Alphabet-Dependent String Searching with Wexponential Search Trees

Johannes Fischer[1] Paweł Gawrychowski[2]

[1]TU Dortmund

[2]University of Warsaw (supported by WCMCS)

July 1, 2015

We consider a fundamental data structure question: how to represent a tree?

(Compacted) Trie

A trie is simply a tree with edges labeled by single characters. A compacted trie is created by replacing maximal chains of unary vertices with single edges labeled by (possibly long) words.

Navigation queries

Given a pattern p, we want to traverse the edges of a compacted trie to find the node corresponding to p. If there is no such node, we would like to compute its longest prefix for which the corresponding node does exist.

We consider a fundamental data structure question: how to represent a tree?

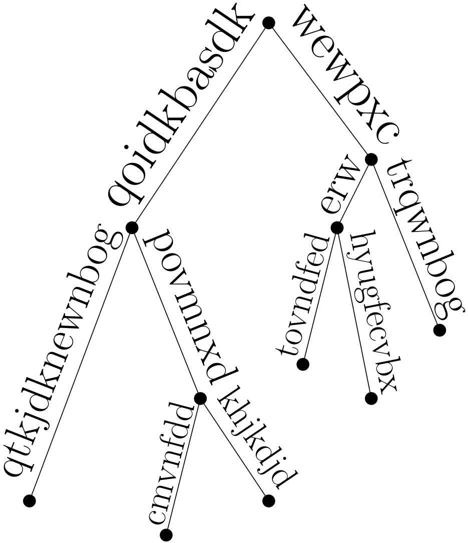
(Compacted) Trie

A trie is simply a tree with edges labeled by single characters. A compacted trie is created by replacing maximal chains of unary vertices with single edges labeled by (possibly long) words.

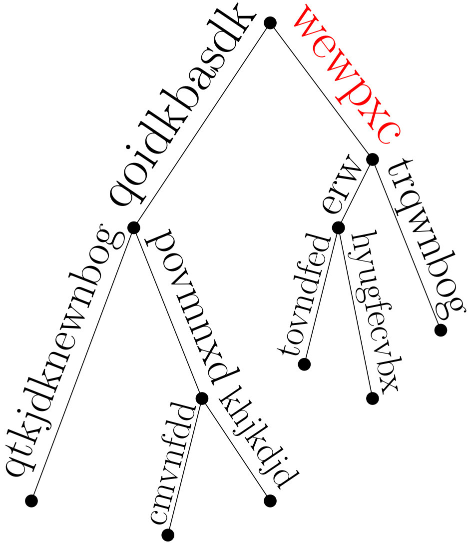
Navigation queries

Given a pattern p, we want to traverse the edges of a compacted trie to find the node corresponding to p. If there is no such node, we would like to compute its longest prefix for which the corresponding node does exist.

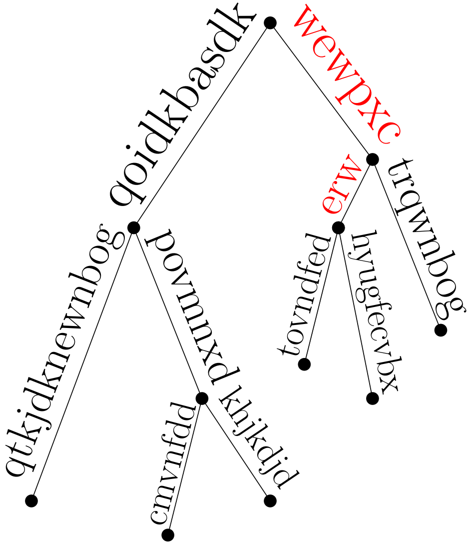
Consider $p = wewpxcwrehyzrt$ and the following compacted trie.



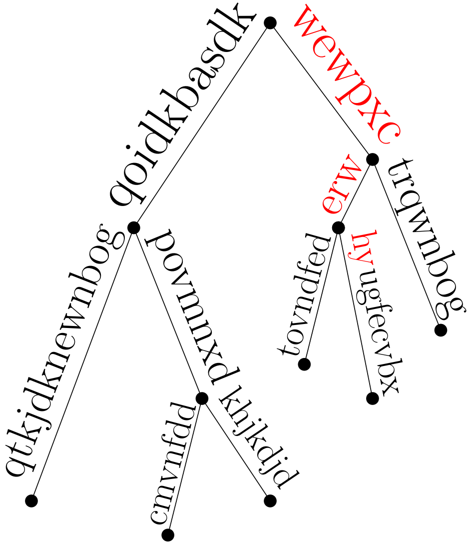
Consider $p =$ *wewpxcwrehyzrt* and the following compacted trie.



Consider $p = \textcolor{red}{wewpxcwre}hyzrt$ and the following compacted trie.

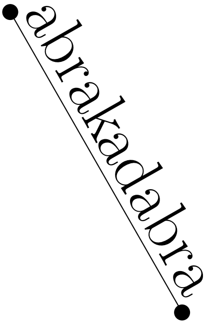


Consider $p = \textcolor{red}{wewpxcwrehy}zrt$ and the following compacted trie.



Splitting an edge

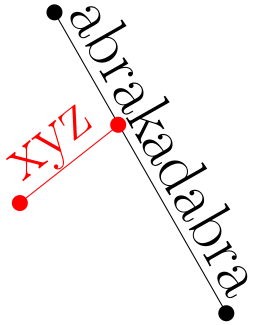
Given an edge, we want to split it into two parts by (possibly) creating a node, and adding a new edge outgoing from this middle node.



Notice that this covers adding a new edge outgoing from an existing node.

Splitting an edge

Given an edge, we want to split it into two parts by (possibly) creating a node, and adding a new edge outgoing from this middle node.



Notice that this covers adding a new edge outgoing from an existing node.

Static case

Given a compacted trie, can we **quickly** construct a **small** structure which allows us to execute navigation queries **efficiently**?

Dynamic case

Can we maintain a compacted trie so that:

1. the resulting structure is **small**,
2. we can execute navigation queries **efficiently**,
3. we can split any edge **efficiently**?

Parameters: the number of nodes in the compacted trie n, the size of the alphabet σ, and the length of the pattern m.

Static case

Given a compacted trie, can we **quickly** construct a **small** structure which allows us to execute navigation queries **efficiently**?

Dynamic case

Can we maintain a compacted trie so that:

1. the resulting structure is **small**,
2. we can execute navigation queries **efficiently**,
3. we can split any edge **efficiently**?

Parameters: the number of nodes in the compacted trie n, the size of the alphabet σ, and the length of the pattern m.

Static case

Given a compacted trie, can we **quickly** construct a **small** structure which allows us to execute navigation queries **efficiently**?

Dynamic case

Can we maintain a compacted trie so that:

1. the resulting structure is **small**,
2. we can execute navigation queries **efficiently**,
3. we can split any edge **efficiently**?

Parameters: the number of nodes in the compacted trie n, the size of the alphabet σ, and the length of the pattern m.

Hashing

For each node store a hash table mapping characters to the corresponding outgoing edges.

Randomized!

Table

Or, for each node store a table of size σ mapping characters to the corresponding outgoing edges.

Space usage is $n\sigma$!

BST

Or, for each node store a binary search tree mapping characters to the corresponding outgoing edges.

Navigation query takes $\mathcal{O}(m \log \sigma)$ time!

Hashing

For each node store a hash table mapping characters to the corresponding outgoing edges.

Randomized!

Table

Or, for each node store a table of size σ mapping characters to the corresponding outgoing edges.

Space usage is $n\sigma$!

BST

Or, for each node store a binary search tree mapping characters to the corresponding outgoing edges.

Navigation query takes $\mathcal{O}(m \log \sigma)$ time!

Hashing

For each node store a hash table mapping characters to the corresponding outgoing edges.

Randomized!

Table

Or, for each node store a table of size σ mapping characters to the corresponding outgoing edges.

Space usage is $n\sigma$!

BST

Or, for each node store a binary search tree mapping characters to the corresponding outgoing edges.

Navigation query takes $\mathcal{O}(m \log \sigma)$ time!

Hashing

For each node store a hash table mapping characters to the corresponding outgoing edges.

Randomized!

Table

Or, for each node store a table of size σ mapping characters to the corresponding outgoing edges.

Space usage is $n\sigma$!

BST

Or, for each node store a binary search tree mapping characters to the corresponding outgoing edges.

Navigation query takes $\mathcal{O}(m \log \sigma)$ time!

Hashing

For each node store a hash table mapping characters to the corresponding outgoing edges.

Randomized!

Table

Or, for each node store a table of size σ mapping characters to the corresponding outgoing edges.

Space usage is $n\sigma$!

BST

Or, for each node store a binary search tree mapping characters to the corresponding outgoing edges.

Navigation query takes $\mathcal{O}(m \log \sigma)$ time!

Hashing

For each node store a hash table mapping characters to the corresponding outgoing edges.

Randomized!

Table

Or, for each node store a table of size σ mapping characters to the corresponding outgoing edges.

Space usage is $n\sigma$!

BST

Or, for each node store a binary search tree mapping characters to the corresponding outgoing edges.

Navigation query takes $\mathcal{O}(m \log \sigma)$ time!

Rules of the game:

1. the solution must be deterministic,
2. the space usage must be linear in n, irrespectively of σ,
3. bound on the update time must be worst-case.

Then it seems that navigation queries must necessarily take $\mathcal{O}(mf(\sigma))$ time, for some function of σ, for instance $f(\sigma) = \log \sigma$, or something better if we use a more sophisticated predecessor structure. Surprisingly, this is not true.

Suffix trays of Cole, Kopelowitz, and Lewenstein ICALP'06

There exists a deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \log \sigma)$ time, which can be construct in linear time.

Rules of the game:

1. the solution must be deterministic,
2. the space usage must be linear in n, irrespectively of σ,
3. bound on the update time must be worst-case.

Then it seems that navigation queries must necessarily take $\mathcal{O}(mf(\sigma))$ time, for some function of σ, for instance $f(\sigma) = \log \sigma$, or something better if we use a more sophisticated predecessor structure. Surprisingly, this is not true.

Suffix trays of Cole, Kopelowitz, and Lewenstein ICALP'06

There exists a deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \log \sigma)$ time, which can be construct in linear time.

Rules of the game:

1. the solution must be deterministic,
2. the space usage must be linear in n, irrespectively of σ,
3. bound on the update time must be worst-case.

Then it seems that navigation queries must necessarily take $\mathcal{O}(mf(\sigma))$ time, for some function of σ, for instance $f(\sigma) = \log \sigma$, or something better if we use a more sophisticated predecessor structure. Surprisingly, this is not true.

Suffix trays of Cole, Kopelowitz, and Lewenstein ICALP'06

There exists a deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \log \sigma)$ time, which can be construct in linear time.

Rules of the game:

1. the solution must be deterministic,
2. the space usage must be linear in n, irrespectively of σ,
3. bound on the update time must be worst-case.

Then it seems that navigation queries must necessarily take $\mathcal{O}(mf(\sigma))$ time, for some function of σ, for instance $f(\sigma) = \log \sigma$, or something better if we use a more sophisticated predecessor structure. Surprisingly, this is not true.

Suffix trays of Cole, Kopelowitz, and Lewenstein ICALP'06

There exists a deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \log \sigma)$ time, which can be construct in linear time.

Rules of the game:

1. the solution must be deterministic,
2. the space usage must be linear in n, irrespectively of σ,
3. bound on the update time must be worst-case.

Then it seems that navigation queries must necessarily take $\mathcal{O}(mf(\sigma))$ time, for some function of σ, for instance $f(\sigma) = \log \sigma$, or something better if we use a more sophisticated predecessor structure.
Surprisingly, this is not true.

Suffix trays of Cole, Kopelowitz, and Lewenstein ICALP'06

There exists a deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \log \sigma)$ time, which can be construct in linear time.

Rules of the game:

1. the solution must be deterministic,
2. the space usage must be linear in n, irrespectively of σ,
3. bound on the update time must be worst-case.

Then it seems that navigation queries must necessarily take $\mathcal{O}(mf(\sigma))$ time, for some function of σ, for instance $f(\sigma) = \log \sigma$, or something better if we use a more sophisticated predecessor structure.

Surprisingly, this is not true.

Suffix trays of Cole, Kopelowitz, and Lewenstein ICALP'06

There exists a deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \log \sigma)$ time, which can be construct in linear time.

What about the updates?

Suffix trists of Cole, Kopelowitz, and Lewenstein ICALP'06

There exists a deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \log \sigma)$ time and splitting edges in $\mathcal{O}(\log \sigma)$.

Application to text indexing

Consider a suffix tree of a text. After prepending a letter, one edge should be split. It is easy to locate it in amortized $\mathcal{O}(1)$ time, but getting a sublinear worst-case bound is not trivial!

What about the updates?

Suffix trists of Cole, Kopelowitz, and Lewenstein ICALP'06

There exists a deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \log \sigma)$ time and splitting edges in $\mathcal{O}(\log \sigma)$.

Application to text indexing

Consider a suffix tree of a text. After prepending a letter, one edge should be split. It is easy to locate it in amortized $\mathcal{O}(1)$ time, but getting a sublinear worst-case bound is not trivial!

What about the updates?

Suffix trists of Cole, Kopelowitz, and Lewenstein ICALP'06

There exists a deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \log \sigma)$ time and splitting edges in $\mathcal{O}(\log \sigma)$.

Application to text indexing

Consider a suffix tree of a text. After prepending a letter, one edge should be split. It is easy to locate it in amortized $\mathcal{O}(1)$ time, but getting a sublinear worst-case bound is not trivial!

Suffix tree oracle of Amir, Kopelowitz, Lewenstein, and Lewenstein SPIRE'05

There exists a suffix tree oracle which locates the edge in $\mathcal{O}(\log n)$ time.

Suffix tree oracle of Breslauer and Italiano SPIRE'11

If $\sigma = \mathcal{O}(1)$, there exists a suffix tree oracle which locates the edge in $\mathcal{O}(\log \log n)$ time.

Suffix tree oracle of Amir, Kopelowitz, Lewenstein, and Lewenstein SPIRE'05

There exists a suffix tree oracle which locates the edge in $\mathcal{O}(\log n)$ time.

Suffix tree oracle of Breslauer and Italiano SPIRE'11

If $\sigma = \mathcal{O}(1)$, there exists a suffix tree oracle which locates the edge in $\mathcal{O}(\log \log n)$ time.

In the Word RAM model, are these $\mathcal{O}(m + \log \sigma)$ and $\mathcal{O}(\log \sigma)$ bounds the best possible?

Andersson and Thorup SODA'01

There exists a deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \sqrt{\frac{\log n}{\log \log n}})$ time and splitting edges in $\mathcal{O}(\sqrt{\frac{\log n}{\log \log n}})$.

Are these bounds are the best possible?

Yes if σ is unbounded in terms of n, and navigation queries actually give us the predecessor of the string.

In the Word RAM model, are these $\mathcal{O}(m + \log \sigma)$ and $\mathcal{O}(\log \sigma)$ bounds the best possible?

Andersson and Thorup SODA'01

There exists a deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \sqrt{\frac{\log n}{\log \log n}})$ time and splitting edges in $\mathcal{O}(\sqrt{\frac{\log n}{\log \log n}})$.

Are these bounds are the best possible?

Yes if σ is unbounded in terms of n, and navigation queries actually give us the predecessor of the string.

In the Word RAM model, are these $\mathcal{O}(m + \log \sigma)$ and $\mathcal{O}(\log \sigma)$ bounds the best possible?

Andersson and Thorup SODA'01

There exists a deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \sqrt{\frac{\log n}{\log \log n}})$ time and splitting edges in $\mathcal{O}(\sqrt{\frac{\log n}{\log \log n}})$.

Are these bounds are the best possible?

Yes if σ is unbounded in terms of n, and navigation queries actually give us the predecessor of the string.

But what if σ is non-constant, yet (significantly) smaller than n?

This paper

There exists a static deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \log\log\sigma)$ time, which can be constructed in linear time.

This paper

There exists a deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \frac{\log^2\log\sigma}{\log\log\log\sigma})$ time and splitting edges in $\mathcal{O}(\frac{\log^2\log\sigma}{\log\log\log\sigma})$.

Full version of the paper

A better suffix tree oracle to locate the edge in $\mathcal{O}(\log\log n + \frac{\log^2\log\sigma}{\log\log\log\sigma})$ time.

But what if σ is non-constant, yet (significantly) smaller than n?

This paper

There exists a static deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \log\log\sigma)$ time, which can be constructed in linear time.

This paper

There exists a deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \frac{\log^2\log\sigma}{\log\log\log\sigma})$ time and splitting edges in $\mathcal{O}(\frac{\log^2\log\sigma}{\log\log\log\sigma})$.

Full version of the paper

A better suffix tree oracle to locate the edge in $\mathcal{O}(\log\log n + \frac{\log^2\log\sigma}{\log\log\log\sigma})$ time.

But what if σ is non-constant, yet (significantly) smaller than n?

This paper

There exists a static deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \log\log\sigma)$ time, which can be constructed in linear time.

This paper

There exists a deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \frac{\log^2\log\sigma}{\log\log\log\sigma})$ time and splitting edges in $\mathcal{O}(\frac{\log^2\log\sigma}{\log\log\log\sigma})$.

Full version of the paper

A better suffix tree oracle to locate the edge in $\mathcal{O}(\log\log n + \frac{\log^2\log\sigma}{\log\log\log\sigma})$ time.

But what if σ is non-constant, yet (significantly) smaller than n?

This paper

There exists a static deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \log\log\sigma)$ time, which can be constructed in linear time.

This paper

There exists a deterministic linear-size structure supporting navigation in $\mathcal{O}(m + \frac{\log^2\log\sigma}{\log\log\log\sigma})$ time and splitting edges in $\mathcal{O}(\frac{\log^2\log\sigma}{\log\log\log\sigma})$.

Full version of the paper

A better suffix tree oracle to locate the edge in $\mathcal{O}(\log\log n + \frac{\log^2\log\sigma}{\log\log\log\sigma})$ time.

To construct a static deterministic linear-size structure, we could simply to try to find a perfect hashing function storing pairs ($node, character$).

Ružić ICALP'08

A static linear-size constant-access dictionary on a set of k keys can be deterministically constructed in time $\mathcal{O}(k \log^2 \log k)$.

Hence we immediately get a static deterministic structure which can be construct in close-to-linear time. Can we do better?

To construct a static deterministic linear-size structure, we could simply to try to find a perfect hashing function storing pairs (*node*, *character*).

Ružić ICALP'08

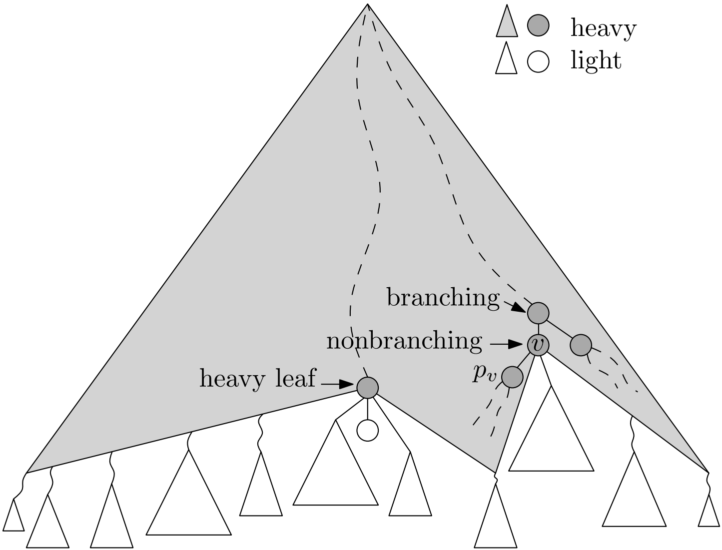
A static linear-size constant-access dictionary on a set of k keys can be deterministically constructed in time $\mathcal{O}(k \log^2 \log k)$.

Hence we immediately get a static deterministic structure which can be construct in close-to-linear time. Can we do better?

We store the edges outgoing from v in a few different ways depending on the size of the subtree rooted at v.

Heavy nodes

A node is heavy if its subtree contains at least $s = \Theta(\log^2 \log \sigma)$ leaves, and otherwise light. Furthermore, a heavy node is branching if it has more than one heavy child.


heavy
light
branching
nonbranching
v
p_v
heavy leaf

We classify edges into three types, and deal with each type separately:

1. from (any) branching node to a light node,
2. from a nonbranching heavy node to (any) heavy node,
3. from a branching heavy node to (any) heavy node.

We classify edges into three types, and deal with each type separately:

1. from (any) branching node to a light node,
2. from a nonbranching heavy node to (any) heavy node,
3. from a branching heavy node to (any) heavy node.

At most one such edge per node, can be stored separately.

We classify edges into three types, and deal with each type separately:

1. from (any) branching node to a light node,
2. from a nonbranching heavy node to (any) heavy node,
3. from a branching heavy node to (any) heavy node.

The total number of such edges is just $\frac{n}{s}$, hence we can afford the super-linear construction time. More precisely, we compute the perfect hashing function for each such node separately in

$$\mathcal{O}(k \log^2 \log k) = \mathcal{O}(k \log^2 \log \sigma) = \mathcal{O}(ks)$$

time, which takes $\mathcal{O}(\frac{n}{s}s) = \mathcal{O}(n)$ time in total.

We classify edges into three types, and deal with each type separately:

1. from (any) branching node to a light node,
2. from a nonbranching heavy node to (any) heavy node,
3. from a branching heavy node to (any) heavy node.

We store all such edges in a predecessor structure. By combining perfect hashing result and Willard's x-fast trees, there exists a linear-size predecessor structure with $\mathcal{O}(\log\log\sigma)$ query time, which can be constructed in linear time.

Observe that any navigation query traverses an edge of type (1) at most once, hence we pay $\mathcal{O}(\log\log\sigma)$ just once (so far). But what happens when we reach a light node?

Each light node contains at most s leaves. We can execute a binary search over those leaves using the suffix array trick, namely in each step we achieve at least one of the following:

1. halve the current interval,
2. consume one character from the pattern.

Hence in $\mathcal{O}(m + \log s)$ time we can locate the predecessor of the pattern among all leaves, and the search actually computes the longest prefix of the pattern which is a prefix of a string corresponding to some leaf.

Observe that any navigation query traverses an edge of type (1) at most once, hence we pay $\mathcal{O}(\log\log\sigma)$ just once (so far). But what happens when we reach a light node?

Each light node contains at most s leaves. We can execute a binary search over those leaves using the suffix array trick, namely in each step we achieve at least one of the following:

1. halve the current interval,
2. consume one character from the pattern.

Hence in $\mathcal{O}(m + \log s)$ time we can locate the predecessor of the pattern among all leaves, and the search actually computes the longest prefix of the pattern which is a prefix of a string corresponding to some leaf.

Observe that any navigation query traverses an edge of type (1) at most once, hence we pay $\mathcal{O}(\log\log\sigma)$ just once (so far). But what happens when we reach a light node?

Each light node contains at most s leaves. We can execute a binary search over those leaves using the suffix array trick, namely in each step we achieve at least one of the following:

1. halve the current interval,
2. consume one character from the pattern.

Hence in $\mathcal{O}(m + \log s)$ time we can locate the predecessor of the pattern among all leaves, and the search actually computes the longest prefix of the pattern which is a prefix of a string corresponding to some leaf.

The total time complexity for a query is

$$\mathcal{O}(m + \log\log\sigma + \log s) = \mathcal{O}(m + \log\log\sigma)$$

and the total construction time is linear.

Now let us consider the dynamic case.

Reduction

The general case can be reduced to maintaining a collection of trees of size $\mathcal{O}(\sigma)$ each and linear total size, so that any update/query can be efficiently translated into an update/query into at most one smaller tree.

From now on we assume that $n = \mathcal{O}(\sigma)$. Instead of the simple two-level scheme we need to partition the nodes into more groups.

Levels of nodes

Let $f(\ell) = 2^{(\frac{3}{2})^{\ell}}$. We say that a node v is of level ℓ when the number of leaves in its subtree belongs to $[f(\ell), 2f(\ell+1)]$. We will maintain an invariant that a level of v doesn't exceed the level of its parent.

Now let us consider the dynamic case.

Reduction

The general case can be reduced to maintaining a collection of trees of size $\mathcal{O}(\sigma)$ each and linear total size, so that any update/query can be efficiently translated into an update/query into at most one smaller tree.

From now on we assume that $n = \mathcal{O}(\sigma)$. Instead of the simple two-level scheme we need to partition the nodes into more groups.

Levels of nodes

Let $f(\ell) = 2^{(\frac{3}{2})^{\ell}}$. We say that a node v is of level ℓ when the number of leaves in its subtree belongs to $[f(\ell), 2f(\ell+1)]$. We will maintain an invariant that a level of v doesn't exceed the level of its parent.

Now let us consider the dynamic case.

Reduction

The general case can be reduced to maintaining a collection of trees of size $\mathcal{O}(\sigma)$ each and linear total size, so that any update/query can be efficiently translated into an update/query into at most one smaller tree.

From now on we assume that $n = \mathcal{O}(\sigma)$. Instead of the simple two-level scheme we need to partition the nodes into more groups.

Levels of nodes

Let $f(\ell) = 2^{(\frac{3}{2})^{\ell}}$. We say that a node v is of level ℓ when the number of leaves in its subtree belongs to $[f(\ell), 2f(\ell+1)]$. We will maintain an invariant that a level of v doesn't exceed the level of its parent.

Now let us consider the dynamic case.

Reduction

The general case can be reduced to maintaining a collection of trees of size $\mathcal{O}(\sigma)$ each and linear total size, so that any update/query can be efficiently translated into an update/query into at most one smaller tree.

From now on we assume that $n = \mathcal{O}(\sigma)$. Instead of the simple two-level scheme we need to partition the nodes into more groups.

Levels of nodes

Let $f(\ell) = 2^{(\frac{3}{2})^{\ell}}$. We say that a node v is of level ℓ when the number of leaves in its subtree belongs to $[f(\ell), 2f(\ell+1)]$. We will maintain an invariant that a level of v doesn't exceed the level of its parent.

Now, we classify the edges into two types:

1. from a node to a node of the same level,
2. from a node to a node of a smaller level,

Now, we classify the edges into two types:

1. from a node to a node of the same level,
2. from a node to a node of a smaller level,

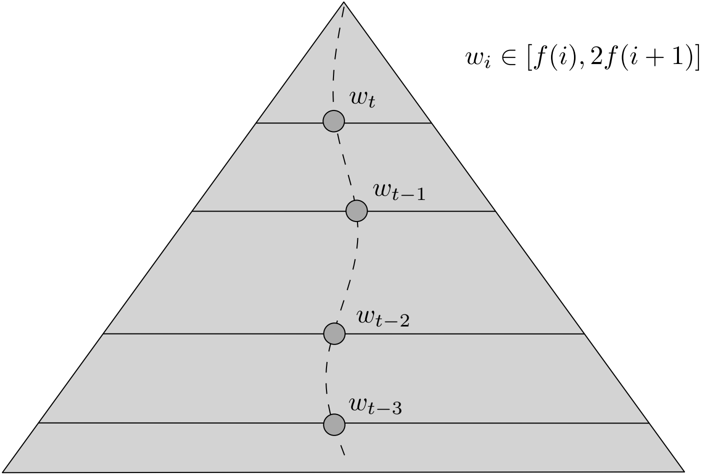
Those edges are stored in a static dictionary with constant access time. We already know that such dictionary can be construct in close-to-linear time, which is enough because of the way we defined the levels. More precisely, it cannot happen too often that a level of a node increases.

Now, we classify the edges into two types:

1. from a node to a node of the same level,
2. from a node to a node of a smaller level,

Those edges are stored in a dynamic dictionary structure. For this we develop a weighted variant of the exponential search trees of Andersson and Thorup.

Even without the modification, the query complexity is $\mathcal{O}(m + \frac{\log^3 \log \sigma}{\log \log \log \sigma})$. This is because there are at most $t = \Theta(\log \log \sigma)$ edges of type (2) on any path descending from the root.



Faster!

The subsequent accesses to the dynamic dictionary structures are not completely independent.

Wexponential search trees

There exists a linear-size dynamic structure storing a collection of n weighted elements from $[1, U]$ with the following bounds:

1. predecessor search takes $\mathcal{O}(\log \frac{\log W}{\log w} \frac{\log\log U}{\log\log\log U})$, where W is the current total weight, and w is the weight of the predecessor,
2. inserting a new element of weight 1 takes $\mathcal{O}(\log\log W)$,
3. increasing a weight of an element of weight w by 1 takes $\mathcal{O}(\log \frac{\log W}{\log w})$.

Telescoping

Now if we use this structure instead of the standard exponential search trees, the total complexity of all queries at nodes where we decrease the current level becomes:

$$\sum_{i=t-1}^{0} \log \frac{\log w_{i+1}}{\log w_i} \frac{\log\log U}{\log\log\log U} = \frac{\log\log U}{\log\log\log U} \log\log w_t$$

$$= \frac{\log\log U}{\log\log\log U} \log\log U = \frac{\log^2 \log U}{\log\log\log U}$$

(ignoring the details necessary to show how to update the structures...)

Telescoping

Now if we use this structure instead of the standard exponential search trees, the total complexity of all queries at nodes where we decrease the current level becomes:

$$\sum_{i=t-1}^{0} \log \frac{\log w_{i+1}}{\log w_i} \frac{\log\log U}{\log\log\log U} = \frac{\log\log U}{\log\log\log U} \log\log w_t$$

$$= \frac{\log\log U}{\log\log\log U} \log\log U = \frac{\log^2 \log U}{\log\log\log U}$$

(ignoring the details necessary to show how to update the structures...)

Telescoping

Now if we use this structure instead of the standard exponential search trees, the total complexity of all queries at nodes where we decrease the current level becomes:

$$\sum_{i=t-1}^{0} \log \frac{\log w_{i+1}}{\log w_i} \frac{\log \log U}{\log \log \log U} = \frac{\log \log U}{\log \log \log U} \log \log w_t$$

$$= \frac{\log \log U}{\log \log \log U} \log \log U = \frac{\log^2 \log U}{\log \log \log U}$$

(ignoring the details necessary to show how to update the structures...)

Telescoping

Now if we use this structure instead of the standard exponential search trees, the total complexity of all queries at nodes where we decrease the current level becomes:

$$\sum_{i=t-1}^{0} \log \frac{\log w_{i+1}}{\log w_i} \frac{\log\log U}{\log\log\log U} = \frac{\log\log U}{\log\log\log U} \log\log w_t$$

$$= \frac{\log\log U}{\log\log\log U} \log\log U = \frac{\log^2 \log U}{\log\log\log U}$$

(ignoring the details necessary to show how to update the structures...)

Wexponential search trees

Imagine that each element of weight w is a fragment of such length, and draw all of them on a $[1, W]$ segment.

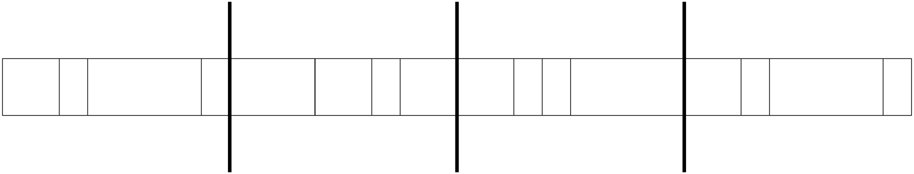
Then choose a set of roughly $\sqrt{W}$ evenly spaced splitters. Store them in a static predecessor structure, and recursively build a smaller wexponential search tree for each of the resulting roughly $\sqrt{W}$ subsets.

Beame and Fich STOC'90

A static predecessor search structure with $\mathcal{O}(\frac{\log\log\sigma}{\log\log\log\sigma})$ query time can be constructed in $\mathcal{O}(k^{1+\epsilon})$ time and space, where k is the number of elements.

Wexponential search trees

Imagine that each element of weight w is a fragment of such length, and draw all of them on a $[1, W]$ segment.



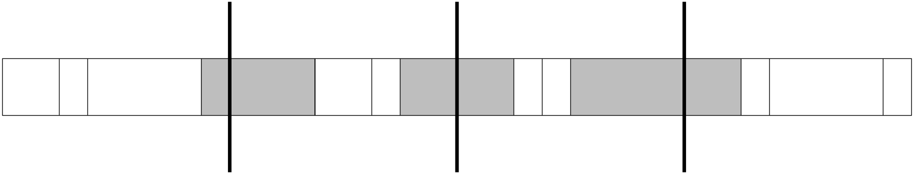
Then choose a set of roughly $\sqrt{W}$ evenly spaced splitters. Store them in a static predecessor structure, and recursively build a smaller wexponential search tree for each of the resulting roughly $\sqrt{W}$ subsets.

Beame and Fich STOC'90

A static predecessor search structure with $\mathcal{O}(\frac{\log\log\sigma}{\log\log\log\sigma})$ query time can be constructed in $\mathcal{O}(k^{1+\epsilon})$ time and space, where k is the number of elements.

Wexponential search trees

Imagine that each element of weight w is a fragment of such length, and draw all of them on a $[1, W]$ segment.



Then choose a set of roughly $\sqrt{W}$ evenly spaced splitters. Store them in a static predecessor structure, and recursively build a smaller wexponential search tree for each of the resulting roughly $\sqrt{W}$ subsets.

Beame and Fich STOC'90

A static predecessor search structure with $\mathcal{O}(\frac{\log\log\sigma}{\log\log\log\sigma})$ query time can be constructed in $\mathcal{O}(k^{1+\epsilon})$ time and space, where k is the number of elements.

Wexponential search trees

Intuition:

1. the larger the weight, the sooner the element is stored in a static predecessor structure,
2. rebuilding a static predecessor structure is very costly, but happens only if there have been multiple insertions/increases.

Worst-case bounds

Very complicated in Andresson&Thorup paper. We follow the simpler idea of Bender, Cole and Raman.

Wexponential search trees

Intuition:

1. the larger the weight, the sooner the element is stored in a static predecessor structure,
2. rebuilding a static predecessor structure is very costly, but happens only if there have been multiple insertions/increases.

Worst-case bounds

Very complicated in Andresson&Thorup paper. We follow the simpler idea of Bender, Cole and Raman.

Wexponential search trees

Intuition:

1. the larger the weight, the sooner the element is stored in a static predecessor structure,
2. rebuilding a static predecessor structure is very costly, but happens only if there have been multiple insertions/increases.

Worst-case bounds

Very complicated in Andresson&Thorup paper. We follow the simpler idea of Bender, Cole and Raman.

Wexponential search trees

Intuition:

1. the larger the weight, the sooner the element is stored in a static predecessor structure,
2. rebuilding a static predecessor structure is very costly, but happens only if there have been multiple insertions/increases.

Worst-case bounds

Very complicated in Andresson&Thorup paper. We follow the simpler idea of Bender, Cole and Raman.

Questions?