Fast Integer Search

Cory Fox
Department of Computer Science
Florida State University
fox@cs.fsu.edu

Charles Weddle
Department of Computer Science
Florida State University
weddle@cs.fsu.edu

November 26, 2005

Abstract

This paper explores the design and implementation of the Fast Integer Search (FIS) algorithm. We discuss work previously done on the topic of integer searches and then present our own solution to this problem. Our solution uses a combination of existing data structures; the trie, the linked list and the hash table in an attempt to achieve a running time of better than O(lg n). We show our implementation compares unfavorably to the C++ STL Map yielding a performance 1000 times worse on inserts, 100 times worse on deletes, 5 times worse on predecessor and successor, and 3 times worse on query. We discuss the likely reasons behind the results and examine the future work that can be done to improve upon our FIS algorithm.

1 Introduction

Our problem is to perform integer searching in better than O(lg n). This constraint on running time includes the performance of 5 operations on a set of integers; insert, delete, query, successor, and predecessor. This problem is difficult because conventional data structures will not provide an acceptable performance for all five operations at the same time. For example, by using a hash table, the query, insert, and delete operations can be performed in constant time which is acceptable but a hash table cannot perform the predecessor and successor operations, which is unacceptable. Using a tree would allow the query, predecessor, and successor operations to be performed in O(lg n) time but this is simply not acceptable. Clearly, an unconventional data structure is needed to perform fast integer searching in better than O(lg n).

2 Background

A similar problem to fast integer search is fast string search. In fact, a binary representation of an integer is a string with an alphabet of size two. A considerable amount of research dealing

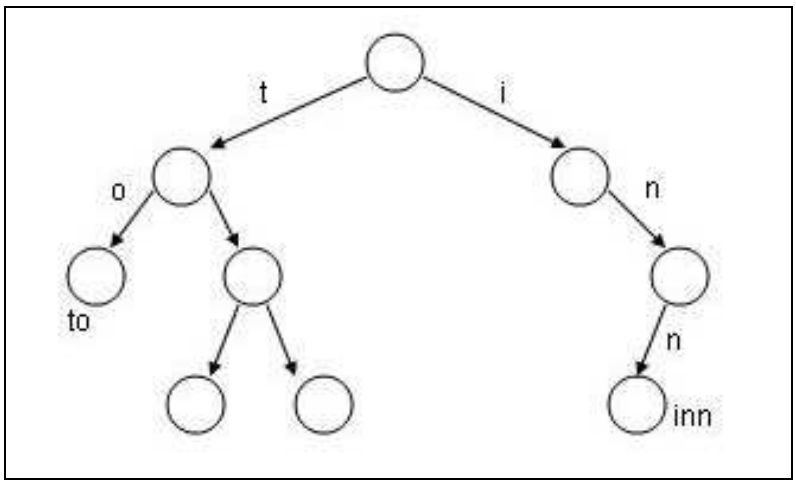

Figure 1: The trie data structure.

with fast string search has already been completed. We can use these results to help build our fast integer search algorithm [1, 2, 4]. Specifically, much of the research work completed to this point used a data structure called a trie to speed up search times. Figure 1 depicts the trie data structure. A trie is a data structure that stores a set of strings using an alphabet containing k symbols in a k-ary tree. Each string corresponds to a unique path in the tree [2]. Figure 1 shows how a trie stores the word **to** and **inn** using the English alphabet.

A LC-Trie is a variant of a trie that compresses the trie levels. This is done by replacing the i highest complete levels with a single node of degree 2^i [2]. Nilsson and Karlsson used an LC-trie to achieve a trie of average height lglg(n) [4]. This work looks promising for our algorithm because a trie depth of lglg(n) would potentially mean performing queries in O(lglg n) time. Unfortunately, since our data structure must be dynamic for inserts and deletes LC-tries cannot be used because single update operations might cause a large and costly restructuring of the trie [5].

Since a trie data structure by itself is not good enough as a solution to our problem, we decided that a combination of data structures must be used. Table 1 shows the upper bounds for several common data structures. A hash data structure is able to perform query, insert and delete in constant time. This would be acceptable as a solution to our problem for those three operations. But the hash table would not perform adequately for the predecessor and successor operations. Something else is needed.

We hypothesize that the combination of a hash table with a linked list and a trie together can perform better than O(lg n) for the insert, delete, predecessor, successor, and query operations. The combination of these data structures would provide for a running time of O(lg w) for the predecessor and successor operations where w is the length of the binary representation of the integer. Therefore, using hash tables, a trie, and a linked list, all five operations can be bounded by O(lg w) as shown in table 1.

Data Structure	Query	Predecessor	Successor	Insert	Delete
Hash	1	n/a	n/a	1	1
Linked List	n	n	n	n	n
Binary Tree	lg(n)	lg(n)	lg(n)	lg(n)	lg(n)
Trie	lg(w)	lg(w)	lg(w)	lg(w)	lg(w)
LC-Trie	lglg(w)	lglg(w)	lglg(w)	n/a	n/a
FIS	1	1	1	lg(w)	lg(w)

Table 1: Upper bounds for common data structures and FIS.

3 Solution

Our Fast Integer Search (FIS) algorithm can achieve a running time of h = lg(w) or less for each of the five operations where h is the height of the trie and w is the bit width of the integer. The FIS algorithm uses three distinct types of data structures to achieve this running time; a hash table, a doubly linked list and a binary trie. The binary trie data structure holds the binary representation of each integer. The height of the trie is the bit width, w, of the integers being inserted. A hash table is created for each level in the trie so that each binary prefix or the full binary value of an integer can be accessed in constant time. The linked list contains an element for each inserted integer and allows for access to an elements predecessor or successor in constant time. The leaf nodes in the trie have a pointer to the corresponding element in the linked list for that integer. The combination of the trie, linked list, and hash tables allow for insert and delete to be performed on average in O(lg w) time. Figure 2 depicts these data structures as used in our solution.

At the start of the algorithm the binary trie, the linked list and the hash tables are empty. When a value is inserted into the trie the leaf node will hold a pointer to a corresponding entry in the linked list. After this insertion a value will exist in each hash table for every prefix for the inserted integer. After this insertion an element will be added to the linked list for that integer that will hold pointers to its predecessor and successor's linked list entries and a pointer to the leaf of the trie. When an integer is deleted the corresponding string path in the trie for that integer is removed including the leaf node. Also, each appropriate entry in the hash tables for this integer are removed and the element for this integer in the linked list is also removed. A detailed description of each operation follows.

Predecessor and Successor

The FIS Predecessor and Successor operations can be performed by using the hash tables to find the greatest prefix node in the trie for that integer. From that node in the trie, the closest leaf node to that integer can be found. The corresponding linked list entry for this closest integer can be found using this leaf node. Now that the element in the linked list is found the predecessor and successor can be determined. If the value in the entry is larger than the integer you have reached the successor. To find the predecessor follow the backward pointer. If the value in this leaf node is less than the integer you have found the predecessor. To find the successor follow the forward pointer.

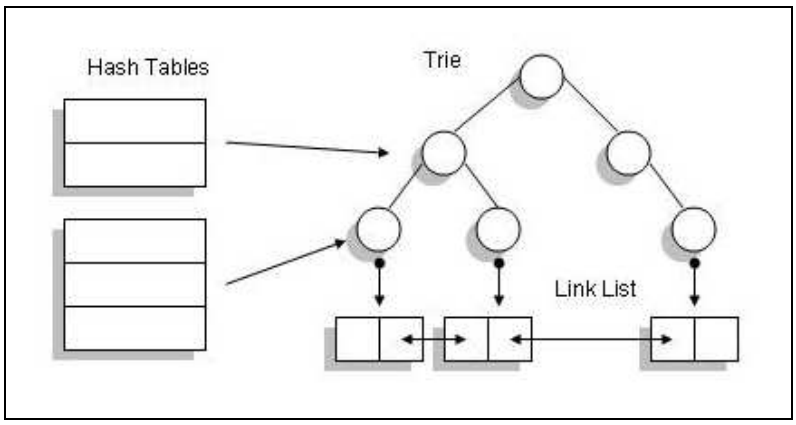

Figure 2: Data structures used in fast integer search.

In the worst case this will have a run time of O(w). This runtime will decrease as more entries are added, approaching an average runtime of better than O(w). The reason for this is as more entries are added there are more longer common prefixes, which means the average depth traveled is less.

Insert

The first step in the FIS insert operation is to find the predecessor or successor of the integer to be inserted as outlined above. Using the predecessor or successor leaf node found in the trie retrieve the associated linked list element. This linked list element is used to insert the integer into the linked list. The next step is to find the greatest common prefix of this integer in the trie. From the greatest common prefix node walk down the trie adding the necessary edges to the trie and the relevant entries into the hash tables. Once the leaf level is reached, the associated linked list element can be retrieved. In the worst case this will have a run time of O(w). As with the predecessor and successor operations, this runtime will decrease as more entries are added, approaching an average runtime of better than O(w).

Delete

The FIS delete operation is performed by first hashing to the leaf node in the trie associated with the integer to delete. Using the linked list pointer in this leaf node, the linked list entry for this integer can be deleted. The next step is to find the greatest common prefix of this integer in the trie. Delete the edges from this greatest common prefix to the leaf node. Also delete the corresponding entries from the hash table. In the worst case this will have a run time of O(w). This runtime will decrease as more entries are added, approaching an average runtime of better than O(w).

Query

The FIS query operation can be performed using the hash table. If an entry does not exist in the hash table for that integer then the integer is not there, otherwise it is. This operation can be done in O(1) time.

Command	Description
i *integer*	Inserts an integer into the FIS data structures.
d *integer*	Deletes an integer from the FIS data structures.
p *integer*	Displays the predecessor for an integer.
s *integer*	Displays the successor for an integer.
q *integer*	Displays whether or not an integer exists.
t	Executes the unit test.
z	Displays the contents of the FIS data structures.

Table 2: FIS user interface commands.

4 Implementation

After designing the solution to the Fast Integer Search problem we implemented the solution in code. One of the most important decisions to be made when implementing a solution is picking a language. Each language has certain advantages and disadvantages. Our first attempt to implement the solution was in Java but ultimately used C++.

Java offered several advantages but had some short comings that could not be overlooked. We had hoped to use existing common data structures available in the Java libraries to avoid redundant coding and reduce development time. In Java there are already existing implementations of the linked list and hash map data structures. It turns out that the Java implementation of the linked list has a bound of O(n) for insertion in all cases, which is simply not acceptable. As well, as much as Java has to offer in pre-built data structures, it does not currently have the trie data structure. Without the availability of these data structures, the interpreted nature of the Java Runtime Environment, which offers slow overall performance, made us look for a different language for implementation.

The solution was ultimately implemented in C++. C++ was chosen mostly for performance reasons. Compiling natively offers fast execution, which the C++ compiler does do. As well, by choosing C++, we were able to use the GNU MP library [3] that supports large bit length integers. For example, by using the MP library, we were able to work with 128 bit length integers.

The first component to the FIS solution is the simple user interface that drives the five operations. This component is implemented in the fis.h and fis.cpp source code files. The FIS user interface provides a simple command line menu for the five integer operations. Table 2 shows the different options offered by the FIS user interface.

The user interface component executes the five operations by making calls into an FIS application programming interface (API). This API hides the details of execution of each operation from the user interface and the FIS unit test. The FIS API is implemented in the source code files operations.h and operations.cpp. The FIS API exposes two methods for each operation. The first method takes a primitive integer type as its only argument. This method is used by the user interface. Since it takes a primitive, the maximum bit length for the integer is bounded by the hardware architecture and operating system being used. For example, on an Intel Pentium 4 running Linux, the maximum bit length for a primitive type is 64 bits. To get around this

Method	Description
insert	Inserts the primitive type integer.
_insert	Inserts the FIS element that contains a GNU MP type.
delete	Deletes the primitive type integer.
_delete	Deletes the FIS element that contains a GNU MP type.
predecessor	Finds the predecessor for the primitive type integer.
_predecessor	Finds the predecessor for an FIS element that contains an GNU MP type.
successor	Finds the successor for the primitive type integer.
_successor	Finds the successor for an FIS element that contains an GNU MP type.
query	Queries for the primitive type integer.
_query	Query's for an FIS element that contains an GNU MP type.

Table 3: FIS API exposed methods.

limitation, a second method was implemented for each operation that takes a GNU MP type as its only argument. This second method allows for very large bit lengths to be used. It should be noted that all code below the FIS API uses the GNU MP types. The first method for each operation simply turns the primitive type into a GNU MP type. Table 3 shows the methods on the FIS API.

The FIS data structures deal with one primary element type called the FIS element. This element is what gets inserted, deleted, queried for, etc. The FIS element is implemented in the fis_e.h source code file. Most importantly, the fis_e structure contains two variables. One is the actual integer that is of GNU MP type. Because this is of the GNU MP type, it can be of varying bit length. The second variable this structure defines is a pointer to a linked list element. By having a pointer into the linked list, the predecessor and successor to this integer can be found by looking at the previous and next linked list elements. Lastly, the fis_e.h header file defines the bit length being used for the integers.

The three data structures implemented for the FIS solution are the linked list, the hash table, and the trie. The linked list is implemented in the ll.h, ll.cpp, ll_e.h and ll_e.cpp source code files. The ll.h and ll.cpp files define the API into the linked list. The linked list API exposes the basic linked list operations that include insert, delete, get, etc. The ll_e.h and ll_e.cpp source code files implement the actual linked list elements that are stored in the linked list. This ll_e structure contains a pointer to the next and previous linked list elements as expected. This structure also contains a pointer to an FIS element as described above so that the integer can be retrieved for any linked list element.

The hash table data structure built for the FIS solution is implemented in the hash.h, hash.cpp, hashtable.h, hashtable.cpp, hashtable_itr.h, hashtable_itr.cpp, and hashtable_private.h source code files. The hash.h and hash.cpp source code files implement the API into the hash table. This includes the basic hash operations like insert, delete, and query but it also includes one vary important operation critical to the FIS solution. That operation is finding the greatest common prefix for an integer. The FIS solution requires that a hash table be kept for every level in the trie. The find greatest common prefix hash operation searches in O(lglg n) time through the trie level hash tables for the matching prefix. This operation achieves O(lglg n) time by dividing the bits for the integer into greater or lesser sections, based off of bit length, and querying for that bit string in

the appropriate hash level. Finding the greatest common prefix is needed for the insert, predecessor, and successor operations. The hashtable.h, hashtable_private.h and hashtable.cpp source code files implement the hash table, while the hashtable_itr.h and hashtable_itr.cpp files implement an iterator for the hash table.

The trie data structure built for the FIS solution is implemented in the trie.h and trie.cpp source code files. The trie.cpp and trie.h files offer the functions trie_insert, trie_delete and trie_closest to manipulate the trie. For FIS we used a path compressed trie which is similar to a regular trie except it collapses one way branches into a single node. The path compressed trie is implemented through the recursive use of a simple structure for each node. The trie_e structure contains the node's prefix value, prefix length, pointers to its children and a pointer to an fis_e element. The prefix value is the prefix that all descendants of this node will have. The prefix length is the length of that prefix. The pointers to the nodes children are indexed 0 and 1 to represent the next bit after the prefix being a 0 or 1. These pointers are NULL if this is a leaf node. The pointer to an FIS element is null if this is an internal node.

When a value is inserted into the trie it is passed the trie node with the greatest common prefix, the length of that prefix and the FIS element to be inserted. The direction for the placement of the new node is determined by looking at the bit after the greatest common prefix for the integer to be inserted. This direction is called the new direction, the other direction is the old direction. Next, two new nodes are created. One of these nodes is set to the values contained in the greatest common prefix node. This node is placed in the child corresponding to the old direction and the appropriate hash tables are updated. The other created node points to the FIS element with its prefix set to the value being inserted and its prefix length set to the bit length. This node is placed in the child corresponding to the new direction and the appropriate hash tables are updated.

When a value is deleted from the trie the trie is passed the trie node that contains the FIS element to be deleted. Using the value of the FIS element entries in the hash table pointing to this node are deleted until the parent node is reached. Once the parent node is reached the values of the child that is not being deleted are copied into the parent node. The prefixes and hash tables are updated accordingly and the appropriate children are deleted.

For the closest function the trie is passed the integer and the greatest common prefix trie node. If the bit after the greatest common prefix is 1 the algorithm follows the trie to a leaf, always following the child associated with 1. Once it reaches a leaf node, it returns the FIS element associated with this leaf node. This is the element that immediately precedes the integer. If the bit after the greatest common prefix is 0 the algorithm follows the trie to a leaf, always following the child associated with 0. Once it reaches a leaf node, it returns the FIS element associated with this leaf node. This is the element that immediately follows the integer.

Now that the general components and data structures have been discussed, it is important to understand how these pieces interact with each other in the implementation. The three data structures are glued together through the use of the following three structures; fis_e, ll_e, and trie_e. The fis_e represents the integer. For each insert operation a new fis_e element is allocated and for each delete operation an fis_e element is deallocated. The _insert method on the FIS API first finds the greatest common prefix for this integer already in the hash tables. This is done by calling the hash_find_gcp method on the hash API. This returns a trie_e element that is a node in the trie and

represents the greatest common prefix for this integer. This trie_e element is then passed to the trie_insert method on the trie API. This method inserts the fis_e element into the trie using the trie_e node as the starting point. The trie_insert method returns the closest leaf node in the trie to this integer. This closest node is of type fis_e and contains a pointer into the linked list for this closest integer. The fis_e element is then inserted into the linked list using this closest fis_e element by calling the ll_insert_at method on the linked list API. Finally, the fis_e element for the inserted integer is inserted into the hash table by calling the hash_insert method on the hash API.

The _delete method on the FIS API first finds the fis_e element in the hash table for the integer to be deleted. This fis_e element is then used to delete the appropriate leaf node and corresponding path nodes in the trie by calling the trie_delete method on the trie API. Next, the fis_e element is deleted from the linked list by calling the ll_delete method on the linked list API. Finally, the fis_e element is deleted from the hash table by calling the hash_delete method on the hash API.

The _predecessor method on the FSI API first finds the greatest common prefix in the trie by calling the hash_find_gcp method on the hash API. This returns a node in the trie that represents the greatest common prefix in the trie for this integer. The closest leaf node to the integer is found in the trie using this greatest common prefix trie node by calling the trie_closest method on the trie API. The trie_closest method returns back an fis_e element that represents the closest integer to the integer being searched. The closest integer is compared to the integer in the _predecessor method for greater or lesser value. If the closest integer is less than the integer, then it is the predecessor. If the closest integer is greater than the integer, then the predecessor is simply found by calling the ll_get_prev method on the linked list API that returns the previous fis_e element in the linked list. The returned fis_e element from ll_get_prev is the predecessor to the integer.

The _successor method on the FSI API works exactly like the _predecessor method except that it finds the successor. The _query method on the FSI API simply calls the hash_query method exposed on the hash API that looks in the hash table for the existence of this integer.

To be able to compare the FIS solution to map, a map had to be implemented using the same language and set of libraries as the FIS implementation. The map implementation uses the same user interface code as that of the FIS implementation. The map user interface is implemented in the mis.h and mis.cpp source code files. The map user interface exposes the same set of commands as that of the FIS implementation. Like the FIS implementation, the map implementation defines a common data type called mis_e that contains a variable of type GNU MP. This mis_e data type is defined in mis_e.h. By using the GNU MP library in the map implementation, it too can handle very large integer bit lengths. The map implementation implements a generic map API that hides the details of the operations from the user interface and unit test like the FIS implementation. This map API also exposes two methods for each operation so that primitive types and GNU MP types can be used. The map API is implemented in the operations.h and operations.cpp source code files. For the actual map, the C++ map STL was used. The operation methods on the map API simply make the appropriate map STL calls. The map implementation was kept as similar to the FIS implementation as possible so that comparison would be fair.

Table 4 is a listing of each source code file in the FIS and map implementations along with the corresponding line count for each file. In total, 3503 lines of code were written for the implementation of the FIS solution.

Source File	Description	Line Count
fis.cpp	FIS user interface.	244
fis.h	FIS user interface.	28
fis_e.h	FIS fis_e element definition.	25
fis_shift.cpp	Bit shifting for GNU MP.	35
fis_shift.h	Bit shifting for GNU MP.	12
hash.cpp	Hash API.	418
hash.h	Hash API.	53
hashtable.cpp	Hash table implementation.	274
hashtable.h	Hash table implementation.	199
hashtable_private.h	Hash table implementation.	85
hashtable_itr.cpp	Hash table iterator implementation.	190
hashtable_itr.h	Hash table iterator implementation.	112
ll.cpp	Linked list API.	223
ll.h	Linked list API.	26
ll_e.cpp	Linked list implementation.	82
ll_e.h	Linked list implementation.	31
operations.cpp	FIS operation API	397
operations.h	FIS operation API	25
trie.cpp	Trie API and implementation	332
trie.h	Trie API and implementation	28
mis.cpp	Map user interface.	242
mis.h	Map user interface.	28
mis_e.h	Map mis_e element definition.	24
operations.cpp	Map operation API	362
operations.h	Map operation API	28

Table 4: FIS API exposed methods.

5 Evaluation

To compare our FIS solution with map, both the FIS implementation and the map implementation had to be tested in the same way. To do this a unit test was implemented in each of the operations.cpp files for the FIS implementation and the map implementation. This unit test is the same for both. The unit test starts off by inserting 10,000 integers. The unit test then finds the predecessor for 10,000 different integers, then the successor for 10,000 different integers, then queries for 10,000 different integers. Finally, the unit test deletes all 10,000 inserted integers. The test keeps the execution time for each of the five operations tested and prints out the time in milliseconds between each test. The unit test will work with varying size bit lengths. This is done by using the GNU MP random number generator. The GNU MP random number generator takes in the desired bit length and produces GNU MP integers with that bit length. These GNU MP integers can be directly passed into the FIS and map API's via the second method for each operation that takes the GNU MP type as its only argument. To help test performance, the 10,000 GNU MP integers are generated before each operation test. This was done because the GNU MP random method takes a considerable amount of time and negatively influenced the results.

Both the FIS implementation and the map implementation were evaluated on an Intel Pentium 4 processor with 1GB of RAM. The evaluation was completed by averaging four unit test runs at 32, 64, and 128 bits for each of the operations.

Figure 3 shows the results for the insert operation for 32 bit, 64 bit, and 128 bit lengths for both FIS and map. Next, Figure 4 shows the results for the delete operation for 32 bit, 64 bit, and 128 bit lengths for both FIS and map. Figure 5 shows the results for the predecessor operation for 32 bit, 64 bit, and 128 bit lengths for both FIS and map. Figure 6 shows the results for the successor operation for 32 bit, 64 bit, and 128 bit lengths for both FIS and map. Lastly, Figure 7 shows the results for the query operation for 32 bit, 64 bit, and 128 bit lengths for both FIS and map.

The results show that on insert FIS was 70 times worse than map for 32 bit length integers, 215 times worse for 64 bit length integers, and 667 times worse for 128 bit length integers. On delete FIS was 11 times worse than map for 32 bit length integers, 27 times worse for 64 bit length integers, and 67 times worse for 128 bit length integers. For predecessor FIS was 4 times worse than map for 32 bit length integers, 5 times worse for 64 bit length integers, and 9 times worse for 128 bit length integers. For successor FIS was 6 times worse than map for 32 bit length integers, 7 times worse for 64 bit length integers, and 11 times worse for 128 bit length integers. For query FIS was 3 times worse than map for 32 bit length integers, 4 times worse for 64 bit length integers, and 4 times worse for 128 bit length integers.

6 Discussion of Results

In our paper we hypothesized that the FIS would outperform the C++ STL map on the operations of insert, delete, query, predecessor and successor. However, in our results section it is apparent that our FIS implementation runs slower than the map STL. After analyzing our FIS program we identified several problem areas in both our design decisions and implementation.

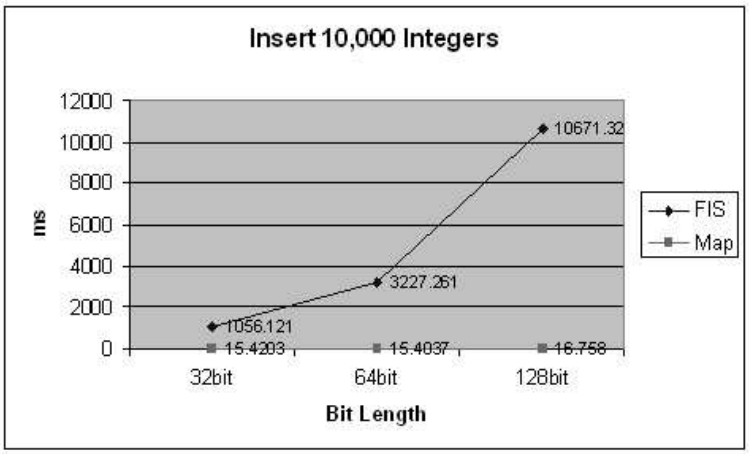

Figure 3: Insert operation at 32, 64, and 128 bit lengths.

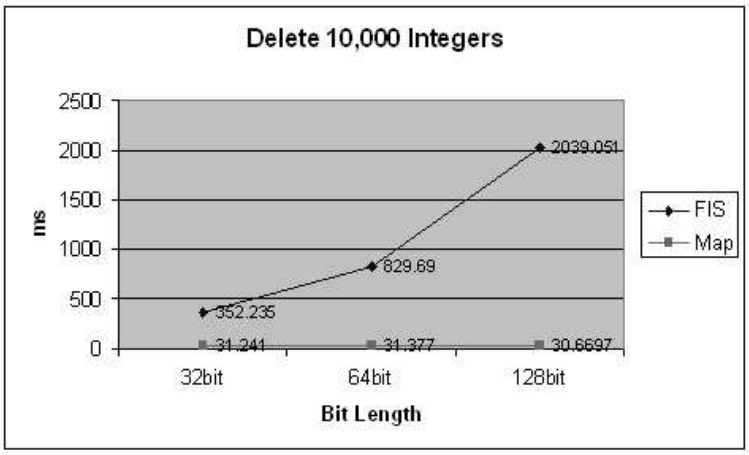

Figure 4: Delete operation at 32, 64, and 128 bit lengths.

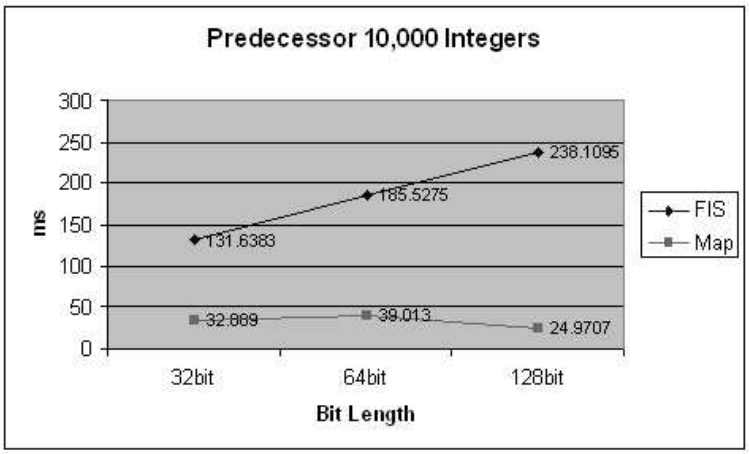

Figure 5: Predecessor operation at 32, 64, and 128 bit lengths.

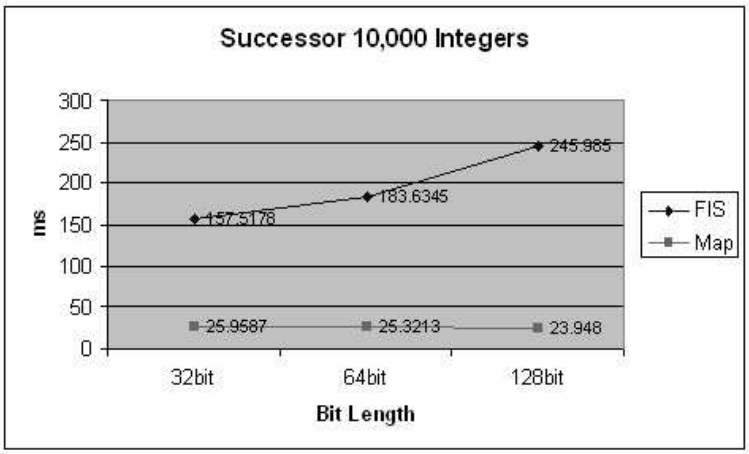

Figure 6: Successor operation at 32, 64, and 128 bit lengths.

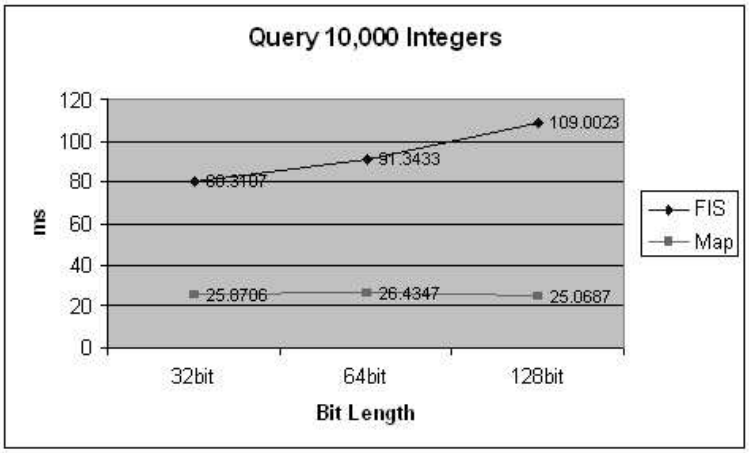

Figure 7: Query operation at 32, 64, and 128 bit lengths.

In our design we hoped to keep insert and delete operations limited to O(lg w) time complexity but since we need to keep an updated hash table for every level of the trie the bound on these functions is actually O(w). The bound O(lg w) is an average case and not achieved by our test cases. The insert operation must update the hash tables representing prefixes greater than the greatest common prefix. Since our test case inserts a small fraction of the maximum number of entries that can be inserted into the trie the greatest common prefix is often very small meaning many hash table updates must be performed. Because of this, we only see the worst case performance for our insert operation. Even if the number of elements in the trie were increased we will still see slower performance due to other sources, such as the GNU-MP library.

The GNU MP library is an open source multi-precision arithmetic library. Since our performance bounds were based on the number of bits of our integer inputs, we used the GNU MP library to test integers of arbitrary bit length. This library often achieves impressive speeds for arithmetic operations but the library lacks some functionality required by our program. Our initial implementation of FIS relied heavily on bit shifting to view prefixes of integers and since the GNU MP library does not support bit shifting we were required to write our own bit shift functions. Since the library offers only a few bit operations and we know very little about their low-level implementation it is difficult to determine how efficient our bit shifting function is. Since this function is called frequently, it became a limiting factor in the speed of our algorithm.

In our test cases even our query function is beat by the STL map algorithm. To perform a query the FIS algorithm hashes straight to the element to check for its existence. This is done in O(1) time. The map algorithm on the other hand traverses a red black tree to find if the element exists. This is done in O(lg n) time. The fact an O(1) algorithm was beat by an O(lg n) algorithm made us look closely at the hash function. We came to the conclusion that to uniquely hash a w bit number, w operations have to be performed. Normally when discussing the running times of algorithms the effect of the bit length of the inputs is overshadowed by other larger constraints. However, in FIS, the running time of the hashing function actually becomes a limiting factor in our algorithm. In the scope of FIS, hashes do not run in O (1) time they run in O (w) time. Also GNU MP does

not have its own hashing function for arbitrary precision integers so we had to write our own hash function. This function used our bit shifting function as well and like the bit shifting function it is difficult to say how efficient the hash function is.

We hoped to see the best results from the predecessor and successor operations since by using hash tables we would be able to move quickly down the tree. However we mentioned earlier that the hash function must perform an operation on every bit in the integer we are searching for. This means the hash tables actually yielded no performance increase. The traditional traversal of a trie could be be performed just as quickly if not faster.

When testing with integers of bit length 32/64/128 we hoped to see a relative increase in speed as the length increased. When the bit length was increased we only saw a slowdown in all the FIS operations. The reason for this is that in our algorithm, increased bit length increases the number of hash table lookups and updates as well as the number of shifts required for those operations. When the bit length was increased there was no noticeable change in the STL map speed. This is because the STL map speed constraint is on number of entries and not the number of bits in their representation.

7 Future Work

After testing our implementation of the FIS algorithm there is still work that could be done to improve FIS. A more in depth code profiling could be performed to find and improve problem areas of the code. For example we have identified that 90 percent of the time it takes to execute the insert operation is spent in the trie_insert method. With a more in depth analysis we can use information like this to improve the performance of FIS. Our algorithm uses a path compressed trie to avoid the unnecessary construction of nodes. A similar improvement can be made by making this a path and level compressed trie. As mentioned earlier we believe that the lack of GNU MP support for the vital functions of bit shifting and hashing for GNU MP integers causes heavy execution delays. Perhaps there is an alternative way to utilize arbitrary size integers without the performance penalties inherent in the GNU MP library. Further work would involve a new design of the FIS algorithm that does not rely as heavily on hash tables for look ups. With the level and path compressed tries, average trie height should be less lg n. This means that on average, all five operations should be performed in less than O(lg n).

8 Conclusion

The exploration for an integer search algorithm faster than O(lg n) is not a trivial endeavor. We learned several lessons in our exploration for a solution to the fast integer search problem. At a high level, we learned that an algorithm that looks good on paper may not transfer well to code. In our case, our solution on paper appeared to meet the requirements asymptotically but as our results show, this solution was crippled by implementation details. Our empirical evaluation of our solution produced worse results on all five operations compared to the C++ map STL. We also learned that mistakes in design may not be exposed until implementation. We feel that there

may be a better solution that does not use a hash table. We did not come to this conclusion until we completed our empirical evaluation. In addition to these lessons, we learned that constraints placed on the design can have negative impact on the results. For example, our solution had to support 128 bit length integers. This constraint made us use the GNU MP libraries. We did not take into consideration the limitations of the GNU MP libraries into our design. As a result, our implementation had to make up for these limitations and these components of the code were costly. Even though the results show that our solution to the fast integer search problem does not perform better than O(lg n), we still feel that this has been a worthwhile endeavor. We take away the lessons learned and a greater understanding of the fast integer search problem domain.

References

[1] ANDERSSON. Faster deterministic sorting and searching in linear space. In *FOCS: IEEE Symposium on Foundations of Computer Science (FOCS)* (1996).

[2] ANDERSSON, A., AND NILSSON, S. Improved behaviour of tries by adaptive branching. *Information Processing Letters 46*, 6 (1993), 295–300.

[3] MP, G. www.swox.com/gmp/.

[4] NILSSON, S., AND KARLSSON, G. Ip-address lookup using lc-tries, 1999.

[5] NILSSON, S., AND TIKKANEN, M. Implementing a dynamic compressed trie. In *ProceedingsWorkshop on Algorithm Engineering* (Saarbrucken, Germany, 1998), K. Mehlhorn, Ed., pp. 1–3.