Exponential search trees

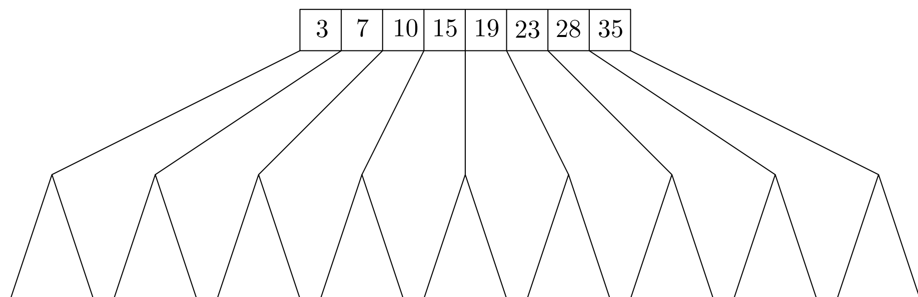
Paweł Gawrychowski

12 czerwca 2014

Fusion trees

Recall that a fusion tree was a B-tree with $B = w^{1/5}$ built on a subsets of $\frac{n}{w^4}$ keys. The query/update time is $\mathcal{O}(\frac{\log n}{\log w} + \log w)$ and we would like to do better. But first, let us look closely at how to update B-tree.

B-trees



Every non-root node stores at least $B-1$ and at most $2(B-1)$ numbers. The root stores at most $2B-1$ numbers.

Inserting into a B-tree

We start with traversing the tree down until we find an appropriate leaf. Then we add the new element there. As a result, the leaf might overflow, and we must fix this. Say that $B = 5$.

3	7	10	15	19	23	28	35

9

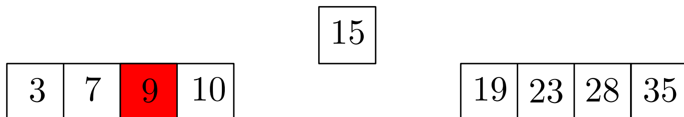
Inserting into a B-tree

We start with traversing the tree down until we find an appropriate leaf. Then we add the new element there. As a result, the leaf might overflow, and we must fix this. Say that $B = 5$.

3	7	9	10	15	19	23	28	35

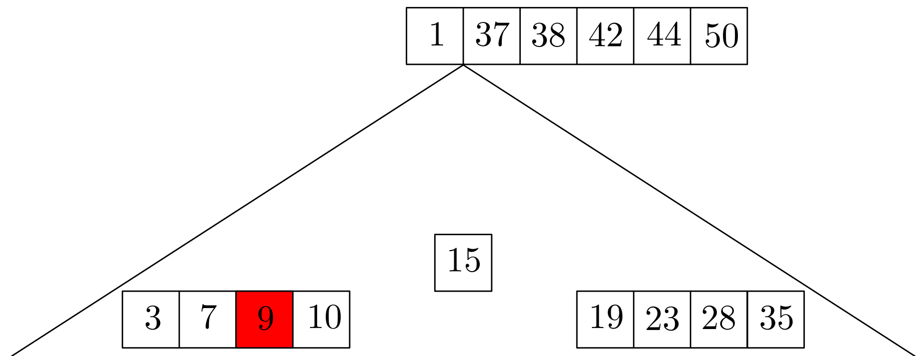
Inserting into a B-tree

We start with traversing the tree down until we find an appropriate leaf. Then we add the new element there. As a result, the leaf might overflow, and we must fix this. Say that $B = 5$.



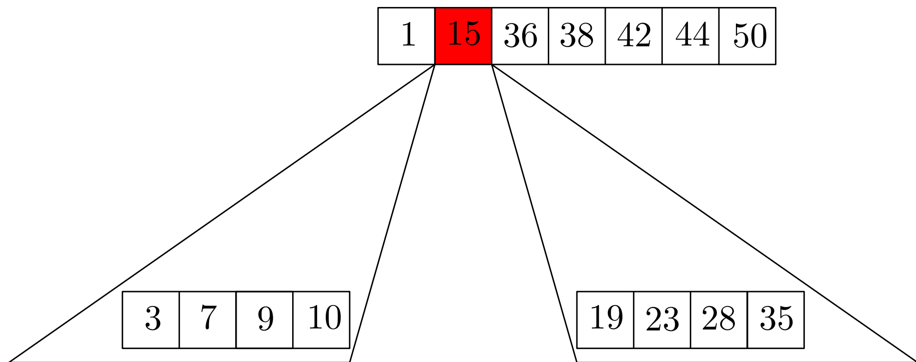
Inserting into a B-tree

We start with traversing the tree down until we find an appropriate leaf. Then we add the new element there. As a result, the leaf might overflow, and we must fix this. Say that $B = 5$.



Inserting into a B-tree

We start with traversing the tree down until we find an appropriate leaf. Then we add the new element there. As a result, the leaf might overflow, and we must fix this. Say that $B = 5$.



Inserting into a B-tree

This actually is the whole idea: given a node storing $2B - 1$ numbers we can split it into two and move the excess up. If the parent now stores $2B - 1$ numbers, we repeat the procedure. If not, we are done. Sooner or later we will either be done, or we will move to the very root of the whole tree. In such case, we create a new root.

Inserting into a B-tree

This actually is the whole idea: given a node storing $2B-1$ numbers we can split it into two and move the excess up. If the parent now stores $2B-1$ numbers, we repeat the procedure. If not, we are done. Sooner or later we will either be done, or we will move to the very root of the whole tree. In such case, we create a new root.

Inserting into a B-tree

This actually is the whole idea: given a node storing $2B-1$ numbers we can split it into two and move the excess up. If the parent now stores $2B-1$ numbers, we repeat the procedure. If not, we are done. Sooner or later we will either be done, or we will move to the very root of the whole tree. In such case, we create a new root.

Deleting from B-tree

More messy! But let's try anyway.

First we locate the number which should be removed by traversing starting from the root. Remove it from the node it belongs to and fill the hole with either the largest number from the left subtree or the smallest number from the right subtree.

Deleting from B-tree

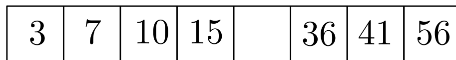
More messy! But let’s try anyway.

First we locate the number which should be removed by traversing starting from the root. Remove it from the node it belongs to and fill the hole with either the largest number from the left subtree or the smallest number from the right subtree.

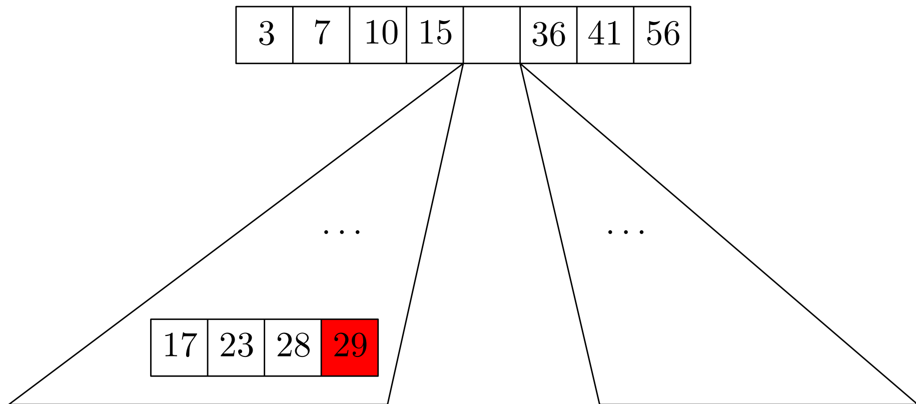
Deleting from B-tree

3	7	10	15	31	36	41	56

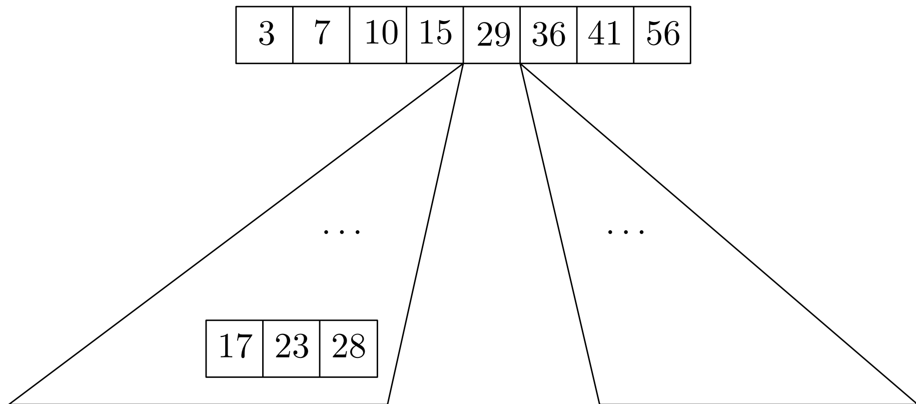
Deleting from B-tree



Deleting from B-tree

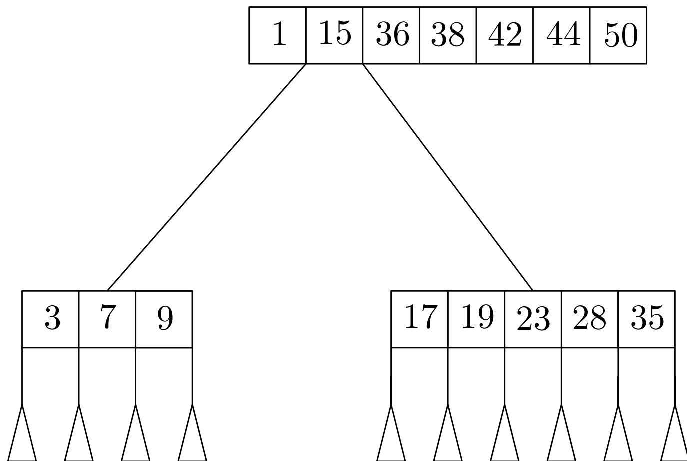


Deleting from B-tree



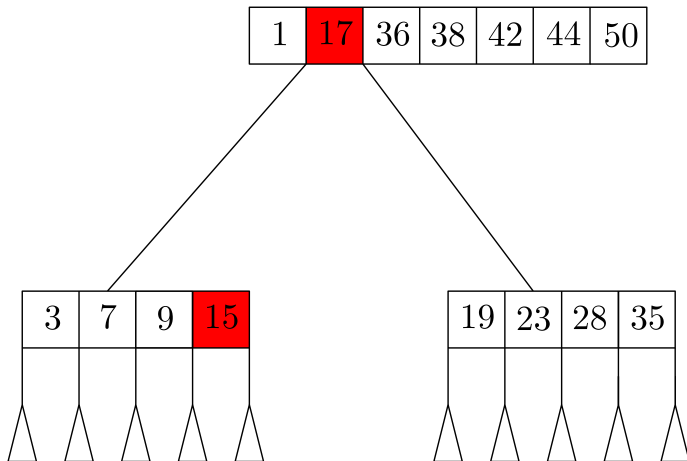
Deleting from B-tree

Now we have a B-tree where some leaf might contain just $B - 2$ numbers, so we must be able to fix this.



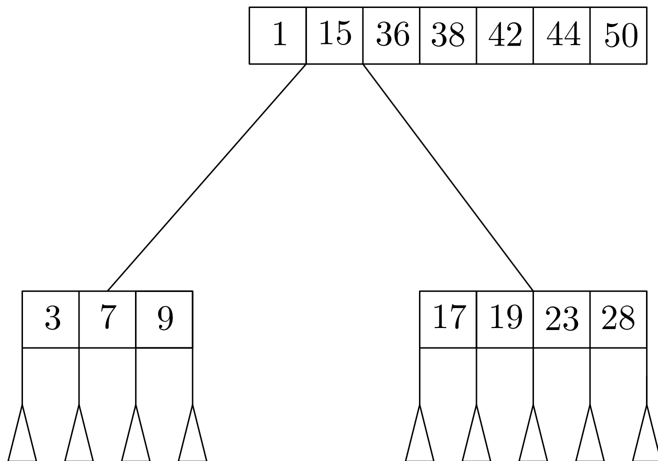
Deleting from B-tree

Now we have a B-tree where some leaf might contain just $B - 2$ numbers, so we must be able to fix this.



Deleting from B-tree

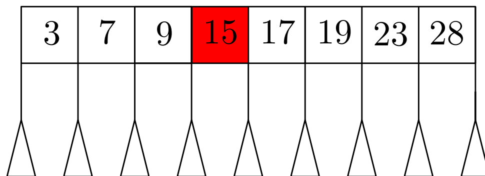
Now we have a B-tree where some leaf might contain just $B - 2$ numbers, so we must be able to fix this.



Deleting from B-tree

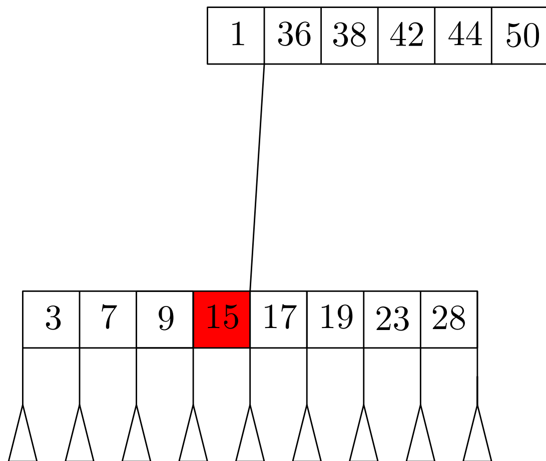
Now we have a B-tree where some leaf might contain just $B - 2$ numbers, so we must be able to fix this.

1		36	38	42	44	50



Deleting from B-tree

Now we have a B-tree where some leaf might contain just $B-2$ numbers, so we must be able to fix this.



Deleting from B-tree

Given a node storing $B - 2$ numbers we look at its sibling. If the sibling stores at least B numbers, we borrow one of them (together with the corresponding child of the sibling). If the sibling stores exactly $B - 1$ numbers, we merge two nodes, sandwiching their nodes with the corresponding number stored at the parent. Then the parent might become a node storing just $B - 2$ numbers, so we repeat the procedure there. Sooner or later we will either be done, or we will move to the very root of the whole tree.

Deleting from B-tree

Given a node storing $B - 2$ numbers we look at its sibling. If the sibling stores at least B numbers, we borrow one of them (together with the corresponding child of the sibling). If the sibling stores exactly $B - 1$ numbers, we merge two nodes, sandwiching their nodes with the corresponding number stored at the parent. Then the parent might become a node storing just $B - 2$ numbers, so we repeat the procedure there. Sooner or later we will either be done, or we will move to the very root of the whole tree.

Deleting from B-tree

Given a node storing $B - 2$ numbers we look at its sibling. If the sibling stores at least B numbers, we borrow one of them (together with the corresponding child of the sibling). If the sibling stores exactly $B - 1$ numbers, we merge two nodes, sandwiching their nodes with the corresponding number stored at the parent. Then the parent might become a node storing just $B - 2$ numbers, so we repeat the procedure there. Sooner or later we will either be done, or we will move to the very root of the whole tree.

Deleting from B-tree

Given a node storing $B-2$ numbers we look at its sibling. If the sibling stores at least B numbers, we borrow one of them (together with the corresponding child of the sibling). If the sibling stores exactly $B-1$ numbers, we merge two nodes, sandwiching their nodes with the corresponding number stored at the parent. Then the parent might become a node storing just $B-2$ numbers, so we repeat the procedure there. Sooner or later we will either be done, or we will move to the very root of the whole tree.

The goal

We want a generic transformation, which takes a static predecessor structure, and constructs a dynamic predecessor structure.

Exponential search trees

Assume that there is a static predecessor structure, which can be constructed in $\mathcal{O}(d^{k-1})$ ($k \geq 2$) time and space given d keys, and then queried in $S(d)$ time. Then we can implement a dynamic linear space predecessor structure storing n keys with insert, delete, and search in $T(n)$ time, where:

$$T(n) \leq \mathcal{O}(S(n)) + T(n^{1-1/k}).$$

A simple consequence: let the static structure be a static fusion tree without the indirection. It can be constructed in, say, $\mathcal{O}(d^4)$ time and answers queries in $\mathcal{O}(1 + \frac{\log d}{\log w})$ time (notice 1+). How fast is the resulting dynamic structure?

The goal

We want a generic transformation, which takes a static predecessor structure, and constructs a dynamic predecessor structure.

Exponential search trees

Assume that there is a static predecessor structure, which can be constructed in $\mathcal{O}(d^{k-1})$ ($k \geq 2$) time and space given d keys, and then queried in $S(d)$ time. Then we can implement a dynamic linear space predecessor structure storing n keys with insert, delete, and search in $T(n)$ time, where:

$$T(n) \leq \mathcal{O}(S(n)) + T(n^{1-1/k}).$$

A simple consequence: let the static structure be a static fusion tree without the indirection. It can be constructed in, say, $\mathcal{O}(d^4)$ time and answers queries in $\mathcal{O}(1 + \frac{\log d}{\log w})$ time (notice 1+). How fast is the resulting dynamic structure?

The goal

We want a generic transformation, which takes a static predecessor structure, and constructs a dynamic predecessor structure.

Exponential search trees

Assume that there is a static predecessor structure, which can be constructed in $\mathcal{O}(d^{k-1})$ ($k \geq 2$) time and space given d keys, and then queried in $S(d)$ time. Then we can implement a dynamic linear space predecessor structure storing n keys with insert, delete, and search in $T(n)$ time, where:

$$T(n) \leq \mathcal{O}(S(n)) + T(n^{1-1/k}).$$

A simple consequence: let the static structure be a static fusion tree without the indirection. It can be constructed in, say, $\mathcal{O}(d^4)$ time and answers queries in $\mathcal{O}(1 + \frac{\log d}{\log w})$ time (notice $1+$). How fast is the resulting dynamic structure?

The idea

We maintain a multiway search tree, where the leaves correspond to the keys in the current set.

More precisely:

1. the root stores $d = \Theta(n^{1/k})$ splitters in a static structure,
2. the splitters partition all keys into $d + 1$ ranges, the i-th child is a recursive structure storing all the keys in the i-th range,
3. each child stores $\Theta(n^{1-1/k})$ keys.

The splitters might or might not be in the current set.

The idea

We maintain a multiway search tree, where the leaves correspond to the keys in the current set.

More precisely:

1. the root stores $d = \Theta(n^{1/k})$ splitters in a static structure,
2. the splitters partition all keys into $d+1$ ranges, the i-th child is a recursive structure storing all the keys in the i-th range,
3. each child stores $\Theta(n^{1-1/k})$ keys.

The splitters might or might not be in the current set.

The idea

We maintain a multiway search tree, where the leaves correspond to the keys in the current set.

More precisely:

1. the root stores $d = \Theta(n^{1/k})$ splitters in a static structure,
2. the splitters partition all keys into $d + 1$ ranges, the i-th child is a recursive structure storing all the keys in the i-th range,
3. each child stores $\Theta(n^{1-1/k})$ keys.

The splitters might or might not be in the current set.

The idea

We maintain a multiway search tree, where the leaves correspond to the keys in the current set.

More precisely:

1. the root stores $d = \Theta(n^{1/k})$ splitters in a static structure,
2. the splitters partition all keys into $d + 1$ ranges, the i-th child is a recursive structure storing all the keys in the i-th range,
3. each child stores $\Theta(n^{1-1/k})$ keys.

The splitters might or might not be in the current set.

The idea

We maintain a multiway search tree, where the leaves correspond to the keys in the current set.

More precisely:

1. the root stores $d = \Theta(n^{1/k})$ splitters in a static structure,
2. the splitters partition all keys into $d + 1$ ranges, the i-th child is a recursive structure storing all the keys in the i-th range,
3. each child stores $\Theta(n^{1-1/k})$ keys.

The splitters might or might not be in the current set.

The search

To find the leaf corresponding to a given number, we start at the root and traverse down using the static structures to determine the appropriate child. What is the total time?

$$T(n) = \mathcal{O}(S(n^{1/k})) + T(\mathcal{O}(n^{1-1/k})) = \mathcal{O}(S(n)) + T(n^{1-1/k})$$

(see the blackboard why)

Now consider the case when the given number is currently not in our set, and we want to find its predecessor. As previously, we traverse the tree down. Then we take the last visited node which is not the leftmost among its siblings, and look at its left brother/sister. The largest number there is our predecessor. So, we need to either store the largest number in every subtree, or traverse the sibling.

We implicitly used the fast that every subtree is nonempty. Otherwise, the left brother/sister might contain no keys inside!

The search

To find the leaf corresponding to a given number, we start at the root and traverse down using the static structures to determine the appropriate child. What is the total time?

$$T(n) = \mathcal{O}(S(n^{1/k})) + T(\mathcal{O}(n^{1-1/k})) = \mathcal{O}(S(n)) + T(n^{1-1/k})$$

(see the blackboard why)

Now consider the case when the given number is currently not in our set, and we want to find its predecessor. As previously, we traverse the tree down. Then we take the last visited node which is not the leftmost among its siblings, and look at its left brother/sister. The largest number there is our predecessor. So, we need to either store the largest number in every subtree, or traverse the sibling.

We implicitly used the fast that every subtree is nonempty. Otherwise, the left brother/sister might contain no keys inside!

The search

To find the leaf corresponding to a given number, we start at the root and traverse down using the static structures to determine the appropriate child. What is the total time?

$$T(n) = \mathcal{O}(S(n^{1/k})) + T(\mathcal{O}(n^{1-1/k})) = \mathcal{O}(S(n)) + T(n^{1-1/k})$$

(see the blackboard why)

Now consider the case when the given number is currently not in our set, and we want to find its predecessor. As previously, we traverse the tree down. Then we take the last visited node which is not the leftmost among its siblings, and look at its left brother/sister. The largest number there is our predecessor. So, we need to either store the largest number in every subtree, or traverse the sibling.

We implicitly used the fast that every subtree is nonempty. Otherwise, the left brother/sister might contain no keys inside!

The search

To find the leaf corresponding to a given number, we start at the root and traverse down using the static structures to determine the appropriate child. What is the total time?

$$T(n) = \mathcal{O}(S(n^{1/k})) + T(\mathcal{O}(n^{1-1/k})) = \mathcal{O}(S(n)) + T(n^{1-1/k})$$

(see the blackboard why)

Now consider the case when the given number is currently not in our set, and we want to find its predecessor. As previously, we traverse the tree down. Then we take the last visited node which is not the leftmost among its siblings, and look at its left brother/sister. The largest number there is our predecessor. So, we need to either store the largest number in every subtree, or traverse the sibling.

We implicitly used the fast that every subtree is nonempty. Otherwise, the left brother/sister might contain no keys inside!

The size

What is the size of the whole structure? The static structures are large, but they store just a few keys....

Let n_i be the number of keys in the i-th subtree. Then:

1. $n = \sum_{i=1}^{d+1} n_i$
2. $n_i = \Theta(n^{1-1/k})$,
3. the root uses space $\mathcal{O}(d^{k-1}) = \mathcal{O}(n^{1-1/k})$, because $d = \Theta(n^{1/k})$.

The total space is:

$$C(n) = \mathcal{O}(n^{1-1/k}) + \sum_i C(n_i) = \mathcal{O}(n)$$

(see the blackboard why)

This also shows that the whole exponential search tree can be constructed in linear time. But how to update it efficiently?

The size

What is the size of the whole structure? The static structures are large, but they store just a few keys....
Let n_i be the number of keys in the i-th subtree. Then:

1. $n = \sum_{i=1}^{d+1} n_i$
2. $n_i = \Theta(n^{1-1/k})$,
3. the root uses space $\mathcal{O}(d^{k-1}) = \mathcal{O}(n^{1-1/k})$, because $d = \Theta(n^{1/k})$.

The total space is:

$$C(n) = \mathcal{O}(n^{1-1/k}) + \sum_i C(n_i) = \mathcal{O}(n)$$

(see the blackboard why)

This also shows that the whole exponential search tree can be constructed in linear time. But how to update it efficiently?

The size

What is the size of the whole structure? The static structures are large, but they store just a few keys....
Let n_i be the number of keys in the i-th subtree. Then:

1. $n = \sum_{i=1}^{d+1} n_i$
2. $n_i = \Theta(n^{1-1/k})$,
3. the root uses space $\mathcal{O}(d^{k-1}) = \mathcal{O}(n^{1-1/k})$, because $d = \Theta(n^{1/k})$.

The total space is:

$$C(n) = \mathcal{O}(n^{1-1/k}) + \sum_i C(n_i) = \mathcal{O}(n)$$

(see the blackboard why)

This also shows that the whole exponential search tree can be constructed in linear time. But how to update it efficiently?

The size

What is the size of the whole structure? The static structures are large, but they store just a few keys....
Let n_i be the number of keys in the i-th subtree. Then:

1. $n = \sum_{i=1}^{d+1} n_i$
2. $n_i = \Theta(n^{1-1/k})$,
3. the root uses space $\mathcal{O}(d^{k-1}) = \mathcal{O}(n^{1-1/k})$, because $d = \Theta(n^{1/k})$.

The total space is:

$$C(n) = \mathcal{O}(n^{1-1/k}) + \sum_i C(n_i) = \mathcal{O}(n)$$

(see the blackboard why)

This also shows that the whole exponential search tree can be constructed in linear time. But how to update it efficiently?

The update

Weights

The weight $w(v)$ of a node v is the number of leaves in its subtree, i.e., the number of stored keys.

We only need to make sure that the weight of every child is $\Theta(n^{1-1/k})$. The bound on the number of children will follow automatically.

The update

Weights

The weight $w(v)$ of a node v is the number of leaves in its subtree, i.e., the number of stored keys.

We only need to make sure that the weight of every child is $\Theta(n^{1-1/k})$. The bound on the number of children will follow automatically.

Insertions only

When the weight of a child becomes $2n^{1-1/k}$, we split it into two. What do we need to do?

1. Construct these two new subtrees.
2. Rebuild the whole static structure stored at the node.

Both steps need $\mathcal{O}(n^{1-1/k})$ time. We amortize this by defining the potential of each child v to be $w(v) - n^{1-1/k}$. Then, when we need to rebuild, we can use $n^{1-1/k}$ credits to pay for the expensive rebuilding.

Maintaining the potential

Whenever we insert a new key, we increase the weight of up to $\log \log n$ nodes. This is at least as large as the cost of the traversal anyway, so we can afford to allocate the new credits.

Insertions only

When the weight of a child becomes $2n^{1-1/k}$, we split it into two. What do we need to do?

1. Construct these two new subtrees.
2. Rebuild the whole static structure stored at the node.

Both steps need $\mathcal{O}(n^{1-1/k})$ time. We amortize this by defining the potential of each child v to be $w(v) - n^{1-1/k}$. Then, when we need to rebuild, we can use $n^{1-1/k}$ credits to pay for the expensive rebuilding.

Maintaining the potential

Whenever we insert a new key, we increase the weight of up to $\log\log n$ nodes. This is at least as large as the cost of the traversal anyway, so we can afford to allocate the new credits.

Insertions only

When the weight of a child becomes $2n^{1-1/k}$, we split it into two. What do we need to do?

1. Construct these two new subtrees.
2. Rebuild the whole static structure stored at the node.

Both steps need $\mathcal{O}(n^{1-1/k})$ time. We amortize this by defining the potential of each child v to be $w(v) - n^{1-1/k}$. Then, when we need to rebuild, we can use $n^{1-1/k}$ credits to pay for the expensive rebuilding.

Maintaining the potential

Whenever we insert a new key, we increase the weight of up to $\log\log n$ nodes. This is at least as large as the cost of the traversal anyway, so we can afford to allocate the new credits.

Insertions only

When the weight of a child becomes $2n^{1-1/k}$, we split it into two. What do we need to do?

1. Construct these two new subtrees.
2. Rebuild the whole static structure stored at the node.

Both steps need $\mathcal{O}(n^{1-1/k})$ time. We amortize this by defining the potential of each child v to be $w(v) - n^{1-1/k}$. Then, when we need to rebuild, we can use $n^{1-1/k}$ credits to pay for the expensive rebuilding.

Maintaining the potential

Whenever we insert a new key, we increase the weight of up to $\log \log n$ nodes. This is at least as large as the cost of the traversal anyway, so we can afford to allocate the new credits.

Insertions only

When the weight of a child becomes $2n^{1-1/k}$, we split it into two. What do we need to do?

1. Construct these two new subtrees.
2. Rebuild the whole static structure stored at the node.

Both steps need $\mathcal{O}(n^{1-1/k})$ time. We amortize this by defining the potential of each child v to be $w(v) - n^{1-1/k}$. Then, when we need to rebuild, we can use $n^{1-1/k}$ credits to pay for the expensive rebuilding.

Maintaining the potential

Whenever we insert a new key, we increase the weight of up to $\log \log n$ nodes. This is at least as large as the cost of the traversal anyway, so we can afford to allocate the new credits.

Insertions and deletions

When the weight becomes $\frac{1}{2}n^{1-1/k}$, try to merge with one of the neighbors. We skip the details.

Beame&Fich made dynamic

Recall that the static structure of Beame and Fich was REALLY static. So, maybe we should apply the new exponential hammer here?

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.

We use the above for the static structure. Then the complexity of any operation becomes:

$$T(n) = \mathcal{O}(\frac{\log\log U}{\log\log\log U}) + T(n^{1/2})$$

which solves to $\mathcal{O}(\log\log n \frac{\log\log U}{\log\log\log U})$. This is the best known bound for deterministic linear space structures! (when U is small)

Beame&Fich made dynamic

Recall that the static structure of Beame and Fich was REALLY static. So, maybe we should apply the new exponential hammer here?

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.

We use the above for the static structure. Then the complexity of any operation becomes:

$$T(n) = \mathcal{O}(\frac{\log\log U}{\log\log\log U}) + T(n^{1/2})$$

which solves to $\mathcal{O}(\log\log n \frac{\log\log U}{\log\log\log U})$. This is the best known bound for deterministic linear space structures! (when U is small)

An application

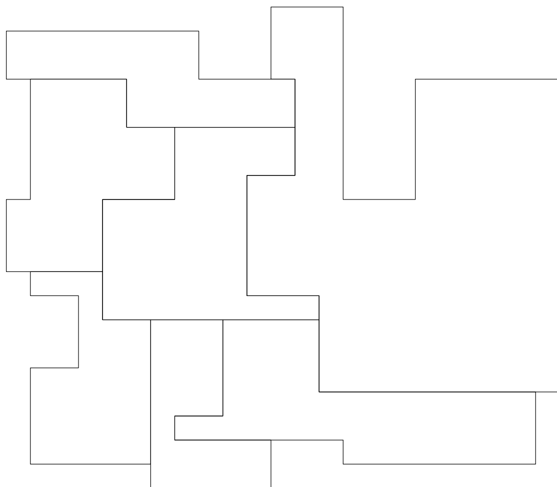
We want to preprocess a map, so that given a point, we can quickly determine the country it belongs to.

An application

We want to preprocess a map, so that given a point, we can quickly determine the country it belongs to.

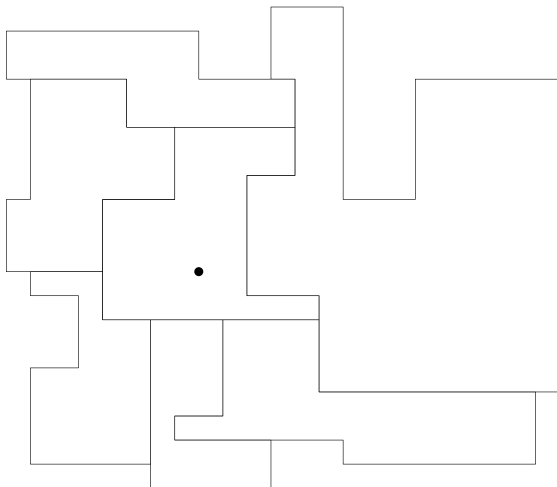
An application, simplified

Approximate each country with a rectilinear polygon.



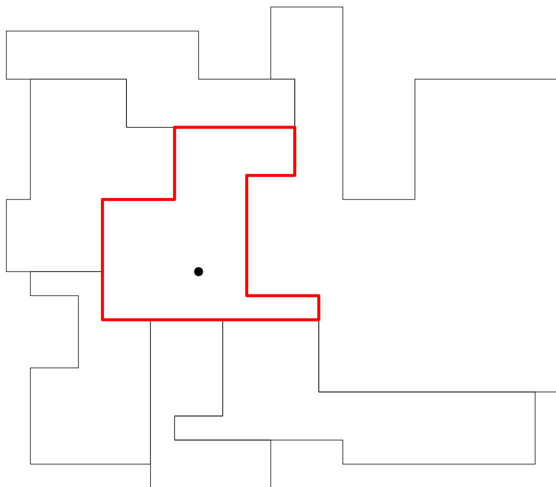
An application, simplified

Approximate each country with a rectilinear polygon.



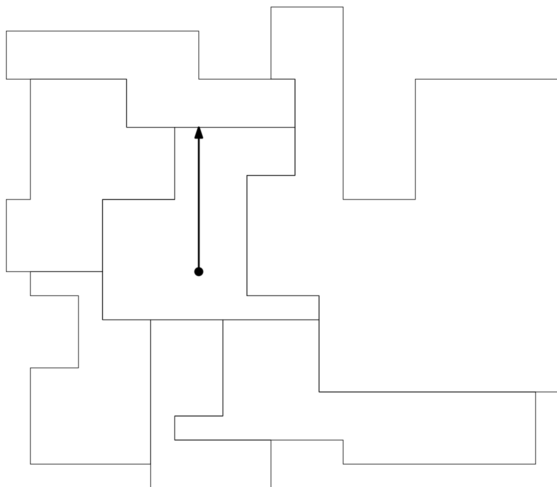
An application, simplified

Approximate each country with a rectilinear polygon.



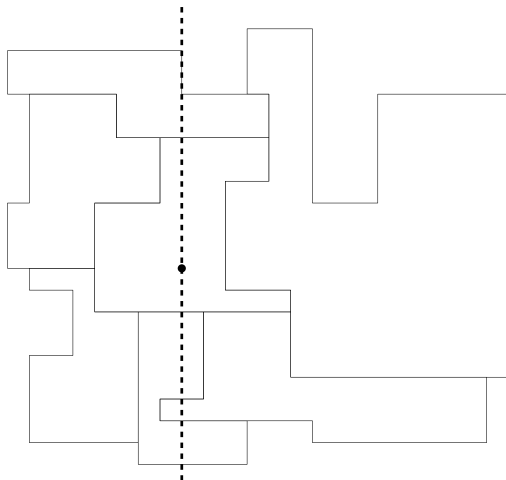
An application, simplified

Approximate each country with a rectilinear polygon.



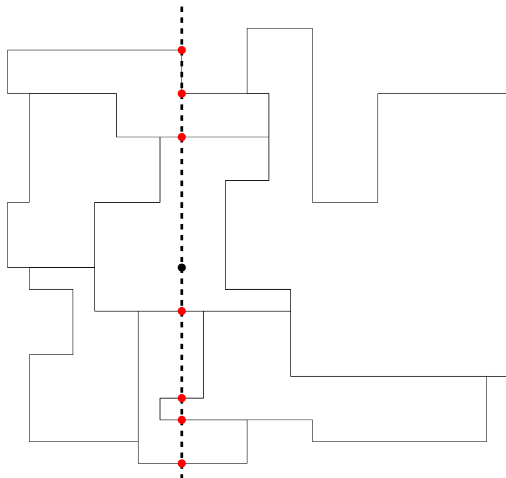
An application, simplified

Approximate each country with a rectilinear polygon.



An application, simplified

Approximate each country with a rectilinear polygon.



Solution

For each possible x coordinate, store a predecessor structure storing the y coordinates of all intersections with the current line.

We actually only need to consider just the "essentially different" x coordinates, which are the ones where something changes.
So, we could sweep from left to right over the interesting x coordinates, and maintain a predecessor structure. This allows us to answer a single query.
But we want to build some structure allowing us to (later) answer many queries efficiently. Maybe we could store each "version" of the predecessor structure?

Partially persistent predecessor search

We want to implement a predecessor structure, so that each insert/delete creates a new copy of the structure. The old copy can be queried, but cannot be modified.

Solution

For each possible x coordinate, store a predecessor structure storing the y coordinates of all intersections with the current line.

We actually only need to consider just the "essentially different" x coordinates, which are the ones where something changes.

So, we could sweep from left to right over the interesting x coordinates, and maintain a predecessor structure. This allows us to answer a single query.

But we want to build some structure allowing us to (later) answer many queries efficiently. Maybe we could store each "version" of the predecessor structure?

Partially persistent predecessor search

We want to implement a predecessor structure, so that each insert/delete creates a new copy of the structure. The old copy can be queried, but cannot be modified.

Solution

For each possible x coordinate, store a predecessor structure storing the y coordinates of all intersections with the current line.

We actually only need to consider just the “essentially different” x coordinates, which are the ones where something changes.
So, we could sweep from left to right over the interesting x coordinates, and maintain a predecessor structure. This allows us to answer a single query.
But we want to build some structure allowing us to (later) answer many queries efficiently. Maybe we could store each “version” of the predecessor structure?

Partially persistent predecessor search

We want to implement a predecessor structure, so that each insert/delete creates a new copy of the structure. The old copy can be queried, but cannot be modified.

Solution

For each possible x coordinate, store a predecessor structure storing the y coordinates of all intersections with the current line.

We actually only need to consider just the “essentially different” x coordinates, which are the ones where something changes.
So, we could sweep from left to right over the interesting x coordinates, and maintain a predecessor structure. This allows us to answer a single query.
But we want to build some structure allowing us to (later) answer many queries efficiently. Maybe we could store each “version” of the predecessor structure?

Partially persistent predecessor search

We want to implement a predecessor structure, so that each insert/delete creates a new copy of the structure. The old copy can be queried, but cannot be modified.

Solution

For each possible x coordinate, store a predecessor structure storing the y coordinates of all intersections with the current line.

We actually only need to consider just the "essentially different" x coordinates, which are the ones where something changes.
So, we could sweep from left to right over the interesting x coordinates, and maintain a predecessor structure. This allows us to answer a single query.
But we want to build some structure allowing us to (later) answer many queries efficiently. Maybe we could store each "version" of the predecessor structure?

Partially persistent predecessor search

We want to implement a predecessor structure, so that each insert/delete creates a new copy of the structure. The old copy can be queried, but cannot be modified.

Questions?