7.4 Augmenting Data Structures

Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

1. We choose a red-black tree as the underlying data-structure.
2. We store in each node v the size of the sub-tree rooted at v.
3. We need to be able to update the size-field in each node without asymptotically affecting the running time of insert, delete, and search. We come back to this step later...

7.4 Augmenting Data Structures

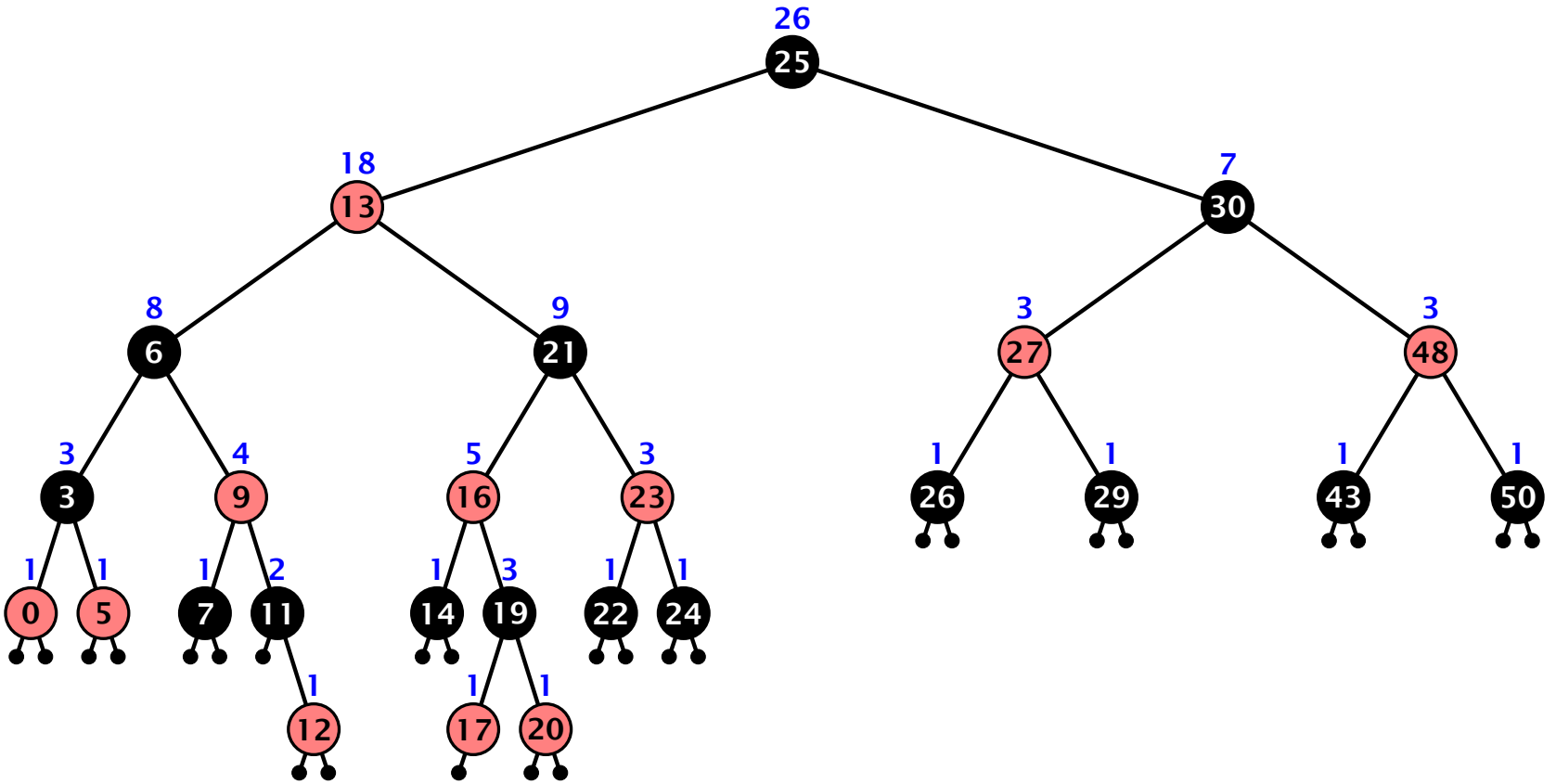
Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

4. How does find-by-rank work?
   Find-by-rank(k) := Select(root,k) with

Algorithm 11 Select(x, i)

```
1: if x = null then return error
2: if left[x] ≠ null then r ← left[x].size +1 else r ← 1
3: if i = r then return x
4: if i < r then
5:         return Select(left[x], i)
6: else
7:         return Select(right[x], i − r)
```

Select(x, i)

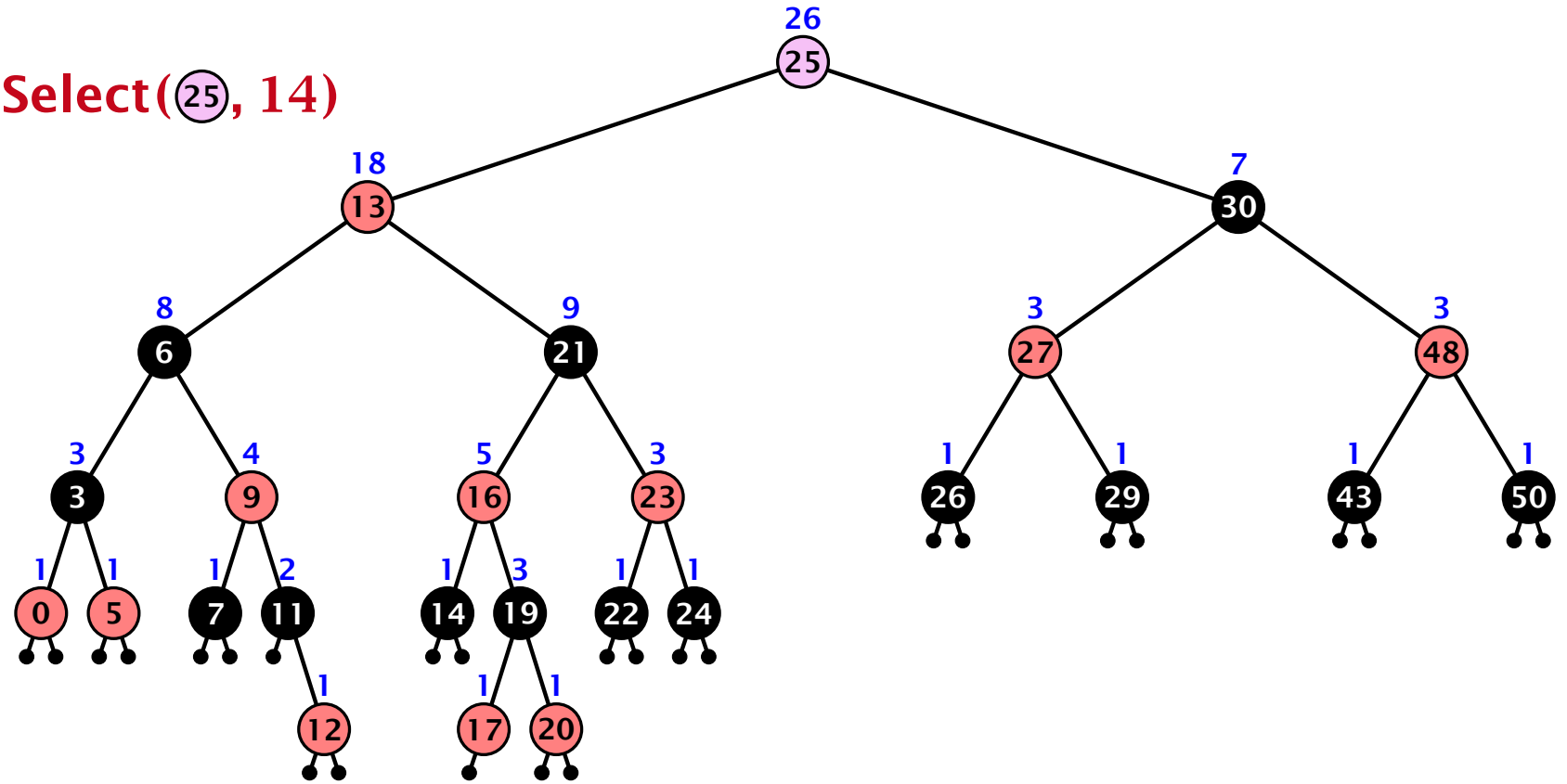


Find-by-rank:

- decide whether you have to proceed into the left or right sub-tree
- adjust the rank that you are searching for if you go right

Select(x, i)

Select(25, 14)

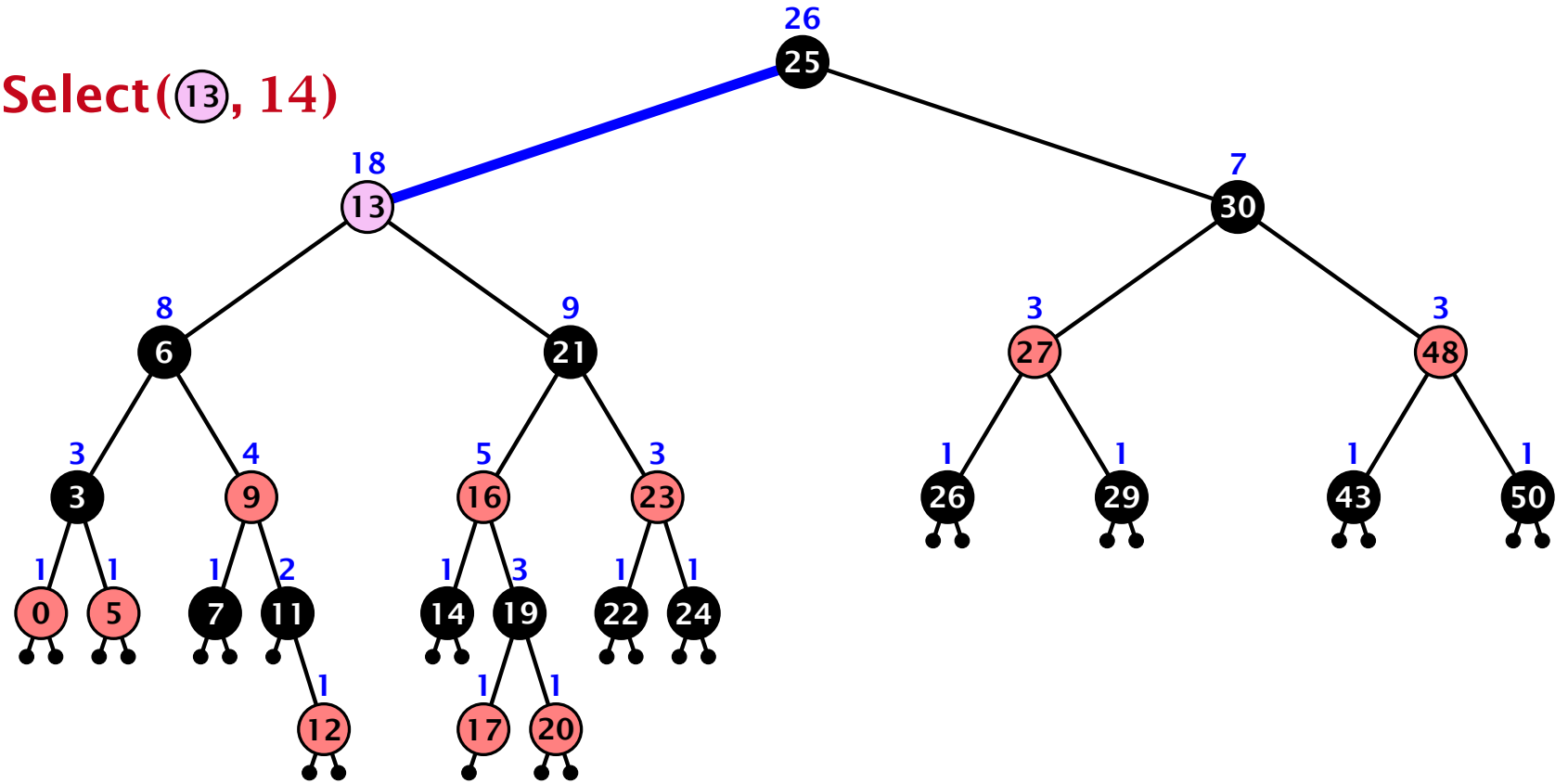


Find-by-rank:

- decide whether you have to proceed into the left or right sub-tree
- adjust the rank that you are searching for if you go right

Select(x, i)

Select(13, 14)

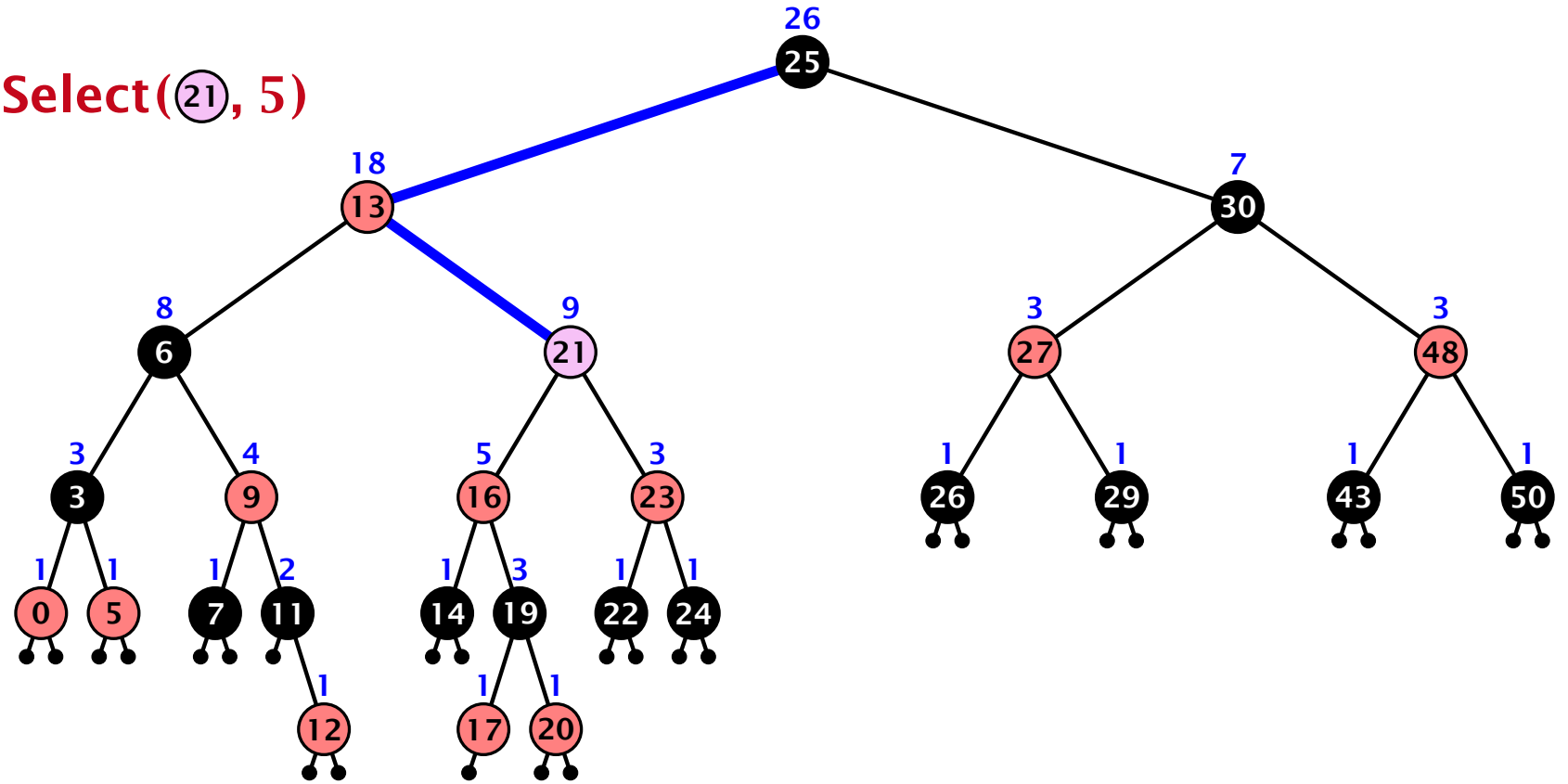


Find-by-rank:

- decide whether you have to proceed into the left or right sub-tree
- adjust the rank that you are searching for if you go right

Select(x, i)

Select(21, 5)

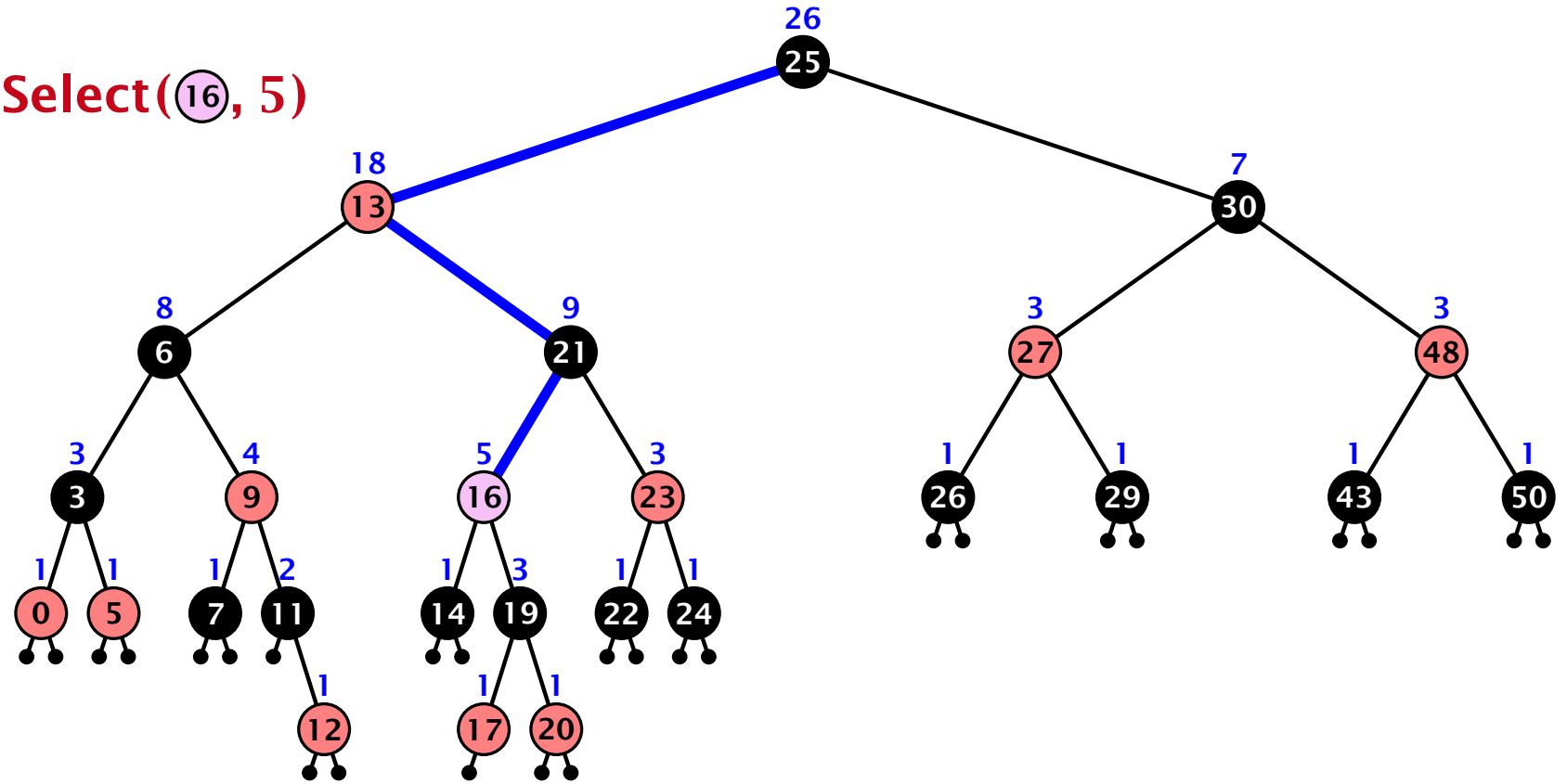


Find-by-rank:

- decide whether you have to proceed into the left or right sub-tree
- adjust the rank that you are searching for if you go right

Select(x, i)

Select(16, 5)

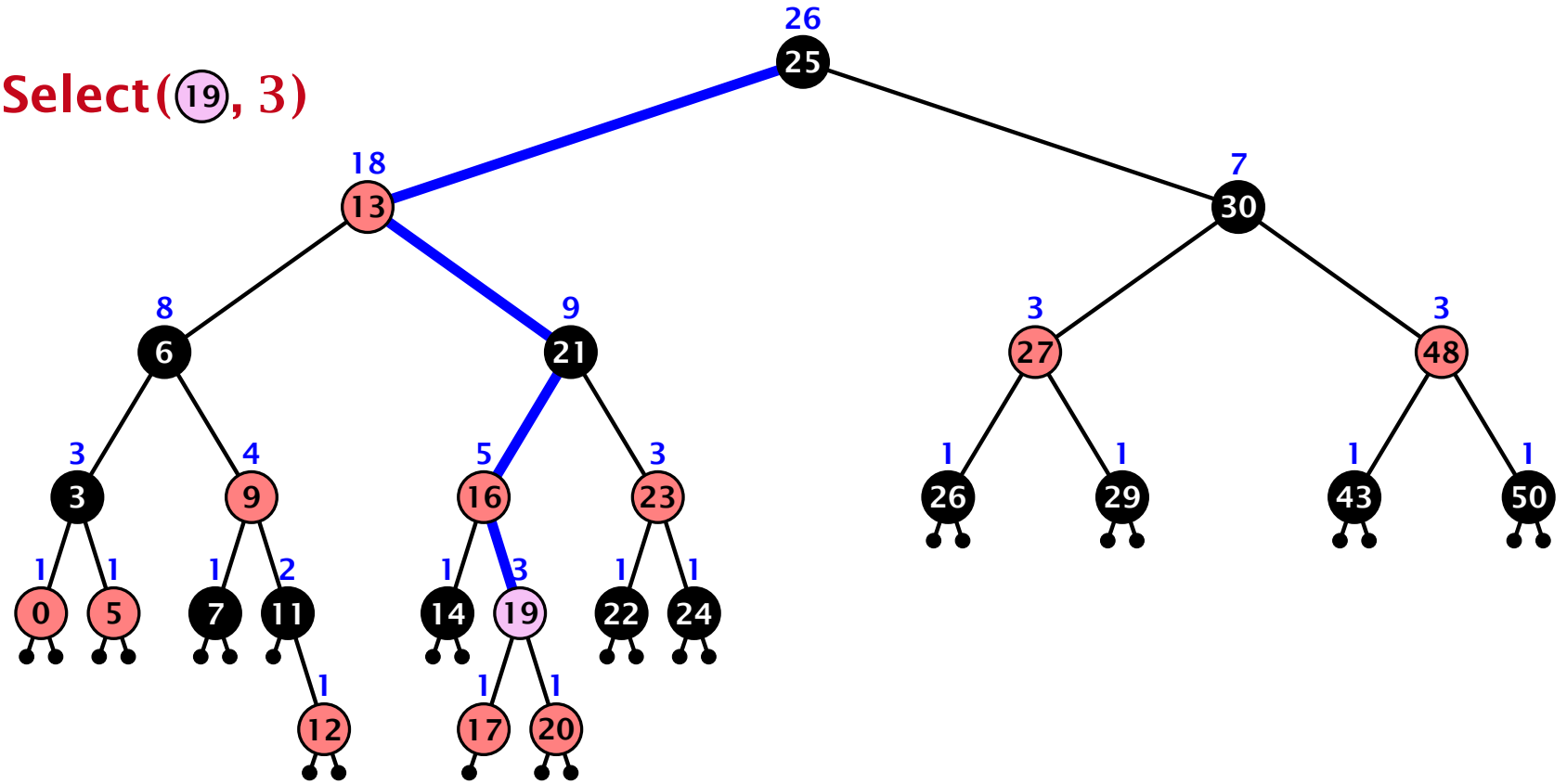


Find-by-rank:

- decide whether you have to proceed into the left or right sub-tree
- adjust the rank that you are searching for if you go right

Select(x, i)

Select(19, 3)

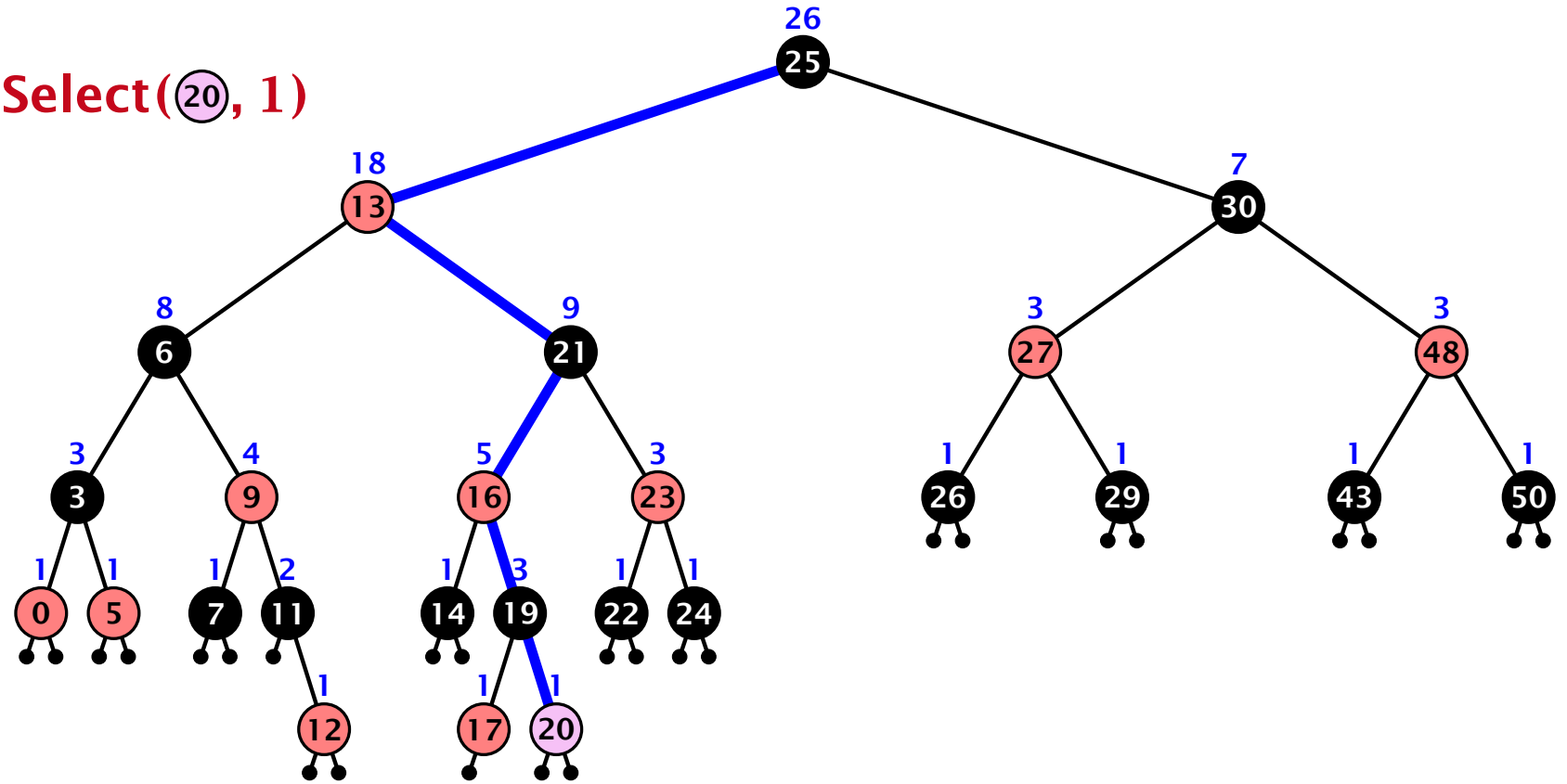


Find-by-rank:

- decide whether you have to proceed into the left or right sub-tree
- adjust the rank that you are searching for if you go right

Select(x, i)

Select(20, 1)



Find-by-rank:

- decide whether you have to proceed into the left or right sub-tree
- adjust the rank that you are searching for if you go right

7.4 Augmenting Data Structures

Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

3. How do we maintain information?

Search(k): Nothing to do.

Insert(x): When going down the search path increase the size field for each visited node. Maintain the size field during rotations.

Delete(x): Directly after splicing out a node traverse the path from the spliced out node upwards, and decrease the size counter on every node on this path. Maintain the size field during rotations.

7.4 Augmenting Data Structures

Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

3. How do we maintain information?

Search(k): Nothing to do.

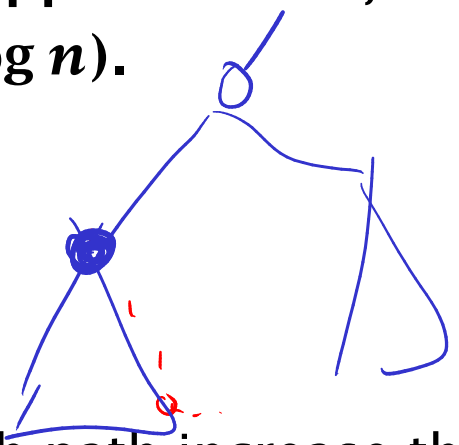
Insert(x): When going down the search path increase the size field for each visited node. Maintain the size field during rotations.

Delete(x): Directly after splicing out a node traverse the path from the spliced out node upwards, and decrease the size counter on every node on this path. Maintain the size field during rotations.

7.4 Augmenting Data Structures

Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

3. How do we maintain information?



Search(k): Nothing to do.

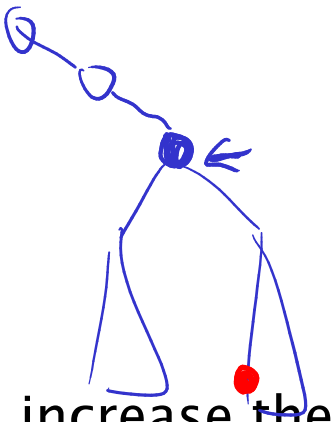
Insert(x): When going down the search path increase the size field for each visited node. Maintain the size field during rotations.

Delete(x): Directly after splicing out a node traverse the path from the spliced out node upwards, and decrease the size counter on every node on this path. Maintain the size field during rotations.

7.4 Augmenting Data Structures

Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

3. How do we maintain information?



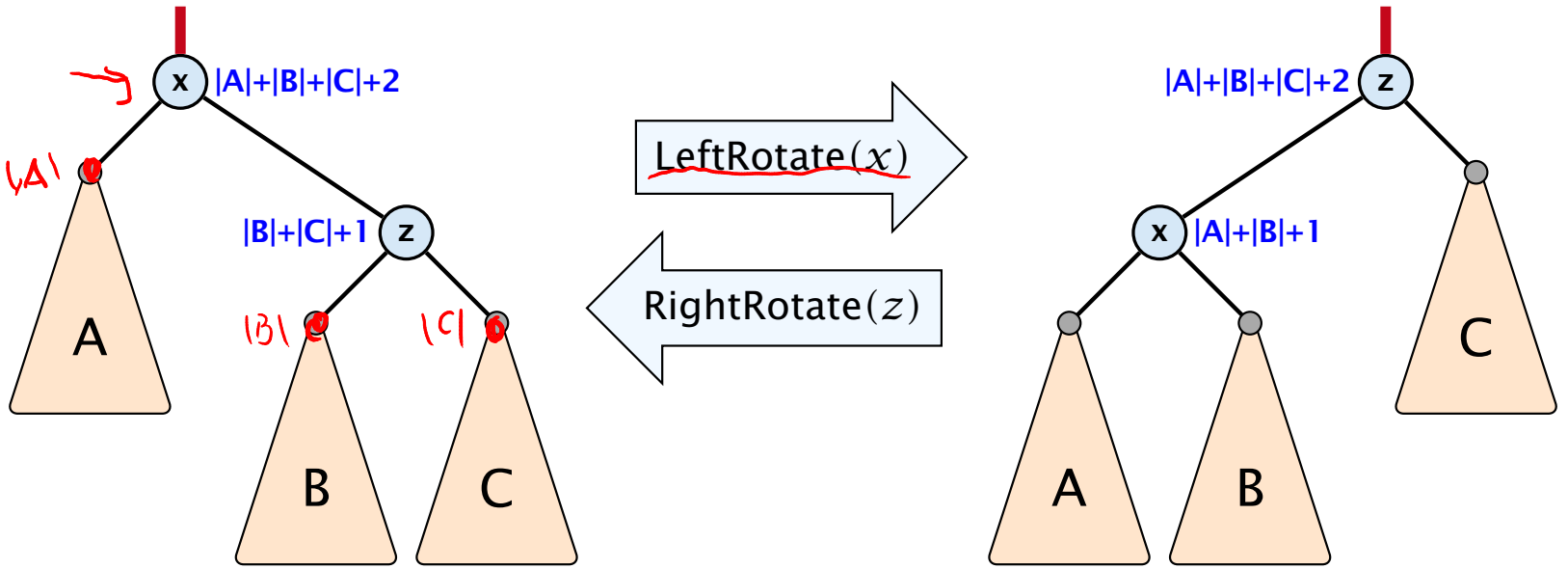
Search(k): Nothing to do.

Insert(x): When going down the search path increase the size field for each visited node. Maintain the size field during rotations.

Delete(x): Directly after splicing out a node traverse the path from the spliced out node upwards, and decrease the size counter on every node on this path. Maintain the size field during rotations.

Rotations

The only operation during the fix-up procedure that alters the tree and requires an update of the size-field:



The nodes x and z are the only nodes changing their size-fields.

The new size-fields can be computed locally from the size-fields of the children.

7.5 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$

7.5 Skip Lists

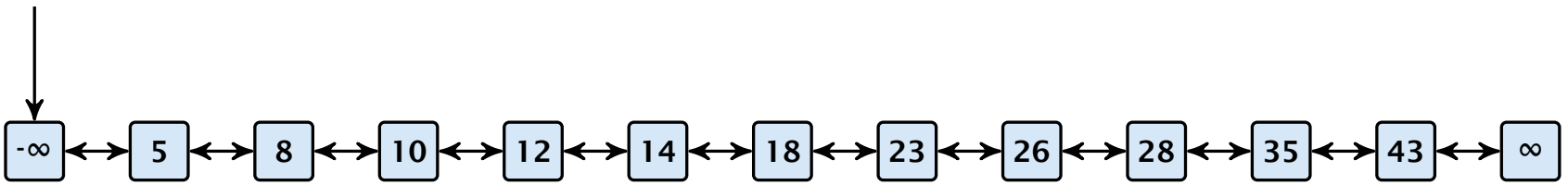
Why do we not use a list for implementing the ADT Dynamic Set?

- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$

7.5 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

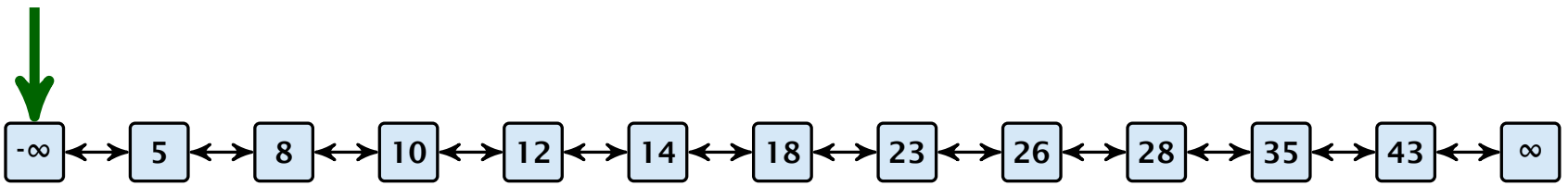
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$



7.5 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

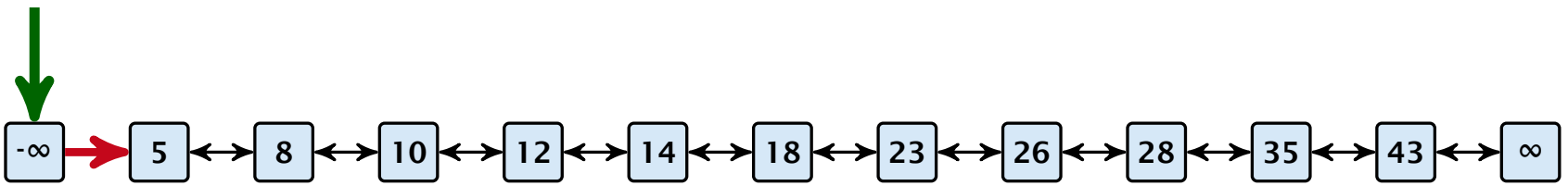
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$



7.5 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

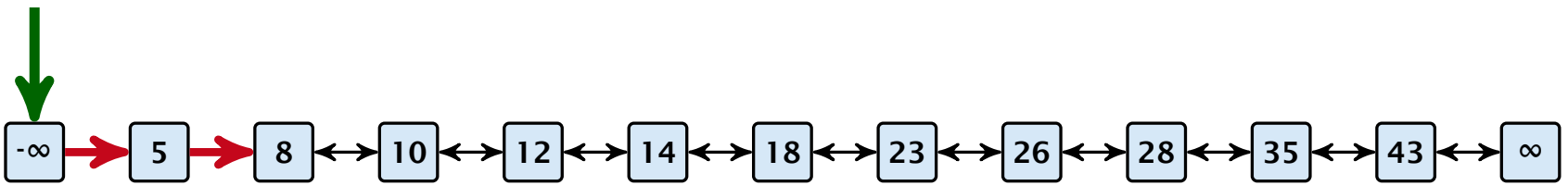
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$



7.5 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

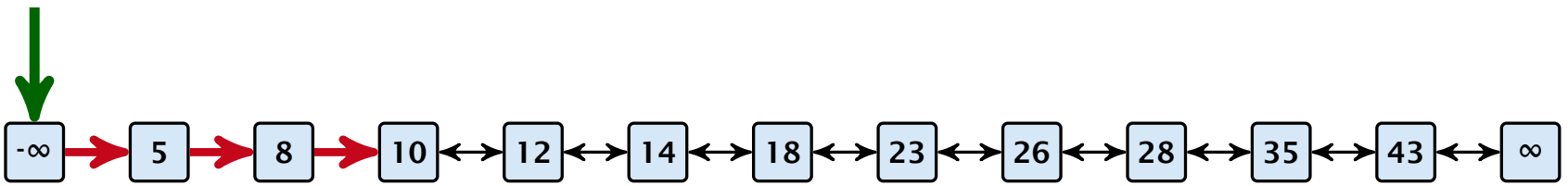
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$



7.5 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

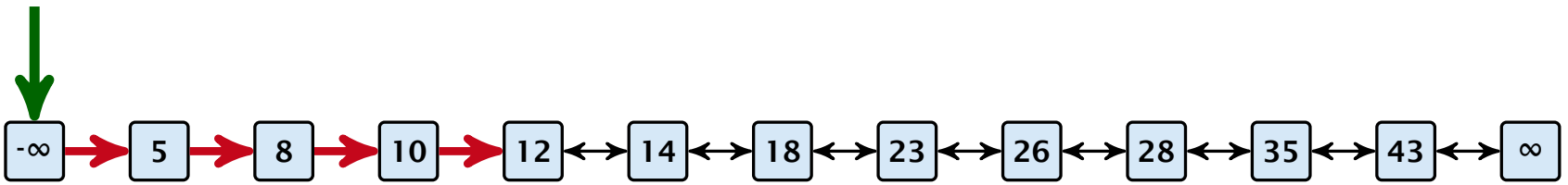
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$



7.5 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

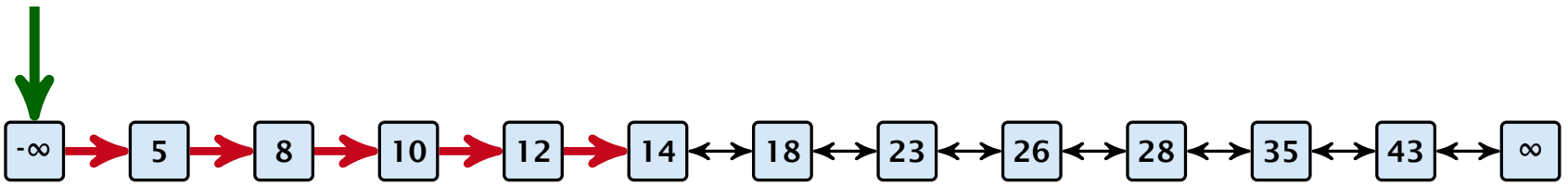
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$



7.5 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

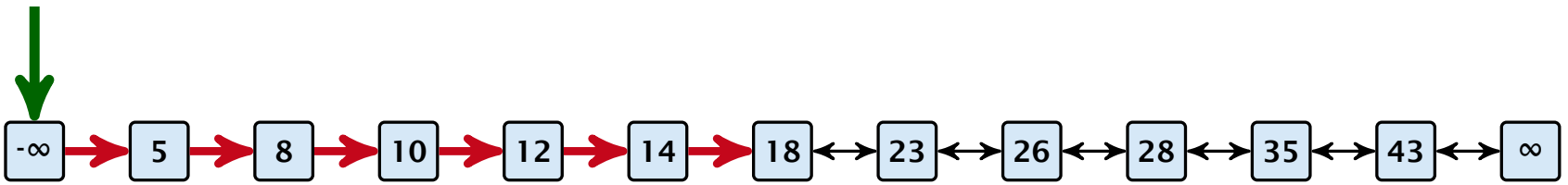
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$



7.5 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

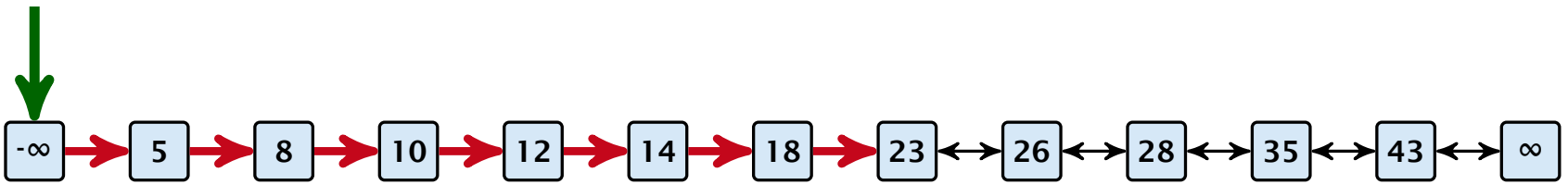
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$



7.5 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

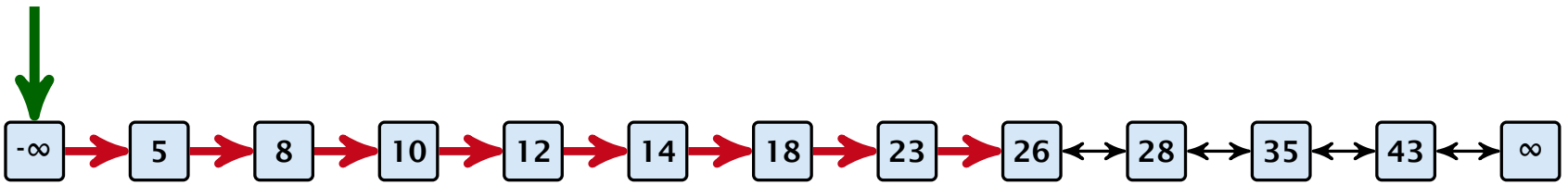
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$



7.5 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$



7.5 Skip Lists

How can we improve the search-operation?

7.5 Skip Lists

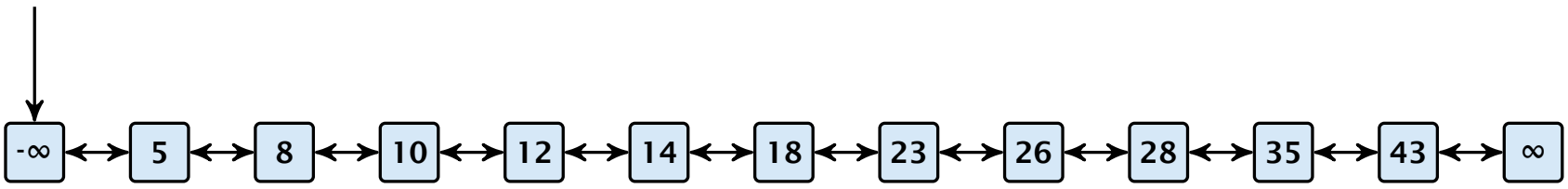
How can we improve the search-operation?

Add an express lane:

7.5 Skip Lists

How can we improve the search-operation?

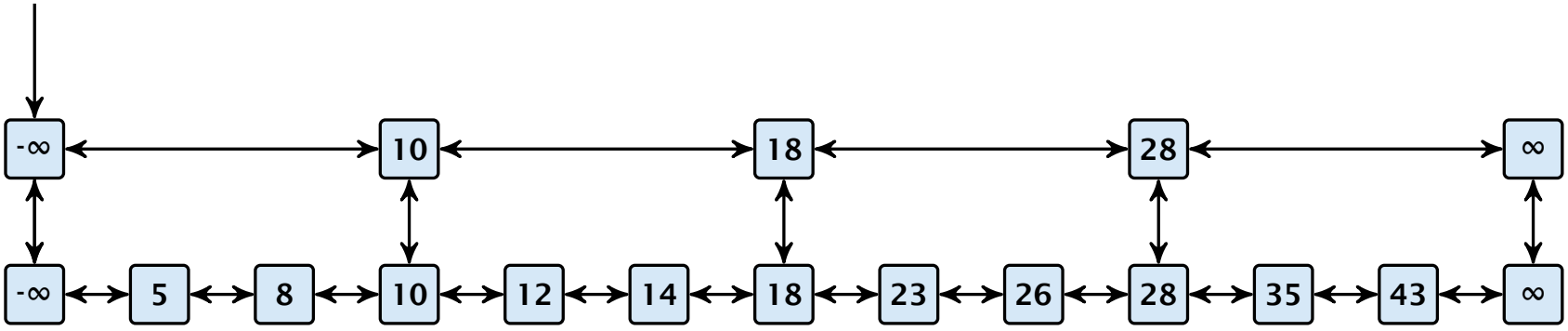
Add an express lane:



7.5 Skip Lists

How can we improve the search-operation?

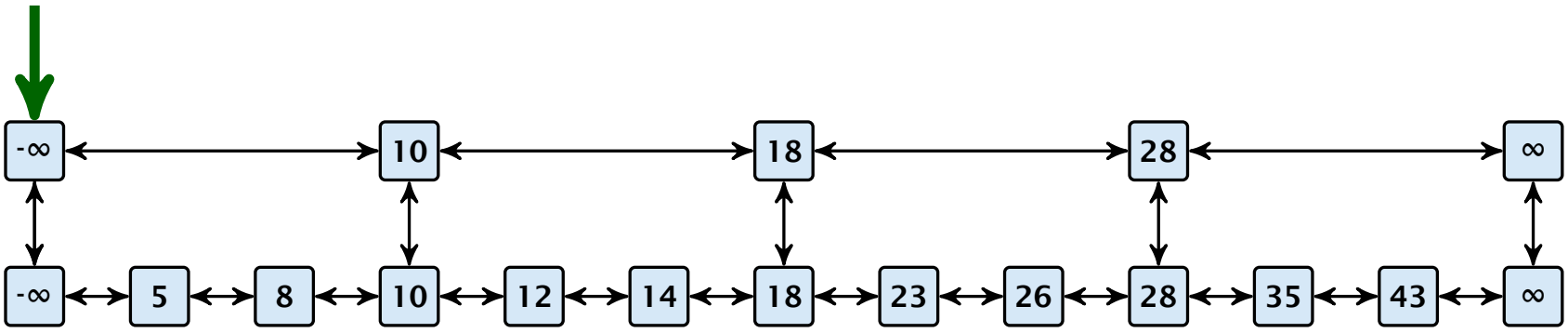
Add an express lane:



7.5 Skip Lists

How can we improve the search-operation?

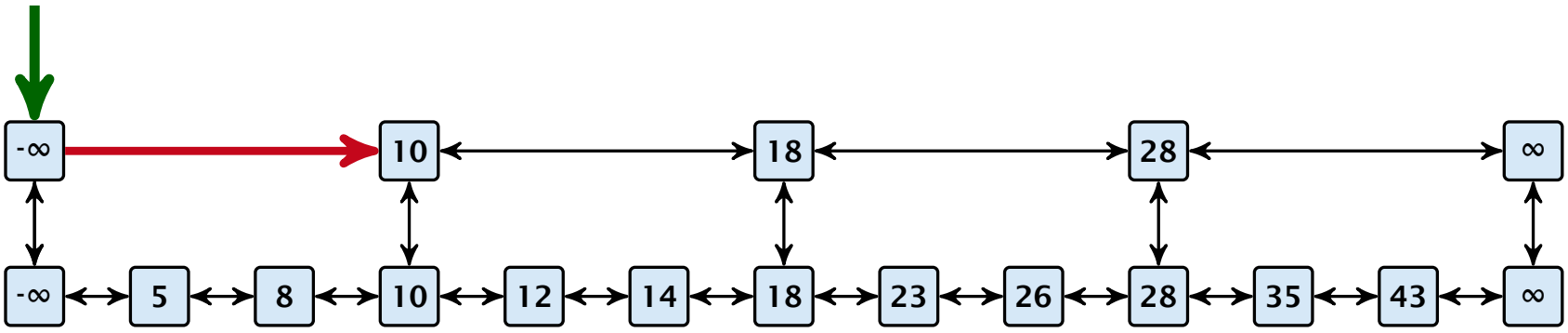
Add an express lane:



7.5 Skip Lists

How can we improve the search-operation?

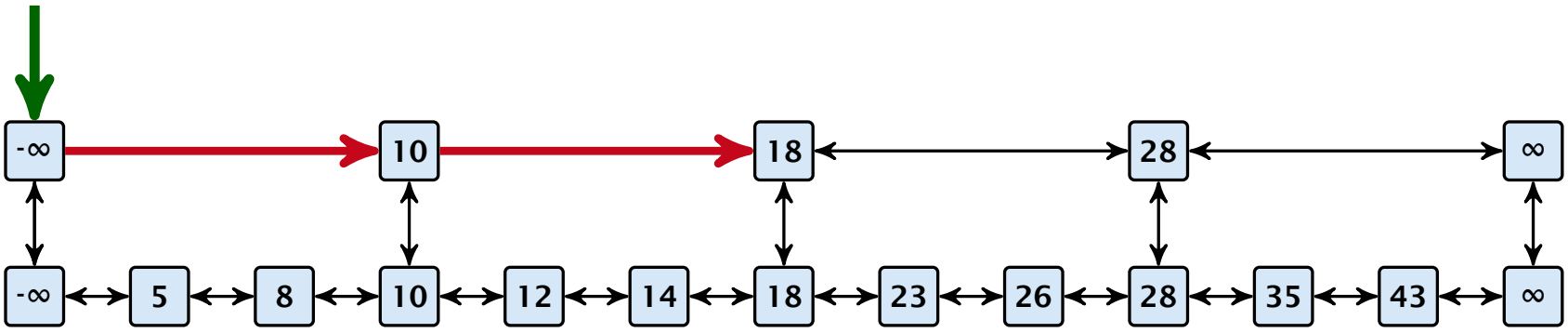
Add an express lane:



7.5 Skip Lists

How can we improve the search-operation?

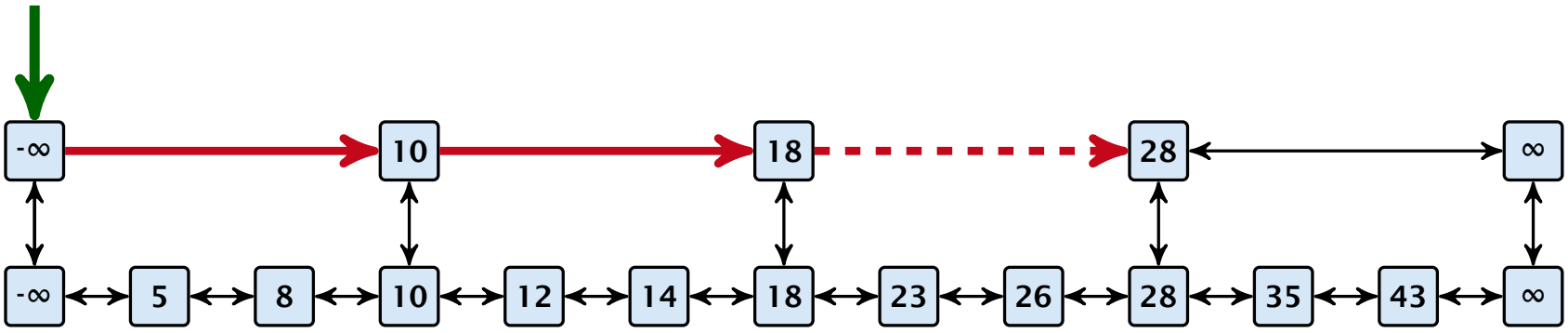
Add an express lane:



7.5 Skip Lists

How can we improve the search-operation?

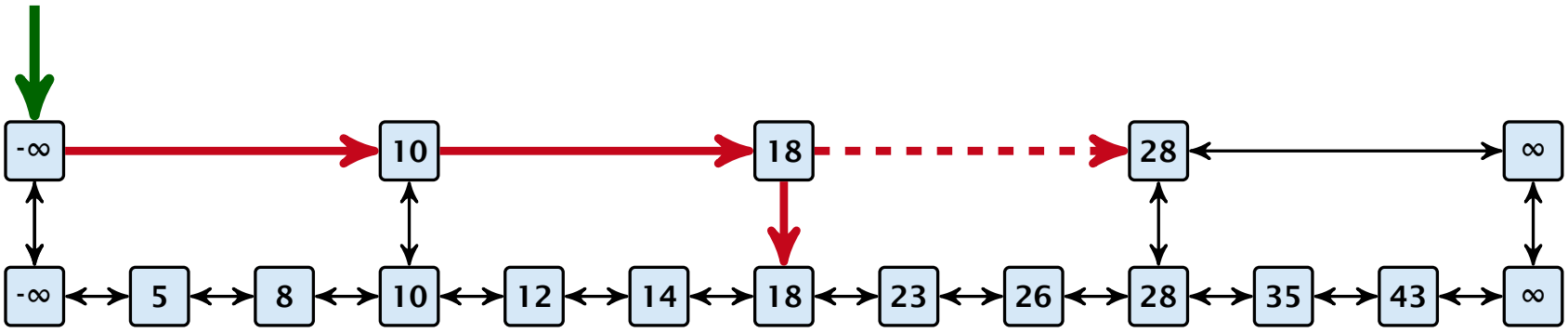
Add an express lane:



7.5 Skip Lists

How can we improve the search-operation?

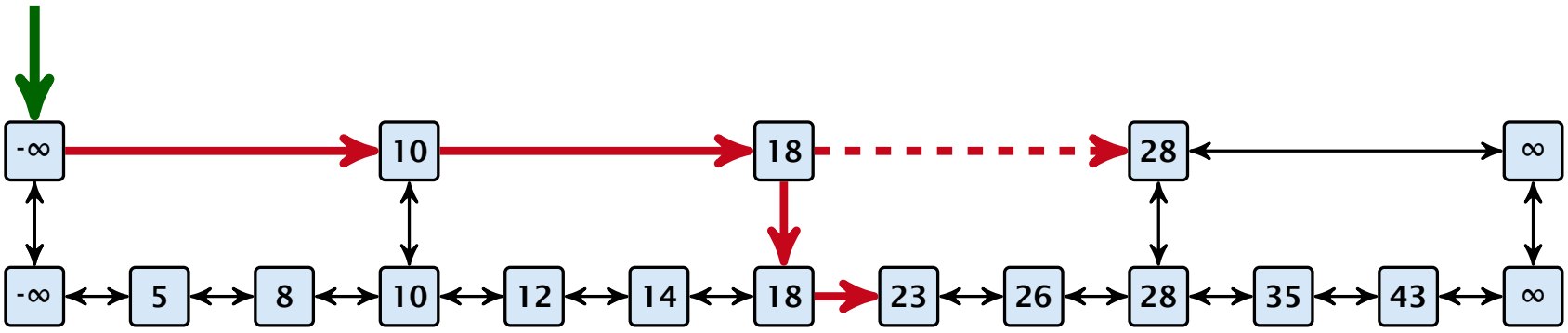
Add an express lane:



7.5 Skip Lists

How can we improve the search-operation?

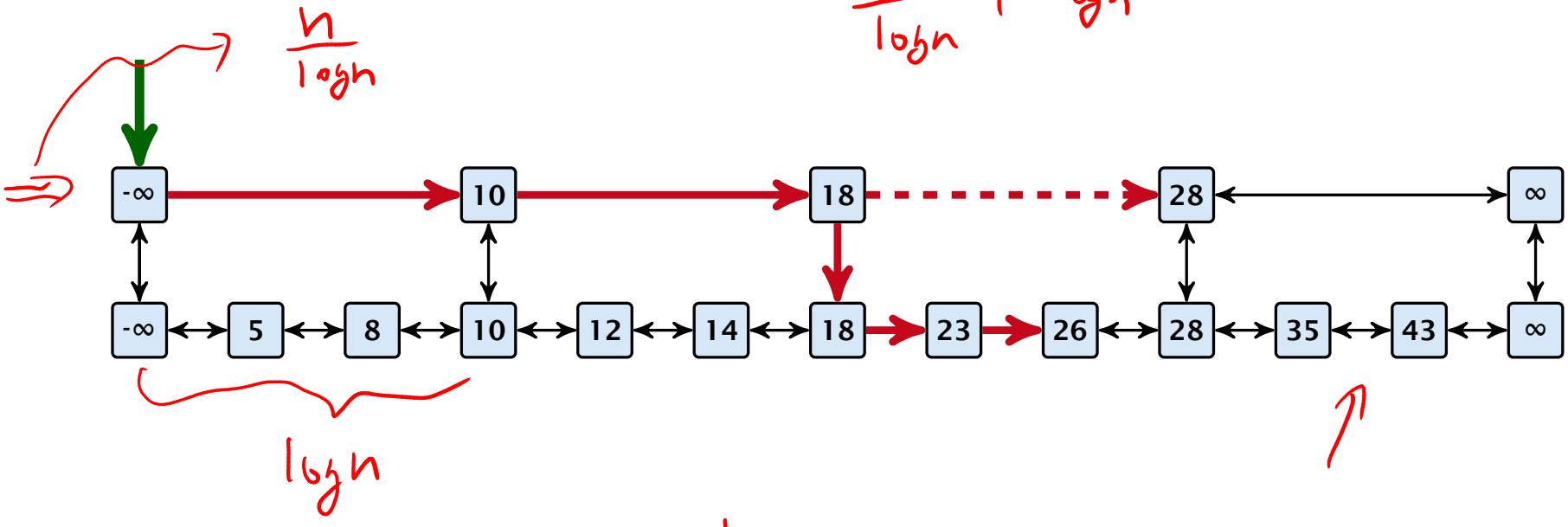
Add an express lane:



7.5 Skip Lists

How can we improve the search-operation?

Add an express lane:



$\frac{n}{\log n}$

$\frac{n}{\log n} + \log n$

$\log n$

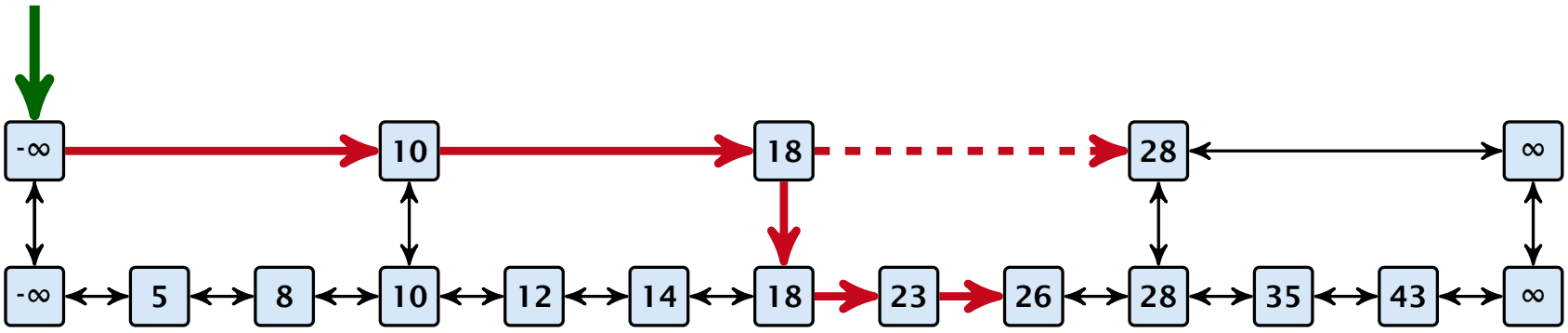
$\frac{n}{s} + s \leq 2\sqrt{n}$

$s = \sqrt{n}$

7.5 Skip Lists

How can we improve the search-operation?

Add an express lane:

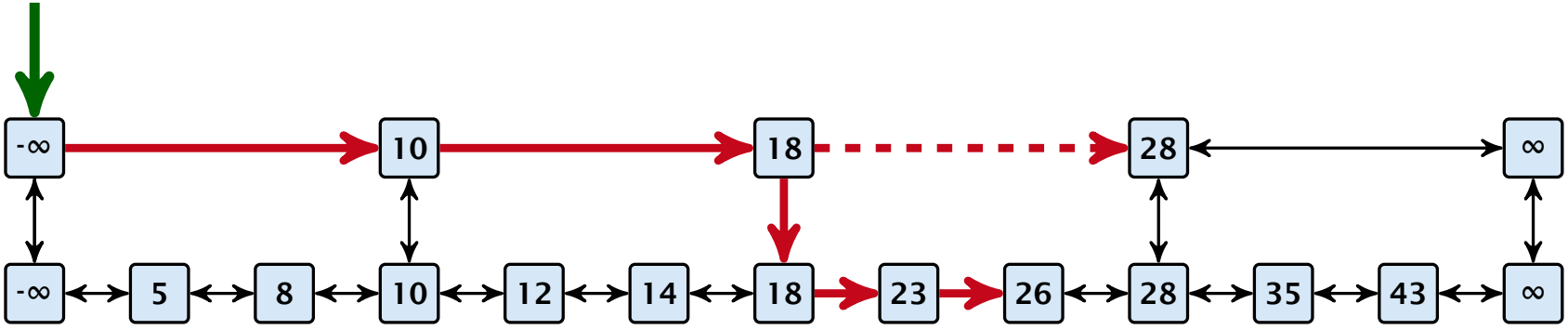


Let $|L_1|$ denote the number of elements in the "express lane", and $|L_0| = n$ the number of all elements (ignoring dummy elements).

7.5 Skip Lists

How can we improve the search-operation?

Add an express lane:



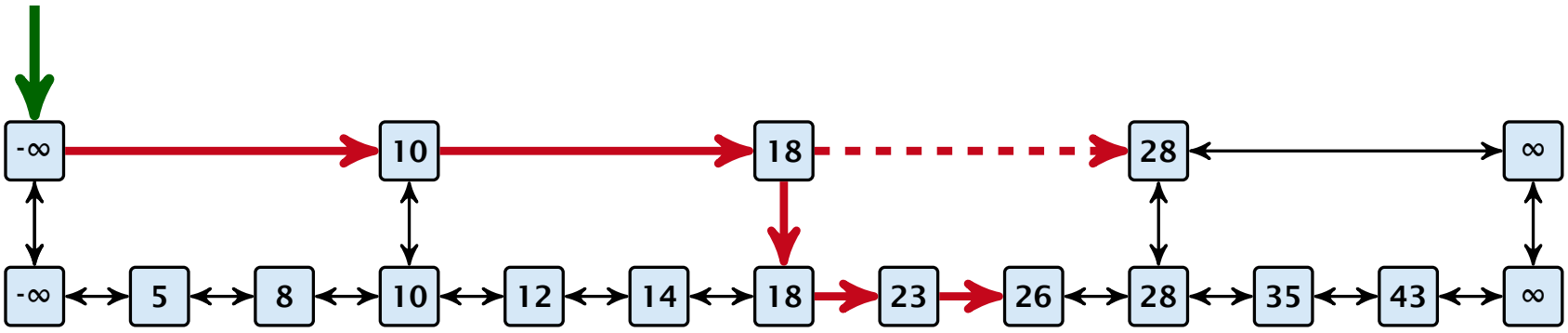
Let $|L_1|$ denote the number of elements in the "express lane", and $|L_0| = n$ the number of all elements (ignoring dummy elements).

Worst case search time: $|L_1| + \frac{|L_0|}{|L_1|}$ (ignoring additive constants)

7.5 Skip Lists

How can we improve the search-operation?

Add an express lane:



Let $|L_1|$ denote the number of elements in the "express lane", and $|L_0| = n$ the number of all elements (ignoring dummy elements).

Worst case search time: $|L_1| + \frac{|L_0|}{|L_1|}$ (ignoring additive constants)

Choose $|L_1| = \sqrt{n}$. Then search time $\Theta(\sqrt{n})$.

7.5 Skip Lists

Add more express lanes. Lane L_i contains roughly every $\frac{L_{i-1}}{L_i}$-th item from list L_{i-1}.

7.5 Skip Lists

Add more express lanes. Lane L_i contains roughly every $\frac{L_{i-1}}{L_i}$-th item from list L_{i-1}.

Search(x) ($k+1$ lists $L_0, \ldots, L_k$)

7.5 Skip Lists

Add more express lanes. Lane L_i contains roughly every $\frac{L_{i-1}}{L_i}$-th item from list L_{i-1}.

Search(x) ($k+1$ lists $L_0, \ldots, L_k$)

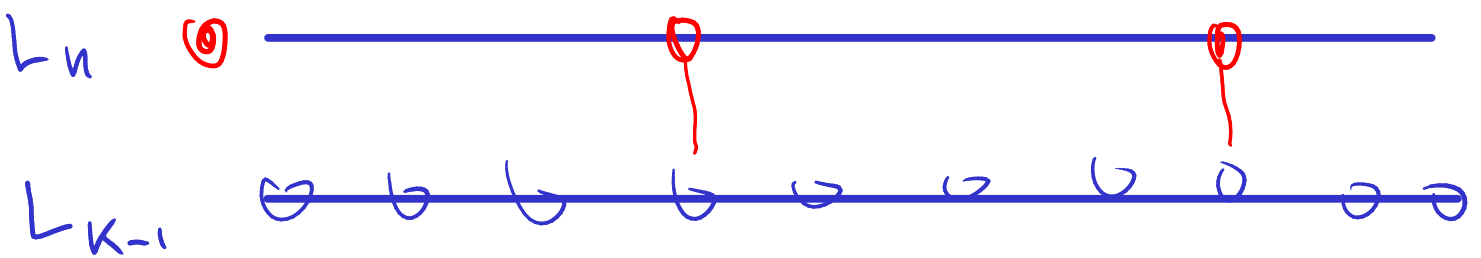
- Find the largest item in list L_k that is smaller than x. At most $|L_k| + 2$ steps.

7.5 Skip Lists

Add more express lanes. Lane L_i contains roughly every $\frac{L_{i-1}}{L_i}$-th item from list L_{i-1}.

Search(x) ($k+1$ lists $L_0, \ldots, L_k$)

- Find the largest item in list L_k that is smaller than x. At most $|L_k| + 2$ steps.
- Find the largest item in list L_{k-1} that is smaller than x. At most $\lceil \frac{|L_{k-1}|}{|L_k|+1} \rceil + 2$ steps.



7.5 Skip Lists

Add more express lanes. Lane L_i contains roughly every $\frac{L_{i-1}}{L_i}$-th item from list L_{i-1}.

Search(x) ($k+1$ lists $L_0, \ldots, L_k$)

- Find the largest item in list L_k that is smaller than x. At most $|L_k| + 2$ steps.
- Find the largest item in list L_{k-1} that is smaller than x. At most $\lceil \frac{|L_{k-1}|}{|L_k|+1} \rceil + 2$ steps.
- Find the largest item in list L_{k-2} that is smaller than x. At most $\lceil \frac{|L_{k-2}|}{|L_{k-1}|+1} \rceil + 2$ steps.

7.5 Skip Lists

Add more express lanes. Lane L_i contains roughly every $\frac{L_{i-1}}{L_i}$-th item from list L_{i-1}.

Search(x) ($k+1$ lists $L_0, \ldots, L_k$)

- Find the largest item in list L_k that is smaller than x. At most $|L_k| + 2$ steps.
- Find the largest item in list L_{k-1} that is smaller than x. At most $\lceil \frac{|L_{k-1}|}{|L_k|+1} \rceil + 2$ steps.
- Find the largest item in list L_{k-2} that is smaller than x. At most $\lceil \frac{|L_{k-2}|}{|L_{k-1}|+1} \rceil + 2$ steps.
- ...

7.5 Skip Lists

Add more express lanes. Lane L_i contains roughly every $\frac{L_{i-1}}{L_i}$-th item from list L_{i-1}.

Search(x) ($k+1$ lists $L_0, \ldots, L_k$)

- Find the largest item in list L_k that is smaller than x. At most $|L_k| + 2$ steps.
- Find the largest item in list L_{k-1} that is smaller than x. At most $\lceil \frac{|L_{k-1}|}{|L_k|+1} \rceil + 2$ steps. $\leq \frac{|L_{k-1}|}{|L_k|} + 3$
- Find the largest item in list L_{k-2} that is smaller than x. At most $\lceil \frac{|L_{k-2}|}{|L_{k-1}|+1} \rceil + 2$ steps.
- ...
- At most $|L_k| + \sum_{i=1}^{k} \frac{L_{i-1}}{L_i} + 3(k+1)$ steps.

7.5 Skip Lists

Choose ratios between list-lengths evenly, i.e., $\frac{|L_{i-1}|}{|L_i|} = r$, and, hence, $L_k \approx r^{-k} n$.

7.5 Skip Lists

Choose ratios between list-lengths evenly, i.e., $\frac{|L_{i-1}|}{|L_i|} = r$, and, hence, $L_k \approx r^{-k}n$.

Worst case running time is: $\mathcal{O}(r^{-k}n + kr)$.

7.5 Skip Lists

Choose ratios between list-lengths evenly, i.e., $\frac{|L_{i-1}|}{|L_i|} = r$, and, hence, $L_k \approx r^{-k}n$.

Worst case running time is: $\mathcal{O}(r^{-k}n + kr)$.
Choose $r = n^{\frac{1}{k+1}}$. Then

$$r^{-k}n + kr$$

7.5 Skip Lists

Choose ratios between list-lengths evenly, i.e., $\frac{|L_{i-1}|}{|L_i|} = r$, and, hence, $L_k \approx r^{-k}n$.

Worst case running time is: $\mathcal{O}(r^{-k}n + kr)$.
Choose $r = n^{\frac{1}{k+1}}$. Then

$$r^{-k}n + kr = \left(n^{\frac{1}{k+1}}\right)^{-k} n + kn^{\frac{1}{k+1}}$$

7.5 Skip Lists

Choose ratios between list-lengths evenly, i.e., $\frac{|L_{i-1}|}{|L_i|} = r$, and, hence, $L_k \approx r^{-k}n$.

Worst case running time is: $\mathcal{O}(r^{-k}n + kr)$.
Choose $r = n^{\frac{1}{k+1}}$. Then

$$\begin{aligned} r^{-k}n + kr &= \left(n^{\frac{1}{k+1}}\right)^{-k} n + kn^{\frac{1}{k+1}} \\ &= n^{1-\frac{k}{k+1}} + kn^{\frac{1}{k+1}} \end{aligned}$$

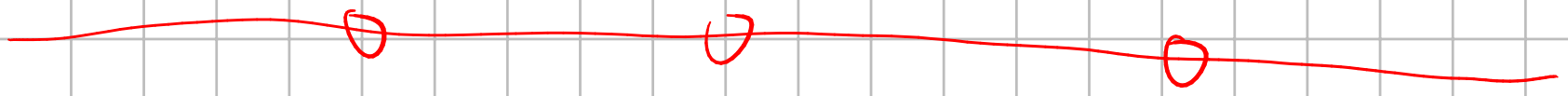
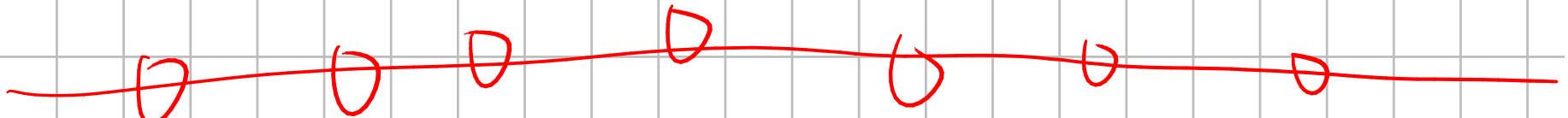
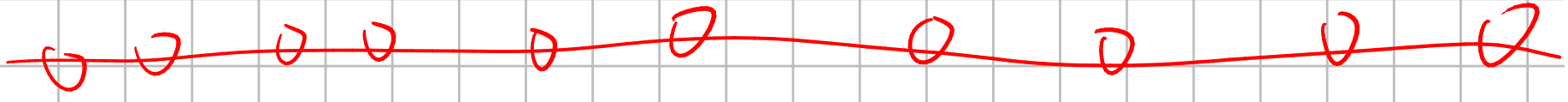
7.5 Skip Lists

Choose ratios between list-lengths evenly, i.e., $\frac{|L_{i-1}|}{|L_i|} = r$, and, hence, $L_k \approx r^{-k} n$.

Worst case running time is: $\mathcal{O}(r^{-k} n + kr)$.
Choose $r = n^{\frac{1}{k+1}}$. Then

$$\begin{aligned} r^{-k} n + kr &= \left(n^{\frac{1}{k+1}}\right)^{-k} n + k n^{\frac{1}{k+1}} \\ &= n^{1-\frac{k}{k+1}} + k n^{\frac{1}{k+1}} \\ &= (k+1) n^{\frac{1}{k+1}}. \end{aligned}$$

$$n^{\frac{1}{\log_2 n}} = \left[2^{\log_2 n}\right]^{\frac{1}{\log_2 n}}$$

7.5 Skip Lists

Choose ratios between list-lengths evenly, i.e., $\frac{|L_{i-1}|}{|L_i|} = r$, and, hence, $L_k \approx r^{-k} n$.

Worst case running time is: $\mathcal{O}(r^{-k} n + kr)$.
Choose $r = n^{\frac{1}{k+1}}$. Then

$$\begin{aligned} r^{-k} n + kr &= \left(n^{\frac{1}{k+1}}\right)^{-k} n + k n^{\frac{1}{k+1}} \\ &= n^{1-\frac{k}{k+1}} + k n^{\frac{1}{k+1}} \\ &= (k+1) n^{\frac{1}{k+1}} \quad . \end{aligned}$$

Choosing $k = \Theta(\log n)$ gives a logarithmic running time.

7.5 Skip Lists

How to do insert and delete?

- If we want that in L_i we always skip over roughly the same number of elements in L_{i-1}, an insert or delete may require a lot of re-organisation.

Use randomization instead!

7.5 Skip Lists

How to do insert and delete?

- If we want that in L_i we always skip over roughly the same number of elements in L_{i-1} an insert or delete may require a lot of re-organisation.

Use randomization instead!

7.5 Skip Lists

How to do insert and delete?

- If we want that in L_i we always skip over roughly the same number of elements in L_{i-1} an insert or delete may require a lot of re-organisation.

Use randomization instead!

7.5 Skip Lists

Insert:

- A search operation gives you the insert position for element x in every list.
- Flip a coin until it shows head, and record the number $t \in \{1, 2, \ldots\}$ of trials needed.
- Insert x into lists $L_0, \ldots, L_{t-1}$.

Delete:

- You get all predecessors via backward pointers.
- Delete x in all lists it actually appears in.

The time for both operations is dominated by the search time.

7.5 Skip Lists

Insert:

- A search operation gives you the insert position for element x in every list.
- Flip a coin until it shows head, and record the number $t \in \{1, 2, \ldots\}$ of trials needed.
- Insert x into lists $L_0, \ldots, L_{t-1}$.

Delete:

- You get all predecessors via backward pointers.
- Delete x in all lists it actually appears in.

The time for both operations is dominated by the search time.

7.5 Skip Lists

Insert:

- A search operation gives you the insert position for element x in every list.
- Flip a coin until it shows head, and record the number $t \in \{1, 2, \ldots\}$ of trials needed.
- Insert x into lists $L_0, \ldots, L_{t-1}$.

Delete:

- You get all predecessors via backward pointers.
- Delete x in all lists it actually appears in.

The time for both operations is dominated by the search time.

7.5 Skip Lists

Insert:

- A search operation gives you the insert position for element x in every list.
- Flip a coin until it shows head, and record the number $t \in \{1, 2, \ldots\}$ of trials needed.
- Insert x into lists $L_0, \ldots, L_{t-1}$.

Delete:

- You get all predecessors via backward pointers.
- Delete x in all lists it actually appears in.

The time for both operations is dominated by the search time.

7.5 Skip Lists

Insert:

- A search operation gives you the insert position for element x in every list.
- Flip a coin until it shows head, and record the number $t \in \{1, 2, \ldots\}$ of trials needed.
- Insert x into lists $L_0, \ldots, L_{t-1}$.

Delete:

- You get all predecessors via backward pointers.
- Delete x in all lists it actually appears in.

The time for both operations is dominated by the search time.

7.5 Skip Lists

Insert:

- A search operation gives you the insert position for element x in every list.
- Flip a coin until it shows head, and record the number $t \in \{1, 2, \ldots\}$ of trials needed.
- Insert x into lists $L_0, \ldots, L_{t-1}$.

Delete:

- You get all predecessors via backward pointers.
- Delete x in all lists it actually appears in.

The time for both operations is dominated by the search time.

7.5 Skip Lists

Insert:

- A search operation gives you the insert position for element x in every list.
- Flip a coin until it shows head, and record the number $t \in \{1, 2, \ldots\}$ of trials needed.
- Insert x into lists $L_0, \ldots, L_{t-1}$.

Delete:

- You get all predecessors via backward pointers.
- Delete x in all lists it actually appears in.

The time for both operations is dominated by the search time.

7.5 Skip Lists

Insert:

- A search operation gives you the insert position for element x in every list.
- Flip a coin until it shows head, and record the number $t \in \{1, 2, \ldots\}$ of trials needed.
- Insert x into lists $L_0, \ldots, L_{t-1}$.

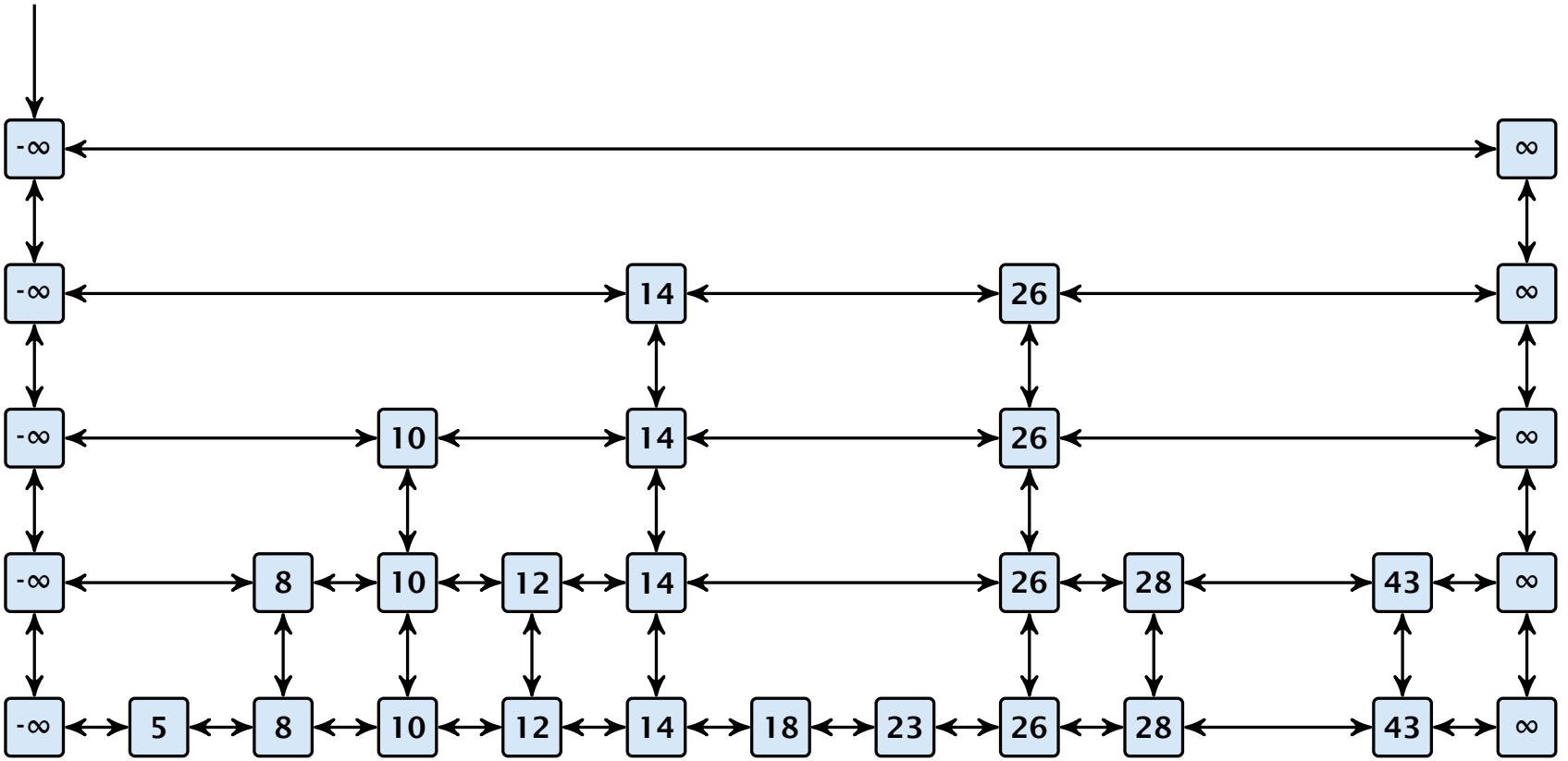
Delete:

- You get all predecessors via backward pointers.
- Delete x in all lists it actually appears in.

The time for both operations is dominated by the search time.

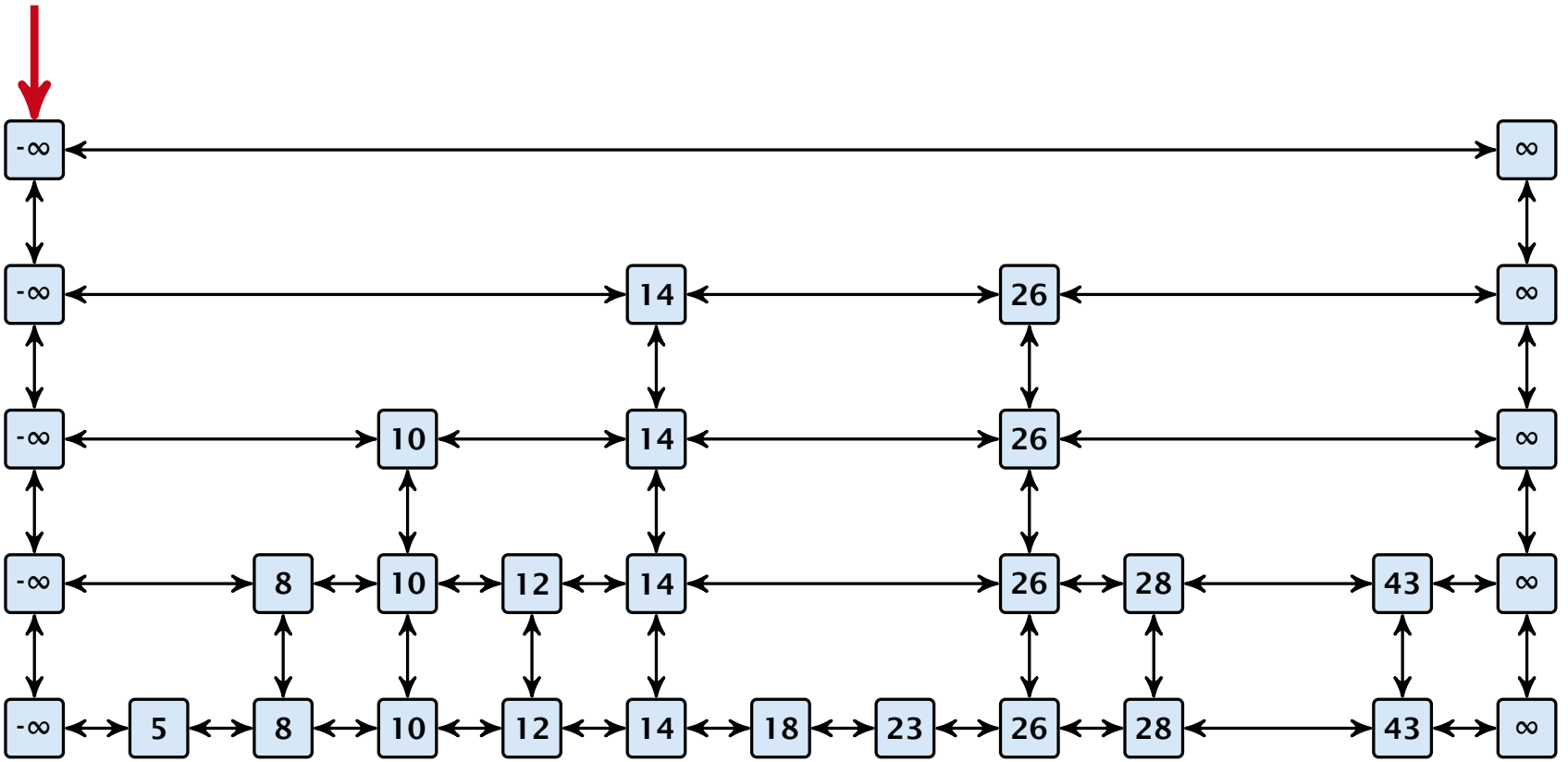
7.5 Skip Lists

Insert (35):



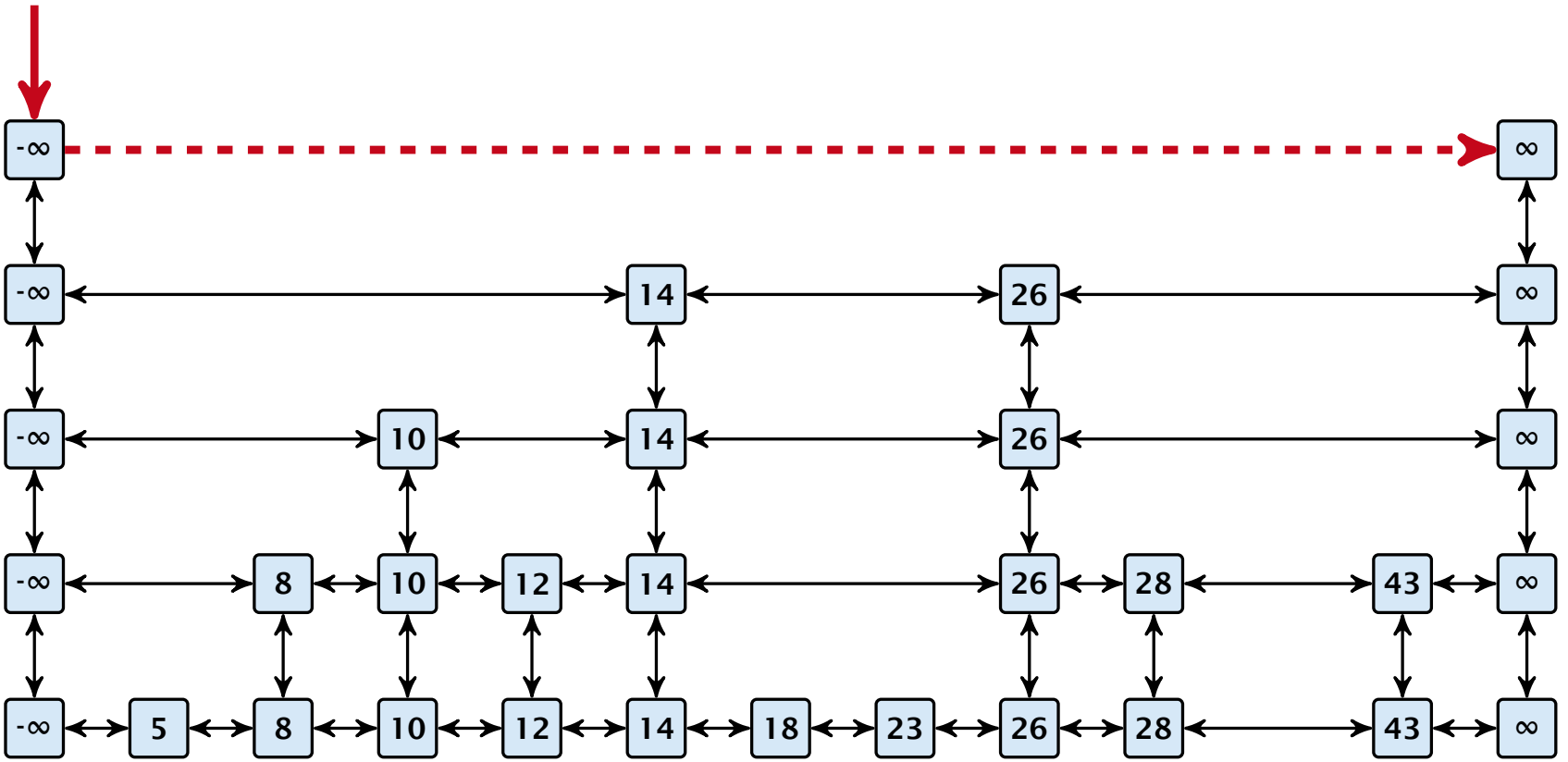
7.5 Skip Lists

Insert (35):



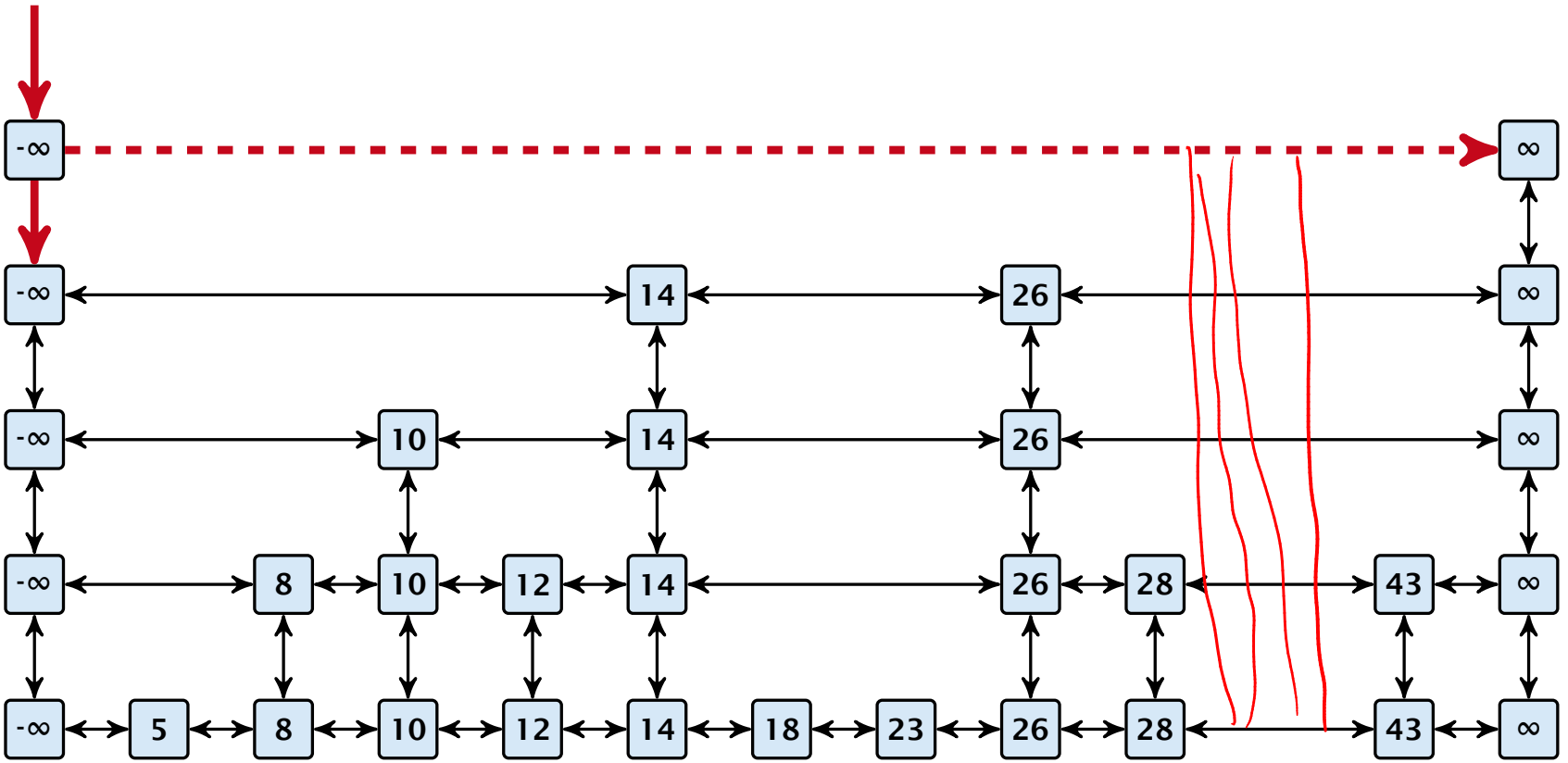
7.5 Skip Lists

Insert (35):



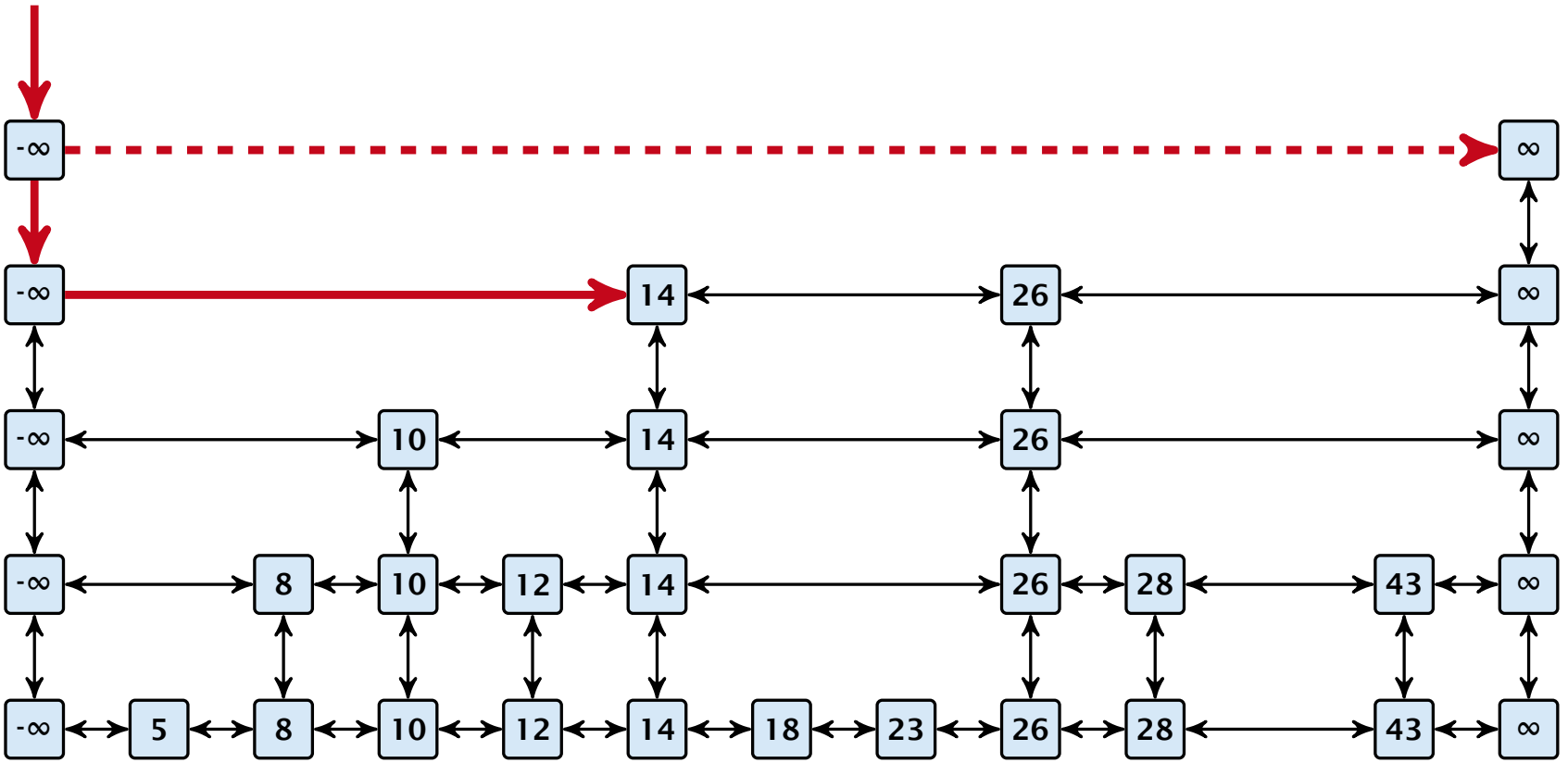
7.5 Skip Lists

Insert (35):



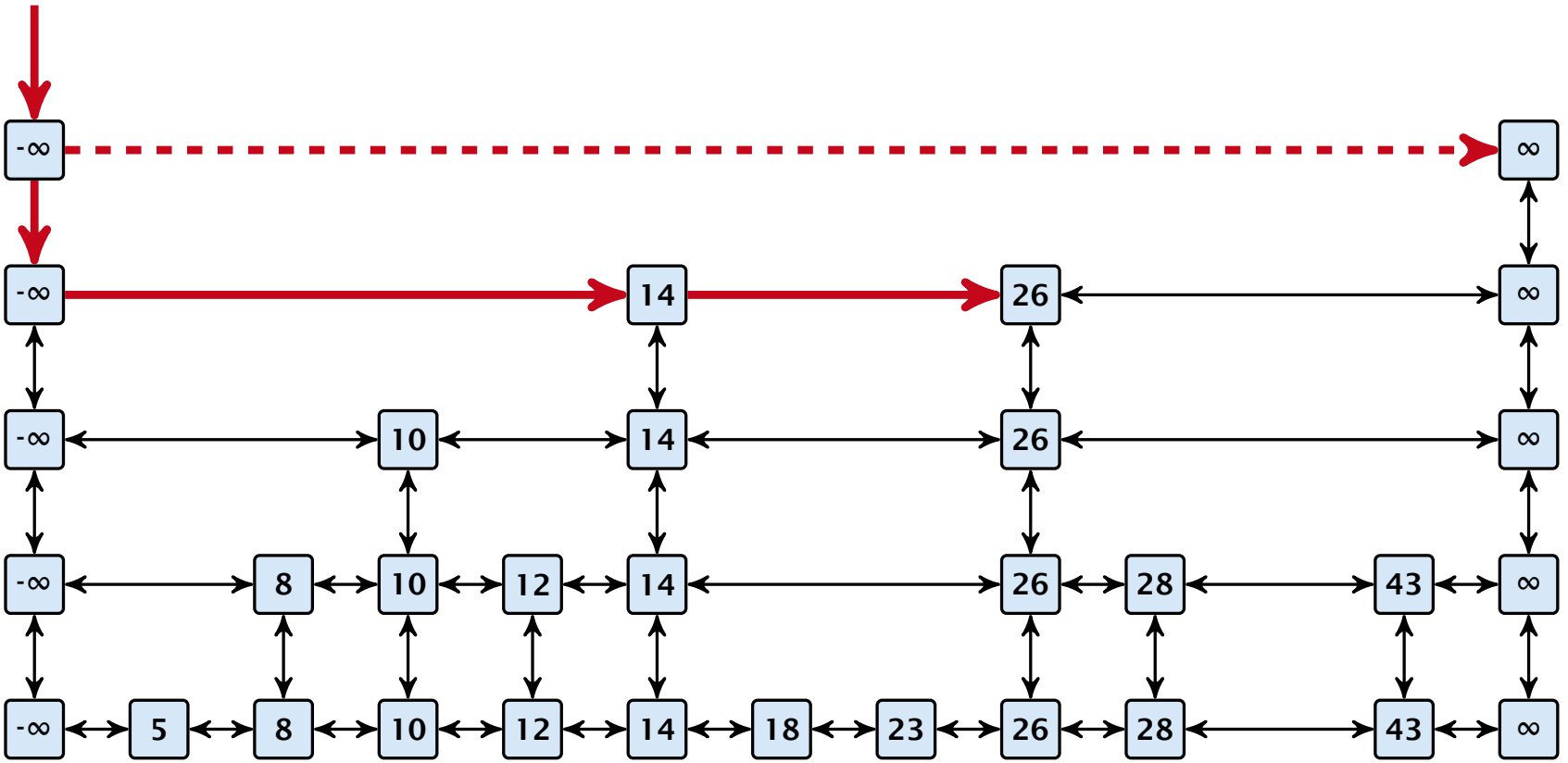
7.5 Skip Lists

Insert (35):



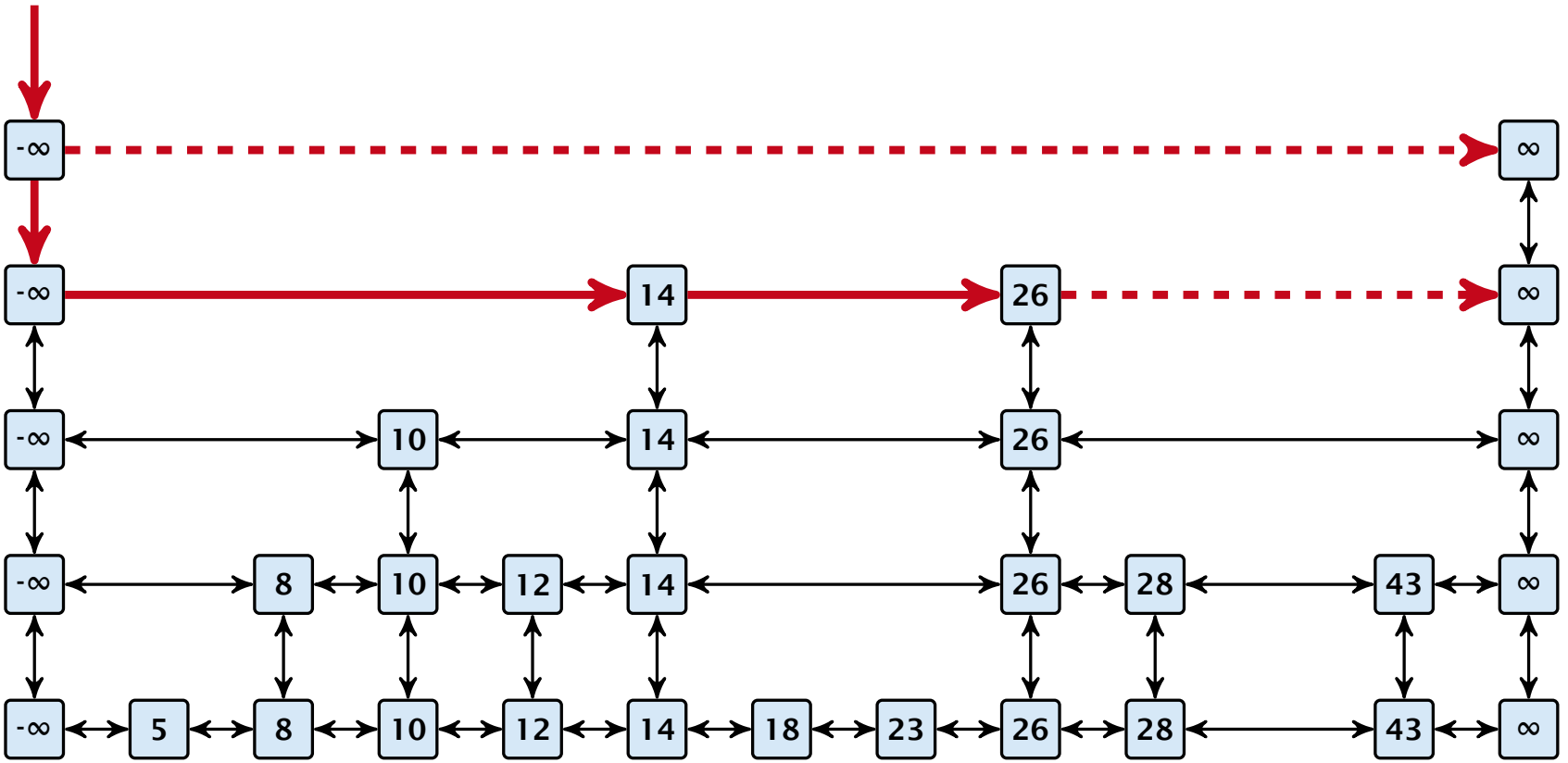
7.5 Skip Lists

Insert (35):



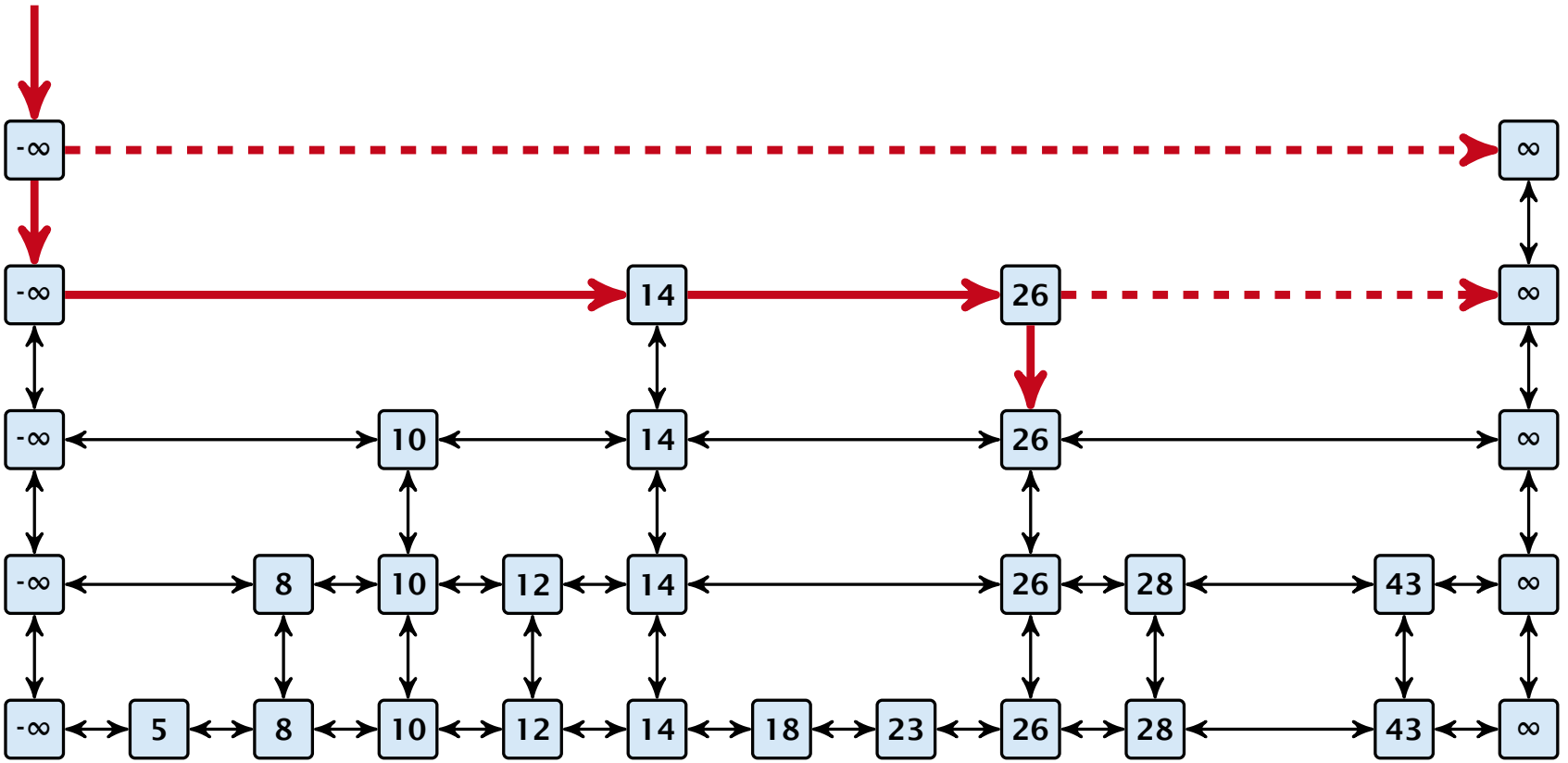
7.5 Skip Lists

Insert (35):



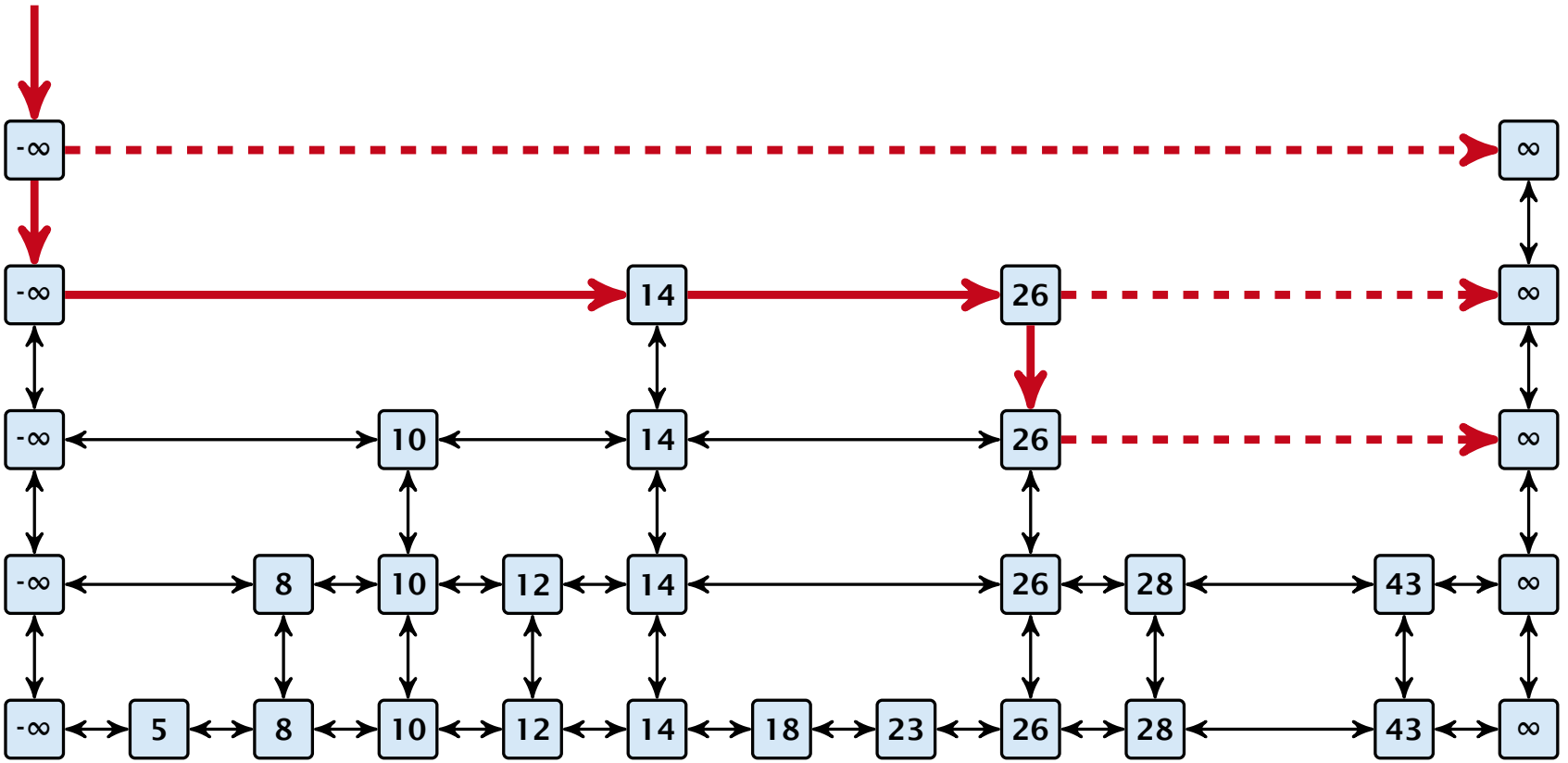
7.5 Skip Lists

Insert (35):



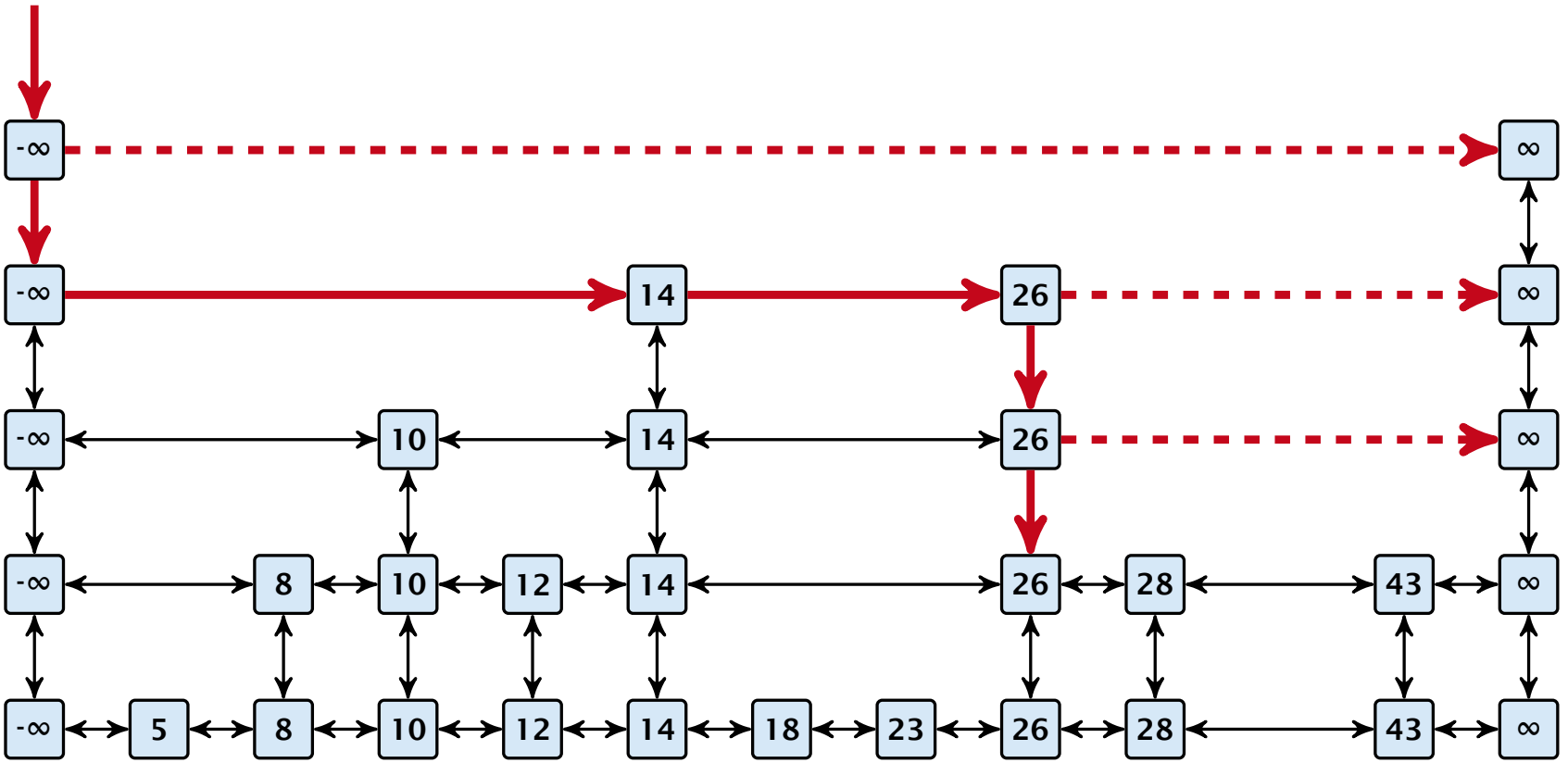
7.5 Skip Lists

Insert (35):



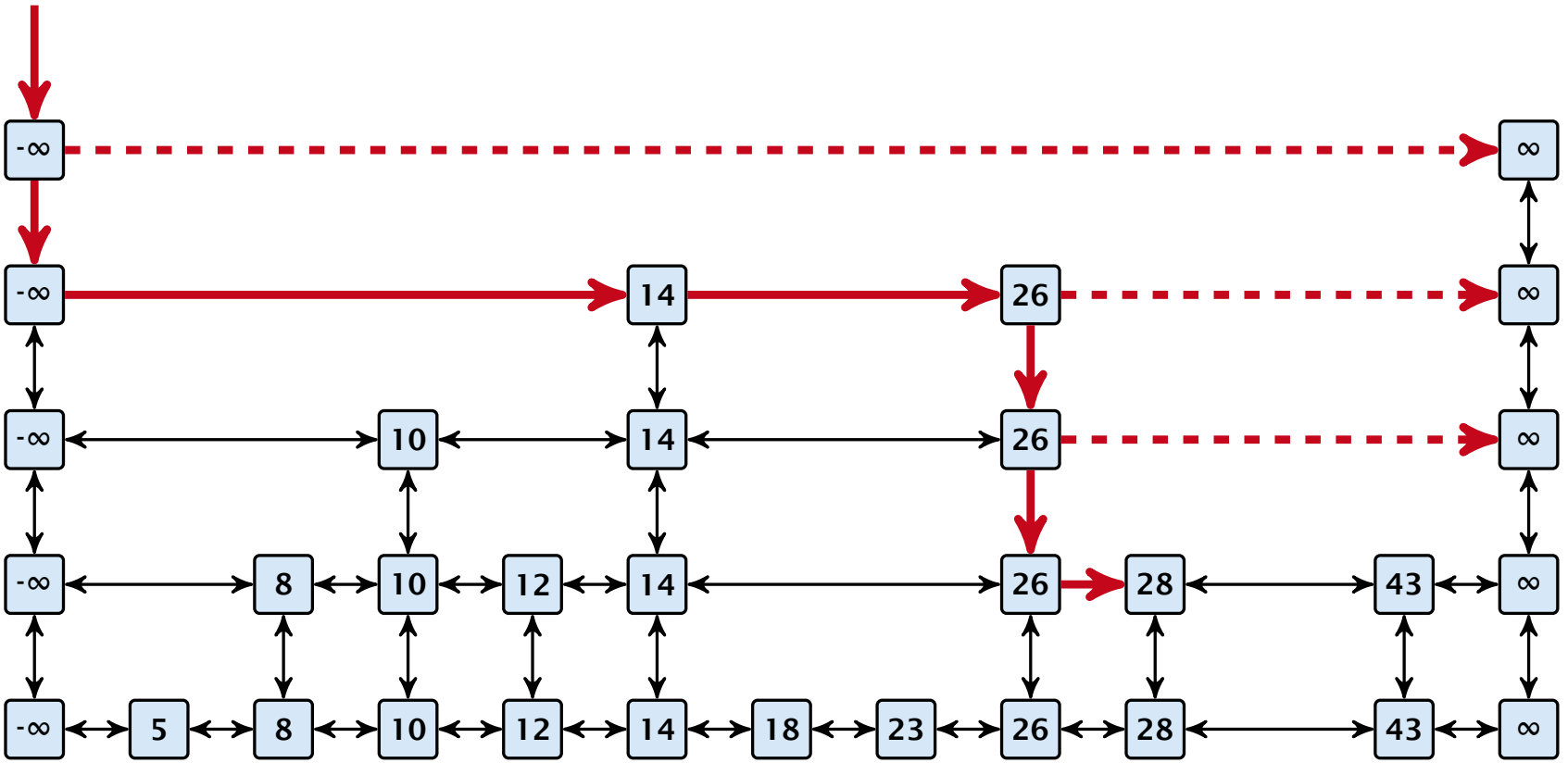
7.5 Skip Lists

Insert (35):



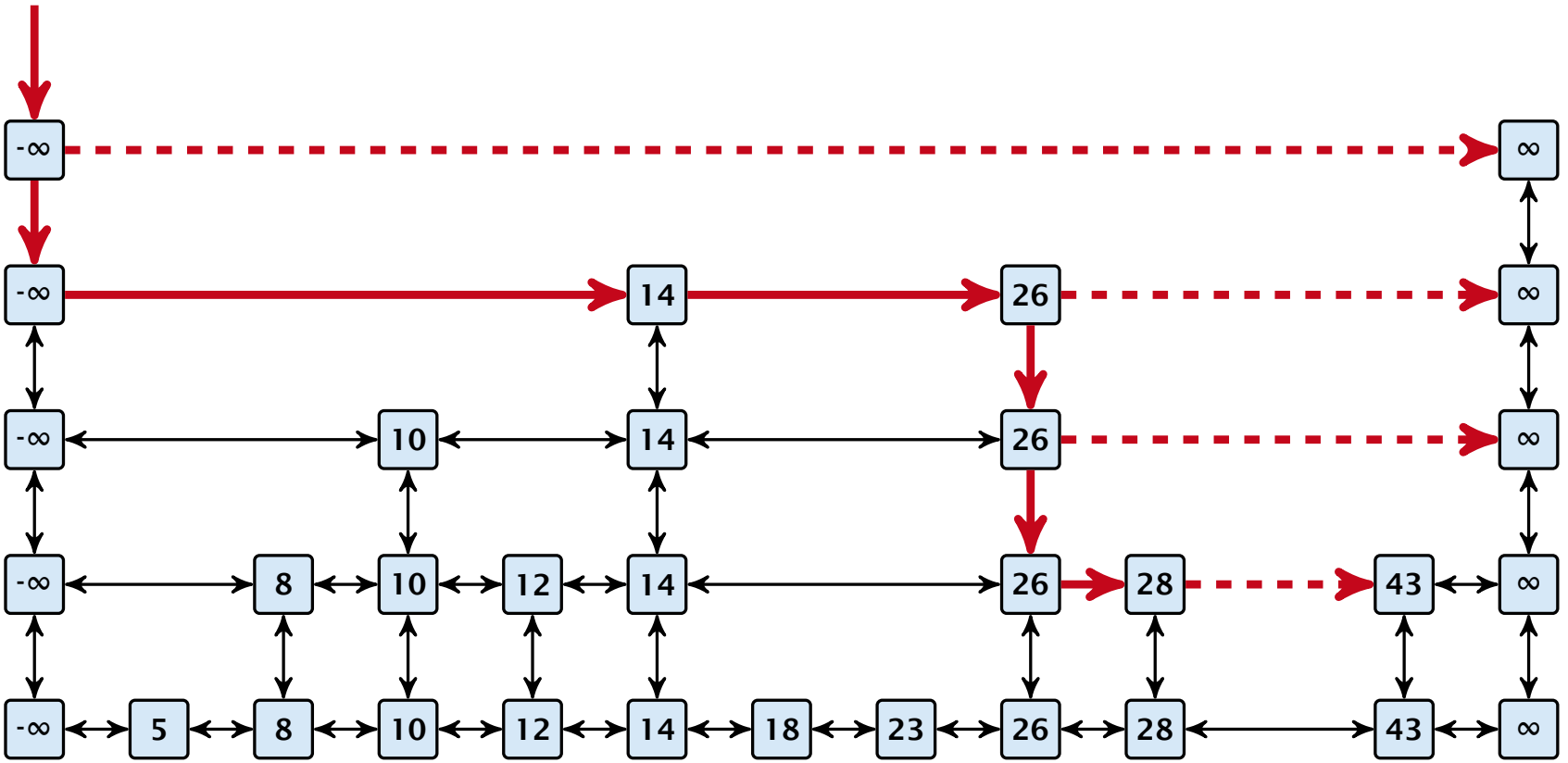
7.5 Skip Lists

Insert (35):



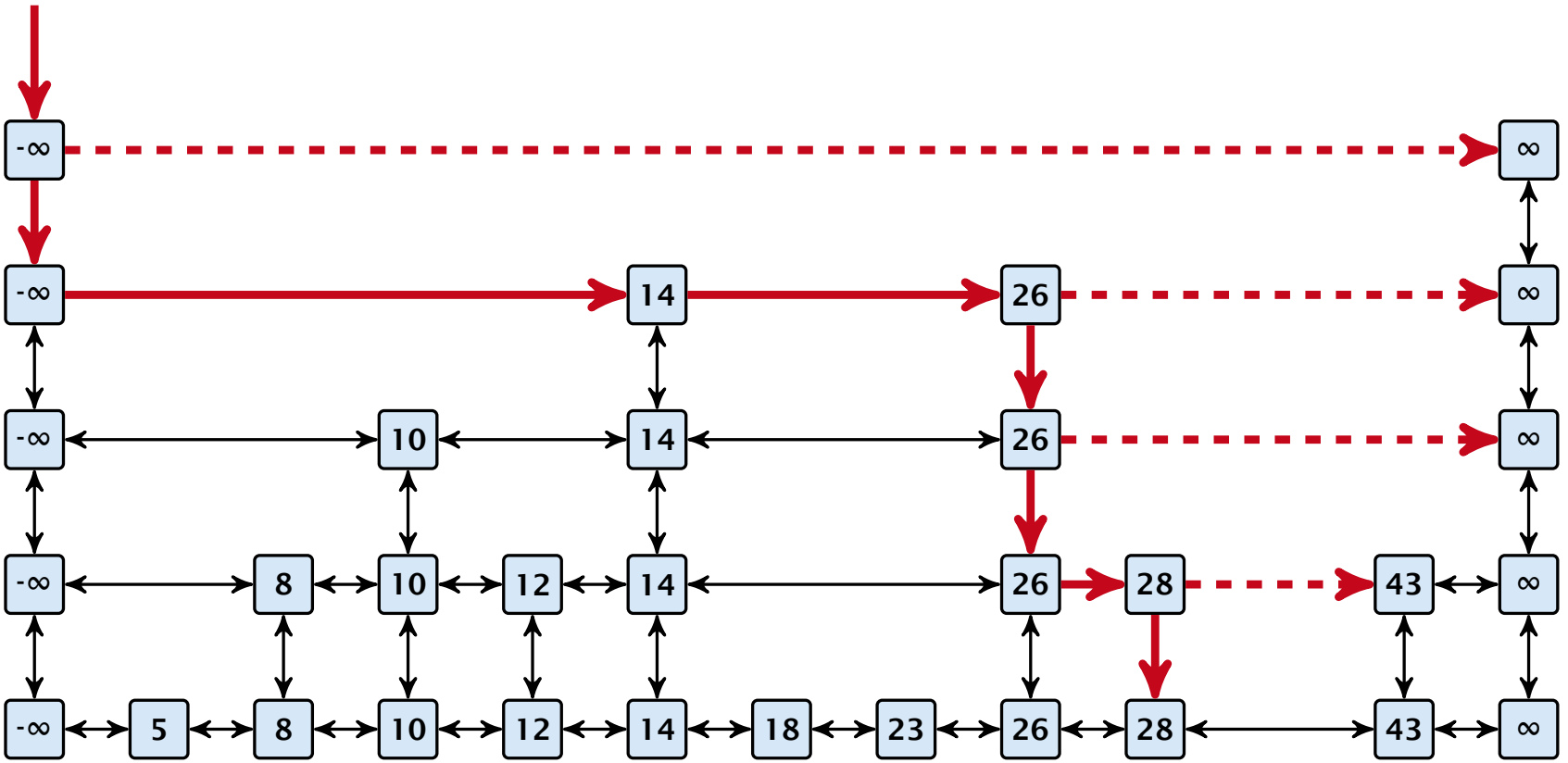
7.5 Skip Lists

Insert (35):



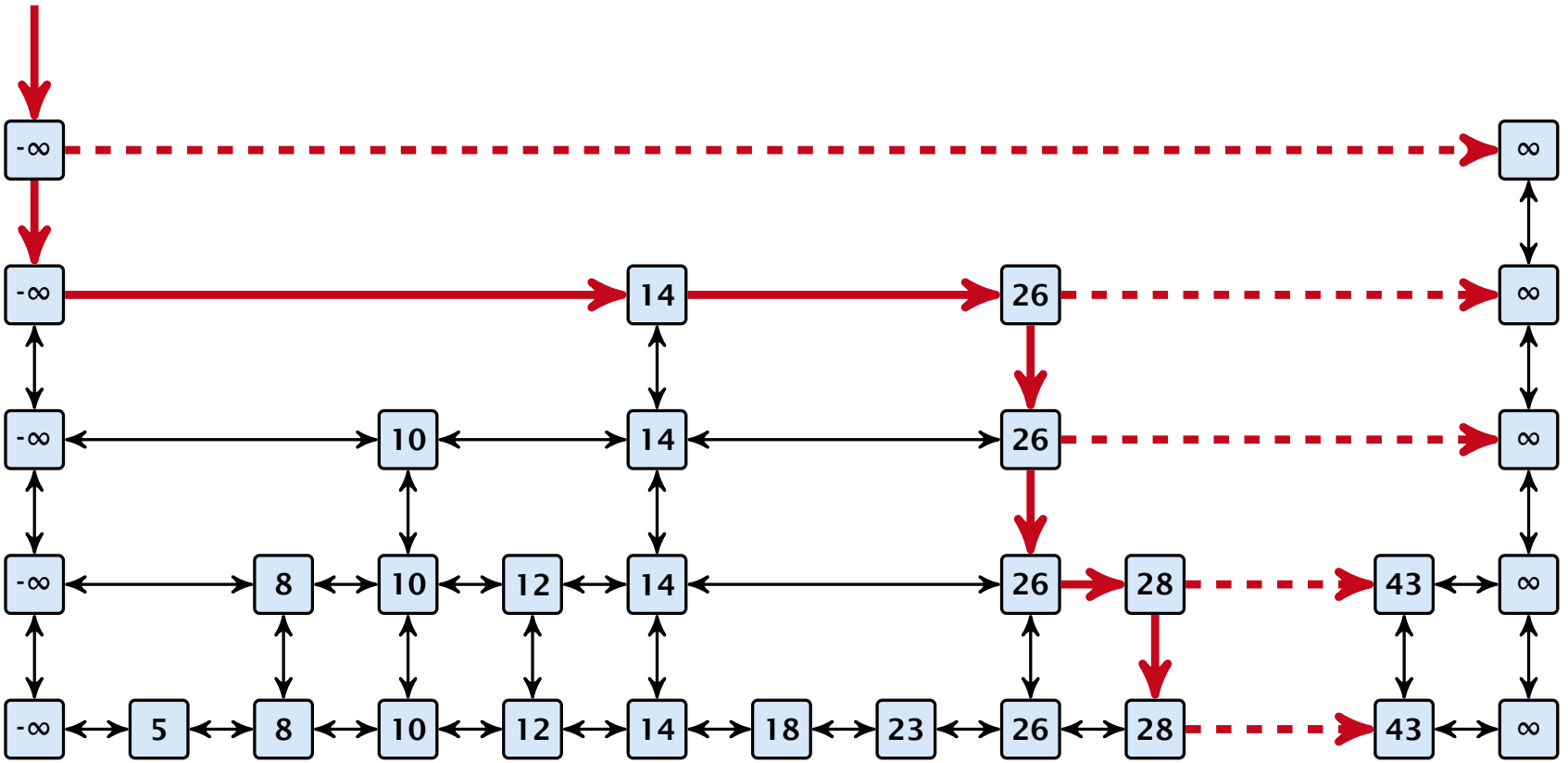
7.5 Skip Lists

Insert (35):



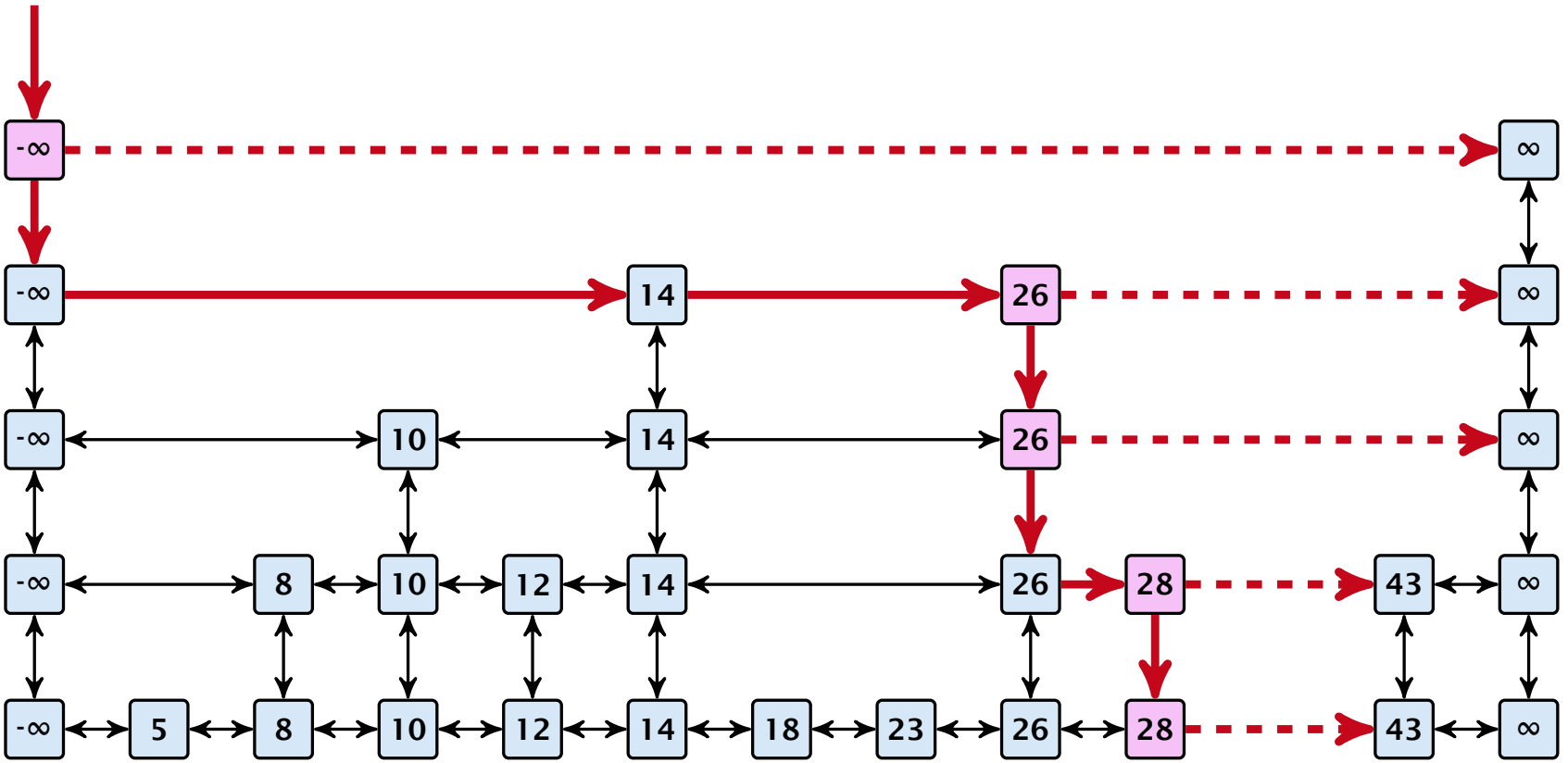
7.5 Skip Lists

Insert (35):



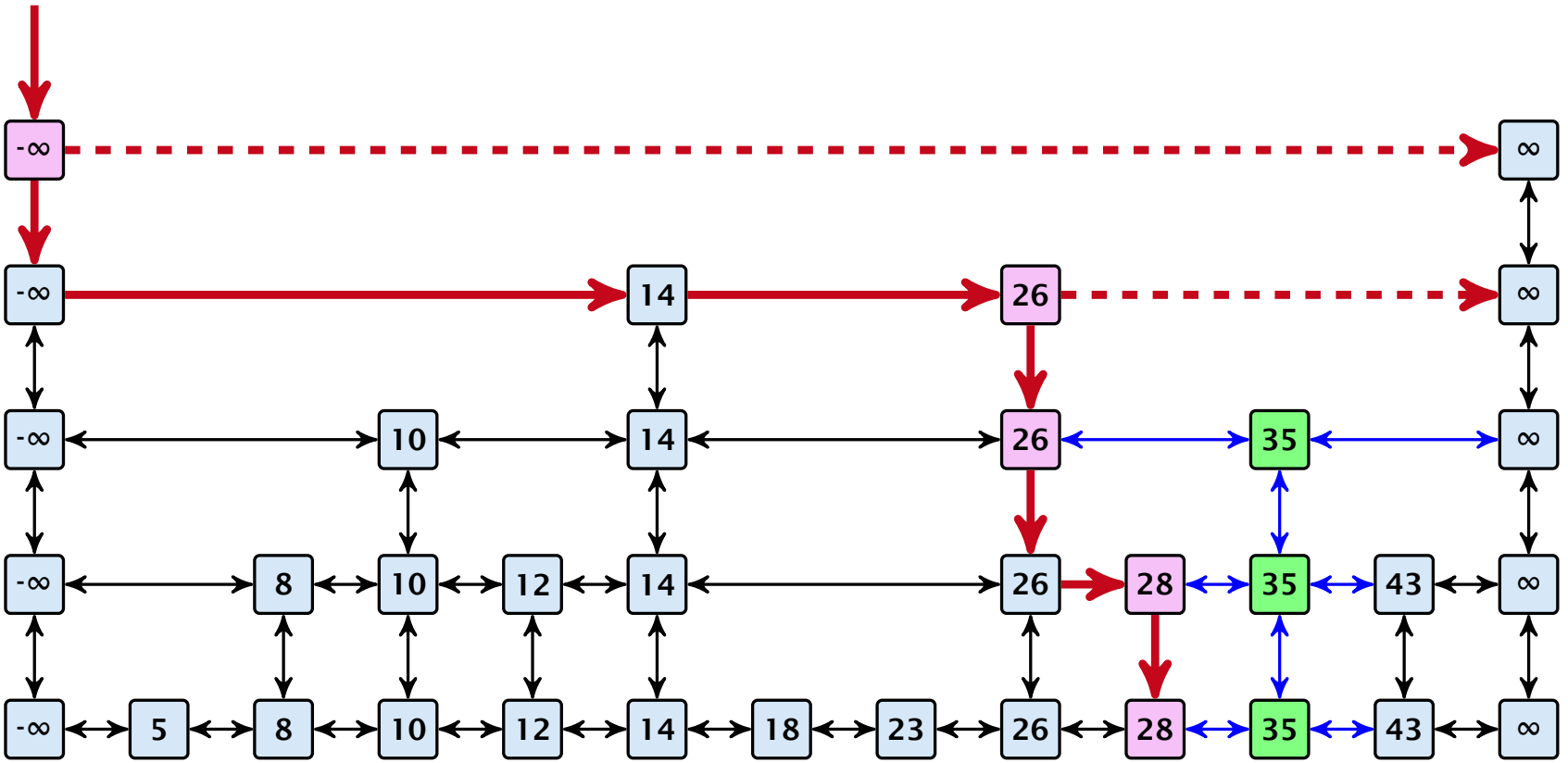
7.5 Skip Lists

Insert (35):



7.5 Skip Lists

Insert (35):



High Probability

$\alpha^2 \cdot \log n + \log\log n = \Theta(\log n)$

Definition 18 (High Probability)

We say a **randomized** algorithm has running time $\mathcal{O}(\log n)$ with high probability if for any constant α the running time is at most $\mathcal{O}(\log n)$ with probability at least $1 - \frac{1}{n^\alpha}$.

Here the $\mathcal{O}$-notation hides a constant that may depend on α.

High Probability

Definition 18 (High Probability)

We say a **randomized** algorithm has running time $\mathcal{O}(\log n)$ with high probability if for any constant α the running time is at most $\mathcal{O}(\log n)$ with probability at least $1 - \frac{1}{n^\alpha}$.

Here the $\mathcal{O}$-notation hides a constant that may depend on α.

High Probability

Suppose there are polynomially many events $E_1, E_2, \ldots, E_\ell$, $\ell = n^c$ each holding with high probability (e.g. E_i may be the event that the i-th search in a skip list takes time at most $\mathcal{O}(\log n)$).

High Probability

Suppose there are polynomially many events $E_1, E_2, \ldots, E_\ell$, $\ell = n^c$ each holding with high probability (e.g. E_i may be the event that the i-th search in a skip list takes time at most $\mathcal{O}(\log n)$).

Then the probability that all E_i hold is at least

$$\Pr[E_1 \wedge \cdots \wedge E_\ell]$$

High Probability

Suppose there are polynomially many events $E_1, E_2, \ldots, E_\ell$, $\ell = n^c$ each holding with high probability (e.g. E_i may be the event that the i-th search in a skip list takes time at most $\mathcal{O}(\log n)$).

Then the probability that all E_i hold is at least

$$\Pr[E_1 \wedge \cdots \wedge E_\ell] = 1 - \Pr[\bar{E}_1 \vee \cdots \vee \bar{E}_\ell]$$

High Probability

Suppose there are polynomially many events $E_1, E_2, \ldots, E_\ell$, $\ell = n^c$ each holding with high probability (e.g. E_i may be the event that the i-th search in a skip list takes time at most $\mathcal{O}(\log n)$).

Then the probability that all E_i hold is at least

$$\begin{aligned}\Pr[E_1 \wedge \cdots \wedge E_\ell] &= 1 - \Pr[\bar{E}_1 \vee \cdots \vee \bar{E}_\ell] \\ &\geq 1 - n^c \cdot n^{-\alpha}\end{aligned}$$

High Probability

Suppose there are polynomially many events $E_1, E_2, \ldots, E_\ell$, $\ell = n^c$ each holding with high probability (e.g. E_i may be the event that the i-th search in a skip list takes time at most $\mathcal{O}(\log n)$).

Then the probability that all E_i hold is at least

$$\begin{aligned}\Pr[E_1 \wedge \cdots \wedge E_\ell] &= 1 - \Pr[\bar{E}_1 \vee \cdots \vee \bar{E}_\ell] \\ &\geq 1 - n^c \cdot n^{-\alpha} \\ &= 1 - n^{c-\alpha} \ .\end{aligned}$$

High Probability

Suppose there are polynomially many events $E_1, E_2, \ldots, E_\ell$, $\ell = n^c$ each holding with high probability (e.g. E_i may be the event that the i-th search in a skip list takes time at most $\mathcal{O}(\log n)$).

Then the probability that all E_i hold is at least

$$\begin{aligned}\Pr[E_1 \wedge \cdots \wedge E_\ell] &= 1 - \Pr[\bar{E}_1 \vee \cdots \vee \bar{E}_\ell] \\ &\geq 1 - n^c \cdot n^{-\alpha} \\ &= 1 - n^{c-\alpha} \ .\end{aligned}$$

This means $\Pr[E_1 \wedge \cdots \wedge E_\ell]$ holds with high probability.