University of Kentucky
UKnowledge

Theses and Dissertations--Computer Science | Computer Science

2018

Improving Table Scans for Trie Indexed Databases

Ethan Toney
University of Kentucky, egto232@gmail.com
Digital Object Identifier: https://doi.org/10.13023/etd.2018.488

Right click to open a feedback form in a new tab to let us know how this document benefits you.

Recommended Citation

Toney, Ethan, "Improving Table Scans for Trie Indexed Databases" (2018). *Theses and Dissertations--Computer Science*. 76.
https://uknowledge.uky.edu/cs_etds/76

This Master's Thesis is brought to you for free and open access by the Computer Science at UKnowledge. It has been accepted for inclusion in Theses and Dissertations--Computer Science by an authorized administrator of UKnowledge. For more information, please contact UKnowledge@lsv.uky.edu.

STUDENT AGREEMENT:

I represent that my thesis or dissertation and abstract are my original work. Proper attribution has been given to all outside sources. I understand that I am solely responsible for obtaining any needed copyright permissions. I have obtained needed written permission statement(s) from the owner(s) of each third-party copyrighted matter to be included in my work, allowing electronic distribution (if such use is not permitted by the fair use doctrine) which will be submitted to UKnowledge as Additional File.

I hereby grant to The University of Kentucky and its agents the irrevocable, non-exclusive, and royalty-free license to archive and make accessible my work in whole or in part in all forms of media, now or hereafter known. I agree that the document mentioned above may be made available immediately for worldwide access unless an embargo applies.

I retain all other ownership rights to the copyright of my work. I also retain the right to use in future works (such as articles or books) all or part of my work. I understand that I am free to register the copyright to my work.

REVIEW, APPROVAL AND ACCEPTANCE

The document mentioned above has been reviewed and accepted by the student's advisor, on behalf of the advisory committee, and by the Director of Graduate Studies (DGS), on behalf of the program; we verify that this is the final, approved version of the student's thesis including all changes required by the advisory committee. The undersigned agree to abide by the statements above.

Ethan Toney, Student

Dr. Jerzy W. Jaromczyk, Major Professor

Dr. Mirek Truszczynski, Director of Graduate Studies

Improving Table Scans for Trie Indexed Databases

THESIS

A thesis submitted in partial
fulfillment of the requirements for
the degree of Master of Science in
the College of Engineering at the
University of Kentucky

By
Ethan Toney
Lexington, Kentucky

Director: Dr. Jerzy W. Jaromczyk, Professor of Computer Science
Lexington, Kentucky 2018

Copyright© Ethan Toney 2018

Abstract of Thesis

Improving Table Scans for Trie Indexed Databases

We consider a class of problems characterized by the need for a string based identifier that reflect the ontology of the application domain. We present rules for string-based identifier schemas that facilitate fast filtering in databases used for this class of problems. We provide runtime analysis of our schema and experimentally compare it with another solution. We also discuss performance in our solution to a game engine. The string-based identifier schema can be used in addition scenarios such as cloud computing. An identifier schema adds metadata about an element. So the solution hinges on additional memory but as long as queries operate only on the included metadata there is no need to load the element from disk which leads to huge performance gains.

KEYWORDS: key-value, identifier schema, pattern schema, compressed prefix tree, game engine, resource manager

Author's signature: Ethan Toney

Date: December 17, 2018

Improving Table Scans for Trie Indexed Databases

By
Ethan Toney

Director of Thesis: Jerzy W. Jaromczyk

Director of Graduate Studies: Mirek Truszczynski

Date: December 17, 2018

To my friends and family

ACKNOWLEDGMENTS

I want to thank my advisor, Dr. Jaromczyk, for providing guidance over the years and for pushing me to meet my goals. I would also like to thank my friend and colleague Cody Barnes for working with me over all these years.

TABLE OF CONTENTS

Acknowledgments iii

Table of Contents iv

List of Tables v

List of Figures vi

Chapter 1 Introduction 1
- 1.1 Goal 1
- 1.2 Background 1
- 1.3 Problem 6
- 1.4 Layout of Thesis 7

Chapter 2 Discussion 8
- 2.1 Modeling Game Engine Resources 8
- 2.2 Selecting Resources 10
- 2.3 Data Structure Selection 10

Chapter 3 Algorithms 18
- 3.1 Insert 19
- 3.2 Delete 24
- 3.3 Get 26
- 3.4 Pattern Select 27
- 3.5 Link 30
- 3.6 Nested Select 31

Chapter 4 Case Studies 35
- 4.1 Resource Management in Game Engines 35
- 4.2 AWS Resource Management 38

Chapter 5 Conclusion 40

Appendices 41
- .1 Implementation 42
- .2 Tests 54

Bibliography 62

Vita 63

LIST OF TABLES

1.1 Complexity of array backed resource manager 4
1.2 Complexity of map backed resource manager 4
1.3 Complexity of tree backed resource manager 6

LIST OF FIGURES

1.1 MVC Game Engine Architecture 2
1.2 Screenshot of Bulwark 3

2.1 Example of game engine ontology 8
2.2 Trie (left) and compressed prefix tree (right) containing words: a, and, are, arrow, as, at, be, but, by 11
2.3 Trie (left) and compressed prefix tree (right) containing words: a, aa, aaa, aaaa, aaaaa 12

3.1 Tree containing: a, and, are, as, at, be, but, by. Purple line shows inorder traversal path 18
3.2 Compressed prefix tree containing: a, and, are, as, at, be, but, by. Purple line shows inorder traversal path 19
3.3 Pattern selection: tree vs hash map: $g = 1, p = 0, q = 1$ 29
3.4 Pattern selection: tree vs hash map: $g = 2, p = 0, q = 1$ 30
3.5 Pattern selection: tree vs hash map: $g = 1, p = 1, q = 2$ 30
3.6 Selection: tree vs hash map: $l = 1, q = 1, g = 1, r = 1$ 32
3.7 Selection: tree vs hash map: $l = 1, q = 1, g = 2, r = 1$ 33
3.8 Selection: tree vs hash map: $l = 1, q = 2, g = 1, r = 1$ 33
3.9 Selection: tree vs hash map: $l = 2, q = 1, g = 1, r = 1$ 34
3.10 Selection: tree vs hash map: $l = 2, q = 1, g = 1, r = 2$ 34

Chapter 1 Introduction

1.1 Goal

The goal of this thesis is to present and propose a solution to the problem of resource management. The thesis accomplishes this by narrowing down the problem, mentioning insights, presenting an algorithmic solution, and then comparing it to the most complicated but optimal solution.

1.2 Background

Since 2015 I have been working on a general purpose game engine with a few colleagues. Over the years we have had to address various problems ranging from rendering to serialization. The most fundamental problem addressed was storing all of the resources used by the programmers, artists, and writers. The portion of code responsible for handling this is called the resource manager. The problem of managing resources is not unique to the gaming industry. Almost every field has its version of the problem. However, the gaming industry presents a unique set of restrictions. Consider recent AAA video games; they render these beautifully intricate scenes to the screen. Each scene comprised of models, meshes, masks, and filters. Unlike in the movie industry, which has hours to render a single frame of animation, the video game industry has at most sixteen milliseconds to display the next scene. The reason it is sixteen milliseconds is due to games being rendered at sixty frames a second; anything slower than this appears visually choppy to the human eye. Some video games have even faster screen refresh rates. Specifically, virtual reality games which require the display to be updated ninety times a second; anything less than ninety frames a second can lead to motion sickness. When meeting ninety frames a second, the graphics engine roughly has 11 milliseconds to update the screen!

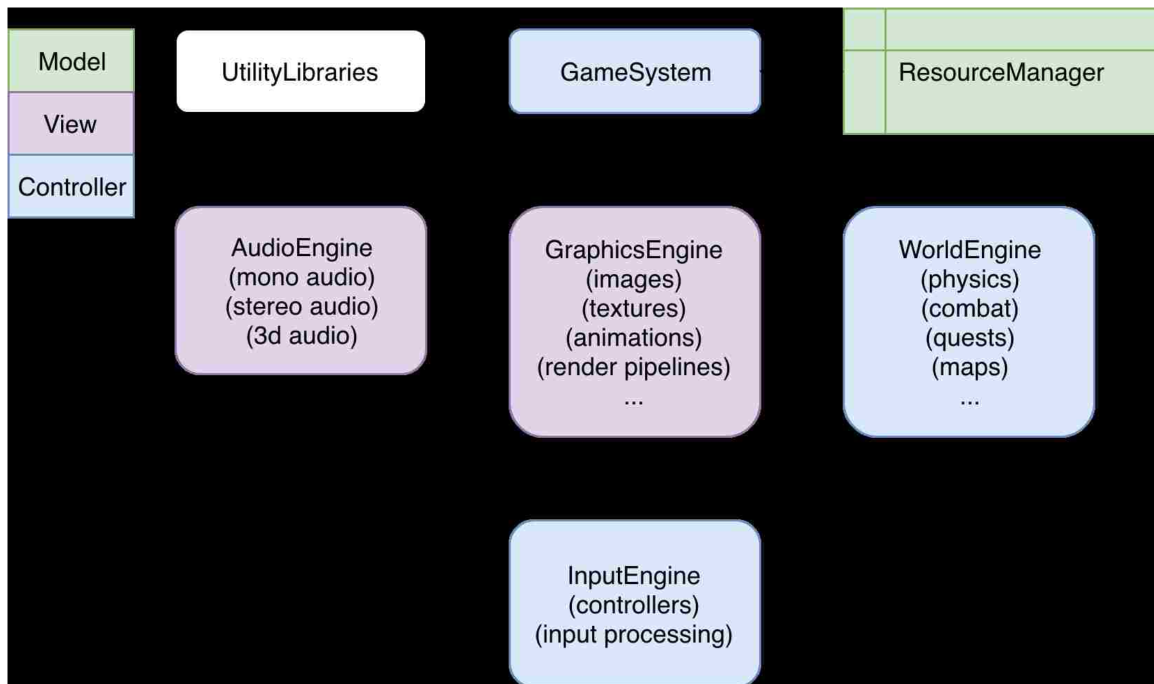
To illustrate this requirement, consider a virtual reality game which is running on a screen that is 1980 by 1028 pixels. This screen has around two million pixels. That means the program pushes two million pixels to the screen every 11 milliseconds. Also, consider that the process to compute every single pixel involves doing complex lighting computations as well as various depth filters. Given all of this information, it's clear to see that there are extreme graphical constraints placed on video game developers. Despite this, it has resulted in the gaming industry spearheading advancements in real-time rendering.

The industry as a whole has been given a keen sense for performance and optimization. One of the most well-known optimizations is called, "fast inverse square root." This optimization was to find a way to quickly estimate the value of $1/\sqrt{x}$ before computers could efficiently compute the square root of a variable. It involved bit shifting the float representation of the number by a magic constant and performing newtons method on the result. All this work allowed the developers to do real-time lighting computations within an acceptable error which users couldn't visually detect. This

situation doesn't directly apply to the discussion but emphasizes how much work is put into optimizing games. Despite all the optimizations over the years, games spend the vast majority of their time on graphics. From my own experience, the game spends approximately 70-80% of its time rendering each frame.

Rendering requires so much time that the remainder of the game has to run efficiently. Therefore the audio, world, physics, input, and all other parts of the engine don't have much time each frame; keep in mind we are talking about a portion of the 11-16 milliseconds allocated for each frame. To get a better idea of what components make up a game engine, consider figure 1.1. The figure is color coordinated, breaking up each portion into a model view controller, MVC, component system. Under an MVC system, the model is the underlying data. The display of the model to the user is the view. The controller determines what to display in the model.

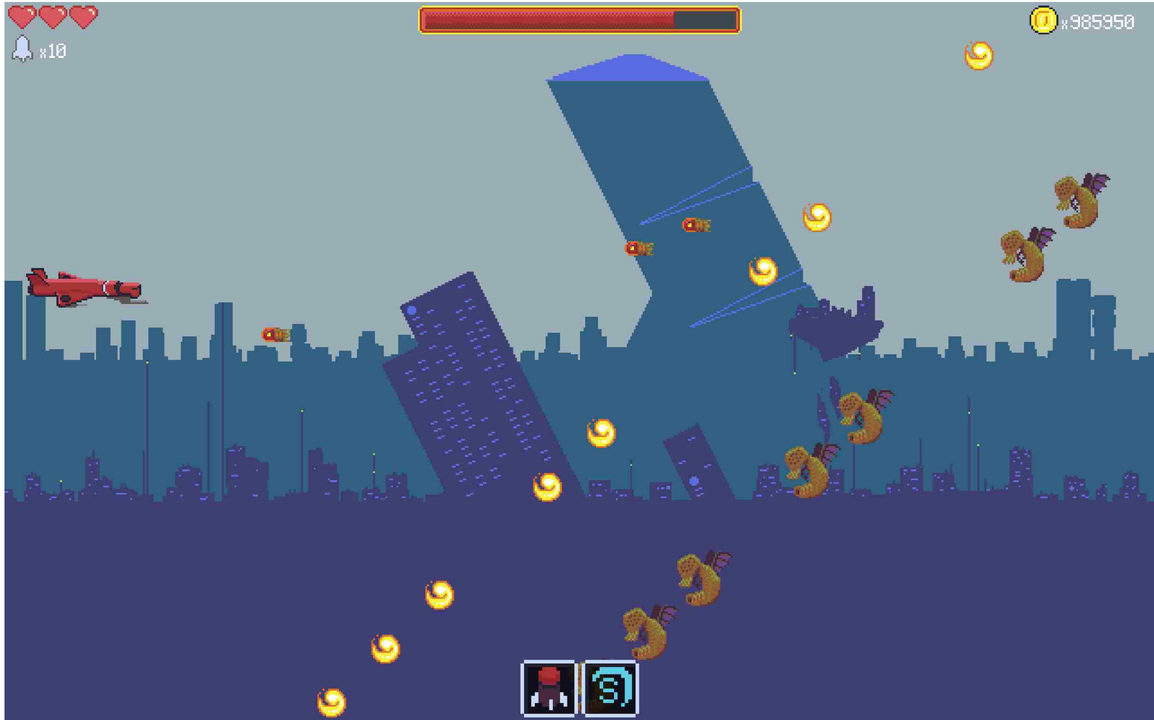
Figure 1.1: MVC Game Engine Architecture



Keep in mind that this is just an example of a simplified game engine. In practice, there are many more moving pieces and components to consider. Regardless of what components are present, a resource manager is always required. The resource manager is responsible for loading, saving, accessing, searching, and organizing all of the resources each component requires. Examples are given in figure 1.1 showing what kind of resources each component requires. With engine components and resources in mind, consider figure 1.2 containing a screenshot from a game called Bulwark. Bulwark was developed over a single weekend for the University of Kentucky's 2018 Engineering Day.

In this one screenshot, there are many different resources. The first type of resource is the images. This type includes the background, player, enemies, projectiles, user interface, and fonts. The second type is Java objects; they keep track of each

Figure 1.2: Screenshot of Bulwark



entity's properties such as position, health, and other values. The next type, which isn't visible, is audio. Examples include the techno background music and shooting sound effects. In addition to the types mentioned, there are many others currently being managed by the resource manager in this one frame. This screenshot illustrates the sheer variability in the type of resources which the resource manager has to manage.

In a more general sense, a resource manager is a data structure which supports operations such as insert, remove, update, and iteration. It also must be able to hold any object and be able to operate on each of them as if they were all the same. There are many database implementations which meet the requirements. Three existing solutions are presented to establish a foundation for further discussion. These three include a trivial solution, a complex but optimal solution, and a pre-packaged solution.

Array Backed

The first solution is to use a simple one-dimensional array. Due to its simplicity, this is a trivial solution. An implementation maintains one dynamically sized array. Insertion can be supported by just placing a pointer to the new resource into the first empty spot in the array. Removal can be supported by setting the pointer in the array to null. Keep in mind that both of these operations could result in the size of the array changing. The process of resizing the array is rarely trivial and typically involves copying the entire array into a different location in memory with more space. The other operations such as update, search, and iteration could be

supported by iterating over the array. No resizing would be needed. Overall, this results in every operation in an array backed resource manager being linear relative to the number of resources.

An important note here is that there is only one array holding all of the resources. Instead, multiple arrays could be used as long as determining the appropriate array is done quickly. This modification speeds up the solution, especially if resources were grouped into sets that are always queried together. Therefore, currently the best performance attainable for iteration over a subset of resources to be linear relative to the size of the subset.

Despite this solution being the trivial solution, a lot can be said about its performance and applicability. Further improvements focus on the insert, remove, and update operations.

Table 1.1: Complexity of array backed resource manager

	insert	**remove**	**update**	**search**	**iteration**
array	O(n)	O(n)	O(n)	O(n)	O(n)

Map Backed

The second solution is to use multiple maps, one for each group. The most appropriate map variant for this problem would be the array backed hash map. This decision was made to ensure minimum time complexity for iteration; storing the pointers next to each other in memory reduces the number of cache misses ultimately speeding up the algorithm. Hash maps are useful because they store key-value pairs and maintain constant time insertion, removal, search, and update. In addition, hash maps support linear time iteration, like the array; giving the hash map solution all of the benefits of the array solution but with the added benefit of maintaining constant time for other operations.

Despite this solution having the minimum time complexity, it comes with a large set of cons. From a development standpoint, maintaining a large set of hash maps is incredibly difficult. Consider the use case where the developer wants to save the state of the resource manager. This use case requires all of the separate hash maps first to be locked and then serialized to a file. Deserialization is equally as challenging. Especially compared to the next solution. It also doesn't scale well as the number of selection groups increases. Each resource can only be stored in one map, meaning that each resource can only be selected one way.

Table 1.2: Complexity of map backed resource manager

	insert	**remove**	**update**	**search**	**iteration**
map	O(1)	O(1)	O(1)	O(1)	O(n)

Database Backed

The third solution is to use a database. It completely abstracts the difficulty of maintaining some data structure to whoever implemented the database. It also has the bonus of being an out of the box solution. However, as you would expect, "there is no such thing as a free lunch." Databases come with many drawbacks when used in a game engine. The database implementation bloats the end product's file size. After all, the game engine can't assume that the player has internet access to an external database. In addition, it adds another layer of abstraction that the creative team has to work around when modifying in-game assets. Also from a debugging perspective, manually editing the save file can be difficult and dangerous depending on how the database is stored. It also requires the programmers to use a transaction query every time they want to access a resource; otherwise, a middleware would have to be added that does this.

Databases do present an interesting idea though. They store data with tree-based data structures. The most common structure used is the B-tree. The definition is any tree with a constant height, where given some k every node except the root and leaves have at least $k + 1$ children, and that each node has at most $2k + 1$ children. This definition requires that the B-tree has minimum height given the number of tuples. An important note is that in a tree the complexity of the operations is determined by the height of the tree. Therefore by minimizing the height of the tree, the total number of nodes that need to be loaded from memory and looked at is minimized. With these requirements iteration through the B-tree would be in linear logarithmic time, with respect to the number of nodes. [1]

The reason that B-Trees are commonly used is that these nodes can be stored in pages and only loaded when needed. By having control over the size of each node it allows for the implementation to load as little as necessary from disk when performing searches. Ideally, the entire database would be loaded into memory so that searches could be run quickly across any field. However, due to the volume of elements contained in a database, this is not always practical. To account for the volume of tuples, databases index a specific field constructing a smaller data structure – so-called indexing – that can be stored in memory or at least mostly in memory.

Consequently, any query involving an unindexed field is commonly very slow due to the necessary IO operations. The indexing structure has taken many forms since the database's introduction. One notable option is a simple hash map. The benefit of using a hashmap is that the time complexity to add and remove a tuple is constant on average. Nevertheless, most databases use trees, even though they are slower.

In contrast to hash maps, trees have logarithmic complexity. There are reasons behind using a tree: trees allow for ranges of elements to be quickly found in logarithmic time (a hash map's time complexity for this operation would be linear); scaling the size of trees is relatively efficient. Also, hash maps are only really applicable when the entire structure can be stored in memory; this is not always possible as the database grows. Trees have no concept of a fixed capacity, so they are never required to resize in the same way an array-backed structure would. In comparison, hash maps would have to resize, requiring the entire collection to be reordered; for large index

tables, this would require a massive amount of IO. Hash maps are typically used in smaller databases that can be stored in memory.

As memory limits increased a modification was made to the B-tree, called the B+tree, allowing for linear time iteration (previously linear logarithmic). In B+trees neighboring leaf nodes are linked together. With this information, a leaf node finds the next leaf node in constant time. In comparison, B-trees find the next node in logarithmic time. [3] With this modification, the time complexity for adding and removing tuples from a tree remains logarithmic while the complexity for iteration is linear. This modification makes the database solution more competitive when compared to the previous solutions.

Table 1.3: Complexity of tree backed resource manager

	insert	**remove**	**update**	**search**	**iteration**
b-tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(nlog(n))
b+tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

1.3 Problem

Any of the presented data structures can be used to implement a resource manager but each fall short in some way. While it is unlikely that a silver bullet exists, this thesis focuses on addressing as many of these issues as possible. In summary, there are four major limitations that the resource manager must address.

It must be able to operate on large sets of externally developed resources. Specifically, the creative team members for a game are the ones primarily creating and modifying any resources that the player might interact with. Meaning that the creative team should be able to modify any of the existing resources without having to compile or do anything other than save the file in the correct location.

The resource manager should be easily serialized and deserialized. This requirement means that all resources should be stored in one large data structure; simplifying the saving and loading process to be writing and reading a single file into one structure. This simplification not only reduces load times but also results in an easily maintainable code base.

The resource manager should be designed to run on a single machine and be as lightweight as possible. The majority of the game loop must be spent on rendering so the time spent on accessing and iterating through resources should be minimized. In other words, the time complexity of the resource manager for these operations should be as minimal as possible.

The resource manager should have the ability to store resources in a human-readable format. This requirement is an added bonus but is extremely useful for smaller development companies that don't have the resources to maintain various editors for each file type that occurs in the game.

The subject of this thesis is choosing and designing a data structure, which meets all of the listed limitations. However, not all of the listed limitation is directly addressed by the data structure. With this in mind, a reduced list of limitations is

presented which applies to the data structure. The data structure should be able to store all resources in a single structure and match the time complexity of a hash map.

1.4 Layout of Thesis

The thesis starts by taking a closer look at all the different resources game engines have to support and coming up with a model that accurately represents these groups. Then an identifier schema is derived from the resource model. A pattern schema is carefully crafted around the identifier schema to be as intuitive and minimalistic as possible. Examples are given to emphasize the usefulness and applicability of each schema. Limited by the previously formalized schemas, a data structure is presented which supports the necessary operations and builds off of the existing solutions. Pseudocode, an informal complexity discussion, and complexity comparisons are given for each algorithm on the data structure. Finally, use cases for the solution are given to further the potential of the solution and to show how it applies in practice.

An example implementation of the presented resource manager is present in the appendix. The appendix also includes various test cases demonstrating and testing the various algorithms. In addition, the test cases resulted in the implementation having a 95% code coverage. The remaining 5% is mostly print functions and sanity checks which were used for debugging purposes.

Copyright© Ethan Toney, 2018.

Chapter 2 Discussion

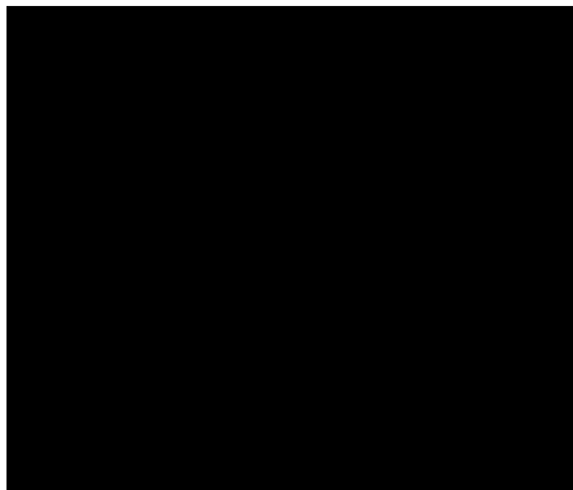
In the introduction, three existing solutions for creating a game engine's resource manager were presented. This chapter focuses on the identifier schema, pattern matching schema, and taking the tree-based solution and building upon it.

2.1 Modeling Game Engine Resources

As mentioned before, each component of the game engine maintains it's own independent set of resources. However, one important note is that these resources are coupled together. For example, the world component maintains a list of entities that are currently in the world. These entities have graphical representations which are called sprites. Then the sprite has a set of animations that are all layered on top of each other. Finally, each of these independent animations has images associated with them. Meaning that a single world entity has a chain of four linked resources to represent it. If the world entity were to be removed from the resource manager, then the linked resources should also be removed if that entity was the only reference pointing to them. This restriction adds a whole new level of complexity that must be considered, but first, a model of the resources is established.

The first observation made is that all of the resources can be modeled as an ontology, a set of categories that show specific properties of an entity and their relation. In other words, if the graphics component manages a resource, then the resource's top-level ontological category is the graphics component. To better explain this concept, consider figure 2.1 showing a few of the ontological categories present in a game engine.

Figure 2.1: Example of game engine ontology



Looking at the figure, the specific region 'lexington' can be found. All of the ontological categories above it describe a specific property of Lexington. Specifically

that it is located in the state of Kentucky, which is a region, and that regions are managed by the world component of the game engine. Notice how the ontological properties describe more than just the bottom level resource; it describes all resources in that category. This description is useful when it comes to grouping resources by some common property.

With this ontology described, the next observation is that the concatenation of each level of the ontological relationship produces a unique identifier. Using Lexington as an example, the identifier would start with some game engine prefix to signify that the string represents a resource; this gives us the working identifier of 'vid' (viduus identifier). Then the next category is appended. At this point, a delimiter must be used to mark the boundaries between ontological categories; a colon is used as the delimiter. However, the delimiter could be anything. After this concatenation, the working identifier becomes 'vid:world'. Repeating this process results in the final identifier being 'vid:world:region:kentucky:lexington'.

Formally, the identifier schema is the concatenation of each ontological category of a resource delineated by a colon. This identifier schema was deliberately designed to work efficiently with the following proposed data structure, but there are many other identifier schemas. Some include hungarian notation where the type of the underlying resource is distinguished by a predefined prefixed. There is also camel case where the first letter in every word present in an identifier is capitalized other than the first word. Similarly, there is snake case where an underscore separates words in an identifier. So far all of these identifiers have been fairly different from the ones proposed. However, a major inspiration for this identifier was Amazon Web Service's, AWS, identifiers. In AWS's identifier schema each section of the identifier is separated by colons. An example being 'arn:aws:iam::123456789012:user/David'. Despite the two being similar AWS employs a more complicated naming schema allowing for specific categories in the identifier being blank. In addition, specific categories are always present at the beginning of the identifier as opposed to the presented identifier schema. Also, each ontological category is not necessarily more specific than its parent.

A few examples of identifiers using the proposed vid identifier schema are listed below.

- `vid:world:actor:zombie.BasicZombie`
- `vid:world:player:ethan.BasicPlayer`
- `vid:graphics:animation:base_humanoid:walk:left`
- `vid:audio:music:main_menu`
- `vid:graphics:image:backgrounds/tile_set1`

Notice how even though the last identifier includes a forward slash, it is still valid. Given the rules for identifiers, it gives each ontological category the freedom to use whatever relatively unique identifier schema it wants, even if that schema is just a

file path. This freedom allows the developer to design identifiers which are easy to understand. Another benefit is that given an identifier the resource manager should be able to find and resolve the underlying data source. This use case is shown in the first and second examples above. In these examples, the specific class type of zombie and player are given even though the identifier up to that point is unique. This freedom allows the system to dynamically resolve the underlying objects.

2.2 Selecting Resources

Selecting specific groups of resources has already been presented as a necessary operation. However, deriving an effective way to query for these groups is a nontrivial problem. Here is where the design of the identifier schema begins to become clear. Consider a specific resource's identifier, say for example 'vid:world:region:lexington'. If this string by itself represents a single resource then consider the following pattern, 'vid:world:region:*'. The '*' character should be interpreted as a character that could represent any number of characters, even none. By performing pattern matching, this identifier would match any resource in the 'region' ontological category.

A few more examples. 'vid:*' would match all resources contained in the resource manager. 'vid:*:*:*' would match all resource at least two ontological categories below 'vid'. 'vid:*:player' would match every single resource with 'player' as the last relatively unique identifier. '*' would behave the same as 'vid:*' since all resources start with 'vid'.

This wildcard character solves the issue of trying to select a single group of resources, but it would be nice if the pattern syntax could resolve resource relationships. As previously mentioned, resources can be linked to each other. To support this, a unique delimiter is used to separate layers in a pattern. For this pattern matching syntax, the '—' delimiter is used. Let's say we want to select all resources which are directly related to the Lexington region. First Lexington can be selected with 'vid:world:region:lexington'. From this point, all resources can be selected with a simple '*' pattern. After combining the two pieces, the pattern becomes '*—vid:world:region:lexington', which selects all resources directly attached to the region. The pattern could easily be modified to select all players inside of Lexington vid:world:player:*—'vid:world:region:lexington'. Or to select all enemies in any region 'vid:world:enemy:*—vid:world:region:*'.

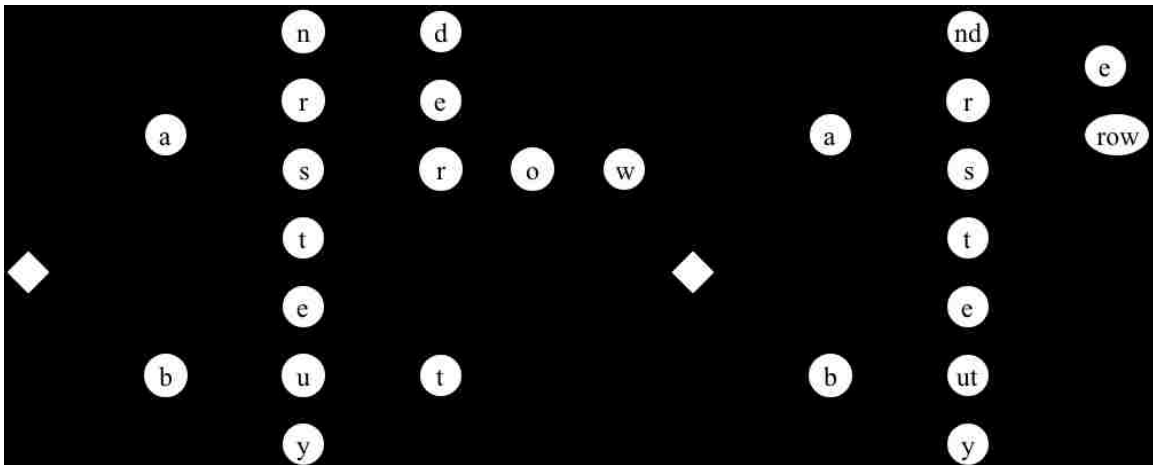
2.3 Data Structure Selection

Trees are incredibly useful for storing related information like this, but the previously discussed B+tree will not be used. The problem of resolving elements with the outlined identifier schema can be broken down into a prefix matching problem. Luckily, the problem of prefix matching has been a focus of previous research. The data structure which is the focus of this thesis is the compressed prefix tree. This data structure builds off of the more basic prefix tree. A prefix tree, commonly called a trie, is a tree where each node in the tree corresponds to a single character in a

string. Each intermediate node has a map of characters to children nodes; more basic implementations use a statically sized array instead of a map. Since each node corresponds to a single character, all elements contained in a trie are strings. The usefulness of this structure is that given any node all strings located under that node share a common prefix. The prefix is the concatenation of all the ancestors of this node with the node's character. However, since each node only represents one character the space complexity of this tree is poor. In addition, the height of the tree is equal to the length of the longest contained string. [4]

The prefix metadata, intrinsic to the prefix tree, is the foundation for the optimizations developed in this thesis. We use the compressed prefix tree which includes two addition rules: no node without a value may have exactly one child; nodes correspond to a string of characters as opposed to a single character. These rules address space complexity concerns. [5] Figure 2.2 shows the structural differences these rules make to the prefix tree.

Figure 2.2: Trie (left) and compressed prefix tree (right) containing words: a, and, are, arrow, as, at, be, but, by



By merging chains of nodes containing only one child, or one forward edge, the total number of nodes is reduced. This reduction not only decreased the space complexity but also decreased computational complexity.

Concerns

Using a compressed prefix tree does come with its own set of concerns. First off tree compression does not always improve computational complexity. Even though operations such as insert and remove retain the same logarithmic complexity, since the height is always larger these operations are slower than the equivalent B-tree. A special case where the height of the prefix tree cannot be compressed into a smaller tree is shown in Figure 2.3.

In this example, the height of the tree is linear relative to the number of contained nodes; which results in linear time complexity for the add and remove operations. More formally, linear height growth occurs in any prefix tree which has n strings that

Figure 2.3: Trie (left) and compressed prefix tree (right) containing words: a, aa, aaa, aaaa, aaaaa

all grow in length while containing the previous string as a prefix. In this situation, the total number of nodes is n. Although this being the best case for space complexity, the height of the tree is also n resulting in the worst case for computational complexity. This situation is important but can easily be avoided and therefore is ignored in this thesis.

Another concern of using a compressed prefix tree is that the number of required character comparisons remains constant; despite the number of nodes decreasing. Each node now contains a string of characters, and every character in the string must still be looked at during a tree decent. Despite this, the decreased number of nodes is still beneficial since the total number of necessary disk reads decreases. In addition, since this prefix would have to be validated eventually, validating all resource's prefixes at once speeds up the selection process.

Another property of a compressed prefix tree that could be a concern is the fact that it is not balanced. The term balanced does not refer to the height of the tree being constant. Due to the differing densities of the ontological categories, each category may contain a very different number of resources. This difference means that the computational complexity associated with one category is potentially vastly different from a category differing in size. From a pure computational perspective this isn't ideal; when considering the best alternative, hash maps, it should be observed that the complexity remains the same. With hash maps, the complexity is relative to the size of the ontological groups, and that remains the same for the compressed prefix tree.

Another important property to consider is that when compared to an equivalent B-tree a compressed prefix tree typically contains more nodes, thereby running slower. However, as long as the number of nodes grows linearly relative to the number of resources, this can be ignored.

The average height of a compressed prefix tree is analyzed. First, consider a prefix tree containing n strings all with the same length. In this case, every stored string is at a leaf node; giving us n leaf nodes. In addition, given the compression rules, every non-leaf node has at least two children nodes. Looking at the tree level by level, the first contains exactly one node which is the root node. The second level has at least two nodes. The third has at least four. This doubling pattern is repeated until a level with exactly n nodes is found. Mathematically this gives us the total number of nodes, m, to be $1 + 2 + 4 + 8 + 16 + ... + 2^{h-1} + 2^h = m \Rightarrow m = 2^{h+1} - 1$ and $n = 2^h$, h is the height of the tree, and n is the number of strings. The first equation $m = 2^{h+1} - 1$ can be simplified to $h = O(log(m))$. Then plug m back into the equation $h = O(log(2^{h+1} - 1)) = O(log(2^{h+1})) = O(log(2 * 2^h)) = O(log(2n))$. Therefore the total number of nodes in the tree is bounded by $O(log(2n))$ when all strings are the same length. This case is one of the best cases and is what the developer should

strive for when designing an identifier schema.

With an identifier schema like this, the logarithmic complexity for basic operations is maintained. However, this does not mean that the compressed prefix tree is as fast as a B-tree. The height and size of a B-tree are almost always less than that of the equivalent compressed prefix tree; resulting in lower average case time complexity for operations. In practice, this fact should be kept in mind when deciding to use a compressed prefix tree.

Despite this, the complexity of a compressed prefix tree has been shown to be competitive to B-trees. Due to this, a few commercial database implementations allow the developer to index a string-based field using compressed prefix trees. In addition, prior research has compared the two. Finding that the actual complexity is very similar and that each perform better in individual situations. [2]

Comparison to Databases

Much has been said about databases and their applicability to resource management. Their applicability stems from the fact that they efficiently provide an out of the box solution to resource management, albeit with a few negatives. Given the identifier schema and data structure selection, both will be compared to a database. This aids in comprehension, and better illustrates the ideas behind the algorithms that are later presented. In addition, it shows how the techniques presented could also be implemented in a database system using the presented data structure.

Formally a database is a structure which holds a set of tuples; they facilitate inserting, deleting, selecting, and updating any of these tuples. Each tuple contains values indexed with a column name, or field name. If two tuples share a common field name, then the data pointed to by that field name is of same structure or type. Another essential characteristic of a database is that all operations on a database are accessible through a general-purpose query language. This general purpose query language is evaluated by a query optimizer and turned into an execution plan. This software is typically very complex and can lead to drastic performance differences between database systems. A good comparison to a query optimizer would be a compiler. Compilers do complex logic analysis in order to remove unnecessary portions of code. The same can be said for query optimizers.

Consequently, the presented pattern matching schema can be thought of as a general-purpose query language. Therefore comparing the presented schema and a database's query language will better illustrate the differences in using the two. However, as mentioned a database could technically use the presented data structure, identifier schema, and even pattern matching schema.

As previously mentioned, databases are commonly implemented with some form of a tree structure, the most common being the b-tree. When using a tree the complexity of single value selection, insertion, deletion, and update is $O(log(n))$ due to how each operation can be implemented by walking down the underlying tree. Only one operation presents a problem; this operation is multi-value selection. The operation could also be called filtering or selecting. For now, the operation is called

selection. Selection operations are broken into three categories, each more complex than the last.

The first of which is a selection on a single index. A selection in this category returns exactly one tuple. This category is typically implemented by just walking down the tree following the passed index until a leaf node is reached. Then the value at this leaf node is the resulting tuple. Since all we did was walk the tree the complexity for this category is $O(log(n))$.

The second category is a selection with an index based conditional. A conditional in this context is some test that takes a tuple and returns a boolean value of true or false. An example of a conditional would be, all tuples which have an index greater than the number 10. A selection in this category returns the set of tuples where the conditional evaluated to true. Luckily since the conditional is based on the indexed field, the b-tree can be used to gather a range of candidate matches. While this does not give logarithmic complexity like the previous category, it does give us a complexity of $O(n)$ where n is the number of candidate tuples.

The third category is a selection with a non-index based conditional. This category is the worst case for a database, or a tree, since the underlying tree provides no information which could be used to find candidate tuple matches. Since there is no information to work with the database must do a full table scan comparing each tuple against the given conditional. While a full scan across the database does result in a similar complexity as the second category of selects, $O(n)$, n in this case, is the total number of tuples in the database. As mentioned before the IO involved with loading each tuple into memory slows down any solution. Selections in this category should be avoided when possible.

Another important note is that selection operations can commonly be chained together. The behavior of this chaining behaves similarly to the layers concept in the pattern matching schema. Implementations for selection operations vary drastically from one database implementation to another; since it is up to the query optimizer to create an execution plan.

Examples

Now for some examples. These examples illustrate the different types of selection operations while basing the discussion in existing concepts. A PSQL-like, PostgreSQL, syntax is used to show how someone could write a query to accomplish the example. A discussion about how a database, or resource manager, would execute the query is given.

Every example works with a version of the presented pattern matching schema, a simplified version of the presented identifier schema, and the presented data structure. This simplified identifier schema consists of identifiers which contain precisely one ontological category followed by a relatively unique identifier. An example of an identifier following this format would be 'dog:sandy'. In this situation, the category name is 'dog', and the relatively unique identifier is 'sandy'. Any number of categories can be used as long as they are all unique. For example, another identifier following this identifier schema could be 'cat:cacey'.

Example 1

The most basic example is to select all elements in a specific ontological category, let's say the dog category. The program executes this query by walking down the tree, following a pattern that matches only the dog ontological category. The following query implements this example, (A PSQL-like query structure is used) `'SELECT * FROM resources WHERE id LIKE 'dog:%''`. The equivalent query using the pattern matching syntax would be, `'dog:*'`.

To execute this query on the modified compressed prefix tree, the program starts by walking from the root node to the 'dog:' sub-node (assuming that other ontological categories exist). In a more general sense the program walks down the tree starting at the root node until it matches the entire pattern, 'dog:' in this example. From this point, the lexicographically least edge can be followed until a leaf node under 'dog:' is found. This process is then repeated to find the lexicographically last leaf node. Leaf nodes are linked to their neighboring nodes with values. An interesting observation is that the previous neighbor is the largest leaf node that is still less than the current node. In an opposite sense, the next neighbor is the smallest leaf node that is still larger than this node. Due to the neighbor relations, there is a path from the lexicographically least and most nodes under the 'dog:' node. Therefore by walking from one to the other, all resources in the dog ontological category are be found.

Example 2

The next, more complicated example, is selecting all elements in a specific ontological category that meet some conditional. In this case, the problem is selecting all dogs who like to bark. Assume that the property of whether or not a dog likes barking is stored as a flag in the tuple under the field name 'likes_barking'. The following query implements this example, `'SELECT * FROM resources WHERE id LIKE 'dog:%' AND likes_barking = TRUE'`.

This example cannot be done using only the present pattern matching schema. The syntax is modified to include conditional statements by including an optional conditional in parenthesis at the end of each layer's pattern. With this modification, a query which implements this example is `'dog:*(likes_barking=TRUE)'`.

The execution of this query would begin, as before, by walking down the tree starting at the root node. Then the left and right bounds on the dog ontology would be found. The set of nodes between the left and right bounds, including the endpoints, is called the candidate set. The program iterates through the candidate set checking each value against the conditional statement. Elements in the candidate set where the conditional is true are added to a result set. Once iteration is complete, the result set is the answer. In comparison to the previous example, this example is very similar; the only difference being the additional conditional check on the candidate set.

Example 3

Example three is selecting an element based off of how it has a link to another element. This concept of linking was introduced earlier, but it is the process of creating a cross edge between two leaf nodes in the resource tree. In another sense, it represents a coupling between the two elements in some way. For this example, it is to select the owner of the dog sandy (owners are represented through cross edges). This problem boils down to selecting the element in one ontological category which has a cross edge into another ontological category. Using a PSQL like syntax this query would be `'SELECT * FROM resources WHERE id LIKE 'person:%' AND (SELECT id FROM table WHERE id = 'dog:sandy') = ANY(cross_edges)'`. The equivalent syntax using the presented pattern matching syntax is `'person:*|dog:sandy'`.

This query is executed by evaluating both layers in the second query, using the pattern matching syntax which can be accomplished by using the logic discusses in the previous two examples. Once complete, the program has two sets of elements. The first set contains all elements in the person ontology and the second set contains only 'dog:sandy'. From this point on the person set can be filtered by iterating through each element removing those which do not have cross edges to the dog set containing only 'dog:sandy'. Once this filtering is complete, the person set is the result set.

Example 4

The next example is selecting all elements in an ontological category that have a cross edge to a specific ontological category. The problem is to select all people who have a cat. In PSQL the query implementing this is `'SELECT * FROM resources WHERE id LIKE 'person:%' AND (SELECT id FROM resources WHERE id LIKES 'cat:%') = ANY(cross_edges)'`. The equivalent pattern matching query is `'person:*|cat:*'`.

The logic for this execution is the same as example 3. First, each layer's result set is computed. Then each layer is filtered from right to left ensuring that cross edges exist. This process applies to examples that include even more than two layers. There is no specific example of this, but it is possible.

Example 5

Example five is selecting elements based off of prefixes and postfixes, relative to the wildcard character. In this example, all dogs whose name ends with a 'y' are selected. This example is necessary to show how the tree is not able to speed up all operations. Specifically, it is only useful for resolving prefix patterns. This limitation is largely desirable due to the layout of the identifier schema, but there are many use cases for patterns which include postfix patterns. Also being able to handle patterns that include multiple wildcard characters is also useful. That being said, the PSQL query supporting this is `'SELECT * FROM resources WHERE id LIKE 'dog:%y''`. The equivalent pattern matching query is `'dog:*y'`.

Executing this pattern on a compressed prefix tree is viewed as a two-part process. First, the prefix pattern, 'dog:*', (which we already know how to handle) should

resolve to a candidate set. Next, the candidate set is iterated, and each element is run against the remaining pattern '*y'. Note how the wildcard appears in both patterns; this is valid because the candidate set only includes elements which met the first pattern. Once the candidate set is filtered by the remaining portion of the pattern, it can be returned as the result set.

While no real optimization occurs in the evaluation of the post wildcard pattern, keep in mind that the number of elements that needed to have the pattern matching algorithm run on was largely reduced.

Copyright© Ethan Toney, 2018.

The examples in the previous chapter discuss implementations for the select operation on a compressed prefix tree. This chapter takes the discussion and presents pseudocode and simple complexity analysis for every necessary operation. The chapter also compares the found complexity of each algorithm to the complexity of a hash map based solution.

Comparison with a hash map is necessary to show how a compressed prefix tree based solution compares to the best case existing solution.

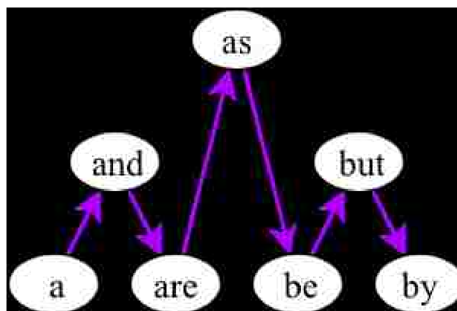
The presented operations are insert, delete, and select. Operations such as update and get are supported through the select operation. Another operation not getting an explicit implementation is iteration. It is supported by first running a select operation to get a set of resources. Then the set of resources can trivially be iterated.

Additional operations were added to improve clarity; they act as helper functions. To list a few: prefix filter, regular filter, search left, and search right. Each major operation describes helper functions as they are required.

Despite the similarities between compressed prefix trees and other trees, there are many substantial differences which make supporting some of the outlined requirements non-trivial. The most notable complication being, maintaining nearest neighbor links between nodes containing values. The difficulty stems from the differences in ordering between the two structures. In most trees, inorder tree traversals are used to order nodes.

Tree traversal is the process of walking through all of the nodes in a tree. They always start with the root node and move throughout the rest of the tree. The type of tree traversal determines the order that nodes are visited. Inorder tree traversals visit the left nodes, itself, and right nodes in that order. Left nodes are the children who have values less than their parent. Right nodes are the children who have values greater than their parent. Figure 3.1 illustrates an inorder traversal on a binary search tree.

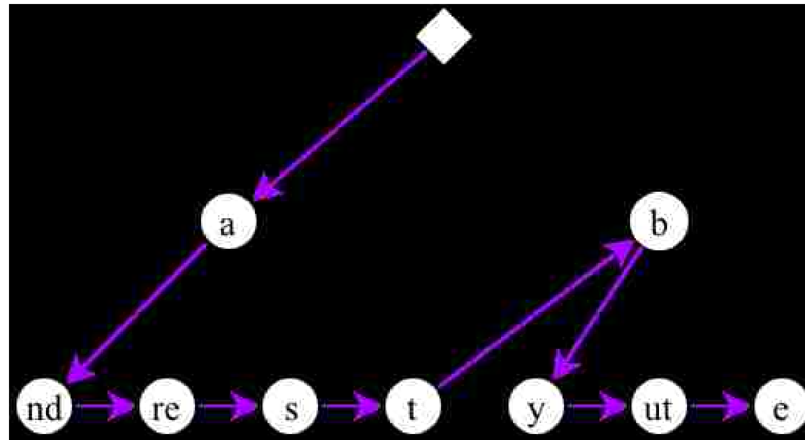
Figure 3.1: Tree containing: a, and, are, as, at, be, but, by. Purple line shows inorder traversal path



Unlike most trees, which allow nodes to contain children with values less than and larger than the parent's value, compressed prefix trees only contain children who are

lexicographically greater. While technically an inorder tree traversal still works on a compressed prefix tree, since all children nodes are right nodes, it is more appropriate to use a preorder tree traversal. In comparison to an inorder traversal, a preorder traversal visits all children after visiting itself. Figure 3.2 illustrates a preorder or inorder traversal on a compressed prefix tree.

Figure 3.2: Compressed prefix tree containing: a, and, are, as, at, be, but, by. Purple line shows inorder traversal path



The ordering problem is vital to maintaining neighbor relationships between value containing nodes. Preorder and inorder traversal could accomplish the task; however, performing a tree traversal is computationally costly. To avoid performing a tree traversal rules are given that allow the algorithm to maintain these relationships efficiently.

3.1 Insert

The first algorithm to be specified is the insert operation. Starting with this algorithm is essential since it describes many of the fundamental concepts for the following algorithms.

There are many properties of the compressed prefix tree for which the insert operation has to maintain; one of those properties is neighboring relationships. The B+tree inspired this modification since it maintains links to neighboring nodes itself. However, due to their differences, a compressed prefix tree only wants to link neighboring nodes with values. A compressed prefix tree frequently has many nodes which do not contain values but are present for grouping purposes. This modification allows for iteration over a range of elements to occur in $O(n)$ time. Previously iteration would have taken $O(nlog(n))$ time. One way to maintain this relationship is to perform insert as usual and then update relations with a preorder tree traversal. While this would work, the complexity of iterating over every single node in the tree is asymptotically more significant than the complexity of insertion by itself. To match the complexity of insertion only tree descents and ascents may be used.

One straightforward solution which meets the criterion works by finding the previous and next nodes once the insert operation inserts the new node. The algorithm finds the previous neighbor by walking up the three and either encountering a node with a value or encountering a forward edge lexicographically less than the last traversed edge. In the first case where the algorithm finds a node with a value, the found node is the previous neighbor. However, in the second case, when the algorithm encounters a lexicographically less edge, it performs a tree descent starting with that edge. The descent works by following the lexicographically last edge on every step. Once at the bottom of the tree, the found node is the previous neighbor. In the worst case, the algorithm does two subsequent tree walks resulting in a complexity of $2log(n)$.

Next is finding the next node. The algorithm does this by searching the inserted node's children for the lexicographically least child. The search walks down the remaining portion of the tree following the lexicographically smallest edge. If the search does not find a child, then the tree must be ascended. The next node pointer is updated to point to the first node lexicographically following a traversed edge. The node pointed to by this pointer is descended, following the leftmost edge, until a node with a value is found. In the worst case, the tree is ascended and then descended again. Resulting in a total of $2log(n)$ nodes visited. Taking into account the complexity from finding the previous node the total complexity becomes $4log(n)$. This algorithm is not horrible but is worst than keeping track of nodes while descending the tree.

To keep track of neighbor nodes while descending the tree, a simple set of rules are used. The previous node pointer is assigned when a node with a value is found during descent. If the current node does not have a value but has more than one child, then the previous pointer is assigned to the first node lexicographically less than the node about to be traversed. One exception to the rules is that the destination node is ignored when computing the previous node pointer.

Once at the desired node the previous node pointer is resolved. If the pointer was assigned due to an intermediate node having a value, then the previous node pointer is the previous node. If it was assigned due to the intermediate node not having a value, then the previous node is the rightmost child of the node pointed to by the previous node pointer. In the worst case, the algorithm performs exactly one additional tree descent; this results in the worst case complexity being $log(n)$. Pseudocode for both the left and right descent is shown below.

Algorithm 1 node.search_left()

```
if this node has a value then
    return this
end if
if this node has no forward edges then
    return this
end if
get lexicographically first child, first
return result from recursive search_left call on first
```

Algorithm 2 node.search_right()

```
if this node has no forward edges then
    return this
end if
get lexicographically last child, last
return result from recursive search_right call on last
```

The process for computing the next node is much simpler than that of the previous node. As the insert operation descents the tree, the next node pointer is set to the lexicographically next edge. Once at the result node, if that node has children, then the next node is the leftmost child. If this is the case, then the algorithm performs a tree descent following the lexicographically least edge. Once at a node with a value, that node is the next neighbor. Otherwise, the portion of the tree below the next node pointer, which was assigned due to a lexicographically larger edge during traversal, must be descended following the leftmost edge. Once a node with a value is found it is the next neighbor. The algorithm at most performs one additional tree descent; this results in a worst case complexity of $O(log(n))$.

By adding together the complexity for computing both neighbors, the complexity becomes $2log(n)$. If the complexity for the insert operation is taken into account, the total complexity becomes $3log(n)$. This complexity is asymptotically similar to the complexity of a regular insert operation on a tree, thus meeting the complexity requirement. Below is pseudocode for an insert operation using the described behavior.

Algorithm 3 insert(obj)

```
get the object's identifier
run get_node() on the root node with the found identifier
if a left node was found then
    set this node's previous neighbor equal to the left node
end if
if a right node was found then
    set this node's next neighbor equal to the right node
end if
set the value of the center node to be the passed object
```

The helper function `get_node()` performed the primary portion of the tree descent while maintaining pointers to the previous and next node. The helper function is also responsible for finding the exact match of the given pattern. In addition, the helper function is responsible for maintaining all properties of a compressed prefix tree; since the tree is regularly modified during insertion. The pseudocode presents a recursive algorithm and is broken into subparts for explanation purposes. Below is pseudocode for this algorithm.

Algorithm 4 node.get_node(i, identifier, result)

```
1: if this node can become the identifier then
2:     set the key equal to the identifier after index i
3:     update result when answer
4:     return result
5: end if
```

The first section of pseudocode shows the base case of the recursion. This case applies whenever a node is new, or before the element has any child nodes. It sets the key of the current node to be equal to the remaining portion of the identifier and returns. As a definition, a node's key is the portion of an object's identifier that the node matches; it is not necessarily the complete identifier up to this point in the tree.

```
6: find the first difference between the key and identifier
7: if search did not make it through the key then
8:     create a copy of this node, child_node
9:     set child_node's key equal to the unmatched portion of the key
10:    reset this node's state
11:    add child_node as a forward edge
12:    if child_node has a next neighbor node then
13:        update it's next neighbor node to point to it
14:    end if
15:    for each node in child_node's forward edges do
16:        update their back edge to point to child_node
17:    end for
18:    if the identifier ended partway through the key then  ▷ this means that this
   node will get the new value
19:        set the next node to be child_node
20:        update this node's identifier
21:        update result when answer
22:        return result
23:    else                    ▷ when the identifier only matches part of the key
24:        if this node had a value then
25:            update previous node to point to child_node
26:        end if
27:        create a new forward node, forward_node
28:        set forward_node's key to be the unmatched portion of the
29:        identifier
30:        set this key to be the matched portion of the identifier
31:        update result when intermediate
32:        return the value of a recursive get call on forward_node
33:    end if
34: end if
```

This portion of pseudocode handles when the current node's key is not entirely matched; which only occurs when the current node must be split into two nodes. There are two distinct processes for splitting a node. The first process occurs when the identifier is completely matched. This situation happens when the current node should be assigned the new value, and the previous value should be a child of the current node. The new child node must maintain all of the relations that the current node had before the split. This first case is handled by the if statement on line 18.

The second process occurs when the identifier is partially matched. When this occurs, the current node is split into three nodes. The first node corresponds to the matched portion of the identifier. The second node corresponds to the unmatched portion of the current node's key; this node maintains all of the current node's relationships. The third node corresponds to the portion of the identifier that was not matched; this node is assigned the inserted value. This second case is handled by the else if statement on line 23 in the pseudocode. The reason this case is relatively large is due to the complexity of maintaining prior relationships.

```
35: if search did not make it through the identififer then
36:     get first unmatched character in identifier, edge_char
37:     if edge_char is not in the set of forward edges then
38:         create a new forward node
39:         add the new forward node as the edge_char edge
40:     end if
41:     get the forward node correspond to edge_char, forward_node
42:     update result when intermediate
43:     return the value of a recursive get call on forward_node
44: end if
45: update result when answer
46: return result
```

This pseudocode is the final portion of the *get_node*() algorithm; it is responsible for the case when this node's key was completely matched, but there is still more of the identifier to go through. When this occurs either the result is in the subtree below this node. The next edge is the first character in the unmatched identifier. If there does not exist a forward edge for this character, then it needs to be created and added. Then the forward edge can be followed.

It is important to note that the 'update result when intermediate' and 'update result when answer' lines correspond to the process for next and previous node selection discussed before. Consequentially the result is represented as a tuple of five values. The first value is the matched node. The next two values are the previous and next node pointers. Finally, the last two values in the tuple are booleans that represent whether or not the previous and next nodes are exact matches. These values are necessary since in both algorithms might find exact matches. Exact in this case meaning that an additional tree descent is not necessary. Below is pseudocode for each of these algorithms.

Algorithm 5 node.update_result_answer(result, node)

set result's center to be this node
if this node does not have any forward edges **then**
    set result's right to be the left most nested child node
end if

Algorithm 6 node.update_result_intermediate(result, edge)

if this node contains an edge before the passed edge, previous **then**
    set result's left to be previous
else if this node has a value **then**
    set result's left to be this node
end if
if this node contains an edge after the passed edge, next **then**
    set result's right to be next
end if

Once *get_node*() returns the result, the result must be resolved. If the exact match flags are not set, then an additional tree descent is run on both the left and right node pointers. In the case of the previous node, this means that the rightmost child must be found ignoring nodes with values. In the case of the next node, the leftmost child must be found stopping at the first node with a value. Once resolved, the previous and next neighbors must be linked to the center node, the result.

As stated above, resolving the previous and next neighbors each can be resolved in $log(n)$ time, meaning that the total complexity for the insert operation is $3log(n)$, as stated above.

3.2 Delete

The next operation is delete. Much like insert, delete descends through the tree to find a node which matches the given identifier. However, unlike in the insert algorithm, delete does not need to keep track of pointers to the previous and next nodes during descent. This consequence is because the matched node knows what its previous and next neighbors are. In addition, the algorithm that delete needs to use is a read-only algorithm. Both of these simplifications allow for a much simpler tree descent algorithm to be used.

Therefore, the delete algorithm first searches for the node to be removed. Once found, the neighbors of that node need to be connected as neighboring nodes. Now that the neighbor relationships on the removed node have been fixed, the value at the removed node can be set to nothing. In addition, any cross edges that the removed node has also need to be deleted. With all relationships removed the tree can finally be cleaned.

This cleaning process checks for any nodes that break the rules of a compressed prefix tree. The most important rule is that no node without a value has less than

two child nodes. Therefore, the tree can be ascended starting at the removed node. If the current node has a value, then the process is complete. Otherwise, if the node has one child, then the current node must be merged with the child node. If the node does not have any children then it must be removed from its parent node, then the clean operation must be run of the parent node, since it may only have one child now. The pseudocode for the delete algorithm is shown below.

Algorithm 7 delete(*obj*)

```
get the object's identifier
run find_node() on the root node with the computed identifier
if no node was found, or the found node does not have a value then
    return
end if
remove the value from the found node
run a clean_node() operation on the found node
if a left and right neighboring node were found then
    link the left and right node together as neighbors
else if a left neighbor was found then
    remove the left node's next neighbor
else if a right neighbor was found then
    remove the right node's previous neighbor
end if
```

This algorithm relies on two helper functions. The first being $find_node()$. This algorithm is responsible for performing the read-only search of the tree. Keep in mind that this search has the potential to return nothing if the identifier is not found. Pseudocode for this helper algorithm is shown below.

Algorithm 8 node.find_node(i, identifier, result)

```
find the first difference between the key and identifier
if marching made it through the key and only part of the identifier then
    get the forward edge following the next character of the identifier
    if the forward edge does not exist then
        return result
    end if
    return the value of a recursive find call on the forward edge
end if
if exact match between identifier and key then
    update result
end if
return result
```

This algorithm first checks if the entire key was matched and the identifier was not completely matched. In this case, the forward edge corresponding to the first

unmatched character in the identifier must be followed and returned. If this forward edge does not exist, then nothing should be returned. The check after this is for an exact match. When this occurs, the result is this node and can be immediately returned. Otherwise, assuming that none of the previous conditions were met, nothing can be returned.

Now, in the delete algorithm above there is a *clean_node*() operation. This operation is responsible for maintaining all of the requirements of a compressed prefix tree. Its behavior was discussed before. The pseudocode for this helper algorithm is shown below.

Algorithm 9 node.clean_node()

```
if this node has a value then
    return
end if
if this node has no forward edges then
    remove this node from back edge
    remove all cross edges to this node
    recursively call clean_nodes on the back edge
else if this node has precisely one forward edge then
    merge this node with its forward edge
    update references to point to the new merged node
end if
```

All in all, the delete operation descends the tree exactly once to get the removed node. Then in the worst case, it ascends the entire tree, merging nodes along the way, resulting in the total worst case complexity of the delete operation being $2log(n)$.

3.3 Get

The next operation is the most basic; supporting the ability to query for a specific element with the element's identifier. It does this by using the same method that the delete operation used to do an exact match in the tree. Since the same operation is used the $find_node()$ helper algorithm can be utilized. Pseudocode for the get operation is shown below.

Algorithm 10 get(identifier)

```
run find_node() on the root node with the given identifier
if no node was found, or the found node does not have a value then
    return nothing
end if
return the found node
```

The complexity of this algorithm is that of the $find_node()$ helper algorithm. Which is $log(n)$.

3.4 Pattern Select

To simplify the selection algorithm, it is broken into two parts. The first takes a pattern and returns all nodes in the tree that match that pattern. The restriction is that the pattern must only be for one layer. The case for multiple layers is handled by the second filter algorithm. What makes the pattern select operation important is that it has the potential to reduce the number of nodes that need to be pattern matched. The pseudocode for this algorithm is shown below.

Algorithm 11 pattern_search(*pattern*)

---

if the pattern contains a wildcard character **then**
  get the portion of the *pattern* before the wildcard character, $prefix$
  get the portion of the *pattern* after the wildcard character, $suffix$
  run a match_node() operation with the $prefix$ on the root node
  if no node was matched **then**
    return an empty set
  end if
  do a left and right tree decent to get the first and last matched nodes respectively
  for each matched node **do**
    if identifier matches the $suffix$ **then**
      add this node to the result set
    end if
  end for
  return the result set
else
  do a find_node() operation on the pattern
  if no node was found **then**
    return an empty set
  end if
  return a set containing the found node
end if

First, this algorithm checks to see if the wildcard operator is present in the specified pattern. If it is not, then this operation is as simple as the previous get algorithm. Otherwise, the algorithm finds the prefix of the given pattern. Once found, an approximate match on the tree is run starting at the root node. An approximate match should return the node in the tree which contains the prefix, but may not necessarily be the prefix. The helper algorithm *match_node*() is necessary since both of the previous *get_node*() and $find_node()$ operations perform exact matching, which is not appropriate for this algorithm. Much like the $find_node()$ operation, *match_node*() is read-only and does not need to keep track of previous and next pointers. In addition, since exact matches are not necessary the *match_node*() operation can be simplified even more. The pseudocode for this algorithm is shown below.

Algorithm 12 node.match_node(i, identifier, result)

```
find the first difference between the key and identifier
if the key and identifier are exact matches then
    set result's center to this node
end if
if the full key was matched but not the entire identifier then
    get forward edge following next char in identifier
    if forward edge does not exist then
        set result's center to null
        return result
    end if
    return the value of a recursive match call on the forward edge
end if
```

Now that the matched node was found the algorithm finds the start and end nodes for the candidate matches. The start is found by walking down the tree following the leftmost path to the bottom. The end is found similarly but by following the rightmost path. Now the algorithm walks from the start to end following the next neighbor relationship; at each node it does an additional pattern match on the portion of the pattern excluding the prefix. If the candidate node matches the remaining pattern, then it is added to an answer set. Once the end node is reached the answer set is returned.

Subsequently, the pattern filter algorithm does at most two tree descents. This worst case occurs when the root node is the matched node. Then it iterates through all of the nodes between the start and end nodes resulting in a complete search. In total this gives the pattern filter algorithm a complexity of $m + 2log(n)$, or $O(m)$ where $m < n$. In this case, m is the size of the candidate set.

Consequently, in the worst case, this selection algorithm does about the same as an array and a hash map who's complexity is bounded by the candidate set size. However, in most use cases this algorithm is much faster than an array. This improvement comes from the fact that instead of comparing all elements in the tree against a pattern, a smaller set of elements can be found which all match the prefix of the provided pattern. Then a smaller pattern can be used to evaluate the smaller candidate set. This reason is why it is essential to think about a good identifier schema beforehand.

The complexity equation is recomputed with more precise bounds to get a better idea of the complexity of the selection operation. This way it can be given some 'test' values for comparison to a normal database's table scan.

Let g represent the number of categories. For simplicity's sake let each category contain an even number of elements. Let n be the total number of elements in the tree. Let q be the length of the pattern. Let $p = floor(q/2)$ be the length of the pattern before the wildcard character, approximately half the pattern is a prefix. Given the logic stated before, the candidate set is iterated across. Well, let's assume that the pattern is selecting a single candidate set. Let's also assume that the complexity of

executing the portion of the pattern after the wildcard is linear relative to the size of the query. Putting this together, the size of the candidate set is n/g, and for each element the pattern is $q - p$ characters long. Together this gives a complexity of $(q - p)n/g$ to construct the result set with the candidate set. Now to find the complexity of computing the candidate set. It involves walking down the tree for every character in the prefix pattern, p. Then making two tree descents from that point to the base of the tree results in $2(log(n) - p)$ node visits. All together this gives a complexity of $p + 2(log(n) - p) + (q - p)n/g$.

Given all of this the complexity of a balanced prefix trie-like this (balanced in terms of group size) would be $p+2(log(n)-p)+(q-p)n/g$. While the corresponding complexity for a hash map would be qn/g since it already has the candidate set computed.

Figure 3.3: Pattern selection: tree vs hash map: $g = 1, p = 0, q = 1$

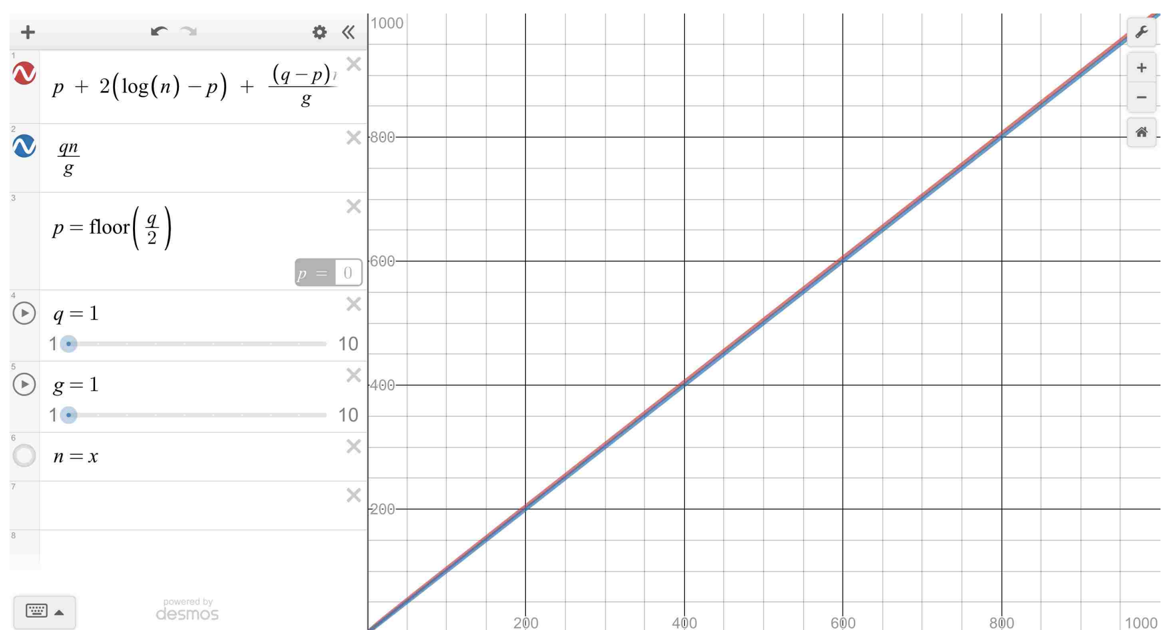


Figure 3.4: Pattern selection: tree vs hash map: $g = 2, p = 0, q = 1$

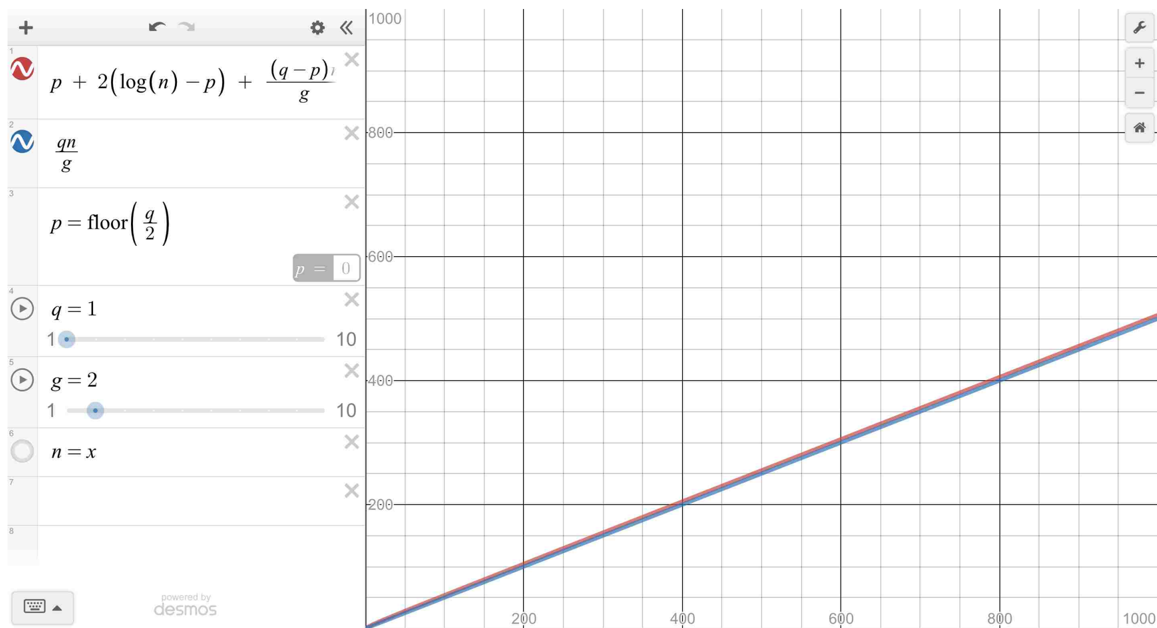
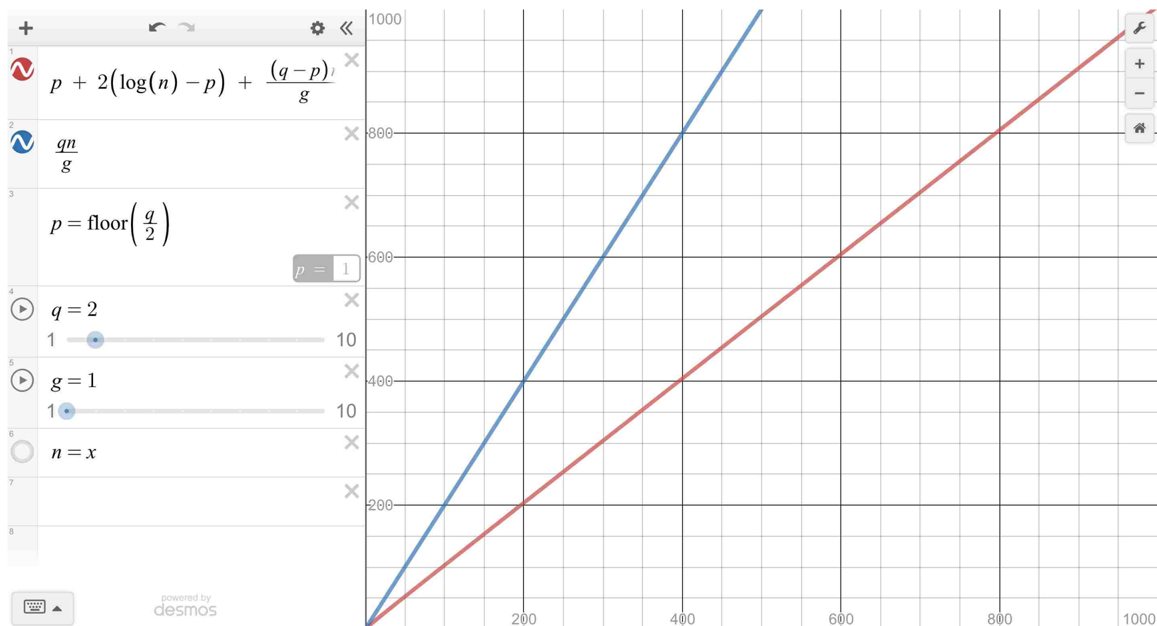


Figure 3.5: Pattern selection: tree vs hash map: $g = 1, p = 1, q = 2$



3.5 Link

The next operation is link. This operation takes two elements and creates a cross edge between them. No particular condition needs to be met, so the implementation is rather straightforward. This algorithm is implemented by using $find_node()$ to get both individual nodes. If either of the two nodes is not found, then the algorithm

can not continue and should return. If both elements were found in the tree, then a cross edge is added between the two elements.

Algorithm 13 link(obj_1, obj_2)

```
get the identifiers for both objects
run a find_node() operation for both identifiers on the root node
if either nodes were not found then
    return
end if
add a cross edge between the two found nodes
```

The complexity of this operation is the time it takes to find both nodes. The other operations can largely be ignored since they are all simple operations. Thus, the worst case complexity of this algorithm is $2 * find_node() = 2log(n)$.

3.6 Nested Select

Now that the process of adding cross edges between nodes has been established, a nested select algorithm can be presented. This selection algorithm is the second part of the selection process. It is responsible for taking the sets produced by the pattern select algorithm and ensuring that cross edges exist between nodes in the adjacent set. The general process for this algorithm was outlined before. In greater detail, using a hash set to represent the sets produced by the pattern filter algorithm, relationships between adjacent sets can quickly be evaluated. This algorithm is called select in the pseudocode because in an actual implementation this would be the complete selection algorithm. The pseudocode is shown below.

Algorithm 14 select($pattern$)

```
split the pattern on the layer delimiter
for each layer do
    run a prefix_match() operation on the layer pattern
    store the result by layer for later use
end for
create an answer set equal to the nodes in the first layer
for each layer, start with the second layer do
    create a candidate set
    for each node in the layer's matched set do
        if the node has a cross edge to the current answer set then
            add this node to the candidate set
        end if
    end for
    update the answer set to be the candidate set
end for
return the answer set
```

This algorithm begins by running $prefix_match()$ for each layer to get a matching set of elements for each layer. In the worst case this requires n operations l times, l is the number of layers in the pattern. Then starting with the second layer, every element in the layer is checked to make sure that a cross edge exists to the previous layer. Each node can check its cross edges in $O(n)$ time since at most each node has n cross edges, and each cross edge can be checked in $O(1)$ since a hash set is being used to represent the nodes in the previous layer. In total the complexity is $layermatching + layerfiltering = ln + lnn$, resulting in a worst-case complexity of $O(ln^2)$. This worst-case complexity is fairly bad (despite being equal to that of a normal table scan) but is extremely unlikely to occur. It also assumes that the developer has enough memory to spend to maintain a cross edge between every node. The total number of cross edges would be massive.

Due to the rarity of the worse case, the exact complexity is estimated. By using example values, like in the previous pattern filter algorithm, additional comparisons can be shown between a normal table scan and this cumulative filter algorithm. Let l be the number of linked patterns. Let r be the average number of relationships between elements. To simplify the complexity analysis without misleading results, assume that the length of each linked pattern is the same. Taking the previous equation $p + 2(log(n) - p) + (q - p)n/g$. The portion pertaining to set selection is repeated once for every layer changing the equation to $l(p+2(log(n)-p)+(q-p)n/g)$. To resolve relationships between all the sets every set has to go through additional filtering except for the last set giving a complexity of $(l-1)rn/g$. Using these variables the complexity assuming a balanced prefix trie is $l(p + 2(log(n) - p) + (q - p)n/g) + (l - 1)rn/g$. The corresponding complexity for a hash map is the same as before with the same process to filter sets, $l(qn/g) + (l - 1)rn/g$

Figure 3.6: Selection: tree vs hash map: $l = 1, q = 1, g = 1, r = 1$

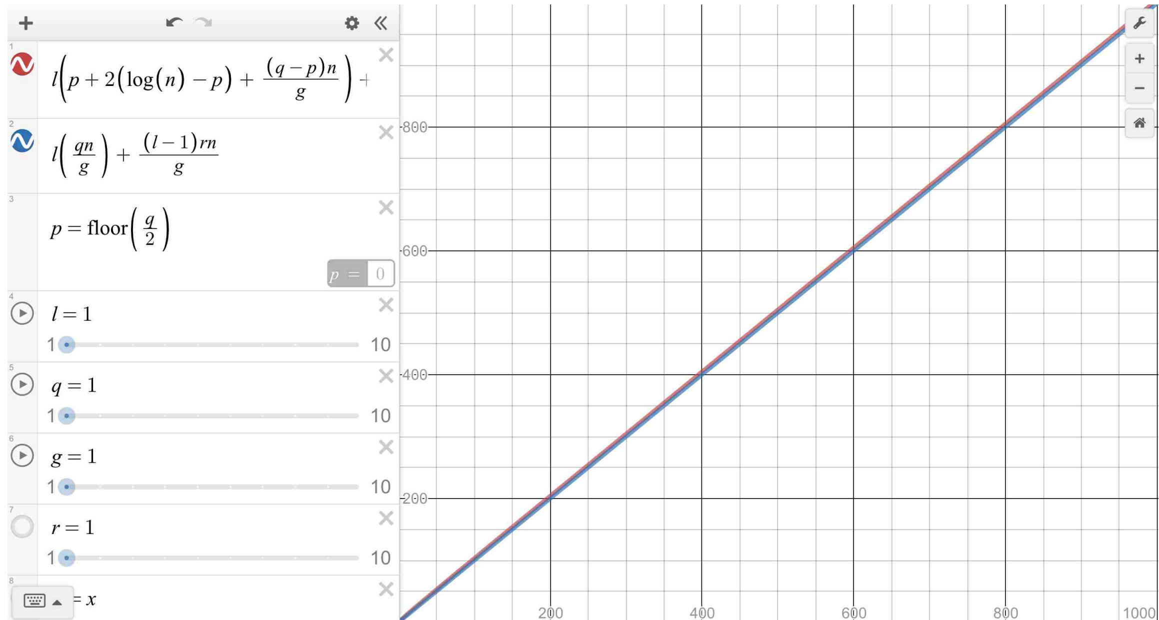


Figure 3.7: Selection: tree vs hash map: $l = 1, q = 1, g = 2, r = 1$

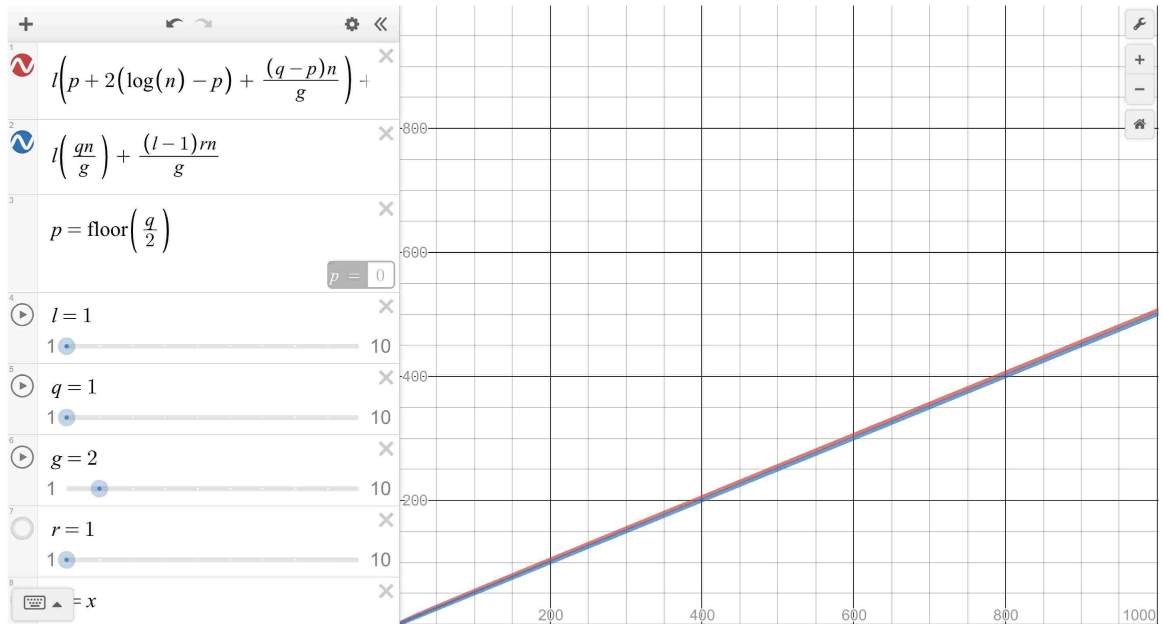


Figure 3.8: Selection: tree vs hash map: $l = 1, q = 2, g = 1, r = 1$

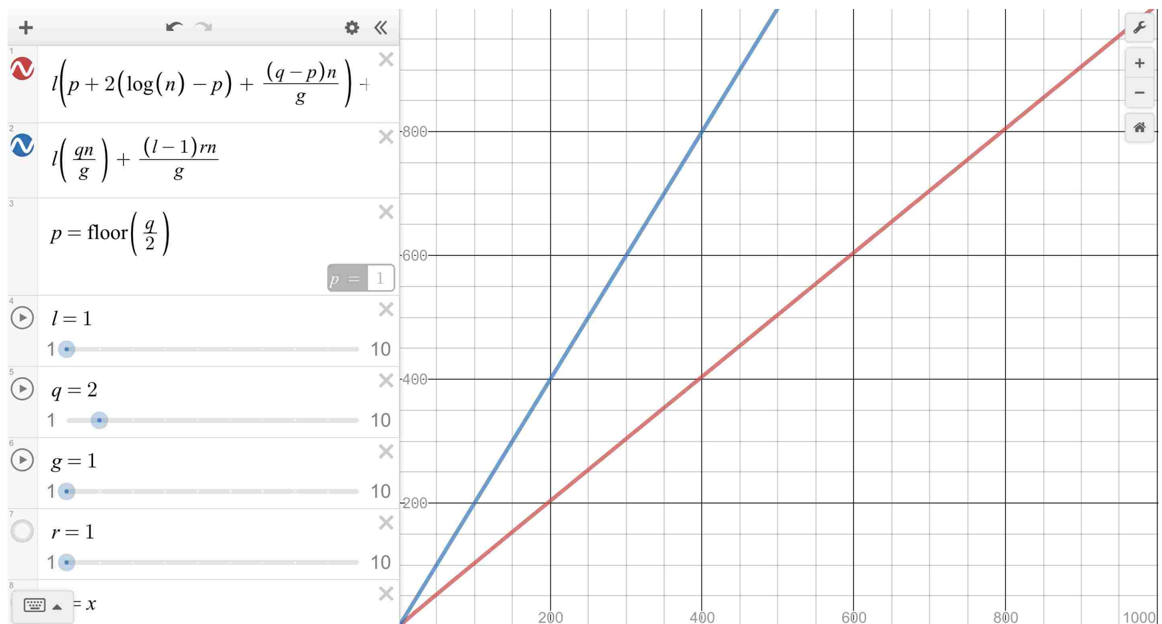


Figure 3.9: Selection: tree vs hash map: $l = 2, q = 1, g = 1, r = 1$

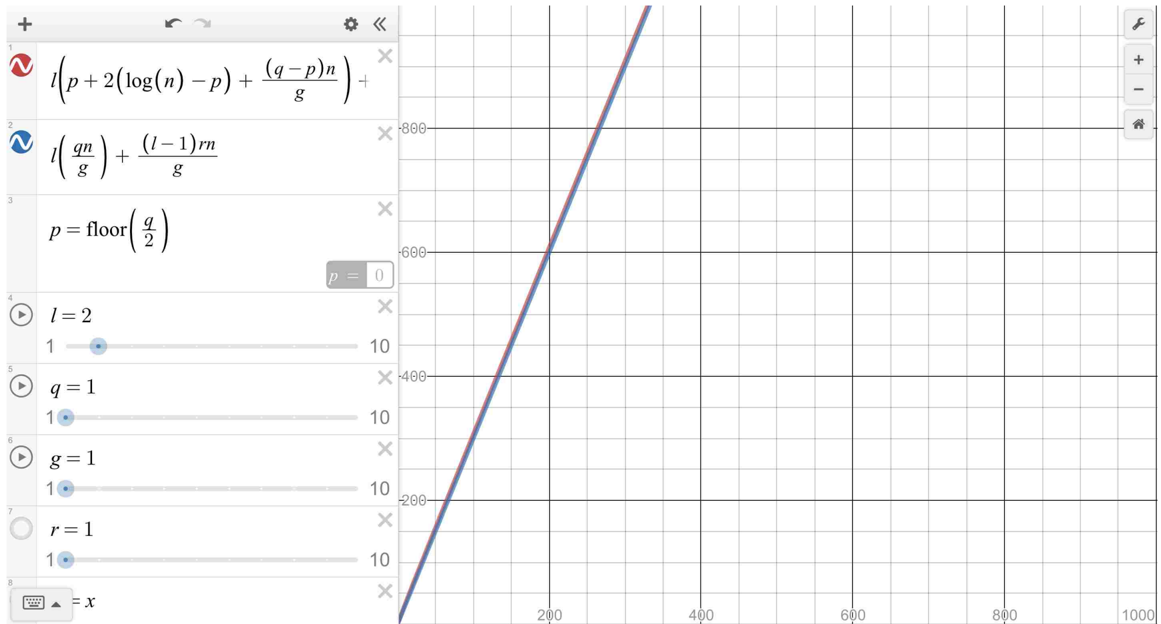
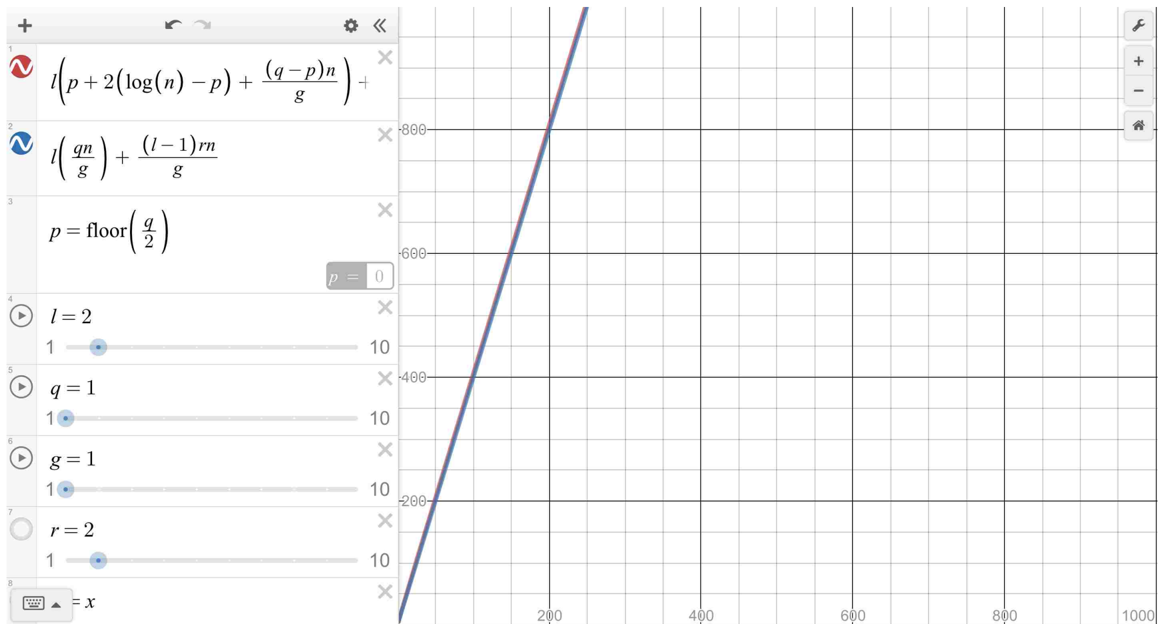


Figure 3.10: Selection: tree vs hash map: $l = 2, q = 1, g = 1, r = 2$



Chapter 4 Case Studies

The algorithms presented in this thesis were designed with the intention of being used in a resource manager for a game engine. This chapter focuses on the performance of each part of the solution and discusses all the real use cases it solved. The proposed solution is by no means a silver bullet, but it is another useful tool that should be considered when designing a resource manager.

4.1 Resource Management in Game Engines

I have had a passion for video games for a long time, and have spent a significant amount of time designing and implementing game engines. As mentioned before, a common problem in game engine design is resource management. In the most recent engine, a somewhat limiting requirement was to give writers and artists the ability to modify in-game content without needing to re-release the game.

Facilitating this requirement is immensely challenging on many fronts. The first challenge being the sheer number of resources that the developer may want to use or modify. The second being the various types of resources which need to be stored; each of which has its own set of unique requirements. The simplest solution is to give every resource a unique identifier. Then the developer could look up the identifier for a specific resource every time they wanted to use it. At first glance, this seems like a winner, mainly since a hash map could be used to facilitate a structure like this. However, when identifiers only represent the uniqueness of a resource, they fail to give any information about the underlying resource. Using an identifier schema like this is not scalable in any sizeable system.

Also, a common feature that engines choose to support is modding. Modding is a term referring to the process of modifying a published work to contain additional content that wasn't provided or developed by the original publisher. When the developers use a randomly generated id schema a comprehensive list of identifiers has to be released. If not, the modder has to guess at identifiers and construct a list of resources so that they can avoid naming conflicts. Due to this, releasing a list of identifiers is not avoidable.

Despite their many drawbacks, randomly generated identifiers cannot be avoided in a game engine. Consider the case where new instances of a specific resource are repetitively spawned; each instance is completely independent of the previous instances. Generating unique identifiers for each of these instances would be immensely difficult without using a randomly generated identifier.

The team decided to create a naming scheme that uses categories to meet these restrictions, just like the identifier schema presented. By using categories in the naming schema, specific metadata about a resource can be queried without having to look at the referenced resource on disk directly. This design is immensely useful from a development perspective. Given an identifier, the developer has all necessary information about the resource such as its type, what it is called, as well as how

it acts. Besides, this allows each category to have different naming restrictions and formatting. As a developer, memorizing the structure of an identifier is much easier than memorizing a list of identifiers. A categorized naming schema also allows for a lookup table to be structured and easily searched. It also allows the creator of the resource to specify the name, resulting in the people primarily working with the resource to already know what it is called. Identifiers have a randomly generated sequence appended to them to address the random generation problem.

An important note is that all of the added identifier rules do not break any of the existing rules specified in the description. Such as information in the identifier going from least to most specific; this is true even in randomly generated resources since the random sequence appears at the end. Random resources can even be selected without knowing the random sequence if the developer knows there will only ever be one of them.

Using the naming system described a few of the identifiers in various applications are listed below.

1. 'vid:region:Test'
2. 'vid:animation:character/human_body'
3. 'vid:audio:planet1/city_theme'

However, the system needed more flexibility than just being able to load objects from disk. If this were the only feature available, then a modder or developer would never be able to instantiate resources with generated identifiers in their custom content. In consideration of this, the ability to create new objects with an identifier was added. It meant that in many cases resources were acting more as templates than modifiable resources. Luckily the presented naming schema was carefully designed to be able to support this functionality. The implementation in the engine involved binding resolvers to each ontological category. This resolver is responsible for defining the behavior of the resource manager when the requested resource is not present in the tree. The resolver allowed each category to support resource resolution and resource initialization individually and independently of other resolvers. Implementing this became reasonably tricky due to dynamic object resolution in Java, but since each ontological category contained objects of the same type most resolvers could create a new instance of the underlying type. Otherwise, the class name needed to be inferred or explicitly stated so that the appropriate implementation could be dynamically loaded for instantiation. A list containing identifiers, which behave as constructors, is shown below. The resolved resource may not necessarily have the same identifier as the constructor.

1. 'vid:actor:zombie.BasicZombie'
2. 'vid:player:ethan.BasicPlayer'

Another requirement was the ability to resolve a new copy of the resource each time it is requested. Consider having audio special effects stored in the resource

manager. If multiple actors are all trying to play the same special effect, then an error would occur because each of the actors would be restarting the same effect when they tried to play it. In reality, each of these actors wants their copy of the resolved resource. Luckily, in these cases, the resolver can specify if identifiers within their managed ontological category should be returned as direct references or clones.

Another requirement is that the resource manager must be able to be serialized and deserialized to a file at any point. Being able to save and load the state of the entire application is a critical use case and can become very complicated as more data sources are added. Most games save their state extraordinarily often to prevent the player from getting mad when they die after not saving for a while.

It may seem excessive at this point, but there are even more requirements. Keep in mind that at the end of the day this is the resource manager for every resource that the engine could ever potentially use. In many cases, game engines are highly parallelized to improve performance. Usually, game loops and render loops are broken up into independent sections that can all run in parallel. However, with one managed data source this could be a problem. Consequentially, the resource manager must have a solution in place for the readers-writers problem. This problem is discussed in greater detail later on, but there are many clever ways to support thread safety across a tree indexed resource manager.

Finally, the last restriction is support for set searches in linear time. Supporting this requirement has been a significant focus of the thesis, so no further discussion is given beyond stating that it is a requirement.

Many restrictions were presented. In summary, an itemized list containing all the restrictions for a resource manager in a game engine is shown below.

1. Caching resources given an identifier
2. Resolving new resource instances given an identifier
3. Loading resources given an identifier
4. Control over returning clone or reference of a resource
5. Gather an entire category of objects given a pattern
6. Gather a subset of a category given a pattern
7. Serialize the data structure to a file
8. Deserialize the data structure from a file
9. Thread safe resource management

These restrictions and the presented naming facilitate many additional features for free. One of which is maintaining references between two resources. Since an identifier is sufficient to resolve the underlying resource, it does not need to be resolved until necessary. Meaning that large sections of the tree can be left empty and lazy-loaded when needed. This lazy-loading allows for load times to be reduced.

It was mentioned earlier, but the most basic solution to this problem which meets all of the requirements is to use a set of hash maps all of which have resolvers. While this solution does reach the minimum case complexity of $O(1)$ insertion and removal with $O(n)$ searches this greatly complicates any functional implementation. The engine would have to maintain a large set of 'buckets' each of which would need to be combined during saving and loading.

Since no existing solutions seemed like a good fit, this compressed prefix tree backed resource manager implementation was designed and implemented. Also, by having a simple custom implementation resolvers could be bound directly into each category for resolution. The ending implementation ended up being a couple of hundred lines long and is very easy to maintain and test. In addition, saving the loading the state of the trie is as simple as deserializing and serializing the tree to a file respectively. By doing this, a new game is easily created by 'seeding' a new tree with a game's default resources. Then to save the game, any category marked as modifiable saves a complete serialization of their altered copy of the default resource; allowing for the player to modify resources in the game without breaking the developer's initial design.

However, the proposed resource manager was not a perfect replacement over a set of hash maps. The only drawback is its worse complexity for insert and delete. A small caching system was layered on top of the tree allowing for the most frequently accessed resources to be immediately returned on request. This system works incredibly well since the set of resources requested every frame rarely changes. Even in the case of a cache miss, it only results in a worse case complexity of $log(n)$.

4.2 AWS Resource Management

AWS is the general name for Amazon's web services. It includes services for distributed computing, data storage, security, and much more. One of the more notable properties of AWS is that each resource the user can interact with has a unique identifier. These identifiers are categorized into each of the different provided services and are typically relative to specific physical regions. A few example identifiers are listed below.

- 'arn:aws:codebuild:us-east-1:123456789012:project/my-demo-project'
- 'arn:aws:iam::123456789012:user/David'
- 'arn:aws:rds:eu-west-1:123456789012:db:mysql-db'
- 'arn:aws:s3:::my_corporate_bucket/exampleobject.png'
- 'arn:aws:codecommit:us-east-1:123456789012:MyDemoRepo'

When comparing the presented identifier schema and this identifier schema, the two look almost the same. Each identifier has a category with a relatively unique identifier. The similarity between the two is entirely intentional. AWS's identifiers

were a significant inspiration for the compressed prefix tree backed resource manager shown in this thesis. Using identifiers in this way allows for a ubiquitous syntax for all resources across the entire platform. Also, the type and function of each identifier are made clear even with no knowledge of the underlying resource.

Their identifier structure was not the only inspiration. Another feature that AWS provides is resource matching. One notable implementation of this is describing security policies. When granting or revoking access to a specific resource on a policy, the user is required to specify to which resources the policy applies. Their syntax uses the UNIX like wildcard syntax for matching. However, AWS also uses an additional '::' as a shorthand for ':*:'. This shorthand is commonly used for resources which do not require a specific identifier.

AWS probably does not implement its resource management with one massive database, but a compressed prefix tree backed database or resource manager could be used. Granted the presented tree backed resource manager would have to be substantially modified for this use case. There are way too many resources than what could be reasonably managed by a single machine. Due to this, a distributed architecture would have to be applied to the proposed solution. It seems possible since a distributed local network could each own subsections of the tree. However, a solution like this is not proposed or outlined here.

Copyright© Ethan Toney, 2018.

Chapter 5 Conclusion

By using compressed prefix trees along with a carefully designed identifier and pattern schema large sets of data can be selected without any substantial performance hits when compared to existing solutions. Of course, tree backed resource managers do come with downsizes many of these can be ignored for various applications. The most suitable application being video game engines due to the unique set of requirements they demand. Modeling resources as being in hierarchical groups of ontologies end up being an intuitive structure to form an identifier schema around. These identifiers also provide an easy interface for technical and nontechnical developers to work with the underlying resources.

Future Work

A comparison to databases was made, but it was more as a baseline than a real comparison. All of the algorithm presented in this work can be directly implemented in existing database applications. To that end, many databases already support using a trie-like structure. I think an accurate runtime comparison between the two implementations would be interesting to see. If the database implementation were minimal enough, it would be an ideal solution for resource management in a game engine, especially if it allowed for resources to be resolved as they are requested.

Copyright© Ethan Toney, 2018.

Appendices

.1 Implementation

../implementation/src/main/java/edu/uky/util/Trie.java

```
package edu.uky.util;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.function.Function;
import java.util.regex.Pattern;

public class Trie<T> {

  TrieNode<T> root_node;

  private final Function<T, String> obj_identifier;

  public Trie() {
    this(Object::toString);
  }

  public Trie(Function<T, String> obj_identifier) {
    this.root_node = new TrieNode<T>(null);
    this.obj_identifier = obj_identifier;
  }

  public T find(String identifier) {
    TrieSearchResult<T> result = root_node.find(0, identifier.
        toCharArray(), new TrieSearchResult<T>());

    if (result.center == null)
      return null;

    return result.center.value;
  }

  public boolean insert(T obj) {
    String identifier = obj_identifier.apply(obj);

    TrieSearchResult<T> result = root_node.get(0, identifier.toCharArray
        (), new TrieSearchResult<T>());

    // follow subtrees for left and right branches
    result.resolve();

    // establish links between leaf nodes
    if (result.left != null) {
      result.center.setPrevious(result.left);
    }
    if (result.right != null) {
      result.center.setNext(result.right);
    }
```

```java
  // update value
  T prev_value = result.center.value;
  result.center.value = obj;
  return !obj.equals(prev_value);
}

/**
 * Link two objects together. Creates a cross edge between the two
    objects if they exist.
 *
 * @param obj_1 - The first object in the link.
 * @param obj_2 - The second object in the link.
 */
public void link(T obj_1, T obj_2) {
  String identifier_1 = obj_identifier.apply(obj_1);
  String identifier_2 = obj_identifier.apply(obj_2);

  TrieNode<T> node_1 = root_node.find(0, identifier_1.toCharArray(),
      new TrieSearchResult<T>()).center;
  TrieNode<T> node_2 = root_node.find(0, identifier_2.toCharArray(),
      new TrieSearchResult<T>()).center;

  if (node_1 == null || node_2 == null)
    return;

  node_1.addCrossEdge(node_2);
}

public List<T> match(String pattern) {
  String[] pattern_layers = pattern.split(":");
  List<List<TrieNode<T>>> layers = new ArrayList<>();
  List<T> result = new ArrayList<>();

  // gather data sets
  for (String layer : pattern_layers) {
    List<TrieNode<T>> subset = new ArrayList<>();

    // do pattern matching
    if (layer.contains("*")) {
      int wildcard_index = layer.indexOf("*");
      String prefix = layer.substring(0, wildcard_index);
      String suffix = (wildcard_index == layer.length()) ? "" : layer.
          substring(wildcard_index+1);

      TrieSearchResult<T> search_result = root_node.match(0, prefix.
          toCharArray(), new TrieSearchResult<T>());

      if (search_result.center == null)
        return result;

      TrieNode<T> first = search_result.center.left();
      TrieNode<T> last = search_result.center.right();

      String[] suffix_parts = suffix.split("\\*");
```

```
      for (int i=0 ; i<suffix_parts.length ; i++)
        suffix_parts[i] = Pattern.quote(suffix_parts[i]);
      suffix = ".*" + String.join(".*", suffix_parts);
      if (layer.endsWith("*"))
        suffix += ".*";
      Pattern regex = Pattern.compile(suffix);
      while (true) {
        String identifier = obj_identifier.apply(first.value);
        if (regex.matcher(identifier).matches()) {
          subset.add(first);
        }
        if (first.equals(last))
          break;
        first = first.next;
      }

    // do exact lookup
    } else {
      TrieSearchResult<T> search_result = root_node.find(0, layer.
          toCharArray(), new TrieSearchResult<T>());

      if (search_result.center == null || search_result.center.value
          == null)
        return result;

      subset.add(search_result.center);
    }

    layers.add(subset);
  }

  // resolve relations
  HashSet<TrieNode<T>> filter_set = null;
  if (layers.size() > 1) {
    // construct and add first set
    filter_set = new HashSet<>();
    filter_set.addAll(layers.get(0));
    // for each subsequent set filter the filter set
    for (int i=1 ; i<layers.size()-1 ; i++) {
      HashSet<TrieNode<T>> next_set = new HashSet<>();

      for (TrieNode<T> node : layers.get(i))
        if (node.valid(filter_set))
          next_set.add(node);

      filter_set = next_set;
    }
  }

  // construct answer set
  for (TrieNode<T> node : layers.get(layers.size()-1)) {
    // if filter set is not null filter with it
    if (filter_set != null && !node.valid(filter_set)) {
      continue;
```

```
      }
      result.add(node.value);
    }

    return result;
  }

  public boolean remove(T obj) {
    String identifier = obj_identifier.apply(obj);

    TrieSearchResult<T> result = root_node.find(0, identifier.
        toCharArray(), new TrieSearchResult<T>());

    // follow subtrees for left and right branches
    result.resolve();

    // sanity check
    if (result.center == null || result.center.value == null)
      return false;

    // update value
    result.center.value = null;
    result.center.clean(this);

    // update references
    if (result.left != null && result.right != null) {
      result.left.setNext(result.right);
    } else if (result.left != null) {
      result.left.setNext(null);
    } else if (result.right != null) {
      result.right.setPrevious(null);
    }
    return true;
  }

  @Override
  public String toString() {
    StringBuilder str = new StringBuilder();
    root_node.toString('^', str, 0);
    return str.toString();
  }

}
```

../implementation/src/main/java/edu/uky/util/TrieNode.java

```
package edu.uky.util;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.TreeSet;
```

```java
public class TrieNode<T> {

  TrieNode<T> next, previous;
  T value;

  private HashMap<Character, TrieNode<T>> forward_edges = new HashMap
      <>();
  private TreeSet<Character> sorted_forward_keys = new TreeSet<>();
  private List<TrieNode<T>> cross_edges = new ArrayList<>();
  private char[] key = new char[0];
  private TrieNode<T> back_edge = null;

  public TrieNode(TrieNode<T> parent) {
    this.back_edge = parent;
  }

  public TrieNode(TrieNode<T> parent, TrieNode<T> copy) {
    back_edge = parent;

    forward_edges = new HashMap<>(copy.forward_edges);
    sorted_forward_keys = new TreeSet<>(copy.sorted_forward_keys);
    cross_edges = new ArrayList<>(copy.cross_edges);
    previous = copy.previous;
    next = copy.next;
    key = copy.key;
    value = copy.value;
  }

  void addCrossEdge(TrieNode<T> n) {
    cross_edges.add(n);
    n.cross_edges.add(this);
  }

  void addEdge(char c, TrieNode<T> n) {
    forward_edges.put(c, n);
    sorted_forward_keys.add(c);
  }

  /**
   * Removes unnecessary nodes from tree
   */
  void clean(Trie<T> trie) {
    if (value != null)
      return;

    // if no children remove from parent
    int num_forward_edges = forward_edges.size();
    if (num_forward_edges == 0) {
      back_edge.removeEdge(key[0]);
      clearEdges();
      back_edge.clean(trie);

    // if one child merge
```

```
  } else if (num_forward_edges == 1 ) {
    TrieNode<T> child = forward_edges.get(sorted_forward_keys.first())
        ;

    // modify this nodes key
    child.prependKey(key);
    child.back_edge = back_edge;

    if (back_edge == null) {
      trie.root_node = child;
    } else {
      back_edge.addEdge(key[0], child);
    }

    clearEdges();
  }
}

void clearEdges() {
  forward_edges.clear();
  sorted_forward_keys.clear();
  for (TrieNode<T> edge : cross_edges) {
    edge.cross_edges.remove(this);
  }
  cross_edges.clear();
}

TrieSearchResult<T> find(int i, char[] identifier, TrieSearchResult<T>
     result) {
  // find the first difference between the identifier and this node
  int j = 0; // marks the split point in this nodes identifier
  int k = i; // marks the split point in the passed identifier
  for (; j<key.length && k<identifier.length && key[j]==identifier[k]
      ; j++, k++) {}

  // if we made it through the key but not the identifier
  if (j == key.length && k < identifier.length) {
    char edge = identifier[k];

    updateResultAlongPath(result, edge);

    TrieNode<T> forward_edge = forward_edges.get(edge);

    // if no forward edge to take
    if (forward_edge == null)
      return result;

    return forward_edge.find(k, identifier, result);
  }

  // if exact match
  if (k == identifier.length && j == key.length) {
    updateResultWhenAnswer(result);
  }
```

```
  return result;
}

TrieSearchResult<T> get(int i, char[] identifier, TrieSearchResult<T>
    result) {
  // find the first difference between the identifier and this node
  int j = 0; // marks the split point in this nodes identifier
  int k = i; // marks the split point in the passed identifier
  for (; j<key.length && k<identifier.length && key[j]==identifier[k]
      ; j++, k++) {}

  // if this node can become the identifier (base node)
  if (isEmpty() && isLeafNode()) {
    setKey(i, identifier.length, identifier);
    updateResultWhenAnswer(result);
    return result;
  }

  // if we didn't get through the entire key, this node must be split
  if (j < key.length) {
    // copy this node
    TrieNode<T> prev = new TrieNode<T>(this, this);

    // update previous state
    prev.setKey(j, key.length, key);
    boolean has_value = value != null;

    // update this state
    value = null;
    clearEdges();
    addEdge(key[j], prev);

    // update links to previous node
    if (prev.next != null) {
      prev.next.setPrevious(prev);
    }

    // update back links to previous node
    for (TrieNode<T> node : prev.forward_edges.values()) {
      node.back_edge = prev;
    }

    // if the identifier ended part way through this node
    if (k == identifier.length) {
      setNext(prev);

      // make sure our key bounds are accurate
      k = identifier.length - 1;

      setKey(0, j, key);

      updateResultWhenAnswer(result);
```

```
      return result;

    // if the identifier only matched part of the key
    } else {
      if (has_value) {
        setNext(null);
        prev.setPrevious(previous);
      }

      TrieNode<T> next = new TrieNode<T>(this);
      next.setKey(k, identifier.length, identifier);

      addEdge(identifier[k], next);

      setKey(0, j, key);

      updateResultAlongPath(result, identifier[k]);

      return next.get(k, identifier, result);
    }
  }

  // if we didn't get through the entire identifier, a leaf node must
     be added or searched
  if (k < identifier.length) {
    char edge = identifier[k];
    TrieNode<T> forward_edge = forward_edges.get(edge);
    // if no forward edge create it and continue search
    if (forward_edge == null) {
      forward_edge = new TrieNode<T>(this);
      addEdge(edge, forward_edge);
    }
    updateResultAlongPath(result, edge);
    return forward_edge.get(k, identifier, result);
  }

  // if we got through both the identifier and the key then, this node
      is an exact match
  updateResultWhenAnswer(result);
  return result;
}

private boolean isEmpty() {
  return value == null;
}

private boolean isLeafNode() {
  return forward_edges.size() == 0;
}

TrieNode<T> left() {
  if (value != null)
    return this;
```

```
    if (sorted_forward_keys.size() == 0)
      return this;

    Character edge_key = sorted_forward_keys.first();
    if (edge_key == null)
      return this;

    TrieNode<T> forward_edge = forward_edges.get(edge_key);
    if (forward_edge == null)
      return this;

    return forward_edge.left();
  }

  TrieSearchResult<T> match(int i, char[] identifier, TrieSearchResult<T
     > result) {
    // find the first difference between the identifier and this node
    int j = 0; // marks the split point in this nodes identifier
    int k = i; // marks the split point in the passed identifier
    for (; j<key.length && k<identifier.length && key[j]==identifier[k]
        ; j++, k++) {}

    if (j==key.length || k==identifier.length) {
      result.center = this;
    }

    if (j==key.length && k<identifier.length) {
      char edge = identifier[k];

      TrieNode<T> forward_edge = forward_edges.get(edge);

      // if no forward edge to take
      if (forward_edge == null) {
        result.center = null;
        return result;
      }

      return forward_edge.match(k, identifier, result);
    }

    return result;
  }

  TrieNode<T> next(char edge) {
    if (sorted_forward_keys.size() == 0)
      return null;

    Character edge_key = sorted_forward_keys.ceiling((char) (edge + 1));
    if (edge_key == null)
      return null;

    return forward_edges.get(edge_key);
  }
```

```java
private void prependKey(char[] identifier) {
  char[] new_key = Arrays.copyOf(identifier, identifier.length + key.
      length);
  for (int i=identifier.length, j=0 ; j<key.length ; i++, j++) {
    new_key[i] = key[j];
  }
  key = new_key;
}

TrieNode<T> previous(char edge) {
  if (sorted_forward_keys.size() == 0)
    return null;

  Character edge_key = sorted_forward_keys.floor((char) (edge - 1));
  if (edge_key == null)
    return null;

  return forward_edges.get(edge_key);
}

void removeEdge(char c) {
  forward_edges.remove(c);
  sorted_forward_keys.remove(c);
}

TrieNode<T> right() {
  if (sorted_forward_keys.size() == 0)
    return this;

  Character edge_key = sorted_forward_keys.last();
  if (edge_key == null)
    return this;

  TrieNode<T> forward_edge = forward_edges.get(edge_key);
  if (forward_edge == null)
    return this;

  return forward_edge.right();
}

private void setKey(int start, int end, char[] identifier) {
  char[] new_key = new char[end - start];
  for (int i=0, j=start ; j<end ; j++, i++) {
    new_key[i] = identifier[j];
  }
  key = new_key;
}

void setNext(TrieNode<T> node) {
  next = node;
  if (node != null)
    next.previous = this;
}
```

```
void setPrevious(TrieNode<T> node) {
  previous = node;
  if (node != null)
    previous.next = this;
}

@Override
public String toString() {
  String str = "" + super.hashCode();
  if (value != null) {
    str = value + ":" + str;
  }
  return str;
}

private void updateResultAlongPath(TrieSearchResult<T> result, char
    edge) {
  // previous
  // -> last leaf node of closest adjacent neighbor (right node)
  TrieNode<T> prev = previous(edge);
  if (prev != null) {
    result.left = prev;
    result.left_exact = false;

  // -> last parent node
  } else if (value != null) {
    result.left = this;
    result.left_exact = true;
  }

  // next
  // -> first leaf node of closest adjacent neighbor (left node)
  TrieNode<T> next = next(edge);
  if (next != null) {
    result.right = next;
    result.right_exact = false;
  }
}

private void updateResultWhenAnswer(TrieSearchResult<T> result) {
  result.center = this;

  // next
  // -> first leaf node below this node (left node)
  if (forward_edges.size() != 0) {
    // temporarily remove value for left operation
    T prev_val = value;
    value = null;

    TrieNode<T> right = left();
    if (!this.equals(right)) {
      result.right = right;
      result.right_exact = true;
    }
```

```
      value = prev_val;
    }
  }

  boolean valid(HashSet<TrieNode<T>> edges) {
    for (TrieNode<T> cross_edge : cross_edges) {
      if (edges.contains(cross_edge))
        return true;
    }
    return false;
  }

  public void toString(char c, StringBuilder str, int tab_count) {
    for (int i=0 ; i<tab_count ; i++)
      str.append("\t");
    str.append(String.format("[%c]%s", c, new String(key)));
    if (value != null) {
      str.append(" => "+this);
      if (previous != null)
        str.append(" (previous:"+previous+")");
      if (next != null)
        str.append(" (next:"+next+")");
    }
    if (back_edge != null)
      str.append(" (back:"+new String(back_edge.key)+")");

    str.append("\n");
    for (char edge_key : sorted_forward_keys) {
      forward_edges.get(edge_key).toString(edge_key, str, tab_count + 1)
          ;
    }
  }

}
```

../implementation/src/main/java/edu/uky/util/TrieSearchResult.java

```
package edu.uky.util;

public class TrieSearchResult<T> {

  boolean left_exact = false, right_exact = false;
  TrieNode<T> left, center, right;

  public void resolve() {
    if (left != null && !left_exact) {
      left = left.right();
    }
    if (right != null && !right_exact) {
      right = right.left();
    }
  }
```

```java
  @Override
  public String toString() {
    StringBuilder str = new StringBuilder();
    if (left != null) {
      str.append(left + " <- ");
    }
    if (center != null) {
      str.append(center);
    }
    if (right != null) {
      str.append(" -> " + right);
    }
    return str.toString();
  }

}
```

.2 Tests

../implementation/src/test/java/edu/uky/util/TrieTest.java

```java
package edu.uky.util;

import static org.hamcrest.CoreMatchers.equalTo;
import static org.junit.Assert.assertThat;

import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.Random;

import org.hamcrest.collection.IsIterableContainingInOrder;
import org.hamcrest.core.IsNull;
import org.junit.Test;

public class TrieTest {

  private String[][] tests = {
    {
      "object 1", "object 2", "object 3", "objection!", "object 124", "
          object 123"
    }, {
      "abc", "abcd", "abc1", "abcde", "ab", "a", "abcfed"
    }, {
      "person.alice", "person.bob", "person.carl", "dog.sandy", "dog.
          phroto", "dog.sam",
      "cat.calypso", "street.rose", "street.euclid", "street.main"
    }, {
      "ZNY441H", "TpB66ql", "MLOhlxE", "Nrh9sCf", "bJZCfJ6", "PS6bxlK",
          "HtkISdq", "Q9EIF6O",
      "IrhitsO", "majSoX8", "cNlZSQg", "s7NNKGU", "WlgV8SD", "T9XgsT6",
          "Sur7RmM", "BjD511K",
      "tGXKPb2", "4u2QimI", "ejcayld", "wTKylJC", "3O3YMZQ", "SnWMRVQ",
          "ofvnjiB", "AlPg1zr",
```

```
"L8EuptN", "duIg8Db", "5dcsu3F", "UPh6jso", "BvTysHS", "Faz91RX",
    "FAt7R0K", "p0SqQC8",
"QEZBrwp", "5QLQh5c", "bjMLzcT", "LKykgSe", "DP95bt9", "ZV219Vm",
    "XIKTNqG", "AEv3qu8",
"aWXkMph", "uUhPXlg", "fHyk8wK", "N2Rzccl", "BOvTNdr", "g0XHfT7",
    "O9L51Xn", "ghIcYMV",
"I1v5HB2", "THoXmj7", "2MClqak", "LLqMTfP", "UWOxYpw", "wbDXWy3",
    "w9nN3xt", "JJF9Wgz",
"gQfwee6", "vqq8EsO", "tI0iaY7", "ikM4vSk", "AgP9cVc", "Ci7wPe1",
    "t0TuIH0", "9Zz3w5d",
"VWp4dJu", "F08cTky", "c9KLjQZ", "u3cpk5s", "TtGeFjc", "Huu0sk0",
    "b1IQaNF", "DoYndaK",
"HcZW1Cg", "o9qJ5vT", "V7s79ss", "d3WqAVw", "qLk0ADl", "Maf8RyV",
    "K8E0Nwn", "ai3YTU3",
"m2cSizA", "xLFn0ED", "cBiYDW4", "Lvanch8", "MQ6MHri", "dTaQ2J8",
    "QCO3BZh", "oDGY8Vj",
"V8ouSCN", "feRphBq", "TBMe0Qe", "Hl4XGil", "vr5tj5y", "BQOicE2",
    "GZ0Zv8V", "RFpYDTH",
"gFu6GLd", "xQgBB9N", "xAjuXu5", "1gkE2th", "TSi0qX0", "eqdBrXb",
    "MqMtGPJ", "SGROVFI",
"xcmSorJ", "Ks86QxN", "nh9kb1K", "VDTeZDg", "XIWFqYs", "3yfrMRu",
    "msZyUWH", "NArNZ8E",
"qYVIzxN", "677DcTN", "1JOD4oi", "DPsY60D", "NkPYpgA", "KWWsjnY",
    "EBp6fSs", "1xaHuxr",
"cw4QVxZ", "OKhMDNg", "TTFSma4", "051Pihe", "A3qFqJ6", "m6yNLlB",
    "FqwLgvc", "RBPUWLh",
"8HvfXUU", "GQRhglT", "55OS7QN", "WaAtB8l", "bPxdU6X", "mKUwNeC",
    "aAR10qR", "H62qJm0",
"qNXBujk", "KouqtTn", "7oOy16v", "6icrj4V", "sLPMqRU", "twtFT7g",
    "KXl7Rqz", "icMVPRZ",
"woeKesr", "jEWAcMa", "Ahh1uvf", "V9YhfHY", "wJziFxo", "BSxkLrm",
    "Zw9gWBg", "433Enjv",
"aSdKAAS", "TUBWJM4", "anqlJkO", "zPCwFnb", "RWUjwYI", "3FNigdD",
    "zCrxnJh", "dyJIDx2",
"9L2Au1K", "HHg5crS", "DP2wGim", "8SYOAj5", "XNyKVoK", "YfxdqYI",
    "yMCAvtx", "IlcWdXj",
"L6Y6xgJ", "doBTyWm", "gsU1f35", "xYlUr29", "xnGGSyd", "IkuUU4d",
    "eSSV0Lw", "cwyGiVL",
"Kt4pPU8", "WcsL6mX", "SYcNqxm", "PSHmFUZ", "E2xmrPP", "xWesm42",
    "UUmwEF1", "ltbcRA4",
"ivhJB4J", "Ftwh2dP", "FDZemxo", "BEg4xed", "DqJdsgv", "WscqfkP",
    "6w2huXN", "w5MUfs3",
"Sh6WCeF", "youDNAO", "KoQ4tTX", "5tpCB8h", "OuvlC8t", "OTQAHbB",
    "lkbS57q", "ZHmemOI"
}, {
"sePK", "BgRr", "Hssn", "zDD1", "HK1M", "rb8A", "6YVh", "I96c", "
    gIXL", "KOyv", "jtVY",
"RMZw", "QwY0", "xrzL", "OfQD", "5Lfu", "v7OH", "VjdF", "6gxl", "
    JSF1", "nUNE", "J4Wq",
"xxCr", "57US", "fBsK", "8vPA", "BYxb", "BYNN", "0yHp", "44QP", "
    kq6k", "3Mvk", "d4h1",
"LXFl", "qKyp", "zbqo", "YBAJ", "SOtY", "HxC0", "NfWG", "CYAo", "
    LkU6", "718V", "I2kj",
"oSwS", "hOb8", "Pdbz", "Z1oV", "VT3J", "xz6w", "MxE2", "Hqcn", "
```

mffa", "1LZK", "4ufE",
"OF85", "GTkI", "vj4r", "ZWPb", "dGNk", "5RjS", "NmWp", "XuHg", "
ZIYr", "ndeU", "ID7x",
"shXB", "jmAh", "i2PO", "t8Bc", "fkCq", "daDL", "ANzX", "SpAX", "
WTa8", "OPIR", "KWU1",
"MbUU", "MBkq", "TMT9", "qfB3", "d8xc", "OiQL", "b5do", "eNtS", "
EO2I", "0z3m", "Y8FO",
"K0hh", "hYAH", "Xir3", "43Na", "ZtmW", "VgG7", "GQ0T", "ZGr0", "1
b9D", "QsLt", "9D9W",
"eMtm", "TC5k", "JIMN", "ahSb", "AEbq", "2wyi", "It2E", "11No", "
CWAn", "brz7", "kT5f",
"VXmu", "zT6q", "PqiN", "KR5s", "hMQy", "KXyY", "NAe0", "BIKI", "
dg3Z", "TnzL", "5Vww",
"EWwP", "IE8L", "ILEK", "zdgU", "leW5", "JWYi", "KBVE", "mI0x", "
ggbH", "liLk", "0hN2",
"xhiT", "a6za", "smpb", "dzQs", "KDdq", "jxaf", "Fhbp", "v0yX", "
Ys9M", "U8zc", "xQnU",
"e4UM", "tqxj", "sUbp", "SBNI", "7kyB", "d858", "LSan", "kBe6", "
Qry1", "9CKS", "gkc3",
"JsGR", "NojN", "22Ms", "D5sk", "L5Hf", "vPM4", "Jmnx", "qTDz", "
uwr3", "twZm", "iW4n",
"2JlS", "c2Ot", "2Nja", "faqU", "Ttns", "UmZT", "4GXV", "XHou", "
nECZ", "jdNj", "nWg3",
"kWh5", "0x6T", "4ed2", "PAAk", "TdXI", "6eAR", "MjYm", "mcDA", "
om1Y", "i5Yu", "10eM",
"Fiku", "MCxh", "r7My", "me2e", "OR1V", "EYQ1", "jUtO", "o7cV", "2
a4V", "JelY", "zZza",
"w4m8", "YUnw", "7C6p", "iceY", "NIiH", "xKqa", "pw2a", "oaaQ", "
h4No", "DLHE", "E2oO",
"7uLH", "bHsn", "XpjB", "Y0UP", "9Np6", "8TrQ", "dLjb", "BOOf", "
T46t", "xC4M", "4Qc2",
"NjDq", "lEru", "8noa", "T2Hz", "aRm5", "arO6", "UmGX", "FkF5", "
hG6o", "gKcU", "Edii",
"6QzY", "e2Ow", "H2YP", "5C4L", "o8rR", "wtk0", "wme0", "otMQ", "
LC0m", "z3mq", "21tb",
"xlJE", "O5tr", "aKga", "3Uv8", "R0Oi", "Lr70", "M9hO", "6TDb", "
lLW0", "WSQv", "GgV7",
"jjK6", "6aWZ", "5OIf", "47fC", "aRwK", "VkTK", "eg6c", "XicU", "
p8Ye", "xtqr", "sJk5",
"oeJE", "LhPh", "5kYs", "zrAD", "VWhO", "JxjX", "5n6W", "b2LL", "
LO7H", "6Jg5", "LBjn",
"R8b5", "Nw7v", "8X8Q", "pRnh", "1vH0", "rjDR", "49K7", "i71L", "
x79b", "FzvJ", "eKqi",
"aOeI", "CAzH", "MaFG", "p4Lo", "fniR", "tqUW", "abKy", "cqfV", "
R3vu", "vzud", "VdeL",
"bCpk", "MXq2", "2AlL", "7fzv", "MFKn", "VwFd", "rYhs", "L0Kc", "
83sM", "IwnU", "egRj",
"gvMx", "H4K7", "gxws", "JeBw", "kxu1", "dwrx", "0Yht", "r4Gq", "
K6qz", "2ze6", "fQUI",
"dLSW", "xjig", "r44S", "18Ns", "rWZi", "ee87", "qf3V", "1b1l", "
xCNu", "3FpQ", "bL9G",
"2zwt", "3gmA", "ZdUg", "6kwu", "yDH4", "Dz9x", "c7wM", "R5Vc", "
nbom", "m3lE", "jTwg",
"hfq0", "8q6b", "D1UP", "AGm4", "HmiN", "c8ds", "gm3W", "qqZp", "

FjWe", "sbIi", "FdoW",
"3za4", "T4pE", "hrPu", "ys0s", "Kon2", "ODuQ", "KLRu", "2Sf7", "
iE51", "Se8L", "zXT4",
"fn2w", "D0Ad", "DhM3", "WOnF", "bI03", "he2m", "dFKs", "TG7a", "
aoGC", "kAzB", "u1yA",
"PKlM", "nlKD", "A2Bw", "XiZx", "HDtv", "WfLI", "7MNt", "GyyV", "
xc1L", "G83e", "U5qz",
"hM6V", "LnOo", "y058", "tC6K", "1xXz", "vuBu", "NV5E", "htrH", "
htRV", "spra", "gYDP",
"lX1Z", "YMLs", "OJ9V", "yQVI", "ePnp", "jlJz", "87bH", "7qcw", "
mqlK", "BxV0", "zrWD",
"qPQS", "gWPU", "E0Eo", "Yl9k", "dSCU", "k1ol", "NDOQ", "crjN", "
LDHC", "ahbM", "ElTY",
"V7v9", "o4Ag", "Layo", "S9y2", "WjrH", "5kS6", "l4sG", "WEIE", "
pZDT", "6ID1", "3tj7",
"9wNG", "woO5", "DWWl", "myyd", "88D5", "SBn8", "4xoq", "6rYE", "
kbRx", "U5Kx", "dDwk",
"6T7E", "wu2j", "5ybX", "RC3H", "y2ib", "C7ec", "916d", "3Kfc", "
giED", "GwM3", "Py1I",
"3zsy", "o41b", "CVwj", "ZOtv", "nhWa", "ft0c", "HvOv", "mg0x", "
o1rC", "ID4N", "ECCV",
"DKkY", "okRP", "o8Sp", "NFbE", "GKoP", "4dJg", "At49", "El5H", "
xVuv", "ewwj", "Vtu4",
"dNSk", "9CNx", "UgXp", "zVof", "O1bf", "tqKR", "5afd", "TSta", "
19iz", "vpVA", "hfvZ",
"5PJ0", "lY1L", "Hjj7", "vFIB", "tazd", "dCYt", "nLZf", "64Vo", "
iv1s", "YU0f", "4spn",
"QjKK", "IqP7", "s8Gm", "jt6j", "3byq", "21Ea", "ugHy", "yw1O", "
QIA3", "UE9V", "cajO",
"aXGh", "NgIU", "doMi", "WuXm", "z8aj", "FvZS", "0t8I", "sHFp", "
SXC2", "yy50", "edly",
"pPdF", "eLM3", "YKwI", "sq3G", "Yu4t", "IuGq", "1Mz8", "crQV", "
rW19", "4I3j", "okVW",
"8T19", "3iee", "oyGo", "X1Zk", "x8M5", "aLhl", "2i0R", "jtdE", "
PAIl", "flfy", "cDag",
"9yiM", "YIis", "Dnvy", "6FYC", "oMry", "GNGP", "h88V", "Zb6d", "
fFMi", "7r8X", "JyQf",
"bEJQ", "XmMT", "T5Zi", "uDPa", "Tqgo", "6d0e", "arMc", "gwm9", "
SFGX", "BSEd", "3z70",
"lLrX", "5grd", "2gzf", "jpTG", "qMrd", "8Eug", "8FiZ", "3Xy9", "
twdz", "NMxK", "Ifze",
"5UPn", "rUYi", "escG", "rQYU", "7W7f", "UhPv", "Ogtj", "2Y2X", "2
bCo", "dqvW", "9wZl",
"1rvz", "NKrJ", "4iCh", "Cvb2", "n2OP", "uq8d", "mJws", "lmOV", "
Gbbi", "zvof", "boqb",
"kngv", "YnP3", "x974", "jLYl", "pGre", "NVNT", "Vsra", "jil9", "6
Mh0", "syPr", "Ud9u",
"HkL8", "yos3", "OQU1", "WMLi", "mD6D", "HhQo", "L4Qu", "3ZSF", "
qUFA", "XMmi", "gc4p",
"nntQ", "FGxC", "xzha", "uK7Q", "pau8", "nTrJ", "eaX7", "OTyW", "
mcxJ", "QOkb", "Ylig",
"KsBd", "uYNC", "Y4sc", "xaXd", "Masj", "QqpY", "eXT8", "KPh9", "
bOqH", "42I8", "P3S3",
"7rYr", "lRha", "g1GR", "jhlI", "7GbR", "GITd", "UInD", "ETwc", "

VG7U", "IVbv", "WhxY",
"0f2f", "Dce3", "sKxZ", "FzEI", "ONlt", "zSe1", "OHZR", "tAvE", "
aAM5", "nZf4", "lXv5",
"pC66", "r9XZ", "InCS", "eaMI", "5A0e", "Ftmk", "FUn9", "hVJw", "
i1rG", "h4j7", "fKYr",
"oxoQ", "kl3S", "Mhwv", "zeiw", "kVmo", "tYi6", "ORmV", "Px7x", "
zyoH", "dl9h", "qK1m",
"bq2R", "7ZMC", "yvNu", "MdNl", "C6BC", "ehvU", "1r7U", "q5qe", "4
M3e", "ekOE", "pfkO",
"bLFg", "UBh2", "cXJj", "uuCY", "gFYH", "0JZa", "VV7L", "kvOm", "
NprX", "EqqW", "WG2Q",
"hMtc", "BKOF", "6cdI", "CROP", "J9Ht", "C5YQ", "OK5G", "uwBj", "
D43j", "jioh", "9UfZ",
"rpFl", "aXnt", "qAkN", "HW2q", "v7ff", "TxRh", "qVVs", "AzqR", "
aNGc", "uAbH", "UHmI",
"Q7hM", "4XfM", "9VWL", "FNR9", "1ujF", "bI5S", "5sL0", "WfLo", "
PVj7", "gsDb", "rfOL",
"kXq0", "J2q1", "4Sty", "YcFQ", "KQze", "ruUR", "ZL0c", "qA4q", "3
m88", "PpwZ", "G6Gg",
"LKow", "llod", "I0Jv", "6k0S", "MvOB", "4a87", "eBzs", "H1p6", "
ss0H", "ca40", "B9tN",
"3neC", "cIg1", "gw72", "YkhQ", "3lrj", "L6rl", "xDf5", "UCIE", "8
vJz", "WXGL", "jPND",
"UywW", "ZBbx", "Rb2y", "8q8g", "BThL", "GGPa", "9tMO", "n21C", "
j1fe", "QHZY", "KPAI",
"4JDm", "Drjm", "4htN", "coVh", "z5gr", "5CR1", "usz2", "gLvu", "
wi2M", "xTa5", "NI4u",
"DZVt", "wJJL", "vpkI", "yYn6", "SoJW", "r10L", "fDid", "RUx5", "
I2wz", "PWeB", "qFDw",
"eEGv", "YOMf", "6YHD", "uOF7", "LNln", "maLV", "h5Lw", "Fu1S", "
HYLA", "pzis", "Defu",
"eBxf", "dw72", "zIyo", "wXZn", "1hTa", "ZBUv", "BErJ", "YEe9", "
SLtS", "upfz", "WtMH",
"EifT", "kO71", "AanA", "euOq", "TXkI", "dKXF", "q0rT", "H0LH", "
Usym", "Gehr", "Trae",
"KelN", "Fksp", "Bde3", "KbUa", "NYYz", "pQGL", "ZsCR", "9u55", "
kRb4", "ZS7Z", "RKyQ",
"HBqA", "ehII", "rHtN", "zDYE", "1gS7", "eOM9", "LqfK", "LI3r", "2
Ial", "LLmO", "QGCJ",
"N99M", "3Ytx", "mYFO", "fUEB", "F8Ue", "IyK4", "plJo", "eUT9", "
42dE", "4MGD", "sAid",
"Ytzf", "6VVk", "z3KU", "jBEJ", "bf7m", "XndL", "alR9", "avHT", "
cT6P", "22El", "hv5O",
"3VNm", "pdaC", "kAyL", "HbCO", "QWYq", "Kbmo", "TCUh", "1t7O", "
r7Pi", "rxTt", "isYm",
"oHgU", "rDCK", "fmdl", "UphB", "GOd6", "rzqS", "LCrN", "RhXQ", "
CBAC", "ah1I", "YpEj",
"aREZ", "p8vZ", "Fsr9", "8AcT", "z7r1", "hCHQ", "iLjL", "TC9v", "
JIR6", "1qdu", "9ztO",
"y3IX", "HjNm", "Pacr", "qzQ1", "u7U0", "yxnM", "5az8", "X9XA", "
qmcg", "lEsK", "WZH0",
"jNBP", "dJYp", "4Egm", "T30a", "VRdw", "rQsg", "wf7i", "yJut", "
UcOk", "aT5W", "pWeZ",
"ZLo1", "LZfV", "spDw", "xDwP", "fj7T", "IHsG", "Dqof", "z3Ik", "

```
        vcMg", "8k2o", "joQg",
      "fSDr", "rDKL", "Dc9E", "wfnl", "aqnU", "6iUG", "vnc3", "1WNg", "
        OrhP", "KNV7", "UFGg",
      "OPQW", "LWp7", "Zhur", "VcWo", "5taS", "bPz8", "dv0G", "D6FI", "
        mVS8", "TZmm", "wpGg",
      "0w7t", "bmwa", "TIXe", "21dU", "TDvO", "Ul5G", "68d9", "IsfV", "
        re73", "Zh1g", "jI46",
      "f0h0", "3CRT", "ZqCI", "PsNw", "NROo", "BPsJ", "ubBu", "kdPo", "
        vz5S", "dFUL", "iEOQ",
      "pBfH", "xlm6", "oUYA", "fet8", "jtML", "F6fq", "R6We", "MCMj", "
        JROj", "eTmP"
    }
  };

  private int shuffle_iterations = 100;

  /**
   * Test inserting all of the test sets and check if all elements are
       returned in the correct
   * order. 2 WscqfkP 3 XIKTNqG 1 XIWFqYs
   */
  @Test
  public void insert() {
    for (String[] objects : tests) {
      for (int i=0 ; i<shuffle_iterations ; i++) {
        List<String> object_list = Arrays.asList(objects);
        Collections.shuffle(object_list);
        Trie<String> trie = new Trie<>();

        // test regular insert
        object_list.forEach(trie::insert);

        // test duplicate insert
        if (i == 0)
          object_list.forEach(trie::insert);

        Collections.sort(object_list);
        assertThat(trie.match("*"), IsIterableContainingInOrder.contains
            (objects));
      }
    }
  }

  @Test
  public void find() {
    for (String[] objects : tests) {
      for (int i=0 ; i<shuffle_iterations ; i++) {
        List<String> object_list = Arrays.asList(objects);
        Collections.shuffle(object_list);
        Trie<String> trie = new Trie<>();
        object_list.forEach(trie::insert);

        // check that all objects are still in the tree
        object_list.forEach(object -> assertThat(trie.find(object),
```

```
        equalTo(object)));
      assertThat(trie.find("does_not_exist"), IsNull.nullValue());
      Collections.sort(object_list);
      assertThat(trie.match("*"), IsIterableContainingInOrder.contains
          (objects));
    }
  }
}

@Test
public void delete() {
  Random rng = new Random();
  for (String[] objects : tests) {
    for (int i=0 ; i<shuffle_iterations ; i++) {
      List<String> object_list = Arrays.asList(objects);
      Collections.shuffle(object_list);
      Trie<String> trie = new Trie<>();
      object_list.forEach(trie::insert);

      int num_deleted = rng.nextInt(object_list.size());
      int num_objects = objects.length;
      for (int j=num_deleted ; j>0 ; j--) {
        trie.remove(object_list.get(num_objects-j));
      }

      String[] subset_objects = Arrays.copyOf(objects, num_objects -
          num_deleted);
      object_list = Arrays.asList(subset_objects);
      Collections.sort(object_list);
      assertThat(trie.match("*"), IsIterableContainingInOrder.contains
          (subset_objects));
    }
  }
}

@Test
public void match() {
  Trie<String> trie = new Trie<>();
  List<String> object_list = Arrays.asList(tests[3]);
  object_list.forEach(trie::insert);

  assertThat(trie.match("W*"), IsIterableContainingInOrder.contains("
      WaAtB8l", "WcsL6mX", "WlgV8SD", "WscqfkP"));
  assertThat(trie.match("W*l"), IsIterableContainingInOrder.contains("
      WaAtB8l"));
  assertThat(trie.match("*w"), IsIterableContainingInOrder.contains("
      UWOxYpw", "d3WqAVw", "eSSV0Lw"));
  assertThat(trie.match("W*8*"), IsIterableContainingInOrder.contains(
      "WaAtB8l", "WlgV8SD"));
  assertThat(trie.match("W*a*8*"), IsIterableContainingInOrder.
      contains("WaAtB8l"));
  assertThat(trie.match("W*a*8").size(), equalTo(0));
  assertThat(trie.match("W").size(), equalTo(0));
  assertThat(trie.match("WASD").size(), equalTo(0));
```

```
    }

    @Test
    public void link() {
      Trie<String> trie = new Trie<>();
      List<String> object_list = Arrays.asList(tests[2]);
      object_list.forEach(trie::insert);

      trie.link("street.main", "person.alice");
      trie.link("street.rose", "person.carl");
      trie.link("street.euclid", "person.bob");
      trie.link("person.carl", "dog.sandy");
      trie.link("person.bob", "dog.sam");
      trie.link("person.alice", "cat.calypso");

      assertThat(trie.match("person.*:dog.*"), IsIterableContainingInOrder
          .contains("dog.sam", "dog.sandy"));
      assertThat(trie.match("dog.*:person.*"), IsIterableContainingInOrder
          .contains("person.bob", "person.carl"));
      assertThat(trie.match("street.rose:person.*:dog.*"),
          IsIterableContainingInOrder.contains("dog.sandy"));
      assertThat(trie.match("dog.*:person.*:street.*"),
          IsIterableContainingInOrder.contains("street.euclid", "street.
          rose"));
      assertThat(trie.match("cat.*:person.*:street.*"),
          IsIterableContainingInOrder.contains("street.main"));

      trie.remove("dog.sandy");

      assertThat(trie.match("person.*:dog.*"), IsIterableContainingInOrder
          .contains("dog.sam"));
      assertThat(trie.match("dog.*:person.*:street.*"),
          IsIterableContainingInOrder.contains("street.euclid"));

      trie.remove("person.bob");

      assertThat(trie.match("person.*:dog.*").size(), equalTo(0));
      assertThat(trie.match("street.rose:person.*:dog.*").size(), equalTo
          (0));
      assertThat(trie.match("person.*:street.*"),
          IsIterableContainingInOrder.contains("street.main", "street.rose
          "));
    }

    @Test
    public void toStringTest() {
      Trie<String> trie = new Trie<>();
      Arrays.asList(tests[2]).forEach(trie::insert);
      System.out.println(trie);
    }

}
```

Copyright© Ethan Toney, 2018.

Bibliography

[1] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, pages 107–141, New York, NY, USA, 1970. ACM.

[2] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. Hot: A height optimized trie index for main-memory database systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 521–534, New York, NY, USA, 2018. ACM.

[3] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[4] Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, September 1960.

[5] Kurt Maly. Compressed tries. *Commun. ACM*, 19(7):409–415, July 1976.

Vita

Ethan Toney

Education

May 2018, BSc in Computer Science, University of Kentucky

Professional Employment

- Co-Owner, Viduus Entertainment, March 2017 - Current
- Algorithm Developer, Scriyb, July 2018 - October 2018
- Full Stack Engineer, Fooji, March 2017 - April 2018
- Web Development Intern, Big Ass Solutions, December 2014 - December 2015

Academic Employment

- Undergraduate Researcher under Dr. Jaromczyk, Bioinformatics, University of Kentucky, Summer 2016
- Undergraduate Grader, University of Kentucky, August 2015 - May 2017

Publications

- Cody R. Barnes, Ethan G. Toney, and Jerzy W. Jaromczyk. 2016. Comparison of Network Architectures for a Telemetry System in the Solar Car Project. Proceedings of the Federated Conference on Computer Science and Information Systems 751–755.

Awards

- 6th in the US, 95th in the world, IEEExtreame11, 2017
- 5th in the US, 119th in the world, IEEExtreame10, 2016

Presentations

- Ethan G. Toney. Comparison of Network Architectures for a Telemetry System in the Solar Car Project. 3rd International Workshop on Cyber-Physical Systems, University of Gdansk, Gdansk, Poland, 12 September 2016.
- Cody R. Barnes, and Ethan G. Toney. Comparison of Network Architectures for a Telemetry System in the Solar Car Project. 30th National Conference on Undergraduate Research, University of North Carolina, Asheville, North Carolina, 8 April 2016.