6.851 – Final Project
Implementing Fusion Trees with AVX Registers

Rogerio A. Guimaraes Junior and Mateus Bezrutchka

October 11, 2018

A fusion tree is a static data structure designed to make the predecessor query in constant time, given a set of number with constant limitation to its maximum size. It's main application is improving the time complexity of a search for an element in a B-tree of arbitrary size. In a standard B-tree of branching factor $k > 1$, it takes $O(\log k)$ time to make a predecessor query to a node of the B-tree. Also it seems to be impossible to make a constant time search because only reading the bits of the elements in the node would take $O(kw)$ time, where w is the word size of the computer in a standard word RAM model. However, since a B-tree node has a constant size limitation, fusion trees can do it because they do not store all the w bits of an element. With constant time predecessor queries in its nodes, a B-tree of size n and branching factor $w^{1/5}$ can answer a predecessor query in $O(\log_w n)$ time.

It is clear that the efficiency of a fusion tree depends on the log word size of the computer that uses it. However, mos computers use words of 64 bits, which is not enough to make a great difference. However, moder AVX vector registers can make parallel operations in up to 4 or 8 integers, which can be used to enhance the theoritical performance of a fusion tree. Our approach to it is using AVX registers to implement big integers making their arithmetical operations as fast as possible using the parallelism of the vector registers.

This project consists of two main tasks: implementing a fusion tree and implementing the big integer. The coding of both tasks have two goals: performance and readability for further use and improvements. The first part of the project report will be focused on the implementation of the fusion tree. The project is done in C++, for efficiency reasons. Also, the implementation is based on the process described by Erik Demain in [1], and he will be referenced many times to make clear what parts of his algorithm each part of the code is implementing.

1. Fusion Tree Reference

As stated before, the given implementation of a fusion tree has two goals: if focus on performance, but also readability and adaptation capacity. The code is carefully written and commented, so that any contributer in the future can make improvements in the implementations of the fusion tree operations. Also, the type of the element stored by the fusion tree is not a standard C++ type, it is a big_int class implemented by the authors which is imported by the fusion tree class, which makes possible that any author implement their own big int class, without need to modify the fusion tree code.

The project consists of some classes. The class `big_int` is declared in the file "big_int.hpp", and implemented in the file "big_int.cpp". They will be further discussed. Now we will focus on how to use two classes: `fusiontree` and `environment`, which are both defined in the file "fusiontree.hpp" and implemented in the file "fusiontree.cpp".

To declare a fusiontree, it is first necessary to import the fusion tree library and also to declare an environment. The following code describrs the declaration of a fusion tree object. We need to import the fusion tree library, the "fusiontree.hpp" header file, to use the `class`, and also the big integer library, where we find the type of the elements in the fusion tree, `big_int`. The following code declares an environment and a fusion tree. It also imports `iostream` because `big_int` has an implementation of the `<<` operator used by this library to print a `big_int`.

```
#include <iostream>
#include ``fusiontree.hpp''
#include ``big_int.hpp''

int main(){

  vector<big_int> v;

  for(int i=0; i<5; i++){
    v.push_back(i);
  }

  environment *env = new environment;
  fusiontree ft(v, env);
}
```

First, it declares a `vector` of `big_int` with the elements that we want to be stored in the fusion tree. Then, it declares a pointer to an object of the type environment. This object will be used as a way to make all the fusion trees in a same b-tree communicate. It will keep all the variables and specifications they will share and, most importantly, it will be used to precalculate many values that will be used in the fusion trees bit operstions. Precalculating many values and keeping them in a single object improves the efficiency of the bit operations made by the fusion tree.

The constructor of an environment can be found in the file “fusiontree.hpp”, line 39. It has five arguments, which already have default values.

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

The first argument, word_size_, is an integer that describes the size of the type big_int, i.e., its number of bits. The second argument, element_size_, is an integer that defines the number of bits of the elements stored in the fusion tree. The third argument, capacity_, is an integer that defines the maximum number of elements that can be stored in a fusion tree. It must be, at least, the branch factor of the B-Tree that is using these fusion trees.

These variables have some restrictions, which will be better understood after a more deep explanation of the fusion tree implementation. The variable element_size_ must be a perfect square, otherwise the declaration of the environment will raise the error ”element_size is not a square”. This helps the implementation of the “Most Significant Set Bit” operation, described in [1], pages 6-8. Also element_size_ must be greater than capacity_^{5}, so that the approximated sketched version of all the elements stored in the tree fit in a single big integer, as described in [1], pages 3-5. At last, word_size_ must be bigger than capacity_^{5}+capacity_^{4}, so that the process of finding the mask m ([1], pp 4-5) can be done without extrapolating the number of bits in a word.

After we already have a pointer to an environment, we declare a fusion tree.

The constructor of a fusion tree can be found in the file “fusiontree.hpp”, line 117. It has two arguments, with no default values.

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

The first argument, v_, is a reference to a std vector of big_int. This vector should contain the elements that will be stored in the fusion tree. The second argument, my_env_, is a pointer to an environment that will have the specifications of the fusion tree that is being declared.

The public methods of a fusion tree ca be found in file “fusiontree.hpp”, lines 104-120.

```
const int size() const;

const big_int pos(int i) const;

const int find_predecessor(const big_int &x) const;

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

~fusiontree();
```

- The first method, `size()`, returns an `int`: the number of integers stored in the fusion tree.
- The second method, `pos(int i)`, receives `int i` as an argument and returns a `big_int`: the integer stored in position `i` of the fusion tree, indexing from 0.
- The third method, `find_predecessor`, is the most important method of the class. It receives a `big_int` `x` as an argument and return the index of the greatest element `y` in the fusion tree such that `y`$\leq$`x`, or -1 if there is no such element. All the methods listed so far, including `find_predecessor`, run in constant time.
- The fourth method, the constructor, has already been described above.
- The fifth method is simply the deconstructor of the fusion tree.

The public methods in `environment` should only be used by the fusion tree class, so they do not require an explanation here.

The main usage of a fusion tree is being a node of a B-Tree, and to do so the public functions are all the necessary. In short, a B-Tree node should keep some elements and be able to answer a predecessor query. To do it in a fusion tree, we are going to use the methods `pos(` and `find_predecessor`. We will give a code example of how to create a fusion tree with the elements 1, 4, 9, 16, 25 and answer some predecessor queries.

```
#include <iostream>
#include ‘‘fusiontree.hpp’’
#include ‘‘big_int.hpp’’

int main(){

  vector<big_int> v;

  for(int i=0; i<5; i++){
    v.push_back((i+1)*(i+1));
  }

  environment *env = new environment;
  fusiontree ft(v, env);

  int i1 = ft.find_predecessor(3);
  int i2 = ft.find_predecessor(9);
  int i3 = ft.find_predecessor(0);

  cout << i1 << ‘‘ ‘‘ << i2 << ‘‘ ‘‘ << i3 << endl;
  cout << ft.pos(i1) << ‘‘ ‘‘ << ft.pos(i2) << endl;

  cout << return 0 << ednl;
}
```

This code will generate the following output:

```
0 2 -1
4 9
```

This output means that, in the fusion tree, 0 is the index of the predecessor of 3, which is 1; and 2 is index of the predecessor of 9, which is 9. The -1 means that 0 has no predecessor in the tree, i.e., there is no element k in the fusion tree such that $k \leq 0$.

What was showed is everything thst is necessary to use a fusion tree in a C++ code. Thus, we will now focus in the implementation of the fusion tree, so that any contributer can understand and modify the code for any reason she wants.

2. Implementation

First, let's see a brief introduction about how a fusion tree is able to perform its operations in constant time. Parallel comparison is an algorithm to answer predecessor queries in constant time. However, it requires that all the data be stored in a single word. Since each element stored in the fusion tree already has the size of a word, we can not simply apply the algorith to the raw data. However, we do not need all the bits of the elements to answer predecessor queries. In fact (as described in the sketching process in [1], pp 4-5) the important bits of a set of integers are all the bits necesseary to extract all the information about the integers ordering. Suppose, for example, that we have binary integers 010, 100, and 110 stored. We only need the first two bits to define their ordering. If we only had the first two bits of each element, we would have 01, 10 and 11, and thus would be able to correctly sort them. Figure 1 shows an example with 4 integers of size 6. In the figure, it is easy to see that the important bits of a set of integers are the ones in which there is branching in a trie representation of the set.

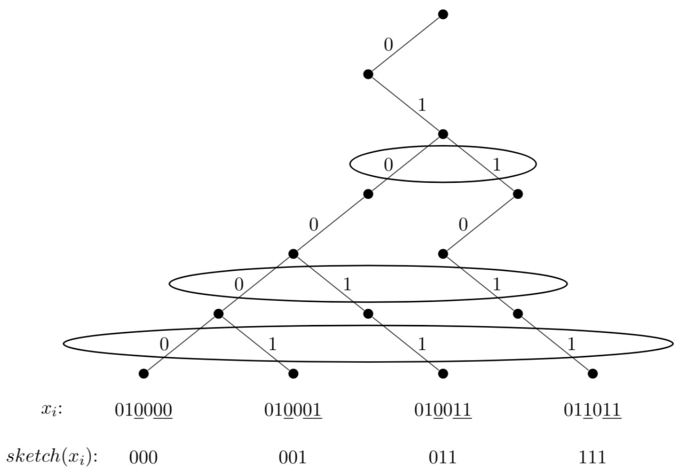
0
1
0
1
0
0
0
1
1
0
1
1
1
x_i:
01$\underline{0}$0$\underline{00}$
01$\underline{0}$0$\underline{01}$
01$\underline{0}$0$\underline{11}$
01$\underline{1}$0$\underline{11}$
$sketch(x_i)$:
000
001
011
111

Figure 1: A representation of the elements inside the fusion tree as a trie [1]. Such representation clearly shows that the important bits are the ones with branches, and the sketches of the elements

It seems that we can substantialy decrease the size of the elements since we only need $n-1$ important bits in a set of n integers. However, we can not extract the k important bits of an integer to a word of size exactly k in constant time. In other words, perfect sketching of a number is not possible. However, we can do an approximate sketch of a number in constant time, i.e., extract the k important bits to a word of size $O(k^4)$, by just multiplying it by a certain integer m. Thus, we can store $O(k)$ approximate sketches in a single word, and apply parallel comparison to them in constant time.

Therefore, we can essentialy divide the implmentation of a fusion tree in two parts: construction and querying. The construction is totally done by the constructor and basically includes finding the approximate sketches of the integers and preparing everything that is needed to apply parallel comparison when answering the queries.

Let's take a better look at some basic elements of classses `environment` and `fusiontree`. In the environment class, we can find, among others, the following objects, in lines 18-24:

```
class environment {
 public:
  int word_size;
  int element_size;
  int sqrt_element_size;
  int capacity;

  [...]
}
```

The variables `word_size`, `element_size`, and `capacity` just keep the values of the variables with similar names that are passed as arguments to the environment constructor. The variable `sqrt_element_size` just keeps the value of the square root of `element_size`.

Now, in the fusion tree class, we can find the following:

```
class fusiontree {
 private:
  environment *my_env;

  big_int data;

  big_int *elements;

  [...]

  big_int m;
  int *m_indices;
  big_int sketch_mask;

  int important_bits_count;
  big_int mask_important_bits;
  int *important_bits;

  [...]
}
```

`my_env` is a pointer to the environment with the specifications of the fusiontree. The variable `data` is the big integer that will keep the approximated sketched version of all elements in the fusion tree. The pointer `*elements` points to the beginning of an array of big integer with the original form of the elements of the tree. The variable `m` represents the same m defined in [1] (pp 4-5), `*m_indices` points to an array with the indices of the set bits in m, and `sketched_mask` is a bit mask that has set where the important bits of a number x is after we multiply x by m. At last, `important_bits_count` keeps the number of important bit and `mask_important_bits` is a bitmask that has set the important bits.

Construction

These are all the most basic and important elements a fusion tree must have in order to work. Now, I will describe what happens, step by step, when we construct and use a fusion tree. The implementation of the constructor can be found in lines 565-597 of the file "fusiontree.hpp".

```
fusiontree::fusiontree(vector<big_int> &elements_, environment *my_env_) {
  my_env = my_env_;

  elements = new (big_int[my_env->word_size]);
  m_indices = new (int[my_env->word_size]);
  important_bits = new (int[my_env->capacity]);
  data = 0;
  important_bits_count = 0;

  add_in_array(elements_);

  find_important_bits();

  find_m();

  set_parallel_comparison();
}
```

the first things the constructor does is to assign the value of `my_env` which is a pointer to the fusion tree's environment, passed as an argument to the constructor. It then uses the constructor specifications to assign values to other variables, which have already been presented before. Static arrays are declared as pointers because we want the values of `word_size` and `capacity` not to be constant, but adaptable to different environments, but we can not declare arrays of variable lenght as elements of a C++ class. After this, it calls the function `add_in_array(elements_)`, which just puts the elements from vector `elements_` in the static array `elements`, and sorts then in ascending order, which takes $O(k \log k)$, where k=`capacity`.

After this, the constructor finds the important bits of the elements in the fuison tree. The constructor does it by callig the method `find_important_bits()`. This function simulates inserting the elements one by one. At each insertion it finds the place where the element would creates a new branch on the tree. This is the same of, when inserting element s_i, finding the element s_j already in the tree such that the first different bit between them ins the lowest possible; the branch will be created in the level of that bit, which is a important. To find the first different bit, `find_important_bits()` uses a methd from the `environment` of the fusion tree called `fast_first_diff()`, which will be further discussed. After having found all the important bits, `find_important_bits()` updates the values of `important_bits_count`, `mask_important_bits`, and `important_bits` in the fusion tree. This brute force approach takes $O(k^2)$, since `fast_first_diff()` is $O(1)$.

After the constructor finds the important bits of the elements in the fusion tree, it finds a value for m calling the method `find_m()`. It takes $O(k^4)$ and finds m using a brute force approach. We want $x \cdot m$ to be the approximate sketch of x. To do so, we need that all of the important bits of x be found in a range of size $O(k^4)$ of $x \cdot m$, so that we can simple shift and extract them. We can find m such that each important bit of x is in a known position in an interval of size k^3 in$x \cdot m$. We just need to find k bits m_i such that all possible sums $m_i + b_j$ (b_i is the i-th important bit) are unique $\mod k^3$, which can be done with brute force, since $m < k^3$. However, we also need them to be in order. Therefore, we will separate the bits of $x \cdot m$ in k intervals of k^3, and we want each important bit to go to a different interval. We will put bit i in the i-th interval and thus, in the end, we will have all the important bits in an interval of size k^4. In short, after findind the values of all m_i, which work $\mod k^3$, we need to spread them, putting each of them, in order, in one interval of size k^3 of m. The specifcs of the proces can be seem in the comments in the function body and in [1], Approximating Sketch (pp 3-5).

The last method the constructor calls is `set_parallel_comparison()`. First, it sets the value of `data`, calculating the approximate sketch of all elements in the tree and concateneting all of them in a single word.

It calculates the approximate sketch calling the function `aproximate_sketch(big_int x)`. This function is very simple: it extracts the important bits of x, multiply them by m and extract only the important bits in the right place (bit i in the i-th interval of size k^3 bits), using `sketch_mask`, then it just shifts the result to the right until the first important bit becomes the first bit.

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

After setting data, `set_parallel_comparison()` sets the value of important constants to the fusiontree, which are `repeat_int`, `extract_interposed_bits`, and `extract_interposed_bits_sum`. Their definitions and values will be further discussed when parallel comparison be presented.

Predecessor Query

Now, let's see how the method `find_predecessor(x)` can answer a predecessor query in constant time. The first thing this function does is finding the position of the approximate sketch of x. to do so, it calls the method `find_sketch_predecessor`. Since data is already precalculated, and has all the sketches of the elements of the fusion tree in a single word, it simply uses parallel comparison to find the position of the approximate sketch of x among the other sketches. Parallel comaprison is done in constant time, and finding the approximate sketch of x to, thus this step is done in constant time.

Therefore, we have the value of idx_1 and $idx_2 = idx_1 + 1$, which are the indices of the skethches immediatly before (x_1) and after (x_2) the approximage sketch of x, in the fusion tree. The sketch neighbors are not necesserely the real neighbors of x, however, they tell us where x creates a new branch in the true representation of the fusion tree. Calculate the lowest common ancestor between x and both x_1 and x_2. Whichever is the lowest represents the branching point of x in the trie. The code also carefully handles the corner cases in which either x_1 or x_2 do not exist. These LCAs can be calculated computing the longest common prefixes between x and both x_1 and x_2, which can be also done in constant time by the environment method `fast_first_diff`. The lowest LCA n is the branching point of x because the subtree of n in which x is found is empty in the fusion tree. This is true because, otherwise, n would set an important bit and the other element in the subtree would be the sketch predecessor or sucessor of the sketch of x, which is a contradiction. To better understand it, look at Figure 2, which shows where an element q branches from an fusion tree with other 4 elements.

Therefore, `find_predecessor(x)` identifies whether x is in the left or in the right subtree of its LCA in the tree. the variable `lca` is the index of the bit in which x branches from the tree. Thus, if bit in position `lca` is 1, we know x is in the right subtree, thus its predecessor is in the left. Since the sketches keep the order of the elements, we just need to find the rightmost sketch that falls in this left subtree. This is checked in `if ((x & my_env->shift_1[lca]) != big_int(0))`, which is just using environment methods to calculate fast the value of `1<<(lca)`.

To find the righmost element in the leftsubtree, `find_predecessor(x)` simply searches for the sketch predecessor of `e = p0111...111`, where p is the common prefix between x and its LCA. Since e begins with `p0`, we know it falls in the left subtree, and since all the remaining bits of e are 1, it will be greater or equal to any other element in the fusion tree. Thus, the element which is the sketch predecessor of e is the predecessor of x. The program calculates e in a very straightforward way, and uses `find_sketch_predecessor(e)` to find its sketch predecessor.

The program does an analog approach when x is in the left subtree. We know that its sucessor is in the right subtree, thus it finds the sketch predecessor of the smallest element that can be in the right subtree, which is $p1000...000$. It also handles the corner cas in which it returns the sucessor of x (when $p1000...000$ is the tree), and just subtracts 1 from the returned index.

All of the steps described use standard C++ bit operations and the function `find_sketch_predecessor`, and both have constant time complexity, thus the whole `find_predecessor` method runs in constant time.

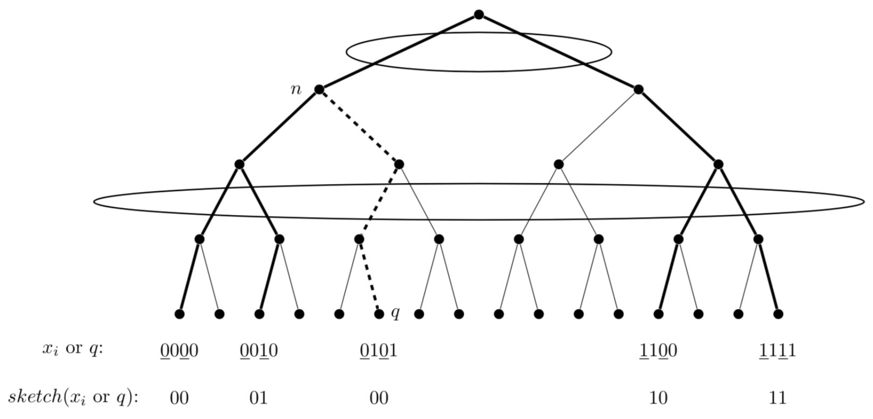

Figure 2: A representation of searching for the predecessor of a number q [1]. The sketch predecessors of q are in indices 0 and 1, and n is where q creates a new branch in the trie representation of the fusion tree.

The code is more specifically explained line by line in its comments, and the whole algorithm is more specifically described in [1].

Now, we only have to describe how the program does parallel comparison and first defferent bit in constant time.

First Different Bit

`fast_first_diff()` is a method from environment used to calculate the longest common prefix between two big integers x and y. It simply uses the method `fast_most_significant_bit` to find the most significant bit of `x^y`. The method `fast_most_significant_bit` also belongs to environment and runs in constant time.

First, `fast_most_significant_bit(x)` will divide x in $\sqrt{l}$ buckets of size $\sqrt{l}$ bits, where l is the number of bits in an element of a fusion tree. First, we will find the first bucket with significant bits. To do it, the method will first look for significant bits in the first bit of each bucket, extracting all of them using the environment constant `clusters_first_bits`, initialized at construction with set bits in the first bit of each cluster, and save it in `x_clusters_first_bits`. After, the function sets all first bits of all buckets into 0 and saves this number in `x_remain`. Notice that if make the subtraction `cluster_first_bits-x_remain` every bucket that had set bits inside the bucket (except for the first bit of the bucket) will have its first bit set to zero, as described in [1] (p 6). Then, again, we extract only the first bits and save them in `x_remain`. Thus, the big integer `x_significant_clusters = x_remain | x_clusters_first_bits` will have the first bit of each bucket set if such bucket has a significant bit, or unset if it has no significant bit.

Since we know where it is each of the first bits of each bucket,the function perfectly sketches then to the first $\sqrt{l}$ bits of `x_significant_clusters`. It does it just multiplying `x_significant_clusters` by `perfect_sketch_m`, shifting it l bits to the right and extracting the first l bits.

`perfect_sketch_m` is also a environment constant defined on construction, as described in [1] (p. 7).

```
for (int i = 0; i < sqrt_element_size; i++) {
    perfect_sketch_m =
        perfect_sketch_m | (shift_1[element_size - (sqrt_element_size - 1) -
                                     i * sqrt_element_size + i]);
  }
```

Now, the most significant bit of `x_significant_clusters` has the index of the first significant bucket of x. Then, `fast_most_significant_bit(x)` calls the environment method `cluster_most_significant_bit()`, that finds the most significant bit of the first $\sqrt{l}$ bits of a big integer, to find such index. Then, it simply extracts the first significant cluster, and shifts the whole number to the right until it is the first cluster, and applies `cluster_most_significant_bit()` again to find the first significant bit in the cluster. If c is the index of the first significant cluster, and d the index of the first significant bit inside this cluster, the functions just returns $c\sqrt{l} + d$, which is the first significant bit of x, also handling the case in which $x = 0$.

`fast_most_significant_bit(x)` only uses only C++ standard bit operations and also calls the method `cluster_most_significant_bit()`. Therefore, we only have to show that `cluster_most_significant_bit()` runs in constant time to show that `fast_most_significant_bit(x)` runs in constant time too. Finding the most significant bit is the same as finding the greatest power of two that is not greater than x. Since the number has $\sqrt{l}$ bits, we only need to search until $2^{\sqrt{l}}$. Therefore, we can use the environment constant `powers_of_two`, which is exactly the $\sqrt{l}$ powers of two concatenated in a single word, and simply do parallel comparison to ask a predecessor query in constant time. For a better understanding of the implementation, you can check the comments on the code, and for a better understanding of the algorithm, check [1] (pp. 6-8)

Parallel Comparison

At last, we have to explain how we implement parallel comparison. To do so, we will explain the implementation in textttcluster_most_significant_bit(), which is basically the same done in `find_sketch_predecessor`, except for the size of the elements used. The algorithm used is described in [1] (pp. 5-6).

```
const int environment::cluster_most_significant_bit(big_int x) const {
  x = x * repeat_int;
  x = x | interposed_bits;
  x = x - powers_of_two;
  x = x & interposed_bits;
  x = x * repeat_int;
  x = x >> ((element_size) + (sqrt_element_size - 1));
  x = x & (~shift_neg_0[sqrt_element_size + 1]);
  return (int)x - 1;
}
```

First, `cluster_most_significant_bit(x)` concatenates $\sqrt{l}$ repetitions of x in a single word, with a single bit between repetitions, by multiplying x by `repeat_int`, an environment constant with a set bit in every position p such that $p \equiv 0 \mod \sqrt{l} + 1$. Then, the code sets all the bits between the repetitions in x (which we will call interposed bits) to 1, using the environment constant `interposed_bits`, which just has all the positions of interposed bits set. The environment constant `powers_of_two` has all powers of two from 2^0 to $2^{\sqrt{l}}$, written in $\sqrt{l}$ bits, concatenated in a single word, also with interposed bits between then. Therefore, when the function performs the operation `x = x - powers_of_two;`, the i-th interposed bit of x will only remain set if $x \geq 2^i$. Then, we extract only the remaining interposed bits, and multiply the whole number again by `repeat_int`. Now, every cluster i will be repeated in all clusters $j > i$, therefore the cluster $\sqrt{l}$ (indexing from 0), will have the sum of all clusters. Since every cluster, only had its first bit set if $x \geq 2^i$, the cluster $\sqrt{l}$ will have the binary form of the number of powers of two which are not greater than x. Therefore, we just shift x to the right, turnign cluster $\sqrt{l}$ into the first cluster, and extrct only its bits. The greatest power of two which is not greater than x will be $i-1$, in which i is the greatest integer smaller than $\sqrt{l}$ such that $x \geq 2^i$. THe function only used C++ standard bit operations, thus, it runs in constant time.

It is easy to see that we could change the value of `powers_of_two` to a concatanation of any set of integers that fits in a word, and the same algorithm could be applied to find the position of x within those elements, and that is what we do to find sketch predecessors.

References

[1] Demaine, Erik (2012), "Advanced Data Structures: Lecture 12, Fusion Tree notes" [PDF] , MIT CS 6.851 (Spring 2012), Cambridge, MA, USA, Retrieved from https://courses.csail.mit.edu/6.851/fall17/scribe/lec12.pdf