Fusion trees

Paweł Gawrychowski

5 czerwca 2014

Fusion trees

So far all none of our structure was able to beat the naive binary search without assuming something about U. Now we will show that one can, in fact, unconditionally beat $\mathcal{O}(\log n)$.

Fusion trees of Fredman and Willard 1990

Static predecessor can be preprocessed in linear space to achieve $\mathcal{O}(\frac{\log n}{\log w})$ query time.

The essence of the above result is the following.

Predecessor in small sets

Any set of at most $w^{1/5}$ keys can be stored in linear space, so that a predecessor query takes $\mathcal{O}(1)$ time.

Then we simply construct a B-tree with degree $B = w^{1/5}$. This gives us predecessor search in $\mathcal{O}(\log_B n) = \mathcal{O}(\frac{\log n}{\log w})$ time and linear space.

Fusion trees

So far all none of our structure was able to beat the naive binary search without assuming something about U. Now we will show that one can, in fact, unconditionally beat $\mathcal{O}(\log n)$.

Fusion trees of Fredman and Willard 1990

Static predecessor can be preprocessed in linear space to achieve $\mathcal{O}(\frac{\log n}{\log w})$ query time.

The essence of the above result is the following.

Predecessor in small sets

Any set of at most $w^{1/5}$ keys can be stored in linear space, so that a predecessor query takes $\mathcal{O}(1)$ time.

Then we simply construct a B-tree with degree $B = w^{1/5}$. This gives us predecessor search in $\mathcal{O}(\log_B n) = \mathcal{O}(\frac{\log n}{\log w})$ time and linear space.

Fusion trees

So far all none of our structure was able to beat the naive binary search without assuming something about U. Now we will show that one can, in fact, unconditionally beat $\mathcal{O}(\log n)$.

Fusion trees of Fredman and Willard 1990

Static predecessor can be preprocessed in linear space to achieve $\mathcal{O}(\frac{\log n}{\log w})$ query time.

The essence of the above result is the following.

Predecessor in small sets

Any set of at most $w^{1/5}$ keys can be stored in linear space, so that a predecessor query takes $\mathcal{O}(1)$ time.

Then we simply construct a B-tree with degree $B = w^{1/5}$. This gives us predecessor search in $\mathcal{O}(\log_B n) = \mathcal{O}(\frac{\log n}{\log w})$ time and linear space.

Fusion trees

So far all none of our structure was able to beat the naive binary search without assuming something about U. Now we will show that one can, in fact, unconditionally beat $\mathcal{O}(\log n)$.

Fusion trees of Fredman and Willard 1990

Static predecessor can be preprocessed in linear space to achieve $\mathcal{O}(\frac{\log n}{\log w})$ query time.

The essence of the above result is the following.

Predecessor in small sets

Any set of at most $w^{1/5}$ keys can be stored in linear space, so that a predecessor query takes $\mathcal{O}(1)$ time.

Then we simply construct a B-tree with degree $B = w^{1/5}$. This gives us predecessor search in $\mathcal{O}(\log_B n) = \mathcal{O}(\frac{\log n}{\log w})$ time and linear space.

Fusion trees

So far all none of our structure was able to beat the naive binary search without assuming something about U. Now we will show that one can, in fact, unconditionally beat $\mathcal{O}(\log n)$.

Fusion trees of Fredman and Willard 1990

Static predecessor can be preprocessed in linear space to achieve $\mathcal{O}(\frac{\log n}{\log w})$ query time.

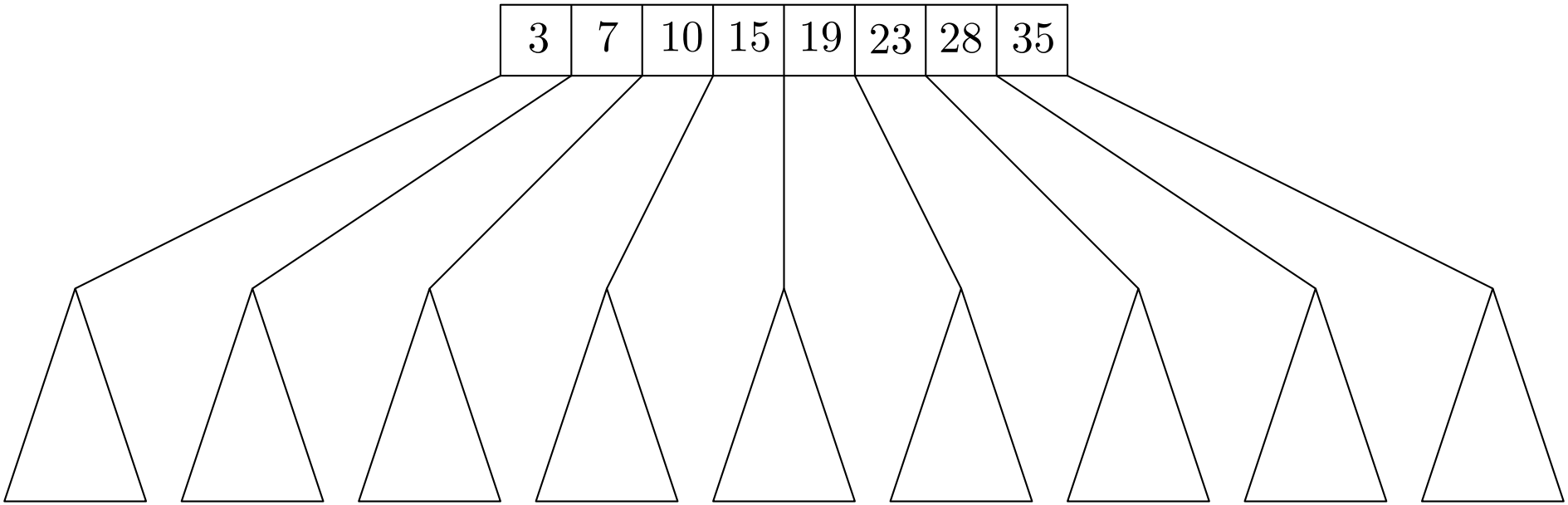
The essence of the above result is the following.

Predecessor in small sets

Any set of at most $w^{1/5}$ keys can be stored in linear space, so that a predecessor query takes $\mathcal{O}(1)$ time.

Then we simply construct a B-tree with degree $B = w^{1/5}$. This gives us predecessor search in $\mathcal{O}(\log_B n) = \mathcal{O}(\frac{\log n}{\log w})$ time and linear space.

B-tree?



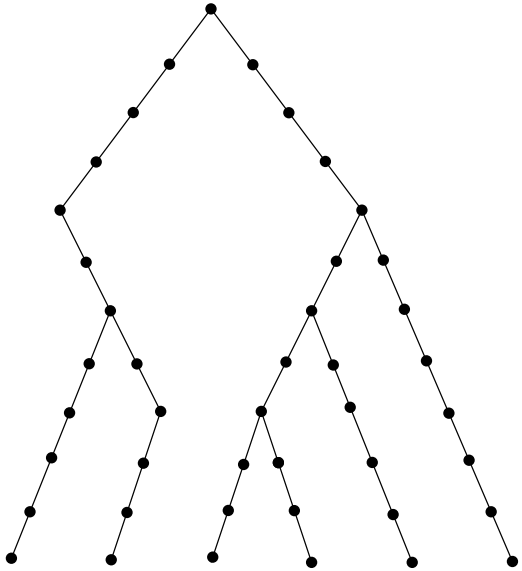
Every non-root node stores at least $B - 1$ and at most $2(B - 1)$ numbers. The root stores at most $2B - 1$ numbers.

Predecessor in small sets

Any set of at most $w^{1/5}$ keys can be stored in linear space, so that a predecessor query takes $\mathcal{O}(1)$ time.

Our word size is just w, so we cannot store all $w^{1/5}$ keys in a single word. But...

...create a binary trie containing all the keys.



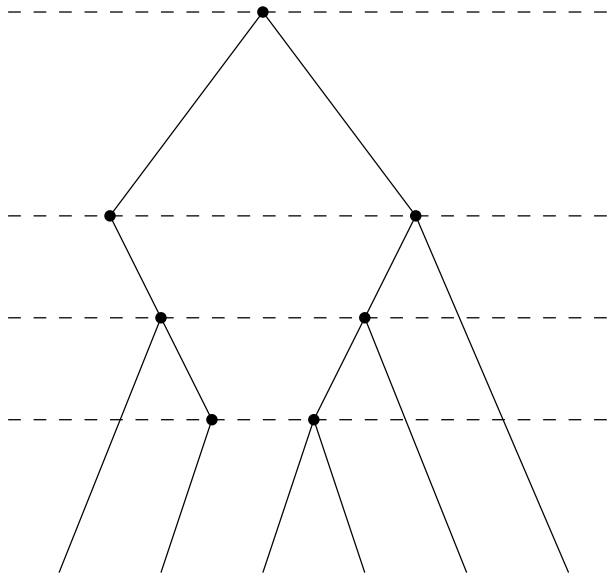
Then there are less than $w^{1/5}$ branching nodes there, hence only bits at some at most $w^{1/5}$ positions really matter.

Sketch

We will call these important positions $b_1, b_2, \ldots, b_r$. From every key we keep only bits at positions $b_1, b_2, \ldots, b_r$ packed one after another.

Every such sketch takes just $r \leq w^{1/5}$ bits, so we can easily put all of them into a single word.

...create a binary trie containing all the keys.



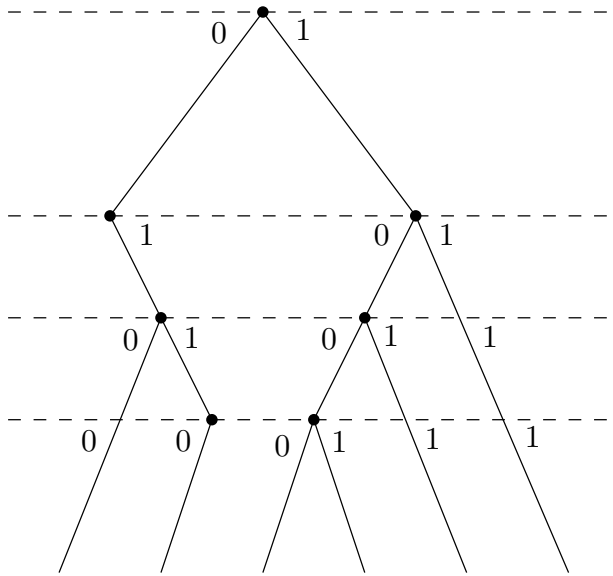
Then there are less than $w^{1/5}$ branching nodes there, hence only bits at some at most $w^{1/5}$ positions really matter.

Sketch

We will call these important positions $b_1, b_2, \ldots, b_r$. From every key we keep only bits at positions $b_1, b_2, \ldots, b_r$ packed one after another.

Every such sketch takes just $r \leq w^{1/5}$ bits, so we can easily put all of them into a single word.

...create a binary trie containing all the keys.



```
0....1.0.0...
0....1.1.0...
1....0.0.0...
1....0.0.1...
1....0.1.1...
1....1.1.1...
```

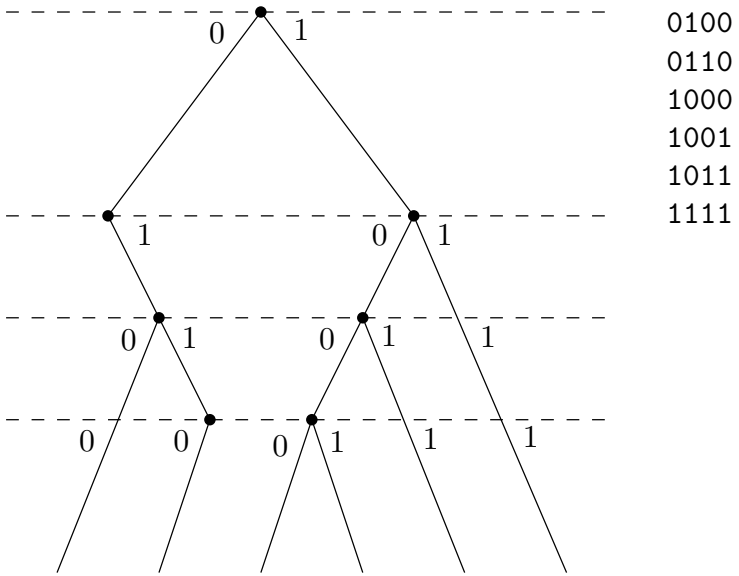
Then there are less than $w^{1/5}$ branching nodes there, hence only bits at some at most $w^{1/5}$ positions really matter.

Sketch

We will call these important positions $b_1, b_2, \ldots, b_r$. From every key we keep only bits at positions $b_1, b_2, \ldots, b_r$ packed one after another.

Every such sketch takes just $r \leq w^{1/5}$ bits, so we can easily put all of them into a single word.

...create a binary trie containing all the keys.



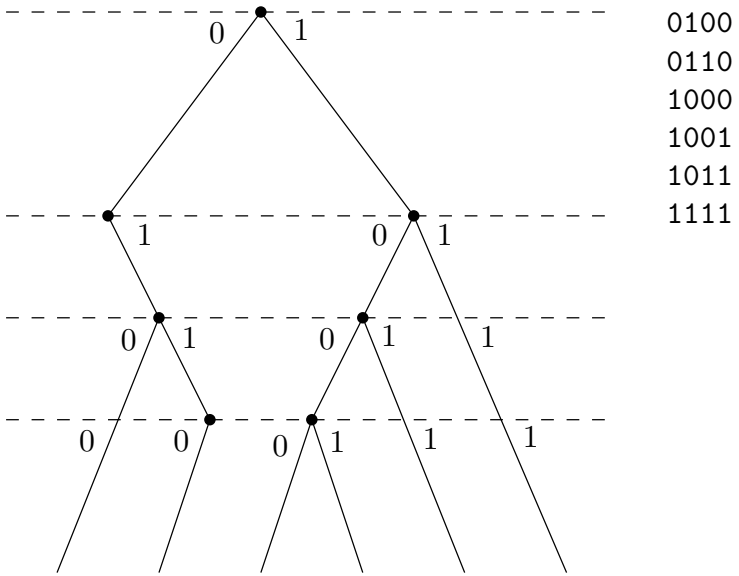
Then there are less than $w^{1/5}$ branching nodes there, hence only bits at some at most $w^{1/5}$ positions really matter.

Sketch

We will call these important positions $b_1, b_2, \ldots, b_r$. From every key we keep only bits at positions $b_1, b_2, \ldots, b_r$ packed one after another.

Every such sketch takes just $r \leq w^{1/5}$ bits, so we can easily put all of them into a single word.

...create a binary trie containing all the keys.



Then there are less than $w^{1/5}$ branching nodes there, hence only bits at some at most $w^{1/5}$ positions really matter.

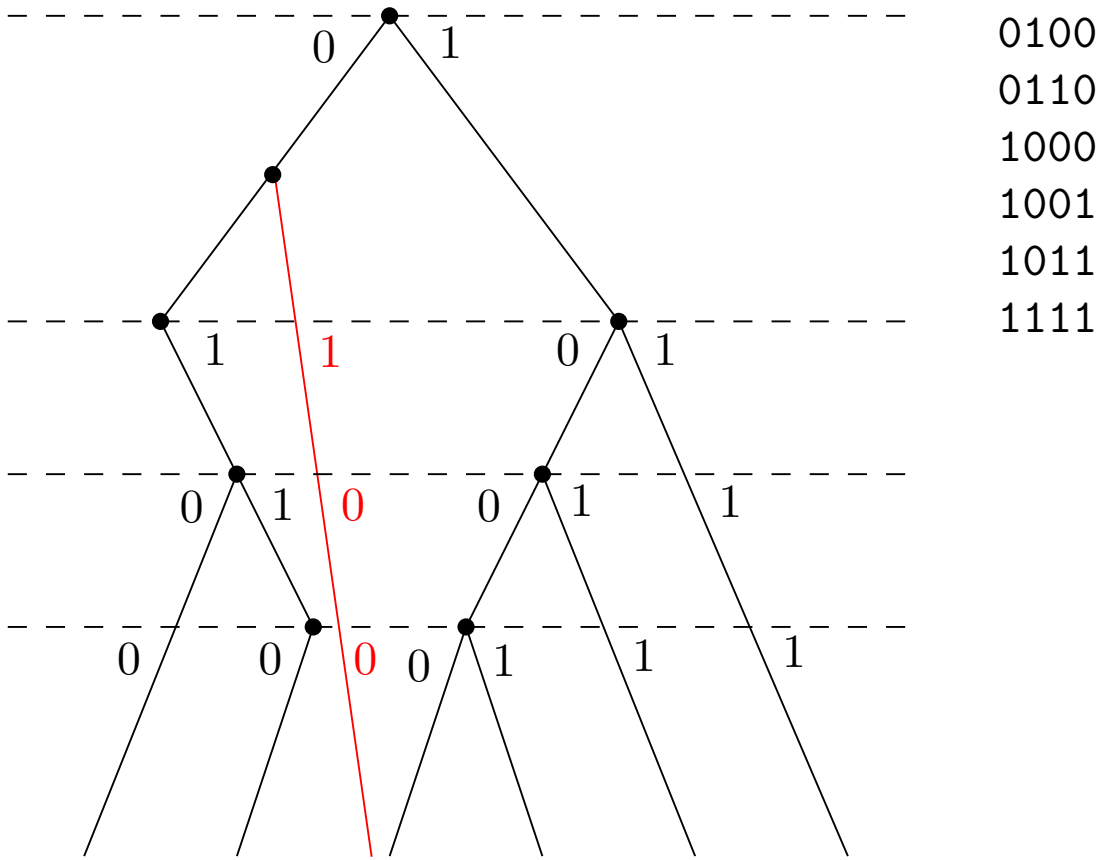
Sketch

We will call these important positions $b_1, b_2, \ldots, b_r$. From every key we keep only bits at positions $b_1, b_2, \ldots, b_r$ packed one after another.

Every such sketch takes just $r \leq w^{1/5}$ bits, so we can easily put all of them into a single word.

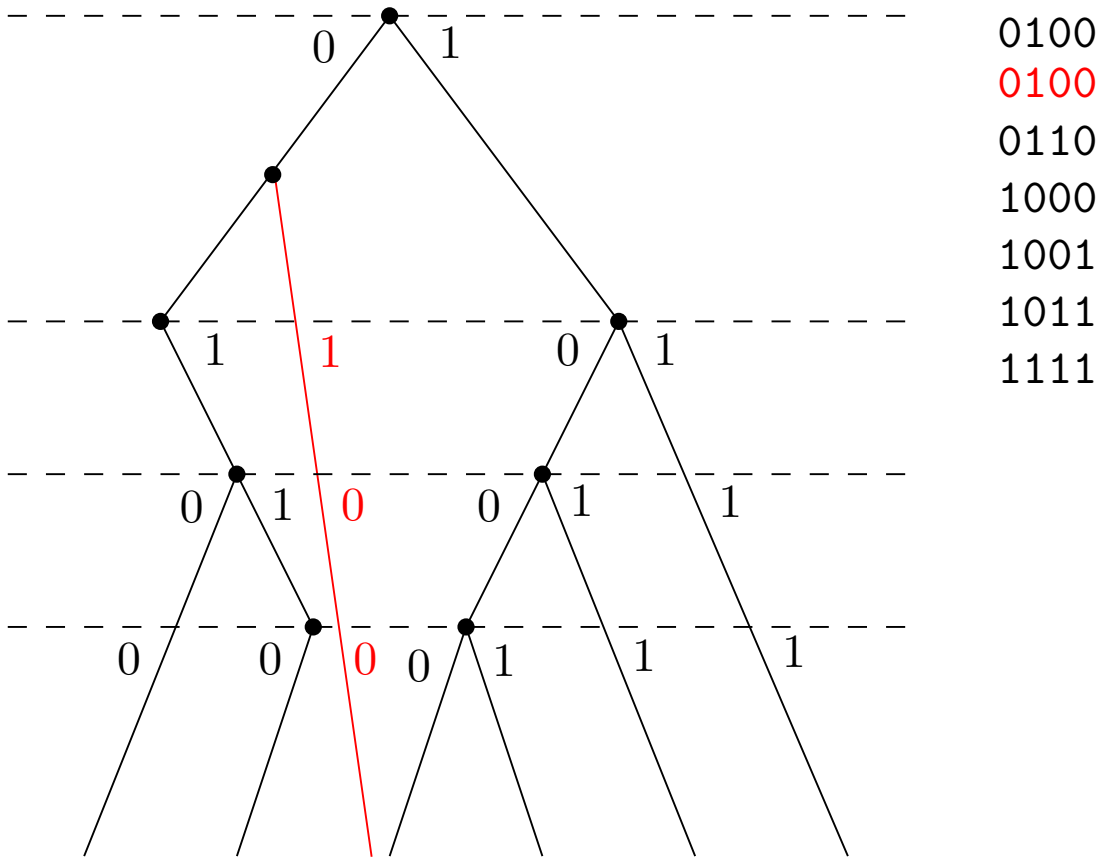
Predecessor in small sets

Observe that if the keys are sorted, so are their sketches. Moreover, if we want to find the predecessor of x, we can first compute its sketch and find its predecessor in the collection of all sketches. This almost gives us the predecessor of x.



Predecessor in small sets

Observe that if the keys are sorted, so are their sketches. Moreover, if we want to find the predecessor of x, we can first compute its sketch and find its predecessor in the collection of all sketches. This almost gives us the predecessor of x.



Predecessor in small sets

Almost?

Say that the sketch of x is between the sketches of a_j and a_{j+1}. The actual predecessor of x might be much larger than a_j, but:

1. The (true) longest common prefix of x and any number in the set is either the longest common prefix of x and a_j or a_{j+1}. So with one XOR and MSB we know the longest common prefix $x_1..x_\ell$!
2. If $x_{\ell+1} = 1$ we return the predecessor of $x_1..x_\ell 011..11$.
3. If $x_{\ell+1} = 0$ we return the successor of $x_1..x_\ell 100..00$.

In both cases, we again look at the sketches. This time the predecessor/success of the sketch will correspond to the true predecessor though, see the blackboard.

Predecessor in small sets

Almost?

Say that the sketch of x is between the sketches of a_j and a_{j+1}. The actual predecessor of x might be much larger than a_j, but:

1. The (true) longest common prefix of x and any number in the set is either the longest common prefix of x and a_j or a_{j+1}. So with one XOR and MSB we know the longest common prefix $x_1..x_\ell$!
2. If $x_{\ell+1} = 1$ we return the predecessor of $x_1..x_\ell 011..11$.
3. If $x_{\ell+1} = 0$ we return the successor of $x_1..x_\ell 100..00$.

In both cases, we again look at the sketches. This time the predecessor/success of the sketch will correspond to the true predecessor though, see the blackboard.

Predecessor in small sets

Almost?

Say that the sketch of x is between the sketches of a_j and a_{j+1}. The actual predecessor of x might be much larger than a_j, but:

1. The (true) longest common prefix of x and any number in the set is either the longest common prefix of x and a_j or a_{j+1}. So with one XOR and MSB we know the longest common prefix $x_1..x_\ell$!
2. If $x_{\ell+1} = 1$ we return the predecessor of $x_1..x_\ell 011..11$.
3. If $x_{\ell+1} = 0$ we return the successor of $x_1..x_\ell 100..00$.

In both cases, we again look at the sketches. This time the predecessor/success of the sketch will correspond to the true predecessor though, see the blackboard.

Predecessor in small sets

Almost?

Say that the sketch of x is between the sketches of a_j and a_{j+1}. The actual predecessor of x might be much larger than a_j, but:

1. The (true) longest common prefix of x and any number in the set is either the longest common prefix of x and a_j or a_{j+1}. So with one XOR and MSB we know the longest common prefix $x_1..x_\ell$!
2. If $x_{\ell+1} = 1$ we return the predecessor of $x_1..x_\ell 011..11$.
3. If $x_{\ell+1} = 0$ we return the successor of $x_1..x_\ell 100..00$.

In both cases, we again look at the sketches. This time the predecessor/success of the sketch will correspond to the true predecessor though, see the blackboard.

Predecessor in small sets

Almost?

Say that the sketch of x is between the sketches of a_j and a_{j+1}. The actual predecessor of x might be much larger than a_j, but:

1. The (true) longest common prefix of x and any number in the set is either the longest common prefix of x and a_j or a_{j+1}. So with one XOR and MSB we know the longest common prefix $x_1..x_\ell$!
2. If $x_{\ell+1} = 1$ we return the predecessor of $x_1..x_\ell 011..11$.
3. If $x_{\ell+1} = 0$ we return the successor of $x_1..x_\ell 100..00$.

In both cases, we again look at the sketches. This time the predecessor/success of the sketch will correspond to the true predecessor though, see the blackboard.

Most significant bit

We assumed that we can compute in $\mathcal{O}(1)$ time the position of the most significant one in a word. Can we? It might be in our instruction set, but it doesn't have to.

MSB(x)

Such operation can be simulated with a constant number of ANDSs, XORs, ORs, and multiplications.

This is nontrivial, but mostly because of the somehow tedious details: the idea is to use sketching twice, first to determine the answer divided by $\sqrt{w}$, and then the answer modulo $\sqrt{w}$. If this sounds interesting to you, ask me during the tutorial.

Most significant bit

We assumed that we can compute in $\mathcal{O}(1)$ time the position of the most significant one in a word. Can we? It might be in our instruction set, but it doesn't have to.

MSB(x)

Such operation can be simulated with a constant number of ANDSs, XORs, ORs, and multiplications.

This is nontrivial, but mostly because of the somehow tedious details: the idea is to use sketching twice, first to determine the answer divided by $\sqrt{w}$, and then the answer modulo $\sqrt{w}$. If this sounds interesting to you, ask me during the tutorial.

Most significant bit

We assumed that we can compute in $\mathcal{O}(1)$ time the position of the most significant one in a word. Can we? It might be in our instruction set, but it doesn't have to.

MSB(x)

Such operation can be simulated with a constant number of ANDSs, XORs, ORs, and multiplications.

This is nontrivial, but mostly because of the somehow tedious details: the idea is to use sketching twice, first to determine the answer divided by $\sqrt{w}$, and then the answer modulo $\sqrt{w}$. If this sounds interesting to you, ask me during the tutorial.

Searching in a collection of sketches

So now our situation is that we have a sorted collection of r bit numbers packed in a single word. How to find the predecessor of another r bit number in such collection efficiently?

Parallel searching for x

We assume that the numbers are separated by single bits, so we actually store $0a_1 0a_2 .. 0a_r$, and $a_1 < a_2 < \ldots < a_r$.

1. OR with $10^r 10^r .. 10^r$.
2. subtract $0x0x..0x$.
3. AND with $10^r 10^r .. 10^r$.
4. compute the MSB.

This counts how many a_i's are smaller than x, which gives us the position of the predecessor.

Searching in a collection of sketches

So now our situation is that we have a sorted collection of r bit numbers packed in a single word. How to find the predecessor of another r bit number in such collection efficiently?

Parallel searching for x

We assume that the numbers are separated by single bits, so we actually store $0a_1 0a_2 .. 0a_r$, and $a_1 < a_2 < \ldots < a_r$.

1. OR with $10^r 10^r .. 10^r$.
2. subtract $0x0x..0x$.
3. AND with $10^r 10^r .. 10^r$.
4. compute the MSB.

This counts how many a_i's are smaller than x, which gives us the position of the predecessor.

Searching in a collection of sketches

So now our situation is that we have a sorted collection of r bit numbers packed in a single word. How to find the predecessor of another r bit number in such collection efficiently?

Parallel searching for x

We assume that the numbers are separated by single bits, so we actually store $0a_1 0a_2 .. 0a_r$, and $a_1 < a_2 < \ldots < a_r$.

1. OR with $10^r 10^r .. 10^r$.
2. subtract $0x0x..0x$.
3. AND with $10^r 10^r .. 10^r$.
4. compute the MSB.

This counts how many a_i's are smaller than x, which gives us the position of the predecessor.

Searching in a collection of sketches

So now our situation is that we have a sorted collection of r bit numbers packed in a single word. How to find the predecessor of another r bit number in such collection efficiently?

Parallel searching for x

We assume that the numbers are separated by single bits, so we actually store $0a_1 0a_2 .. 0a_r$, and $a_1 < a_2 < \ldots < a_r$.

1. OR with $10^r 10^r .. 10^r$.
2. subtract $0x0x..0x$.
3. AND with $10^r 10^r .. 10^r$.
4. compute the MSB.

This counts how many a_i's are smaller than x, which gives us the position of the predecessor.

Searching in a collection of sketches

So now our situation is that we have a sorted collection of r bit numbers packed in a single word. How to find the predecessor of another r bit number in such collection efficiently?

Parallel searching for x

We assume that the numbers are separated by single bits, so we actually store $0a_1 0a_2 .. 0a_r$, and $a_1 < a_2 < \ldots < a_r$.

1. OR with $10^r 10^r .. 10^r$.
2. subtract $0x0x..0x$.
3. AND with $10^r 10^r .. 10^r$.
4. compute the MSB.

This counts how many a_i's are smaller than x, which gives us the position of the predecessor.

Searching in a collection of sketches

So now our situation is that we have a sorted collection of r bit numbers packed in a single word. How to find the predecessor of another r bit number in such collection efficiently?

Parallel searching for x

We assume that the numbers are separated by single bits, so we actually store $0a_1 0a_2 .. 0a_r$, and $a_1 < a_2 < \ldots < a_r$.

1. OR with $10^r 10^r .. 10^r$.
2. subtract $0x0x..0x$.
3. AND with $10^r 10^r .. 10^r$.
4. compute the MSB.

This counts how many a_i's are smaller than x, which gives us the position of the predecessor.

Searching in a collection of sketches

So now our situation is that we have a sorted collection of r bit numbers packed in a single word. How to find the predecessor of another r bit number in such collection efficiently?

Parallel searching for x

We assume that the numbers are separated by single bits, so we actually store $0a_1 0a_2 .. 0a_r$, and $a_1 < a_2 < \ldots < a_r$.

1. OR with $10^r 10^r .. 10^r$.
2. subtract $0x0x..0x$.
3. AND with $10^r 10^r .. 10^r$.
4. compute the MSB.

This counts how many a_i's are smaller than x, which gives us the position of the predecessor.

Approximate sketching

Unfortunately, it turns out that computing the sketch of a given x is difficult to implement in $\mathcal{O}(1)$ time.

Clever idea

We don't actually need all the bits at positions $b_1, b_2, \ldots, b_r$ to be arranged one after another. We can spread them a little bit, as long as they appear in the same order.

We will show how to compute an *approximate sketch* of any x in $\mathcal{O}(1)$ time.

Approximate sketch

If the important positions, i.e., the ones corresponding to a branching node in the trie, are $b_1, b_2, \ldots, b_r$, the approximate sketch of x is of length r^4 and contains the bits $x_{b_r}, x_{b_{r-1}}, \ldots, x_{b_1}$ separated by zeroes. The order of these bits MUST be the same as in the original x!

Approximate sketching

Unfortunately, it turns out that computing the sketch of a given x is difficult to implement in $\mathcal{O}(1)$ time.

Clever idea

We don't actually need all the bits at positions $b_1, b_2, \ldots, b_r$ to be arranged one after another. We can spread them a little bit, as long as they appear in the same order.

We will show how to compute an *approximate sketch* of any x in $\mathcal{O}(1)$ time.

Approximate sketch

If the important positions, i.e., the ones corresponding to a branching node in the trie, are $b_1, b_2, \ldots, b_r$, the approximate sketch of x is of length r^4 and contains the bits $x_{b_r}, x_{b_{r-1}}, \ldots, x_{b_1}$ separated by zeroes. The order of these bits MUST be the same as in the original x!

Approximate sketching

Unfortunately, it turns out that computing the sketch of a given x is difficult to implement in $\mathcal{O}(1)$ time.

Clever idea

We don't actually need all the bits at positions $b_1, b_2, \ldots, b_r$ to be arranged one after another. We can spread them a little bit, as long as they appear in the same order.

We will show how to compute an *approximate sketch* of any x in $\mathcal{O}(1)$ time.

Approximate sketch

If the important positions, i.e., the ones corresponding to a branching node in the trie, are $b_1, b_2, \ldots, b_r$, the approximate sketch of x is of length r^4 and contains the bits $x_{b_r}, x_{b_{r-1}}, \ldots, x_{b_1}$ separated by zeroes. The order of these bits MUST be the same as in the original x!

Approximate sketching

Unfortunately, it turns out that computing the sketch of a given x is difficult to implement in $\mathcal{O}(1)$ time.

Clever idea

We don't actually need all the bits at positions $b_1, b_2, \ldots, b_r$ to be arranged one after another. We can spread them a little bit, as long as they appear in the same order.

We will show how to compute an *approximate sketch* of any x in $\mathcal{O}(1)$ time.

Approximate sketch

If the important positions, i.e., the ones corresponding to a branching node in the trie, are $b_1, b_2, \ldots, b_r$, the approximate sketch of x is of length r^4 and contains the bits $x_{b_r}, x_{b_{r-1}}, \ldots, x_{b_1}$ separated by zeroes. The order of these bits MUST be the same as in the original x!

Approximate sketching

First of all, we AND x with $\sum_{i=1}^{r} 2^{b_i}$ to get rid of the boring bits. We call the result x'.

We will multiply x' by some appropriately chosen M, which depend only on $b_1, b_2, \ldots, b_r$. Say that $M = \sum_{j=1}^{r} 2^{m_j}$ and look at $x' \cdot M$:

$$x' \cdot M = \sum_{i=1}^{r} x_{b_i} 2^{b_i} \sum_{j=1}^{r} 2^{m_j} = \sum_{i=1}^{r} \sum_{j=1}^{r} x_{b_i} 2^{b_i + m_j}$$

We want:

1. all $b_i + m_j$ to be different,
2. $b_1 + m_1 < b_2 + m_2 < \ldots < b_r + m_r$,
3. $b_1 + m_1$ and $b_r + m_r$ differ by at most r^4.

If we can achieve this, then we can take $x' \cdot M$, AND it with $\sum_{i=1}^{r} 2^{b_i + m_i}$, and finally shift the result so that it fits in the r^4 least significant bits. Notice that during the computation we might need double precision, because both x' and M will consist of w bits. (But this is OK.)

Approximate sketching

First of all, we AND x with $\sum_{i=1}^{r} 2^{b_i}$ to get rid of the boring bits. We call the result x'.

We will multiply x' by some appropriately chosen M, which depend only on $b_1, b_2, \ldots, b_r$. Say that $M = \sum_{j=1}^{r} 2^{m_j}$ and look at $x' \cdot M$:

$$x' \cdot M = \sum_{i=1}^{r} x_{b_i} 2^{b_i} \sum_{j=1}^{r} 2^{m_j} = \sum_{i=1}^{r} \sum_{j=1}^{r} x_{b_i} 2^{b_i + m_j}$$

We want:

1. all $b_i + m_j$ to be different,
2. $b_1 + m_1 < b_2 + m_2 < \ldots < b_r + m_r$,
3. $b_1 + m_1$ and $b_r + m_r$ differ by at most r^4.

If we can achieve this, then we can take $x' \cdot M$, AND it with $\sum_{i=1}^{r} 2^{b_i + m_i}$, and finally shift the result so that it fits in the r^4 least significant bits. Notice that during the computation we might need double precision, because both x' and M will consist of w bits. (But this is OK.)

Approximate sketching

First of all, we AND x with $\sum_{i=1}^{r} 2^{b_i}$ to get rid of the boring bits. We call the result x'.

We will multiply x' by some appropriately chosen M, which depend only on $b_1, b_2, \ldots, b_r$. Say that $M = \sum_{j=1}^{r} 2^{m_j}$ and look at $x' \cdot M$:

$$x' \cdot M = \sum_{i=1}^{r} x_{b_i} 2^{b_i} \sum_{j=1}^{r} 2^{m_j} = \sum_{i=1}^{r} \sum_{j=1}^{r} x_{b_i} 2^{b_i + m_j}$$

We want:

1. all $b_i + m_j$ to be different,
2. $b_1 + m_1 < b_2 + m_2 < \ldots < b_r + m_r$,
3. $b_1 + m_1$ and $b_r + m_r$ differ by at most r^4.

If we can achieve this, then we can take $x' \cdot M$, AND it with $\sum_{i=1}^{r} 2^{b_i + m_i}$, and finally shift the result so that it fits in the r^4 least significant bits. Notice that during the computation we might need double precision, because both x' and M will consist of w bits. (But this is OK.)

Approximate sketching

First of all, we AND x with $\sum_{i=1}^{r} 2^{b_i}$ to get rid of the boring bits. We call the result x'.

We will multiply x' by some appropriately chosen M, which depend only on $b_1, b_2, \ldots, b_r$. Say that $M = \sum_{j=1}^{r} 2^{m_j}$ and look at $x' \cdot M$:

$$x' \cdot M = \sum_{i=1}^{r} x_{b_i} 2^{b_i} \sum_{j=1}^{r} 2^{m_j} = \sum_{i=1}^{r} \sum_{j=1}^{r} x_{b_i} 2^{b_i + m_j}$$

We want:

1. all $b_i + m_j$ to be different,
2. $b_1 + m_1 < b_2 + m_2 < \ldots < b_r + m_r$,
3. $b_1 + m_1$ and $b_r + m_r$ differ by at most r^4.

If we can achieve this, then we can take $x' \cdot M$, AND it with $\sum_{i=1}^{r} 2^{b_i + m_i}$, and finally shift the result so that it fits in the r^4 least significant bits. Notice that during the computation we might need double precision, because both x' and M will consist of w bits. (But this is OK.)

Approximate sketching

First of all, we AND x with $\sum_{i=1}^{r} 2^{b_i}$ to get rid of the boring bits. We call the result x'.

We will multiply x' by some appropriately chosen M, which depend only on $b_1, b_2, \ldots, b_r$. Say that $M = \sum_{j=1}^{r} 2^{m_j}$ and look at $x' \cdot M$:

$$x' \cdot M = \sum_{i=1}^{r} x_{b_i} 2^{b_i} \sum_{j=1}^{r} 2^{m_j} = \sum_{i=1}^{r} \sum_{j=1}^{r} x_{b_i} 2^{b_i + m_j}$$

We want:

1. all $b_i + m_j$ to be different,
2. $b_1 + m_1 < b_2 + m_2 < \ldots < b_r + m_r$,
3. $b_1 + m_1$ and $b_r + m_r$ differ by at most r^4.

If we can achieve this, then we can take $x' \cdot M$, AND it with $\sum_{i=1}^{r} 2^{b_i + m_i}$, and finally shift the result so that it fits in the r^4 least significant bits. Notice that during the computation we might need double precision, because both x' and M will consist of w bits. (But this is OK.)

Approximate sketching

First of all, we AND x with $\sum_{i=1}^{r} 2^{b_i}$ to get rid of the boring bits. We call the result x'.

We will multiply x' by some appropriately chosen M, which depend only on $b_1, b_2, \ldots, b_r$. Say that $M = \sum_{j=1}^{r} 2^{m_j}$ and look at $x' \cdot M$:

$$x' \cdot M = \sum_{i=1}^{r} x_{b_i} 2^{b_i} \sum_{j=1}^{r} 2^{m_j} = \sum_{i=1}^{r} \sum_{j=1}^{r} x_{b_i} 2^{b_i + m_j}$$

We want:

1. all $b_i + m_j$ to be different,
2. $b_1 + m_1 < b_2 + m_2 < \ldots < b_r + m_r$,
3. $b_1 + m_1$ and $b_r + m_r$ differ by at most r^4.

If we can achieve this, then we can take $x' \cdot M$, AND it with $\sum_{i=1}^{r} 2^{b_i + m_i}$, and finally shift the result so that it fits in the r^4 least significant bits.

Notice that during the computation we might need double precision, because both x' and M will consist of w bits. (But this is OK.)

Approximate sketching

1. all $b_i + m_j$ are different,
2. $b_1 + m_1 < b_2 + m_2 < \ldots < b_r + m_r$,
3. $b_1 + m_1$ and $b_r + m_r$ differ by at most r^4.

First we choose the remainder of every m_i modulo r^3 as to guarantee (1). Then we add appropriate multiples of r^3 as to guarantee (2).

Choosing the remainders

Say that we have already chosen $m_1, m_2, \ldots, m_{j-1}$. What are the constraints on m_j? $b_i + m_j \neq b_{i'} + m_{j'} \bmod r^3$ for all i, i' and $j' < j$. There are less than r^3 "blocked" values of m_j.

Spreading

We want to choose the final values $m'_j = m_j + \alpha_j r^3$ so that (2) holds. It is enough to guarantee that $w + r^3(i-1) < m'_i + b_i < w + r^3 i$, and this is always possible by adjusting α.

Approximate sketching

1. all $b_i + m_j$ are different,
2. $b_1 + m_1 < b_2 + m_2 < \ldots < b_r + m_r$,
3. $b_1 + m_1$ and $b_r + m_r$ differ by at most r^4.

First we choose the remainder of every m_i modulo r^3 as to guarantee (1). Then we add appropriate multiples of r^3 as to guarantee (2).

Choosing the remainders

Say that we have already chosen $m_1, m_2, \ldots, m_{j-1}$. What are the constraints on m_j? $b_i + m_j \neq b_{i'} + m_{j'} \bmod r^3$ for all i, i' and $j' < j$. There are less than r^3 "blocked" values of m_j.

Spreading

We want to choose the final values $m'_j = m_j + \alpha_j r^3$ so that (2) holds. It is enough to guarantee that $w + r^3(i-1) < m'_i + b_i < w + r^3 i$, and this is always possible by adjusting α.

Approximate sketching

1. all $b_i + m_j$ are different,
2. $b_1 + m_1 < b_2 + m_2 < \ldots < b_r + m_r$,
3. $b_1 + m_1$ and $b_r + m_r$ differ by at most r^4.

First we choose the remainder of every m_i modulo r^3 as to guarantee (1). Then we add appropriate multiples of r^3 as to guarantee (2).

Choosing the remainders

Say that we have already chosen $m_1, m_2, \ldots, m_{j-1}$. What are the constraints on m_j? $b_i + m_j \neq b_{i'} + m_{j'} \bmod r^3$ for all i, i' and $j' < j$. There are less than r^3 "blocked" values of m_j.

Spreading

We want to choose the final values $m'_j = m_j + \alpha_j r^3$ so that (2) holds. It is enough to guarantee that $w + r^3(i-1) < m'_i + b_i < w + r^3 i$, and this is always possible by adjusting α.

Approximate sketching

1. all $b_i + m_j$ are different,
2. $b_1 + m_1 < b_2 + m_2 < \ldots < b_r + m_r$,
3. $b_1 + m_1$ and $b_r + m_r$ differ by at most r^4.

First we choose the remainder of every m_i modulo r^3 as to guarantee (1). Then we add appropriate multiples of r^3 as to guarantee (2).

Choosing the remainders

Say that we have already chosen $m_1, m_2, \ldots, m_{j-1}$. What are the constraints on m_j? $b_i + m_j \neq b_{i'} + m_{j'} \bmod r^3$ for all i, i' and $j' < j$. There are less than r^3 "blocked" values of m_j.

Spreading

We want to choose the final values $m'_j = m_j + \alpha_j r^3$ so that (2) holds. It is enough to guarantee that $w + r^3(i-1) < m'_i + b_i < w + r^3 i$, and this is always possible by adjusting α.

Approximate sketching

Because $w + r^3(i-1) < m'_i + b_i < w + r^3 i$, $b_1 + m_1 \geq w$ and $b_r + m_r < w + r^4$, so (3) is guaranteed as well.

Finally, we need to pack r approximate sketches in a single word, so we choose r so that $r \cdot r^4 = w$. This is why $r = w^{1/5}$.

Approximate sketching

Because $w + r^3(i-1) < m'_i + b_i < w + r^3 i$, $b_1 + m_1 \geq w$ and $b_r + m_r < w + r^4$, so (3) is guaranteed as well.
Finally, we need to pack r approximate sketches in a single word, so we choose r so that $r \cdot r^4 = w$. This is why $r = w^{1/5}$.

Construction time

So far we haven't been very concerned with the construction time, but actually it was always (maybe after a tweak or two) linear in the size of the structure. Is this still true for fusion trees?

Constructing the small structures

It is not very clear how to find all m_i' faster than in r^4 time, so the construction time for the small structures is around $\mathcal{O}(r^4)$, which is polynomial in its size (so, not very bad). But not linear (so, not very good)!

Of course, we already known a trick which allows us to fix this: use indirection with blocks of size B^4 (recall that $B = w^{1/5}$). Then the search time becomes $\mathcal{O}(\frac{\log n}{\log w} + \log w)$.

Dynamization

This trick is actually enough to maintain a dynamic fusion tree in linear space and $\mathcal{O}(\frac{\log n}{\log w} + \log w)$ time for insert, delete, and search.

Construction time

So far we haven't been very concerned with the construction time, but actually it was always (maybe after a tweak or two) linear in the size of the structure. Is this still true for fusion trees?

Constructing the small structures

It is not very clear how to find all m'_i faster than in r^4 time, so the construction time for the small structures is around $\mathcal{O}(r^4)$, which is polynomial in its size (so, not very bad). But not linear (so, not very good)!

Of course, we already known a trick which allows us to fix this: use indirection with blocks of size B^4 (recall that $B = w^{1/5}$). Then the search time becomes $\mathcal{O}(\frac{\log n}{\log w} + \log w)$.

Dynamization

This trick is actually enough to maintain a dynamic fusion tree in linear space and $\mathcal{O}(\frac{\log n}{\log w} + \log w)$ time for insert, delete, and search.

Construction time

So far we haven't been very concerned with the construction time, but actually it was always (maybe after a tweak or two) linear in the size of the structure. Is this still true for fusion trees?

Constructing the small structures

It is not very clear how to find all m'_i faster than in r^4 time, so the construction time for the small structures is around $\mathcal{O}(r^4)$, which is polynomial in its size (so, not very bad). But not linear (so, not very good)!

Of course, we already known a trick which allows us to fix this: use indirection with blocks of size B^4 (recall that $B = w^{1/5}$). Then the search time becomes $\mathcal{O}(\frac{\log n}{\log w} + \log w)$.

Dynamization

This trick is actually enough to maintain a dynamic fusion tree in linear space and $\mathcal{O}(\frac{\log n}{\log w} + \log w)$ time for insert, delete, and search.

Dynamic fusion tree

Two ingredients:

1. As usually, the bottom structures of size B^4 are implemented using any BST, which adds $\mathcal{O}(\log B^4) = \mathcal{O}(\log w)$ time to every operation.
2. For the top structure, we use a B-tree. We need to show how to update such tree efficiently.

We will show that any insert/delete operation on a B-tree requires modifying just $\log_B n$ "big nodes".

Now, modifying a node of the top structure costs B^4, so the total cost of an insert/delete operation is $\mathcal{O}(B^4 \log_B n)$. But we need to update the top structure just once per roughly B^4 insert/delete operations, so the amortized cost of an insert/delete operation is:

1. $\mathcal{O}(\log_B n)$ to locate the appropriate bottom structure,
2. $\mathcal{O}(\log B^4)$ to update the bottom structure,
3. $\mathcal{O}(\log_B n)$ to update the top structure.

Dynamic fusion tree

Two ingredients:

1. As usually, the bottom structures of size B^4 are implemented using any BST, which adds $\mathcal{O}(\log B^4) = \mathcal{O}(\log w)$ time to every operation.
2. For the top structure, we use a B-tree. We need to show how to update such tree efficiently.

We will show that any insert/delete operation on a B-tree requires modifying just $\log_B n$ "big nodes".

Now, modifying a node of the top structure costs B^4, so the total cost of an insert/delete operation is $\mathcal{O}(B^4 \log_B n)$. But we need to update the top structure just once per roughly B^4 insert/delete operations, so the amortized cost of an insert/delete operation is:

1. $\mathcal{O}(\log_B n)$ to locate the appropriate bottom structure,
2. $\mathcal{O}(\log B^4)$ to update the bottom structure,
3. $\mathcal{O}(\log_B n)$ to update the top structure.

Dynamic fusion tree

Two ingredients:

1. As usually, the bottom structures of size B^4 are implemented using any BST, which adds $\mathcal{O}(\log B^4) = \mathcal{O}(\log w)$ time to every operation.
2. For the top structure, we use a B-tree. We need to show how to update such tree efficiently.

We will show that any insert/delete operation on a B-tree requires modifying just $\log_B n$ "big nodes".

Now, modifying a node of the top structure costs B^4, so the total cost of an insert/delete operation is $\mathcal{O}(B^4 \log_B n)$. But we need to update the top structure just once per roughly B^4 insert/delete operations, so the amortized cost of an insert/delete operation is:

1. $\mathcal{O}(\log_B n)$ to locate the appropriate bottom structure,
2. $\mathcal{O}(\log B^4)$ to update the bottom structure,
3. $\mathcal{O}(\log_B n)$ to update the top structure.

Dynamic fusion tree

Two ingredients:

1. As usually, the bottom structures of size B^4 are implemented using any BST, which adds $\mathcal{O}(\log B^4) = \mathcal{O}(\log w)$ time to every operation.
2. For the top structure, we use a B-tree. We need to show how to update such tree efficiently.

We will show that any insert/delete operation on a B-tree requires modifying just $\log_B n$ "big nodes".

Now, modifying a node of the top structure costs B^4, so the total cost of an insert/delete operation is $\mathcal{O}(B^4 \log_B n)$. But we need to update the top structure just once per roughly B^4 insert/delete operations, so the amortized cost of an insert/delete operation is:

1. $\mathcal{O}(\log_B n)$ to locate the appropriate bottom structure,
2. $\mathcal{O}(\log B^4)$ to update the bottom structure,
3. $\mathcal{O}(\log_B n)$ to update the top structure.

Dynamic fusion tree

Two ingredients:

1. As usually, the bottom structures of size B^4 are implemented using any BST, which adds $\mathcal{O}(\log B^4) = \mathcal{O}(\log w)$ time to every operation.
2. For the top structure, we use a B-tree. We need to show how to update such tree efficiently.

We will show that any insert/delete operation on a B-tree requires modifying just $\log_B n$ "big nodes".

Now, modifying a node of the top structure costs B^4, so the total cost of an insert/delete operation is $\mathcal{O}(B^4 \log_B n)$. But we need to update the top structure just once per roughly B^4 insert/delete operations, so the amortized cost of an insert/delete operation is:

1. $\mathcal{O}(\log_B n)$ to locate the appropriate bottom structure,
2. $\mathcal{O}(\log B^4)$ to update the bottom structure,
3. $\mathcal{O}(\log_B n)$ to update the top structure.

Dynamic fusion tree

Two ingredients:

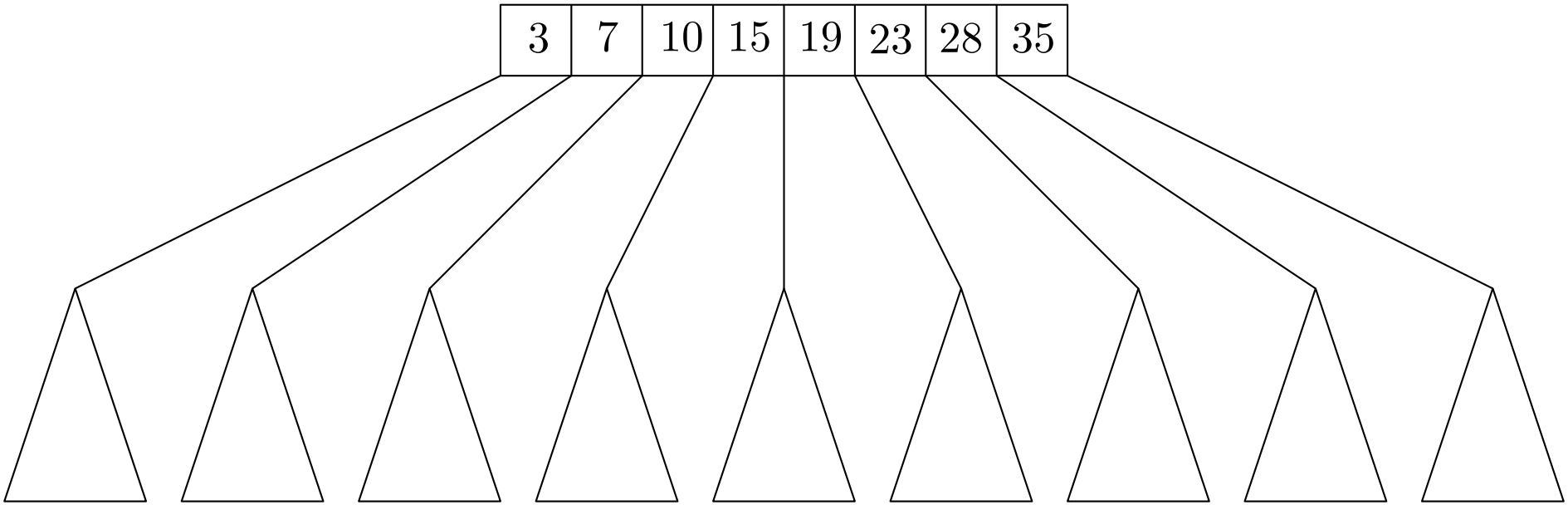
1. As usually, the bottom structures of size B^4 are implemented using any BST, which adds $\mathcal{O}(\log B^4) = \mathcal{O}(\log w)$ time to every operation.
2. For the top structure, we use a B-tree. We need to show how to update such tree efficiently.

We will show that any insert/delete operation on a B-tree requires modifying just $\log_B n$ "big nodes".

Now, modifying a node of the top structure costs B^4, so the total cost of an insert/delete operation is $\mathcal{O}(B^4 \log_B n)$. But we need to update the top structure just once per roughly B^4 insert/delete operations, so the amortized cost of an insert/delete operation is:

1. $\mathcal{O}(\log_B n)$ to locate the appropriate bottom structure,
2. $\mathcal{O}(\log B^4)$ to update the bottom structure,
3. $\mathcal{O}(\log_B n)$ to update the top structure.

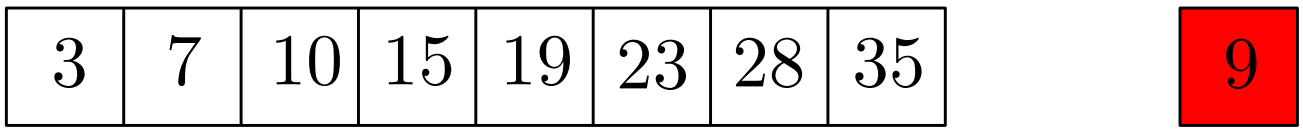
B-trees



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

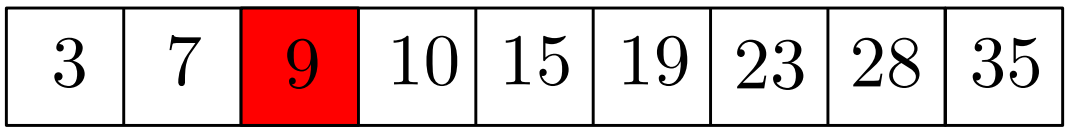
Inserting into a B-tree

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 $B = 5$.



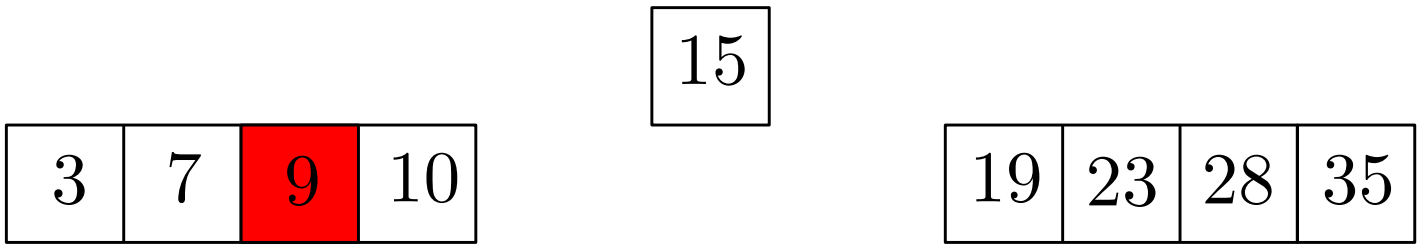
Inserting into a B-tree

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 $B = 5$.



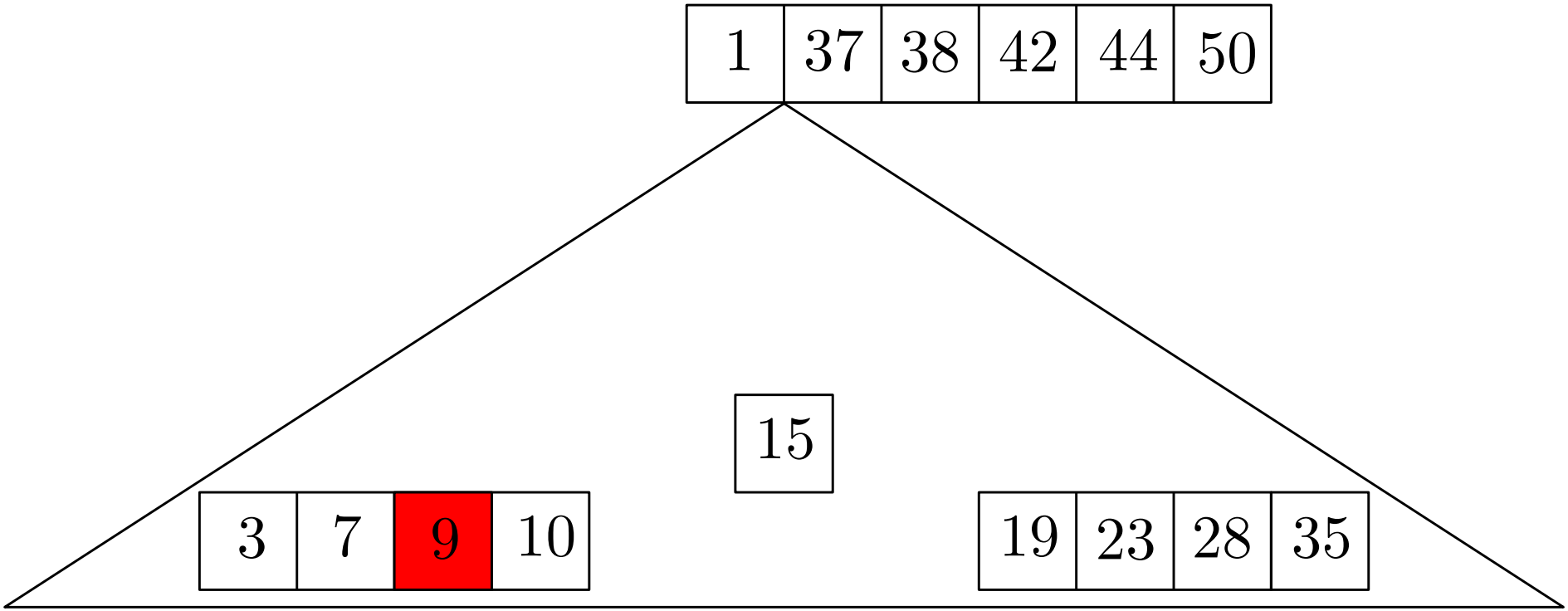
Inserting into a B-tree

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 $B = 5$.



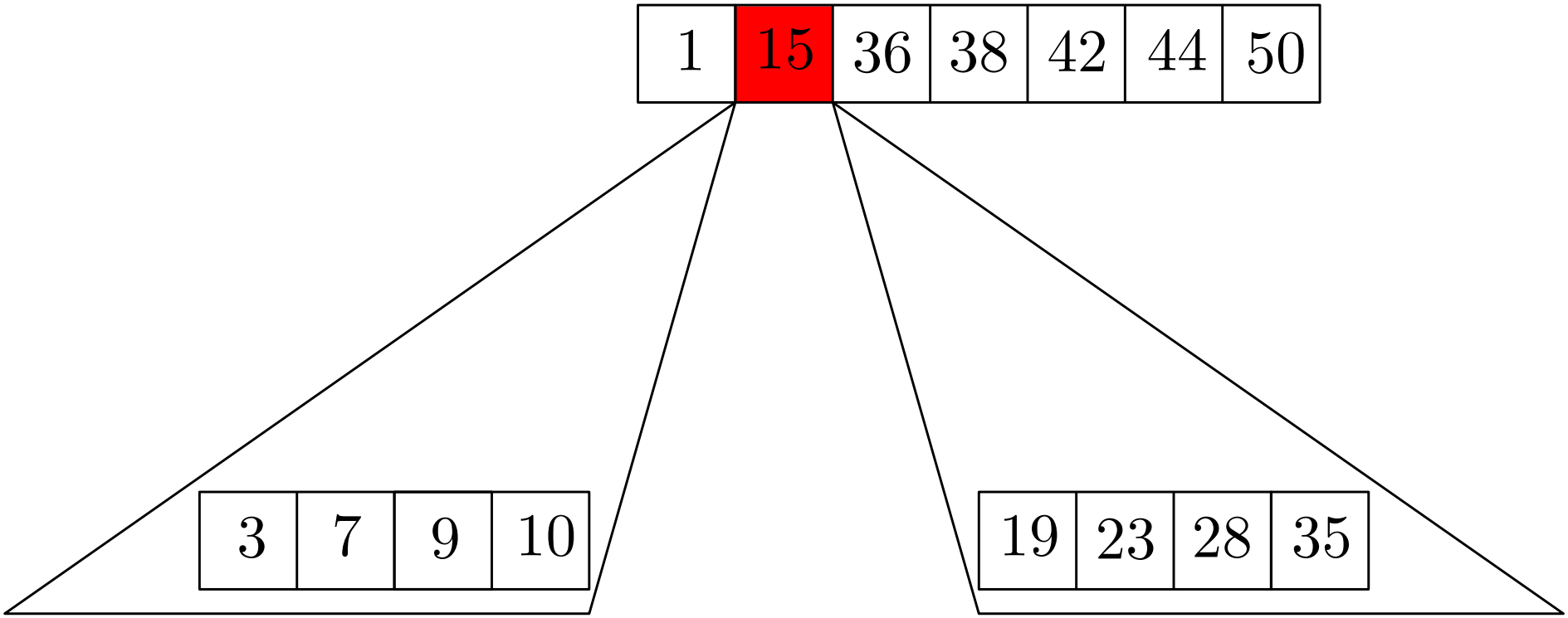
Inserting into a B-tree

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

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.

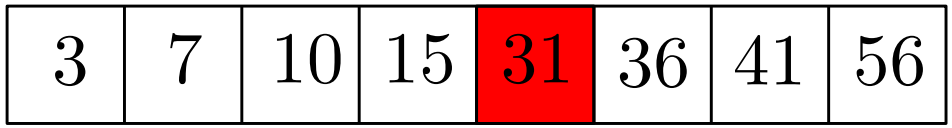
First we locate the number which should be removed by traversing starting from the root. Remove it from the node it belongs to and fill the hole with either the largest number from the left subtree or the smallest number from the right subtree.

Deleting from B-tree

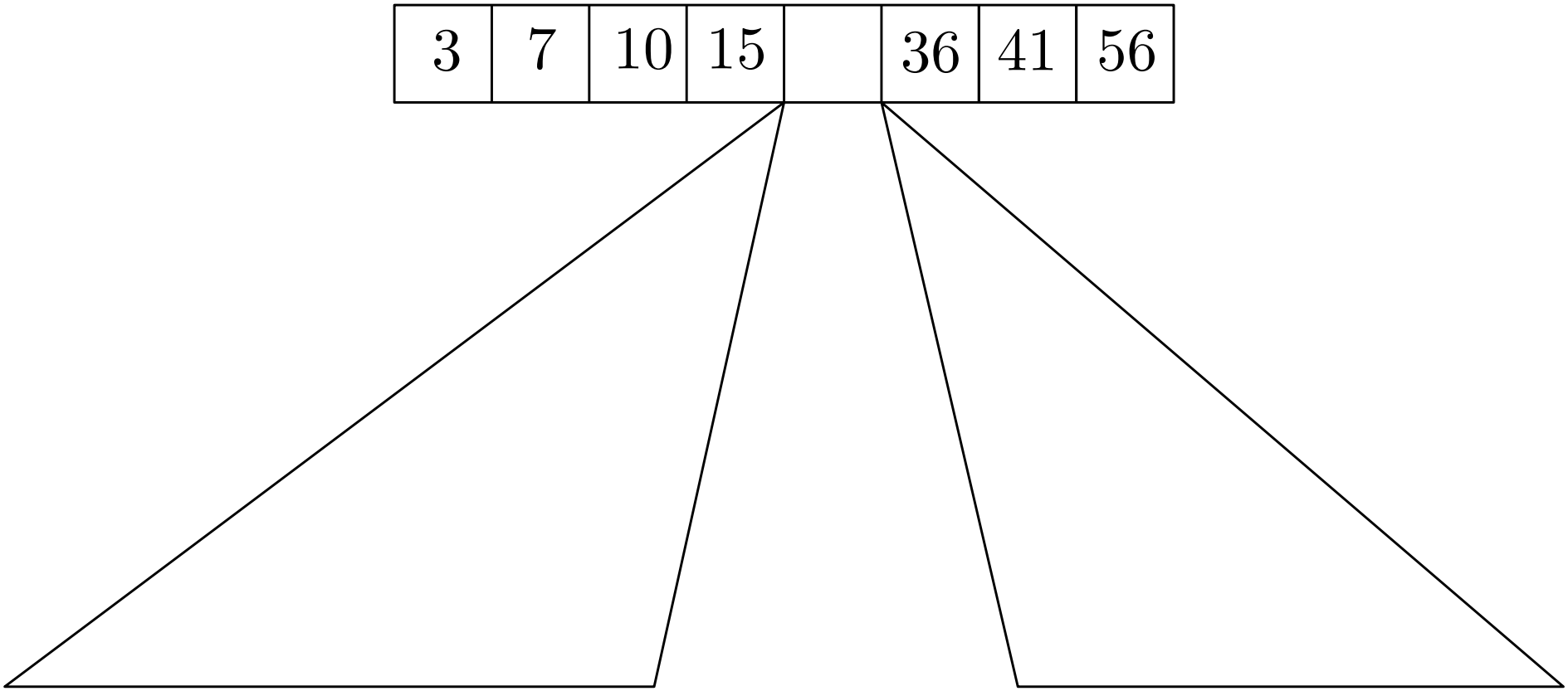
More messy! But let's try anyway.

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.

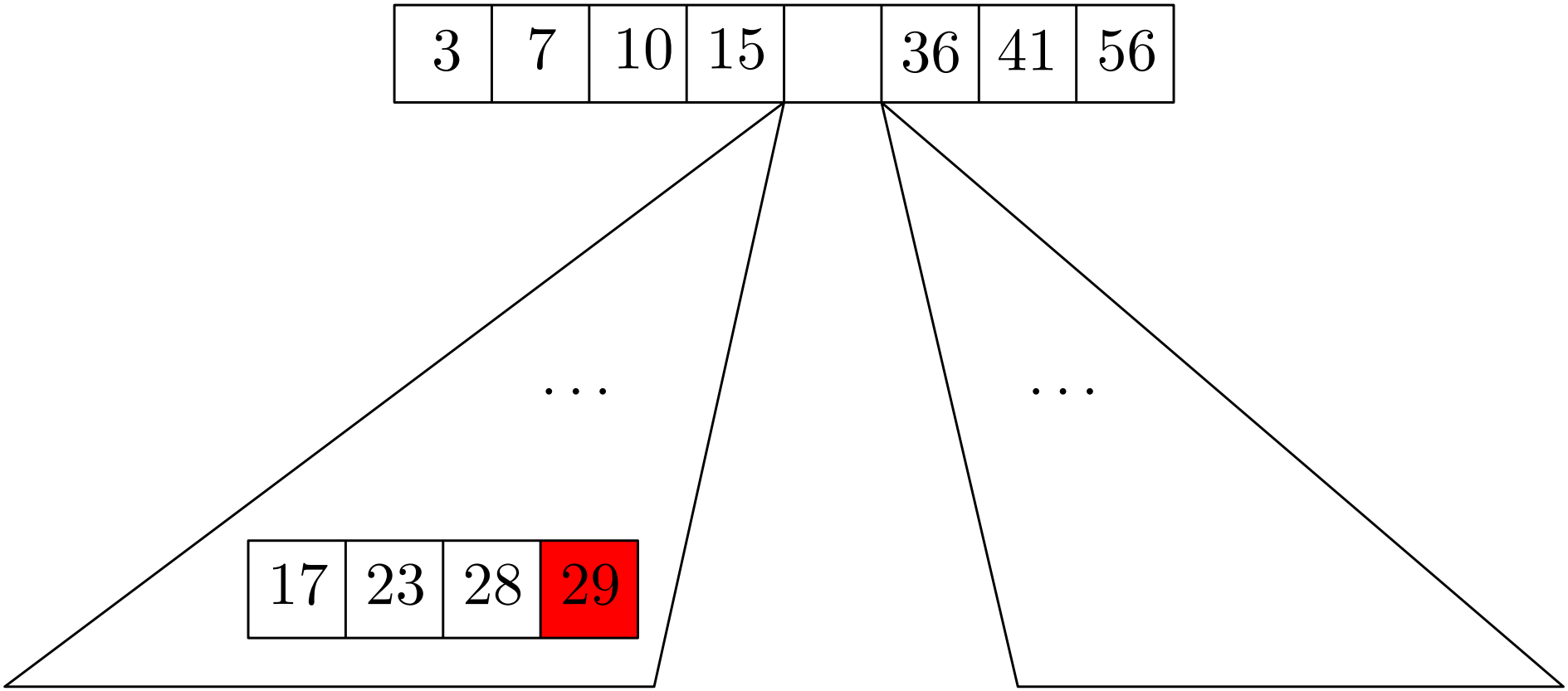
Deleting from B-tree



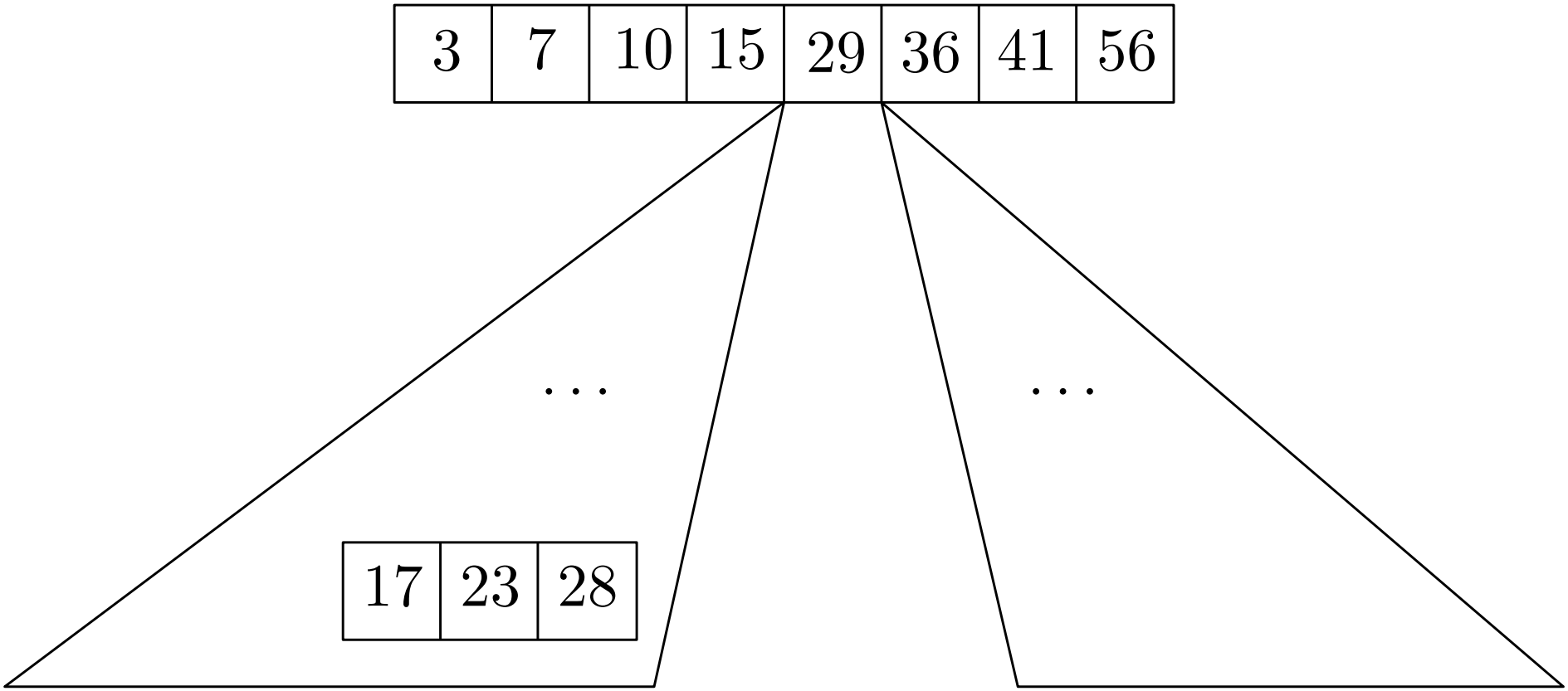
Deleting from B-tree



Deleting from B-tree

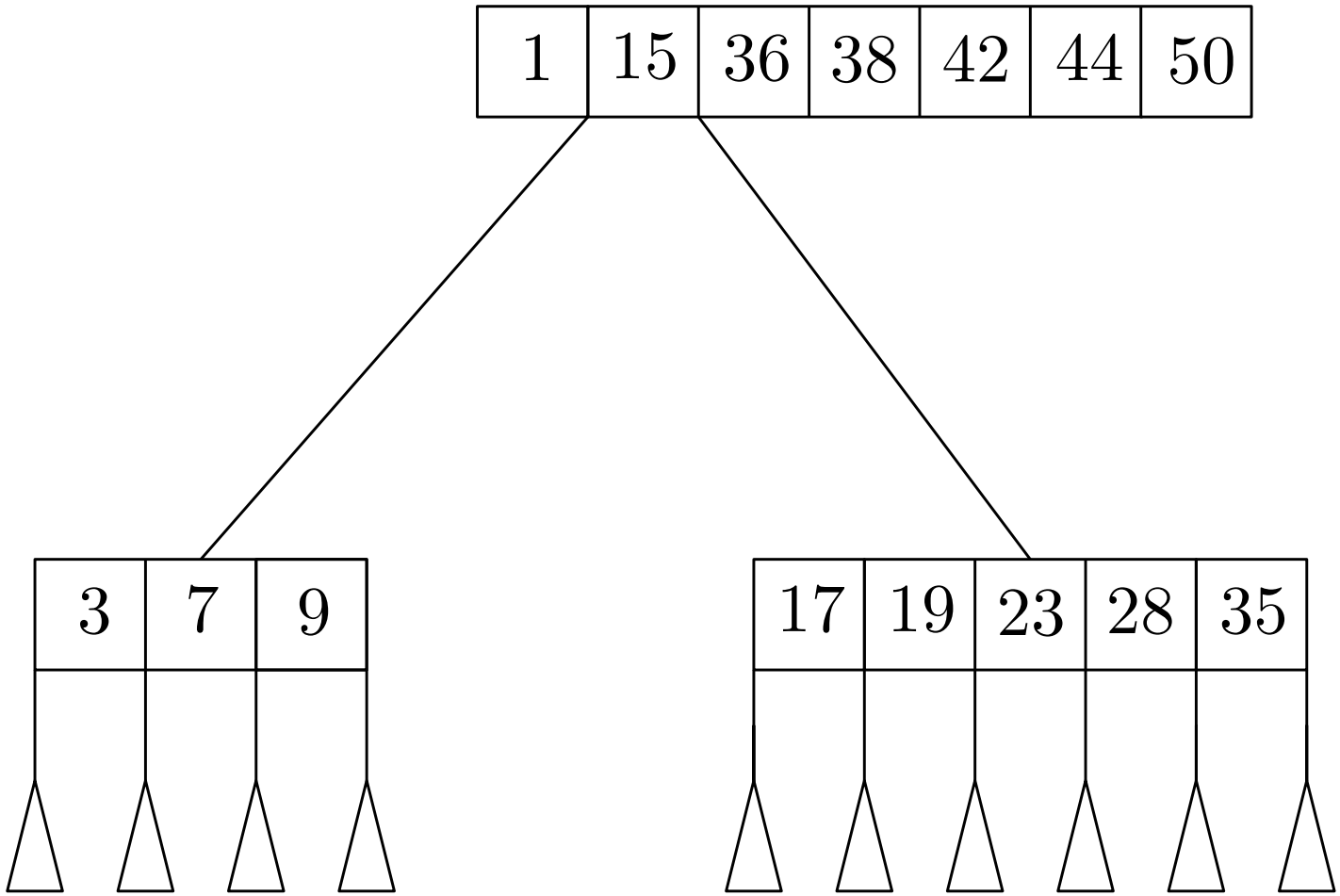


Deleting from B-tree



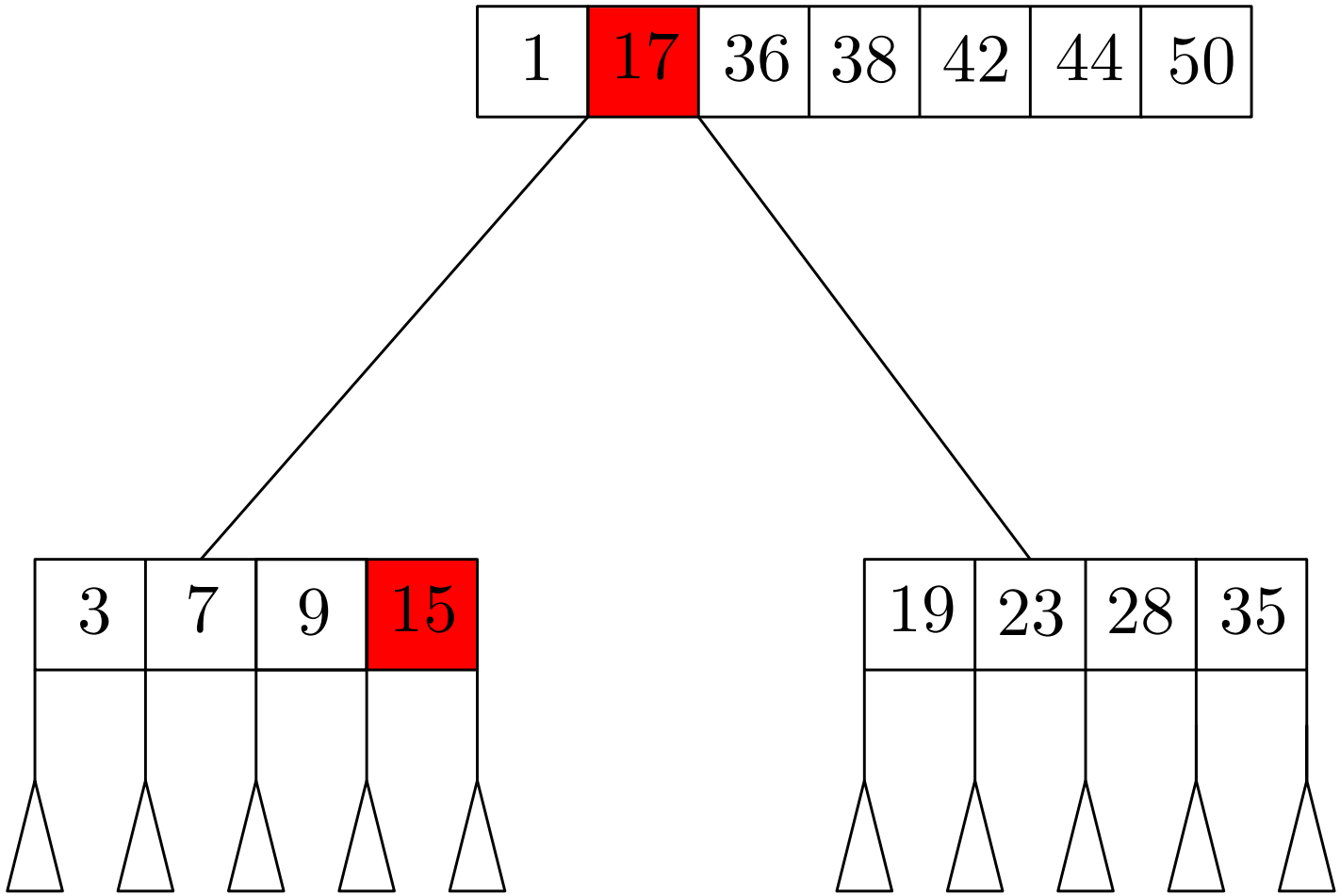
Deleting from B-tree

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



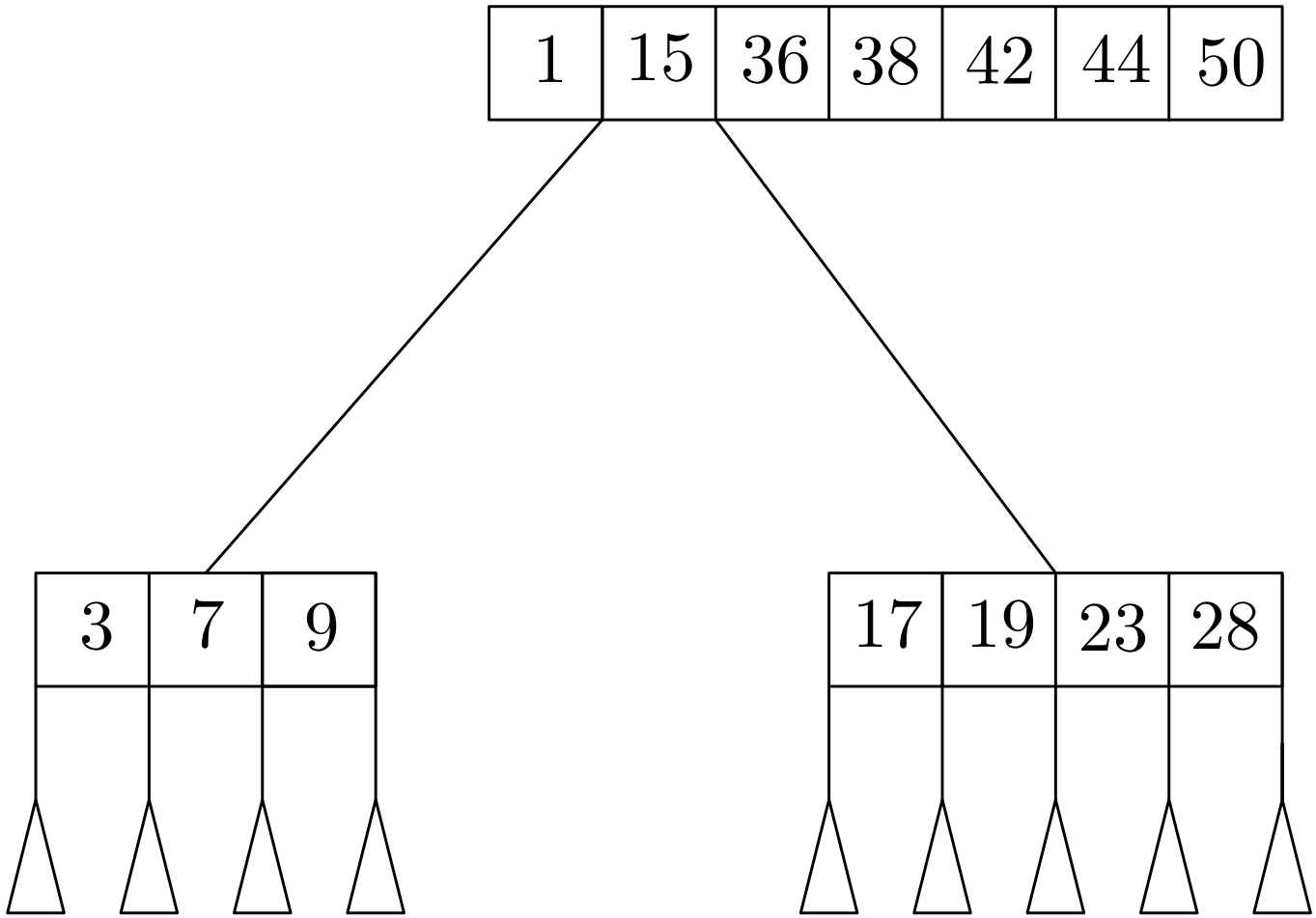
Deleting from B-tree

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



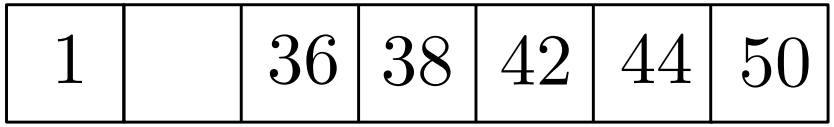
Deleting from B-tree

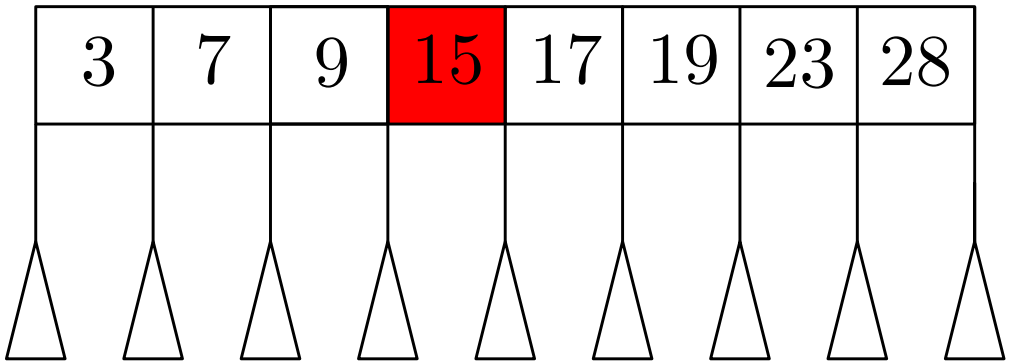
Now we have a B-tree where some leaf might contain just $B - 2$ numbers, so we must be able to fix this.



Deleting from B-tree

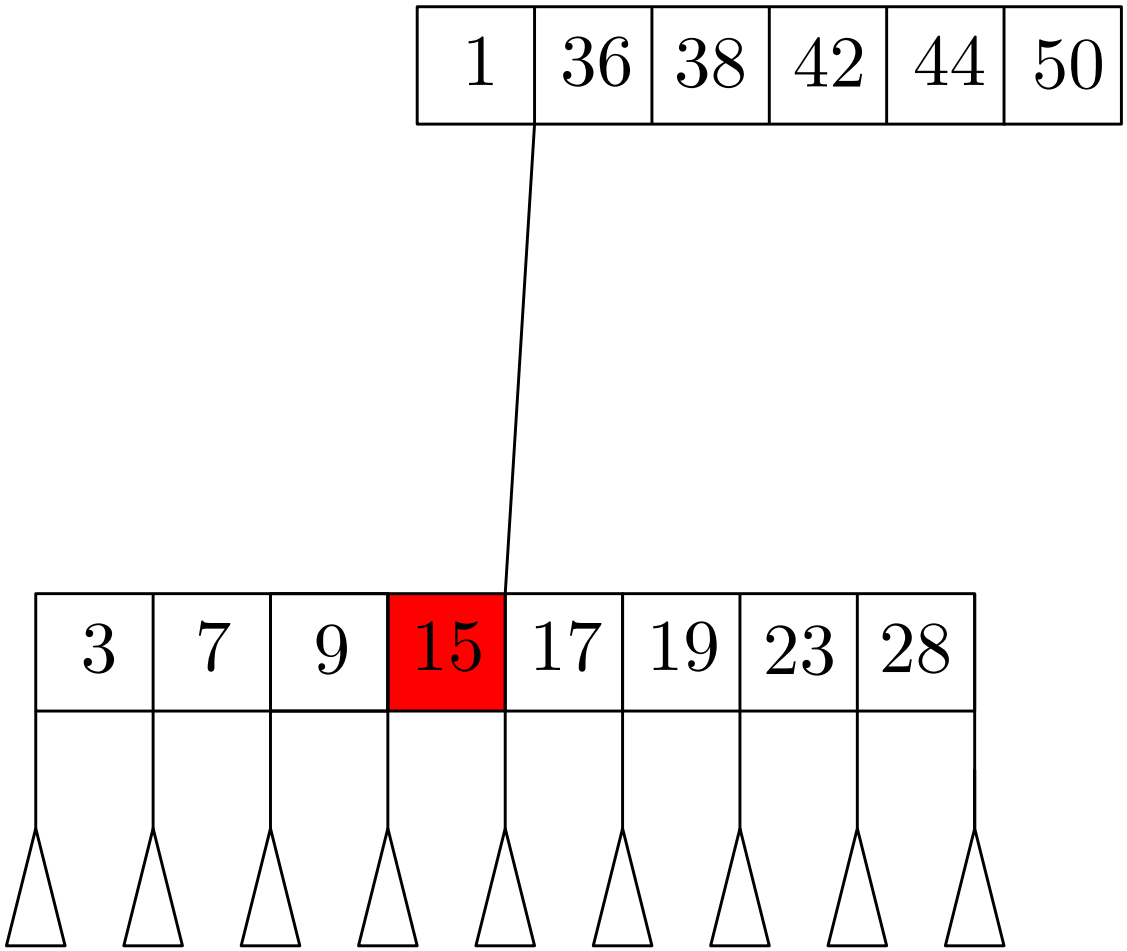
Now we have a B-tree where some leaf might contain just $B - 2$ numbers, so we must be able to fix this.





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.

Questions?