The improved static structure of Beame and Fich

Paweł Gawrychowski

2 czerwca 2014

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 $a_i \leq x$.

Last week we have seen a simple linear space solution, which needs just $\mathcal{O}(\log \log U)$ time to answer a query. Its first approximation needs slightly more space.

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 $a_i \leq x$.

Last week we have seen a simple linear space solution, which needs just $\mathcal{O}(\log \log U)$ time to answer a query. Its first approximation needs slightly more 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.

Answering a query

Binary search for the longest prefix of x which exists in the binary trie. The information stored at the corresponding node is enough to determine the predecessor.

The prefect hash table stores $\mathcal{O}(n \log U)$ elements. The query complexity is $\mathcal{O}(\log \log U)$, because we binary search over range $[0, \log U]$, and every steps does one lookup in the hash table.

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.

Answering a query

Binary search for the longest prefix of x which exists in the binary trie. The information stored at the corresponding node is enough to determine the predecessor.

The prefect hash table stores $\mathcal{O}(n \log U)$ elements. The query complexity is $\mathcal{O}(\log \log U)$, because we binary search over range $[0, \log U]$, and every steps does one lookup in the hash table.

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.

Answering a query

Binary search for the longest prefix of x which exists in the binary trie. The information stored at the corresponding node is enough to determine the predecessor.

The prefect hash table stores $\mathcal{O}(n \log U)$ elements. The query complexity is $\mathcal{O}(\log \log U)$, because we binary search over range $[0, \log U]$, and every steps does one lookup in the hash table.

To get the space down to linear, apply indirection with $B = \log U$, i.e., split the input into blocks of such length, and store just one element per block in a x-fast tree. Do a binary search inside a block, because of how B was chosen this doesn't increase the total query complexity.

y-fast tree

Static predecessor search can be solved in linear space and $\mathcal{O}(\log \log U)$ time per query.

Let's spend a few minutes thinking about the dynamic predecessor search. Assuming that we can implement dynamic perfect hashing, x-fast trees need $\mathcal{O}(\log U)$ time per update, which is rather sad. What about y-fast trees?

To get the space down to linear, apply indirection with $B = \log U$, i.e., split the input into blocks of such length, and store just one element per block in a x-fast tree. Do a binary search inside a block, because of how B was chosen this doesn't increase the total query complexity.

y-fast tree

Static predecessor search can be solved in linear space and $\mathcal{O}(\log \log U)$ time per query.

Let's spend a few minutes thinking about the dynamic predecessor search. Assuming that we can implement dynamic perfect hashing, x-fast trees need $\mathcal{O}(\log U)$ time per update, which is rather sad. What about y-fast trees?

To get the space down to linear, apply indirection with $B = \log U$, i.e., split the input into blocks of such length, and store just one element per block in a x-fast tree. Do a binary search inside a block, because of how B was chosen this doesn't increase the total query complexity.

y-fast tree

Static predecessor search can be solved in linear space and $\mathcal{O}(\log \log U)$ time per query.

Let's spend a few minutes thinking about the dynamic predecessor search. Assuming that we can implement dynamic perfect hashing, x-fast trees need $\mathcal{O}(\log U)$ time per update, which is rather sad. What about y-fast trees?

Dynamic indirection in y-fast trees

We maintain a partition of the current sequence into blocks. The blocks should be chosen so that:

1. the size of every block is $\mathcal{O}(B)$,
2. there are $\mathcal{O}(\frac{n}{B})$ blocks.

where $B = \log U$. The first requirement could be relaxed, but we won't need to do so today (wait till the end of June!). We keep one element (say, the largest) from every block in a top predecessor structure, and for every block we have a separate smaller bottom balanced search tree.

A different view

It is more convenient to think that we choose $\mathcal{O}(\frac{n}{B})$ *representatives* $x_1 < x_2 < \ldots x_s$ and the i-th block contains all numbers in our sequence which belong to $[x_i, x_{i+1})$. Note that the representative might or might not belong to the current sequence. We don't care!

The representatives are kept in the top predecessor structure.

Dynamic indirection in y-fast trees

We maintain a partition of the current sequence into blocks. The blocks should be chosen so that:

1. the size of every block is $\mathcal{O}(B)$,
2. there are $\mathcal{O}(\frac{n}{B})$ blocks.

where $B = \log U$. The first requirement could be relaxed, but we won't need to do so today (wait till the end of June!). We keep one element (say, the largest) from every block in a top predecessor structure, and for every block we have a separate smaller bottom balanced search tree.

A different view

It is more convenient to think that we choose $\mathcal{O}(\frac{n}{B})$ *representatives* $x_1 < x_2 < \ldots x_s$ and the i-th block contains all numbers in our sequence which belong to $[x_i, x_{i+1})$. Note that the representative might or might not belong to the current sequence. We don't care!

The representatives are kept in the top predecessor structure.

Dynamic indirection in y-fast trees

We maintain a partition of the current sequence into blocks. The blocks should be chosen so that:

1. the size of every block is $\mathcal{O}(B)$,
2. there are $\mathcal{O}(\frac{n}{B})$ blocks.

where $B = \log U$. The first requirement could be relaxed, but we won't need to do so today (wait till the end of June!). We keep one element (say, the largest) from every block in a top predecessor structure, and for every block we have a separate smaller bottom balanced search tree.

A different view

It is more convenient to think that we choose $\mathcal{O}(\frac{n}{B})$ *representatives* $x_1 < x_2 < \ldots x_s$ and the i-th block contains all numbers in our sequence which belong to $[x_i, x_{i+1})$. Note that the representative might or might not belong to the current sequence. We don't care!

The representatives are kept in the top predecessor structure.

Dynamic indirection in y-fast trees

Say that we only insert new numbers. Then the following invariant can be maintained:

The size of every block is from $[B, 2B)$.

(unless $n < B$, when there is just one block)
To insert a number, we use the top predecessor structure to find the appropriate block, and insert the number into its bottom BST. It might happen that the size of the block becomes $2B$ then the invariant breaks. We don't want our invariants to break.
We choose the B-th element of the block and make it a new representative. Then we split the block into two blocks of size B, which restores the invariant. But how much time do we need for that? Well, $\mathcal{O}(B)$ (to insert the representative, splitting the block can be done faster, but it doesn't help), which seems like a lot. But it isn't.

Dynamic indirection in y-fast trees

Say that we only insert new numbers. Then the following invariant can be maintained:

The size of every block is from $[B, 2B)$.

(unless $n < B$, when there is just one block)
To insert a number, we use the top predecessor structure to find the appropriate block, and insert the number into its bottom BST. It might happen that the size of the block becomes $2B$ then the invariant breaks. We don't want our invariants to break.
We choose the B-th element of the block and make it a new representative. Then we split the block into two blocks of size B, which restores the invariant. But how much time do we need for that? Well, $\mathcal{O}(B)$ (to insert the representative, splitting the block can be done faster, but it doesn't help), which seems like a lot. But it isn't.

Dynamic indirection in y-fast trees

Say that we only insert new numbers. Then the following invariant can be maintained:

The size of every block is from $[B, 2B)$.

(unless $n < B$, when there is just one block)

To insert a number, we use the top predecessor structure to find the appropriate block, and insert the number into its bottom BST. It might happen that the size of the block becomes $2B$ then the invariant breaks. We don't want our invariants to break.

We choose the B-th element of the block and make it a new representative. Then we split the block into two blocks of size B, which restores the invariant. But how much time do we need for that? Well, $\mathcal{O}(B)$ (to insert the representative, splitting the block can be done faster, but it doesn't help), which seems like a lot. But it isn't.

Dynamic indirection in y-fast trees

Say that we only insert new numbers. Then the following invariant can be maintained:

The size of every block is from $[B, 2B)$.

(unless $n < B$, when there is just one block)
To insert a number, we use the top predecessor structure to find the appropriate block, and insert the number into its bottom BST. It might happen that the size of the block becomes $2B$ then the invariant breaks. We don't want our invariants to break.
We choose the B-th element of the block and make it a new representative. Then we split the block into two blocks of size B, which restores the invariant. But how much time do we need for that? Well, $\mathcal{O}(B)$ (to insert the representative, splitting the block can be done faster, but it doesn't help), which seems like a lot. But it isn't.

Dynamic indirection in y-fast trees

Say that we only insert new numbers. Then the following invariant can be maintained:

The size of every block is from $[B, 2B)$.

(unless $n < B$, when there is just one block)
To insert a number, we use the top predecessor structure to find the appropriate block, and insert the number into its bottom BST. It might happen that the size of the block becomes $2B$ then the invariant breaks. We don't want our invariants to break.
We choose the B-th element of the block and make it a new representative. Then we split the block into two blocks of size B, which restores the invariant. But how much time do we need for that? Well, $\mathcal{O}(B)$ (to insert the representative, splitting the block can be done faster, but it doesn't help), which seems like a lot. But it isn't.

Dynamic indirection in y-fast trees

Amortized complexity

We want to to argue that it cannot happen too often that we pay $\mathcal{O}(B)$ to split a block. More formally, we want to argue, that for any sequence of N inserts, we spend just $\mathcal{O}(N)$ time in total to split the blocks.

This is done by assigning credits to elements of our structure. We maintain another invariant:

A block of size $\ell \in [B, 2B)$ has $\ell - B$ credits.

Whenever we insert a new element, we allocate its credits, and transfer it to the block we are inserting into. Then when we need to split a block, we use its accumulated credits to pay for the expensive operation. This works because we have:

before the split one block of size $2B$, so with B credits available,

after the split two blocks of size B, so they don't need any credits.

Therefore, we have B credits to spend. Exactly how much do we need!

Dynamic indirection in y-fast trees

Amortized complexity

We want to to argue that it cannot happen too often that we pay $\mathcal{O}(B)$ to split a block. More formally, we want to argue, that for any sequence of N inserts, we spend just $\mathcal{O}(N)$ time in total to split the blocks.

This is done by assigning credits to elements of our structure. We maintain another invariant:

A block of size $\ell \in [B, 2B)$ has $\ell - B$ credits.

Whenever we insert a new element, we allocate its credits, and transfer it to the block we are inserting into. Then when we need to split a block, we use its accumulated credits to pay for the expensive operation. This works because we have:

before the split one block of size $2B$, so with B credits available,

after the split two blocks of size B, so they don't need any credits.

Therefore, we have B credits to spend. Exactly how much do we need!

Dynamic indirection in y-fast trees

Amortized complexity

We want to to argue that it cannot happen too often that we pay $\mathcal{O}(B)$ to split a block. More formally, we want to argue, that for any sequence of N inserts, we spend just $\mathcal{O}(N)$ time in total to split the blocks.

This is done by assigning credits to elements of our structure. We maintain another invariant:

A block of size $\ell \in [B, 2B)$ has $\ell - B$ credits.

Whenever we insert a new element, we allocate its credits, and transfer it to the block we are inserting into. Then when we need to split a block, we use its accumulated credits to pay for the expensive operation. This works because we have:

before the split one block of size $2B$, so with B credits available,

after the split two blocks of size B, so they don't need any credits.

Therefore, we have B credits to spend. Exactly how much do we need!

Dynamic indirection in y-fast trees

Amortized complexity

We want to to argue that it cannot happen too often that we pay $\mathcal{O}(B)$ to split a block. More formally, we want to argue, that for any sequence of N inserts, we spend just $\mathcal{O}(N)$ time in total to split the blocks.

This is done by assigning credits to elements of our structure. We maintain another invariant:

A block of size $\ell \in [B, 2B)$ has $\ell - B$ credits.

Whenever we insert a new element, we allocate its credits, and transfer it to the block we are inserting into. Then when we need to split a block, we use its accumulated credits to pay for the expensive operation. This works because we have:

before the split one block of size $2B$, so with B credits available,

after the split two blocks of size B, so they don't need any credits.

Therefore, we have B credits to spend. Exactly how much do we need!

Dynamic indirection in y-fast trees

Amortized complexity

We want to to argue that it cannot happen too often that we pay $\mathcal{O}(B)$ to split a block. More formally, we want to argue, that for any sequence of N inserts, we spend just $\mathcal{O}(N)$ time in total to split the blocks.

This is done by assigning credits to elements of our structure. We maintain another invariant:

A block of size $\ell \in [B, 2B)$ has $\ell - B$ credits.

Whenever we insert a new element, we allocate its credits, and transfer it to the block we are inserting into. Then when we need to split a block, we use its accumulated credits to pay for the expensive operation. This works because we have:

before the split one block of size $2B$, so with B credits available,

after the split two blocks of size B, so they don't need any credits.

Therefore, we have B credits to spend. Exactly how much do we need!

Dynamic indirection in y-fast trees

Amortized complexity

We want to to argue that it cannot happen too often that we pay $\mathcal{O}(B)$ to split a block. More formally, we want to argue, that for any sequence of N inserts, we spend just $\mathcal{O}(N)$ time in total to split the blocks.

This is done by assigning credits to elements of our structure. We maintain another invariant:

A block of size $\ell \in [B, 2B)$ has $\ell - B$ credits.

Whenever we insert a new element, we allocate its credits, and transfer it to the block we are inserting into. Then when we need to split a block, we use its accumulated credits to pay for the expensive operation. This works because we have:

before the split one block of size $2B$, so with B credits available,

after the split two blocks of size B, so they don't need any credits.

Therefore, we have B credits to spend. Exactly how much do we need!

Dynamic indirection in y-fast trees

There is one somehow subtle issue here: B is actually $\lfloor \log n \rfloor$. This value might also change after an insertion, and then some of our blocks might be too small. This issue is usually ignored because of the following generic trick.

Doubling

As soon as we the value of $\lfloor \log n \rfloor$ increases by one, completely rebuild the whole structure (using the new B) and then discard the old version.

In our case (and also in most of the other usual cases), the rebuilding can be done in $\mathcal{O}(n)$ time. Then it can be seen that the rebuilding doesn't cost that much, i.e., it amortizes.

The structure has $n - 2^{\lfloor \log n \rfloor}$ credits.

An insert increases the required number of credits by one, which is small enough. When the value of $\lfloor \log n \rfloor$ increases, we have $\frac{n}{2}$ credits, and we can use all of them.

Dynamic indirection in y-fast trees

There is one somehow subtle issue here: B is actually $\lfloor \log n \rfloor$. This value might also change after an insertion, and then some of our blocks might be too small. This issue is usually ignored because of the following generic trick.

Doubling

As soon as we the value of $\lfloor \log n \rfloor$ increases by one, completely rebuild the whole structure (using the new B) and then discard the old version.

In our case (and also in most of the other usual cases), the rebuilding can be done in $\mathcal{O}(n)$ time. Then it can be seen that the rebuilding doesn't cost that much, i.e., it amortizes.

The structure has $n - 2^{\lfloor \log n \rfloor}$ credits.

An insert increases the required number of credits by one, which is small enough. When the value of $\lfloor \log n \rfloor$ increases, we have $\frac{n}{2}$ credits, and we can use all of them.

Dynamic indirection in y-fast trees

There is one somehow subtle issue here: B is actually $\lfloor \log n \rfloor$. This value might also change after an insertion, and then some of our blocks might be too small. This issue is usually ignored because of the following generic trick.

Doubling

As soon as we the value of $\lfloor \log n \rfloor$ increases by one, completely rebuild the whole structure (using the new B) and then discard the old version.

In our case (and also in most of the other usual cases), the rebuilding can be done in $\mathcal{O}(n)$ time. Then it can be seen that the rebuilding doesn't cost that much, i.e., it amortizes.

The structure has $n - 2^{\lfloor \log n \rfloor}$ credits.

An insert increases the required number of credits by one, which is small enough. When the value of $\lfloor \log n \rfloor$ increases, we have $\frac{n}{2}$ credits, and we can use all of them.

Dynamic indirection in y-fast trees

There is one somehow subtle issue here: B is actually $\lfloor \log n \rfloor$. This value might also change after an insertion, and then some of our blocks might be too small. This issue is usually ignored because of the following generic trick.

Doubling

As soon as we the value of $\lfloor \log n \rfloor$ increases by one, completely rebuild the whole structure (using the new B) and then discard the old version.

In our case (and also in most of the other usual cases), the rebuilding can be done in $\mathcal{O}(n)$ time. Then it can be seen that the rebuilding doesn't cost that much, i.e., it amortizes.

The structure has $n - 2^{\lfloor \log n \rfloor}$ credits.

An insert increases the required number of credits by one, which is small enough. When the value of $\lfloor \log n \rfloor$ increases, we have $\frac{n}{2}$ credits, and we can use all of them.

Dynamic indirection in y-fast trees

There is one somehow subtle issue here: B is actually $\lfloor \log n \rfloor$. This value might also change after an insertion, and then some of our blocks might be too small. This issue is usually ignored because of the following generic trick.

Doubling

As soon as we the value of $\lfloor \log n \rfloor$ increases by one, completely rebuild the whole structure (using the new B) and then discard the old version.

In our case (and also in most of the other usual cases), the rebuilding can be done in $\mathcal{O}(n)$ time. Then it can be seen that the rebuilding doesn't cost that much, i.e., it amortizes.

The structure has $n - 2^{\lfloor \log n \rfloor}$ credits.

An insert increases the required number of credits by one, which is small enough. When the value of $\lfloor \log n \rfloor$ increases, we have $\frac{n}{2}$ credits, and we can use all of them.

Dynamic indirection in y-fast trees

Now say that we also delete numbers. Then some blocks might become too small or even completely empty. The former just increases the size of the top predecessor structure, but the latter kind of breaks our solution.

The size of every block is from $(0, 2B)$.

Fixing the invariant after an insertion is done as previously. If, after a deletion, the size of the block drops down to zero, we simply get rid of it, and delete one representative from the top predecessor structure. We also join any two adjacent blocks of size less than $\frac{1}{2}B$ whenever possible. How to amortize this?

A block of size ℓ has $|\ell - B|$ credits.

Then when we are joining two blocks, we can use (some of their) credits, see the blackboard. The total number of blocks is small, because there are no two adjacent blocks of size less than B.

Dynamic indirection in y-fast trees

Now say that we also delete numbers. Then some blocks might become too small or even completely empty. The former just increases the size of the top predecessor structure, but the latter kind of breaks our solution.

The size of every block is from $(0, 2B)$.

Fixing the invariant after an insertion is done as previously. If, after a deletion, the size of the block drops down to zero, we simply get rid of it, and delete one representative from the top predecessor structure. We also join any two adjacent blocks of size less than $\frac{1}{2}B$ whenever possible. How to amortize this?

A block of size ℓ has $|\ell - B|$ credits.

Then when we are joining two blocks, we can use (some of their) credits, see the blackboard. The total number of blocks is small, because there are no two adjacent blocks of size less than B.

Dynamic indirection in y-fast trees

Now say that we also delete numbers. Then some blocks might become too small or even completely empty. The former just increases the size of the top predecessor structure, but the latter kind of breaks our solution.

The size of every block is from $(0, 2B)$.

Fixing the invariant after an insertion is done as previously. If, after a deletion, the size of the block drops down to zero, we simply get rid of it, and delete one representative from the top predecessor structure. We also join any two adjacent blocks of size less than $\frac{1}{2}B$ whenever possible. How to amortize this?

A block of size ℓ has $|\ell - B|$ credits.

Then when we are joining two blocks, we can use (some of their) credits, see the blackboard. The total number of blocks is small, because there are no two adjacent blocks of size less than B.

Dynamic indirection in y-fast trees

Now say that we also delete numbers. Then some blocks might become too small or even completely empty. The former just increases the size of the top predecessor structure, but the latter kind of breaks our solution.

The size of every block is from $(0, 2B)$.

Fixing the invariant after an insertion is done as previously. If, after a deletion, the size of the block drops down to zero, we simply get rid of it, and delete one representative from the top predecessor structure. We also join any two adjacent blocks of size less than $\frac{1}{2}B$ whenever possible. How to amortize this?

A block of size ℓ has $|\ell - B|$ credits.

Then when we are joining two blocks, we can use (some of their) credits, see the blackboard. The total number of blocks is small, because there are no two adjacent blocks of size less than B.

Dynamic indirection in y-fast trees

Now say that we also delete numbers. Then some blocks might become too small or even completely empty. The former just increases the size of the top predecessor structure, but the latter kind of breaks our solution.

The size of every block is from $(0, 2B)$.

Fixing the invariant after an insertion is done as previously. If, after a deletion, the size of the block drops down to zero, we simply get rid of it, and delete one representative from the top predecessor structure. We also join any two adjacent blocks of size less than $\frac{1}{2}B$ whenever possible. How to amortize this?

A block of size ℓ has $|\ell - B|$ credits.

Then when we are joining two blocks, we can use (some of their) credits, see the blackboard. The total number of blocks is small, because there are no two adjacent blocks of size less than B.

Improved static structure

We go back to the static version. The obvious question is whether the $\mathcal{O}(\log\log U)$ bound on the query time is the best we can do! Well, yes and no.

Pătraşcu and Thorup 2006

If the allowed space is $\tilde{\mathcal{O}}(n)$, then the best possible query time is $\mathcal{O}(\log\log U)$.

But what if we allow more space?

Beame and Fich 2002

If the allowed space is $\mathcal{O}(n^{1+\epsilon})$, then one can achieve $\mathcal{O}(\frac{\log\log U}{\log\log\log U})$ query time.

And it turns out that this is tight (by Pătraşcu and Thorup 2006).

Improved static structure

We go back to the static version. The obvious question is whether the $\mathcal{O}(\log\log U)$ bound on the query time is the best we can do! Well, yes and no.

Pătraşcu and Thorup 2006

If the allowed space is $\tilde{\mathcal{O}}(n)$, then the best possible query time is $\mathcal{O}(\log\log U)$.

But what if we allow more space?

Beame and Fich 2002

If the allowed space is $\mathcal{O}(n^{1+\epsilon})$, then one can achieve $\mathcal{O}(\frac{\log\log U}{\log\log\log U})$ query time.

And it turns out that this is tight (by Pătraşcu and Thorup 2006).

Improved static structure

We go back to the static version. The obvious question is whether the $\mathcal{O}(\log\log U)$ bound on the query time is the best we can do! Well, yes and no.

Pătraşcu and Thorup 2006

If the allowed space is $\tilde{\mathcal{O}}(n)$, then the best possible query time is $\mathcal{O}(\log\log U)$.

But what if we allow more space?

Beame and Fich 2002

If the allowed space is $\mathcal{O}(n^{1+\epsilon})$, then one can achieve $\mathcal{O}(\frac{\log\log U}{\log\log\log U})$ query time.

And it turns out that this is tight (by Pătraşcu and Thorup 2006).

Improved static structure

We go back to the static version. The obvious question is whether the $\mathcal{O}(\log\log U)$ bound on the query time is the best we can do! Well, yes and no.

Pătraşcu and Thorup 2006

If the allowed space is $\tilde{\mathcal{O}}(n)$, then the best possible query time is $\mathcal{O}(\log\log U)$.

But what if we allow more space?

Beame and Fich 2002

If the allowed space is $\mathcal{O}(n^{1+\epsilon})$, then one can achieve $\mathcal{O}(\frac{\log\log U}{\log\log\log U})$ query time.

And it turns out that this is tight (by Pătraşcu and Thorup 2006).

Improved static structure

We go back to the static version. The obvious question is whether the $\mathcal{O}(\log\log U)$ bound on the query time is the best we can do! Well, yes and no.

Pătraşcu and Thorup 2006

If the allowed space is $\tilde{\mathcal{O}}(n)$, then the best possible query time is $\mathcal{O}(\log\log U)$.

But what if we allow more space?

Beame and Fich 2002

If the allowed space is $\mathcal{O}(n^{1+\epsilon})$, then one can achieve $\mathcal{O}(\frac{\log\log U}{\log\log\log U})$ query time.

And it turns out that this is tight (by Pătraşcu and Thorup 2006).

Indirection again

They actually proved a seemingly weaker statement:

Beame and Fich 2002

If the allowed space is $\mathcal{O}(n^2)$, then one can achieve $\mathcal{O}(\frac{\log\log U}{\log\log\log U})$ query time.

but we can apply indirection again. More precisely, the following holds:

Assume that we know how to implement a structure in space $\mathcal{O}(n^\alpha)$. Then we can implement a structure with the same query time (up to constant factors, of course) and space $\mathcal{O}(n^{\frac{1}{2}+\alpha/2})$.

See the blackboard. So, from now on we focus on $\alpha = 2$.

Indirection again

They actually proved a seemingly weaker statement:

Beame and Fich 2002

If the allowed space is $\mathcal{O}(n^2)$, then one can achieve $\mathcal{O}(\frac{\log\log U}{\log\log\log U})$ query time.

but we can apply indirection again. More precisely, the following holds:

Assume that we know how to implement a structure in space $\mathcal{O}(n^\alpha)$. Then we can implement a structure with the same query time (up to constant factors, of course) and space $\mathcal{O}(n^{\frac{1}{2}+\alpha/2})$.

See the blackboard. So, from now on we focus on $\alpha = 2$.

Indirection again

They actually proved a seemingly weaker statement:

Beame and Fich 2002

If the allowed space is $\mathcal{O}(n^2)$, then one can achieve $\mathcal{O}(\frac{\log\log U}{\log\log\log U})$ query time.

but we can apply indirection again. More precisely, the following holds:

Assume that we know how to implement a structure in space $\mathcal{O}(n^\alpha)$. Then we can implement a structure with the same query time (up to constant factors, of course) and space $\mathcal{O}(n^{\frac{1}{2}+\alpha/2})$.

See the blackboard. So, from now on we focus on $\alpha = 2$.

Indirection again

They actually proved a seemingly weaker statement:

Beame and Fich 2002

If the allowed space is $\mathcal{O}(n^2)$, then one can achieve $\mathcal{O}(\frac{\log\log U}{\log\log\log U})$ query time.

but we can apply indirection again. More precisely, the following holds:

Assume that we know how to implement a structure in space $\mathcal{O}(n^\alpha)$. Then we can implement a structure with the same query time (up to constant factors, of course) and space $\mathcal{O}(n^{\frac{1}{2}+\alpha/2})$.

See the blackboard. So, from now on we focus on $\alpha = 2$.

Indirection again

They actually proved a seemingly weaker statement:

Beame and Fich 2002

If the allowed space is $\mathcal{O}(n^2)$, then one can achieve $\mathcal{O}(\frac{\log\log U}{\log\log\log U})$ query time.

but we can apply indirection again. More precisely, the following holds:

Assume that we know how to implement a structure in space $\mathcal{O}(n^\alpha)$. Then we can implement a structure with the same query time (up to constant factors, of course) and space $\mathcal{O}(n^{\frac{1}{2}+\alpha/2})$.

See the blackboard. So, from now on we focus on $\alpha = 2$.

High-level idea

In y-fast trees we used binary search over $[0, \log U]$. Now we will try to tweak the search so that, in a single step, we narrow the remaining interval by a factor of $\log U$.

Formalizing this requires more work. It turns out that we cannot always guarantee that the remaining intervals shrinks significantly. What we can guarantee, however, is that either the remaining intervals shrinks, or the number of remaining elements which could be our predecessor decreases significantly.

High-level idea

In y-fast trees we used binary search over $[0, \log U]$. Now we will try to tweak the search so that, in a single step, we narrow the remaining interval by a factor of $\log U$.
Formalizing this requires more work. It turns out that we cannot always guarantee that the remaining intervals shrinks significantly. What we can guarantee, however, is that either the remaining intervals shrinks, or the number of remaining elements which could be our predecessor decreases significantly.

Main tool

THE REDUCTION

Say that we have an instance of static predecessor search with n keys of length ℓ. Choose integer parameters $q \geq 2$ dividing n and $h \geq 2$ dividing ℓ. We can reduce the original problem into a number of subproblems, each of which is easier in one of two ways:

length-reduced the length of the keys is at most $\frac{\ell}{h}$,

size-reduced the number of keys is reduced to at most $\frac{n}{q}$,

The reduction takes $\mathcal{O}(1)$ time. The space is $\mathcal{O}(q^{3h} + m)$, where $m \in [0, n]$ is some number determined by the reduction. The total size of all size-reduced subproblems is at most $n - m$ and the total size of all length-reduced subproblems is m.

Additionally, we want the reduction to split the original problem into more than one subproblem. This is easy to ensure (for example, by always removing the largest element).

Main tool

THE REDUCTION

Say that we have an instance of static predecessor search with n keys of length ℓ. Choose integer parameters $q \geq 2$ dividing n and $h \geq 2$ dividing ℓ. We can reduce the original problem into a number of subproblems, each of which is easier in one of two ways:

length-reduced the length of the keys is at most $\frac{\ell}{h}$,

size-reduced the number of keys is reduced to at most $\frac{n}{q}$,

The reduction takes $\mathcal{O}(1)$ time. The space is $\mathcal{O}(q^{3h} + m)$, where $m \in [0, n]$ is some number determined by the reduction. The total size of all size-reduced subproblems is at most $n - m$ and the total size of all length-reduced subproblems is m.

Additionally, we want the reduction to split the original problem into more than one subproblem. This is easy to ensure (for example, by always removing the largest element).

Main tool

THE REDUCTION

Say that we have an instance of static predecessor search with n keys of length ℓ. Choose integer parameters $q \geq 2$ dividing n and $h \geq 2$ dividing ℓ. We can reduce the original problem into a number of subproblems, each of which is easier in one of two ways:

length-reduced the length of the keys is at most $\frac{\ell}{h}$,

size-reduced the number of keys is reduced to at most $\frac{n}{q}$,

The reduction takes $\mathcal{O}(1)$ time. The space is $\mathcal{O}(q^{3h} + m)$, where $m \in [0, n]$ is some number determined by the reduction. The total size of all size-reduced subproblems is at most $n - m$ and the total size of all length-reduced subproblems is m.

Additionally, we want the reduction to split the original problem into more than one subproblem. This is easy to ensure (for example, by always removing the largest element).

Main tool

THE REDUCTION

Say that we have an instance of static predecessor search with n keys of length ℓ. Choose integer parameters $q \geq 2$ dividing n and $h \geq 2$ dividing ℓ. We can reduce the original problem into a number of subproblems, each of which is easier in one of two ways:

length-reduced the length of the keys is at most $\frac{\ell}{h}$,

size-reduced the number of keys is reduced to at most $\frac{n}{q}$,

The reduction takes $\mathcal{O}(1)$ time. The space is $\mathcal{O}(q^{3h} + m)$, where $m \in [0, n]$ is some number determined by the reduction. The total size of all size-reduced subproblems is at most $n - m$ and the total size of all length-reduced subproblems is m.

Additionally, we want the reduction to split the original problem into more than one subproblem. This is easy to ensure (for example, by always removing the largest element).

Main tool

THE REDUCTION

Say that we have an instance of static predecessor search with n keys of length ℓ. Choose integer parameters $q \geq 2$ dividing n and $h \geq 2$ dividing ℓ. We can reduce the original problem into a number of subproblems, each of which is easier in one of two ways:

length-reduced the length of the keys is at most $\frac{\ell}{h}$,

size-reduced the number of keys is reduced to at most $\frac{n}{q}$,

The reduction takes $\mathcal{O}(1)$ time. The space is $\mathcal{O}(q^{3h} + m)$, where $m \in [0, n]$ is some number determined by the reduction. The total size of all size-reduced subproblems is at most $n - m$ and the total size of all length-reduced subproblems is m.

Additionally, we want the reduction to split the original problem into more than one subproblem. This is easy to ensure (for example, by always removing the largest element).

Consequences of the reduction

For the time being assume that the reduction holds. How can we implement a predecessor structure? We just apply the reduction recursively! But how to choose the parameters?

Intuition

In the worst case $m = n$, and hence it makes sense to select $q^{3h} = n$. The value of n changes in the smaller recursive structures, and then it is difficult to analyze the whole thing, so to make things simpler we denote the size of the original problem by N and always choose the same q and h such that $q^{3h} = N$.

Query time

We spend $\mathcal{O}(1)$ time and then recurse in either a length-reduced or size-reduced structure. How many times can we recurse?

length-reduced we start with keys of length $\log U$, so at most $\frac{\log \log U}{\log h}$,

size-reduced we start with N keys, so at most $\frac{\log N}{\log q}$.

Consequences of the reduction

For the time being assume that the reduction holds. How can we implement a predecessor structure? We just apply the reduction recursively! But how to choose the parameters?

Intuition

In the worst case $m = n$, and hence it makes sense to select $q^{3h} = n$. The value of n changes in the smaller recursive structures, and then it is difficult to analyze the whole thing, so to make things simpler we denote the size of the original problem by N and always choose the same q and h such that $q^{3h} = N$.

Query time

We spend $\mathcal{O}(1)$ time and then recurse in either a length-reduced or size-reduced structure. How many times can we recurse?

length-reduced we start with keys of length $\log U$, so at most $\frac{\log \log U}{\log h}$,

size-reduced we start with N keys, so at most $\frac{\log N}{\log q}$.

Consequences of the reduction

For the time being assume that the reduction holds. How can we implement a predecessor structure? We just apply the reduction recursively! But how to choose the parameters?

Intuition

In the worst case $m = n$, and hence it makes sense to select $q^{3h} = n$. The value of n changes in the smaller recursive structures, and then it is difficult to analyze the whole thing, so to make things simpler we denote the size of the original problem by N and always choose the same q and h such that $q^{3h} = N$.

Query time

We spend $\mathcal{O}(1)$ time and then recurse in either a length-reduced or size-reduced structure. How many times can we recurse?

length-reduced we start with keys of length $\log U$, so at most $\frac{\log \log U}{\log h}$,

size-reduced we start with N keys, so at most $\frac{\log N}{\log q}$.

Consequences of the reduction

Size of the structure

Recall that the sizes of the subproblems sum up to n, and moreover there is more than one subproblem. Assuming $q^{3h} = N$, the reduction takes $\mathcal{O}(N)$ space in every subproblem, so $\mathcal{O}(N^2)$ in total.

Now we need to select q. The query time is:

$$\frac{\log \log U}{\log h} + \frac{\log N}{\log q}$$

see the blackboard.

Choosing appropriately we get $\mathcal{O}(\frac{\log \log U}{\log \log \log U})$ time per query using $\mathcal{O}(N^2)$ space as promised.

Consequences of the reduction

Size of the structure

Recall that the sizes of the subproblems sum up to n, and moreover there is more than one subproblem. Assuming $q^{3h} = N$, the reduction takes $\mathcal{O}(N)$ space in every subproblem, so $\mathcal{O}(N^2)$ in total.

Now we need to select q. The query time is:

$$\frac{\log \log U}{\log h} + \frac{\log N}{\log q}$$

see the blackboard.

Choosing appropriately we get $\mathcal{O}(\frac{\log \log U}{\log \log \log U})$ time per query using $\mathcal{O}(N^2)$ space as promised.

Consequences of the reduction

Size of the structure

Recall that the sizes of the subproblems sum up to n, and moreover there is more than one subproblem. Assuming $q^{3h} = N$, the reduction takes $\mathcal{O}(N)$ space in every subproblem, so $\mathcal{O}(N^2)$ in total.

Now we need to select q. The query time is:

$$\frac{\log \log U}{\log h} + \frac{\log N}{\log q}$$

see the blackboard.

Choosing appropriately we get $\mathcal{O}(\frac{\log \log U}{\log \log \log U})$ time per query using $\mathcal{O}(N^2)$ space as promised.

The reduction

The main trick is to use parallel hashing. Say that we want to store a set Z of q keys, each consisting of h characters (not necessarily binary). We denote this by $x = x_1 x_2 \ldots x_h$.

Lemma

Using additional space $\mathcal{O}(q^{3h})$ we can preprocess such a set, so that given a query x, we can find in $\mathcal{O}(1)$ time the longest common prefix (in whole characters) between x and any key in Z.

The idea is simple and neat: use h instances of some perfect hashing in parallel.

The reduction

The main trick is to use parallel hashing. Say that we want to store a set Z of q keys, each consisting of h characters (not necessarily binary). We denote this by $x = x_1 x_2 \ldots x_h$.

Lemma

Using additional space $\mathcal{O}(q^{3h})$ we can preprocess such a set, so that given a query x, we can find in $\mathcal{O}(1)$ time the longest common prefix (in whole characters) between x and any key in Z.

The idea is simple and neat: use h instances of some perfect hashing in parallel.

The reduction

The main trick is to use parallel hashing. Say that we want to store a set Z of q keys, each consisting of h characters (not necessarily binary). We denote this by $x = x_1 x_2 \ldots x_h$.

Lemma

Using additional space $\mathcal{O}(q^{3h})$ we can preprocess such a set, so that given a query x, we can find in $\mathcal{O}(1)$ time the longest common prefix (in whole characters) between x and any key in Z.

The idea is simple and neat: use h instances of some perfect hashing in parallel.

Parallel hashing

We want to find a hash function h, which is perfect on every "column". The function will be applied to all columns at once. We choose H at random from some family of functions $\mathcal{H}$ and require that:

1. H hashes into small $[m]$,
2. for any i and two characters $x \neq y$ which appear in the i-th column, the chance that $H(x) = H(y)$ is less than $\frac{2}{m}$,
3. we can compute $H(x_1), H(x_2), \ldots, H(x_h)$ in $\mathcal{O}(1)$ time, assuming that all $x_1, x_2, \ldots, x_h$ are given in a single machine word.

We choose $m = q^2 h$. Then, if we choose $H \in \mathcal{H}$ at random, the chance that we get a collision in the i-th column is less than $\frac{q^2}{m} = \frac{1}{h}$. Hence the chance that we get collision in **any** column is less than one, so it is possible to choose H so that there is no collision.

We choose a function of the form $H_a(x) = \left\lfloor \frac{a*x}{2^\alpha} \right\rfloor \bmod 2^\beta$.

See the blackboard for an explanation of why it works.

Parallel hashing

We want to find a hash function h, which is perfect on every "column". The function will be applied to all columns at once. We choose H at random from some family of functions $\mathcal{H}$ and require that:

1. H hashes into small $[m]$,
2. for any i and two characters $x \neq y$ which appear in the i-th column, the chance that $H(x) = H(y)$ is less than $\frac{2}{m}$,
3. we can compute $H(x_1), H(x_2), \ldots, H(x_h)$ in $\mathcal{O}(1)$ time, assuming that all $x_1, x_2, \ldots, x_h$ are given in a single machine word.

We choose $m = q^2 h$. Then, if we choose $H \in \mathcal{H}$ at random, the chance that we get a collision in the i-th column is less than $\frac{q^2}{m} = \frac{1}{h}$. Hence the chance that we get collision in **any** column is less than one, so it is possible to choose H so that there is no collision.

We choose a function of the form $H_a(x) = \left\lfloor \frac{a*x}{2^\alpha} \right\rfloor \bmod 2^\beta$.

See the blackboard for an explanation of why it works.

Parallel hashing

We want to find a hash function h, which is perfect on every "column". The function will be applied to all columns at once. We choose H at random from some family of functions $\mathcal{H}$ and require that:

1. H hashes into small $[m]$,
2. for any i and two characters $x \neq y$ which appear in the i-th column, the chance that $H(x) = H(y)$ is less than $\frac{2}{m}$,
3. we can compute $H(x_1), H(x_2), \ldots, H(x_h)$ in $\mathcal{O}(1)$ time, assuming that all $x_1, x_2, \ldots, x_h$ are given in a single machine word.

We choose $m = q^2 h$. Then, if we choose $H \in \mathcal{H}$ at random, the chance that we get a collision in the i-th column is less than $\frac{q^2}{m} = \frac{1}{h}$. Hence the chance that we get collision in **any** column is less than one, so it is possible to choose H so that there is no collision.

We choose a function of the form $H_a(x) = \left\lfloor \frac{a*x}{2^\alpha} \right\rfloor \bmod 2^\beta$.

See the blackboard for an explanation of why it works.

Parallel hashing

We want to find a hash function h, which is perfect on every "column". The function will be applied to all columns at once. We choose H at random from some family of functions $\mathcal{H}$ and require that:

1. H hashes into small $[m]$,
2. for any i and two characters $x \neq y$ which appear in the i-th column, the chance that $H(x) = H(y)$ is less than $\frac{2}{m}$,
3. we can compute $H(x_1), H(x_2), \ldots, H(x_h)$ in $\mathcal{O}(1)$ time, assuming that all $x_1, x_2, \ldots, x_h$ are given in a single machine word.

We choose $m = q^2 h$. Then, if we choose $H \in \mathcal{H}$ at random, the chance that we get a collision in the i-th column is less than $\frac{q^2}{m} = \frac{1}{h}$. Hence the chance that we get collision in **any** column is less than one, so it is possible to choose H so that there is no collision.

We choose a function of the form $H_a(x) = \left\lfloor \frac{a*x}{2^\alpha} \right\rfloor \bmod 2^\beta$.

See the blackboard for an explanation of why it works.

Parallel hashing

We want to find a hash function h, which is perfect on every "column". The function will be applied to all columns at once. We choose H at random from some family of functions $\mathcal{H}$ and require that:

1. H hashes into small $[m]$,
2. for any i and two characters $x \neq y$ which appear in the i-th column, the chance that $H(x) = H(y)$ is less than $\frac{2}{m}$,
3. we can compute $H(x_1), H(x_2), \ldots, H(x_h)$ in $\mathcal{O}(1)$ time, assuming that all $x_1, x_2, \ldots, x_h$ are given in a single machine word.

We choose $m = q^2 h$. Then, if we choose $H \in \mathcal{H}$ at random, the chance that we get a collision in the i-th column is less than $\frac{q^2}{m} = \frac{1}{h}$. Hence the chance that we get collision in **any** column is less than one, so it is possible to choose H so that there is no collision.

We choose a function of the form $H_a(x) = \left\lfloor \frac{a*x}{2^\alpha} \right\rfloor \bmod 2^\beta$.

See the blackboard for an explanation of why it works.

Questions?