chess was one of the first games to be played on the computer with artificial intelligence; it is a very ancient game and can be played with minimum requirements online, as it doesn't require any extra sensory devices. It is also a well-known fact that playing chess improves intelligence. Although playing chess requires two players, it can be played on a computer with the help of artificial intelligence.

With the advancements in technology, chess has been taken to a whole different level by making the game more interesting and accessible on devices such as mobile phones and tablets. Chess players across the globe can have a quick game by virtually sitting across the table with the devices in their hand. Given the choice, however, most chess players would pick playing a real game over playing a virtual game. The feeling and pleasure are more real when playing face down with the opponent sitting across from you. Also, understanding the different patterns of moves which lead to eventual win or loss of the game are important if one wants to understand the game in depth and master it. Recently, with advancements in technology and increased access to social media, many applications have been created for chess games. All these applications follow the same genre and they do not possess the capability of displaying

the leading and losing probabilities effectively. There is no technology currently in place which reads the moves from real chess board and maps it on the mobile device.

1.2 Aim and objectives

The aim of this research is to develop a mobile application which would read the chess moves from the smart chess board and record them to the database. The mobile application communicates with the smart chess board using USB, when a game is being played on the smart chess board the mobile device is connected to the board. Any step made on the chess board will be recorded to the mobile application, at the end of the game the whole game can be replayed on the mobile device. The idea behind this feature is that the steps in a recorded game can be looked through after the game and serves as a retrospective for the player.

The recorded steps are also important for statistical purposes. Each and every step taken by the player is analyzed to understand its direction towards success or failure of that game. This statistical data is again used as a guidance system to let the user know his probability of winning a game based on the step taken by the player. The application is not only compatible with the smart chess board but also the player can register himself to log-in and look for players online. Even the game played online is recorded on the server for recording all the steps taken by the player. Application should be a portable that is it can be installed on any platform. It should be cross-platform application.

The following three questions are studied in this thesis:

1) How to read the moves of the players from the chess board, record them and map them to the game in the mobile device, which would also record all the moves made by players.
2) How to keep track of the progress in the chess game to incorporate a mechanism that can analyze the position of the player and help make the best move at any given time in the game. This not only increases the chances of winning but also aids in the learning process.
3) How to enable a user to play the game online over the Internet, in which case, a player can randomly select another player and play. The game moves of the players in the online game would be recorded to the server application.

1.3 Approach to the problem

Based on the functionalities of the modules in the system, there are three components in the application:

Component 1: The component of the application is developed which read the signals from the smart chess board using USB. The pattern of the whole chess board can be represented in a 16 byte data chunk. This component reads 16 byte of data every time a move is made on the chess board. For each and every move made on the chess board, the corresponding move is encapsulated in a 16 byte data chunk and transferred to the smart phone using USB. The application then reads the 16 byte data and maps the corresponding move on the mobile device.

Component 2: A parser for the application is developed. The parser basically analyses millions of chess games and all the moves in each game to develop a statistical database on the server. This database is later used to determine how each step taken by the player will increase or decrease his probability of winning or losing the game. The process of collecting the games and creating the required statistics would necessitate a well-organized data structure that can perform the operations such as insertion, deletion and search. Though there are numerous existing data structures, they operate inefficiently with large amounts of data which slows down the search function.

Component 3: In this phase the component of the application is developed which enables two players to play a chess game online. The whole game is recorded using web-services. Every time a new game is started or any steps are played by the player, they are recorded in the cache of the server through web service. When a player logs in, it displays a list of users who are online. The player has a choice to pick his/her opponent from this list. There is a clock component which synchronizes each move in a timely fashion. After each move is made, the clock halts to allow the opponent to make his/her move.

1.4 Definition of Terms

Tree

Tree is a collection of nodes and links where each node contains a value. The height of a tree is the longest path from root node to leaf node. There are different types of trees like Binary search tree, AVL tree, red-black tree and splay tree which differ in their properties. It performs operations like inserting, deleting and searching. Node is a basic element of unordered data structure which contains a value or a condition. Each node in a data structure has zero or more child nodes. A node that has a child is called an

ancestor node. A leaf node is a node which doesn't have any childs. The peak node in a tree is called root node.

Activation code

An ID issued by the GCM servers to the Android application that allows it to receive messages. Once the Android application has the registration ID, it sends it to the 3rd-party application server, which uses it to identify each device that has registered to receive messages for a given Android application. In other words, a registration ID is tied to a particular Android application running on a particular device.

1.5 Thesis structure

The remainder of the thesis is organized as follows. Chapter 2 presents the architecture of the system and parsing phases. Chapter 3 describes how trie data structure used in chess games parsing and its effective results of trie. Implementation of trie data structure, REST web service and push communication techniques are discussed in Chapter 4 and conclusions are given in Chapter 5.

CHAPTER 2: ARCHITECTURE OF THE SYSTEM

Typical architecture of the system can be illustrated as shown in Figure1.

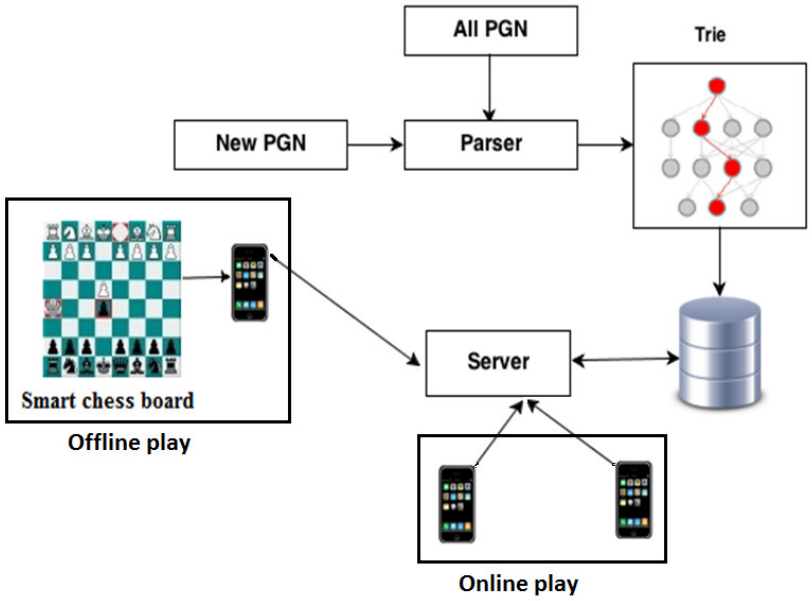

Figure 1: Architecture of the system

The architecture of the system can be divided into three components

1) Offline Play (smart chess board play)
2) Online Play
3) Parser

2.1 Offline Play

Smart chess board is developed by the engineering department of WKU which allows player to play on board. It transfers the movements of the game to the connected mobile. The whole board is divided into 16 sections. Each section can be represented by 8bits. For each and every move made on the chess board, the corresponding move is encapsulated in a 16 byte data chunk and transferred to the smart phone using USB. The application then reads the 16 byte data and maps the corresponding move on the mobile device. No web service is required to play a game in this mode. Even an application not using any web service for game playing, end web service is used to store the game on

database. To read chess moving signals from this board, smart phone has to connect to the board and starts the application. This application reads signal from the board and implicate on the phone with respect to specific moves.

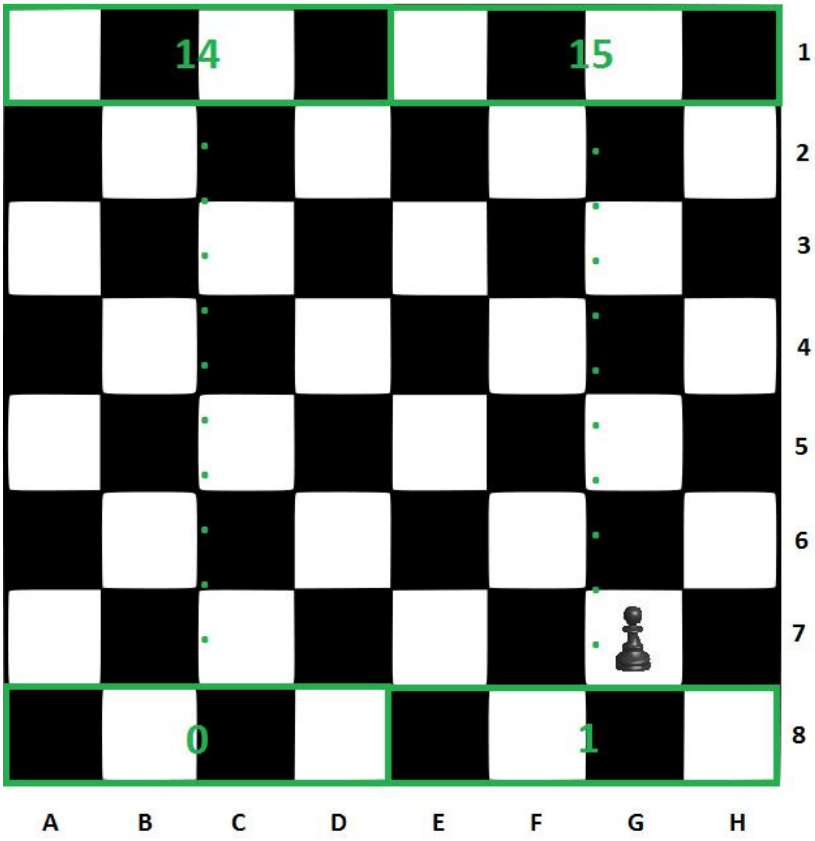

Figure 2: Representation of chess board

The complete board is represented by 16 bytes where each row takes 2bytes. Every consequence 4 squares considered as a section therefore total board contains 16 sections. One byte of data represents one section. On every move operation, application will get 16bytes of data. The sequence of steps which performs move operation is given below.

- Initially, chess game positions are set on an application.
- Application reads 16 bytes of data from the chess board.
- Convert data to binary strings.
- Compare converted binary strings to the previous positions.
- Identify the move positions.
- Perform move operation on the application.
- Repeat these steps up to the end of the game.

Initial chess board positions and its binary representations given below.

Section	Binary String	Equivalent
Section 0	00001111	15
Section 1	00011111	31
Section 2	00101111	47
Section 3	00111111	63
Section 4	01000000	64
Section 5	01010000	80
Section 6	01100000	96
Section 7	01110000	112
Section 8	10000000	128
Section 9	10010000	144
Section 10	10100000	160
Section 11	10110000	176
Section 12	11001111	207
Section 13	11011111	223

Section 14	11101111	239
Section 15	11111111	255

Table 1: Representation of chess board initially

2.2 Online Play

Live chess game playing is totally dependent on web service. When an application initiates, each player has to login to the application. Later it displays a list of players who are available. Player has to select any one from the list to start a game. When a new game is started, both player's session id will be stored on object game at server. All game objects are stored on cache memory. Each game object contains session details and user details like user Id, user name and user attributes. Each user has to register online before login to the application. When player moves a piece on board, it sends a web service request to the server which later pushes the same move to another player. This process continues up to the end of the game. When the game finishes, it sends another web service request to the server which closes the session and saves the game on database. To play any user online, player has to register on server, and as a result each user will get one unique username and password. Clock is a special component that exists in the application. Here there exist 2 clocks: one for black and another for white. Initially, both are set 5 minutes which can be altered later.

2.3 Parser

The main component in this phase is parser, which takes all pgn files from files and computes chess move statistics. These statistics give the leading and loosing percentage of players with the respect of game moves. The parser uses trie data structure to generate statistics which performs operations faster than other data structures. The parser can also

take single pgn files as an argument and alter the existed game statistics without creating the new statistics. After computation of chess game statistics, parser stores them into database. The parser can also read statistics from database. List of parser objectives are given below.

- Get rid of comments and tag list.
- Separate context and result.
- Create an empty trie data structure.
- Add/update trie data structure.
- Update chess game statistics.

Phases of parser

The purpose of parsing is to analyze each chess game and add/update chess game statistics. Each ".pgn" file may contain any number of games. The following figure illustrates a list of operations performed by parser.

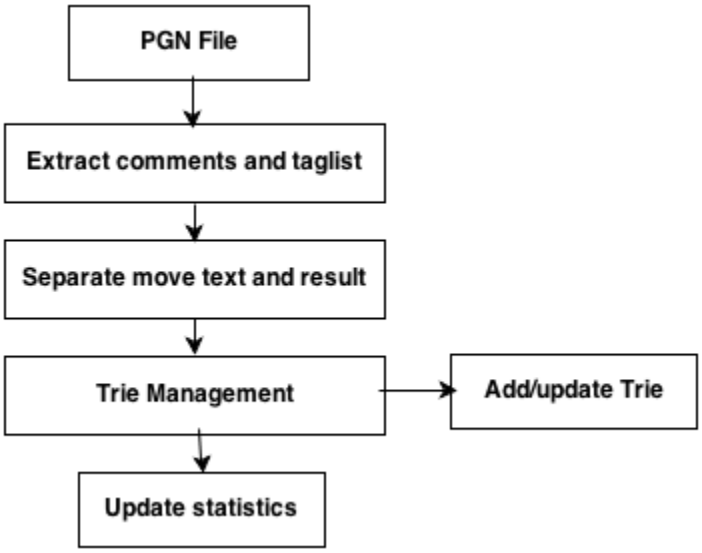

Figure 3: phases of parser

Extract comments and tag list

The initial operation performed by the parser is that it skips the special control codes like escape sequence characters, comments, SPC (superfluous embedded spaces) from PGN. It also extracts the tag list by identifying all statements starts with initial left bracket '['.

Extract move text and result

In the second phase, it breaks the moving text in to small tokens where each token represents a move and the last token can be consider as a game result. This module is connected with trie management to which it passes a list of moves along with result as an argument for each game.

There are different chess notations which records a sequence of moves. For every different notation we use different delimiters to separate a set of moves. Here listed few chess notations.

1. **Algebraic:** In this notation space is considered as delimiter.
2. **Long algebraic:** Here dash (-) is used as delimiter between a set of move. Each move contains from square and to square position.
3. **Reversible algebraic:** It is same as long algebraic but it adds an additional letter for the piece that was captured.
4. **Concise reversible:** It is same like reversible algebraic, but omits the file or rank if it is not needed to disambiguate the move.
5. **Smith:** It is a straight forward chess notation and same like reversible without any delimiter.

Algebraic	Long algebraic	Reversible	Reversible	Smith
e4 e5	e2-e4 e7-e5	e2-e4 e7-e5	e24 e75	e2e4 e7e5
d3 Bb4+	d2-d3 Bf8-b4+	d2-d3 Bf8-b4+	d23 Bf8b4+	d2d3 f8b4
O-O Bxc3	O-O Bb4xc3	O-O Bb4xNc3	O-O Bb4:Nc3	e1g1c b4c3n

Table 2: list of chess game notations

Trie management

The trie management handles different operations on trie data structure like adding a new game or updates an existing game to trie. It performs look up operation on trie before it perform addition to confirm whether the game already existed or not. Trie management adds new game to trie only if new game not exists in trie. Each add/update operation on trie updates all the nodes having on the same path of new game path.

Update Statistics

The parsing of millions of games, results statistics of chess game moves, like it gives a winning and losing probability of white/black piece. For each new PGN game, game statistics are updated. The new statistics are updated based on the following formulas.

- **White winning percentage**

  New number of games white won (NW_{white}) = Number of game white won (W_{white}) + 1;

  W_{temp} = [New number of games white won (NW_{white})/*Total games*]

  White winning percentage = W_{temp} * 100;

- **Black winning percentage**

  New number of games black won (NW_{black}) = Number of game black won (W_{black}) + 1;

  B_{temp} = [New number of games black won (NW_{black})/*Total games*]

  Black winning percentage = B_{temp} * 100;

- **White losing percentage**

  New number of games white lose (NL_{white}) = Number of game white lose (L_{white}) + 1;

  W_{temp} = [New number of games white lose (NL_{white})/*Total games*]

  White losing percentage = W_{temp} * 100;

- **Black losing percentage**

  New number of games black lose (NL_{black}) = Number of game black lose (L_{black}) + 1;

  B_{temp} = [New number of games black lose (NL_{black})/*Total games*]

  Black losing percentage = B_{temp} * 100;

- **Tie percentage**

  New number of games tie (NG_{tie}) = Number of game tie (G_{tie}) + 1;

  T_{temp} = [New number of games tie (NG_{tie})/*Total games*]

  Tie winning percentage = T_{temp} * 100;

CHAPTER 3: METHODOLOGY AND RESULTS

3.1 Trie data structure

A Trie, also called digital tree or prefix tree, is an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings. It stores words with a common prefix under the same sequence of edges in the tree, eliminating the need for storing the same prefix each time for each word.

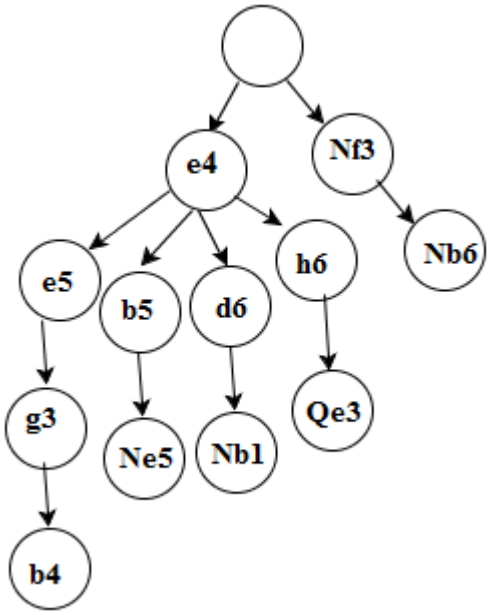

Figure 4: Sample trie data structure

In Figure 4 above the Trie contains the moves [e4, e5, g3, b4], [e4, b5, Ne5], [e4, d6, Nb1].

Let G_1 and G_2 are two games with a sequence of moves $M_1 = (m_1m_2m_3\ldots m_k...m_i)$, $M_2= (m_1m_2m_3\ldots m_k...m_j)$ and prefix $M_p = (m_1m_2m_3\ldots.m_k)$, then these two games follow the same path up to the prefix M_p and path gets separated at the point of m_k then G_1 makes separate path from m_k to m_i and G_2 makes separate path from m_k to m_j. If length of M_1 is denoted with L_1, then the leaf node of G_1 game path would be at the level of L_1.

Each path from root to leaf node represents a game. Leaf nodes in the trie are differentiated with a flag called “terminal” [12]. Each node in the trie contains the following three elements.

- Tag: The element tag contains the value of the node. It acts as a key. No two nodes contain same tag value with same prefix.
- Child: Each node in the trie can have any number of child’s. We can’t limit the number of children’s in the trie so that we don’t get an overflow problem. This element is a small set of records retained as a simple data structure such as a list. All key values of depth‘d’ would have the same length.
- Terminal: It is a boolean type which determines whether a node is leaf node or not. The true value of this terminal represents that the node is leaf i.e. end of the path.

```
class Node {
<access_specifier> <type> tag;
<access_specifier> List of Node type childs;
<access_specifier> boolean terminal;
<access_specifier> <type> other_details;
}
```

Figure 5: Node type ADT

Real World Scenario using trie

Scenario 1: Discovering network motifs.

Finding network motifs is a computationally hard problem, since we have to match the desired motifs with sub graph patterns [3]. This leads to graph isomorphism, which does not have any polynomial time algorithm known. Therefore, the time needed to

discover network motifs grows exponentially as the size of the motifs increases. Tries are a novel data structure specialized in representing collections of sub graphs. Tries are multi-way trees that take advantage of common substructures in the sub graphs [4]. It is easy to implement algorithms for inserting sub graphs, discovering their frequency on another larger graph and on how to break sub graph symmetries that could lead to redundant computations using tries. The proposed data structure minimizes the time required for network motifs operations, including lookup, insertion, and deletion, and also reduces the number of memory accesses. The following mentioned are more advantages of tries over other data structures in designing dynamic router tables.

1. Tries support ordered iteration where as iteration over other data structures will result in a random order.
2. Tries facilitating longest prefix matching [6].
3. Since no additional functions are used, Tries performs a little bit faster than other data structures.

Existing data structures and drawbacks

Binary search trees

In a binary search tree (BST), each tree node stores a string and two pointers to left and right child nodes [7]. A search to find a query string involves, at each node, comparing the query string to the node's string to determine whether the string has been found or, otherwise, whether to branch left or right. At the root, the string comparison typically terminates after inspection of a single character; as the search progresses, the number of characters inspected at each string comparison gradually increases.

A data structure accessed beginning at the root node. Each node is either a leaf or an interior node. An interior node has one or more child nodes and is called the parent of its child nodes. More formally, a connected forest. Contrary to a physical tree, the root is usually depicted at the top of the structure, and the leaves are depicted at the bottom.

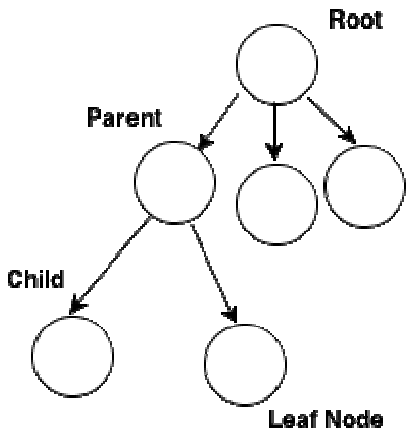

Figure 6: Tree data structure

Although the allocation of strings to nodes is determined by the insertion order, for a skew distribution it is reasonable to expect that common words occur close to the beginning of the text collection and are therefore close to the root of the BST. Assuming the distribution is stable, accesses to a common term should be fast, since the first levels of the tree are usually kept in cache and only a few string comparisons are required. On the other hand, if strings are inserted in sorted order or the distribution changes, the behavior of a BST can be extremely poor. In the absence of a rebalancing technique to prevent sticks from occurring or to move common nodes towards the root, a BST is of limited use in practice for vocabulary accumulation, despite, as in our earlier work, good performance on typical data. There are several well-known variants of BSTs that maintain balance or approximate balance, in particular AVL trees [8] and red-black trees

[9]. With these techniques, the tree is reorganized on insertion or deletion, thus ensuring that leaves are at approximately the same depth. On the one hand, use of rebalancing ensures that for n nodes there is an O(logn) upper limit to the length of a search path. On the other hand, the rebalancing does not consider the frequency of access to each node, so common words can be placed in leaves.

Advantages:

- Stores keys in the nodes in a way that searching, insertion and deletion can be done efficiently.
- Implementation of binary search tree is simple.
- Nodes in the binary search tree are dynamic.

Dis-advantages:

- The shape of binary search tree is depends on the order of insertions and it is degenerated.
- When inserting or searching for an element, the key of each visited node has to be compared with the key of the element to be inserted/found.
- Keys in the tree may be long and the run time may increase.

Splay trees

A splay tree is a variant of a BST, in which, on each search, the node accessed is moved to the root by a series of node rotations, an operation known as splaying [10]. Splaying has several effects that would be expected to be beneficial for vocabulary accumulation. The following tree demonstrates splaying of node B.

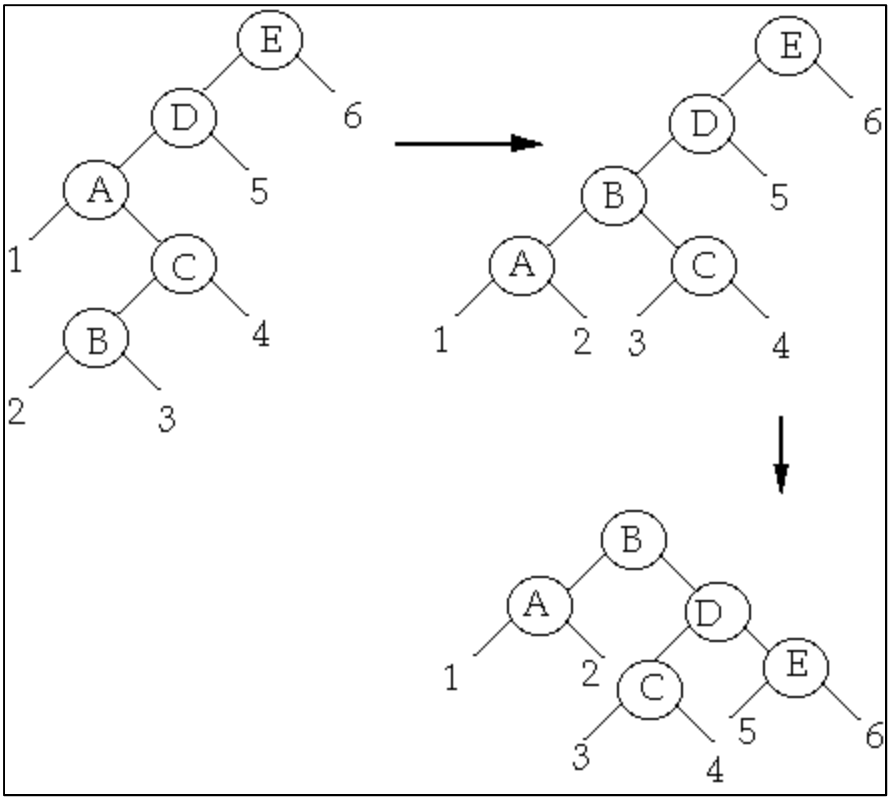

Figure 7: Splay a node B

Intuitively, commonly-accessed nodes should remain near the root, thus allowing them to be accessed rapidly; the tree quickly adapts to local changes in vocabulary; and the use of splaying guarantees that the amortized cost of accessing a tree of n nodes is at most O(logn). Splay trees have significant disadvantages. In comparison to a BST, a splay tree requires more memory, since an efficient implementation of splaying requires that each node have a pointer to its parent. In real time, it is used to implement caches and it has the ability of not to store any data, which results in minimization of memory requirements and It can also be used for data compression.

Advantages:

- It is easy to implement than other self branching binary search trees, such as Red black trees or AVL trees. Much simpler to code than AVL, Red Black trees.
- It requires less space as no balance information is required.

Dis-advantages:

- More local adjustments during Search operations.
- Individual operations can be expensive, drawback for real-time applications.
- The main disadvantage of Splay trees is the height. After accessing all 'n' elements in the tree, the height of the tree corresponds to worst case access time.

Hash tables

The most efficient form of hash table for vocabulary accumulation is based on chaining. In such hash tables, a large array is used to index a set of linked lists of nodes, each of which is said to be at a slot. On search, an array index is computed by hashing the query string. The string is then sought for in the linked list for that index. The items in hash tables are ordered randomly based on how hash function results [15]. Therefore, there is no efficient way to locate an entry whose key is nearest to a given key. It exhibits poor performance when the number of data items increases.

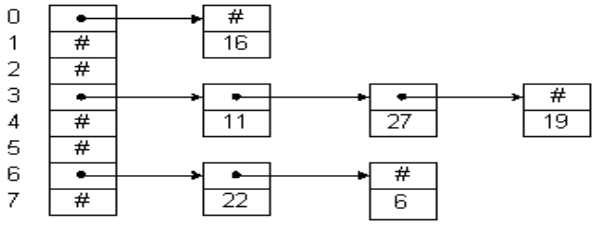

Figure 8: Hash table

Advantages:

- Hash tables are efficient when the maximum number of entries can be predicting in advance.
- Hash function is collision free.

Dis-advantages:

- It is not efficient when the entries are very small like string processing applications, such as spell-checking hash tables may be less efficient than tries.
- Listing all *n* entries in some specific order generally requires a separate sorting which increases the cost of an operation.
- Although the average cost per operation is constant and fairly small, the cost of a single operation may be quite high.
- The data to be accessed is distributed seemingly at random in memory is poor. Because hash tables cause access patterns that jump around, this can trigger microprocessor cache misses that cause long delays.

Performance analysis of trie:

— **Search speed analysis:** The quality of search is very difficult to verify, and can vary from one application to another application depending on the different frameworks and methodologies. In order to qualify the search results retrieved from the experiment is considered as an important parameter to judge the efficiency of the search performance.

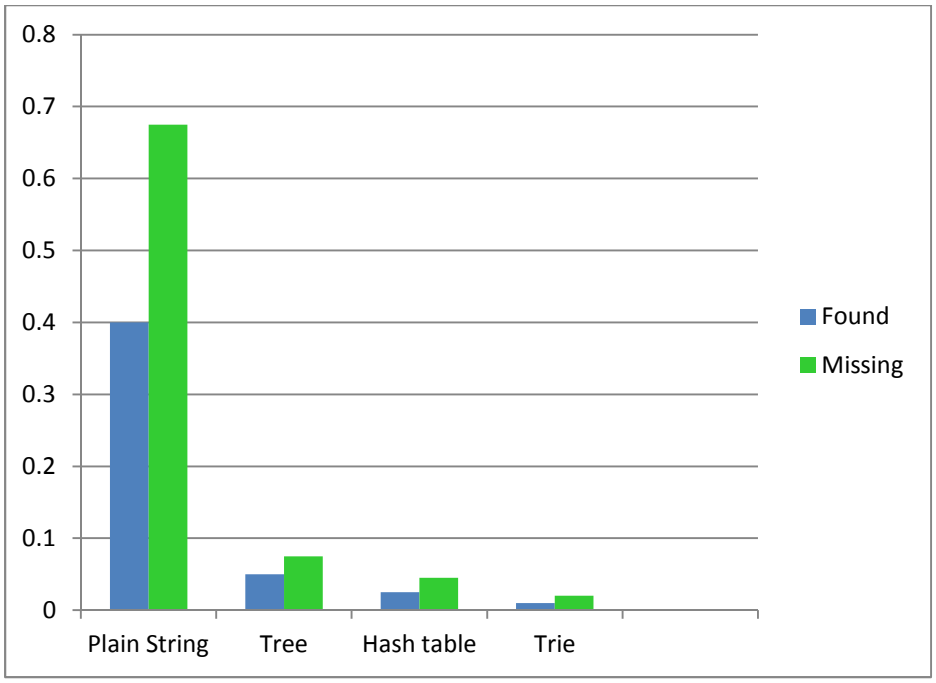

Figure 9: Representation of search speed analysis

As the above graph represents, this is a linear search through the string and will undoubtedly be much slower than using a Hash table, tree or a Trie lookup.

— **Memory usage analysis:** Memory is one of the main factors to calculate the performance. Using too much memory will cause an application to take more time for executing. Trie needs a linear storage space of $\sum^{n}_{i=0} L(i) = O(n)$ bytes to save the n moves stored in it, where L(i) is the length of the i^{th} move. It needs 2n-1 nodes to store exactly n moves so, the number of nodes after level compression will be less than or equal to 2n-1. The representation of memory usage of different data structures is given below.

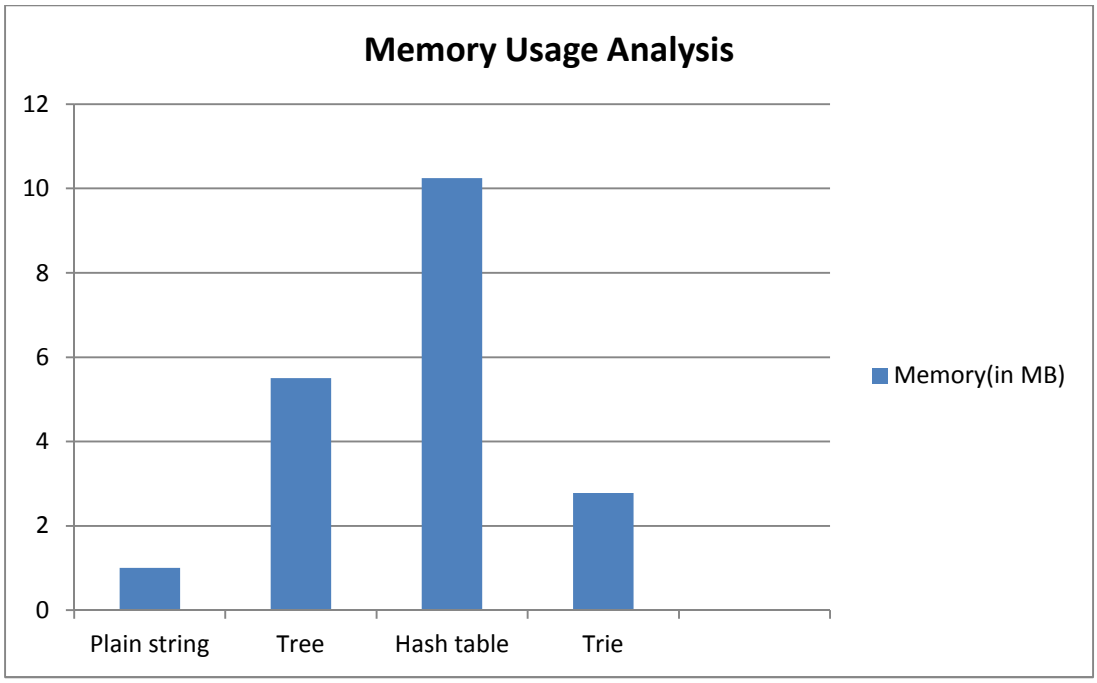

Figure 10: Representation of memory usage

— **Load speed analysis:** load speed performance is calculated based on the time taking to load pgn game from file to memory. It is a little bit difficult to calculate with accurate results because load speed performance depends on system configuration. It varies from one system to another system.

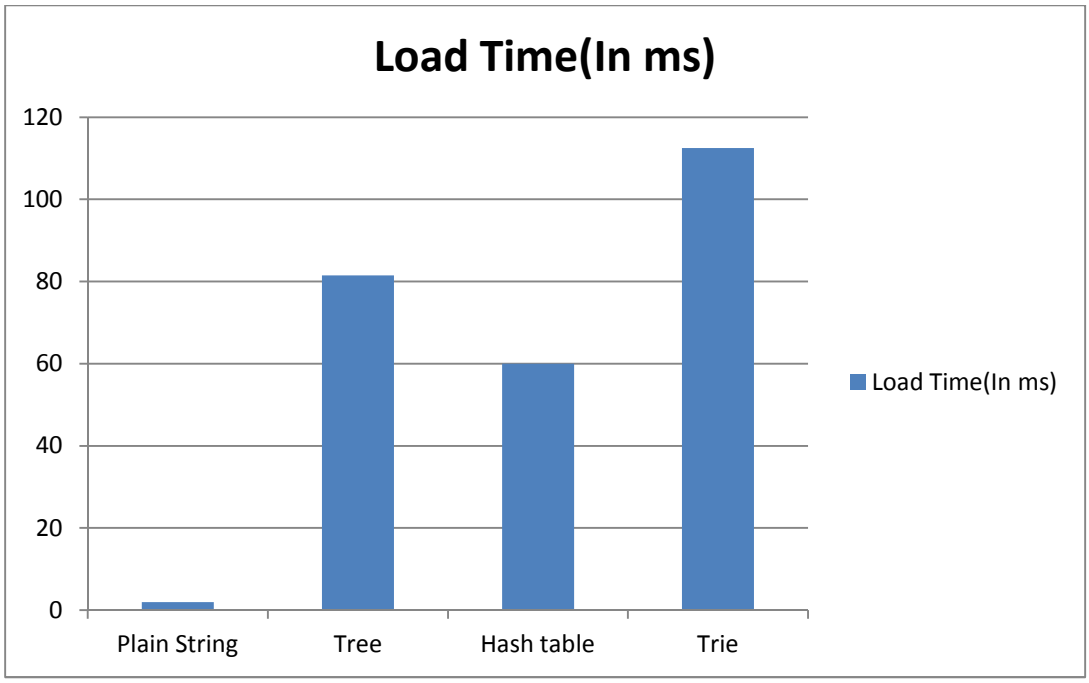

Figure 11: Representation of load speed analysis

3.2 PGN Format

The process of parsing is analyzing a chess game along with its result from a pool of chess games. Chess games are recorded in the format of PGN (Portable Game Notation) supported by many chess programmers.

Portable game notation is a standard, well-organized format of chess game which can be easily readable and writable by human users. The extension of filename is ".pgn". PGN is for recording and describing the moves in a game of chess. It is a set of tag pairs and move text. Each tag pair contains tag name along with tag value and it starts with initial left bracket '['followed by tag name and tag value in double quotes. Each tag set is ended with right bracket ']'. No escape sequence characters, carriage returns and special characters in PGN file. Comments can be given in PGN either using semi colon ';' or a set of braces "{}".

```
[Event "WKU Match"]
[Site "Bowling green, KY,USA"]
[Date "1992.11.04"]
[Round "29"]
[White "Gopu, Srujan"]
[Black "Raja, David"]
[Result "1/2-1/2"]

1. e4 e5 2. Nf3 Nc6 3. Bb5 a6
4. Ba4 Nf6 5. O-O Be7 6. Re1 b5 7. Bb3 d6
8. c3 O-O 9. h3 Nb8  10. d4 Nbd7
11. c4 c6 12. cxb5 axb5 13. Nc3 Bb7 14.
Bg5 b4 15. Nb1 h6 16. Bh4 c5 17. dxe5
Nxe4 18. Bxe7 Qxe7 19. exd6 Qf6 20. Nbd2
Nxd6 21. Nc4 Nxc4 22. Bxc4 Nb6
23. Ne5 Rae8 24. Bxf7+ Rxf7 25. Nxf7 Rxe1
+ 26. Qxe1 Kxf7 27. Qe3 Qg5 28. Qxg5
hxg5 29. b3 Ke6 30. a3 Kd6 31. axb4 cxb4
32. Ra5 Nd5 33. f3 Bc8 34. Kf2 Bf5
35. Ra7 g6 36. Ra6+ Kc5 37. Ke1 Nf4 38.
g3 Nxh3 39. Kd2 Kb5 40. Rd6 Kc5 41. Ra6
Nf2 42. g4 Bd3 43. Re6 1/2-1/2
```

Figure 12: Sample chess game PGN format

As depicted in the above figure, there are 7 tag pairs known as STR (seven tag roaster) explained below.

1. Event: The name of the match event.
2. Site: The location of event.
3. Date: The starting date of game.
4. Round: The playing round ordinal of the game within the event.

5. White: The player name of white pieces.
6. Black: The player name of black pieces.
7. Result: The result of the game. The result “1-0” indicates white winning, “0-1” indicates black winning, “1/2-1/2” indicates tie and “*” indicates the game is ongoing.

The move text describes the actual moves of the game. There are some standard letters of abbreviations used in PGN like ‘K’ for king, ‘Q’ for queen, ‘R’ for rook, ‘B’ for bishop and ‘N’ for knight, and the pawn is given with an empty abbreviation. Every square on chess board represents a unique id which is a combination of characters and numbers. Rows are given with numbers 1 to 8 and columns are given with letters ‘a’ to ‘h’. Therefore, the leftmost square closest to white is ‘a1’; the rightmost square closest to the white is ‘h1’, and rightmost square closest to black side is ‘h8’.

There are some unique moves indicated by special characters such as ‘O-O” for kingside casting, “O-O-O” for queenside casting, pawn promotions are noted by appending “=” to the destination square, ‘+’ sign for indicating checking move and ‘#’ for checkmating move. Each PGN is ended with game result.

3.3 Indexing

Unlike in binary search tree, trie data structure can have any number of Childs. Root node is associated with empty string. In chess game indexing, each move of a game represents a node. When prefix of two games move text is the same, it follows the same path up to the length of prefix. Indexing of pgn files is illustrated with the following example:

```
[Event "WKU event"]
[Site "Bowling Green,KY,USA"]
[Date "1989.5.15"]
[Round ""]
[White "Jessen, Hilke"]
[Black "Freter, Anke"]
[Result "1/2-1/2"]
[ECO "E79"]

1.e4 c5 2.Nf3 d6 3.d4 cxd4 4.Nxd4 g6 5.c4 Bg7 6.Nc3 Nc6 1/2-1/2
```

Figure 13: PGN format of $game_1$

The above pgn game depicts, a series of moves which results the game tie. When the above game interpreted by parser, it extracts all tag list and move text along with result from it.

```
[Event "WKU event"]
[Site "Bowling Green,KY,USA"]
[Date "1989.5.15"]
[Round ""]
[White "Srujan,Gopu"]
[Black "Raja,David"]
[Result "1/2-1/2"]
[ECO "E79"]

1.e4 c5 2.Nf3 d6 3.d4 cxd4 4.Nxd4 g6 5.Bb3 d6 6.d4 Nbd7 1/2-1/2
```

Figure 14: PGN format of $game_2$

If we observe the above two games, they have same prefix up to four moves. Therefore, they follow the same path up to four moves, and then the path would be separated into two. The indexing of these two games is shown below

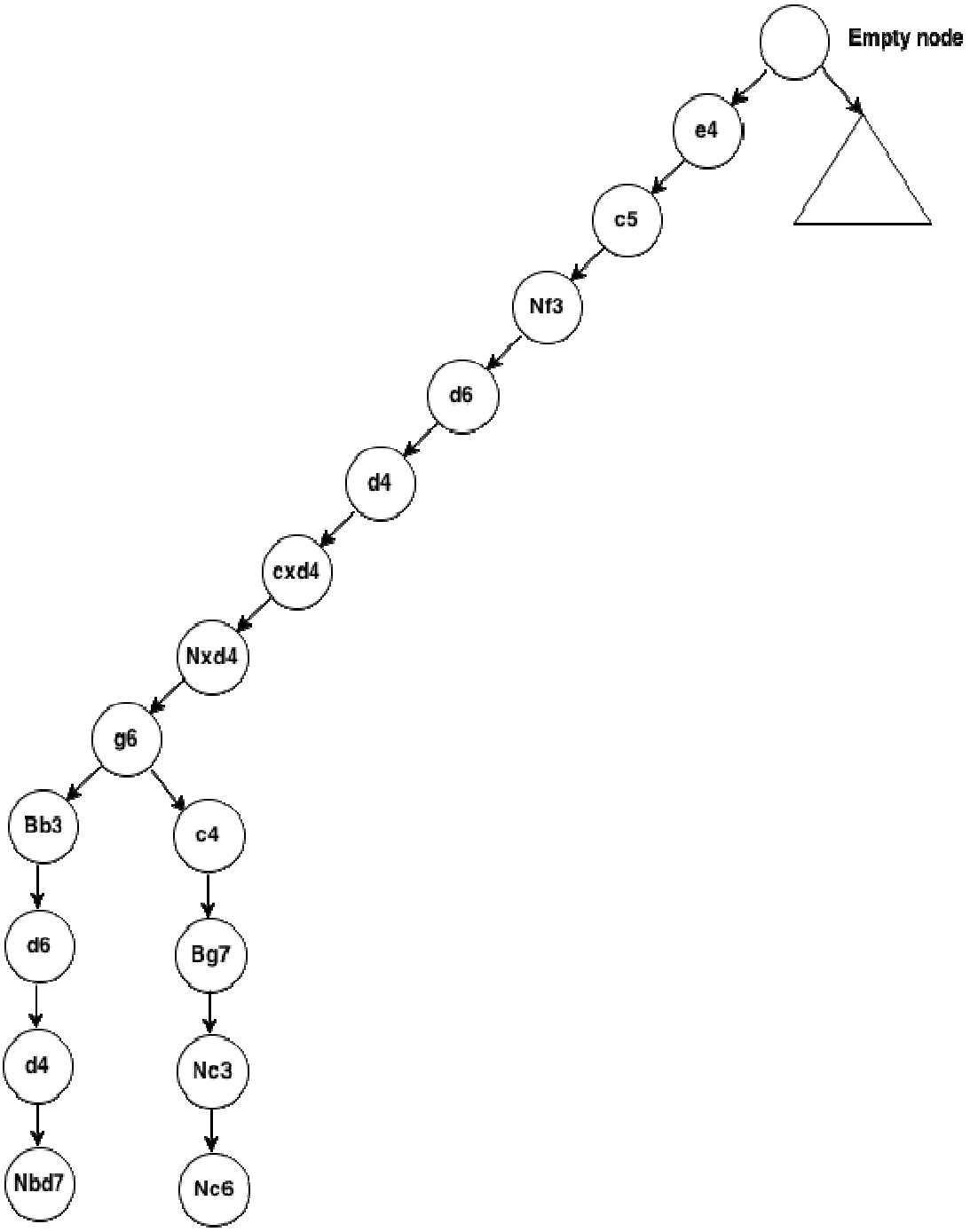

Figure 15: Trie indexing of game1 and game2

Calculation of statistics

The following figure depicts that representation of trie along with number of white winning, black winning and tie games per each node.

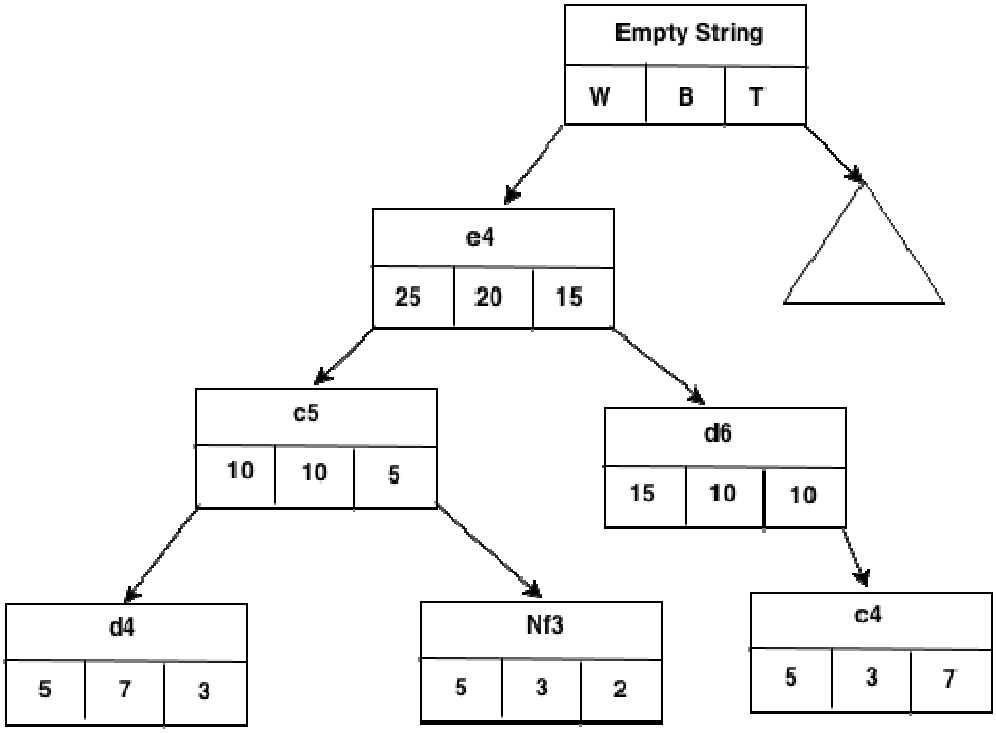

Figure 16: Sample trie with statistics

The total number of games = 25 + 20 + 15 = 60.

If the first move is e4,

Winning percentage = (number of winning games/total games) * 100 = 25/60*100 = 41.66%

Losing percentage = (number of games lose/total games) * 100 = 20/60*100 = 33.33%

Tie percentage = (number of games tie/total games) * 100 = 15/60*100 = 25%

If the moving sequence is [e4,d6,c4],

Winning percentage = 5/15*100 = 33.33%

Losing percentage = 3/15*100 = 20%

Tie percentage = 7/15*100 = 46.67%

If the moving sequence is [e4, c5, d4],

Winning percentage = 5/15*100 = 33.33%

Losing percentage = 7/15*100 = 46.67%

Tie percentage = 3/15*100 = 20%

Database design: Seven tables are used in this database design and they are listed below

- User
- UserAttribute
- Role
- RoleAssignment
- Game
- GameAttribute
- GameStatistic

User table:

Attribute	Type	Primary key
Userid	Long	1
Username	String	0
Firstname	String	0
Lastname	String	0
Password	String	0
passwdExpirationDate	Date	0
Attributes	List<UserAttribute>	0
roleAssignments	List<RoleAssignment>	0
Status	Integer	0

Table 3: User table

UserAttribute table:

Attribute	Type	Primary key
UserAttributeid	Long	1
attributeName	String	0
attributeValue	String	0

Table 4: Userattribute table

Role table:

Attribute	Type	Primary key
roleId	Long	1
Name	String	0
Desc	String	0
Status	Integer	0

Table 5: Role table

RoleAssignment table:

Attribute	Type	Primary key
assignmentid	Long	1
User	User	0
Role	Role	0

Table 6: Roleassignment table

Game table:

Attribute	Type	Primary key
gameid	Long	1
date	Date	0
whiteUser	User	0
blackUser	User	0
gameResult	String	0
Movetext	String	0
Attributes	List<GameAttribute>	0
Status	Integer	0

Table 7: Game table

GameAttribute table:

Attribute	Type	Primary key
gameAttributeId	Long	1
attributeName	String	0
attributeValue	String	0

Table 8: Gameattribte table

GameStatistic table:

Attribute	Type	Primary key
gameStatisticId	Long	1
tag	String	0
White	String	0
Black	String	0
Tie	String	0
Status	Integer	0

Table 9: Gamestatistic table

CHAPTER 4: IMPLEMENTATION AND USER INTERFACE

4.1 Trie implementation

In this section, we put forward the implementation of trie data structure during chess games parsing. The primary design goal for the trie was to reduce the average number of string comparisons during string search. The chess moves are sequential and can be easily simulated in the trie data structure which also comprises of sequential nodes. For each node in the trie is associated with a move in the game, therefore for every different moves in dissimilar games, the level of trie would be increased. Every game initiates from white coin move, as a result all odd level nodes are associated with white coin moves and even level nodes are associated with black moves. The root node is associated with empty string.

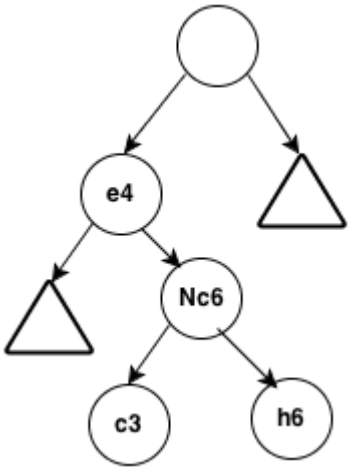

Figure 17: Example of trie data structure.

We now consider the two main operations applied to the trie data structure: insertion and searching.

Searching

Searching for a query string (key) in trie data structure is effortless. This operation always initiates from the root node. When this function gets a sequence of moves as a query string, it starts searching first move at level 1, second move at child nodes of resulted level 1 node, third move at child nodes of resulted level 2 node, and this process continues up to the last move of query string. If all the moves are available in a single path of trie, searching results true. Otherwise it results false. In the worst case scenario, it takes O (m) time to search a key of length ‘m’. Trie data structure performs this function at a faster pace compared to other tree data structures.

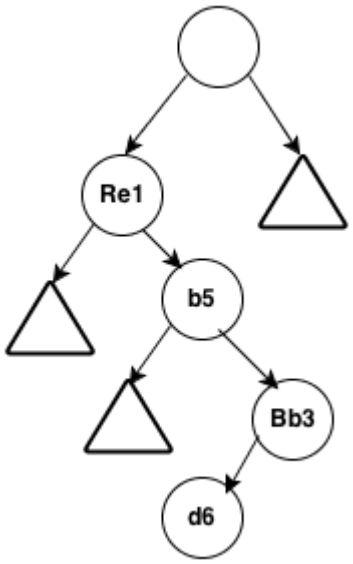

Figure 18: Sample trie for searching a query string

For example, to search for a sequence of moves “Re1, b5, Bb3, d6” in the trie in Figure 3, first move‘Re1’ search in level1, ’b5’ search in level2 and continue so that it returns the reference of node ‘d6’.

Search algorithm

Input for search algorithm is a query string Q = q1, q2….qn of n moves. Output is a reference to the successful search result node or null in the case of unsuccessful search.

```
Algorithm search (Q)
curr_refer ← rootnode
While (for each move q)
for each (pick node m from childs of curr_refer)
if ( q is equal to the tag of m)
return m if this is the last move;
curr_refer ← m
end-if
end-for
end-while
return null;
end
```

Figure 19: Search for a sequence of moves in trie data structure

Insertion

Inserting a sequence of chess game moves to trie is done in two phases: In first phase trie search's whether the given sequence of moves existed in trie or not. If the moves are not existed in trie data structure, it adds new sequence of moves to trie in second phase. After for every insertion operation in trie, all the nodes in the path would be updated and depth of trie increases by one on that path. If any moves having the same prefix, then it merges to the prefix path. Initially, the depth of trie is zero. If no prefix is matching with the given moves, then the given moves are added to the root node.

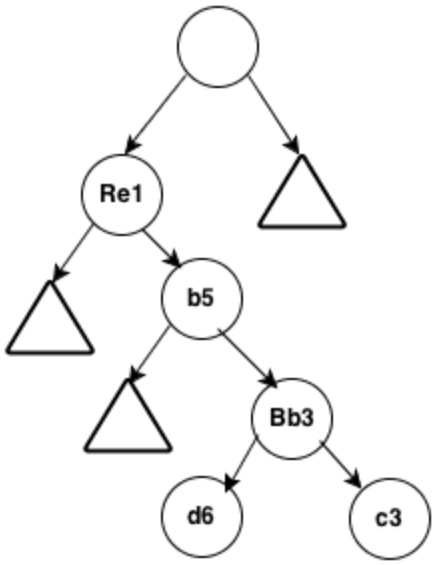

Figure 20: It depicts after insertion of moves “Re1, b5, Bb3, c3” on trie of Figure3.

Insertion algorithm

Input for insertion algorithm is a string $Q = q_1, q_2 \ldots q_n$ of n moves and outcome is an addition of moves to trie data structure.

```
Algorithm insertion(Q)
curr_refer ← rootnode
for i ←1 to n do
result ← search (q1q2..qi)
if (result is equal to null)
create a node with qi and add it to curr_refer
else
curr_refer ← result
end-for
end
```

Figure 21: Algorithm to insert a sequence of moves in trie data structure.

4.2 REST (Representational State Transfer)

REST web service in its rudimentary stage was not famous, but with time and increased usage, it became a renowned web service used globally [13]. REST defines a set of principles using which a user can design a web service between client and server.

- Be state-less: Rest provides a service which is state-less i.e. each request is independent from other. Since it is a state-less, a request can be forward from one server to another as needed to drop the whole response time. The behavior of state-less improves the performance of rest web service and simplifies the design and implementation of server side components because the lack of state on the server removes the need to synchronize session data with an external application. State-less server side components are light-weighted and less-complicated to design and distribute across the servers.
- Directory structure: REST web service URIs should be perceptive, simple, inevitable and easy understanding, for easy accessing of resources. To achieve this level of usability, resources should be defined as directory structure like URIs. Here mentioned few guidelines to make URIs of web service efficient.
    - Avoid file extensions in the web service URI like (.jsp ,.htm ,.php).
    - All characters should be in lowercase.
    - Prefer to user underscores instead of spaces.
- XML Transfer: When client requests to the server, the response of REST web server turned in to xml or JSON. To give client applications the ability to request a specific content type that's best suited for them; construct your service so that it makes use of the built-in HTTP Accept header, where the value of the header is a

MIME type. Some common MIME types used by Restful services are shown in Table 1.

MIME-Type	**Content Type**
JSON	application/json
XML	application/xml
XHTML	application/xhtml+xml

Table 10: MIME types used by web service

This service can be used by variety of clients written in different languages running on different platforms and devices. Among them xml is easy to use

Having described REST web service, its implementation in the research is elicited below

- New game: This web service is used for creating new instance of the game. It requires two player details for creating new game. The template of new game web service is figured below.

<newgame>
<user1> </user2>
<user2> </user2>
</newgame>

Figure 22: Template of new game web service

- Move: This is the web service executed on server for every move operation. Whenever a player moves a coin in the game, client push's move details like 'from', 'to' and 'score' to the server by using move web service. This service records the movements of each game and stores in the records.

<moverequest>
<user1> </user1>
<user2> </user2>
<from> </from>
<to> </to>
<promoteTo> </promoteTo>
<score> </score>
</moverequest>

Figure 23: Template of move web service

- Save game: This web service deals with the aspect of storing a game on database. At the end of each game, client sends the moving text along with the result to the server and this would be stored on database. Table 11 shows a list of game result constants.

Constant	**Game Result**
"1-0"	White win
"0-1"	Black win
"1/2-1/2"	Tie

Table 11: List of game result constants

4.3 Push and Polling

With the advent of technology, modes of communication have vastly expanded. There are even more advanced communicative measures between mobile devices and servers.

Polling is one of the techniques which periodically poll a server to obtain data from it [11]. In polling, mobile device has to send request to the server for every certain

period of time to check whether data is modified or not. If the data is modified, it gets the modified content from server.

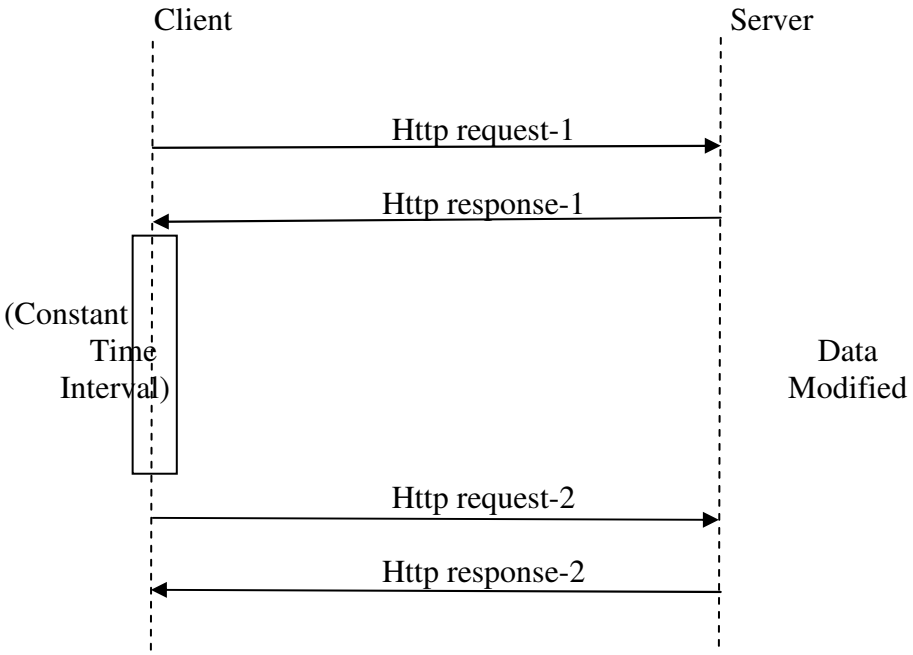

Figure 24: Overview of polling technique

As an example for polling technique, a chat based application will poll the server for every 10seconds to see whether if new chat messages are available. It means the browser will open a connection to the server every time data is required.

Even it is easy to implement, it has numerous disadvantages mentioned below if there are too many devices associated with server.

- High bandwidth is required.
- It consumes more battery power when mobile device repeatedly polls the host for data.

- There is a high probability to get conflicts when more numbers of devices continuously polls on server as a result, server gets more burdens.
- The client gets empty signal frequently from the server until new modifications on host.
- It consumes all resources for every time checking with the server for modifications.
- It is a scalability communication, that is, the number of requests made to the server can be extremely high if the frequency of polling is set to a small value.

Most of the applications need data collection from the server through internet. There are several methods to communicate to the host from mobile device [14]. Push is one of the technologies which push data from server to device with less bandwidth.

Push is a light-weighted mechanism to push the data from host to client using HTTP protocol. It uses less bandwidth for data transmission. The host routes the data to the device. If the device is in offline, data will be delivered later when the device turns to online. This communication is highly optimized to minimize the battery consumption. The GCM service handles all aspects of queuing of messages and delivery to the target Android application running on the target device.

We proposed the implementation of push technology with three parties: the application server, Google cloud to device messaging and android application [16]. The application server sends data to an android application via HTTP Google cloud to device messaging. Every mobile device is identified with a unique identification called “device

id". Push service always maintains a connection with Google server. The connection is highly optimized to minimize bandwidth and battery consumption.

Figure 25: Implementation of push communication

The implementation of push technology is done in five steps

— In the initial phase, mobile has to register with GCM server as a result, client will get activation code. This registration is done by sending request to the GCM server from mobile application. We use "com.google.android.c2dm.intent.REGISTER" service for requesting GCM server from android application.

— When mobile makes request to GCM server, it sends activation code as response. This activation code is unique for each mobile device which is used to get messages from GCM server.

— In third phase, mobile application sends application code which received from GCM server and device id to third party servers. To implement this we can use any technology which sends these two parameters to third party servers.

— The objective of third party server is that to send message to mobile application. This is happened via GCM server. Before the third party server using GCM server, it has to register with GCM server for each mobile client. When third party server request GCM server for registration, it would get an id called “registration id” for each mobile client. This registration id is used to send messages from third party server to mobile applications via GCM server.

  Implementation of this is done to send POST type of request to the link “https://android.googleapis.com/gcm/send“with the following parameter values:

  1. Authorization: key=YOUR_API_KEY
  2. Content-Type: application/json for JSON; application/x-www-form urlencoded;charset=UTF-8 for plain text
  3. Registration_id = YOUR_REGISTRATION_ID
  4. Data = Message

— This is the last phase in which message transfers to the mobile automatically. We need not to implement anything in this phase. Everything got implemented in

GCM server. When notifies as getting message from third party server, it immediately sends that message to mobile client.

4.4 User Interface:

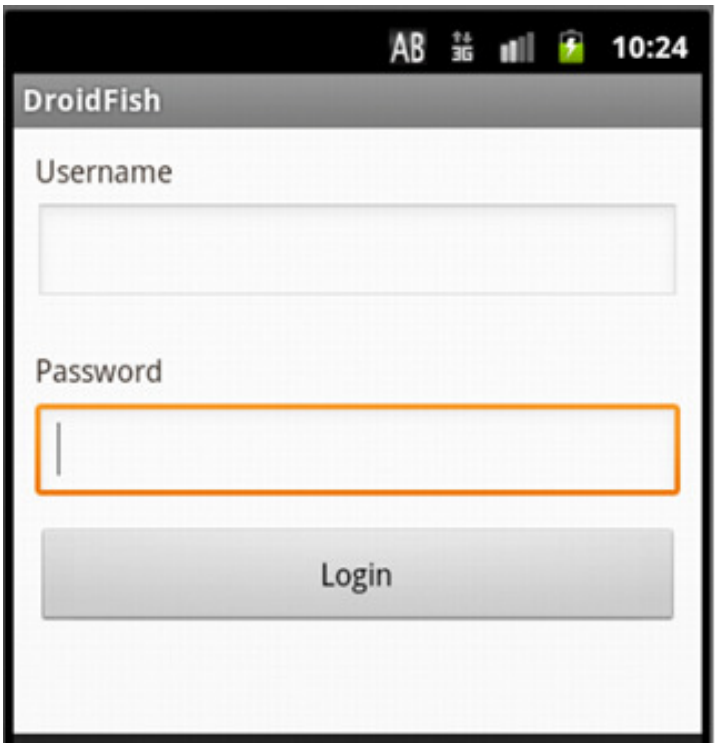

Figure 26: Login interface

The player enters username and password in the above screenshot. It navigates to the game.

Figure 27: User playing in offline mode

Figure 28: Users playing live in online

CHAPTER 5: CONCLUSIONS

With the recent advancement in the area of game applications, chess applications can definitely be considered as one of the most important game applications. It lacks predicting of winning and losing percentage based off the chess moves. Our study and results show that how to determine chess game statistics, such as winning and losing probability based on the moves.

We have presented new succinct representation for tries that guarantee low average height and enable the compression of the labels. Our experimental analysis has shown that they obtain the best space use when compared to the other data structures and trie are very efficient for managing large sets of strings in memory. In large-scale experiments with a variety of real data sets, we have shown that tries are more compact than binary trees or splay trees and are over two times faster and are close to hash tables in efficiency, yet keep the data in sorted order. The performance of the trie depends on the distribution of strings. The worst case we observed was that the strings are of equal length and have a relatively flat probability distribution. Even in this case, performance was comparable to the other tree structures. Analytical results have shown that the height of a trie is expected to be logarithmic. We therefore believe that the trie is the data structure of choice for practical string management.

BIBLIOGRAPHY

[1] de la Briandais, R. (1959). "File Searching Using Variable Length Keys". Proceedings of the Western Joint Computer Conference: 295–298.

[2] Thomas H. Cormen [et al.] (2009). Introduction to Algorithms (3rd ed.). Massachusetts Institute of Technology. pp. 253–280.

[3] Milo R, Shen-Orr SS, Itzkovitz S, Kashtan N, Chklovskii D, Alon U (2002). "Network motifs: simple building blocks of complex networks". Science **298** (5594): 824–827.

[4] Schreiber F, Schwobbermeyer H (2005). "MAVisto: a tool for the exploration of network motifs".Bioinformatics **21** (17): 3572–3574.

[5] M. C. Costanzo, M. E. Crawford, J. E. Hirschman, J. E. Kranz, P. Olsen, L. S. Robertson, M. S. Skrzypek, B. R. Braun, K. L. Hopkins, P. Kondu, C. Lengieza, J. E. Lew-Smith, M. Tillberg, and J. I. Garrels. Ypd(tm), pombepd(tm), and wormpd(tm): model organism volumes of the bioknowledge(tm) library, an integrated resource for protein information. Nucleic Acids Res., 29:75–79, 2001.

[6] Computer Networks and Internets (5th ed.). 2008. pp. 368. ISBN 978-0-13-606698-9.

[7] Gilberg, R.; Forouzan, B. (2001), "8", Data Structures: A Pseudocode Approach With C++, Pacific Grove, CA: Brooks/Cole, p. 339, ISBN 0-534-95216-X.

[8] Adelson-Velskii, G.; E. M. Landis (1962). "An algorithm for the organization of information". Proceedings of the USSR Academy of Sciences **146**: 263–266.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7 . Chapter 13: Red–Black Trees, pp. 273–301.

[10] Lucas, Joan M. (1991), "On the Competitiveness of Splay Trees: Relations to the Union-Find Problem", Online Algorithms, Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) Series in Discrete Mathematics and Theoretical Computer Science Vol. 7: 95–124.

[11] Bumsuk Lee, "A Temporal Analysis of Posting Behavior in Social Media Streams," In Proc. of the AAAI ICWSM 2012.

[12] Knuth, Donald (1997). "6.3: Digital Searching". The Art of Computer Programming Volume 3: Sorting and Searching (2nd ed.). Addison-Wesley. p. 492. ISBN 0-201-89685-0.

[13] Scribner, Kenn; Seely, Scott (2009), Effective REST Services via .NET, Boston: Addison-Wesley, ISBN 978-0-321-61325-7.

[14] Server-Push Documents (HTML & XHTML: The Definitive Guide) O'Reilly book explaining server-push.

[15] Donald Knuth (1998). 'The Art of Computer Programming'. 3: Sorting and Searching (2nd ed.). Addison-Wesley. pp. 513–558. ISBN 0-201-89685-0.

[16] "Android Cloud to Device Messaging Framework". Google Inc.

https://developers.google.com/android/c2dm/

[17] Bumsuk Lee, "A Temporal Analysis of Posting Behavior in Social Media Streams," In Proc. of the AAAI ICWSM 2012

[18] Sia, K. C., Cho, J., and Cho, H. K., "Efficient Monitoring Algorithm for Fast News Alerts, 2007" IEEE TKDE, Vol. 19, Issue 7, pp. 950-961

[19] Franklin Mark Liang (1983). *Word Hy-phen-a-tion By Com-put-er* (Doctor of Philosophy thesis). Stanford University. Archived from the original on 2010-05-19. Retrieved 2010-03-28