Van Emde Boas, *x*-fast and *y*-fast trees

Paweł Gawrychowski

31 maja 2014

Word RAM Memory is divided into cells of size $w \geq \log n$ called **words**. There is a fixed set of $\mathcal{O}(1)$ time C-style operations, one of them being indirect addressing, so given a word containing x, we can access the cell x (not the case in the pointer machine model!). The input consists of numbers stored in single words.

AC^0 RAM All operations must be implemented by constant-depth, unbounded fan-in, polynomial-size (in w) circuits. No multiplication.

Practical RAM Just addition, shift, and bitwise boolean operations.

Cell probe model We only pay for accessing cells. The computation itself is free and the model is nonuniform. Good for lower bounds!

So far we have focused mostly on keeping the space small. During the next few lectures we will be more interested in decreasing the time complexity. We will start with a very basic problem.

Static predecessor search

Preprocess a sequence of integers $0 \leq a_1 < a_2 < \ldots < a_n < U$ so that later, given any x, we can efficiently determine the largest i such that $a_i \leq x$.

The problem has two parameters: n and U. We assume that the word size is large enough, meaning $w \geq \log n$ and $w \geq \log U$ (it might be easier to simply think that $U = 2^w$). We want to achieve low query complexity while keeping the size of our structure small.

What space complexity means in our model?

If the space usage is S, then the only cells that we are allowed to use are at offsets $1, 2, \ldots, S$. This is to forbid some weird tricks.

So far we have focused mostly on keeping the space small. During the next few lectures we will be more interested in decreasing the time complexity. We will start with a very basic problem.

Static predecessor search

Preprocess a sequence of integers $0 \leq a_1 < a_2 < \ldots < a_n < U$ so that later, given any x, we can efficiently determine the largest i such that $a_i \leq x$.

The problem has two parameters: n and U. We assume that the word size is large enough, meaning $w \geq \log n$ and $w \geq \log U$ (it might be easier to simply think that $U = 2^w$). We want to achieve low query complexity while keeping the size of our structure small.

What space complexity means in our model?

If the space usage is S, then the only cells that we are allowed to use are at offsets $1, 2, \ldots, S$. This is to forbid some weird tricks.

So far we have focused mostly on keeping the space small. During the next few lectures we will be more interested in decreasing the time complexity. We will start with a very basic problem.

Static predecessor search

Preprocess a sequence of integers $0 \leq a_1 < a_2 < \ldots < a_n < U$ so that later, given any x, we can efficiently determine the largest i such that $a_i \leq x$.

The problem has two parameters: n and U. We assume that the word size is large enough, meaning $w \geq \log n$ and $w \geq \log U$ (it might be easier to simply think that $U = 2^w$). We want to achieve low query complexity while keeping the size of our structure small.

What space complexity means in our model?

If the space usage is S, then the only cells that we are allowed to use are at offsets $1, 2, \ldots, S$. This is to forbid some weird tricks.

So far we have focused mostly on keeping the space small. During the next few lectures we will be more interested in decreasing the time complexity. We will start with a very basic problem.

Static predecessor search

Preprocess a sequence of integers $0 \leq a_1 < a_2 < \ldots < a_n < U$ so that later, given any x, we can efficiently determine the largest i such that $a_i \leq x$.

The problem has two parameters: n and U. We assume that the word size is large enough, meaning $w \geq \log n$ and $w \geq \log U$ (it might be easier to simply think that $U = 2^w$). We want to achieve low query complexity while keeping the size of our structure small.

What space complexity means in our model?

If the space usage is S, then the only cells that we are allowed to use are at offsets $1, 2, \ldots, S$. This is to forbid some weird tricks.

Static predecessor search

Preprocess a sequence of integers $0 \leq a_1 < a_2 < \ldots < a_n < U$ so that later, given any x, we can efficiently determine the largest i such that $a_i < x$.

Naive solution I

Use binary search. The query time is $\mathcal{O}(\log n)$ and the space is $\mathcal{O}(n)$.

Naive solution II

Store answers to all possible queries. The query time is $\mathcal{O}(1)$ and the space is $\mathcal{O}(U)$.

Static predecessor search

Preprocess a sequence of integers $0 \leq a_1 < a_2 < \ldots < a_n < U$ so that later, given any x, we can efficiently determine the largest i such that $a_i < x$.

Naive solution I

Use binary search. The query time is $\mathcal{O}(\log n)$ and the space is $\mathcal{O}(n)$.

Naive solution II

Store answers to all possible queries. The query time is $\mathcal{O}(1)$ and the space is $\mathcal{O}(U)$.

Static predecessor search

Preprocess a sequence of integers $0 \leq a_1 < a_2 < \ldots < a_n < U$ so that later, given any x, we can efficiently determine the largest i such that $a_i < x$.

Naive solution I

Use binary search. The query time is $\mathcal{O}(\log n)$ and the space is $\mathcal{O}(n)$.

Naive solution II

Store answers to all possible queries. The query time is $\mathcal{O}(1)$ and the space is $\mathcal{O}(U)$.

Dynamic predecessor search

Maintain a set of integers $\{a_1, a_2, \ldots, a_n\} \subseteq [U]$ under inserting and removing elements so that, given any x, we can efficiently determine the largest i such that $a_i \leq x$.

Naive solution I

Store the numbers in a balanced search tree. The query and update time is $\mathcal{O}(\log n)$ and the space is $\mathcal{O}(n)$.

Naive solution II

Store answers to all possible queries. The query time is $\mathcal{O}(1)$ and the space is $\mathcal{O}(U)$, but the update time is $\mathcal{O}(U)$ in the worst possible case (although maybe we are lucky and it is smaller).

Dynamic predecessor search

Maintain a set of integers $\{a_1, a_2, \ldots, a_n\} \subseteq [U]$ under inserting and removing elements so that, given any x, we can efficiently determine the largest i such that $a_i \leq x$.

Naive solution I

Store the numbers in a balanced search tree. The query and update time is $\mathcal{O}(\log n)$ and the space is $\mathcal{O}(n)$.

Naive solution II

Store answers to all possible queries. The query time is $\mathcal{O}(1)$ and the space is $\mathcal{O}(U)$, but the update time is $\mathcal{O}(U)$ in the worst possible case (although maybe we are lucky and it is smaller).

Dynamic predecessor search

Maintain a set of integers $\{a_1, a_2, \ldots, a_n\} \subseteq [U]$ under inserting and removing elements so that, given any x, we can efficiently determine the largest i such that $a_i \leq x$.

Naive solution I

Store the numbers in a balanced search tree. The query and update time is $\mathcal{O}(\log n)$ and the space is $\mathcal{O}(n)$.

Naive solution II

Store answers to all possible queries. The query time is $\mathcal{O}(1)$ and the space is $\mathcal{O}(U)$, but the update time is $\mathcal{O}(U)$ in the worst possible case (although maybe we are lucky and it is smaller).

The goal

We will see a number of better solutions.

Static predecessor search

Query time $\mathcal{O}(\log \log U)$ in $\mathcal{O}(n)$ space is possible.

Static predecessor search

Query time $\mathcal{O}(\frac{\log \log U}{\log \log \log U})$ in $\mathcal{O}(n^{1+\epsilon})$ space is possible.

Static predecessor search

Query time $\mathcal{O}(\frac{\log n}{\log w})$ in $\mathcal{O}(n)$ space is possible.

The goal

We will see a number of better solutions.

Static predecessor search

Query time $\mathcal{O}(\log \log U)$ in $\mathcal{O}(n)$ space is possible.

Static predecessor search

Query time $\mathcal{O}(\frac{\log \log U}{\log \log \log U})$ in $\mathcal{O}(n^{1+\epsilon})$ space is possible.

Static predecessor search

Query time $\mathcal{O}(\frac{\log n}{\log w})$ in $\mathcal{O}(n)$ space is possible.

The goal

We will see a number of better solutions.

Static predecessor search

Query time $\mathcal{O}(\log \log U)$ in $\mathcal{O}(n)$ space is possible.

Static predecessor search

Query time $\mathcal{O}(\frac{\log \log U}{\log \log \log U})$ in $\mathcal{O}(n^{1+\epsilon})$ space is possible.

Static predecessor search

Query time $\mathcal{O}(\frac{\log n}{\log w})$ in $\mathcal{O}(n)$ space is possible.

The goal

We will see a number of better solutions.

Static predecessor search

Query time $\mathcal{O}(\log\log U)$ in $\mathcal{O}(n)$ space is possible.

Static predecessor search

Query time $\mathcal{O}(\frac{\log\log U}{\log\log\log U})$ in $\mathcal{O}(n^{1+\epsilon})$ space is possible.

Static predecessor search

Query time $\mathcal{O}(\frac{\log n}{\log w})$ in $\mathcal{O}(n)$ space is possible.

The goal

Using a certain dynamization technique, called the exponential search trees, we will then dynamize these solutions to get the following.

Static predecessor search

Query time $\mathcal{O}(\min(\log\log n + \frac{\log n}{\log w}, \log\log n + \frac{\log\log U}{\log\log\log U}))$ in $\mathcal{O}(n)$ space is possible.

This is really interesting, because the minimum is always at most $\sqrt{\frac{\log n}{\log\log n}}$. So we can always beat the naive $\log n$ time solution while keeping the space linear!

Assumptions

Most of the time, we will be interested in linear space and deterministic bounds. We will explicitly mention whenever this is not the case.

The goal

Using a certain dynamization technique, called the exponential search trees, we will then dynamize these solutions to get the following.

Static predecessor search

Query time $\mathcal{O}(\min(\log\log n + \frac{\log n}{\log w}, \log\log n + \frac{\log\log U}{\log\log\log U}))$ in $\mathcal{O}(n)$ space is possible.

This is really interesting, because the minimum is always at most $\sqrt{\frac{\log n}{\log\log n}}$. So we can always beat the naive $\log n$ time solution while keeping the space linear!

Assumptions

Most of the time, we will be interested in linear space and deterministic bounds. We will explicitly mention whenever this is not the case.

The goal

Using a certain dynamization technique, called the exponential search trees, we will then dynamize these solutions to get the following.

Static predecessor search

Query time $\mathcal{O}(\min(\log\log n + \frac{\log n}{\log w}, \log\log n + \frac{\log\log U}{\log\log\log U}))$ in $\mathcal{O}(n)$ space is possible.

This is really interesting, because the minimum is always at most $\sqrt{\frac{\log n}{\log\log n}}$. So we can always beat the naive $\log n$ time solution while keeping the space linear!

Assumptions

Most of the time, we will be interested in linear space and deterministic bounds. We will explicitly mention whenever this is not the case.

The goal

Using a certain dynamization technique, called the exponential search trees, we will then dynamize these solutions to get the following.

Static predecessor search

Query time $\mathcal{O}(\min(\log\log n + \frac{\log n}{\log w}, \log\log n + \frac{\log\log U}{\log\log\log U}))$ in $\mathcal{O}(n)$ space is possible.

This is really interesting, because the minimum is always at most $\sqrt{\frac{\log n}{\log\log n}}$. So we can always beat the naive $\log n$ time solution while keeping the space linear!

Assumptions

Most of the time, we will be interested in linear space and deterministic bounds. We will explicitly mention whenever this is not the case.

Van Emde Boas trees

We start with a solution which is either randomized or takes superlinear space. Nevertheless, it is easy to understand and historically the first to achieve $\log \log U$ query time.

The idea is to use recursion. We chop every a_i into the high and low part. We allocate a huge table of length $2^{U/2}$ with one entry corresponding to every possible high part. For every possible high part we store:

1. a smaller bottom structure storing all low parts of the numbers with such high part,
2. a single bit denoting if there are any numbers with such high part,
3. the largest number with such high part.
4. the smallest number with such high part.

Additionally, we store a smaller top structure storing the high parts of all numbers.

The smaller structures are over $[\sqrt{U}]$.

Van Emde Boas trees

We start with a solution which is either randomized or takes superlinear space. Nevertheless, it is easy to understand and historically the first to achieve $\log \log U$ query time.

The idea is to use recursion. We chop every a_i into the high and low part. We allocate a huge table of length $2^{U/2}$ with one entry corresponding to every possible high part. For every possible high part we store:

1. a smaller bottom structure storing all low parts of the numbers with such high part,
2. a single bit denoting if there are any numbers with such high part,
3. the largest number with such high part.
4. the smallest number with such high part.

Additionally, we store a smaller top structure storing the high parts of all numbers.

The smaller structures are over $[\sqrt{U}]$.

Van Emde Boas trees

We start with a solution which is either randomized or takes superlinear space. Nevertheless, it is easy to understand and historically the first to achieve $\log \log U$ query time.
The idea is to use recursion. We chop every a_i into the high and low part. We allocate a huge table of length $2^{U/2}$ with one entry corresponding to every possible high part. For every possible high part we store:

1. a smaller bottom structure storing all low parts of the numbers with such high part,
2. a single bit denoting if there are any numbers with such high part,
3. the largest number with such high part.
4. the smallest number with such high part.

Additionally, we store a smaller top structure storing the high parts of all numbers.

The smaller structures are over $[\sqrt{U}]$.

Van Emde Boas trees

We start with a solution which is either randomized or takes superlinear space. Nevertheless, it is easy to understand and historically the first to achieve $\log \log U$ query time.
The idea is to use recursion. We chop every a_i into the high and low part. We allocate a huge table of length $2^{U/2}$ with one entry corresponding to every possible high part. For every possible high part we store:

1. a smaller bottom structure storing all low parts of the numbers with such high part,
2. a single bit denoting if there are any numbers with such high part,
3. the largest number with such high part.
4. the smallest number with such high part.

Additionally, we store a smaller top structure storing the high parts of all numbers.

The smaller structures are over $[\sqrt{U}]$.

Van Emde Boas trees

We start with a solution which is either randomized or takes superlinear space. Nevertheless, it is easy to understand and historically the first to achieve $\log \log U$ query time.
The idea is to use recursion. We chop every a_i into the high and low part. We allocate a huge table of length $2^{U/2}$ with one entry corresponding to every possible high part. For every possible high part we store:

1. a smaller bottom structure storing all low parts of the numbers with such high part,
2. a single bit denoting if there are any numbers with such high part,
3. the largest number with such high part.
4. the smallest number with such high part.

Additionally, we store a smaller top structure storing the high parts of all numbers.

The smaller structures are over $[\sqrt{U}]$.

Van Emde Boas trees

We start with a solution which is either randomized or takes superlinear space. Nevertheless, it is easy to understand and historically the first to achieve $\log \log U$ query time.
The idea is to use recursion. We chop every a_i into the high and low part. We allocate a huge table of length $2^{U/2}$ with one entry corresponding to every possible high part. For every possible high part we store:

1. a smaller bottom structure storing all low parts of the numbers with such high part,
2. a single bit denoting if there are any numbers with such high part,
3. the largest number with such high part.
4. the smallest number with such high part.

Additionally, we store a smaller top structure storing the high parts of all numbers.

The smaller structures are over $[\sqrt{U}]$.

Van Emde Boas trees

We start with a solution which is either randomized or takes superlinear space. Nevertheless, it is easy to understand and historically the first to achieve $\log \log U$ query time.
The idea is to use recursion. We chop every a_i into the high and low part. We allocate a huge table of length $2^{U/2}$ with one entry corresponding to every possible high part. For every possible high part we store:

1. a smaller bottom structure storing all low parts of the numbers with such high part,
2. a single bit denoting if there are any numbers with such high part,
3. the largest number with such high part.
4. the smallest number with such high part.

Additionally, we store a smaller top structure storing the high parts of all numbers.

The smaller structures are over $[\sqrt{U}]$.

Van Emde Boas trees

We start with a solution which is either randomized or takes superlinear space. Nevertheless, it is easy to understand and historically the first to achieve $\log \log U$ query time.
The idea is to use recursion. We chop every a_i into the high and low part. We allocate a huge table of length $2^{U/2}$ with one entry corresponding to every possible high part. For every possible high part we store:

1. a smaller bottom structure storing all low parts of the numbers with such high part,
2. a single bit denoting if there are any numbers with such high part,
3. the largest number with such high part.
4. the smallest number with such high part.

Additionally, we store a smaller top structure storing the high parts of all numbers.

The smaller structures are over $[\sqrt{U}]$.

Van Emde Boas trees

How to implement predecessor? We chop x into the high and low part.

1. If there are no numbers with such high part, or all such numbers are larger or equal to x, we find the predecessor of the high part in the top structure, and return the largest number with such high part.
2. Otherwise we find the predecessor of the low part in the bottom structure corresponding to the high part of x, and return a number composed of that high part and the found predecessor as the low part.

The query time is $T(U) = \mathcal{O}(1) + T(\sqrt{U})$, so as promised. But what is the space and update time?

Van Emde Boas trees

How to implement predecessor? We chop x into the high and low part.

1. If there are no numbers with such high part, or all such numbers are larger or equal to x, we find the predecessor of the high part in the top structure, and return the largest number with such high part.
2. Otherwise we find the predecessor of the low part in the bottom structure corresponding to the high part of x, and return a number composed of that high part and the found predecessor as the low part.

The query time is $T(U) = \mathcal{O}(1) + T(\sqrt{U})$, so as promised. But what is the space and update time?

Van Emde Boas trees

How to implement predecessor? We chop x into the high and low part.

1. If there are no numbers with such high part, or all such numbers are larger or equal to x, we find the predecessor of the high part in the top structure, and return the largest number with such high part.
2. Otherwise we find the predecessor of the low part in the bottom structure corresponding to the high part of x, and return a number composed of that high part and the found predecessor as the low part.

The query time is $T(U) = \mathcal{O}(1) + T(\sqrt{U})$, so as promised. But what is the space and update time?

Van Emde Boas trees

How to implement predecessor? We chop x into the high and low part.

1. If there are no numbers with such high part, or all such numbers are larger or equal to x, we find the predecessor of the high part in the top structure, and return the largest number with such high part.
2. Otherwise we find the predecessor of the low part in the bottom structure corresponding to the high part of x, and return a number composed of that high part and the found predecessor as the low part.

The query time is $T(U) = \mathcal{O}(1) + T(\sqrt{U})$, so as promised. But what is the space and update time?

Van Emde Boas trees

The space is $S(U) = \mathcal{O}(\sqrt{U}) + (1 + \sqrt{U})S(\sqrt{U})$. This might seem superlinear, but actually isn't, see the blackboard.

The update time is $T(U) = \mathcal{O}(1) + 2T(\sqrt{U})$, because we might need to update both the top structure and one of the bottom structures. Unfortunately, this solves to $\mathcal{O}(\log U)$.

The trick

We don't store all low parts in a bottom structure. Recall that for every possible high part we have two separate fields where we put the smallest and the largest low part. The largest low part is not stored in the bottom structure, we keep it **only** in the designated field.

The time for insertion becomes $T(U) = \mathcal{O}(1) + T(\sqrt{U})$, because we need to update either the top structure or one of the bottom structures. Same for deletions, but here we need to be more careful and update the largest and the smallest number after returning from the recursive call. The recurrence solves to $T(U) = \mathcal{O}(\log \log U)$.

Van Emde Boas trees

The space is $S(U) = \mathcal{O}(\sqrt{U}) + (1 + \sqrt{U})S(\sqrt{U})$. This might seem superlinear, but actually isn't, see the blackboard.
The update time is $T(U) = \mathcal{O}(1) + 2T(\sqrt{U})$, because we might need to update both the top structure and one of the bottom structures. Unfortunately, this solves to $\mathcal{O}(\log U)$.

The trick

We don't store all low parts in a bottom structure. Recall that for every possible high part we have two separate fields where we put the smallest and the largest low part. The largest low part is not stored in the bottom structure, we keep it **only** in the designated field.

The time for insertion becomes $T(U) = \mathcal{O}(1) + T(\sqrt{U})$, because we need to update either the top structure or one of the bottom structures. Same for deletions, but here we need to be more careful and update the largest and the smallest number after returning from the recursive call. The recurrence solves to $T(U) = \mathcal{O}(\log \log U)$.

Van Emde Boas trees

The space is $S(U) = \mathcal{O}(\sqrt{U}) + (1 + \sqrt{U})S(\sqrt{U})$. This might seem superlinear, but actually isn't, see the blackboard.
The update time is $T(U) = \mathcal{O}(1) + 2T(\sqrt{U})$, because we might need to update both the top structure and one of the bottom structures. Unfortunately, this solves to $\mathcal{O}(\log U)$.

The trick

We don't store all low parts in a bottom structure. Recall that for every possible high part we have two separate fields where we put the smallest and the largest low part. The largest low part is not stored in the bottom structure, we keep it **only** in the designated field.

The time for insertion becomes $T(U) = \mathcal{O}(1) + T(\sqrt{U})$, because we need to update either the top structure or one of the bottom structures. Same for deletions, but here we need to be more careful and update the largest and the smallest number after returning from the recursive call. The recurrence solves to $T(U) = \mathcal{O}(\log \log U)$.

Van Emde Boas trees

The space is $S(U) = \mathcal{O}(\sqrt{U}) + (1 + \sqrt{U})S(\sqrt{U})$. This might seem superlinear, but actually isn't, see the blackboard.
The update time is $T(U) = \mathcal{O}(1) + 2T(\sqrt{U})$, because we might need to update both the top structure and one of the bottom structures. Unfortunately, this solves to $\mathcal{O}(\log U)$.

The trick

We don't store all low parts in a bottom structure. Recall that for every possible high part we have two separate fields where we put the smallest and the largest low part. The largest low part is not stored in the bottom structure, we keep it **only** in the designated field.

The time for insertion becomes $T(U) = \mathcal{O}(1) + T(\sqrt{U})$, because we need to update either the top structure or one of the bottom structures. Same for deletions, but here we need to be more careful and update the largest and the smallest number after returning from the recursive call. The recurrence solves to $T(U) = \mathcal{O}(\log \log U)$.

Van Emde Boas trees

Of course the $\mathcal{O}(U)$ space bound is disappointing. We will show how to decrease it to $\mathcal{O}(n)$.

Dynamic perfect hashing [Dietzfelbinger, Karlin, Mehlhorn, Meyer auf der Heide, Rohnert, Tarjan 1990]

We can implement a hash table in deterministic worst-case $\mathcal{O}(1)$ time per lookup, expected amortized $\mathcal{O}(1)$ time per update, and $\mathcal{O}(n)$ space.

(this is a dynamization of the FKS scheme that you have already seen, but we won't go into the details)
Now observe that instead of allocating a huge table of length $\sqrt{U}$, we can use a hash table! The time complexities remain the same, except that now the updates are expected and amortized. But what will be the resulting space complexity?

Van Emde Boas trees

Of course the $\mathcal{O}(U)$ space bound is disappointing. We will show how to decrease it to $\mathcal{O}(n)$.

Dynamic perfect hashing [Dietzfelbinger, Karlin, Mehlhorn, Meyer auf der Heide, Rohnert, Tarjan 1990]

We can implement a hash table in deterministic worst-case $\mathcal{O}(1)$ time per lookup, expected amortized $\mathcal{O}(1)$ time per update, and $\mathcal{O}(n)$ space.

(this is a dynamization of the FKS scheme that you have already seen, but we won't go into the details)
Now observe that instead of allocating a huge table of length $\sqrt{U}$, we can use a hash table! The time complexities remain the same, except that now the updates are expected and amortized. But what will be the resulting space complexity?

Van Emde Boas trees

Of course the $\mathcal{O}(U)$ space bound is disappointing. We will show how to decrease it to $\mathcal{O}(n)$.

Dynamic perfect hashing [Dietzfelbinger, Karlin, Mehlhorn, Meyer auf der Heide, Rohnert, Tarjan 1990]

We can implement a hash table in deterministic worst-case $\mathcal{O}(1)$ time per lookup, expected amortized $\mathcal{O}(1)$ time per update, and $\mathcal{O}(n)$ space.

(this is a dynamization of the FKS scheme that you have already seen, but we won't go into the details)

Now observe that instead of allocating a huge table of length $\sqrt{U}$, we can use a hash table! The time complexities remain the same, except that now the updates are expected and amortized. But what will be the resulting space complexity?

Van Emde Boas trees

Of course the $\mathcal{O}(U)$ space bound is disappointing. We will show how to decrease it to $\mathcal{O}(n)$.

Dynamic perfect hashing [Dietzfelbinger, Karlin, Mehlhorn, Meyer auf der Heide, Rohnert, Tarjan 1990]

We can implement a hash table in deterministic worst-case $\mathcal{O}(1)$ time per lookup, expected amortized $\mathcal{O}(1)$ time per update, and $\mathcal{O}(n)$ space.

(this is a dynamization of the FKS scheme that you have already seen, but we won't go into the details)
Now observe that instead of allocating a huge table of length $\sqrt{U}$, we can use a hash table! The time complexities remain the same, except that now the updates are expected and amortized. But what will be the resulting space complexity?

Van Emde Boas trees

If we use the trick with storing the largest number separately, then counting the space (in words) per element gives us the recurrence $S(U) = 1 + S(\sqrt{U})$, so $\mathcal{O}(n \log \log U)$ in total. Disappointing, we want linear space!

Can we get linear space for the simpler static case?

Yes! We will use an insight from succinct data structures. Instead of counting words we count bits. We insert the numbers one-by-one and count how many new bits of spaces do we need. In the worst case, we need to set the largest and smallest number with a given high part and recurse in the smaller structure. Possibly, we also need to create that structure, which requires storing a new pointer in the hash table. Ignoring the pointer, the space in bits per element is $S(U) = \log U + S(\sqrt{U})$, which solves to just $\mathcal{O}(\log U)$! So, $\mathcal{O}(n)$ words in total. But what about the pointers?

Van Emde Boas trees

If we use the trick with storing the largest number separately, then counting the space (in words) per element gives us the recurrence $S(U) = 1 + S(\sqrt{U})$, so $\mathcal{O}(n \log \log U)$ in total. Disappointing, we want linear space!

Can we get linear space for the simpler static case?

Yes! We will use an insight from succinct data structures. Instead of counting words we count bits. We insert the numbers one-by-one and count how many new bits of spaces do we need. In the worst case, we need to set the largest and smallest number with a given high part and recurse in the smaller structure. Possibly, we also need to create that structure, which requires storing a new pointer in the hash table. Ignoring the pointer, the space in bits per element is $S(U) = \log U + S(\sqrt{U})$, which solves to just $\mathcal{O}(\log U)$! So, $\mathcal{O}(n)$ words in total. But what about the pointers?

Van Emde Boas trees

If we use the trick with storing the largest number separately, then counting the space (in words) per element gives us the recurrence $S(U) = 1 + S(\sqrt{U})$, so $\mathcal{O}(n \log \log U)$ in total. Disappointing, we want linear space!
Can we get linear space for the simpler static case?
Yes! We will use an insight from succinct data structures. Instead of counting words we count bits. We insert the numbers one-by-one and count how many new bits of spaces do we need. In the worst case, we need to set the largest and smallest number with a given high part and recurse in the smaller structure. Possibly, we also need to create that structure, which requires storing a new pointer in the hash table. Ignoring the pointer, the space in bits per element is $S(U) = \log U + S(\sqrt{U})$, which solves to just $\mathcal{O}(\log U)$! So, $\mathcal{O}(n)$ words in total. But what about the pointers?

Van Emde Boas trees

We cannot ignore the pointers. Observe that $n \leq U$, and we have already proved that the number of words that we need is $\mathcal{O}(n \log \log U)$. Therefore, a pointer to any "place" in our structure takes just $\mathcal{O}(\log U)$ bits, assuming that we arrange all parts of the structure in a contiguous block of memory. So, we first store the hash table (together with the largest/smallest information), then the top structure, and finally the bottom structures one-by-one. Then the additional space for every element is just $\mathcal{O}(\log U)$ in bits, so $\mathcal{O}(n)$ words in total.

OK that was complicated. Cannot we do something simpler?

Van Emde Boas trees

We cannot ignore the pointers. Observe that $n \leq U$, and we have already proved that the number of words that we need is $\mathcal{O}(n \log \log U)$. Therefore, a pointer to any "place" in our structure takes just $\mathcal{O}(\log U)$ bits, assuming that we arrange all parts of the structure in a contiguous block of memory. So, we first store the hash table (together with the largest/smallest information), then the top structure, and finally the bottom structures one-by-one. Then the additional space for every element is just $\mathcal{O}(\log U)$ in bits, so $\mathcal{O}(n)$ words in total.

OK that was complicated. Cannot we do something simpler?

Indirection

Turns out that we can use a simple but powerful tool called indirection. The first goal will be to implement a static van Emde Boas tree without the optimization trick (in the static case we have only queries anyway, so the trick doesn't influence the time complexity).

Indirection

We choose B and we split the whole sequence into blocks of length roughly B. There are $\frac{n}{B}$ of them. We construct a predecessor structure storing the largest number in every block. Additionally, the predecessor structure stores a pointer to the block where a given number comes from. A block stores a sorted list of its number.

A predecessor query reduces to a predecessor query in the large structure and a predecessor query in a block (we might need both!).

Indirection

Turns out that we can use a simple but powerful tool called indirection. The first goal will be to implement a static van Emde Boas tree without the optimization trick (in the static case we have only queries anyway, so the trick doesn't influence the time complexity).

Indirection

We choose B and we split the whole sequence into blocks of length roughly B. There are $\frac{n}{B}$ of them. We construct a predecessor structure storing the largest number in every block. Additionally, the predecessor structure stores a pointer to the block where a given number comes from. A block stores a sorted list of its number.

A predecessor query reduces to a predecessor query in the large structure and a predecessor query in a block (we might need both!).

Indirection

Turns out that we can use a simple but powerful tool called indirection. The first goal will be to implement a static van Emde Boas tree without the optimization trick (in the static case we have only queries anyway, so the trick doesn't influence the time complexity).

Indirection

We choose B and we split the whole sequence into blocks of length roughly B. There are $\frac{n}{B}$ of them. We construct a predecessor structure storing the largest number in every block. Additionally, the predecessor structure stores a pointer to the block where a given number comes from. A block stores a sorted list of its number.

A predecessor query reduces to a predecessor query in the large structure and a predecessor query in a block (we might need both!).

Indirection

Say that we use the simple static van Emde Boas tree without the optimization trick as the large structure. Its size is $\mathcal{O}(\frac{n}{B} \log U)$ in words. When we are in a block, we just binary search for the answer. The time complexity is $\mathcal{O}(\log \log U + \log B)$. Choosing $B = \log U$ gives us linear space and $\mathcal{O}(\log \log U)$ query time.

Dynamic van Emde Boas tree through indirection

This kind of works in the dynamic setting, too. We need to relax the invariants that every block is of length B. Say that we only insist that the size of every block is between $\frac{1}{2}B$ and $2B$.

Indirection

Say that we use the simple static van Emde Boas tree without the optimization trick as the large structure. Its size is $\mathcal{O}(\frac{n}{B} \log U)$ in words. When we are in a block, we just binary search for the answer. The time complexity is $\mathcal{O}(\log \log U + \log B)$. Choosing $B = \log U$ gives us linear space and $\mathcal{O}(\log \log U)$ query time.

Dynamic van Emde Boas tree through indirection

This kind of works in the dynamic setting, too. We need to relax the invariants that every block is of length B. Say that we only insist that the size of every block is between $\frac{1}{2}B$ and $2B$.

Dynamic indirection

The numbers of blocks is still $\frac{n}{B}$. Now the list of a block might change, so we implement it using any balanced search tree. Then a query or an update there takes $\mathcal{O}(\log B)$ time. Instead of saying that the large structure stores the largest number from every block, we relax the condition so that the large structure stores numbers separating the the blocks, see the blackboard. These numbers might or might not still be in the current set.

If we only need to implement insertions, we simply say that as soon as the size of a block reaches $2B$, we split it into two and insert a new element into the large structure. Then an insertion in the large structure happens once every B insertions into the set, so choosing (again) $B = \log U$ gives us amortized time $\mathcal{O}(\log \log U)$ for insertions and worst-case $\mathcal{O}(\log \log U)$ for queries.

Deletions are more tricky. We need to relax the invariant again. One possibility is that the sizes of the blocks are between 1 and $2B$, but there are no two consecutive blocks of size less than $\frac{1}{2}B$.

Dynamic indirection

The numbers of blocks is still $\frac{n}{B}$. Now the list of a block might change, so we implement it using any balanced search tree. Then a query or an update there takes $\mathcal{O}(\log B)$ time. Instead of saying that the large structure stores the largest number from every block, we relax the condition so that the large structure stores numbers separating the the blocks, see the blackboard. These numbers might or might not still be in the current set.

If we only need to implement insertions, we simply say that as soon as the size of a block reaches $2B$, we split it into two and insert a new element into the large structure. Then an insertion in the large structure happens once every B insertions into the set, so choosing (again) $B = \log U$ gives us amortized time $\mathcal{O}(\log \log U)$ for insertions and worst-case $\mathcal{O}(\log \log U)$ for queries.

Deletions are more tricky. We need to relax the invariant again. One possibility is that the sizes of the blocks are between 1 and $2B$, but there are no two consecutive blocks of size less than $\frac{1}{2}B$.

Dynamic indirection

The numbers of blocks is still $\frac{n}{B}$. Now the list of a block might change, so we implement it using any balanced search tree. Then a query or an update there takes $\mathcal{O}(\log B)$ time. Instead of saying that the large structure stores the largest number from every block, we relax the condition so that the large structure stores numbers separating the the blocks, see the blackboard. These numbers might or might not still be in the current set.

If we only need to implement insertions, we simply say that as soon as the size of a block reaches $2B$, we split it into two and insert a new element into the large structure. Then an insertion in the large structure happens once every B insertions into the set, so choosing (again) $B = \log U$ gives us amortized time $\mathcal{O}(\log \log U)$ for insertions and worst-case $\mathcal{O}(\log \log U)$ for queries.

Deletions are more tricky. We need to relax the invariant again. One possibility is that the sizes of the blocks are between 1 and $2B$, but there are no two consecutive blocks of size less than $\frac{1}{2}B$.

x-fast trees

We will conclude with a (much) simpler structure with the same bounds, i.e., $\mathcal{O}(\log \log U)$ for queries and linear space.

x-fast trees

Imagine a binary trie storing all numbers. Create a perfect hash table with all prefixes of the numbers, i.e., pointers from labels of the paths in the trie to the corresponding nodes. In every node, store the children, and the smallest/largest number in the subtree, and the predecessor of the smaller number in the subtree.

The prefect hash table stores $\mathcal{O}(n \log U)$ elements. A query can be performed by binary searching for the longest prefix of x which exists in the binary trie, see the blackboard. So the query complexity is $\mathcal{O}(\log \log U)$!

y-fast tree

But we promised linear space!

y-fast tree

Indirection with $B = \log U$ again.

This also gives good bounds for updates, because now we don't have to create $\log U$ nodes for every new element.

y-fast tree

But we promised linear space!

y-fast tree

Indirection with $B = \log U$ again.

This also gives good bounds for updates, because now we don't have to create $\log U$ nodes for every new element.

y-fast tree

But we promised linear space!

y-fast tree

Indirection with $B = \log U$ again.

This also gives good bounds for updates, because now we don't have to create $\log U$ nodes for every new element.

Questions?