Making array fully persistent

Paweł Gawrychowski

21 lipca 2014

Order maintenance problem

To make arrays fully persistent in the Word RAM, we will need the following tool, which is interesting on its own.

Order maintenance problem

Maintain a list, so that we can:

1. insert x before y into the list,
2. delete x from the list,
3. check if x is before y on the list.

Forget about deletions: we can just pretend that we have done them.

Order maintenance problem

To make arrays fully persistent in the Word RAM, we will need the following tool, which is interesting on its own.

Order maintenance problem

Maintain a list, so that we can:

1. insert x before y into the list,
2. delete x from the list,
3. check if x is before y on the list.

Forget about deletions: we can just pretend that we have done them.

Order maintenance problem

To make arrays fully persistent in the Word RAM, we will need the following tool, which is interesting on its own.

Order maintenance problem

Maintain a list, so that we can:

1. insert x before y into the list,
2. delete x from the list,
3. check if x is before y on the list.

Forget about deletions: we can just pretend that we have done them.

First try

Assign real number to the objects. Initially the list contains two sentinels, assign 0 and 1 to them. Then:

1. to check if x is before y on the list, compare their numbers,
2. to insert x before y, find the predecessor y' of y on the list and assign the average of the numbers of y and y' to x.

Does it work?

First try

Assign real number to the objects. Initially the list contains two sentinels, assign 0 and 1 to them. Then:

1. to check if x is before y on the list, compare their numbers,
2. to insert x before y, find the predecessor y' of y on the list and assign the average of the numbers of y and y' to x.

Does it work?

Second try

Imagine a complete binary tree on 2^M leaves. The elements of the list reside at some leaves of that tree in the same order as on the list. We will show how to implement one phase, where a phase begins with arranging n objects evenly in the leaves, and goes on as long as the number of object on the list is at most $2n$.

Now how to choose M?

1. cannot be too large, because we will store the path from every object to the root in a single machine word, called its tag.
2. cannot be too small, because otherwise the algorithm breaks.

Second try

Imagine a complete binary tree on 2^M leaves. The elements of the list reside at some leaves of that tree in the same order as on the list. We will show how to implement one phase, where a phase begins with arranging n objects evenly in the leaves, and goes on as long as the number of object on the list is at most $2n$.
Now how to choose M?

1. cannot be too large, because we will store the path from every object to the root in a single machine word, called its tag.
2. cannot be too small, because otherwise the algorithm breaks.

Second try

Density

The density of a node of the binary tree is the ratio of the number of object in the leaves of its subtree to the number of these leaves.

We want the density of a node at level i to be at most T^{-i}, i.e., the leaves have density at most 1, their parents have density at most T^{-1}, and so on, where $T \in (1, 2)$.
The tree should be large enough so that the density of the root breaks only when the number of objects becomes $2n$. This means that:

$$T^{-M} = \frac{2n}{2^M}$$

which solves to $M = \frac{\log 2n}{1 - \log T}$.
It means that we are using just $\mathcal{O}(\log n)$ bits for every tag. Good.

Second try

Density

The density of a node of the binary tree is the ratio of the number of object in the leaves of its subtree to the number of these leaves.

We want the density of a node at level i to be at most T^{-i}, i.e., the leaves have density at most 1, their parents have density at most T^{-1}, and so on, where $T \in (1, 2)$.

The tree should be large enough so that the density of the root breaks only when the number of objects becomes $2n$. This means that:

$$T^{-M} = \frac{2n}{2^M}$$

which solves to $M = \frac{\log 2n}{1 - \log T}$.

It means that we are using just $\mathcal{O}(\log n)$ bits for every tag. Good.

Second try

Density

The density of a node of the binary tree is the ratio of the number of object in the leaves of its subtree to the number of these leaves.

We want the density of a node at level i to be at most T^{-i}, i.e., the leaves have density at most 1, their parents have density at most T^{-1}, and so on, where $T \in (1, 2)$.
The tree should be large enough so that the density of the root breaks only when the number of objects becomes $2n$. This means that:

$$T^{-M} = \frac{2n}{2^M}$$

which solves to $M = \frac{\log 2n}{1 - \log T}$.
It means that we are using just $\mathcal{O}(\log n)$ bits for every tag. Good.

Second try

Density

The density of a node of the binary tree is the ratio of the number of object in the leaves of its subtree to the number of these leaves.

We want the density of a node at level i to be at most T^{-i}, i.e., the leaves have density at most 1, their parents have density at most T^{-1}, and so on, where $T \in (1, 2)$.
The tree should be large enough so that the density of the root breaks only when the number of objects becomes $2n$. This means that:

$$T^{-M} = \frac{2n}{2^M}$$

which solves to $M = \frac{\log 2n}{1 - \log T}$.
It means that we are using just $\mathcal{O}(\log n)$ bits for every tag. Good.

Second try

Now to insert x before y, we locate the predecessor y' of y, and put x in any leaf between y' and y.

Well, except when y' and y are next to each other in the tree. In such case, we find the lowest ancestor of y such that the density of its subtree is below the threshold, and evenly relabel its whole subtree, including x just before the relabeling happens.

Second try

Now to insert x before y, we locate the predecessor y' of y, and put x in any leaf between y' and y.
Well, except when y' and y are next to each other in the tree. In such case, we find the lowest ancestor of y such that the density of its subtree is below the threshold, and evenly relabel its whole subtree, including x just before the relabeling happens.

Second try

How much time to relabel a subtree rooted at u with 2^i leaves? $\mathcal{O}(\frac{2^i}{T^i})$.

What is the density of the children of u after the relabeling? $\leq \frac{1}{T^i}$.

When we relabel u again, what is the density of its child? $\geq \frac{1}{T^{i-1}}$.

Therefore, between two consecutive relabelings of the same u we have inserted $(\frac{1}{T^{i-1}} - \frac{1}{T^i})2^{i-1}$ nodes into its subtree.

So we pay $\mathcal{O}(\frac{2^i}{T^i})$ once per $(\frac{1}{T^{i-1}} - \frac{1}{T^i})2^{i-1}$ insertions, so the amortized cost is $\mathcal{O}(1)$.

Then the amortized cost of an insertion is $\mathcal{O}(\log n)$, because every insertion charge $\mathcal{O}(\log n)$ nodes.

Second try

How much time to relabel a subtree rooted at u with 2^i leaves? $\mathcal{O}(\frac{2^i}{T^i})$.

What is the density of the children of u after the relabeling? $\leq \frac{1}{T^i}$.

When we relabel u again, what is the density of its child? $\geq \frac{1}{T^{i-1}}$.

Therefore, between two consecutive relabelings of the same u we have inserted $(\frac{1}{T^{i-1}} - \frac{1}{T^i})2^{i-1}$ nodes into its subtree.

So we pay $\mathcal{O}(\frac{2^i}{T^i})$ once per $(\frac{1}{T^{i-1}} - \frac{1}{T^i})2^{i-1}$ insertions, so the amortized cost is $\mathcal{O}(1)$.

Then the amortized cost of an insertion is $\mathcal{O}(\log n)$, because every insertion charge $\mathcal{O}(\log n)$ nodes.

Second try

How much time to relabel a subtree rooted at u with 2^i leaves? $\mathcal{O}(\frac{2^i}{T^i})$.

What is the density of the children of u after the relabeling? $\leq \frac{1}{T^i}$.

When we relabel u again, what is the density of its child? $\geq \frac{1}{T^{i-1}}$.

Therefore, between two consecutive relabelings of the same u we have inserted $(\frac{1}{T^{i-1}} - \frac{1}{T^i})2^{i-1}$ nodes into its subtree.

So we pay $\mathcal{O}(\frac{2^i}{T^i})$ once per $(\frac{1}{T^{i-1}} - \frac{1}{T^i})2^{i-1}$ insertions, so the amortized cost is $\mathcal{O}(1)$.

Then the amortized cost of an insertion is $\mathcal{O}(\log n)$, because every insertion charge $\mathcal{O}(\log n)$ nodes.

Second try

How much time to relabel a subtree rooted at u with 2^i leaves? $\mathcal{O}(\frac{2^i}{T^i})$.

What is the density of the children of u after the relabeling? $\leq \frac{1}{T^i}$.

When we relabel u again, what is the density of its child? $\geq \frac{1}{T^{i-1}}$.

Therefore, between two consecutive relabelings of the same u we have inserted $(\frac{1}{T^{i-1}} - \frac{1}{T^i})2^{i-1}$ nodes into its subtree.

So we pay $\mathcal{O}(\frac{2^i}{T^i})$ once per $(\frac{1}{T^{i-1}} - \frac{1}{T^i})2^{i-1}$ insertions, so the amortized cost is $\mathcal{O}(1)$.

Then the amortized cost of an insertion is $\mathcal{O}(\log n)$, because every insertion charge $\mathcal{O}(\log n)$ nodes.

Second try

How much time to relabel a subtree rooted at u with 2^i leaves? $\mathcal{O}(\frac{2^i}{T^i})$.
What is the density of the children of u after the relabeling? $\leq \frac{1}{T^i}$.
When we relabel u again, what is the density of its child? $\geq \frac{1}{T^{i-1}}$.
Therefore, between two consecutive relabelings of the same u we have inserted $(\frac{1}{T^{i-1}} - \frac{1}{T^i})2^{i-1}$ nodes into its subtree.
So we pay $\mathcal{O}(\frac{2^i}{T^i})$ once per $(\frac{1}{T^{i-1}} - \frac{1}{T^i})2^{i-1}$ insertions, so the amortized cost is $\mathcal{O}(1)$.
Then the amortized cost of an insertion is $\mathcal{O}(\log n)$, because every insertion charge $\mathcal{O}(\log n)$ nodes.

Second try

How much time to relabel a subtree rooted at u with 2^i leaves? $\mathcal{O}(\frac{2^i}{T^i})$.
What is the density of the children of u after the relabeling? $\leq \frac{1}{T^i}$.
When we relabel u again, what is the density of its child? $\geq \frac{1}{T^{i-1}}$.
Therefore, between two consecutive relabelings of the same u we have inserted $(\frac{1}{T^{i-1}} - \frac{1}{T^i})2^{i-1}$ nodes into its subtree.
So we pay $\mathcal{O}(\frac{2^i}{T^i})$ once per $(\frac{1}{T^{i-1}} - \frac{1}{T^i})2^{i-1}$ insertions, so the amortized cost is $\mathcal{O}(1)$.
Then the amortized cost of an insertion is $\mathcal{O}(\log n)$, because every insertion charge $\mathcal{O}(\log n)$ nodes.

Second try

How much time to relabel a subtree rooted at u with 2^i leaves? $\mathcal{O}(\frac{2^i}{T^i})$.
What is the density of the children of u after the relabeling? $\leq \frac{1}{T^i}$.
When we relabel u again, what is the density of its child? $\geq \frac{1}{T^{i-1}}$.
Therefore, between two consecutive relabelings of the same u we have inserted $(\frac{1}{T^{i-1}} - \frac{1}{T^i})2^{i-1}$ nodes into its subtree.
So we pay $\mathcal{O}(\frac{2^i}{T^i})$ once per $(\frac{1}{T^{i-1}} - \frac{1}{T^i})2^{i-1}$ insertions, so the amortized cost is $\mathcal{O}(1)$.
Then the amortized cost of an insertion is $\mathcal{O}(\log n)$, because every insertion charge $\mathcal{O}(\log n)$ nodes.

Second try

How much time to relabel a subtree rooted at u with 2^i leaves? $\mathcal{O}(\frac{2^i}{T^i})$.
What is the density of the children of u after the relabeling? $\leq \frac{1}{T^i}$.
When we relabel u again, what is the density of its child? $\geq \frac{1}{T^{i-1}}$.
Therefore, between two consecutive relabelings of the same u we have inserted $(\frac{1}{T^{i-1}} - \frac{1}{T^i})2^{i-1}$ nodes into its subtree.
So we pay $\mathcal{O}(\frac{2^i}{T^i})$ once per $(\frac{1}{T^{i-1}} - \frac{1}{T^i})2^{i-1}$ insertions, so the amortized cost is $\mathcal{O}(1)$.
Then the amortized cost of an insertion is $\mathcal{O}(\log n)$, because every insertion charge $\mathcal{O}(\log n)$ nodes.

Second try

How much time to relabel a subtree rooted at u with 2^i leaves? $\mathcal{O}(\frac{2^i}{T^i})$.
What is the density of the children of u after the relabeling? $\leq \frac{1}{T^i}$.
When we relabel u again, what is the density of its child? $\geq \frac{1}{T^{i-1}}$.
Therefore, between two consecutive relabelings of the same u we have inserted $(\frac{1}{T^{i-1}} - \frac{1}{T^i})2^{i-1}$ nodes into its subtree.
So we pay $\mathcal{O}(\frac{2^i}{T^i})$ once per $(\frac{1}{T^{i-1}} - \frac{1}{T^i})2^{i-1}$ insertions, so the amortized cost is $\mathcal{O}(1)$.
Then the amortized cost of an insertion is $\mathcal{O}(\log n)$, because every insertion charge $\mathcal{O}(\log n)$ nodes.

Final tuning

We want the cost of an insertion to be $\mathcal{O}(1)$. This can be achieved with a standard trick: indirection.

Indirection, again

Split the list into blocks of length $\log n$. Store the first element from every block using the previous method. Additionally, for every block use the first naive solution.

This gives amortized $\mathcal{O}(1)$ for all operations. Can be made worst-case, but then it gets really complicated.

Final tuning

We want the cost of an insertion to be $\mathcal{O}(1)$. This can be achieved with a standard trick: indirection.

Indirection, again

Split the list into blocks of length $\log n$. Store the first element from every block using the previous method. Additionally, for every block use the first naive solution.

This gives amortized $\mathcal{O}(1)$ for all operations. Can be made worst-case, but then it gets really complicated.

Final tuning

We want the cost of an insertion to be $\mathcal{O}(1)$. This can be achieved with a standard trick: indirection.

Indirection, again

Split the list into blocks of length $\log n$. Store the first element from every block using the previous method. Additionally, for every block use the first naive solution.

This gives amortized $\mathcal{O}(1)$ for all operations. Can be made worst-case, but then it gets really complicated.

Fully persistent array

Recall that we want to implement a persistent data structure with two operations:

1. update, which changes the value at a given address,
2. lookup, which returns the value at a given address.

As mentioned before, we think of all versions of the data structure using the notion of the version tree. So, both operations take a node of the version tree. Update attaches a new leaf there and we say that the leaf is labelled with the corresponding address.

Implementing lookup

To implement a lookup, we just need to find the lowest ancestor of the given node labelled with the address that we want to query with.

Fully persistent array

Recall that we want to implement a persistent data structure with two operations:

1. update, which changes the value at a given address,
2. lookup, which returns the value at a given address.

As mentioned before, we think of all versions of the data structure using the notion of the version tree. So, both operations take a node of the version tree. Update attaches a new leaf there and we say that the leaf is labelled with the corresponding address.

Implementing lookup

To implement a lookup, we just need to find the lowest ancestor of the given node labelled with the address that we want to query with.

Fully persistent array

Recall that we want to implement a persistent data structure with two operations:

1. update, which changes the value at a given address,
2. lookup, which returns the value at a given address.

As mentioned before, we think of all versions of the data structure using the notion of the version tree. So, both operations take a node of the version tree. Update attaches a new leaf there and we say that the leaf is labelled with the corresponding address.

Implementing lookup

To implement a lookup, we just need to find the lowest ancestor of the given node labelled with the address that we want to query with.

Euler tours

We reduced the problem to a simple tree question. Well, is it really simple?

Euler tour of a tree

Start at the root and traverse the tree in the natural order. When entering the node v for the first time, output v^+, and when leaving the node v and going to its parent (or terminating the whole process) output v^-.

This captures all information about the ancestors of a given node that we need in the following sense. Consider a subset of the Euler tour corresponding to nodes v labelled with the same address. To find the lowest ancestor of a given node labelled with that address, we just need to find its predecessor in the subset.

The predecessor itself doesn't have to be the answer. Do you see why and why this is not a problem?

Euler tours

We reduced the problem to a simple tree question. Well, is it really simple?

Euler tour of a tree

Start at the root and traverse the tree in the natural order. When entering the node v for the first time, output v^+, and when leaving the node v and going to its parent (or terminating the whole process) output v^-.

This captures all information about the ancestors of a given node that we need in the following sense. Consider a subset of the Euler tour corresponding to nodes v labelled with the same address. To find the lowest ancestor of a given node labelled with that address, we just need to find its predecessor in the subset.

The predecessor itself doesn't have to be the answer. Do you see why and why this is not a problem?

Euler tours

We reduced the problem to a simple tree question. Well, is it really simple?

Euler tour of a tree

Start at the root and traverse the tree in the natural order. When entering the node v for the first time, output v^+, and when leaving the node v and going to its parent (or terminating the whole process) output v^-.

This captures all information about the ancestors of a given node that we need in the following sense. Consider a subset of the Euler tour corresponding to nodes v labelled with the same address. To find the lowest ancestor of a given node labelled with that address, we just need to find its predecessor in the subset.

The predecessor itself doesn't have to be the answer. Do you see why and why this is not a problem?

Euler tours

We reduced the problem to a simple tree question. Well, is it really simple?

Euler tour of a tree

Start at the root and traverse the tree in the natural order. When entering the node v for the first time, output v^+, and when leaving the node v and going to its parent (or terminating the whole process) output v^-.

This captures all information about the ancestors of a given node that we need in the following sense. Consider a subset of the Euler tour corresponding to nodes v labelled with the same address. To find the lowest ancestor of a given node labelled with that address, we just need to find its predecessor in the subset.

The predecessor itself doesn't have to be the answer. Do you see why and why this is not a problem?

First try

So we only need to store a predecessor structure storing the subsets of the Euler tour corresponding to different addresses. Which implementation could we use here?

We need to compare any two elements of the Euler tour. This is a direct application of the order maintenance problem!

More precisely, we maintain an order maintenance structure for the whole Euler tour. It can be seen that whenever we create a new leaf, we know where to insert the new elements into the Euler tour. Also, now we can compare any two of its elements in constant time.

So, we can keep every subset in a BST, which gives us $\mathcal{O}(\log m)$ time per operation, where m is the number of updates so far.

First try

So we only need to store a predecessor structure storing the subsets of the Euler tour corresponding to different addresses. Which implementation could we use here?
We need to compare any two elements of the Euler tour. This is a direct application of the order maintenance problem!
More precisely, we maintain an order maintenance structure for the whole Euler tour. It can be seen that whenever we create a new leaf, we know where to insert the new elements into the Euler tour. Also, now we can compare any two of its elements in constant time.
So, we can keep every subset in a BST, which gives us $\mathcal{O}(\log m)$ time per operation, where m is the number of updates so far.

First try

So we only need to store a predecessor structure storing the subsets of the Euler tour corresponding to different addresses. Which implementation could we use here?
We need to compare any two elements of the Euler tour. This is a direct application of the order maintenance problem!
More precisely, we maintain an order maintenance structure for the whole Euler tour. It can be seen that whenever we create a new leaf, we know where to insert the new elements into the Euler tour. Also, now we can compare any two of its elements in constant time.
So, we can keep every subset in a BST, which gives us $\mathcal{O}(\log m)$ time per operation, where m is the number of updates so far.

First try

So we only need to store a predecessor structure storing the subsets of the Euler tour corresponding to different addresses. Which implementation could we use here?
We need to compare any two elements of the Euler tour. This is a direct application of the order maintenance problem!
More precisely, we maintain an order maintenance structure for the whole Euler tour. It can be seen that whenever we create a new leaf, we know where to insert the new elements into the Euler tour. Also, now we can compare any two of its elements in constant time.
So, we can keep every subset in a BST, which gives us $\mathcal{O}(\log m)$ time per operation, where m is the number of updates so far.

Second try

But we want $\mathcal{O}(\log \log m)$.

Clever idea

Observe that order maintenance gives us a mapping from the stored elements to $[1, 2^{\frac{\log 2m}{1-\log T}+1}] = [1, m^{\alpha}]$. So maybe instead of storing the elements in a BST, we could store their corresponding integers in a y-fast tree?

Yes we could! This gives $\mathcal{O}(\log \log m)$ query time. But what about the updates?

Second try

But we want $\mathcal{O}(\log \log m)$.

Clever idea

Observe that order maintenance gives us a mapping from the stored elements to $[1, 2^{\frac{\log 2m}{1-\log T}+1}] = [1, m^{\alpha}]$. So maybe instead of storing the elements in a BST, we could store their corresponding integers in a y-fast tree?

Yes we could! This gives $\mathcal{O}(\log \log m)$ query time. But what about the updates?

Second try

But we want $\mathcal{O}(\log \log m)$.

Clever idea

Observe that order maintenance gives us a mapping from the stored elements to $[1, 2^{\frac{\log 2m}{1-\log T}+1}] = [1, m^{\alpha}]$. So maybe instead of storing the elements in a BST, we could store their corresponding integers in a y-fast tree?

Yes we could! This gives $\mathcal{O}(\log \log m)$ query time. But what about the updates?

Update are more difficult. Inserting a new element into the order maintenance structure might change the integers assigned to even $\log m$ of the elements (the indirection doesn't really help here, do you see why?). How to fix this?

Another clever idea

Observe that now we are using indirection twice:

1. in every y-fast tree,
2. in the order maintenance structure.

Maybe we could put them together?

See the blackboard for an illustration of the whole situation.

Update are more difficult. Inserting a new element into the order maintenance structure might change the integers assigned to even $\log m$ of the elements (the indirection doesn't really help here, do you see why?). How to fix this?

Another clever idea

Observe that now we are using indirection twice:

1. in every y-fast tree,
2. in the order maintenance structure.

Maybe we could put them together?

See the blackboard for an illustration of the whole situation.

Update are more difficult. Inserting a new element into the order maintenance structure might change the integers assigned to even $\log m$ of the elements (the indirection doesn't really help here, do you see why?). How to fix this?

Another clever idea

Observe that now we are using indirection twice:

1. in every y-fast tree,
2. in the order maintenance structure.

Maybe we could put them together?

See the blackboard for an illustration of the whole situation.

For technical reasons, the indirection should split into fragments of size $\log^2 m$. Alternatively, use van Emde Boas trees instead of y-fast trees.

We say that an element is a *representative* if it is actually stored in some x-fast tree (i.e., it has been chosen in the y-fast tree as starting a new *bucket* of length $\log^2 m$).
The whole Euler tour is partitioned into *blocks* of length $\log^2 m$.

The invariant

Every block contains at most one representative.

Total number of representatives is $\frac{m}{\log^2 m}$, so now we update time becomes is amortized $\mathcal{O}(1)$ for updating the x-fast tree, plus $\mathcal{O}(\log \log m)$ to update the appropriate small BST.

For technical reasons, the indirection should split into fragments of size $\log^2 m$. Alternatively, use van Emde Boas trees instead of y-fast trees.

We say that an element is a *representative* if it is actually stored in some x-fast tree (i.e., it has been chosen in the y-fast tree as starting a new *bucket* of length $\log^2 m$).

The whole Euler tour is partitioned into *blocks* of length $\log^2 m$.

The invariant

Every block contains at most one representative.

Total number of representatives is $\frac{m}{\log^2 m}$, so now we update time becomes is amortized $\mathcal{O}(1)$ for updating the x-fast tree, plus $\mathcal{O}(\log \log m)$ to update the appropriate small BST.

For technical reasons, the indirection should split into fragments of size $\log^2 m$. Alternatively, use van Emde Boas trees instead of y-fast trees.

We say that an element is a *representative* if it is actually stored in some x-fast tree (i.e., it has been chosen in the y-fast tree as starting a new *bucket* of length $\log^2 m$).
The whole Euler tour is partitioned into *blocks* of length $\log^2 m$.

The invariant

Every block contains at most one representative.

Total number of representatives is $\frac{m}{\log^2 m}$, so now we update time becomes is amortized $\mathcal{O}(1)$ for updating the x-fast tree, plus $\mathcal{O}(\log \log m)$ to update the appropriate small BST.

For technical reasons, the indirection should split into fragments of size $\log^2 m$. Alternatively, use van Emde Boas trees instead of y-fast trees.

We say that an element is a *representative* if it is actually stored in some x-fast tree (i.e., it has been chosen in the y-fast tree as starting a new *bucket* of length $\log^2 m$).
The whole Euler tour is partitioned into *blocks* of length $\log^2 m$.

The invariant

Every block contains at most one representative.

Total number of representatives is $\frac{m}{\log^2 m}$, so now we update time becomes is amortized $\mathcal{O}(1)$ for updating the x-fast tree, plus $\mathcal{O}(\log \log m)$ to update the appropriate small BST.

For technical reasons, the indirection should split into fragments of size $\log^2 m$. Alternatively, use van Emde Boas trees instead of y-fast trees.

We say that an element is a *representative* if it is actually stored in some x-fast tree (i.e., it has been chosen in the y-fast tree as starting a new *bucket* of length $\log^2 m$).
The whole Euler tour is partitioned into *blocks* of length $\log^2 m$.

The invariant

Every block contains at most one representative.

Total number of representatives is $\frac{m}{\log^2 m}$, so now we update time becomes is amortized $\mathcal{O}(1)$ for updating the x-fast tree, plus $\mathcal{O}(\log \log m)$ to update the appropriate small BST.

So we proved the following:

Dietz 1989

Fully persistent array can be implemented in $\mathcal{O}(\log \log m)$ time for update and lookup, where m is the number of updates made so far. The lookup is deterministic, and the update is randomized because of hashing.

Lowerbound

One can ask if $\mathcal{O}(\log \log n)$ slowdown is necessary. Surprisingly, it is, even for partially persistent arrays.

Pătraşcu and Thorup 2007

Any (deterministic or randomized) data structure storing a set of n integers from $[1, n^2]$ in $\mathcal{O}(n \operatorname{polylog})$ space needs $\Omega(\log \log n)$ time for predecessor queries.

We will show that predecessor search can be reduced to storing a partially persistent array $A[1..n]$, so that predecessor queries take $\mathcal{O}(1)$ time plus one lookup. See the blackboard.

Lowerbound

One can ask if $\mathcal{O}(\log \log n)$ slowdown is necessary. Surprisingly, it is, even for partially persistent arrays.

Pătraşcu and Thorup 2007

Any (deterministic or randomized) data structure storing a set of n integers from $[1, n^2]$ in $\mathcal{O}(n \operatorname{polylog})$ space needs $\Omega(\log \log n)$ time for predecessor queries.

We will show that predecessor search can be reduced to storing a partially persistent array $A[1..n]$, so that predecessor queries take $\mathcal{O}(1)$ time plus one lookup. See the blackboard.

Lowerbound

One can ask if $\mathcal{O}(\log \log n)$ slowdown is necessary. Surprisingly, it is, even for partially persistent arrays.

Pătraşcu and Thorup 2007

Any (deterministic or randomized) data structure storing a set of n integers from $[1, n^2]$ in $\mathcal{O}(n \operatorname{polylog})$ space needs $\Omega(\log \log n)$ time for predecessor queries.

We will show that predecessor search can be reduced to storing a partially persistent array $A[1..n]$, so that predecessor queries take $\mathcal{O}(1)$ time plus one lookup. See the blackboard.

Questions?