The Nearly Optimal N-ary Search Tree

A Thesis Presented
by
Dominic J. Catalano
to
The Department of Electrical and Computer Engineering
in partial fulfillment of the requirements
for the degree of
Master of Science
in
Electrical and Computer Engineering

Northeastern University
Boston, Massachusetts
November 2016
To those who listened to me mutter to myself: friends, family, coffee.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vii</td>
</tr>
<tr>
<td>List of Acronyms</td>
<td>viii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>ix</td>
</tr>
<tr>
<td>Abstract of the Thesis</td>
<td>x</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Contributions</td>
<td>2</td>
</tr>
<tr>
<td>2 Motivation</td>
<td>4</td>
</tr>
<tr>
<td>3 Related Work</td>
<td>7</td>
</tr>
<tr>
<td>3.1 Hash-Based Indexing</td>
<td>7</td>
</tr>
<tr>
<td>3.2 Tree-Based Indexing</td>
<td>7</td>
</tr>
<tr>
<td>3.3 Nearly Optimal Binary Tree</td>
<td>9</td>
</tr>
<tr>
<td>3.4 Query Optimization</td>
<td>13</td>
</tr>
<tr>
<td>4 Nearly Optimal N-ary Search Tree</td>
<td>14</td>
</tr>
<tr>
<td>4.1 Problem Formulation</td>
<td>14</td>
</tr>
<tr>
<td>4.2 Algorithm Design</td>
<td>16</td>
</tr>
<tr>
<td>4.2.1 Initial Approach</td>
<td>16</td>
</tr>
<tr>
<td>4.2.2 Revisions</td>
<td>17</td>
</tr>
<tr>
<td>4.2.3 Zero Set Frequency Detection</td>
<td>17</td>
</tr>
<tr>
<td>4.2.4 NP Problem</td>
<td>20</td>
</tr>
<tr>
<td>4.2.5 Enhancements</td>
<td>24</td>
</tr>
<tr>
<td>4.2.6 Further Enhancements</td>
<td>26</td>
</tr>
<tr>
<td>5 Evaluation</td>
<td>40</td>
</tr>
</tbody>
</table>
List of Figures

2.1 The population of the United States by % population sorted both by city name and by % population .......................... 5

3.1 A binary tree with an expected search cost of 2.44 .................................................. 8
3.2 A traditionally created B+ tree ................................................................. 9
3.3 A tree constructed using Mehlhorn’s first rule with an expected search cost of 2.46 11
3.4 A malformed tree constructed using Mehlhorn’s first rule with an expected search cost of 3.45 11
3.5 A Nearly Optimal Binary Tree constructed using Mehlhorn’s second rule with an expected search cost of 1.72 12
3.6 A Nearly Optimal Binary Tree constructed using Mehlhorn’s second rule with an expected search cost of 2.01 12
3.7 A Nearly Optimal N-ary Search Tree .................................................. 13

4.1 Sensitivity Analysis of One-directional Algorithm with N = 8 . ........................................ 21
4.2 Sensitivity Analysis of One-directional Algorithm with N = 12 . ........................................ 21
4.3 Sensitivity Analysis of One-directional Algorithm with N = 16 . ........................................ 22
4.4 Sensitivity Analysis of One-directional Algorithm with N = 20 . ........................................ 22
4.5 Sensitivity Analysis of One-directional Algorithm with N = 24 . ........................................ 23
4.6 A process showing how iterators would move and partition the set. Triangles represent the iterators, squares represent nodes with keys and frequencies, and horizontal lines represent partitions in the current set. The tree produced is reproduced below the process ......................... 25
4.7 Sensitivity Analysis of Bi-directional Algorithm with N = 8 ........................................ 27
4.8 Sensitivity Analysis of Bi-directional Algorithm with N = 12 ........................................ 27
4.9 Sensitivity Analysis of Bi-directional Algorithm with N = 16 ........................................ 28
4.10 Sensitivity Analysis of Bi-directional Algorithm with N = 20 ......................................... 28
4.11 Sensitivity Analysis of Bi-directional Algorithm with N = 24 ......................................... 29
4.12 A comparison between the structure of a B+ tree and the nearly optimal tree ............... 30
4.13 Sensitivity Analysis of Enhanced Bi-directional Algorithm with N = 8 ......................... 35
4.14 Sensitivity Analysis of Enhanced Bi-directional Algorithm with N = 12 ......................... 35
4.15 Sensitivity Analysis of Enhanced Bi-directional Algorithm with N = 16 ......................... 36
4.16 Sensitivity Analysis of Enhanced Bi-directional Algorithm with N = 20 ......................... 36
4.17 Sensitivity Analysis of Enhanced Bi-directional Algorithm with N = 24  

4.18 Side by Side Actual Expected Search Costs with N = 8  

4.19 Side by Side Actual Expected Search Costs with N = 12  

4.20 Side by Side Actual Expected Search Costs with N = 16  

4.21 Side by Side Actual Expected Search Costs with N = 20  

4.22 Side by Side Actual Expected Search Costs with N = 24  

5.1 Fill Ratio by N for a set size of 10,000  

5.2 Fill Ratio by N for a set size of 100,000  

5.3 Fill Ratio by N for a set size of 1,000,000  

5.4 Fill Ratio by N for a set size of 10,000,000  

5.5 Fill Ratio by N for a set size of 100,000,000  

5.6 Fill Ratio by N for a set size of 10,000 compared to approximate and actual expected values  

5.7 Performance Ratio vs Fill Ratio  

5.8 Performance Ratio vs Fill Ratio where the traditional expected search cost = 3  

5.9 Performance Ratio vs Fill Ratio where the traditional expected search cost = 4  

6.1 Histogram of ICD10 data where N=6  

6.2 Histogram of ICD10 data where N=99  

6.3 Cumulative histogram of ICD10 data where N=6  

6.4 Cumulative histogram of ICD10 data where N=99  

6.5 Histogram of Census data where N=5  

6.6 Cumulative histogram of Census data where N=5
List of Tables

2.1  Traditional placement vs placement prediction for Census data.  . . . . . . . . . . . . . . 6

6.1  Node placement prediction vs actual node placement for Census data.  . . . . . . . 52
List of Acronyms

**NONST** Nearly Optimal N-ary Search Tree. A data structure for searching that is arranged such that the expected search cost of the tree is minimized.
Acknowledgments

I have utmost gratitude for Professor Ningfang Mi for first convincing me to pursue a thesis. Her guidance and encouragement has been paramount during the entire process leading me up to this point.

Professor Kathleen Durant has proven to be an invaluable resource. In addition to introducing me to the primary structures and concepts involved with this Thesis, she has exemplified herself as an excellent mentor and professor.
Abstract of the Thesis

The Nearly Optimal N-ary Search Tree

by

Dominic J. Catalano

Master of Science in Electrical and Computer Engineering

Northeastern University, November 2016

Dr. Ningfang Mi, Adviser

As the scale and importance of data increases, so does the task of information retrieval. Currently, the best methodology for index based structures offers a near constant search cost by ensuring all data exists within the same number of IO operations. However research has shown that data, especially data representing human interactions, exhibit high variance in terms of access frequency distribution. Therefore, by altering traditional methods whereby more frequently accessed data can be retrieved in fewer IO operations, the nature of the data can be exploited to garner performance gains. By redefining the constraints of a B+ tree, the Nearly Optimal N-ary Search Tree utilizes this variance to reduce the expected search cost of searching in the tree. By placing high frequency nodes on higher levels of the tree, the expected cost becomes reduced. Optimizing the substructure of a tree consequently improves the efficiency of operations dependent on key based search. Case studies of datasets exhibiting power law distributions are shown to benefit from a NONST search structure, achieving search reduction of near 40%. The foundation of NONST is the first step in identifying a generic structure that is applicable and extensible for a number of applications. By utilizing the underlying theory, datasets and databases can be manipulated to achieve better performance based on domain knowledge of the data itself.
Chapter 1

Introduction

The amount of data in the world is experiencing a meteoric growth both in scale and importance. A study supported by EMC reports that the digital universe will expand to 35 petabytes by 2020, a staggering 44 times that of 2009 [1]. Meanwhile, *The Economist* places data almost on the same level of importance to industry as labor and capital [2]. Fortunately, the cost of storing data has also decreased to accommodate this trend [3]. Increases in data intuitively results in an increased need to access it. While big data systems like Hadoop provide fast, cheap, and efficient searching of large-scale indexed data, continued reductions in computing costs allow for a more intelligent approach to data retrieval [4]. By applying a smarter approach to data storage based on known information, overall search cost can be reduced. Faster retrieval means greater performance resulting in better systems.

Current methods of search, particularly using database indexes, rely on classic data structures that pride themselves on constant or near constant search cost. Hashing tables allow data to be divided into buckets. While access to a bucket is fast in terms of I/O operations, buckets themselves become prone to overflow as data is added to a bucket. Hash tables also are not convenient structures to apply range searches across a span of keys. Alternatively, B+ trees use a tree based structure to guarantee a search cost that is logarithmic to the size of the dataset. B+ trees extend a binary tree by making non-leaf nodes into pages of a predetermined number of pointers that can refer to other pages or nodes. B+ trees offer benefits like range searching, insertion, and deletion. While very useful, the B+ tree works by considering all data as equally important.

By utilizing known features of the dataset, particularly the frequency of key access, data structures can be optimized such that key based searches will occur in a minimum amount of time. This thesis introduces the Nearly Optimal N-ary Search Tree (NONST), a data structure based on a
CHAPTER 1. INTRODUCTION

B tree that is designed to reduce overhead costs when accessing the data. To construct a NONST, subsets of the dataset need to be partitioned such that every subset contains the same total frequency. The algorithm is then run recursively until all nodes in the dataset are referenced by a page. We further developed a set of novel algorithms to assign the substructure of the tree based on the frequency of the data. A gambit of tests were used to assess how overall access cost is affected given variabilities in the data’s frequency distribution. Iterations were developed until NONST out performed the B+ tree in all cases, even when data variability was very low. Using the final algorithm, case studies using real datasets demonstrated the effectiveness of using a NONST to structure data since expected search cost was reduced.

We conclude that the reduction of access cost is achieved by placing a higher priority on more frequency accessed or important data in the search tree. Current search methodologies give equal weight to all information in a dataset such that each entry can be retrieved in a constant time. Instead of this communistic approach, the priority of a node can be capitalized such that information of higher priority is retrieved in less time. By exploiting this known information, data structures can be optimized to give higher weight to more influential entities. This line of thoughts lends itself into a discussion of morality when considering relative importance. For example, the concept of Net Neutrality is based on prioritizing Internet speeds for better paying companies or users. While the future of Internet Neutrality is questionable, the desire to give better performance to more profitable data consumers is not [5].

1.1 Contributions

This thesis will discuss the challenges facing traditional key based search and ways to optimize it. The following contributions will be made:

1. Introduction of a new B-tree like data structure, the Nearly Optimal N-ary Search Tree (NONST)
2. Theories suggesting how to structure data inside a NONST
3. Algorithms that will build a NONST
4. Evidence suggesting highly distributed data achieves significant gain when structured in a NONST over a B+ tree using the same data
CHAPTER 1. INTRODUCTION

The remainder of this thesis is organized as follows. Chapter 2 will discuss the importance of improving expected search time. Then, Chapter 3 will lay the foundation for the proposed algorithm in terms of methods currently available and theoretical assumptions. The general theory behind NONST and the algorithm design process will then be explained while showing reasons for the decisions made in Chapter 4. Further analysis on the performance gain will be conducted in Chapter 5. Case studies in Chapter 6 will then show how search cost was reduced for real datasets when compared to the traditional methods. Lastly, Chapters 7 and 8 will lay the groundwork for further enhancements that could be made to the algorithm and subsequent data structure.
Chapter 2

Motivation

Searching problems involve retrieving data from a data structure that matches a specified search key if it exists. There are many different approaches to searching problems ranging from straightforward algorithms like sequential search to more complex and efficient methods like trees. Each approach has pros and cons dependent on the needs of search. When it comes to large-scale, real-world data, it becomes necessary to organize datasets efficiently to support real-world applications[6].

When it comes to storing data, conventional methods generally defer to hash tables or tree structures to implement storage mechanisms due to their scalability, ease of implementation, and understandability [7]. However, these methods do not consider any domain knowledge of the data itself. Work in the artificial intelligence field demonstrates that informed methods of search excel when appropriate heuristics are assigned [8].

Data generated by real-world experiences and processes exhibit distributions in access frequency, especially when considering application or aggregated data. Consider a word counting application, it can be assumed that the words found in a particular text belong to some subset of the English (for the purposes of this example) language. By nature, some words in the dictionary will occur more often than others. So, would it not be intuitive to store these entries or nodes, as will be referred to, such that there is less overhead to retrieve the node from the data structure? Studies have further shown that human driven data exhibits power law distributions. For example, Pareto finds the wealth distribution to follow such a pattern [9]. A simple example of this can be seen in Figure 2.1 Considering the city and state as the composite key of the dataset, the population distribution exhibits a pattern such that a few cities contain the largest quantities of people. The second graph makes this point even clearer by sorting based on the percent of population contained within the city.
CHAPTER 2. MOTIVATION

Therefore it becomes necessary to not only accept that data contains an inherent frequency distribution, but also to exploit and utilize this fact in order to reduce performance overheads in a system when accessing the data structure. In this thesis, we introduce the Nearly Optimal N-ary Search Tree (NONST), a data structure that is designed to reduce overhead costs when accessing the data contained within it. By using the frequencies given by the nodes, a NONST is able to structure itself using the principles of a B+ tree while structuring nodes in a nearly optimal fashion. Meaning, the configuration of the nodes in the tree will lead to an average search cost that is near the minimum possible for that set of nodes. Table 2.1 shows the difference between the traditional approach and the approaches discussed in Chapter 4. While the traditional B+ tree would place all nodes on one level, a NONST takes into consideration the frequency of each node. Thus, nodes with higher frequency are targeted to be placed on higher levels. Conversely, nodes with lesser frequency are permitted to exist below the threshold set for a B+ tree.

This approach looks to optimize the overall performance of the system by reducing the number of IOs necessary to retrieve information. By taking a realistic approach to how data is utilized and accessed, the number of reads required to maintain a system can be reduced. The goal is to create a generic structure that is extensible and applicable to other applications. The proposed algorithm and structure offer a first step in a general structure that allows functions such as insert and delete.
Due to the flexible nature of the structure, a NONST can be used in a variety of ways to enhance the performance of system.

In modern day applications where data access is recurring and predictable, a structure such that the average access time is reduced would be highly beneficial to performance. Indeed research has shown, that when structured accordingly, overall performance increases with respect to average search time. In power law distributions like population density, performance can increase by 40%.

<table>
<thead>
<tr>
<th>Level</th>
<th>Traditional Placement</th>
<th>Theoretical Placement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>49</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>299</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1628</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>7571</td>
</tr>
<tr>
<td>8</td>
<td>81611</td>
<td>15946</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>25572</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>20792</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>7495</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>1300</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>282</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>640</td>
</tr>
</tbody>
</table>

Table 2.1: Traditional placement vs placement prediction for Census data.
Chapter 3

Related Work

In order to understand the proposed data structure and algorithms, first we must look at existing algorithms meant to handle large scale databases as well as lay the foundation for its optimization technique.

3.1 Hash-Based Indexing

Hashing methods first originated in an internal IBM memorandum by H.P. Luhn and provides a structure for the creation of a dictionary of keys and objects \[10\]. It works by using a hash function on the key to assign it to a bucket. For databases, the hash function generally converts the key to binary and uses a number of least significant bits to determine the appropriate bucket where they are stored as pages.

While the appropriate bucket can be usually returned within 2 IOs, Hash-based Indexing does have some drawbacks. If a page becomes full, overflow pages are created, making the bucket more costly to examine. More importantly, Hash tables do not support range searches across a range of keys. In fact, hash indexes only support equality operations as well as less than or greater than functions only if the result is not a range \[11\]. For this reason, many commercial systems prefer tree-based indexing\[12\].

3.2 Tree-Based Indexing

Search trees provide a robust and simple structure for organizing and finding data. The simplest application of such search trees are the binary search tree (BST). As a refresher, a binary
A tree where every subtree of a node has keys less than any other subtree of the node to its right. The keys in a node are conceptually between subtrees and are greater than any keys in subtrees to its left and less than any keys in subtrees to its right[13].

In order to build a balanced binary search tree with a guaranteed search time, nodes must be placed such that the pivotal node in any subtree is the middle node in the set considered, as in Figure 3.1. Searching such a structure is simple because of the key pattern. Starting at the root, traverse the tree left or right based on whether or not the searched key is less than or greater than the current node. For example, Figure 3.1 contains nodes with alphabetical keys and arbitrary frequencies assigned to them. Using these values, we can assume that the expected cost of retrieval would be 2.44.

The B tree expands upon the binary variant of the tree. The standard B-tree operates much like a binary tree except non-leaf nodes contain the information for nodes as well as paths to its $N$ children. However, this implementation is limiting since page space is reserved for the nodes. A B+ tree solves this problem by not storing node information on non-leaf objects. Instead, the B+ tree uses pages of pointers and key values to direct search. An illustration of this can be seen in Figure 3.2, where the root page refers to four child pages by identifying 3 key values. The child pages then refer to up to four child nodes. The pages contain the pointer information as well as key values in order to direct search.

B+ structures are very useful for indexing large datasets because they offer a guaranteed retrieval time equivalent to the height of the tree. Traditional B+ tree implementation dictates that all
data exist on the same level of the tree. This causes all pages to have child references to nodes if and only if it is the level directly above the data level, otherwise child references will point to sub-pages. This permits a guarantee that all information is retrieved in the same amount time-agnostic of the node’s frequency.

Since many node-pointers can fit onto a page, the height of the tree drastically reduces compared to the binary approach. Given a set of 100,000,000 entries and $N = 100$, the binary tree would guarantee an upper bound on search of 27 iterations while the B+ tree would ensure a cost of 4. Traversing the tree would occur as follows. First, access the root page (which is often stored in memory) and find the highest node pointer such that the query key is less than the stored key value. Then, retrieve the page in question from disk. Repeat until the node itself is returned from disk. Reusing our data points from Figure 3.1, we can see Figure 3.2 performs at a constant cost of 2.

### 3.3 Nearly Optimal Binary Tree

Most data generated in the real-world does not have uniform frequency distribution. Some data will intrinsically be accessed more frequently than others, especially considering datasets used in real world applications. The world of statistics gives us a number of these distributions including the Poisson, Normal, and- importantly for later testing-the Geometric Distribution [15]. Much work has been done identifying patterns on identifying access rates when it comes to large socially driven applications. Studies have shown that probability distributions of an individual to social networks follow a scale-free power-law [16]. This implies that a small number of entities account for the majority of actions. This pattern can be seen in populations of cities, intensities of earthquakes, power outages, and illness [17] [18] [19].

Donald Knuth proposed a solution to account for this variability in data frequency [20].
CHAPTER 3. RELATED WORK

Given that every node in the dataset had a known frequency, there exists a layout of the Binary Search Tree (BST) with a minimal weighted path length called an optimum binary search tree. Knuth’s solution uses dynamic programming to construct the tree such that the expected search cost is minimized. The expected search cost can be computed by $\sum f_i \cdot L_i$ where $f_i$ is the known frequency of the node and $L_i$ is the level of the tree where the node exists. However this method takes $O(n^2)$ time and space complexity, making it prohibitive for most applications. Kurt Mehlhorn further examined the issue by analyzing two techniques to create the Nearly Optimal Binary Search Tree\textsuperscript{[21]}.

The first rule examined was to take the most frequent node (as opposed to the median) as the root of the tree and then apply the method recursively in the subsequent sets. However this rule was shown to create poorly constructed trees. Figure\textsuperscript{[3.3]} shows an instance of this. The tree looks very off balanced. While it does exhibit a reduction in expected search cost (2.44 for the regular binary tree vs 2.31 for this rule), counter examples are easy to suggest. If nodes exhibit a linearly decreasing frequency, this method will construct a tree like a list as in Figure\textsuperscript{[3.4]}. Because there is no greater intuition than picking the highest frequency, the root may still have a very small frequency relative to the set size. The expected search cost for this tree is much higher than the binary tree with a cost of 3.45.

The second rule, instead, chooses a central node such that the sum of the frequency on either side of the root was equal or comparable. This rule differentiates itself from rule 1 by considering the entire set as opposed to just the node with the highest frequency. By dividing the set into subsets of equal weight, the tree becomes more balanced and thus more efficient. For the first arbitrary set in Figure\textsuperscript{[3.6]} we see an expected search cost of 2.01, which is less than the binary approach as well as rule 1. For the malformed case in rule one in Figure\textsuperscript{[3.5]} we see a much better expected cost at 1.72. Mehlhorn was able to prove that trees constructed under this paradigm were bounded by the search cost of the balanced BST in a time complexity of $O(n)$.

While this capability to create a Nearly Optimal Binary Search Tree is a very interesting exercise in data structuring, binary trees themselves are not particularly useful when considering large data storage mechanisms. This thesis proposes a new B-tree structure, the Nearly Optimal N-ary Search Tree, which leverages Mehlhorn’s Nearly Optimal Binary Search Tree proofs. The structure proposed will construct a tree based on the B+ approach of pages and leaf nodes.

The structure differentiates itself by disavowing constraints in terms of guaranteed performance in favor of a more flexible structure. By implementing Mehlhorn’s approximation for finding suitable partitions to identify subsets of near equal frequency, the Nearly Optimal N-ary Search Tree
CHAPTER 3. RELATED WORK

Figure 3.3: A tree constructed using Mehlhorn’s first rule with an expected search cost of 2.46

Figure 3.4: A malformed tree constructed using Mehlhorn’s first rule with an expected search cost of 3.45
CHAPTER 3. RELATED WORK

Figure 3.5: A Nearly Optimal Binary Tree constructed using Mehlhorn’s second rule with an expected search cost of 1.72.

Figure 3.6: A Nearly Optimal Binary Tree constructed using Mehlhorn’s second rule with an expected search cost of 2.01.
should produce a data structure that minimizes expected search cost. By taking the utility of the B+ tree and merging it with the theoretical optimization properties of the Nearly Optimal Binary Search Tree, the Nearly Optimal N-ary Search Tree hopes to ensure performance gains in terms of search cost for a variety of datasets.

3.4 Query Optimization

Minimizing search costs based on indexes produces intrinsic performance gains on processes reliant on key based search for its core functionality. In a SQL database, good queries take advantage of the relational structure naive to SQL keys. By evaluating a query plan, a database management system can optimize execution by prioritizing a key comparison operation. Using a key allows the query to reduce the resulting set of the query earlier in the execution. This optimizes the query by reducing the set size of a more intensive operation. Utilization of the key yields fewer I/O operations during the execution of a query [14]. By reducing the cost to traverse and aggregate results in the tree, queries can be completed at a lower cost, improving the efficiency of the database management system leading to an overall improved user experience.
Chapter 4

Nearly Optimal N-ary Search Tree

Given the theoretical description of NONST, we developed various algorithmic iterations until a version guaranteed a lower expected search cost than the de facto standard B+ tree. In this section we describe the different versions of the algorithms as well as its parameters. We define performance metrics while exploring worst case conditions and other interesting scenarios.

4.1 Problem Formulation

In order to determine whether the proposed data structure will indeed achieve performance gain over the traditional B+ structure, metrics and variables need to be defined and assumptions made. Given:

1. \( n \) is the number of entries in the dataset
2. \( f_i \) is the normalized frequency of item \( i \) in the set \( n \)
3. \( F = \sum f_i = 1 \) where \( F \) is the cumulative frequency of the set
4. The entries of \( n \) are sorted

Because this thesis specifically looks to the B+ Tree, we can define additional metrics pertaining to the behavior of a multi-path search tree.

1. \( N \) represents the number of pointers or paths available at a given page or the maximum fanout at a given page
2. $l_i$ represents the depth or level of an entry in structured data or the cost to retrieve it from the set.

3. $E(\text{approx})$ represents the approximate search cost of the dataset constructed via the data structure proposed.

4. $E(\text{actual})$ represents the actual search cost defined by $\sum f_i \times l_i$.

5. $E(\text{traditional})$ represents the guaranteed search cost of the traditional approach.

For the B+ tree, its height is defined by the path from root to leaf. In practice, every page does not have the same number of children, but $\lceil \log_{N} n \rceil$ gives us a minimum height for a given tree containing $n$ entries with a fanout of $N$. This is intuitive because the B+ tree dictates that all entries must exist on the same level. For example, if a tree contains $N + 1$ entries, it must allocate a second level of index pointer pages. While the tree would not fully utilize these new pages, the height will nevertheless increase. This occurs again at $N^2 + 1$, $N^3 + 1$, and so on. Thus, we can assume that the B+ tree has a search cost equivalent to its height.

The knowledge of the height of the traditional B+ tree allows us to define a performance goal. In order to be more efficient than the traditional method, the proposed algorithm must produce a tree with an expected value less than the height of a B+ tree given the same fanout $N$ and set size $n$. Formally, we can call our algorithm successful if the height of the B+ tree is greater than the expected value of the structured set.

In the proposed data structure, the actual performance metrics become more complicated to predict. Because $l_i$ for any given node or entry is dependent upon the implementation of the theory, we must accept an approximation of this value in order to most accurately predict the performance of the method.

Upon careful examination of the goal of the structure, predicting the level of a given node actually becomes quite easy. Consider a set $n$ of 100 entries and an $N$ value of 10. This means at the root page, there are ten possible subtrees or entries attached to it. If an entry has a $f_i$ value of greater than 0.1, then that entry should be directly referenced by the root page. This makes sense because the goal is to provision the subtrees such that the frequency of the aggregate entries are distributed equally among all possible partitions. If an entry surpasses this ideal distribution, then by definition, it should exist by itself in that partition. This idea can be applied to every level of the tree. We can predict the ideal partition frequency by taking the total frequency of the set considered and dividing by $N$. In the example, the root page should be divided into ten sets of 0.1. Stepping into one of those
sets, if the cumulative frequency is indeed 0.1, then we should then create ten more partitions of 0.01. We can therefore predict the \( l_i \) by searching for the level whereby its frequency exceeds the target division frequency. Indeed, this can be summed up by the function \([\log_N \frac{1}{f_i}]\). Therefore, we can safely predict \( E(\text{approx}) = \sum_{i=1}^{n} ([\log_N \frac{1}{f_i}] \cdot f_i) \).

Because we are removing the constraint of guaranteed height, we must consider the worst case scenario for structuring the tree. For given \( N \) and \( n \) values, we can assume that the worst case should be that of the deepest tree. However further investigation proves otherwise. If \( N = 2 \) and \( n = 100,000,000 \), a height of \( n \) is very concerning. However, in order for the height to be \( n \), every page must contain a node and a pointer to another page. Meaning, at level 1, there would be a node and a page. For that node to be placed there, it must contain an access frequency of at least 0.5. Similarly, on level two the node must contain at least 0.25. Indeed, we can calculate the performance of any set of any size following this distribution to be \( \sum_{i=1}^{\infty} \frac{(N-1)}{N^i} \). As \( N \) grows, this expected value becomes lower. Thus, the “worst case” assumption becomes invalid.

For the purposes of testing, we will define one more parameter, \( Z \). A random number generator library (rngs and rvgs) with a set seed will be used to generate a geometric distribution of high variance. This generator was chosen because of its high variance, consistency, and usability. It was designed by Steve Park and Dave Geyer in conjunction with a paper on random number generation [22]. To allow for a finer control of the set variability, we will use \( Z \) to manually adjust the frequency distribution of the set by generating node frequency with the function \( f_i = \text{Geometric}()^Z \). For values of \( Z < 1 \), the distribution will be lessened while \( Z > 1 \) will allow for more variability in the dataset.

### 4.2 Algorithm Design

The following designs are intended to show proof of concept for the data structure proposed. The final design may not be the best possible solution for the implementation. Charts are rendered through a combination of Google Charts and CanvasJS charts and can be viewed interactively at the thesis’s website [23][24][25].

#### 4.2.1 Initial Approach

With base parameters defined, implementation specific variables can be identified. For simplicity sake, let us assume each node the knowledge of the prior cumulative frequency of the set...
such that node \( i \) will know the sum of the frequency from 0 to \( i - 1 \). This allows us a simple method to return the prior cumulative frequency (\( pf_i \)), the cumulative frequency including the node (\( cf_i \)), and the frequency of the node itself (\( f_i \)). This assumption allows easy calculation of the cumulative frequency of any given set, given a subset from \( a \) to \( b \). The set frequency \( f_{ab} \) can be calculated as \( f_{ab} = cf_b - pf_a \). From here, the ideal partition frequency can be determined as \( f_{\text{target}} = \frac{f_{ab}}{N} \).

The first algorithm that was developed used this target frequency to linearly create partitions from \( a \) to \( b \) with iterator \( i \). In order to designate multiple partitions, the left most node not having been assigned yet is designated as \( x \). Therefore once \( cf_i - pf_x \geq f_{\text{target}} \) partition \( x \) to \( i \), set \( x = i + 1 \), and continue creating partitions. If there is only one partition left, designate out the remaining nodes as the last partition. In the event that the set contains fewer nodes than \( N \), then this page will not need to create any child pages. Therefore, assign each node to an available pointer on the page.

A helper function, \( \text{AssignSubstructure} \), is used to contain the process of defining children on a page. It accepts the parameters \( a \) and \( b \) as the start and end index of the set in question as well as the partition designation for the current page. The partition designation refers to the index of the page’s \( N \) possible paths. If the helper is passed a set containing only one node, it directs the page to point to that node directly. If a set is passed, it produces a new tree, assigns the page to point to that tree, and calls OptimallyStructure on that set for that page. This algorithm can be seen in Algorithm 1.

Testing this algorithm became infeasible due to uncaught errors. Therefore, large scale testing was not successful.

### 4.2.2 Revisions

While this algorithm produced trees, it lacked the performance gains that the theory suggested, usually offering a higher \( E(\text{actual}) \) value than \( E(\text{traditional}) \). Thus a number of enhancements were made.

### 4.2.3 Zero Set Frequency Detection

Firstly, in the event that the set considered did not have associated frequency, the algorithm would fail due to divide by zero errors or partitions not being assigned (as the threshold would never be hit). Imagine a set of nodes whereby the first and last node had a frequency of 0.5, and all nodes in between had 0. If \( N = 3 \), the first iteration of the algorithm would identify that the root page should
Algorithm 1 Initial Linear Algorithm

1: procedure OPTIMALLYSTRUCTURE($a$, $b$)
2:     $f_{ab} \leftarrow c_f b - p_f a$
3:     if $b - a < N$ then
4:         for $i = 0; i <= b - a; i++$ do
5:             AssignSubstructure($a + 1, a + i, i$)
6:     return
7:     $f_{\text{target}} \leftarrow \frac{L_o b}{N}$
8:     $x \leftarrow a$
9:     $\text{partitionIdx} \leftarrow 0$
10:    $i \leftarrow a$
11: while $i <= b$ & $\text{partitionIdx} < N - 1$ do
12:     if $c_f i - p_f \text{partitionIdx} >= f_{\text{target}}$ then
13:         AssignSubstructure($x, i, \text{partitionIdx}$)
14:         $\text{partitionIdx} \leftarrow \text{partitionIdx} + 1$
15:     $x \leftarrow i + 1$
16:     $i \leftarrow i + 1$
17: AssignSubstructure($x, b, N - 1$)
CHAPTER 4. NEARLY OPTIMAL N-ARY SEARCH TREE

contain a pointer to the first and last element of the set as well as pointer to another page. However, the subsequent iteration would fail because the total frequency of the subset is 0, and $N/0$ is not permissible. Thus, if all nodes in the set from $a$ to $b$ do not have a preassigned or known frequency ($f_{ab} = 0$), break the set into $N$ equal parts as shown in Algorithm 2.

**Algorithm 2 No Set Frequency Detection Algorithm**

1: if $f_{ab} == 0$ then
2:    subsetSize ← $(b - a)/N$
3:   for $i = 0; i < N - 1; i + +$ do
4:      $\text{AssignSubstructure}(a + \text{subsetSize} \times i, a + \text{subsetSize} \times (i + 1), i)$
5:   $\text{AssignSubstructure}(a + \text{subsetSize} \times (N - 1), b, N - 1)$
6: return

4.2.3.1 Hyper-Frequent Node Detection

Additionally, there needed to be a way to identify nodes with frequencies greater than the target. As it currently stood, when a node caused the threshold to be surpassed, that node would be included in the partition of that set. However, that node should have been partitioned alone. Consider a set of 6 nodes with the frequency distribution $[0.02, 0.02, 0.02, 0.02, 0.02, 0.9]$. This type of node is referred as Hyper-frequent, meaning it contains sufficient frequency to be referenced directly from the current page. Using an $N$ value of 2, the algorithm would have iterated over the entire set without creating a partition. Only once the iterator moves to include the last node with a frequency of 0.9, will the threshold be hit. Thus, the algorithm would attempt to partition the entire set to be partitioned as a child. Since nothing would change on subsequent iterations, the algorithm would fall into an infinite loop.

In order to identify hyper-frequent nodes, a lookahead is used when iterating through the set. If the node in question is hyper-frequent, partition the subset as well as the node itself assuming there are enough partitions remaining as shown in 3. Using the previous example, the algorithm would iterate through most of the set. The key difference is at the position before the 0.9 node. The algorithm would lookahead, identify the node as hyper-frequent, and create a partition at the $n - 1$ index position. Thus, the remaining set will be sent for subsequent iterations while the root will contain a pointer directly to the hyper-frequent node.

19
CHAPTER 4. NEARLY OPTIMAL N-ARY SEARCH TREE

Algorithm 3 Hyper-frequency Detection

1: \textbf{while} \( i < b \& \& \text{partitionIdx} < N - 1 \) \textbf{do}
2: \quad \textbf{if} \( cf_i - pf_x \geq targetF \& \& f_{i+1} \geq targetF \) \textbf{then}
3: \quad \quad \text{AssignSubstructure}(x, i, \text{partitionIdx})
4: \quad \quad \text{partitionIdx} \leftarrow \text{partitionIdx} + 1
5: \quad \quad \text{partitionIdx} \leftarrow \text{partitionIdx} + 1
6: \quad \quad x \leftarrow i + 1
7: \quad \quad i \leftarrow i + 1

4.2.3.2 Results

Because this method was stable, a large range of sensitivity tests were conducted on this version algorithm. This gambit tests the effectiveness of the algorithm thus far, named the One-directional Algorithm. Using a set size of 10,000 and \( N \) values 8, 12, 16, 20, and 24, we were able to plot the average expected values of five individual tests. Each test iterated through a \( Z \) frequency distribution indicator from 0 to 3. The result of each test can be seen in Figure 4.1, Figure 4.2, Figure 4.3, Figure 4.4, and Figure 4.5.

From the results, a few observations were noted. Firstly, the approximation algorithm is an accurate predictor of the actual performance in all cases. Secondly, the algorithm has a lower expected value than that of the traditional in most cases with the exception of Figure 4.5. Thirdly, the approximation algorithm sometimes predicts a higher expected value than the traditional approach that is accurate in some instances based on the trend of the achieved expected value. Fourthly, as predicted, the greater the frequency distribution, the more benefit is exhibited in our method.

At first, there seems to be very little relationship with the \( N \) value and the outputs of the tests, but we do notice a range in Figure 4.5 with a fanout of \( N = 24 \) where the proposed algorithm performs worse than the traditional approach. This contrasts to Figure 4.1, Figure 4.2, Figure 4.3 and Figure 4.4 which offer steady decreases in expected search cost with increases in set distribution. Further analysis is needed to identify why this is the case. The initial hypothesis pointed to the situation where the relative B+ tree has a high fill ratio.

4.2.4 NP Problem

Even after these improvements to the algorithm, testing resulted in stack overflow exceptions and poorly formed trees in some instances. Situations would occur where some sets had a
CHAPTER 4. NEARLY OPTIMAL N-ARY SEARCH TREE

Figure 4.1: Sensitivity Analysis of One-directional Algorithm with N = 8

Figure 4.2: Sensitivity Analysis of One-directional Algorithm with N = 12
CHAPTER 4. NEARLY OPTIMAL N-ARY SEARCH TREE

Figure 4.3: Sensitivity Analysis of One-directional Algorithm with N = 16

Figure 4.4: Sensitivity Analysis of One-directional Algorithm with N = 20
higher density of frequency on the right side of the set, causing large and poorly constructed blocks on the left and often leading the page to be underutilized.

Upon further inspection, it became apparent that the goal to separate a set into an arbitrary number of bins was in fact an NP-complete problem. The partition problem refers to the desire to separate a set of numbers into two sets where the sum of each subset are equal\[26\]. While certain algorithms can roughly approximate solutions in polynomial time for a one partition, the situation in the current case is constrained further by the necessity to maintain the original order of the nodes and the need to accommodate \( N \) number of partitions.

Adding one more partition to make the 3-partition problem is shown to be strongly NP-complete\[27\]. At its core, the decision of where to place partitions in the set is merely one of combination. Given a set of \( n \) and a number of bins \( N \), we are left with \( n - 1 \) locations to place a partition and \( N - 1 \) partitions to place considering the right and left most sides of the set are by definition partitions. This leaves us with \( \frac{(n-1)!}{(n-N-2)!(N-1)!} \) possible combinations of partitions. In the best case, we would be able to compute standard deviation of each combination in a time complexity of \( O(N) \) and select the combination that minimizes the standard deviation in \( O\left(\frac{n!}{(n-N)!N!}\right) \) which resolves to a time complexity of \( O\left(\binom{n}{N}\right) \) for each set and subset.
CHAPTER 4. NEARLY OPTIMAL N-ARY SEARCH TREE

Such a complexity to analyze a single set is too great to be of any feasible value. As such, we must accept an approximation that can select partitions such that the partitions are nearly equal and provide the result desired. By placing an iterator on each side of the set and moving the iterator towards the center, partitions can be independently created.

This mirrors Mehlhorn’s approach to partitioning a set into two partitions. His algorithm decides to move one iterator as opposed to the other by choosing the side which contains a lesser frequency. While this approach cannot be leveraged completely because of the need to accommodate an arbitrary number of blocks and an arbitrary set size, the convergence model still proves useful.

Instead of approaching the problem purely from the left, we approach it from both sides. However, because no nodes are examined more than once per iteration, we can show that each set can be computed in $O(n)$. However, there is no way to suggest how many iterations of this algorithm are necessary to partition the entire set. A complete example can be seen in Figure 4.6. This example shows how a hyper-frequent node will cause immediate partitioning as well as the benefit of addressing the problem from both sides. The expected search cost for this example is 1.95 which is lower than the B+ height of 2 for the same values of $N$ and $n$.

4.2.5 Enhancements

While the algorithm was now producing viable search structures that performed well when compared to the traditional approach, further enhancements where made to improve performance.

4.2.5.1 Underutilized Blocking

The bi-directional algorithm proved to be move effective at creating well formed pages. However basic testing showed there existed some instances where empty child blocks would exist in the center of the page. To prevent the empty center pages, the assignment of substructure to pages had to be moved as a last phase of the algorithm. This way the proposed partitioning is validated and updated if fewer pages are needed. This allows for a cleaner tree structure and a standard format for terminal pages.

4.2.5.2 Dynamic Threshold Frequency

Another optimization made to the algorithm is to dynamically adjust the threshold frequency after each partition is created. This assists better partitioning in the event of hyper-frequent
CHAPTER 4. NEARLY OPTIMAL N-ARY SEARCH TREE

Figure 4.6: A process showing how iterators would move and partition the set. Triangles represent the iterators, squares represent nodes with keys and frequencies, and horizontal lines represent partitions in the current set. The tree produced is reproduced below the process.
nodes as well as creating more even division of the remaining set among the still available partitions. This is particularly helpful when hyper-frequent nodes skew the cumulative frequency of a subset.

Consider again the set of nodes with a frequency distribution of \([0.02, 0.02, 0.02, 0.02, 0.02, 0.9]\) but with an \(N\) value of 3. Using the Bi-directional algorithm with hyper-frequency detection, the last node would be identified by its own partition. Given that the threshold frequency for this set would be \(1/3\), the remaining nodes would not be able to exceed it. Thus, dynamic thresholding allows the algorithm to remove the impact of the node with a frequency of 0.9. After the hyper-frequent node’s frequency is discounted, the target threshold would dynamically become \(1/30\), which can be surpassed by a subset of the remaining nodes.

### 4.2.5.3 Results

Using the same gambit of parameters, the tests were repeated using the latest algorithm, denoted the Bi-directional Algorithm. The result of each test can be seen in Figure 4.7, Figure 4.8, Figure 4.9, Figure 4.10, and Figure 4.11. Generally, observations demonstrated a decrease in expected value when Algorithm 3 was applied to the datasets. This shows that the convergence approach as well as other optimizations provided benefit to the overall structure of the data. This round of testing maintains previous assumptions and observations. Particularly, the bump in expected value in Figure 4.11 is interesting as it matches with the trend of the previous exam even though the expected cost is lessened. The other test casts show steady decreases in cost as variance increases. The slope of these cases have increased when compared to the previous round of tests. This decrease in cost indicates that the bi-directional approach is better at assigning sets of near equal frequency.

### 4.2.6 Further Enhancements

#### 4.2.6.1 Optimally Structuring Low or No Variance Sets

Initial testing of sets with low variance showed an anomaly where the expected search time in a tree constructed using this method was marginally higher than that of a traditionally structured tree. Debugging showed that this was caused by the blind subdivision of a set into \(N\) equal parts. This led to underutilization of the page blocks. For example, if \(N\) was 6, and a set of 8 were considered, the blind algorithm would separate the set into 6 sets of 1 or 2. This is inefficient as one set of 3 could be assigned to a child page while keeping the remaining 5 on the upper page.

The benefit of relaxing the requirement that all nodes have to exist in the same level, as with a B+ Tree, allows us to optimally assign nodes with the same frequency. As such Algorithm 7
CHAPTER 4. NEARLY OPTIMAL N-ARY SEARCH TREE

Figure 4.7: Sensitivity Analysis of Bi-directional Algorithm with N = 8

Figure 4.8: Sensitivity Analysis of Bi-directional Algorithm with N = 12
Figure 4.9: Sensitivity Analysis of Bi-directional Algorithm with $N = 16$

Figure 4.10: Sensitivity Analysis of Bi-directional Algorithm with $N = 20$
was created. The method breaks down a larger set into equal subsets until the set is equal or less than \( N^2 \). When this condition is reached, all elements can be stored at the same level. While the traditional algorithm will create the level and fill the leaves accordingly, this is not always the most efficient structure. Because we can assign nodes to blocks on different levels, we can create as many sub-pages as necessary but then allocate the rest on the current page’s remaining blocks. In this way, we raise as many nodes as possible into a higher level and minimize the expected search cost for this subtree.

This is done by first partitioning as many full pages as possible. Then, another page may be created if the number of nodes remaining is greater than the number of unassigned blocks to prevent overflow. Finally, remaining nodes are assigned into the remaining blocks.

**4.2.6.2 Deciding How to Structure a Set**

With the new algorithm to structure low variance sets, we were able to achieve a state where sets structured with this data structure achieved a lower expected search cost than the traditional B+ tree in all instances except when \( n \) is a power of \( N \) (because all leaf pages would by definition be full). Given sets with low variance, we found that structuring the set to reduce height performed
CHAPTER 4. NEARLY OPTIMAL N-ARY SEARCH TREE

Figure 4.12: A comparison between the structure of a B+ tree and the nearly optimal tree.

better than using the partitioning approach. In order to determine which method to use, we can rely on the approximation of the expected search cost of a set structured according to the partitioning method.

Because the low variance algorithm creates a tree that utilizes the available blocks in order to reduce height, it will always create a tree with a search cost that is equal or less than the traditional structure. However, because the goal of the low variance algorithm is purely height reduction, the relative frequency of the nodes is not considered. This type of structuring is permissible only because the constraints of the B+ have already been removed.

Because the set is structured in a similar but more efficient manner than the B+ tree in terms of expected search cost, the expected performance of the low variance tree is guaranteed to be less than the traditional B+ tree.

4.2.6.3 Results

The final algorithm developed is annotated as the Enhanced Bi-directional Algorithm. The results for this approach can be seen in Figure 4.13, Figure 4.14, Figure 4.15, Figure 4.16 and Figure 4.17. With this set of results, it can be observed that the actual expected value falls below that of the traditional method as well as the approximate expected value in all instances. This verifies the correctness of the conditional that decides whether to structure the set using the bi-directional method or the low variance method.
In order to clearly see the relationship between performances, the average actual expected search costs were plotted against each other for each of the \( N \) test cases. These can be seen in Figure 4.18, Figure 4.19, Figure 4.20, Figure 4.21, and Figure 4.22. It is clear that Enhanced Bi-directional Algorithm generally out-performs the other methods. There is some crossover that occurs where the Enhanced Bi-directional starts to drop from its plateau for low variance sets. The plateau can be attributed to the low variance method for building the structure of the tree. Because this value can be calculated based on the set size, the result is constant agnostic of the distribution of the set. The lag between this value and greater performance gain can further be explained by the nature of the algorithm. The decision to structure the set in one way or the other is based on the approximate expected value that is determined in the first iteration over the set. If the approximation is greater than the traditional, then use the low variance method.

The approximation is not perfect as it generally approximates a higher number than the partitioning method. The reason for this lies in the data structure itself. A node may have an extremely low frequency, thus be approximated to be on a low level. However, the tree may assign that node to a higher level if there is room for it to exist on a page. This trade-off is acceptable though as there is still guaranteed performance gain on the low variance method over the B+ approach for this particular set distribution.

Since performance gain over the B+ tree can now be ensured, the performance goal has been met.
CHAPTER 4. NEARLY OPTIMAL N-ARY SEARCH TREE

Algorithm 4 Convergence Algorithm

1: procedure OPTIMALLYSTRUCTURE($a$, $b$, lowVariance)
2:   $f_{ab} \leftarrow cf_b - pf_a$
3:   if $b - a < N$ then
4:     for $i = 0; i < b - a; i++$ do
5:       AssignSubstructure($a + 1, a + i, i$)
6:     return
7:   if $f_{ab} == 0$ or lowVariance then
8:     OptimallyStructureLowVariance($a, b, true$)
9:   sameF $\leftarrow$ true
10:  partitions $\leftarrow$ GetPartitions($a, b, ref sameF$)
11:  if sameF then
12:     OptimallyStructureLowVariance($a, b, true$)
13:  return
14:  partitions.add($b + 1$)
15:  sort(partitions)
16:  for $p = 0; p < partitions.size() - 1; p++$ do
17:     OptimallyStructureLowVariance($partitions[p], partitions[p + 1], false$)

Algorithm 5 No Variance Partitioning

1: procedure OPTIMALLYSTRUCTURELOWVARIANCE($a$, $b$)
2:   subsetSize $\leftarrow \frac{b - a}{N}$
3:   for $p = 0; p < N; p++$ do
4:     AssignSubstructure($a + subsetSize \times p, a + subsetSize \times (p + 1) - 1, p, true$)
5:     AssignSubstructure($a + subsetSize \times (N - 1), b, N - 1, true$)
CHAPTER 4. NEARLY OPTIMAL N-ARY SEARCH TREE

Algorithm 6 Get Partitions

1: procedure GETPARTITIONS(a, b)
2:    \[ f_{\text{target}} \leftarrow \frac{f_{ab}}{N} \]
3:    \[ x \leftarrow a \]
4:    \[ x_{\text{last}} \leftarrow a \]
5:    \[ y \leftarrow b \]
6:    \[ y_{\text{last}} \leftarrow b \]
7:    \[ \text{partitionIdx}_x \leftarrow 0 \]
8:    \[ \text{partitionIdx}_y \leftarrow N - 1 \]
9:    \[ \text{toggle} \leftarrow \text{true} \]
10:   \[ \textbf{while } x \leq y \& x_{\text{last}} \neq y_{\text{last}} \textbf{ do} \]
11:      \[ f_{\text{target}} \leftarrow \frac{f_{x_{\text{last}}} \cdot \text{partitionIdx}_x}{f_{y_{\text{last}}} \cdot \text{partitionIdx}_x} (\text{partitionIdx}_y - \text{partitionIdx}_x + 1) \]
12:      \[ \text{sameF} \leftarrow (\text{sameF}) \text{AND}(f_x == f_y) \]
13:      \[ \textbf{if } \text{toggle} \textbf{ then} \]
14:         \[ \text{if } (c_{f_x} - p_{f_x_{\text{last}}} \geq f_{\text{target}}) || f_{x+1} \geq f_{\text{target}} \textbf{ then} \]
15:            \[ \text{partitions.add}(x_{\text{last}}) \]
16:            \[ \text{partitionIdx}_x \leftarrow \text{partitionIdx}_x + 1 \]
17:            \[ x_{\text{last}} \leftarrow x + 1 \]
18:            \[ i \leftarrow i + 1 \]
19:            \[ \text{toggle} \leftarrow f_{x+1} >= f_{\text{target}} \]
20:      \[ \textbf{else} \]
21:         \[ \text{if } (c_{f_y} - p_{f_y} \geq f_{\text{target}}) || f_{y-1} \geq f_{\text{target}} \textbf{ then} \]
22:            \[ \text{partitions.add}(y) \]
23:            \[ \text{partitionIdx}_y \leftarrow \text{partitionIdx}_y - 1 \]
24:            \[ y_{\text{last}} \leftarrow y - 1 \]
25:            \[ y \leftarrow y - 1 \]
26:            \[ \text{toggle} \leftarrow f_{y-1} >= f_{\text{target}} \]
27:      \[ \textbf{if } \text{partitionIdx}_y - \text{partitionIdx}_x == 0 \text{ and } y_{\text{last}} \leq x_{\text{last}} \textbf{ then} \]
28:         \[ \text{partitions.add}(x_{\text{last}}) \]
29:      \[ \textbf{if } \text{sameF} \textbf{ then} \]
30:         \[ \textbf{for } i = x_{\text{last}}; i \leq y_{\text{last}}; i + + \textbf{ do} \]
31:            \[ \text{if } x_{\text{last}}! = f_i \textbf{ then} \]
32:               \[ \text{sameF} \leftarrow \text{false} \]
33:   \]
CHAPTER 4. NEARLY OPTIMAL N-ARY SEARCH TREE

Algorithm 7 Final Low Variance Algorithm

1: procedure OPTIMALYSTRUCTURELOWVARIANCE(a, b, lowvar)
2:     nodesRemaining ← b − a + 1
3:     partitionsRemaining ← N
4:     if N * N >= nodesRemaining then
5:         pagesToAlloc ← ⌈nodesRemaining−N−1⌉
6:         block ← 0
7:         highNodes ← N − pagesToAlloc
8:         lastLow ← b − highNodes
9:         while pagesToAlloc > 0 do
10:             AssignSubstructure(a + block * N, min(a + N * (block + 1) − 1, lastLow), block, lowVar)
11:             block ++
12:             partitionsRemaining −−
13:             nodesRemaining− = N
14:             pagesToAlloc −−
15:             leftBreak ← leftBreak = lastLow + 1
16:             while leftBreak <= b do
17:                 AssignSubstructure(leftBreak, leftBreak, block, lowVar)
18:                 leftBreak ++
19:                 block ++
20:                 nodesRemaining −−
21:                 partitionsRemaining −−
22:         else
23:             subsetSize ← nodesRemaining
24:             remaining ← nodesRemaining%N
25:             offset ← 0
26:             for p = 0; p < N; p ++ do
27:                 localPart = (p < remaining)?subsetSize + 1 : subsetSize
28:                 AssignSubstructure(a + offset, s + offset + localPart − 1, p, lowVar)
29:             return
CHAPTER 4. NEARLY OPTIMAL N-ARY SEARCH TREE

Figure 4.13: Sensitivity Analysis of Enhanced Bi-directional Algorithm with N = 8

Figure 4.14: Sensitivity Analysis of Enhanced Bi-directional Algorithm with N = 12
CHAPTER 4. NEARLY OPTIMAL N-ARY SEARCH TREE

Figure 4.15: Sensitivity Analysis of Enhanced Bi-directional Algorithm with N = 16

Figure 4.16: Sensitivity Analysis of Enhanced Bi-directional Algorithm with N = 20
CHAPTER 4. NEARLY OPTIMAL N-ARY SEARCH TREE

Figure 4.17: Sensitivity Analysis of Enhanced Bi-directional Algorithm with N = 24

Figure 4.18: Side by Side Actual Expected Search Costs with N = 8
CHAPTER 4. NEARLY OPTIMAL N-ARY SEARCH TREE

Figure 4.19: Side by Side Actual Expected Search Costs with N = 12

Figure 4.20: Side by Side Actual Expected Search Costs with N = 16
CHAPTER 4. NEARLY OPTIMAL N-ARY SEARCH TREE

Figure 4.21: Side by Side Actual Expected Search Costs with N = 20

Figure 4.22: Side by Side Actual Expected Search Costs with N = 24
Chapter 5

Evaluation

In order for this data structure and algorithm to be useful, it must exhibit gains as well as scalability. Even in the Sensitivity Analysis test bench, there are clear variances in performance gains dependent on the page size that is used. In order to explore this relationship, two indicators must be defined.

Firstly, Fill Ratio is the ratio of the number of nodes in a set over the number of possible slots in the correlating B+ tree. This ratio can be calculated as $\text{FillRatio} = \frac{n}{\lceil N^{\log_{10}(N)} \rceil}$. This ratio is significant because it precludes our algorithm’s ability to assign nodes to higher levels. It can be predicted that combinations with a lower Fill Ratio will exhibit higher amounts of performance gain.

Secondly, Performance Ratio is the ratio of the actual expected value over the traditional expected value or $\frac{e(\text{actual})}{e(\text{traditional})}$. Since the algorithm is guaranteed to always have a lower actual expected value than the traditional, this ratio will always be less than 1. Indeed, the lower the ratio, the higher the performance gain.

The first step in determining the Fill Ratio’s importance is to plot it against varying $N$ values for varying set sizes $n$. Since $N$ can only be an integer, both the function as well as the step function were plotted to show the relationship. The plots of the different values of $n$ vs the corresponding Fill Ratio are shown in Figure 5.1, Figure 5.2, Figure 5.3, Figure 5.4, and Figure 5.5. As can be intuitively predicted, the Fill Ratio is high when a new level must be allocated for the set size. For example, in Figure 5.1 we see spikes in the lower values of $N$ as well as at 10 and 22. This is because the B+ tree with four levels contains 10,000 slots, which would give one slot to each of the nodes. While actual B+ implementation would not allow each page to become this full, that pragmatism is ignored for comparison sake. We also see a spike at 22 because a tree with three levels would contain 10,648 slots. This logic can be likewise applied to Figure 5.2, Figure 5.3, Figure 5.4.
Figure 5.6 shows the same graph as Figure 5.1 but with experimental data for \( Z = 1 \) overlaid on top. For practicality reasons, \( N \) values less than 5 were ignored. The purple series shows the plotted approximate expected value and the green series represents our actual expected value. From this graph we can see a correlation between the Fill Ratio and the Performance Ratio. The Fill Ratio for \( N \in \{8, 12, 16, 20\} \) is lower than for \( N = 24 \). Given that for \( N \neq 24 \) in the Sensitivity Analysis test bench exhibited a higher gain, the hypothesis that Fill Ratio plays a role expected performance gain is validated.

In order to see the relationship in an even clearer way, a scatter plot mapping the two ratios can be seen in Figure 5.7 for \( n = 10,000 \) and \( Z \) values of 1, 2, 3, and 4. This scatterplot shows an undeniable positive correlation between the ratios. However, it does not demonstrate a clear mathematical model or decent \( R^2 \) value.

When separated into categories based on the traditional expected value, the correlation becomes pronounced. The two test scenarios that produced the most results were those where the height of the B+ would be 3 or 4. Thus, they were plotted in Figure 5.8 and Figure 5.9 respectively. Based on these subsets of data, it is shown once again that the algorithm produces a more efficient search structure than that of the B+ tree. Secondly, by choosing an \( N \) value that decreases the Fill Ratio, the performance of a NONST can be further improved. It is important to note that the performance of the B+ tree is not actually calculated as this section would suggest. Instead, B+ tree’s are kept around \( 2/3 \) utilized to account for key insertions. However, the B+ tree’s lower bound of the height follows this formula. Meaning, in the suggested instances where the Fill Ratio is greater than \( 2/3 \), a B+ tree would have already allocated another level. This would result in even higher gain when compared to a NONST.
CHAPTER 5. EVALUATION

Figure 5.1: Fill Ratio by N for a set size of 10,000

Figure 5.2: Fill Ratio by N for a set size of 100,000
CHAPTER 5. EVALUATION

Figure 5.3: Fill Ratio by N for a set size of 1,000,000

Figure 5.4: Fill Ratio by N for a set size of 10,000,000
CHAPTER 5. EVALUATION

Figure 5.5: Fill Ratio by N for a set size of 100,000,000

Figure 5.6: Fill Ratio by N for a set size of 10,000 compared to approximate and actual expected values
CHAPTER 5. EVALUATION

Figure 5.7: Performance Ratio vs Fill Ratio

Figure 5.8: Performance Ratio vs Fill Ratio where the traditional expected search cost = 3
Figure 5.9: Performance Ratio vs Fill Ratio where the traditional expected search cost = 4
Chapter 6

Case Studies

This section will discuss real life datasets that could benefit from this algorithm as well as other potential applications for it.

6.1 ICD10 Data Modeling

The World Health Organization publishes a database of mortality information based on national numbers and the internationally recognized ICD10 schema\[28\]. By using the key structure native to the database and the mortality numbers as frequencies for the database, structuring the data using the proposed method shows a sizable gain in overall expected retrieval time when querying the database.

To better illustrate how the algorithm works on the data and to visualize where gain is achieved, the tree was broken down into histograms based on the percent of nodes that exist within each level on the tree as well as the frequency that exists on that level for the nodes that exist there. A wide range on N values were tested, all showing performance gain.

Figure 6.1 and Figure 6.2 were selected to show the scalability of N. Intuitively, a smaller N value produces taller trees and can thus shows a more impressive histogram. From these graphs, the Expected Search Cost is the average of the blue frequency area and is lower than the traditional cost. The significance of the graph shows how the tree was successfully able to identify nodes that contain the highest frequency and float them towards the top of the tree. The cost of this being that nodes with lower frequencies are deeper in the tree.

However, by viewing these histograms as cumulative histograms as in Figure 6.3 and Figure 6.4 by the time the structure under-performs compared to the traditional model, the vast
CHAPTER 6. CASE STUDIES

Figure 6.1: Histogram of ICD10 data where N=6

Figure 6.2: Histogram of ICD10 data where N=99
majority of frequency in the tree has been exhausted. This trade-off was predicted, and these graphs show why it is acceptable. By rejecting the guarantee of performance in the traditional model, overall performance of the data structure can be improved.

Databases and datasets that are very skewed distributed, like the ICD10, would benefit greatly from the proposed data structure.

6.2 US Census Data

To show scalability, a larger dataset was obtained showing the estimated population of the United States for every city and state in the country in 2010 [29]. Because of the larger size of the dataset, it became a good candidate for showing how effective NONST is at allocating nodes with higher frequency in the tree. For the purposes of this test, frequency of access was determined to be related to the population of the city, where the city and state are the search keys.

If \( N \) is set to 5, the traditional B+ tree would place all 81,611 entries on the 8th level. Figure 6.5 shows that NONST instead produces a tree with an expected search cost of 5.5512, resulting over a 40% reduction in search cost. The histogram shows clearly the frequency population centers over the expected search cost while the node population centers over the B+ cost. Figure 6.6 further shows the efficiency of the tree by visually displaying that the vast majority of searches will be complete before performing worse than the B+ approach.
Because the dataset was very easy to manipulate, the effectiveness of the approximation algorithm could be explored. For each level, the partitioning target frequency on each level was identified. By counting the number of cities that fall above that partitioning target, the level the cities would be placed in the tree could be predicted. After the algorithm was run, the results could be compared to the actual placement of cities on levels. Table 6.1 shows this breakdown. The results of this analysis are interesting because NONST is able to provision nodes on higher levels than what would be reasonably predicted. The reasons for this can be attributed to the many optimization techniques used to increase performance. For example, nodes that are predicted to be on level 14 would find a position on a page higher in the tree where the set containing that node is less than \( N \). Additionally, nodes could find themselves being considered hyper-frequent on a higher level if dynamic threshold adjusted the partition target frequency.

Since population distribution is considered a power law distribution, this experiment shows how effective NONST is at digesting and optimizing datasets that have access distributions modeled in a similar fashion.
CHAPTER 6. CASE STUDIES

Figure 6.5: Histogram of Census data where N=5

Figure 6.6: Cumulative histogram of Census data where N=5
### Table 6.1: Node placement prediction vs actual node placement for Census data.

<table>
<thead>
<tr>
<th>Level</th>
<th>Partition Target</th>
<th>Nodes Fitting Range</th>
<th>Actual Nodes Placed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0.2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0.04</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>8.00E-3</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>1.60E-3</td>
<td>49</td>
<td>57</td>
</tr>
<tr>
<td>5</td>
<td>3.20E-4</td>
<td>299</td>
<td>454</td>
</tr>
<tr>
<td>6</td>
<td>6.40E-5</td>
<td>1628</td>
<td>3345</td>
</tr>
<tr>
<td>7</td>
<td>1.28E-5</td>
<td>7571</td>
<td>16835</td>
</tr>
<tr>
<td>8</td>
<td>2.56E-6</td>
<td>15946</td>
<td>36483</td>
</tr>
<tr>
<td>9</td>
<td>5.12E-7</td>
<td>25572</td>
<td>22706</td>
</tr>
<tr>
<td>10</td>
<td>1.02E-7</td>
<td>20792</td>
<td>1711</td>
</tr>
<tr>
<td>11</td>
<td>2.05E-8</td>
<td>7495</td>
<td>14</td>
</tr>
<tr>
<td>12</td>
<td>4.10E-9</td>
<td>1300</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>8.19E-10</td>
<td>282</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>1.64E-10</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>640</td>
<td>0</td>
</tr>
</tbody>
</table>

*Table 6.1: Node placement prediction vs actual node placement for Census data.*
Chapter 7

Conclusion

Advances in technology have opened the door to innovation. Whereas complex computation and data storage was previously cost prohibitive, more entities are now collecting and storing vast amounts of data. As the scope of this data enters higher orders of magnitude, methods of searching through it must also adapt. Research has shown that data follows certain patterns and distributions. Veritable explosions in social networks and web traffic follows highly distributed and skewed data. Having this domain knowledge, it seems prudent to store information in a manner to maximize performance in data structures.

Current structures offer a wide range of benefits because of their simplistic design and guaranteed performance. This attribute is particularly important to the B+ tree. B+ trees are commonly used database indexes and bear the brunt of information retrieval. They maintain their performance benefits by being very scalable and treating all nodes as equal contributors. By assuming that individual nodes do not contribute equally in terms of access frequency, the structure can be modified such that more commonly requested nodes are higher in the search tree.

The Nearly Optimal N-ary Search Tree reduces the constraints set upon the B+ tree to take advantage of innate frequency distributions in a dataset. Nodes with greater frequency will be placed higher in the tree while infrequently accessed nodes will be placed lower. This placement based on distribution will cause a reduction in expected search cost.

In order to provision the data structure in such a manner, NONST provides an algorithm. First, divide the set into \( N \) partitions. The value of \( N \) can be selected based on system requirements or such that the fill ration for the tree will be reduced. Once \( N \) is determined, partition the data into \( N \) segments. The algorithm described here takes a bi-directional, converging approach. As the algorithm iterates through the set, maintain a target frequency. Once a subset contains that target
frequency, identify it as a partition. If a node contains a frequency greater than the target, ensure it is partitioned alone. As partitions are made, adjust the target frequency to more evenly distribute remaining nodes into partitions. Once all partitions are made, update the pointers on the current page as follows:

1. If a node is in a partition alone, reference the node directly.
2. If a partition contains a subset greater than $N$, create a new page, reference the new page, and apply the algorithm again.
3. If a partition contains a subset less than or equal to $N$, create a new page with references to the individual nodes.

To calculate expected value, traverse the tree while maintaining a sum of the product of the node’s frequency and the level it was placed on in the tree.

Analysis of the test results concluded that NONST-structured data performs better when considering data of high distribution. As the variance increases, so does the performance gain when compared to the height of the corresponding B+ tree. Case studies of real data further verified the hypothesis. Disease data taken from the World Health Organization showed that when structured based on mortality data, estimated search time was reduced by nearly 45% when $N = 6$. Data taken from the U.S. Census from 2010 showed a reduction of over 40% when $N = 5$. Data like diseases and population density follow a power law. Therefore, it can be predicted that other distributions following such a pattern will exhibit similar performance gains when structured in a NONST.

The theory and implementation proposed offers the foundation for a generic search structure that is meant to reduce the average time necessary to perform search queries. In order to become a dynamic structure, methods implementing insertion and deletion from the tree must be created. To fully utilize the concepts underlying the structure, it must additionally be able to adapt to changes in frequency. Further enhancements could be made to provision a NONST in a certain manner to accommodate fluctuations in system usage based on prior knowledge.

The Nearly Optimal N-ary Search Tree continues the trend of analyzing data to make better decisions. The application of heuristics to data structure formation is a step towards efficiently performing and adaptive systems. B+ trees-and other data structures-are limited because of their guarantees. Instead, by trusting domain and prior knowledge of data, search structures can be optimized. While the implementation of this knowledge and the formation of good heuristics is
CHAPTER 7. CONCLUSION

dependent on the data structure itself, a Nearly Optimal N-ary Search Tree consistently provides more efficient data storage over a B+ tree.
Chapter 8

Future Work

In order to make NONST a more general structure, it must be able to implement more operations than formation and search. To be considered a dynamic set, it must be able to perform some modifying operations like insert and delete \[30\]. It would also be beneficial to analyze the effect of the entropy of the dataset.

Deletion will work the same way as a B+ tree. Insertion can occur in two ways: insert a node with no known frequency or insert a node with a known frequency.

Without a known frequency, the tree can be traversed until the appropriate key position is found. As the tree is traversed, identify if there are any open pointer positions that would maintain the key structure. If so, provision the node on the non-leaf page, otherwise traverse to the leaf page. If there is space on the leaf page where the node should be placed, simply add the node. If there is no space on the leaf page, apply the optimally structure method on the set contained within the page.

If a frequency is known, a similar procedure can be applied. Using the level prediction function, follow the key structure until a page on the target level is reached. If there is an open position, add a reference to the node. Otherwise, consider the entire set contained within the tree starting at the page and apply the optimally structure method. By allowing this provisioning, the substructure of the data can be dynamically modified. For example, new records in a database may be accessed more frequently than older records. Thus, by inserting new records with an arbitrarily high frequency, better performance can ensured when retrieving that record in the near future. Conversely, older data will fall deeper into the tree. However this is acceptable because of the domain knowledge that older records will not be accessed as frequently. By tailoring the insertion heuristic, applications can adjust how they would like their NONST-based database to work.

The challenge in maintaining a NONST structure is deciding how and when to update the
CHAPTER 8. FUTURE WORK

underlying structure. Throughout the lifetime of the database, the access frequency might change, or there might be a need to dynamically change the structure. For example, research demonstrates that Internet usage changes dynamically based on the time of day in a given time zone [31]. A large but undistributed system would benefit by prioritizing usage in a certain time zone based on the time of day. In this sense, users would have a higher performance when they are more active. This type of restructuring is left discretionary based on the needs of the consumer of the data structure.
Bibliography


