Implementing a Fusion Tree in C++

Rogério A. Guimarães Jr.
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, 02139
`rjunior@mit.edu`

Abstract

A Fusion Trees is a static Data Structure that allows for predecessor queries in constant time on a set of constant size of numbers, but that requires significant large word sizes to be implemented in a computer that follows the standard word ram model. In this work, we provide the C++ code and reference to our implementation of fusion trees that performs predecessor queries with a constant number of calls to standard arithmetic operations of a general Big Integer.

1 Introduction

The idea that predecessor queries could be processed in constant time in a set with a constant maximum size was first proposed by Ajtai et al. (1984), which later inspired Fredman & Willard (1993) to develop the *Fusion Tree*. A Fusion Tree is a static data structure that can answer predecessor queries in constant time but has a size constrained to $O(w^{1/5})$ elements, in which w is the word size of the computer in which it is implemented, in a standard word RAM model.

Due to its limited maximum size, the role most commonly attributed to a fusion tree is being a node in a B-tree. A B-tree of branching factor $w^{1/5}$ and n elements can answer predecessor queries in $O(\log_w n)$ if it uses fusion trees as its nodes, which means that the time to answer the query can be reduced by a factor of $\log w$.

In our work, we propose a C++ implementation of fusion tree that relies on a general big integer class. Our fusion tree imports this class and performs its predecessor queries using a constant number of calls to basic arithmetic operations of this class. We provide a slow implementation for the big integer class using a standard C++ bitset in order to show the correct functioning of our fusion tree, even though its slow operations make our fusion tree slower than a standard C++ `set`. However, it is a template ready for change in case new computer architectures allow for a faster implementation of arithmetic operations in large integers.

To facilitate the usage of our fusion tree as a B-Tree node, we also provide an Environment class, which will simply hold important constants that define the fusion tree behavior. The environment will hold, for example, initialization constants given by the user, such as the assumed word size of the computational environment and maximum fusion tree capacity. Furthermore, it will also pre-calculate important constants that are used in the predecessor query and that depend only on the initialization constants, and thus can be shared by all of the fusion trees in a B-Tree.

In this report, we will first provide a detailed documentation of our code. We will explain the necessary public methods for a big integer class to work with our fusion tree, as well as the functioning of the public methods of our fusion tree and environment class. Then, we will briefly review the structure and functioning of a fusion tree in order to explore a bit of our implementation and design choices.

2 C++ Reference

In this section of the report, we will provide documentation for our code, which can be found in our repository on `https://github.com/6851-2017/fusiontree.git`.

We will explain the public functions and operators needed from the big integer class, in case the reader wants to use another or implement her own. Then we will show how to declare an environment and a fusion tree and explain explain the public methods available in our fusion tree, as well as show a code example.

2.1 Big Integer

This subsection provides documentation to the `big_int` class that is imported by the `fusiontree` class. The following public methods signatures can be found in the file `big_int.hpp` and are used by our `fusiontree` class. In case the reader wants to use his own big integer class, she needs to provide all these public methods and operators in a class with the same name. The other public methods and operators present in `big_int` are not necessary for `fusiontree`.

- `big_int(int x = 0);`

  Class constructor. Takes an `int x` as input (default is zero) and returns a `big_int` with the value of `x`.

- `operator int() const;`

  Returns an `int` with the value of the `big_int` if it lies between the value bounds of an `int`.

All the following operators must perform the same operation they do in a standard C++ `int`, which can be found on `https://www.cplusplus.com/doc/tutorial/operators/`.

- `big_int operator~() const;`
- `bool operator<(const big_int x) const;`
- `bool operator>(const big_int x) const;`
- `bool operator==(const big_int x) const;`
- `bool operator!=(const big_int x) const;`
- `big_int operator<<(const int x) const;`
- `big_int operator>>(const int x) const;`
- `big_int operator|(const big_int x) const;`
- `big_int operator&(const big_int x) const;`
- `big_int operator^(const big_int x) const;`
- `big_int operator-(const big_int x) const;`
- `big_int operator*(const big_int x) const;`

2.2 Environment

The environment class is very simple to use and is defined in the file fusiontree.hpp. An instance of an environment is used to inform the fusion tree of some constants of the computational environment, as well as pre-calculate and keep many constants that depend only on the computational environment and that are used by the fusiontree class in its operations. The idea is that a B-Tree should have a single environment instance and pass it to all of its fusion trees. The only public function of environment that the user must use is the constructor, which can be seen below:

-
  ```
  environment(int word_size_ = 4000, int element_size_ = 3136,
              int capacity_ = 5);
  ```

  wordsize_: The maximum size (in bits) of the big_int class, which should be the word size assumed for the computer. Must be greater or equal to (capacity_)^5+(capacity_)^4, so that the process of finding the *mask m* can be done without extrapolating the number of bits in a word.

  element_size_: the size (in bits) of the maximum element that can be stored in the fusion tree. Must be a perfect square, which helps implementing the *most significant bit* operation. Must also be greater or equal to (capacity_)^5, which allows that the *approximately sketched* version of all the elements stored in the tree fit in a single big_int.

  capacity_: The maximum number of elements that can be stored in a fusion tree. Thus, it is also the branching factor of a B-Tree that uses fusion trees as nodes.

The default arguments given in the environment allow for a fusion tree that can correctly function and store up to 5 elements. The following command will use the environment constructor to create a pointer to an environment with the arguments passed.

```
environment *env = new environment(word_size, element_size,
                                   capacity);
```

2.3 Fusion Tree

The fusiontree class is defined in the file fusiontree.hpp and is very simple to use. A fusiontree class has the following public methods that can be called by the user:

-
  ```
  fusiontree(vector<big_int> &v_, environment *my_env_);
  ```

  Class constructor. Takes as input a reference to a vector<big_int> &v_ and a pointer to an environment *my_env_ and returns an instance of a fusiontree with the characteristics defined by *my_env_ that contains the elements stored in v_. The length of v_ cannot exceed the capacity with which *my_env_ was initialized.

-
  ```
  const int size() const;
  ```

  Returns the size of the fusiontree instance, *i.e.*, the number of elements stored in it, in constant time.

-
  ```
  const big_int pos(int i) const;
  ```

  Returns the element that occupies position i in the fusiontree instance, in increasing order and starting from zero, in constant time.

-
  ```
  const int find_predecessor(const big_int &x) const;
  ```

  Returns the position of the largest element in the fusion tree that is not larger than x, or -1 if there is no such element.
  Uses a constant number of basic arithmetic operations on instances of big_int to answer the predecessor query. Therefore, its time complexity depends on the complexity of big_int arithmetic operators, and will take constant time if these operators also take constant time.

Notice that the fusiontree is a static, immutable data type.

2.4 Make File

In order to use the classes presented in a program, the user must compile the code using the `Makefile` script that is present in the repository. The user must first create a new C++ file in a directory with the files `big_int.hpp`, `big_int.cpp`, `fusiontree.hpp`, `fusiontree.cpp`, and `Makefile`. This file must import `big_int.hpp` and `fusiontree.hpp` and have a `main` function, as shown below:

```cpp
#include "big_int.hpp"
#include "fusiontree.hpp"

int main() {
    // your code here...
}
```

Then, the user must open the terminal in the directory of these files and execute the `Makefile` script. In a MacOS environment, this is simply typing the following command in the terminal:

```
$ make
```

This command will create all the necessary files for the compilation and execution of the user program. Among the generated files there will be an executable called `main.exe`, which runs the `main` function in the user code. To execute it, the user must type in the terminal the following command:

```
$ ./main.exe
```

The `Makefile` script also offers two other commands that can be useful:

- `$ make clean`

  Removes all the files generated by a previous compilation through a `make` command.

- `$ make format`

  Formats all source code according to Google's format for C++.

2.5 Example

Although its implementation can be quite complicated, using it is fairly simple and relies only on its four public methods. In our repository, the file `example.cpp` can be found and contains a brief example of how to construct and query a fusion tree. Its `main` function can be seen below:

```cpp
int main() {
  vector<big_int> small_squares;
  for (int i = 1; i <= 5; i++) {
    small_squares.push_back(i * i);
  }

  environment *env = new environment;
  fusiontree my_fusiontree(small_squares, env);

  int idx1 = my_fusiontree.find_predecessor(3);
  int idx2 = my_fusiontree.find_predecessor(9);
  int idx3 = my_fusiontree.find_predecessor(0);

  cout << "Fusion Tree size:" << endl;
  cout << my_fusiontree.size() << endl;

  cout << "Queried positions:" << endl;
  cout << idx1 << " " << idx2 << " " << idx3 << endl;

  cout << "Queried elements:" << endl;
  cout << (int)my_fusiontree.pos(idx1) << " "
       << (int)my_fusiontree.pos(idx2) << endl;
}
```

When we compile and run the code in `example.cpp` as described in 2.4, we should obtain the following output:

```
Fusion Tree size:
5
Queried positions:
0 2 -1
Queried elements:
1 9
```

A brief explanation of the example code and its output can be seen below:

In lines 2-5 of our example code, we are simply declaring a vector `small_squares` of `big_int` which contains the first five positive squares, *i.e.*, `[1, 4, 9, 16, 25]`.

```
vector<big_int> small_squares;
for (int i = 1; i <= 5; i++) {
    small_squares.push_back(i * i);
}
```

Then, in line 7 we are declaring a pointer to an `environment` which is initialized with its default arguments, *i.e.*, those of a fusion tree that can store up to five elements.

```
environment *env = new environment;
```

In line 8, we use the `fusiontree` constructor to create an instance of a `fusiontree` using the environment constants defined by `*env` that contains the elements in `small_squares`.

```
fusiontree my_fusiontree(small_squares, env);
```

In lines 10-12, we query the `fusiontree` for the predecessors of 3, 9, and 0, respectively.

```
int idx1 = my_fusiontree.find_predecessor(3);
int idx2 = my_fusiontree.find_predecessor(9);
int idx3 = my_fusiontree.find_predecessor(0);
```

In lines 14-15, we print the size of the `fusiontree` instance, which is 5.

```
cout << "Fusion Tree size:" << endl;
cout << my_fusiontree.size() << endl;
```

In lines 17-18, we print the positions of the predecessor of 3, 9, and 0, respectively, which are 0, 2, and -1 (since no element in the `fusiontree` is smaller or equal to zero).

```
cout << "Queried positions:" << endl;
cout << idx1 << " " << idx2 << " " << idx3 << endl;
```

Then, in lines 20-22, we print the elements in the valid positions 0, and 2, which are 1 and 9.

```
cout << "Queried elements:" << endl;
cout << (int)my_fusiontree.pos(idx1) << " "
     << (int)my_fusiontree.pos(idx2) << endl;
```

3 A Brief Review of Fusion Trees

In order to further discuss the implementation and design choices of our `fusiontree` class, we must first go through a brief review of the Fusion Tree data structure, and how it can perform its predecessor queries in constant time. Our implementation of a fusion tree tries to replicate the description provided by Demaine et al. (2012), and therefore this section will be a brief summary of his notes. We will be concise and provide only the general understanding of a Fusion Tree that is necessary in order to discuss its implementation. The original lecture notes should be consulted if the reader wants further understanding of the data structure beyond what is presented in this report.

3.1 General Overview

In order to answer a predecessor query in constant time, a fusion tree will keep all the data necessary for this operation in a single word. To fit all its elements in a single word, the fusion tree must keep a smaller representation of them that is able to preserve their order, which can be achieved with an operation called *sketching*. When we apply sketching to a set of k numbers, we can represent each element x_i in just $k-1$ bits as $sketch(x_i)$ and still preserve their relative order, *i.e.*, $sketch(x_i) < sketch(x_j) \Leftrightarrow x_i < x_j$. A perfect sketch would represent the integers in $k-1$ bits, but we cannot compute a perfect sketch in constant time in a standard word RAM computing model. However, we can compute an *approximated sketch*, which has the same property of preserving order among elements, but that uses up to k^4 bits. Thus, if we limit the maximum size of a fusion tree to $k = O(w^{1/5})$, the representation of the k approximated sketches will take space $O(k \cdot k^4) = O(w^{1/5} \cdot w^{4/5}) = O(w)$. Therefore, we can use approximated sketch instead of perfect sketch as our sketch function and all the representations of the elements in the fusion tree will still fit in a single word.

When all the elements of a set can be fit in a single word, we can perform predecessor queries in this set in constant time using a technique called *parallel comparison*. Therefore, given a number q, we can use parallel comparison to find the predecessor of $sketch(q)$ among the sketches of the elements in our set in constant time.

However, the sketch predecessor of q, *i.e.*, the largest x_i in the set such that $sketch(x_i) \leq sketch(q)$ is not necessarily the predecessor of q because sketching only preserves relative order among the elements of the set being sketched. However, if we have the sketch predecessor x_i of q, we know that that $sketch(x_i) \leq sketch(q) < sketch(x_{i+1})$, and with just these neighbors, x_i and x_{i+1} we can find the predecessor of q in constant time using a technique called *desketchifying* . It happens that if we look at the *trie* representation of the set of numbers, the LCA of either q and x_i or q and x_{i+1} is the root of the subtree that contains either the predecessor or the successor of q. Then, we only have to query these subtrees for either their leftmost or rightmost element, which can be easily done with parallel comparison. To find the LCA in a trie we only have to find the longest common prefix between the elements. In integers, finding the length of longest common prefix between x and y is the same as finding the *most significant set bit* of $x \oplus y$ which can be done in constant time.

In the following subsections, we will further detail the implementation of the aforementioned techniques, *sketching*, *desketchifying*, *approximating sketch*, *parallel comparison*, and *most significant set bit*.

3.2 Sketching

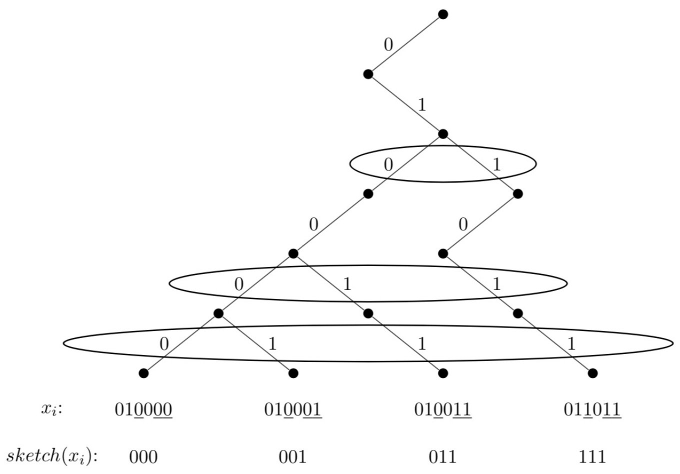

Figure 1: A representation of the elements inside the fusion tree as a trie, which clearly shows why the important bits are the ones in which the trie branches. Notice that sketching elements using their important bits preserves their order. *Figure extracted from* Demaine et al. *(2012)*

We can represent a set of numbers as *trie*, or prefix tree. In such a tree, each number would be represented by a path down a binary tree in which we would turn left at level i if the i-th bit of the number is 0, and turn right if it is 1. Each level of this tree is associated with the position of a bit position in the numbers, and positions associated with levels of in which there is branching in the tree will be called *important bits* $b_0, b_1, ..., b_{r-1}$. Then, $sketch(x_1)$ will simply be the the number formed by the concatenation of the bits of x_i, in order, in positions $b_0, b_1, ..., b_{r-1}$. An illustration of the trie representation of the elements in a fusion tree, their important bits and sketches can be seen in Figure 1.

3.3 Desketchifying

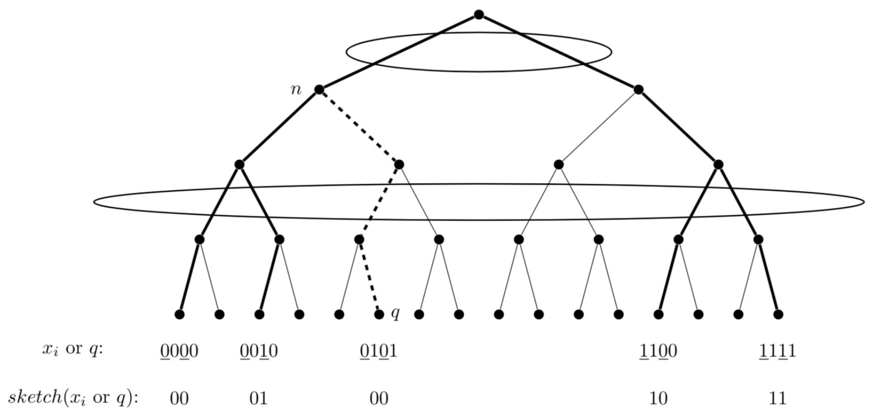
n
q
x_i or q:
$\underline{0}0\underline{0}0$
$\underline{0}0\underline{1}0$
$\underline{0}1\underline{0}1$
$\underline{1}1\underline{0}0$
$\underline{1}1\underline{1}1$
$sketch(x_i$ or $q)$:
00
01
00
10
11

Figure 2: A representation of a search for the predecessor of $q = \underline{0}1\underline{0}1$. The sketch neighbors of q are $\underline{0}0\underline{0}0$ and $\underline{0}0\underline{1}0$. Node n is where q would create a new branch in the trie representation of the fusion tree. *Figure extracted from Demaine et al. (2012)*

In Desketchifying, we assume we know the sketch neighbors x_i and x_{i+1} of a number q in the fusion tree, and we want to find the predecessor of q in constant time. Let n be lowest common ancestor of both q and x_i or q and x_{i+1} (whichever is higher) in the trie representation of the fusion tree. Then, n represents the highest node in the path defined by q that diverges from the tree. This means that the subtree to which q should belong is empty in the fusion tree, but there is some element in the other subtree. Notice that this node must be the LCA of q and one of its neighbors because either the predecessor of q will be in the non-empty subtree (if q takes a turn to the right) or the successor of q will be in the non-empty subtree (if q takes a turn to the left). An example of this can be seen in Figure 2.

Let p be the prefix of q that is above node n and let y be the length of this prefix. If the y-th bit of q is 1 (q branches to the right) and we know its predecessor is in the left subtree of n, then we only need to find the rightmost element of this subtree. Notice that this can be done by finding the sketch predecessor of $p011...1$, which can be done in constant time with parallel comparison. Analogously, if the y-th bit of q is 0 and we know its successor is in the left subtree of n, we simply have to find the sketch predecessor of $p100...0$ in the fusion tree. This search will either find the predecessor or successor of q, and we must check in order to return the predecessor of q.

3.4 Approximating Sketch

Unfortunately, we cannot calculate the perfect sketch of a number in constant time in a standard word RAM computer. However, we can calculate an approximated sketch in constant time. An approximated sketch of x_i is a representation of x_i in which its important bits are in order but spread out with zeros between them. The size of an approximated sketch is $O(r^4)$, in which r is the size of a perfect sketch.

Let x' be x but with all the non-important bits set to 0. To find its approximate sketch, we will find a bitmask m (m_j is the position of the j-th most significant set bit of m) such that when we multiply $x' \cdot m$, we avoid collisions among the important bits, *i.e.*, $b_i + m_i$ are distinct for all i. Therefore, we know where the important bits of x are in $x' \cdot m$: bit b_i will be in position $b_i + m_i$. We also want the order of the important bits to be preserved ($i < j \Rightarrow b_i + m_i < b_j + m_j$) and that the distance between the farthest important bits be at most $O(r^4)$. Then, $x' \cdot m$ will be the approximated sketch of x.

To find this m we first find m' which avoids collision between the important bits but does not preserve their order. We can always find such m with at most r^3 bits. Then, we spread the bits of m' in r buckets of size r^3, positioning m'_i in bucket i, and this will be our m.

3.5 Parallel Comparison

Parallel comparison is a crucial technique to a fusion tree. The idea is that if we have an ordered list of numbers $x_1, x_2, ..., x_k$ and want to find the predecessor of q in constant time in that list. Padding all x_i with zeros so that they have the same number of bits, we would define the number D_1 to be the concatenation of the elements of the list with a 1 before each element, *i.e.,* $D_1 = 1x_1 1x_2 ... 1x_k$. Then we would define D_2 is the concatenation of k repetitions of q with a 0 before each q, *i.e.,* $D_2 = 0q0q...0q$, again padding q with zeros so that it has the same number of bits as the elements of the list.

If we calculate the difference $D_3 = D_1 - D_2$, in all the positions in which there was a 1 between some x_i, this 1 will only be there in D_3 if $x_i \geq q$. Since the x_i are ordered, there will be some position j such that the 1's before all x_i such that $i < j$ will have become 0's in D_3, but will have remained 1's for all $i \geq j$.

If we extract only these bits in the positions placed before the x_i, the number of those bits that became 0's in D_3 will be the position of the predecessor of q in the list. Therefore, we just have to extract these bits and count the number of 1's.

3.6 Most Significant Set Bit

To find the most significant set bit of an integer x of bit size w, we will first divide its bits in buckets of size $\sqrt{w}$, and find the first bucket with a set bit. We can use a technique similar to what we did in (3.5) to find which buckets have set bits. Suppose $\sqrt{w} = 4$. Let x' be x but with zeros in all bits in positions that are multiples of 4. Let $F = 10001000...1000$. If we calculate the difference $t' = F - x'$, the only 1's of F that will remain in t' are those in buckets were there were no set bits in x' (excluding the first bit of each bucket). From t', it is simple to calculate t, a number that has the first bit of each $\sqrt{w}$ bucket set only if x has a bit set in that bucket, we only need to remember to also check the first bit of each bucket of x, since they were set to zero in x'.

If we can find the first set bit if t, we can find the bucket with the first set bit of x. Let us call the first bit of each bucket of t its important bits. Luckily, since we know the position of these bits, we know there is a bit mask m[1] such that $t \cdot m$ is a perfect sketch of the important bits of t. Therefore, we have the bits of t in order and occupying only $\sqrt{w}$ bits. We can then use parallel comparison to find the predecessor of $t \cdot m$ among the list of powers of 2 ($2^0, 2^1, ..., 2^{\sqrt{w}}$) and this will define the first set bit of $t \cdot m$, which will also define the first bucket with a set bit in x. To find the first set bit within that bucket, we only have to apply the same parallel comparison with powers of 2 again, now to the entire bucket we have just found.

4 Implementation and Design Choices

After a brief review of the structure of a fusion tree and the most important techniques on which it relies, we can discuss our particular implementation of this data structure.

4.1 Construction

4.1.1 Environment

The initialization of the Fusion Tree begins with the initialization of an `environment` and all the pre calculated constants that it keeps. In short, when we initialize an `environment`, it will hold

[1]Let m_j be the position of the j-th set but of m. Then, $m_j = w - (\sqrt{w} - 1) - j\sqrt{w} + j$

important values to the `fusiontree` that depend only on the initialization arguments: word size, size of the fusion tree element, and maximum capacity of a fusion tree. For example, among the values stored in the environment, we will have:

- `big_int clusters_first_bits`: Bit mask to extract the first bit of each bucket in (3.6);
- `big_int perfect_sketch_m`: bit mask m used to find the perfect sketch in (3.6);
- `big_int repeat_int`: value that we multiply by a number x to obtain a repetition o x (*i.e.*, $0x0x...0x$) used in (3.5);
- `big_int powers_of_two`: All powers of two up to $2^{\sqrt{w}}$ concatenated and separated by 0's used in (3.5);
- `big_int interposed_bits`: Bit mask used to extract the interposed bits between the concatenated elements in (3.5);

4.1.2 Fusion Tree

When we initialize the `fusiontree`, we need to find the important bits of its elements (3.2), find the m used to compute their approximated sketches and (3.4), and store their approximated sketches concatenated in a single word to further use them in parallel comparison (3.5). We implemented the `fusiontree` constructor as follows:

```
1  fusiontree::fusiontree(vector<big_int> &elements_,
2                         environment *my_env_) {
3    my_env = my_env_;
4
5    elements = new (big_int[my_env->word_size]);
6    m_indices = new (int[my_env->word_size]);
7    important_bits = new (int[my_env->capacity]);
8    data = 0;
9    important_bits_count = 0;
10
11   add_in_array(elements_);
12   find_important_bits();
13   find_m();
14   set_parallel_comparison();
15 }
```

In line 11, we use `add_in_array` to simple sort the input `vector` of elements given to the `fusiontree` constructor and add them to a static array `elements` in which the raw numbers will be kept. This will allow for us to access the element in position i of the fusion tree in constant time, which is used to implement the function `pos(int i)`.

In line 12, the function `find_important_bits` will find the positions of the important bits of the elements in the fusion tree. It does it by simulating their insertion, one by one, in a prefix tree. For each new element, it goes through the elements already inserted looking for the one with the longest common prefix with the new element. This will be the branching point and the length of this common prefix determines the important bit defined by the inserted element. Important bits are kept in the array `important_bits`.

In line 13, the function `find_m` simply implements a search for the m that is used to calculate the approximate sketch in (3.4), and save it in the array `m_indices`. This function simply implements a brute force approach to find m' (3.4), and then spreads its bits to preserve their order, thus defining m.

To find an m' that avoids collision we use the piece of code shown below:

```
for (int i = 0; i < important_bits_count; i++) {
    for (int j = 0; j < important_bits_count_to_3; j++) {
        if ((tag & my_env->shift_1[j]) == big_int(0)) {
            m_indices[i] = j;
            // then, tag all bits of the form
            // j + important_bits[k1] - important_bits[k2]
```

We use a bit mask `tag` to keep track of all forbidden bits in m', *i.e.,* bits that would generate collision if they were set. Then, for each important bit b_i, we search for a position j such that if the bit in position j of m were set, $b_i + j$ would not collide with any other important bit, and set $m_i = j$. It is guaranteed that we will find such position j in the interval $[0, r^3]$, in which r is the number of important bits. After finding the bit m_i, we need to tag all bits it could collide with (which are in the form $j + b_{k_1} - b_{k_2}$), so that we do not pick any of them as further set bits of m'.

After this, we only have to spread out the bits of m' in r^3 buckets so that each important bit in the approximated sketch of x fall into a different bucket, thus allowing us to put them in the correct order. We do so by simply going through all important bit positions i and defining:

$$m_i = m'_i + \left(r^3 \cdot \lfloor (w_e - b_i + ir^3)/r^3 \rfloor\right)$$

in which w_e is the maximum size, in bits, of an element of a fusion tree, as defined in the `environment`.

Finally, in line 14, we use `set_parallel_comparison` to set up the important constants that will be needed to perform parallel comparison and find the sketch predecessor of any number q in the `fusiontree`. Besides setting up some straightforward bit masks and constants that well be used when calculating approximate sketch, this method will also set up the attribute `data` of the fusion tree, which is the concatenation of the sketches of all its elements, in order, in a single `big_int`. It is done as follows:

```
for (int i = 0; i < my_env->capacity; i++) {
    data = data |
           my_env->shift_1[(i + 1) * important_bits_count_to_4 + i];
    data = data | (approximate_sketch(pos(my_env->capacity - 1 - i))
                  << i * (important_bits_count_to_4 + 1));
}
```

For each element in the fusion tree, we first insert the 1' that should come before $sketch(x_i)$ in the concatenation of sketches, and then we calculate the approximate sketch of x_i and insert it in its correct place.

This peace show an important decision that we made for the code. We avoid as much as possible doing operations. Anything that we can pre calculate and save as a constant, we do so. So, for example, instead of using `1<<((i+1)*important_bits_count^4+i)` we have the array `shift_1` in the `environment` that is initialized such that `shift_1[n]=(1<<n)`, and we also keep the value of `important_bits_count^4` stored in `important_bits_count_to_4`.

4.2 Approximated Sketching

After the initialization of the `fusiontree`, calculating the approximated sketch of a number is fairly simple. The function `approximate_sketch` is implemented as follows:

```
const big_int fusiontree::approximate_sketch(const big_int &x) const {
  return ((((x & mask_important_bits) * m) & sketch_mask) >>
          (important_bits[0] + m_indices[0]));
}
```

We simply extract the important bits of x and multiply it by m. We use a bit mask `sketch_mask` to extract only the position of the important bits and shift the result to the right so that the first important bit occupy the first bit position.

4.3 Predecessor Query

4.3.1 Sketch Neighbors

Once the `fusiontree` is initialized, we have all we need to perform the predecessor query in it. The method `find_predecessor(big_int x)` first finds the sketch predecessor of x, using the `find_sketch_predecessor` function. This function is a great example of our implementation of parallel comparison, and can be seen below:

```
 1 const int fusiontree::find_sketch_predecessor(const big_int &x) const{
 2   int important_bits_count_to_4 = important_bits_count*
     important_bits_count*important_bits_count*important_bits_count;
 3
 4   big_int diff = data - multiple_sketches(x);
 5   diff = diff & extract_interposed_bits;
 6   diff = diff * repeat_int;
 7   diff = diff >> ((my_env->capacity * important_bits_count_to_4) +
 8                   (my_env->capacity - 1));
 9   diff = diff & extract_interposed_bits_sum;
10   int answer = size() - (int)diff - 1;
11
12   if (answer + 1 < size() and
13       approximate_sketch(elements[answer + 1]) == approximate_sketch(x
     )) {
14     answer++;
15   }
16   return answer;
17 }
```

In line 4, the method `multiple_sketches(x)` simply calculates $sketch(x)$ using the `approximate_sketch` method and multiplies it by a specific constant to obtain the number $0sketch(x)0sketch(x)...0sketch(x)$ that is necessary for parallel comparison (3.5).

Then, in lines 4-10 we perform the previously described steps of parallel comparison. In line 4 we subtract the concatenation of repetitions of sketches of the query from the concatenation of the sketches of elements in the fusion tree. In line 5 e extract only the bits that were interposed between the concatenated elements. In line 6, we multiply the remaining number by `repeat_int`, the constant that will sum all the set bits in the first bucket of bits. We then shift then to the right and extract their sum to know the number of set bits, which is used to calculate the position i of the sketch predecessor of the query. In lines 12-15 we are simply considering the case in which the sketch predecessor has the same sketch than the queried number.

This function also justifies why we have two different environmental constants: the word size and the maximum size of an element in the fusion tree. Since the sum of set bits in parallel comparison happens *before* the first interposed bit, we need to have extra space in a `big_int` besides the concatenated sketches for those numbers to show up, which requires that the maximum element size of a fusion tree be a bit smaller than the maximum word size, as described in the documentation of `environment` (2.2).

4.3.2 Desketchifying

With the sketch neighbors of the queried number q, we proceed to finding the LCA of q and its sketch neighbors, which is done using `my_env->fast_first_diff(x, y)`, which finds the length of the longest common prefix between two numbers x and y by finding the most significant bit of $x \oplus y$, using the `environment` method `fast_most_significant_bit(big_int x)`, which is discussed in (4.3.3). Whichever neighbor has the longest common prefix with q is the neighbor from which q branches in the prefix tree of the fusion tree.

The length of this common prefix is the level at which q branches from the tree, and we define `int lca` as the bit position in which this branching happens (first bit that is different from any other bit in the tree). As mentioned in (3.3), we must check whether this bit is 1 or 0 to define whether the predecessor of q is in the left subtree of the LCA or whether the successor of q is in the right subtree, respectively. The two cases have a similar implementation, and the implementation of the case in which this bit is 1 (q branches to the right) can be seen below:

```
if ((x & my_env->shift_1[lca]) != big_int(0)) {
    e = x & my_env->shift_neg_1[lca];
    e = e | (my_env->shift_1[lca] - big_int(1));
    answer = find_sketch_predecessor(e);
  }
```

Notice that, again, we use `shift_neg_1[i]=((-1)<<i)`. We use binary operators to define $e = p011...1$ define in (3.3) by first extracting the common prefix of the queried number and then adding the trailing 1's. The predecessor of the queried number will be the sketch predecessor of e.

In the end of `find_sketch_predecessor`, we simply check if we found the index of the predecessor or the successor of the queried number, and return `answer-1` if the latter is the case.

4.3.3 Most Significant Set Bit

The only remaining critical method in our implementation that must still be discussed is the `environment` method `fast_most_significant_bit(big_int x)`, which returns the position of the most significant set bit of `x` in constant time.

First, we must do the bit tricks cited in (3.6) to divide the bits of number x in $\sqrt{w_e}$ buckets of size $\sqrt{w_e}$. Notice that we use w_e, the size of the maximum element in the fusion tree, rather than w, the maximum size of `big_int`. We do so because, again, to perform parallel comparison, we will aggregate some bits *before* the first bucket. Thus, we need extra space in the `big_int`. Also, we opt to require the maximum size of an element to be a perfect square (2.2) to facilitate the implementation of this function. The code for this first part can be seen below:

```
1 big_int x_clusters_first_bits = x & clusters_first_bits;
2 big_int x_remain = x ^ x_clusters_first_bits;
3 x_remain = clusters_first_bits - x_remain;
4 x_remain = x_remain & clusters_first_bits;
5 x_remain = x_remain ^ clusters_first_bits;
6 big_int x_significant_clusters = x_remain | x_clusters_first_bits;
```

We rely on some constants pre calculated in the `environment`. In line 1, we extract the first bit of each cluster and keep it saved. Then we unset all of first bits in line 2 and use a trick similar to parallel comparison in line 3 to turn to 0 the bit of every bucket that contains any set bit. We then remove all the bits that are not the first of each bucket (line 4) and swap the first bits of each bucket (line 5), so that it is 1 if there is a set bit in the bucket, and 0 otherwise. However, we removed the first bit of each bucket in the beginning, so, at last, we merge the extracted original first bits of each bucket (line 6), in order to cover the case in which only the first bit of a bucket is set.

These are the bit tricks outlined in (3.6) and they leave us with a number in which the first bit of every bucket of size $\sqrt{w_e}$ is 1 if there is any set bit in that bucket, or zero, otherwise. Now, the remaining of the function is implemented as follows:

```
x_significant_clusters =
  ((x_significant_clusters * perfect_sketch_m) >> element_size) &
  (~shift_neg_0[sqrt_element_size]);

int most_significant_cluster_idx =
  cluster_most_significant_bit(x_significant_clusters);

big_int most_significant_cluster =
  (x >> (most_significant_cluster_idx * sqrt_element_size)) &
  (~shift_neg_0[sqrt_element_size]);

int ans = most_significant_cluster_idx * sqrt_element_size +
         cluster_most_significant_bit(most_significant_cluster);

if (ans < 0) {
ans = -1;
}

return ans;
```

We have to perform a perfect sketch in order to move all the first bits of each bucket into the last bucket, so that they fit in $\sqrt{w_e}$ bits. As pointed out in (3.6), we know an m such that when we multiply the current number by m, the first bit of each bucket will be perfectly sketched to a spam of no more than $\sqrt{w_e}$ bits. This m was pre calculated in the initialization of the `environment`.

In lines 1-3, we multiply the number by m and shift it to the right to obtain the perfect sketch. Now all the first bits of each cluster are at the last cluster. We then use the method `cluster_most_significant_bit` to find the most significant bit in this cluster (lines 5-6), which is implemented using parallel comparison in a similar way to how we find sketch predecessor, but finding the predecessor of the bucket in a list of powers of two (from 2^0 to $2^{\sqrt{w_e}}$).

We them move the cluster with the most significant bit to the position of the last cluster (lines 8-10) and use the same method `cluster_most_significant_bit` to find out which of its bits is the most significant (lines 12-13).

At last, we simply check whether our search found any significant bit (lines 15-17), since we want to return -1 when we do not find any significant bit.

5 Conclusion

This concludes our report. We have presented a documentation for our code, a brief review of the fusion tree data structure and a more detailed explanation of our implementation of the most important aspects of the fusion tree.

In case there are theoretical aspects that are yet not clear to the reader regarding a fusion tree, we recommend consulting Demaine et al. (2012). If there are specific parts of the code in which the reader would like further understanding, either covered or not covered in the report, we recommend checking the code file in our repository, since both `fusiontree.hpp` and `fusiontree.cpp` are extensively commented.

Acknowledgments

We want to acknowledge and thank Prof. Erik Demaine for his incredible lectures and materials that allowed for the development of this project, as well as for his incredible consideration in accommodating for this late submission.

References

Ajtai, M., Fredman, M., and Komlós, J. Hash functions for priority queues. *Information and Control*, 63(3):217–225, 1984.

Demaine, E., Huynh, K., Klerman, S., and Shyu, E. Advanced data structures: Lecture 12, fusion tree notes. `https://courses.csail.mit.edu/6.851/fall17/scribe/lec12.pdf`. MIT 6.851, 2012. Online; accessed on March 27, 2021.

Fredman, M. L. and Willard, D. E. Surpassing the information theoretic bound with fusion trees. *Journal of computer and system sciences*, 47(3):424–436, 1993.