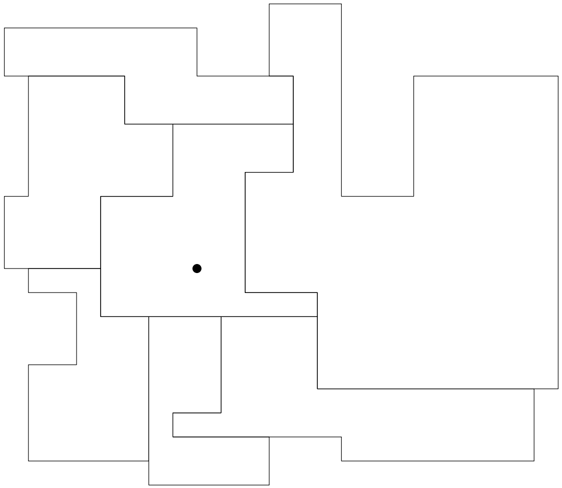
Order maintenance problem

Paweł Gawrychowski

30 czerwca 2014

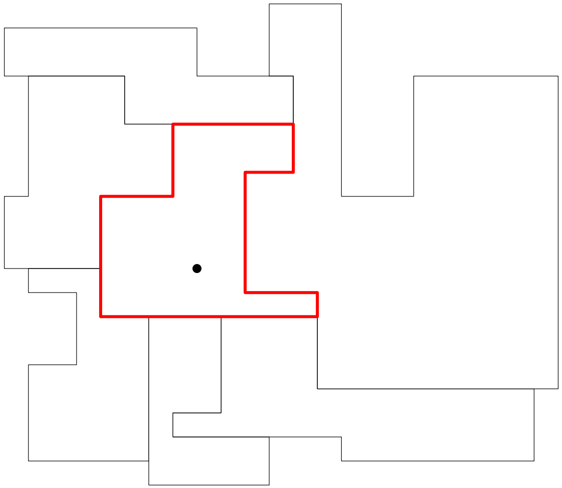
An application

Recall that we would like to solve the following problem: preprocess a collection of rectilinear polygons, so that given any point (x, y), we can quickly determine the smallest polygon it belongs to.



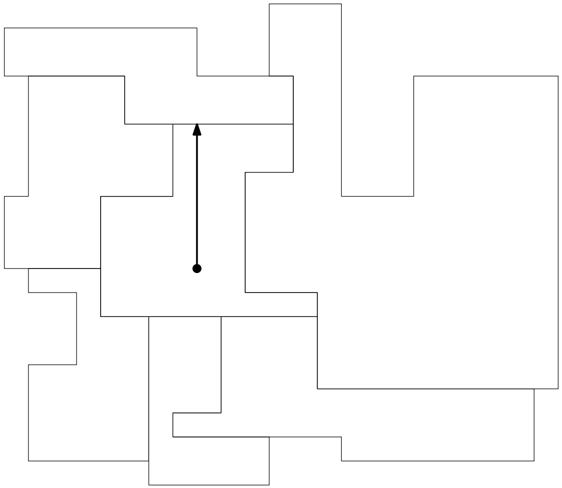
An application

Recall that we would like to solve the following problem: preprocess a collection of rectilinear polygons, so that given any point (x, y), we can quickly determine the smallest polygon it belongs to.



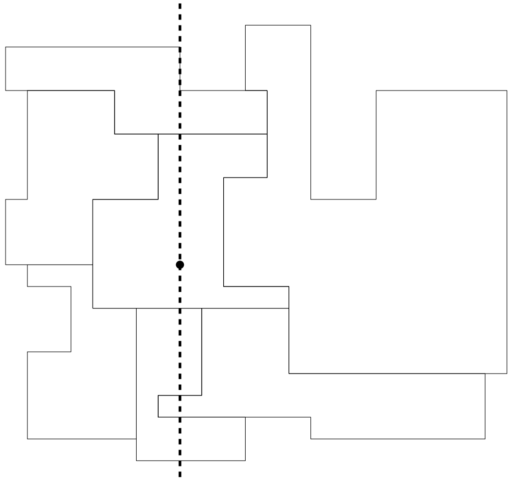
An application

Recall that we would like to solve the following problem: preprocess a collection of rectilinear polygons, so that given any point (x, y), we can quickly determine the smallest polygon it belongs to.



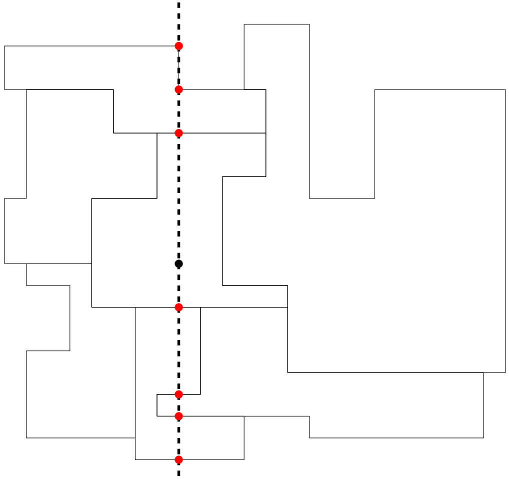
An application

Recall that we would like to solve the following problem: preprocess a collection of rectilinear polygons, so that given any point (x, y), we can quickly determine the smallest polygon it belongs to.



An application

Recall that we would like to solve the following problem: preprocess a collection of rectilinear polygons, so that given any point (x, y), we can quickly determine the smallest polygon it belongs to.



An application

Solution

For every essentially different x coordinate store a predecessor structure with the y coordinates of all intersections with the current line.

By essentially different we mean that some segment begins/ends there, i.e., the predecessor structure changes as we sweep from left to right with the current line. To answer an (x, y) query, we just locate the previous/next x where something changes and then find the predecessor/successor of y.

Storing a separate predecessor structure for every such x might still be very space consuming, but maybe the structure don't have to be separate? Their large parts will be similar, so we could hope to somehow share them between different structures.

An application

Solution

For every essentially different x coordinate store a predecessor structure with the y coordinates of all intersections with the current line.

By essentially different we mean that some segment begins/ends there, i.e., the predecessor structure changes as we sweep from left to right with the current line. To answer an (x, y) query, we just locate the previous/next x where something changes and then find the predecessor/successor of y.

Storing a separate predecessor structure for every such x might still be very space consuming, but maybe the structure don't have to be separate? Their large parts will be similar, so we could hope to somehow share them between different structures.

An application

Solution

For every essentially different x coordinate store a predecessor structure with the y coordinates of all intersections with the current line.

By essentially different we mean that some segment begins/ends there, i.e., the predecessor structure changes as we sweep from left to right with the current line. To answer an (x, y) query, we just locate the previous/next x where something changes and then find the predecessor/successor of y.
Storing a separate predecessor structure for every such x might still be very space consuming, but maybe the structure don't have to be separate? Their large parts will be similar, so we could hope to somehow share them between different structures.

Persistence

Say that we have a data structure allowing executing a number of queries, and performing a number of updates, which change its state. We would like to implement it so that each update creates a new version of the structure without destroying the old one.

Partial persistence

Each update creates a new copy of the structure. We can query any old version, but we can modify only the most recent one.

Full persistence

Each update creates a new copy of the structure. We can query and modify any version.

All versions of a partially persistent structure create a linear list, and all versions of a fully persistent structure create a tree called the version tree.

Persistence

Say that we have a data structure allowing executing a number of queries, and performing a number of updates, which change its state. We would like to implement it so that each update creates a new version of the structure without destroying the old one.

Partial persistence

Each update creates a new copy of the structure. We can query any old version, but we can modify only the most recent one.

Full persistence

Each update creates a new copy of the structure. We can query and modify any version.

All versions of a partially persistent structure create a linear list, and all versions of a fully persistent structure create a tree called the version tree.

Persistence

Say that we have a data structure allowing executing a number of queries, and performing a number of updates, which change its state. We would like to implement it so that each update creates a new version of the structure without destroying the old one.

Partial persistence

Each update creates a new copy of the structure. We can query any old version, but we can modify only the most recent one.

Full persistence

Each update creates a new copy of the structure. We can query and modify any version.

All versions of a partially persistent structure create a linear list, and all versions of a fully persistent structure create a tree called the version tree.

Persistence

Say that we have a data structure allowing executing a number of queries, and performing a number of updates, which change its state. We would like to implement it so that each update creates a new version of the structure without destroying the old one.

Partial persistence

Each update creates a new copy of the structure. We can query any old version, but we can modify only the most recent one.

Full persistence

Each update creates a new copy of the structure. We can query and modify any version.

All versions of a partially persistent structure create a linear list, and all versions of a fully persistent structure create a tree called the version tree.

Fully persistent stack

We start with a simple example: a fully persistent stack. We want to implement the following operations:

1. push a new object onto the stack,
2. pop the topmost object from the stack,
3. access the topmost object on the stack.

Every operation should return a pointer to the new version of the structure. More specifically, every operation takes a pointer to the version of the structure that we are interested in, and (if it is an update) returns a pointer to the new version of the structure.
We can implement a stack as a singly-linked list. Each element of the list is an object and a pointer to the next element. Now it turns out that we get full persistence for free by just not deleting the elements! See the blackboard.

Fully persistent stack

We start with a simple example: a fully persistent stack. We want to implement the following operations:

1. push a new object onto the stack,
2. pop the topmost object from the stack,
3. access the topmost object on the stack.

Every operation should return a pointer to the new version of the structure. More specifically, every operation takes a pointer to the version of the structure that we are interested in, and (if it is an update) returns a pointer to the new version of the structure.

We can implement a stack as a singly-linked list. Each element of the list is an object and a pointer to the next element. Now it turns out that we get full persistence for free by just not deleting the elements! See the blackboard.

Fully persistent stack

We start with a simple example: a fully persistent stack. We want to implement the following operations:

1. push a new object onto the stack,
2. pop the topmost object from the stack,
3. access the topmost object on the stack.

Every operation should return a pointer to the new version of the structure. More specifically, every operation takes a pointer to the version of the structure that we are interested in, and (if it is an update) returns a pointer to the new version of the structure.
We can implement a stack as a singly-linked list. Each element of the list is an object and a pointer to the next element. Now it turns out that we get full persistence for free by just not deleting the elements! See the blackboard.

Fully persistent stack

We start with a simple example: a fully persistent stack. We want to implement the following operations:

1. push a new object onto the stack,
2. pop the topmost object from the stack,
3. access the topmost object on the stack.

Every operation should return a pointer to the new version of the structure. More specifically, every operation takes a pointer to the version of the structure that we are interested in, and (if it is an update) returns a pointer to the new version of the structure.
We can implement a stack as a singly-linked list. Each element of the list is an object and a pointer to the next element. Now it turns out that we get full persistence for free by just not deleting the elements! See the blackboard.

Fully persistent BST

Now we move to a more complex example: a fully persistent balanced search tree. For the sake of concreteness, choose your favorite implementation, mine are red-black trees.

Important property of red-black trees

Any update takes $\mathcal{O}(\log n)$ time, meaning that it starts at the root and visits a subtree consisting of $\mathcal{O}(\log n)$ nodes. Only the nodes in the visited subtree are modified!

Hence after every update, just a small part of the tree changes. Therefore, instead of modifying the visited nodes, we can clone them, i.e., create their fresh copies. If a node is visited, but its (say left) child is not, the left pointer at the clone remains the same. Otherwise it points to the clone of the left child, see the blackboard.

Any reasonable balanced search tree can be made fully persistent without increasing the time complexity. The space complexity becomes as high as the time complexity, though!

Fully persistent BST

Now we move to a more complex example: a fully persistent balanced search tree. For the sake of concreteness, choose your favorite implementation, mine are red-black trees.

Important property of red-black trees

Any update takes $\mathcal{O}(\log n)$ time, meaning that it starts at the root and visits a subtree consisting of $\mathcal{O}(\log n)$ nodes. Only the nodes in the visited subtree are modified!

Hence after every update, just a small part of the tree changes. Therefore, instead of modifying the visited nodes, we can clone them, i.e., create their fresh copies. If a node is visited, but its (say left) child is not, the left pointer at the clone remains the same. Otherwise it points to the clone of the left child, see the blackboard.

Any reasonable balanced search tree can be made fully persistent without increasing the time complexity. The space complexity becomes as high as the time complexity, though!

Fully persistent BST

Now we move to a more complex example: a fully persistent balanced search tree. For the sake of concreteness, choose your favorite implementation, mine are red-black trees.

Important property of red-black trees

Any update takes $\mathcal{O}(\log n)$ time, meaning that it starts at the root and visits a subtree consisting of $\mathcal{O}(\log n)$ nodes. Only the nodes in the visited subtree are modified!

Hence after every update, just a small part of the tree changes. Therefore, instead of modifying the visited nodes, we can clone them, i.e., create their fresh copies. If a node is visited, but its (say left) child is not, the left pointer at the clone remains the same. Otherwise it points to the clone of the left child, see the blackboard.

Any reasonable balanced search tree can be made fully persistent without increasing the time complexity. The space complexity becomes as high as the time complexity, though!

Fully persistent BST

Now we move to a more complex example: a fully persistent balanced search tree. For the sake of concreteness, choose your favorite implementation, mine are red-black trees.

Important property of red-black trees

Any update takes $\mathcal{O}(\log n)$ time, meaning that it starts at the root and visits a subtree consisting of $\mathcal{O}(\log n)$ nodes. Only the nodes in the visited subtree are modified!

Hence after every update, just a small part of the tree changes. Therefore, instead of modifying the visited nodes, we can clone them, i.e., create their fresh copies. If a node is visited, but its (say left) child is not, the left pointer at the clone remains the same. Otherwise it points to the clone of the left child, see the blackboard.

Any reasonable balanced search tree can be made fully persistent without increasing the time complexity. The space complexity becomes as high as the time complexity, though!

Generalization

This can be actually generalized to any constant in-degree structure in the pointer machine model.

Pointer machine model

A structure in the pointer machine model is just a graph. Every node stores some information (for example, a number) and a constant number of pointers to other nodes of the graph. Additionally, one node of the graph is distinguished as the entry.

Driscoll, Sarnak, Sleator, Tarjan 1989

Any constant in-degree structure in the pointer machine model can be made partially persistent with $\mathcal{O}(1)$ amortized multiplicative overhead in the time complexity and $\mathcal{O}(1)$ amortized space per change.

Persistence in the Word RAM model

So we can efficiently achieve persistence in the pointer machine model. But we want to work in the Word RAM model! What then?

The main problem is the we need random access, i.e., arrays. This makes the in- and out-degree of the structure very high, and breaks the previous idea.

Nevertheless, we can do something. From now on we focus on implementing a persistent array. As soon as we can do that, we can make any random access structure persistent.

Partially persistent array

For every cell of the array, we store a list of pairs (timestamp,value). Modifying a cell is just appending a new pair to its list. Retrieving the value of a cell is just a predecessor search on its list.

...and we already know how to solve predecessor search REALLY efficiently, right? So we get $\mathcal{O}(\log \log m)$ time and $\mathcal{O}(m)$ space with dynamic perfect hashing, where m is the number of operations.

Persistence in the Word RAM model

So we can efficiently achieve persistence in the pointer machine model. But we want to work in the Word RAM model! What then?

The main problem is the we need random access, i.e., arrays. This makes the in- and out-degree of the structure very high, and breaks the previous idea.

Nevertheless, we can do something. From now on we focus on implementing a persistent array. As soon as we can do that, we can make any random access structure persistent.

Partially persistent array

For every cell of the array, we store a list of pairs (timestamp,value). Modifying a cell is just appending a new pair to its list. Retrieving the value of a cell is just a predecessor search on its list.

...and we already know how to solve predecessor search REALLY efficiently, right? So we get $\mathcal{O}(\log \log m)$ time and $\mathcal{O}(m)$ space with dynamic perfect hashing, where m is the number of operations.

Persistence in the Word RAM model

So we can efficiently achieve persistence in the pointer machine model. But we want to work in the Word RAM model! What then?

The main problem is the we need random access, i.e., arrays. This makes the in- and out-degree of the structure very high, and breaks the previous idea.

Nevertheless, we can do something. From now on we focus on implementing a persistent array. As soon as we can do that, we can make any random access structure persistent.

Partially persistent array

For every cell of the array, we store a list of pairs (timestamp,value). Modifying a cell is just appending a new pair to its list. Retrieving the value of a cell is just a predecessor search on its list.

...and we already know how to solve predecessor search REALLY efficiently, right? So we get $\mathcal{O}(\log\log m)$ time and $\mathcal{O}(m)$ space with dynamic perfect hashing, where m is the number of operations.

Persistence in the Word RAM model

So we can efficiently achieve persistence in the pointer machine model. But we want to work in the Word RAM model! What then?

The main problem is the we need random access, i.e., arrays. This makes the in- and out-degree of the structure very high, and breaks the previous idea.

Nevertheless, we can do something. From now on we focus on implementing a persistent array. As soon as we can do that, we can make any random access structure persistent.

Partially persistent array

For every cell of the array, we store a list of pairs (timestamp,value). Modifying a cell is just appending a new pair to its list. Retrieving the value of a cell is just a predecessor search on its list.

...and we already know how to solve predecessor search REALLY efficiently, right? So we get $\mathcal{O}(\log\log m)$ time and $\mathcal{O}(m)$ space with dynamic perfect hashing, where m is the number of operations.

Persistence in the Word RAM model

So we can efficiently achieve persistence in the pointer machine model. But we want to work in the Word RAM model! What then?

The main problem is the we need random access, i.e., arrays. This makes the in- and out-degree of the structure very high, and breaks the previous idea.

Nevertheless, we can do something. From now on we focus on implementing a persistent array. As soon as we can do that, we can make any random access structure persistent.

Partially persistent array

For every cell of the array, we store a list of pairs (timestamp,value). Modifying a cell is just appending a new pair to its list. Retrieving the value of a cell is just a predecessor search on its list.

...and we already know how to solve predecessor search REALLY efficiently, right? So we get $\mathcal{O}(\log \log m)$ time and $\mathcal{O}(m)$ space with dynamic perfect hashing, where m is the number of operations.

Order maintenance problem

To generalize this simple idea to full persistence, 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 generalize this simple idea to full persistence, 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 generalize this simple idea to full persistence, 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 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 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:

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

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:

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

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:

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

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:

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

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.

Questions?