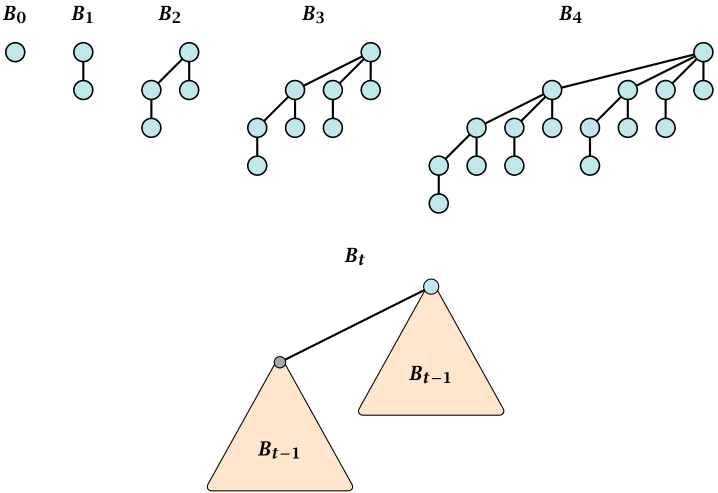
8.2 Binomial Heaps

Operation	Binary Heap	BST	Binomial Heap	Fibonacci Heap*
build	n	$n \log n$	$n \log n$	n
minimum	1	$\log n$	$\log n$	1
is-empty	1	1	1	1
insert	$\log n$	$\log n$	$\log n$	1
delete	$\log n^{**}$	$\log n$	$\log n$	$\log n$
delete-min	$\log n$	$\log n$	$\log n$	$\log n$
decrease-key	$\log n$	$\log n$	$\log n$	1
merge	n	$n \log n$	$\boldsymbol{\log n}$	1

Binomial Trees



© Ernst Mayr, Harald Räcke

Binomial Trees

Properties of Binomial Trees

- B_k has 2^k nodes.
- B_k has height k.
- The root of B_k has degree k.
- B_k has $\binom{k}{\ell}$ nodes on level ℓ.
- Deleting the root of B_k gives trees $B_0, B_1, \ldots, B_{k-1}$.

© Ernst Mayr, Harald Räcke

Binomial Trees

Properties of Binomial Trees

- B_k has 2^k nodes.
- B_k has height k.
- The root of B_k has degree k.
- B_k has $\binom{k}{\ell}$ nodes on level ℓ.
- Deleting the root of B_k gives trees $B_0, B_1, \ldots, B_{k-1}$.

Binomial Trees

Properties of Binomial Trees

- B_k has 2^k nodes.
- B_k has height k.
- The root of B_k has degree k.
- B_k has $\binom{k}{\ell}$ nodes on level ℓ.
- Deleting the root of B_k gives trees $B_0, B_1, \ldots, B_{k-1}$.

Binomial Trees

Properties of Binomial Trees

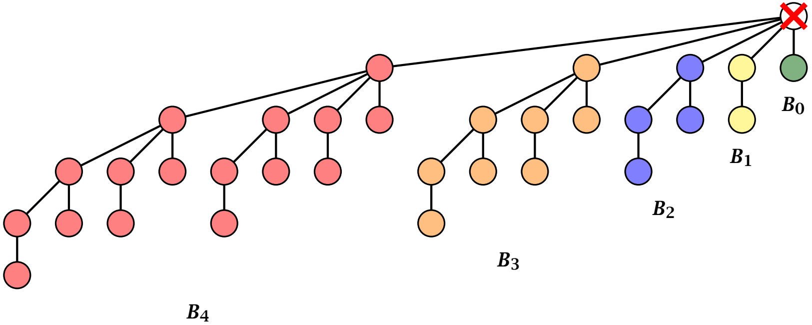
- B_k has 2^k nodes.
- B_k has height k.
- The root of B_k has degree k.
- B_k has $\binom{k}{\ell}$ nodes on level ℓ.
- Deleting the root of B_k gives trees $B_0, B_1, \ldots, B_{k-1}$.

Binomial Trees

Properties of Binomial Trees

- B_k has 2^k nodes.
- B_k has height k.
- The root of B_k has degree k.
- B_k has $\binom{k}{\ell}$ nodes on level ℓ.
- Deleting the root of B_k gives trees $B_0, B_1, \ldots, B_{k-1}$.

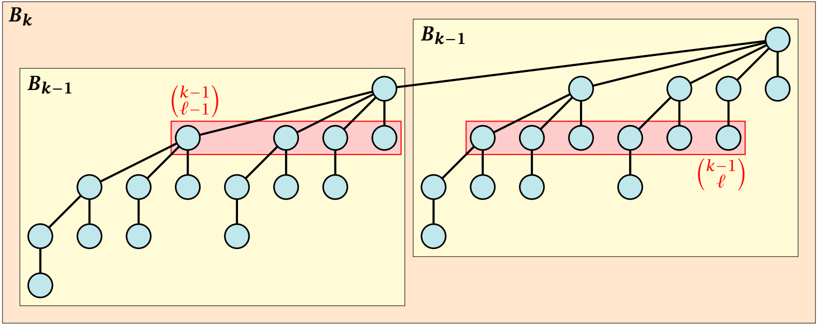
Binomial Trees



Deleting the root of B_5 leaves sub-trees B_4, B_3, B_2, and B_1.

© Ernst Mayr, Harald Räcke

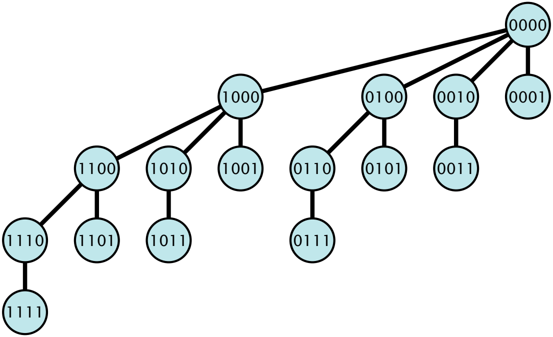
Binomial Trees



The number of nodes on level ℓ in tree B_k is therefore

$$\binom{k-1}{\ell-1} + \binom{k-1}{\ell} = \binom{k}{\ell}$$

Binomial Trees

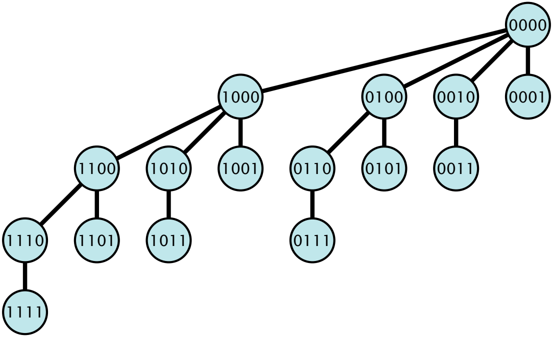


The binomial tree B_k is a sub-graph of the hypercube H_k.

The parent of a node with label $b_n, \dots, b_1, b_0$ is obtained by setting the least significant 1-bit to 0.

The ℓ-th level contains nodes that have ℓ 1's in their label.

Binomial Trees

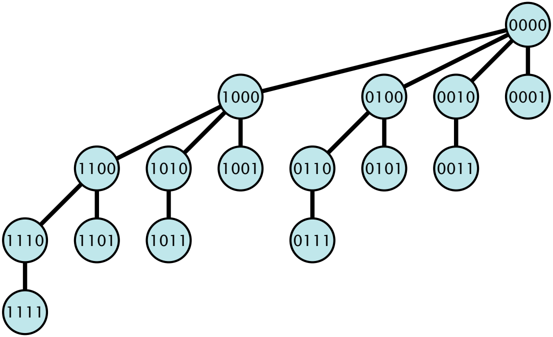


The binomial tree B_k is a sub-graph of the hypercube H_k.

The parent of a node with label $b_n, \ldots, b_1, b_0$ is obtained by setting the least significant 1-bit to 0.

The ℓ-th level contains nodes that have ℓ 1's in their label.

Binomial Trees

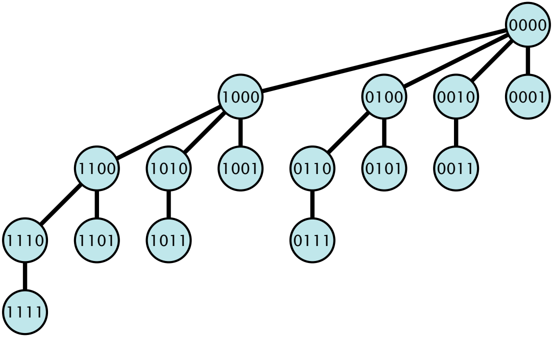


The binomial tree B_k is a sub-graph of the hypercube H_k.

The parent of a node with label $b_n, \ldots, b_1, b_0$ is obtained by setting the least significant 1-bit to 0.

The ℓ-th level contains nodes that have ℓ 1's in their label.

Binomial Trees



The binomial tree B_k is a sub-graph of the hypercube H_k.

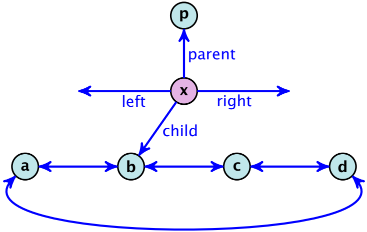
The parent of a node with label $b_n, \ldots, b_1, b_0$ is obtained by setting the least significant 1-bit to 0.

The ℓ-th level contains nodes that have ℓ 1's in their label.

8.2 Binomial Heaps

How do we implement trees with non-constant degree?

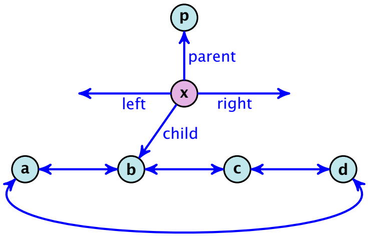
- The children of a node are arranged in a circular linked list.
- A child-pointer points to an arbitrary node within the list.
- A parent-pointer points to the parent node.
- Pointers $x.\mathrm{left}$ and $x.\mathrm{right}$ point to the left and right sibling of x (if x does not have children then $x.\mathrm{left} = x.\mathrm{right} = x$).



8.2 Binomial Heaps

How do we implement trees with non-constant degree?

- The children of a node are arranged in a circular linked list.
- A child-pointer points to an arbitrary node within the list.
- A parent-pointer points to the parent node.
- Pointers $x.\text{left}$ and $x.\text{right}$ point to the left and right sibling of x (if x does not have children then $x.\text{left} = x.\text{right} = x$).

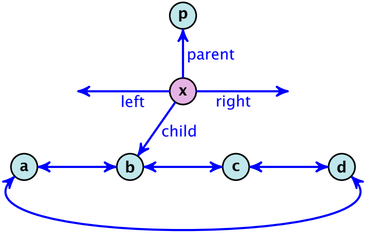


© Ernst Mayr, Harald Räcke

8.2 Binomial Heaps

How do we implement trees with non-constant degree?

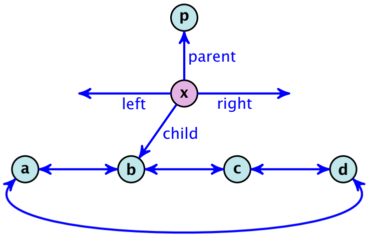
- The children of a node are arranged in a circular linked list.
- A child-pointer points to an arbitrary node within the list.
- A parent-pointer points to the parent node.
- Pointers $x.\mathrm{left}$ and $x.\mathrm{right}$ point to the left and right sibling of x (if x does not have children then $x.\mathrm{left} = x.\mathrm{right} = x$).



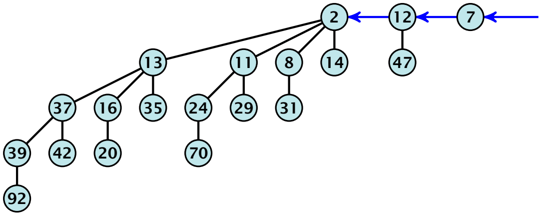
8.2 Binomial Heaps

How do we implement trees with non-constant degree?

- The children of a node are arranged in a circular linked list.
- A child-pointer points to an arbitrary node within the list.
- A parent-pointer points to the parent node.
- Pointers $x.\text{left}$ and $x.\text{right}$ point to the left and right sibling of x (if x does not have children then $x.\text{left} = x.\text{right} = x$).



Binomial Heap

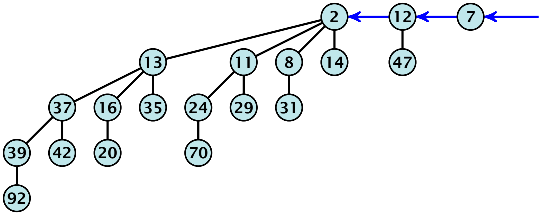


In a binomial heap the keys are arranged in a collection of binomial trees.

Every tree fulfills the heap-property

There is at most one tree for every dimension/order. For example the above heap contains trees B_0, B_1, and B_4.

Binomial Heap

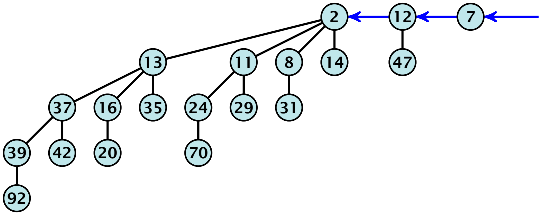


In a binomial heap the keys are arranged in a collection of binomial trees.

Every tree fulfills the heap-property

There is at most one tree for every dimension/order. For example the above heap contains trees B_0, B_1, and B_4.

Binomial Heap

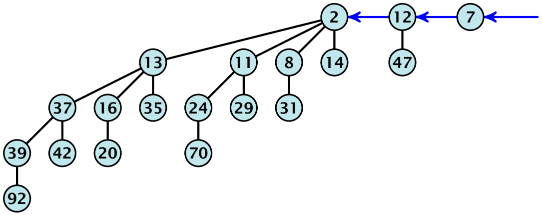


In a binomial heap the keys are arranged in a collection of binomial trees.

Every tree fulfills the heap-property

There is at most one tree for every dimension/order. For example the above heap contains trees B_0, B_1, and B_4.

Binomial Heap



In a binomial heap the keys are arranged in a collection of binomial trees.

Every tree fulfills the heap-property

There is at most one tree for every dimension/order. For example the above heap contains trees B_0, B_1, and B_4.

Binomial Heap: Merge

Given the number n of keys to be stored in a binomial heap we can deduce the binomial trees that will be contained in the collection.

Let $B_{k_1}, B_{k_2}, B_{k_3}, k_i < k_{i+1}$ denote the binomial trees in the collection and recall that every tree may be contained at most once.

Then $n = \sum_i 2^{k_i}$ must hold. But since the k_i are all distinct this means that the k_i define the non-zero bit-positions in the dual representation of n.

Binomial Heap: Merge

Given the number n of keys to be stored in a binomial heap we can deduce the binomial trees that will be contained in the collection.

Let $B_{k_1}, B_{k_2}, B_{k_3}, k_i < k_{i+1}$ denote the binomial trees in the collection and recall that every tree may be contained at most once.

Then $n = \sum_i 2^{k_i}$ must hold. But since the k_i are all distinct this means that the k_i define the non-zero bit-positions in the dual representation of n.

Binomial Heap: Merge

Given the number n of keys to be stored in a binomial heap we can deduce the binomial trees that will be contained in the collection.

Let $B_{k_1}, B_{k_2}, B_{k_3}, k_i < k_{i+1}$ denote the binomial trees in the collection and recall that every tree may be contained at most once.

Then $n = \sum_i 2^{k_i}$ must hold. But since the k_i are all distinct this means that the k_i define the non-zero bit-positions in the dual representation of n.

Binomial Heap: Merge

Given the number n of keys to be stored in a binomial heap we can deduce the binomial trees that will be contained in the collection.

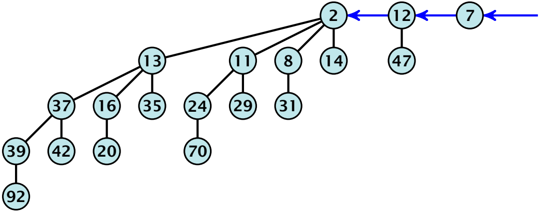
Let $B_{k_1}, B_{k_2}, B_{k_3}, k_i < k_{i+1}$ denote the binomial trees in the collection and recall that every tree may be contained at most once.

Then $n = \sum_i 2^{k_i}$ must hold. But since the k_i are all distinct this means that the k_i define the non-zero bit-positions in the dual representation of n.

Binomial Heap

Properties of a heap with n keys:

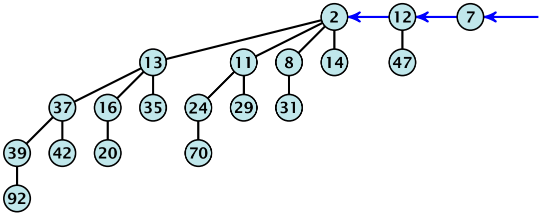
- Let $n = b_d b_{d-1}, \ldots, b_0$ denote the dual representation of n.
- The heap contains tree B_i iff $b_i = 1$.
- Hence, at most $\lfloor \log n \rfloor + 1$ trees.
- The minimum must be contained in one of the roots.
- The height of the largest tree is at most $\lfloor \log n \rfloor$.
- The trees are stored in a single-linked list; ordered by dimension/size.



Binomial Heap

Properties of a heap with n keys:

- Let $n = b_d b_{d-1}, \ldots, b_0$ denote the dual representation of n.
- The heap contains tree B_i iff $b_i = 1$.
- Hence, at most $\lfloor \log n \rfloor + 1$ trees.
- The minimum must be contained in one of the roots.
- The height of the largest tree is at most $\lfloor \log n \rfloor$.
- The trees are stored in a single-linked list; ordered by dimension/size.

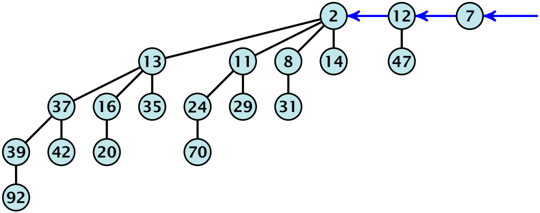


© Ernst Mayr, Harald Räcke

Binomial Heap

Properties of a heap with n keys:

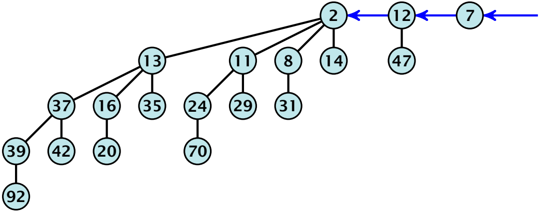
- Let $n = b_d b_{d-1}, \ldots, b_0$ denote the dual representation of n.
- The heap contains tree B_i iff $b_i = 1$.
- Hence, at most $\lfloor \log n \rfloor + 1$ trees.
- The minimum must be contained in one of the roots.
- The height of the largest tree is at most $\lfloor \log n \rfloor$.
- The trees are stored in a single-linked list; ordered by dimension/size.



Binomial Heap

Properties of a heap with n keys:

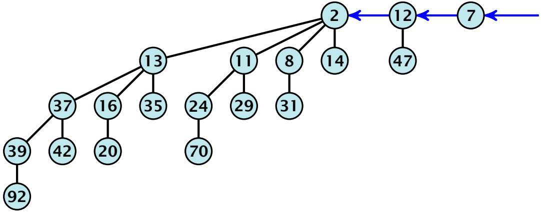
- Let $n = b_d b_{d-1}, \ldots, b_0$ denote the dual representation of n.
- The heap contains tree B_i iff $b_i = 1$.
- Hence, at most $\lfloor \log n \rfloor + 1$ trees.
- The minimum must be contained in one of the roots.
- The height of the largest tree is at most $\lfloor \log n \rfloor$.
- The trees are stored in a single-linked list; ordered by dimension/size.



Binomial Heap

Properties of a heap with n keys:

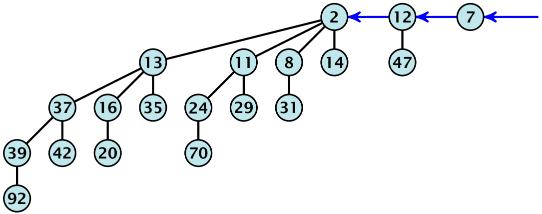
- Let $n = b_d b_{d-1}, \dots, b_0$ denote the dual representation of n.
- The heap contains tree B_i iff $b_i = 1$.
- Hence, at most $\lfloor \log n \rfloor + 1$ trees.
- The minimum must be contained in one of the roots.
- The height of the largest tree is at most $\lfloor \log n \rfloor$.
- The trees are stored in a single-linked list; ordered by dimension/size.



Binomial Heap

Properties of a heap with n keys:

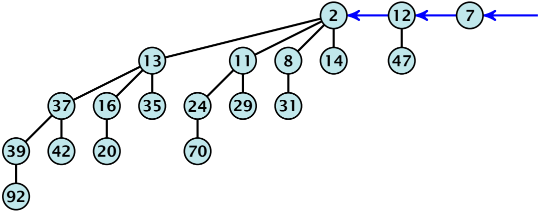
- Let $n = b_d b_{d-1}, \ldots, b_0$ denote the dual representation of n.
- The heap contains tree B_i iff $b_i = 1$.
- Hence, at most $\lfloor \log n \rfloor + 1$ trees.
- The minimum must be contained in one of the roots.
- The height of the largest tree is at most $\lfloor \log n \rfloor$.
- The trees are stored in a single-linked list; ordered by dimension/size.



Binomial Heap

Properties of a heap with n keys:

- Let $n = b_d b_{d-1}, \dots, b_0$ denote the dual representation of n.
- The heap contains tree B_i iff $b_i = 1$.
- Hence, at most $\lfloor \log n \rfloor + 1$ trees.
- The minimum must be contained in one of the roots.
- The height of the largest tree is at most $\lfloor \log n \rfloor$.
- The trees are stored in a single-linked list; ordered by dimension/size.



Binomial Heap: Merge

The merge-operation is instrumental for binomial heaps.

A merge is easy if we have two heaps with different binomial trees. We can simply merge the tree-lists.

Otherwise, we cannot do this because the merged heap is not allowed to contain two trees of the same order.

Merging two trees of the same size: Add the tree with larger root-value as a child to the other tree.

For more trees the technique is analogous to binary addition.

Binomial Heap: Merge

The merge-operation is instrumental for binomial heaps.

A merge is easy if we have two heaps with different binomial trees. We can simply merge the tree-lists.

Otherwise, we cannot do this because the merged heap is not allowed to contain two trees of the same order.

Merging two trees of the same size: Add the tree with larger root-value as a child to the other tree.

Binomial Heap: Merge

The merge-operation is instrumental for binomial heaps.

A merge is easy if we have two heaps with different binomial trees. We can simply merge the tree-lists.

Otherwise, we cannot do this because the merged heap is not allowed to contain two trees of the same order.

Merging two trees of the same size: Add the tree with larger root-value as a child to the other tree.

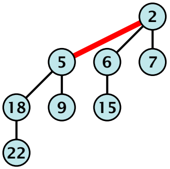
Binomial Heap: Merge

The merge-operation is instrumental for binomial heaps.

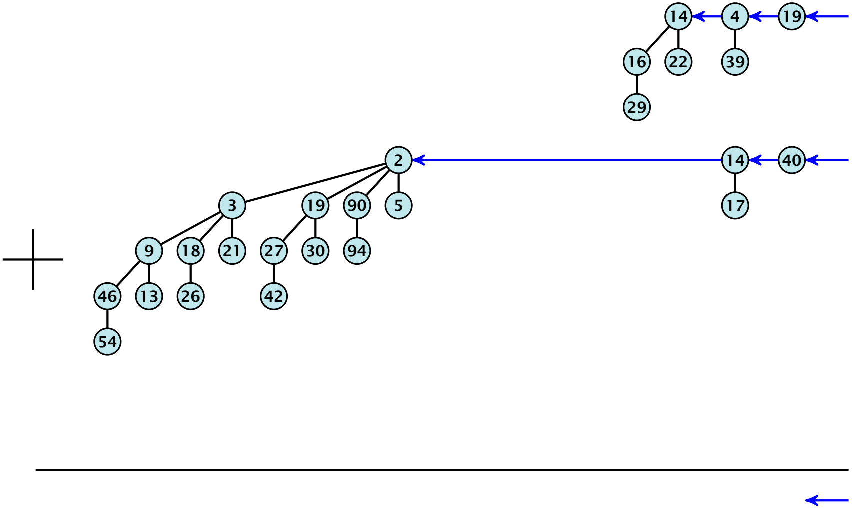
A merge is easy if we have two heaps with different binomial trees. We can simply merge the tree-lists.

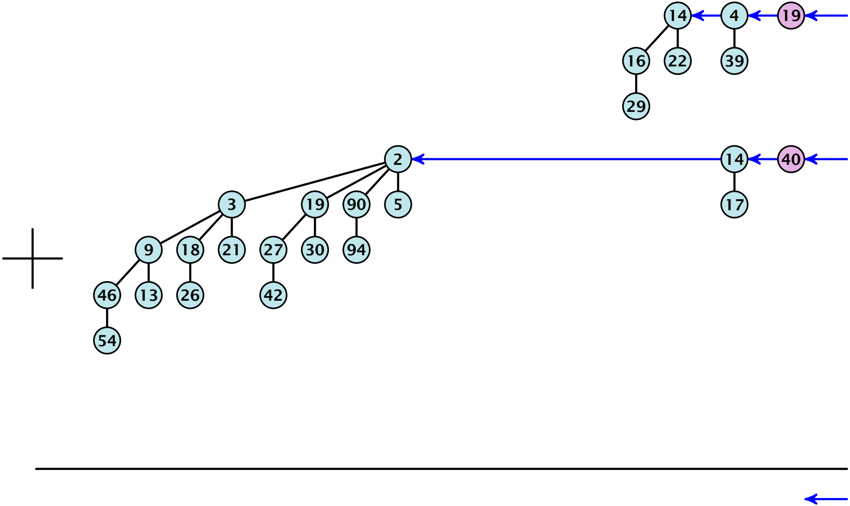
Otherwise, we cannot do this because the merged heap is not allowed to contain two trees of the same order.

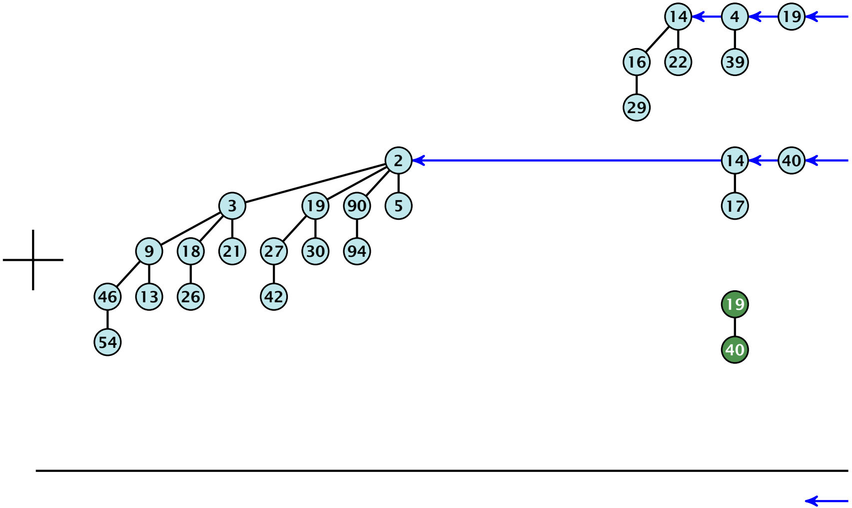
Merging two trees of the same size: Add the tree with larger root-value as a child to the other tree.



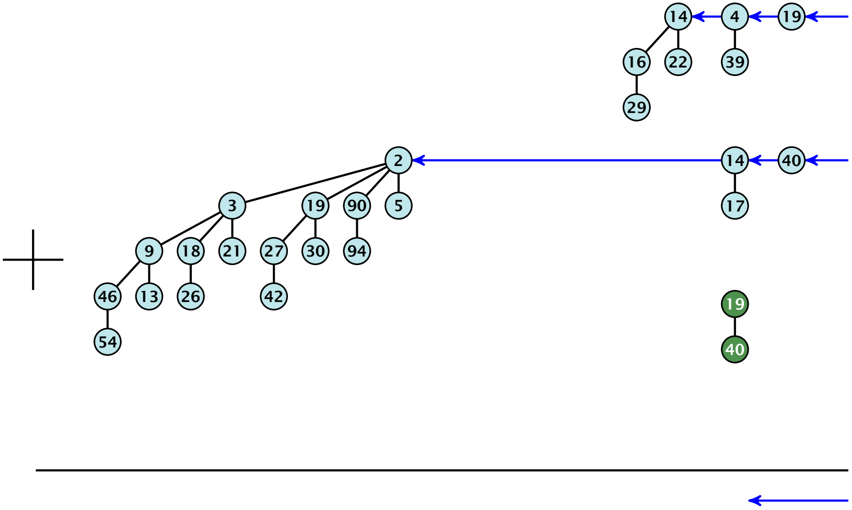
For more trees the technique is analogous to binary addition.


14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
54

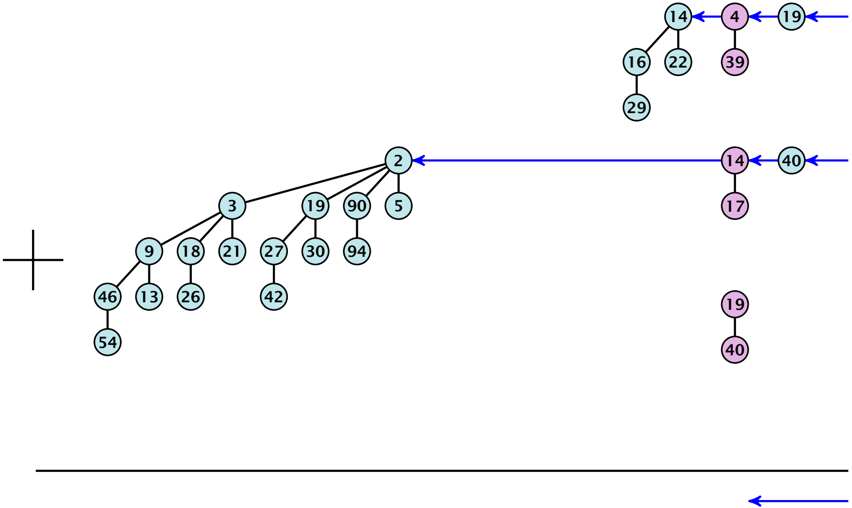

14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
54

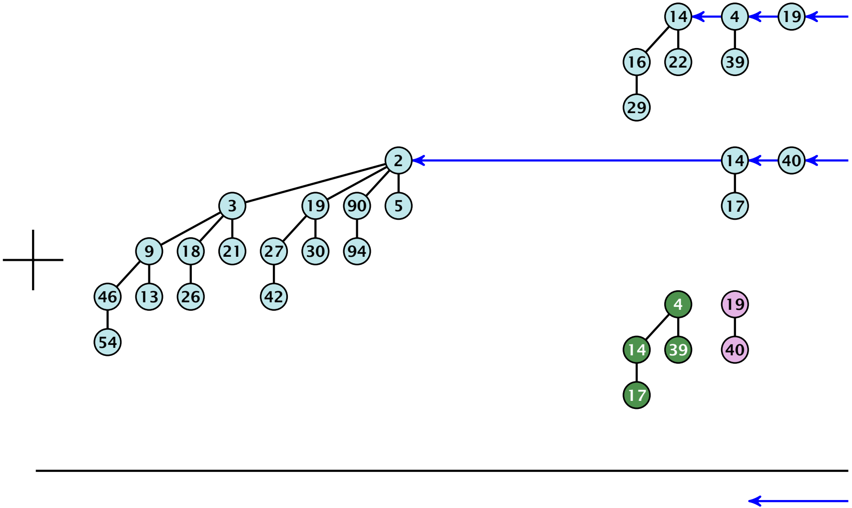


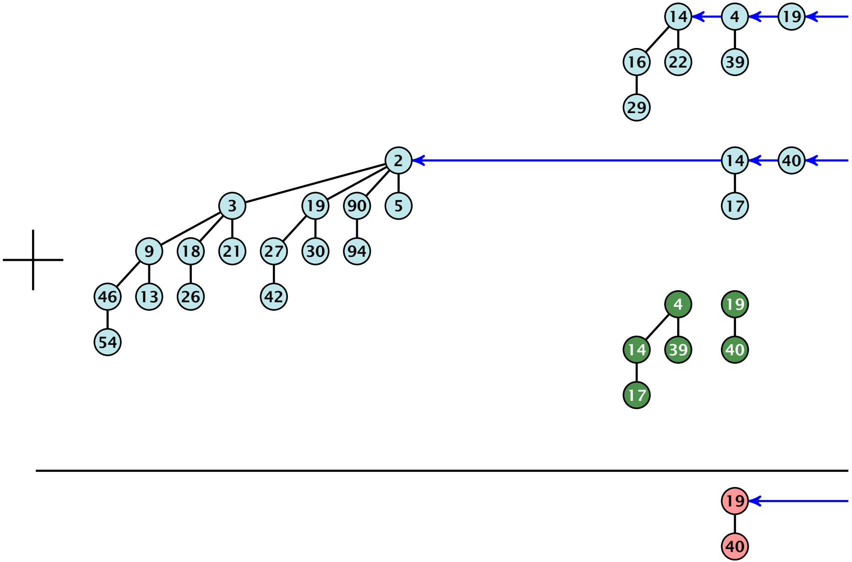
14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
19
40
54



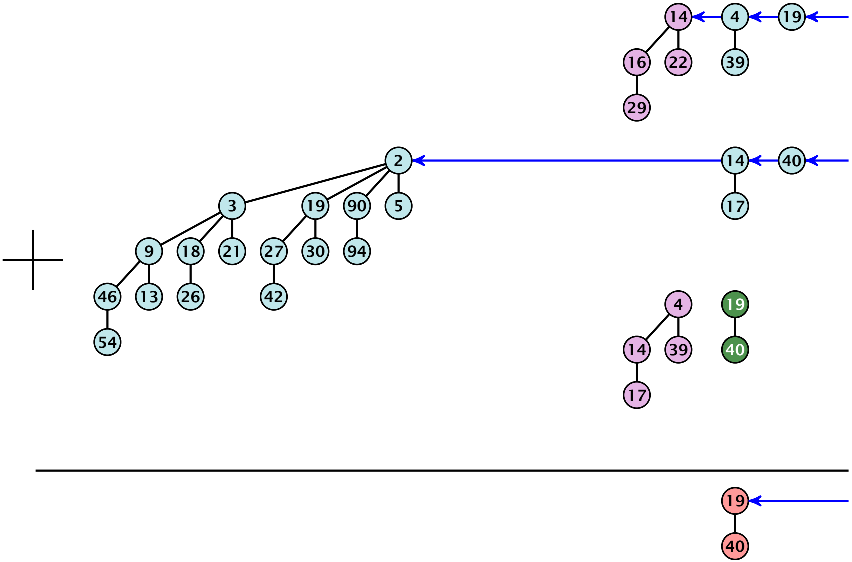
14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
19
54
40


14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
19
54
40

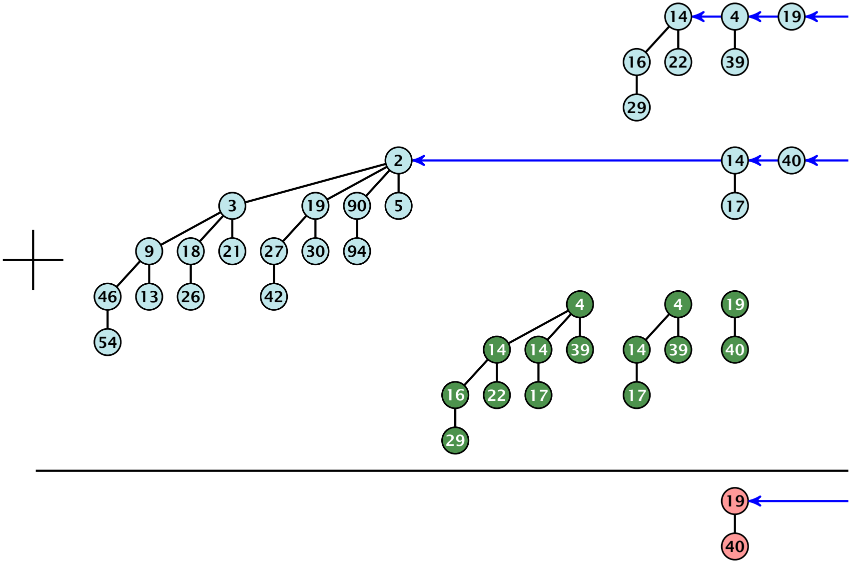

14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
4
19
54
14
39
40
17

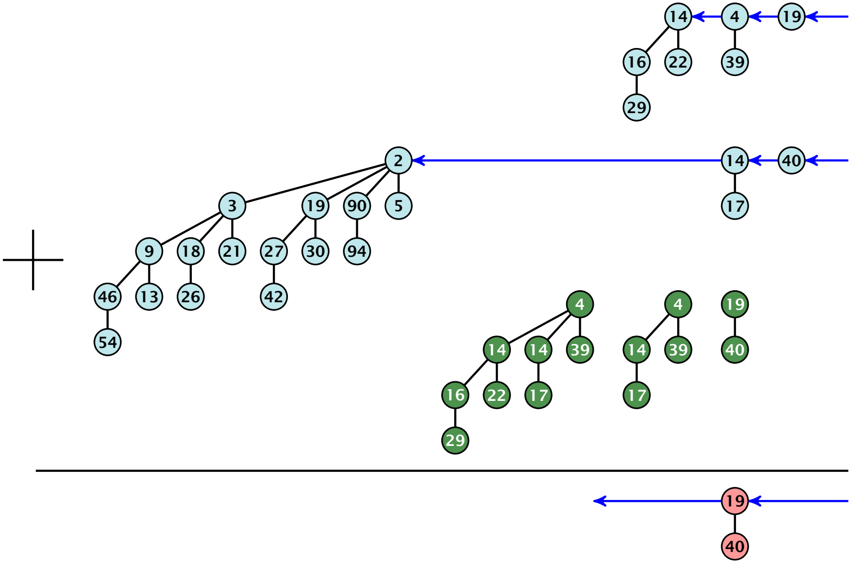


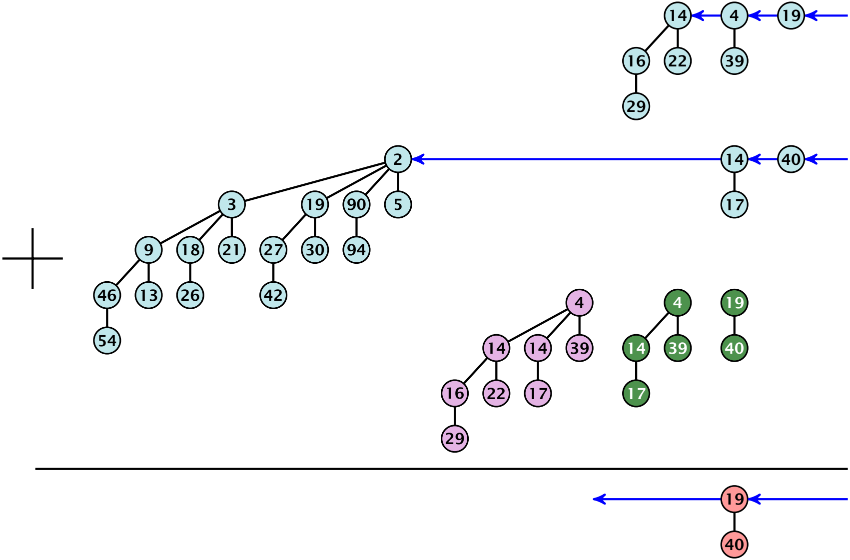
14 4 19
16 22 39
29
2 14 40
3 19 90 5 17
9 18 21 27 30 94
46 13 26 42
54
4 19
14 39 40
17
19
40



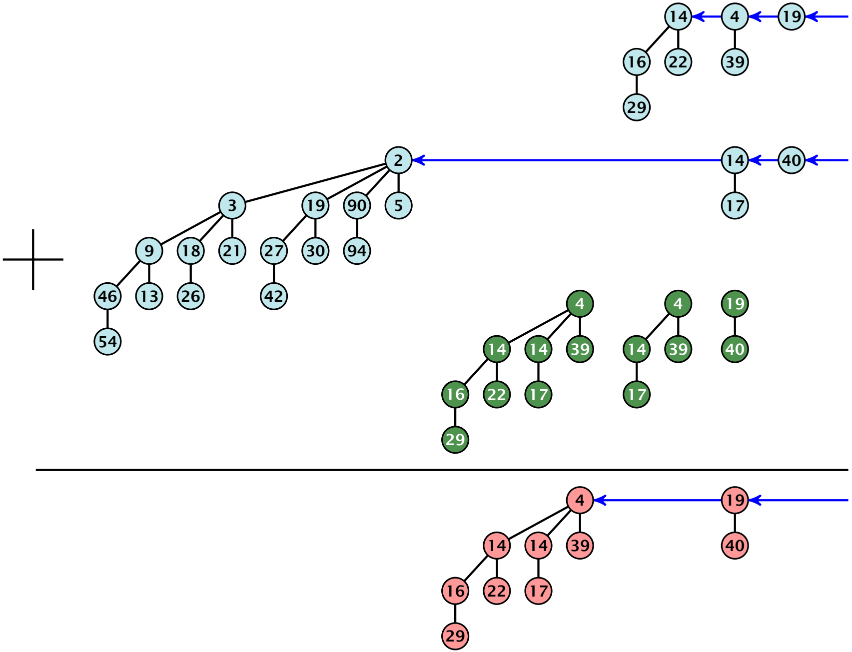
14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
54
4
19
14
39
40
17
19
40

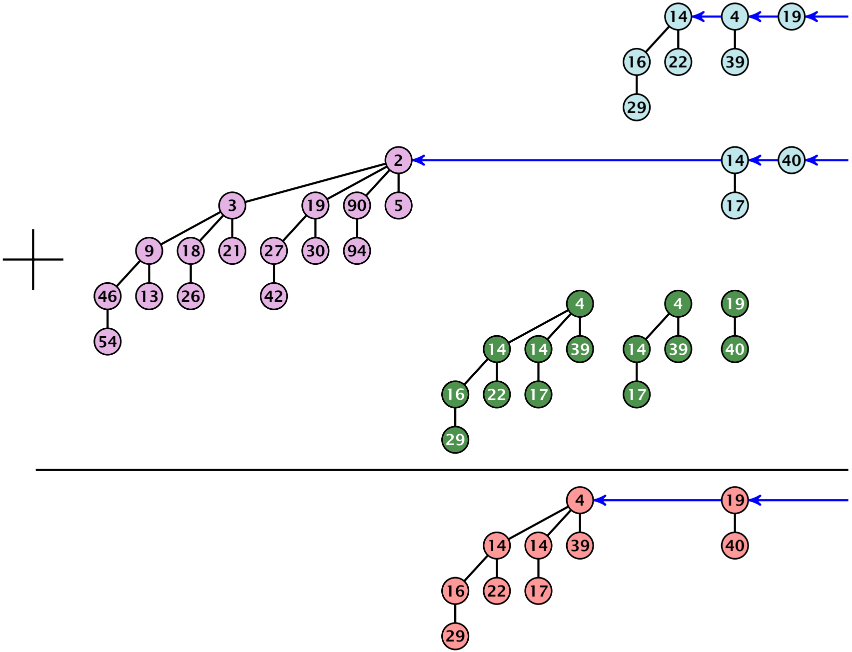



14 4 19
16 22 39
29
2 14 40
3 19 90 5 17
9 18 21 27 30 94
46 13 26 42
54
4 4 19
14 14 39 14 39 40
16 22 17 17
29
19
40

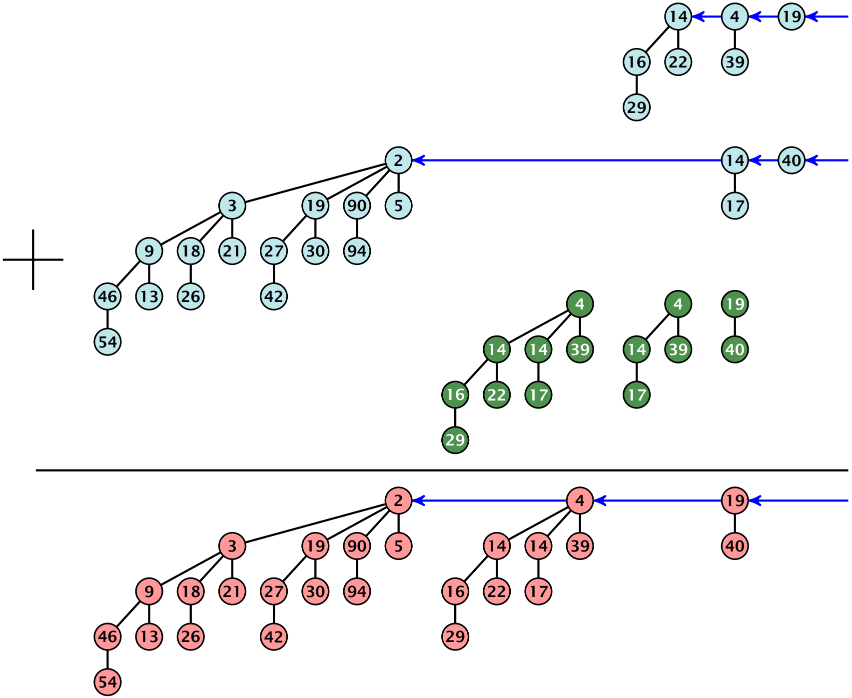


14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
54
4
4
19
14
14
39
14
39
40
16
22
17
17
29
19
40

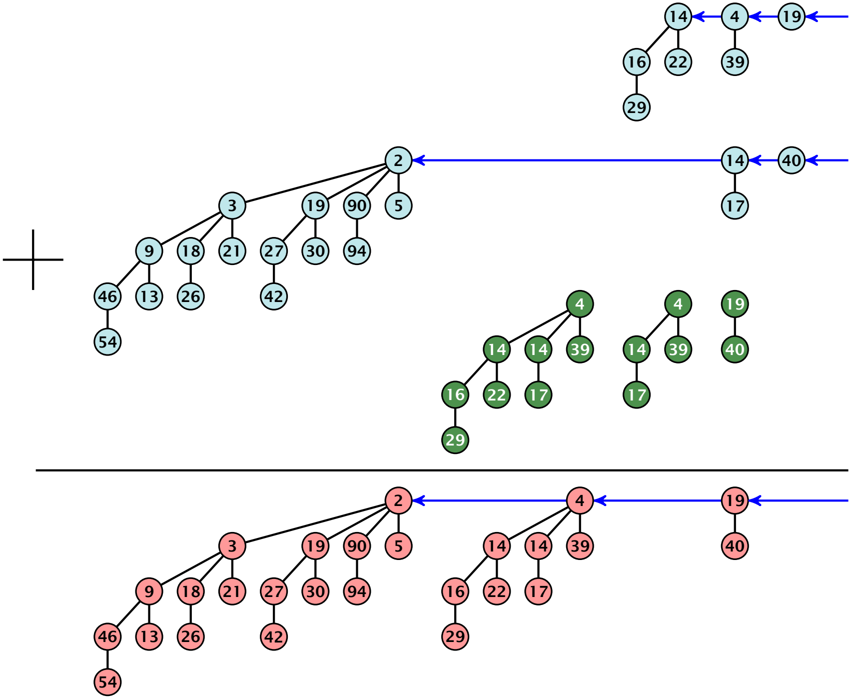

14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
54
4
4
19
14
14
39
14
39
40
16
22
17
17
29
4
19
14
14
39
40
16
22
17
29



14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
54
4
4
19
14
14
39
14
39
40
16
22
17
17
29
4
19
14
14
39
40
16
22
17
29



14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
54
4
4
19
14
14
39
14
39
40
16
22
17
17
29
2
4
19
3
19
90
5
14
14
39
40
9
18
21
27
30
94
16
22
17
46
13
26
42
29
54



14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
54
4
4
19
14
14
39
14
39
40
16
22
17
17
29
2
4
19
3
19
90
5
14
14
39
40
9
18
21
27
30
94
16
22
17
46
13
26
42
29
54

8.2 Binomial Heaps

S_1.merge(S_2):

- Analogous to binary addition.
- Time is proportional to the number of trees in both heaps.
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S_1.merge(S_2):

- Analogous to binary addition.
- Time is proportional to the number of trees in both heaps.
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S_1.merge(S_2):

- Analogous to binary addition.
- Time is proportional to the number of trees in both heaps.
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

All other operations can be reduced to merge().

S.insert(x):

- Create a new heap S' that contains just the element x.
- Execute S.merge(S').
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

All other operations can be reduced to merge().

S.insert(x):

- Create a new heap S' that contains just the element x.
- Execute S.merge(S').
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

All other operations can be reduced to merge().

S.insert(x):

- Create a new heap S' that contains just the element x.
- Execute S.merge(S').
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.minimum():

- Find the minimum key-value among all roots.
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.delete-min():

- Find the minimum key-value among all roots.
- Remove the corresponding tree $T_{\min}$ from the heap.
- Create a new heap S' that contains the trees obtained from $T_{\min}$ after deleting the root (note that these are just $\mathcal{O}(\log n)$ trees).
- Compute S.merge(S').
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.delete-min():

- Find the minimum key-value among all roots.
- Remove the corresponding tree $T_{\min}$ from the heap.
- Create a new heap S' that contains the trees obtained from $T_{\min}$ after deleting the root (note that these are just $\mathcal{O}(\log n)$ trees).
- Compute S.merge(S').
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.delete-min():

- Find the minimum key-value among all roots.
- Remove the corresponding tree $T_{\min}$ from the heap.
- Create a new heap S' that contains the trees obtained from $T_{\min}$ after deleting the root (note that these are just $\mathcal{O}(\log n)$ trees).
- Compute S.merge(S').
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.delete-min():

- Find the minimum key-value among all roots.
- Remove the corresponding tree $T_{\min}$ from the heap.
- Create a new heap S' that contains the trees obtained from $T_{\min}$ after deleting the root (note that these are just $\mathcal{O}(\log n)$ trees).
- Compute S.merge(S').
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.delete-min():

- Find the minimum key-value among all roots.
- Remove the corresponding tree $T_{\min}$ from the heap.
- Create a new heap S' that contains the trees obtained from $T_{\min}$ after deleting the root (note that these are just $\mathcal{O}(\log n)$ trees).
- Compute S.merge(S').
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.delete-min():

- Find the minimum key-value among all roots.
- Remove the corresponding tree $T_{\min}$ from the heap.
- Create a new heap S' that contains the trees obtained from $T_{\min}$ after deleting the root (note that these are just $\mathcal{O}(\log n)$ trees).
- Compute S.merge(S').
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.decrease-key(handle h):

- Decrease the key of the element pointed to by h.
- Bubble the element up in the tree until the heap property is fulfilled.
- Time: $\mathcal{O}(\log n)$ since the trees have height $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.decrease-key(handle h):

- Decrease the key of the element pointed to by h.
- Bubble the element up in the tree until the heap property is fulfilled.
- Time: $\mathcal{O}(\log n)$ since the trees have height $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.decrease-key(handle h):

- Decrease the key of the element pointed to by h.
- Bubble the element up in the tree until the heap property is fulfilled.
- Time: $\mathcal{O}(\log n)$ since the trees have height $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.decrease-key(handle h):

- Decrease the key of the element pointed to by h.
- Bubble the element up in the tree until the heap property is fulfilled.
- Time: $\mathcal{O}(\log n)$ since the trees have height $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.delete(handle h):

- Execute S.decrease-key($h, -\infty$).
- Execute S.delete-min().
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.delete(handle h):

- Execute S.decrease-key$(h, -\infty)$.
- Execute S.delete-min().
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.delete(handle h):

- Execute S.decrease-key$(h, -\infty)$.
- Execute S.delete-min$()$.
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.delete(handle h):

- Execute S.decrease-key$(h, -\infty)$.
- Execute S.delete-min$()$.
- Time: $\mathcal{O}(\log n)$.