CSE 548: Analysis of Algorithms

Lecture 9
(Binomial Heaps)

Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Fall 2019

Mergeable Heap Operations

MAKE-HEAP($\boldsymbol{x}$): return a new heap containing only element x

INSERT($\boldsymbol{H}, \boldsymbol{x}$): insert element x into heap H

MINIMUM($\boldsymbol{H}$): return a pointer to an element in H containing the smallest key

EXTRACT-MIN($\boldsymbol{H}$): delete an element with the smallest key from H and return a pointer to that element

UNION($\boldsymbol{H_1}, \boldsymbol{H_2}$): return a new heap containing all elements of heaps H_1 and H_2, and destroy the input heaps

More mergeable heap operations:

DECREASE-KEY($\boldsymbol{H}, \boldsymbol{x}, \boldsymbol{k}$): change the key of element x of heap H to k assuming $k \leq$ the current key of x

DELETE($\boldsymbol{H}, \boldsymbol{x}$): delete element x from heap H

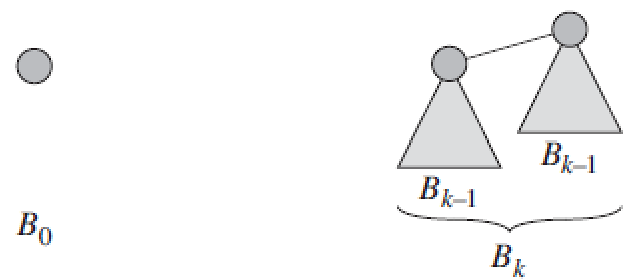
Mergeable Heap Operations

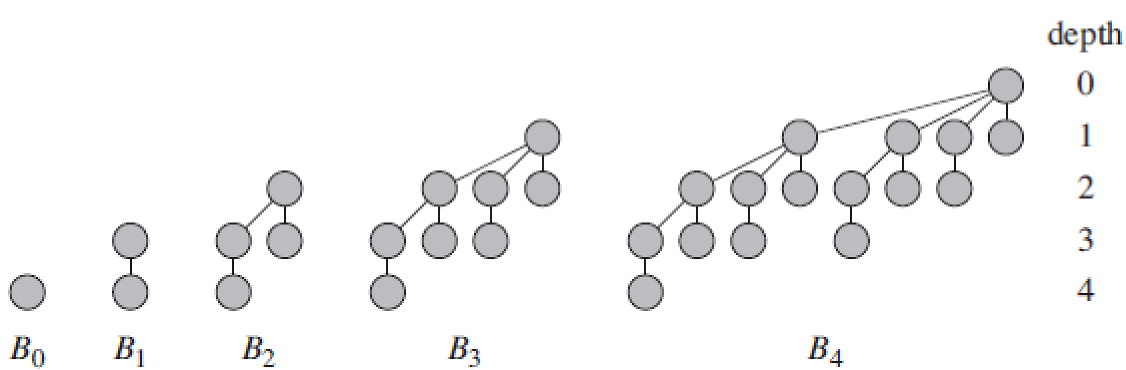
Heap Operation	Binary Heap (worst-case)	Binomial Heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$O(\log n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$O(\log n)$	$O(\log n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$O(\log n)$	—
DELETE	$O(\log n)$	—

Binomial Trees

A *binomial tree* B_k is an ordered tree defined recursively as follows.

- B_0 consists of a single node
- For $k > 0$, B_k consists of two B_{k-1}'s that are linked together so that the root of one is the left child of the root of the other

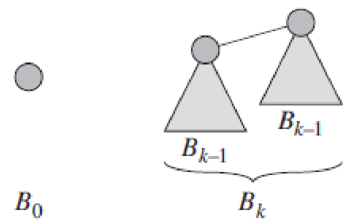


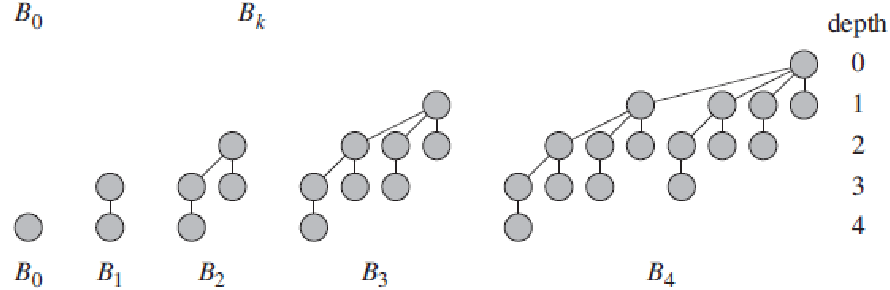


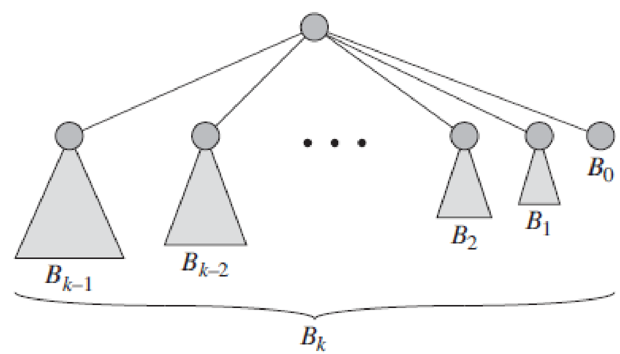
Binomial Trees

Some useful properties of B_k are as follows.

1. it has exactly 2^k nodes
2. its height is k
3. there are exactly $\binom{k}{i}$ nodes at depth $i = 0,1,2, \dots, k$
4. the root has degree k
5. if the children of the root are numbered from left to right by $k-1, k-2, \dots, 0$, then child i is the root of a B_i





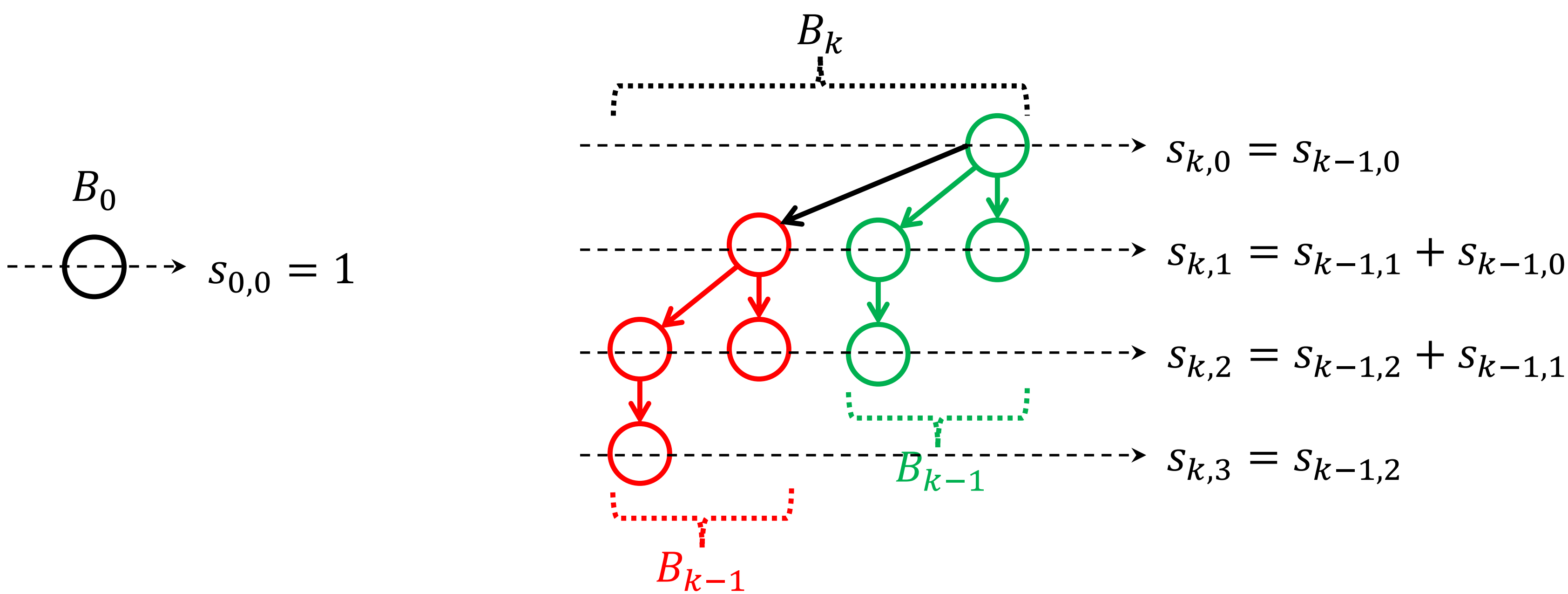


Binomial Trees

Prove: B_k has exactly $\binom{k}{i}$ nodes at depth $i = 0,1,2,\ldots,k$.

Proof: Suppose B_k has $s_{k,i}$ nodes at depth i.

$$s_{k,i} = \begin{cases} 0 & if\ i < 0\ or\ i > k, \\ 1 & if\ i = k = 0, \\ s_{k-1,i} + s_{k-1,i-1} & otherwise. \end{cases}$$



Binomial Trees

$$s_{k,i} = \begin{cases} 0 & if\ i < 0\ or\ i > k, \\ 1 & if\ i = k = 0, \\ s_{k-1,i} + s_{k-1,i-1} & otherwise. \end{cases}$$

$$\Rightarrow s_{k,i} = [k \geq i \geq 0]\big(s_{k-1,i} + s_{k-1,i-1} + [i = k = 0]\big)$$

Generating function: $S_k(z) = s_{k,0} + s_{k,1}z + s_{k,2}z^2 + \ \dots + s_{k,k}z^k$

$$S_{k\geq 0}(z) = \sum_{i=0}^{k} s_{k,i}z^i = \sum_{i=0}^{k} s_{k-1,i}z^i + \sum_{i=0}^{k} s_{k-1,i-1}z^i + [k = 0]\sum_{i=0}^{k} [i = 0]z^i$$

$$= \sum_{i=0}^{k-1} s_{k-1,i}z^i + z\sum_{i=0}^{k-1} s_{k-1,i}z^i + [k = 0]$$

$$= S_{k-1}(z) + zS_{k-1}(z) + [k = 0] = (1 + z)S_{k-1}(z) + [k = 0]$$

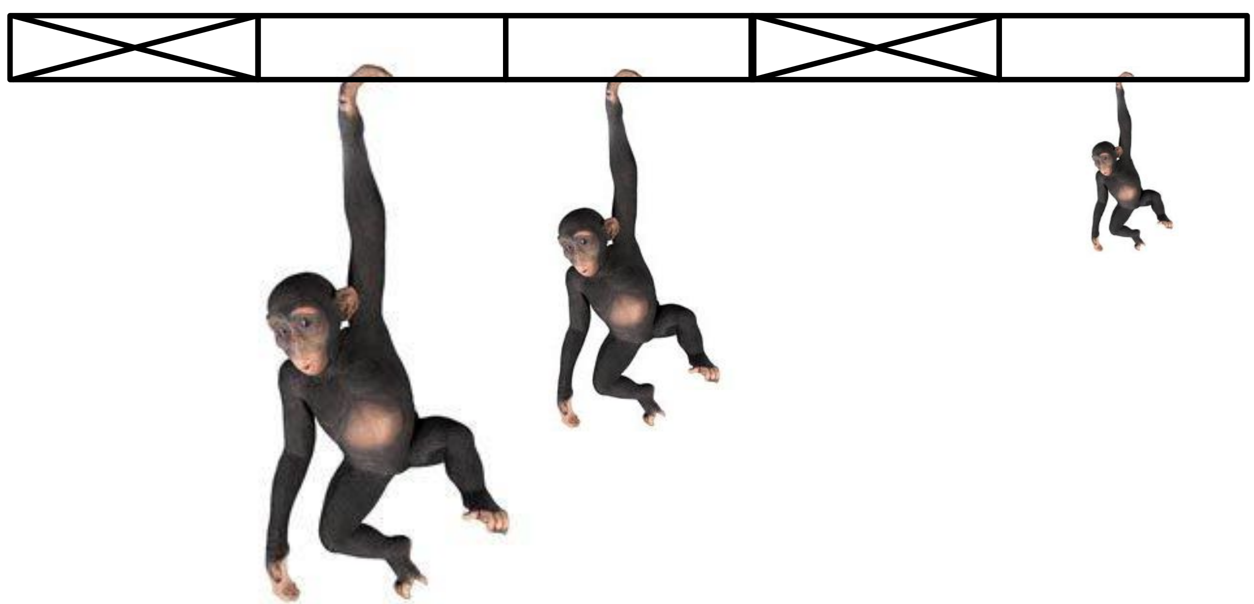
$$\Rightarrow S_k(z) = \begin{cases} 1 & if\ k = 0, \\ (1 + z)S_{k-1}(z) & otherwise. \end{cases}$$

$$= (1 + z)^k$$

Equating the coefficient of z^i from both sides: $s_{k,i} = \binom{k}{i}$

Binomial Heaps

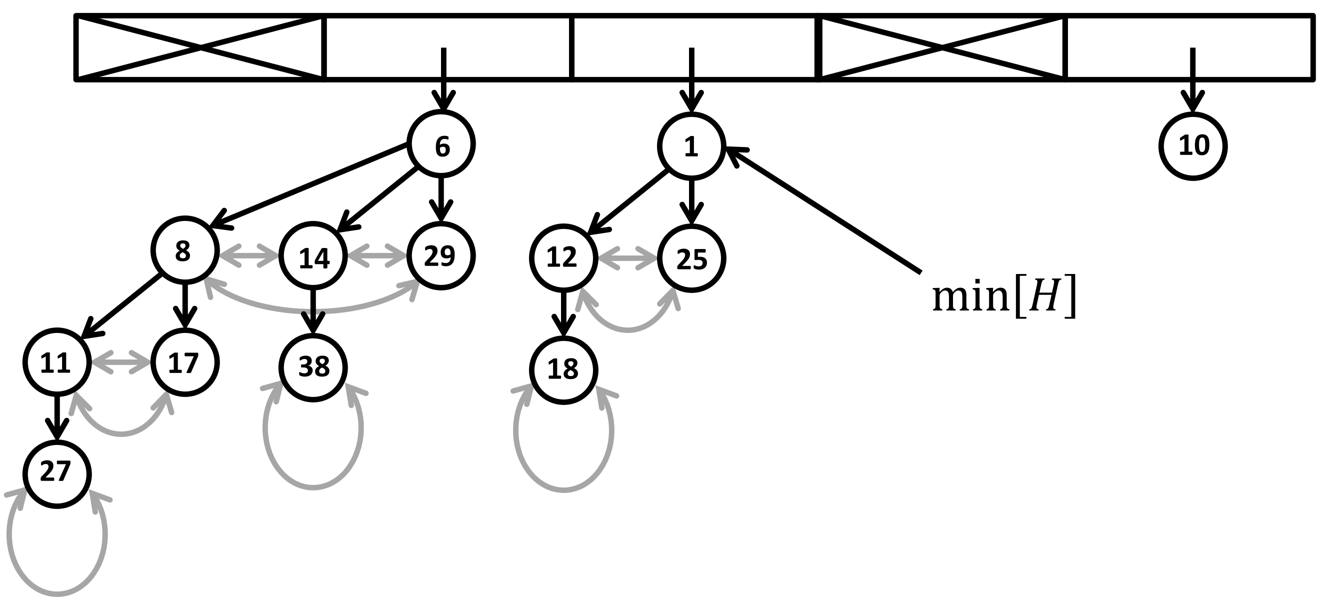
A *binomial heap* H is a set of binomial trees that satisfies the following properties:



Binomial Heaps

A *binomial heap* H is a set of binomial trees that satisfies the following properties:

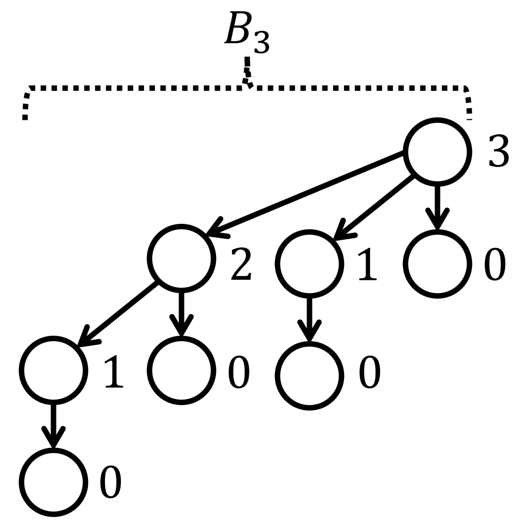
1. each node has a key
2. each binomial tree in H obeys the min-heap property
3. for any integer $k \geq 0$, there is at most one binomial tree in H whose root node has degree k



Rank of Binomial Trees

The *rank* of a binomial tree node x, denoted $rank(x)$, is the number of children of x.

The figure on the right shows the rank of each node in B_3.



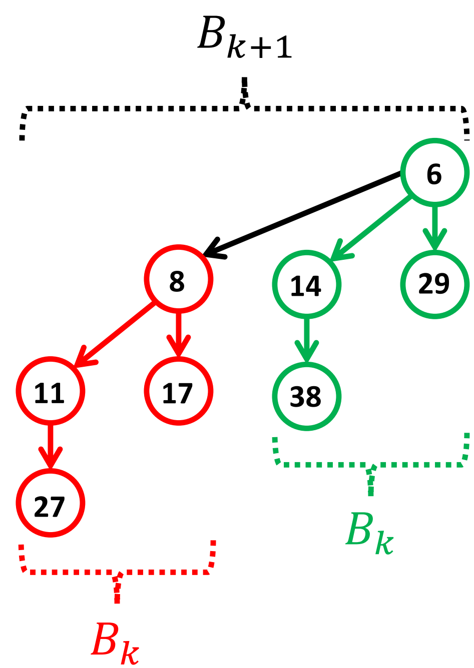
Observe that $rank\big(root(B_k)\big) = k$.

Rank of a binomial tree is the rank of its root. Hence,

$$rank(B_k) = rank\big(root(B_k)\big) = k$$

A Basic Operation: Linking Two Binomial Trees

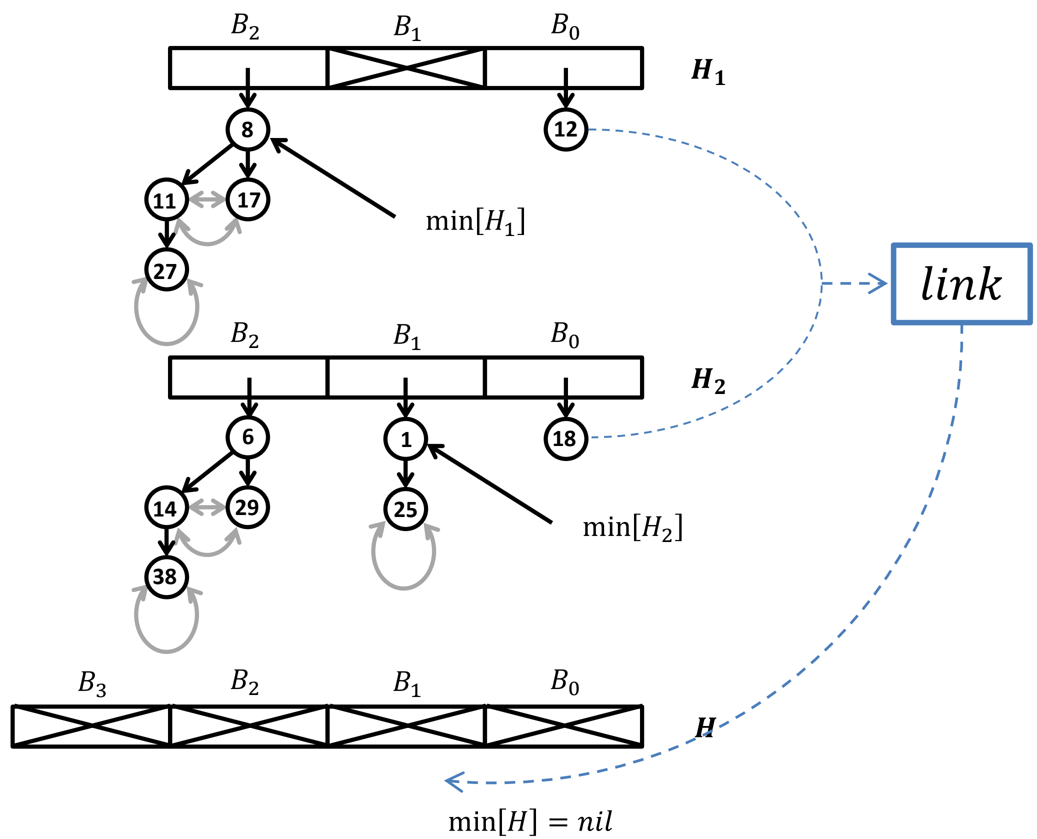
Given *two binomial trees of the same rank*, say, two B_k's, we link them in constant time by making the root of one tree the left child of the root of the other, and thus producing a B_{k+1}.



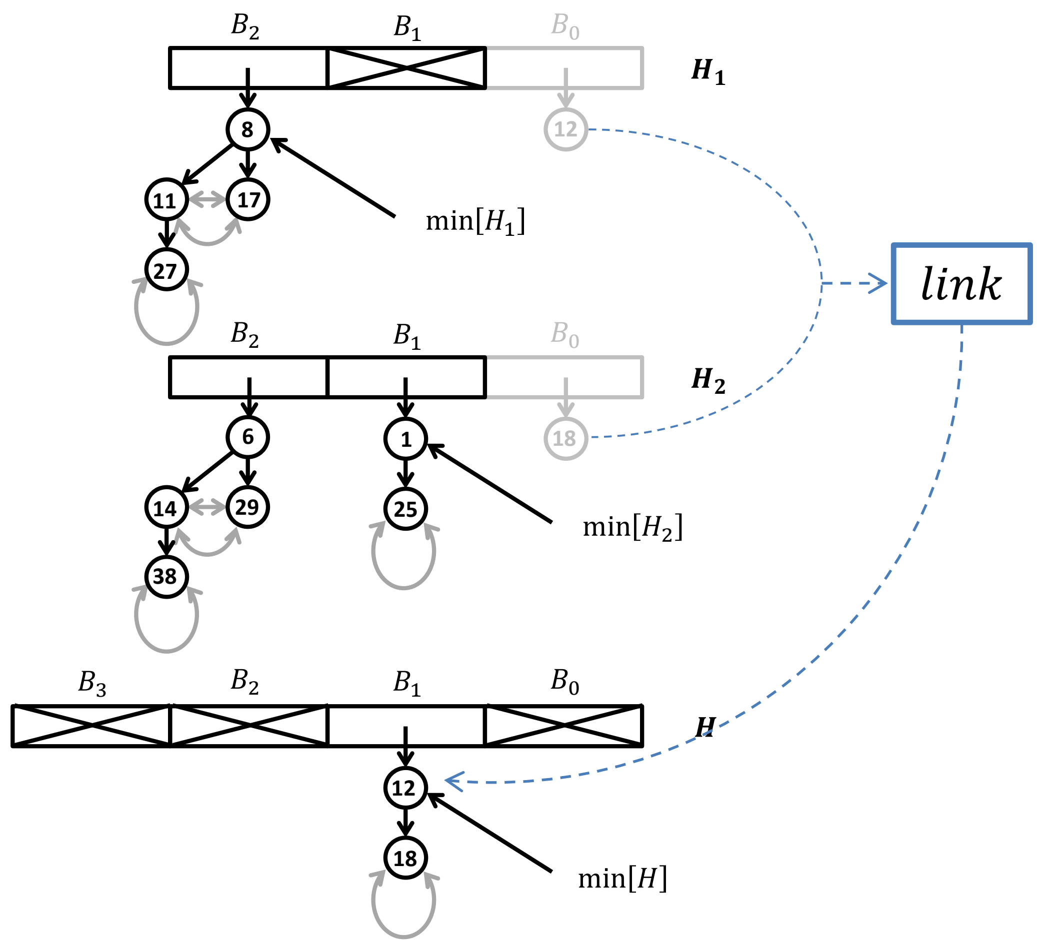
If the trees are part of a binomial min-heap, we always make the root with the smaller key the parent, and the one with the larger key the child.

Ties are broken arbitrarily.

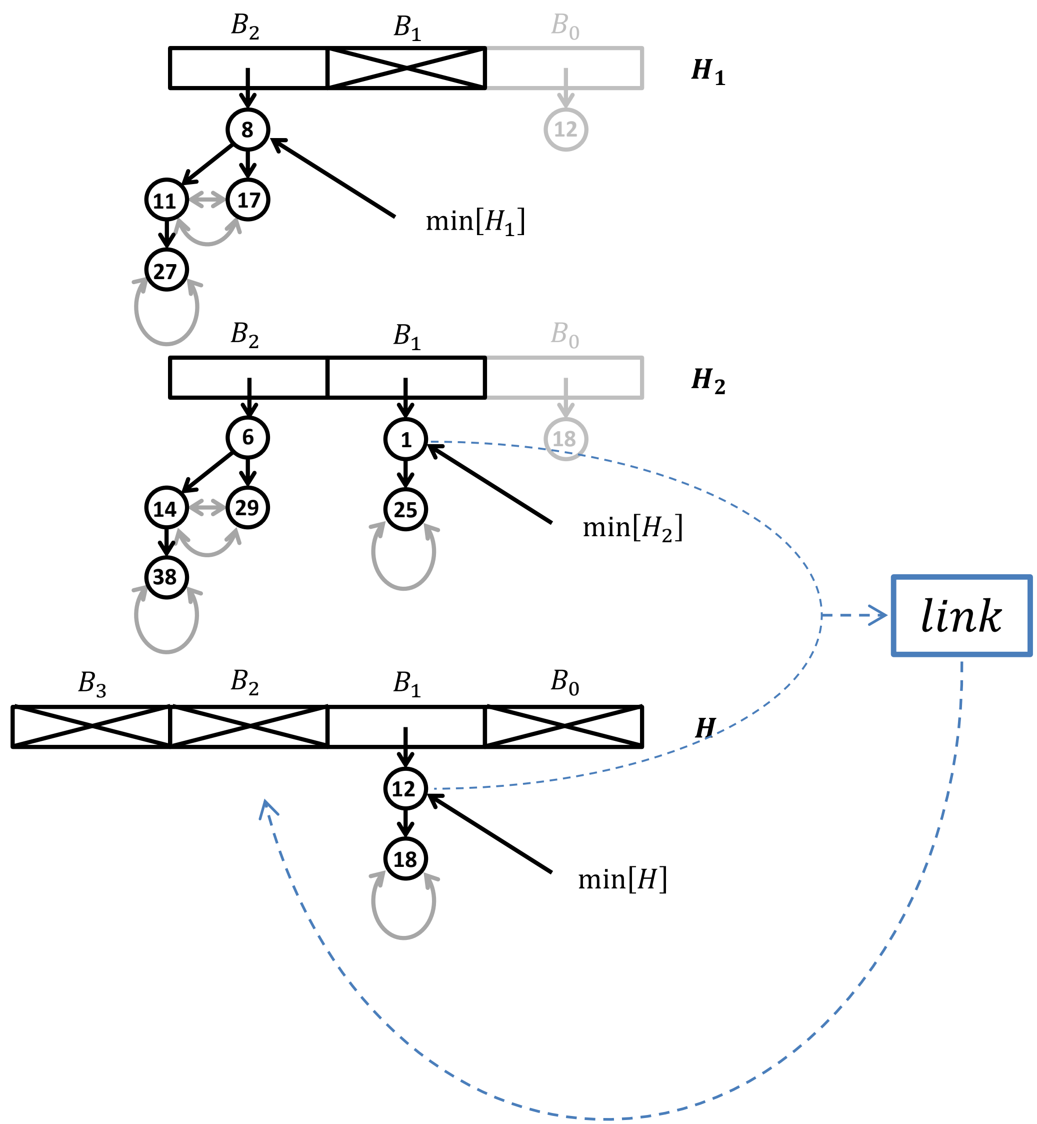
Binomial Heap Operations: UNION(H_1, H_2)



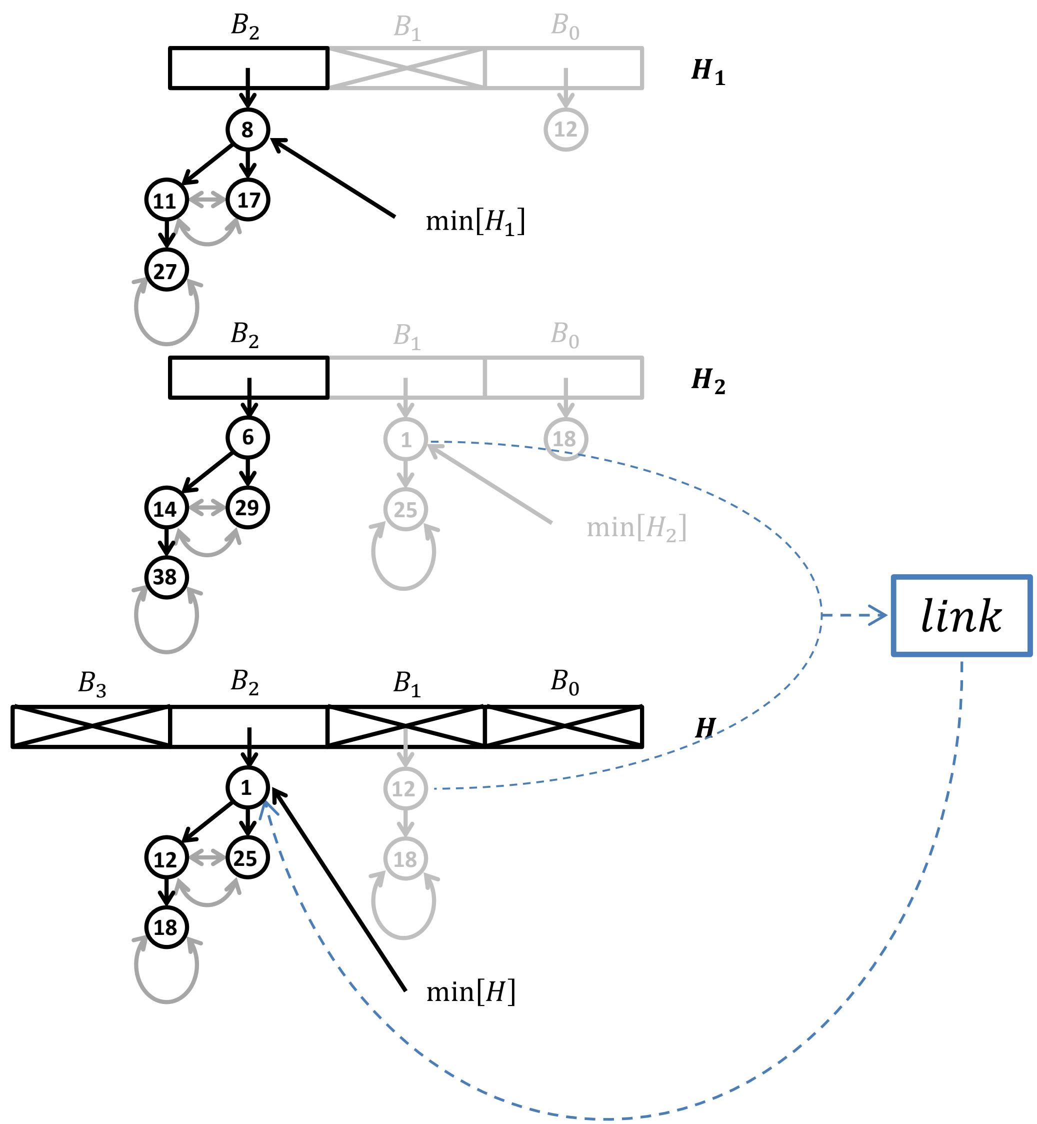
Binomial Heap Operations: UNION(H_1, H_2)



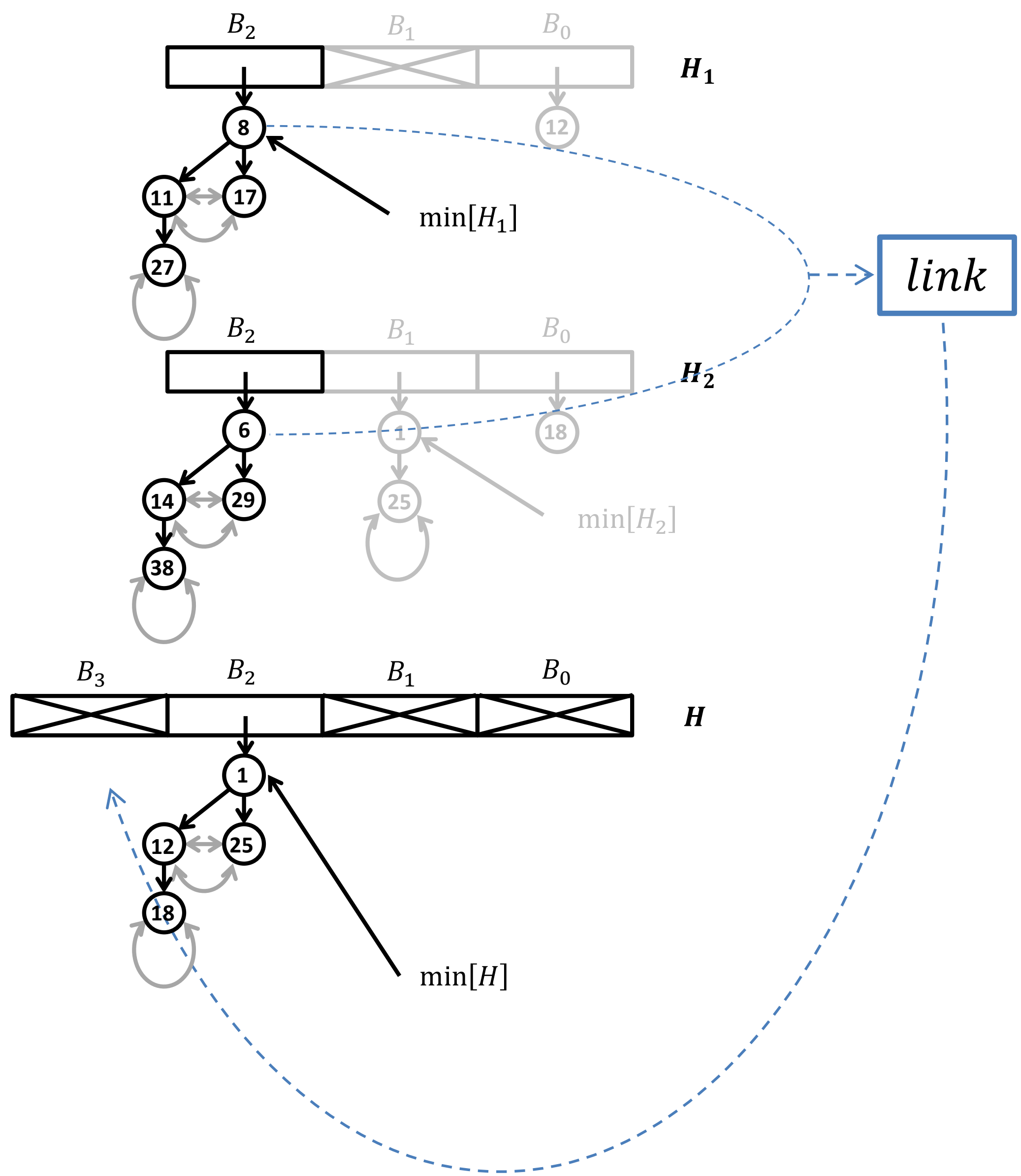
Binomial Heap Operations: UNION(H_1, H_2)



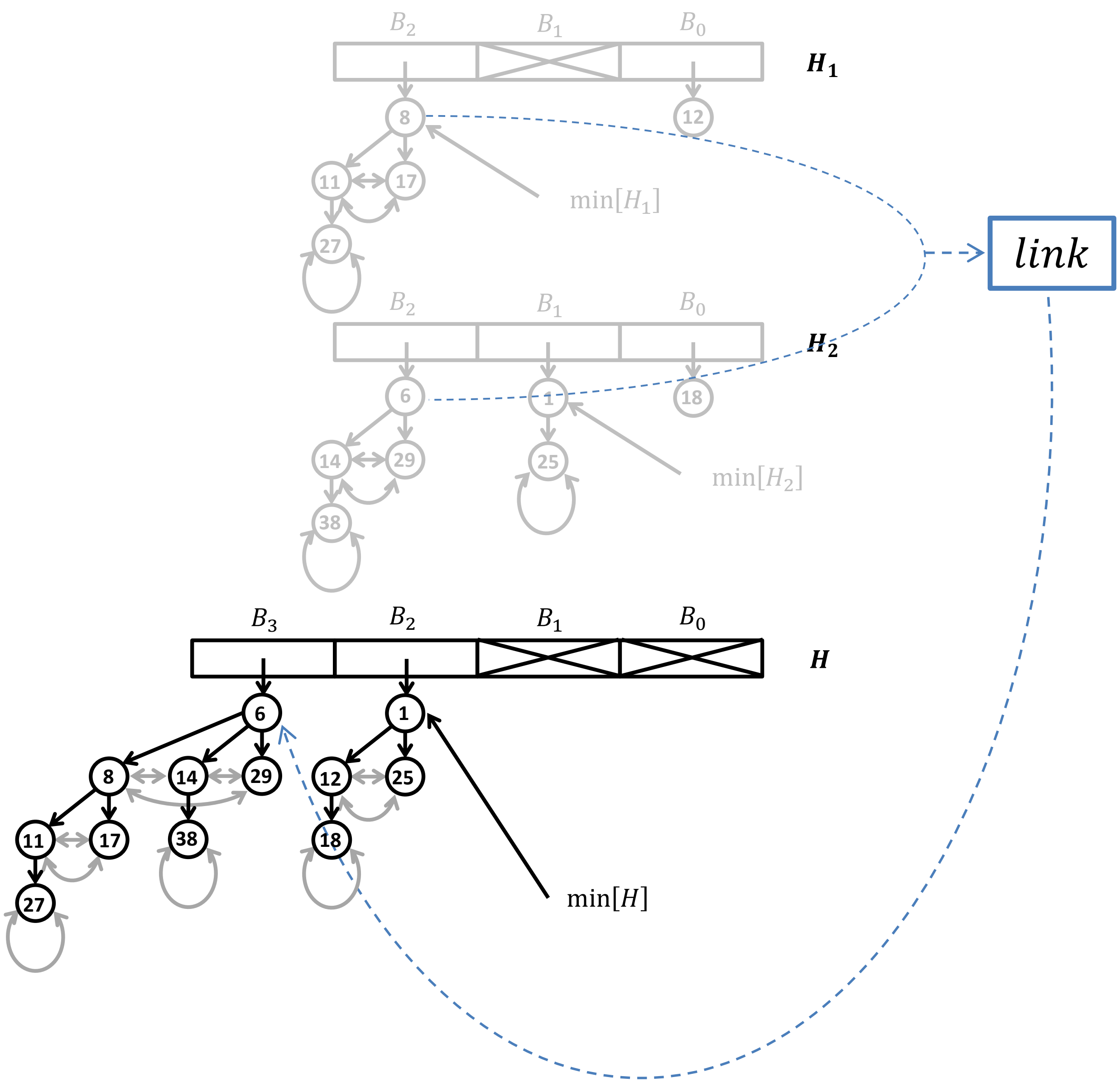
Binomial Heap Operations: UNION(H_1, H_2)



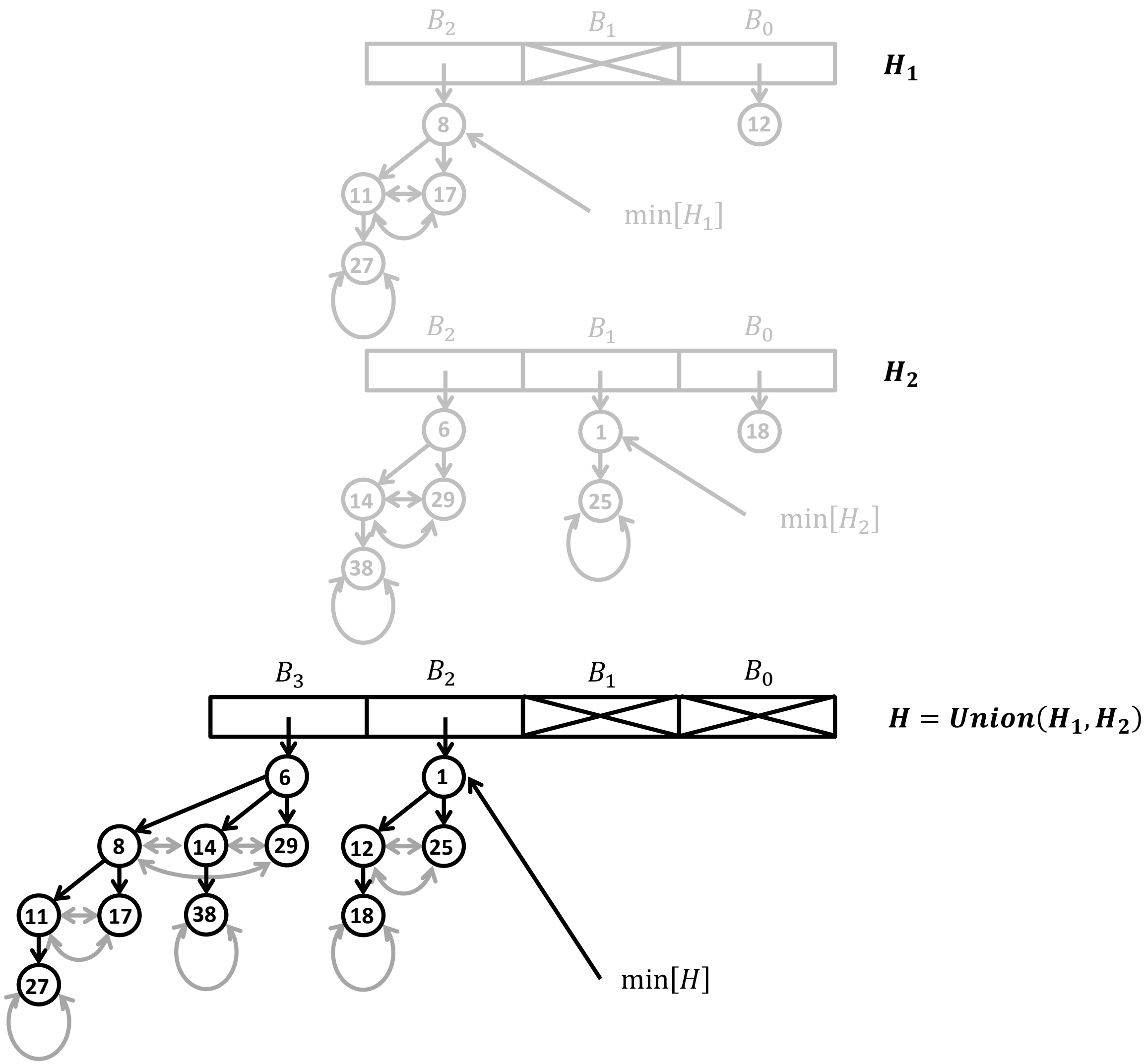
Binomial Heap Operations: Union(H_1, H_2)



Binomial Heap Operations: UNION(H_1, H_2)



Binomial Heap Operations: UNION(H_1, H_2)



Binomial Heap Operations: UNION(H_1, H_2)

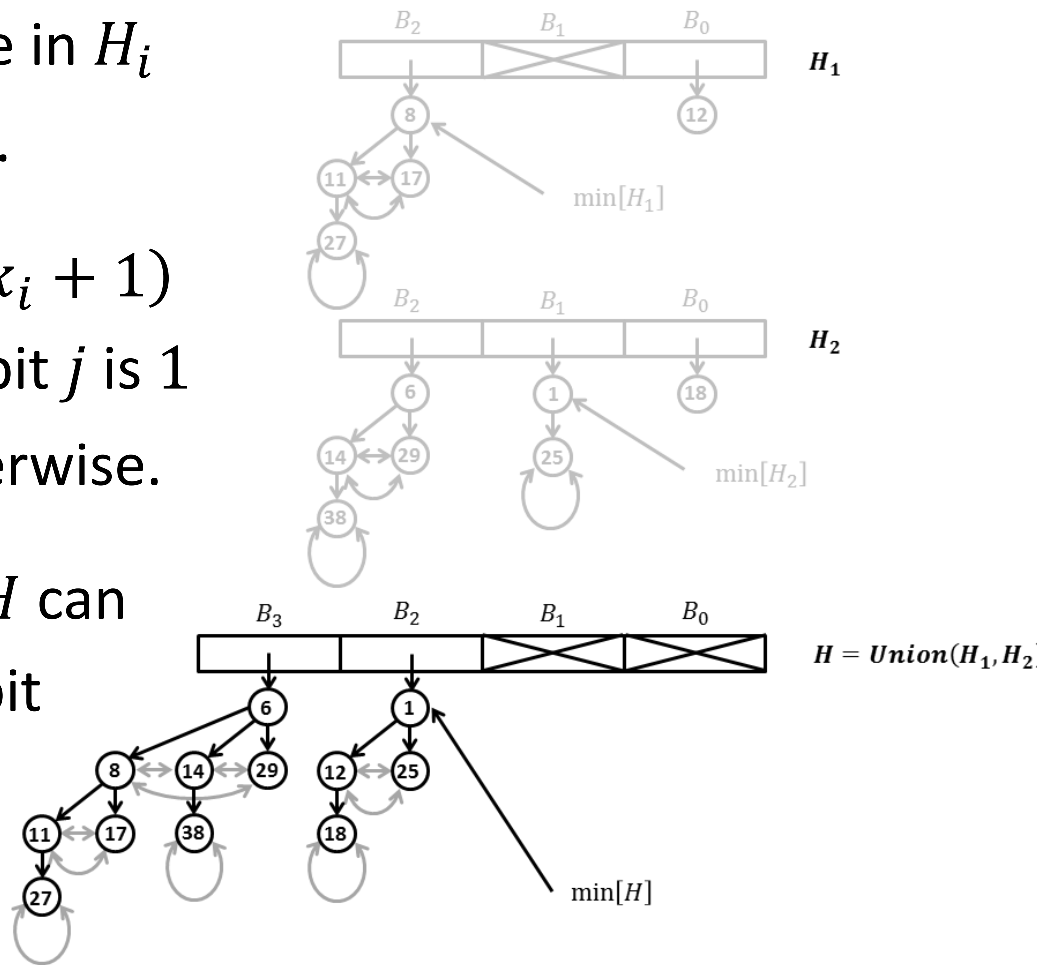
UNION(H_1, H_2) works in exactly the same way as binary addition.

Let n_i be the number of nodes in H_i $(i = 1,2)$.

Then the largest binomial tree in H_i is a B_{k_i}, where $k_i = \lfloor \log_2 n_i \rfloor$.

Thus H_i can be treated as a $(k_i + 1)$ bit binary number x_i, where bit j is 1 if H_i contains a B_j, and 0 otherwise.

If $H = Union(H_1, H_2)$, then H can be viewed as a $k = \lfloor \log_2 n \rfloor$ bit binary number $x = x_1 + x_2$, where $n = n_1 + n_2$.



Binomial Heap Operations: UNION(H_1, H_2)

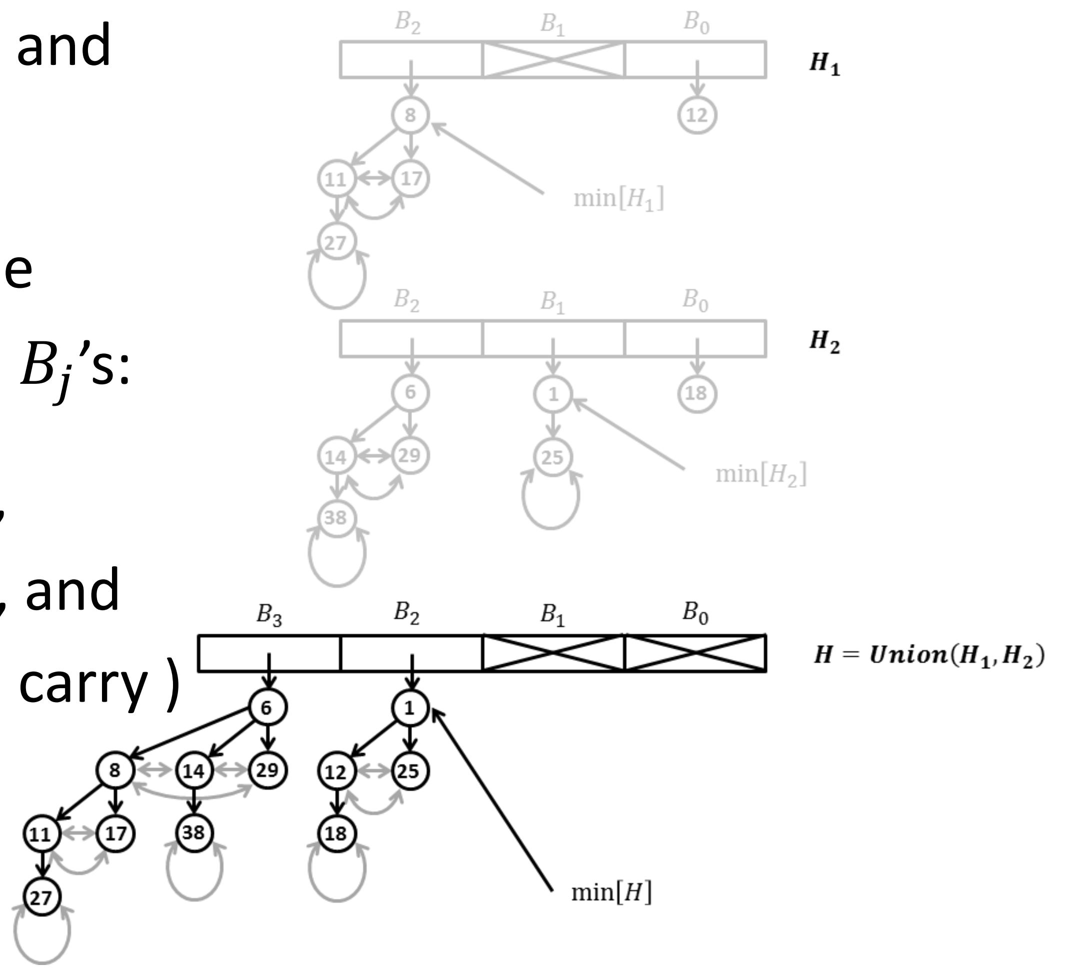
UNION(H_1, H_2) works in exactly the same way as binary addition.

Initially, H does not contain any binomial trees.

Melding starts from B_0 (LSB) and continues up to B_k (MSB).

At each location $j \in [0, k]$, one encounters at most three (3) B_j's:

- at most 1 from H_1 (input),
- at most 1 from H_2 (input), and
- if $j > 0$, at most 1 from H (carry)

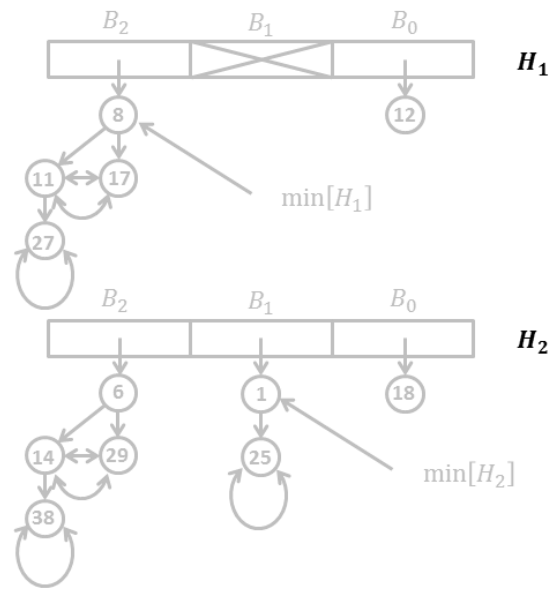


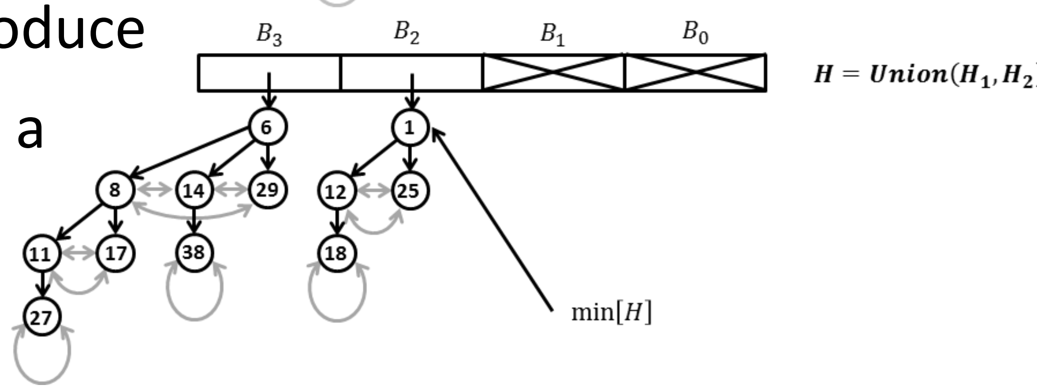
Binomial Heap Operations: UNION(H_1, H_2)

UNION(H_1, H_2) works in exactly the same way as binary addition.

When the number of B_j's at location $j \in [0, k]$ is:

- 0: location j of H is set to nil
- 1: location j of H points to that B_j
- 2: the two B_j's are linked to produce a B_{j+1} which is stored as a carry at location $j+1$ of H, and location j is set to nil
- 3: two B_j's are linked to produce a B_{j+1} which is stored as a carry at location $j+1$ of H, and the 3rd B_j is stored at location j





Binomial Heap Operations: UNION(H_1, H_2)

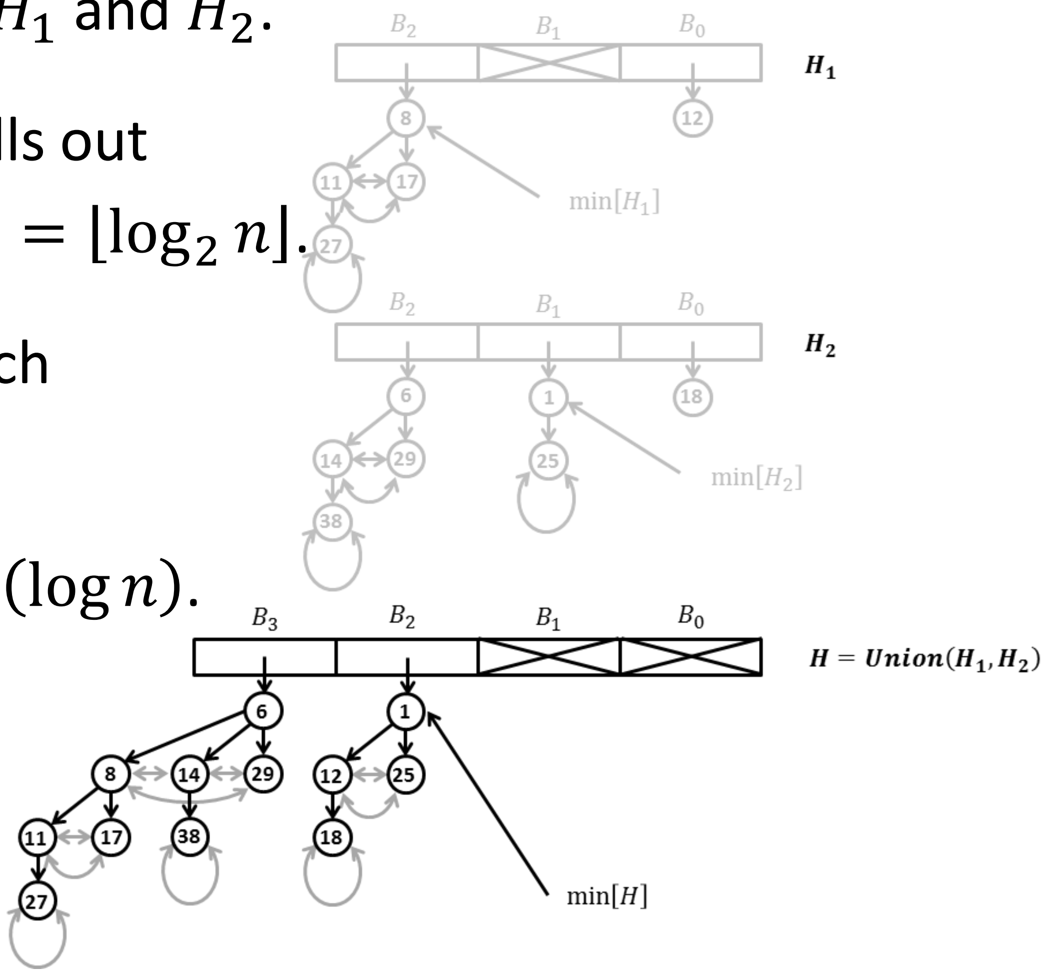
UNION(H_1, H_2) works in exactly the same way as binary addition.

Worst case cost of UNION(H_1, H_2) is clearly $\Theta(\log n)$, where n is the total number of nodes in H_1 and H_2.

Observe that this operation fills out $k + 1$ locations of H, where $k = \lfloor \log_2 n \rfloor$.

It does only $\Theta(1)$ work for each location.

Hence, total cost is $\Theta(k) = \Theta(\log n)$.



Binomial Heap Operations: UNION(H_1, H_2)

One can improve the performance of UNION(H_1, H_2) as follows.

W.l.o.g., suppose H_2 is at least as large as H_1, i.e., $n_2 \geq n_1$.

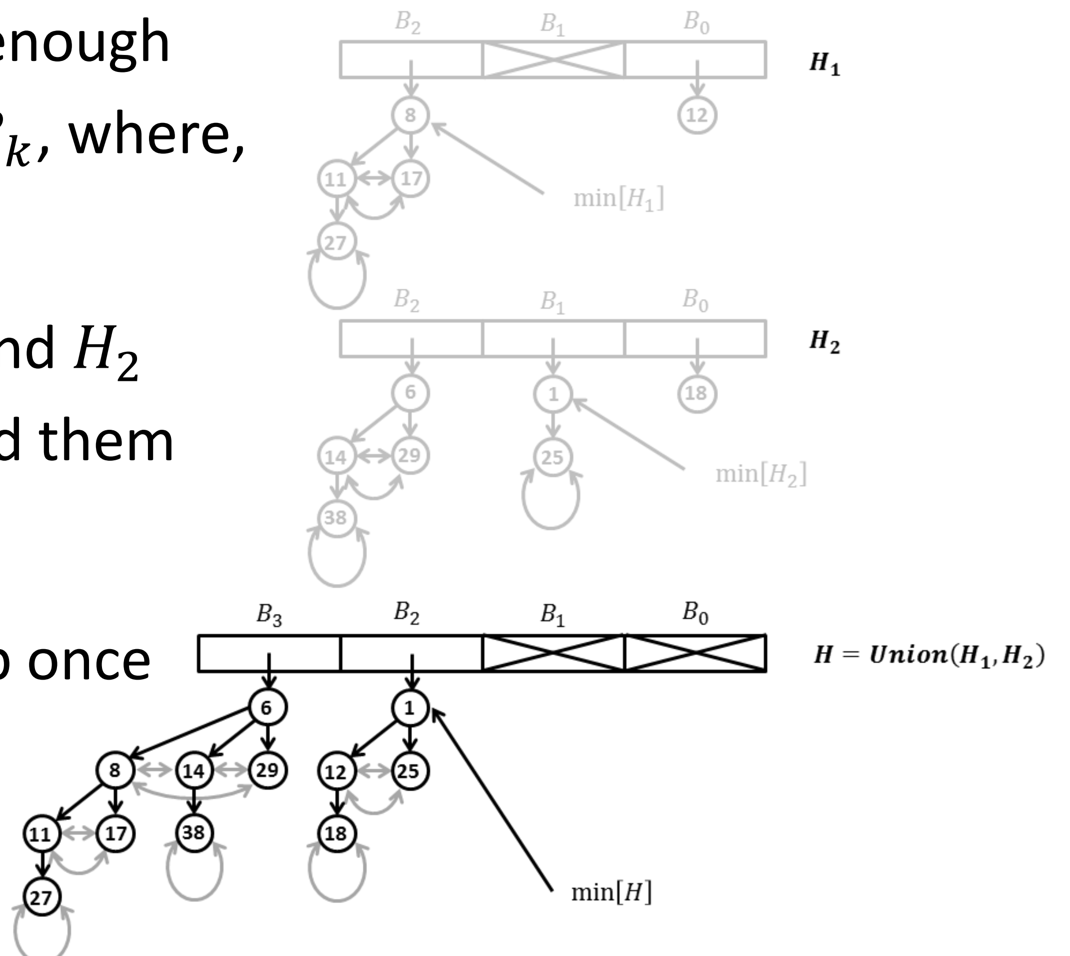
We also assume that H_2 has enough space to store at least up to B_k, where, $k = \lfloor \log_2(n_1 + n_2) \rfloor$.

Then instead of melding H_1 and H_2 to a new heap H, we can meld them in-place at H_2.

After melding till B_{k_1}, we stop once the carry stops propagating.

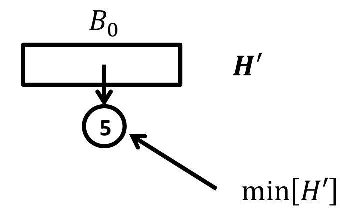
The cost is $\Omega(k_1)$, but $O(k_2)$.

Worst-case cost is still $O(k) = O(\log n)$.



Binomial Heap Operations: INSERT(H, x)

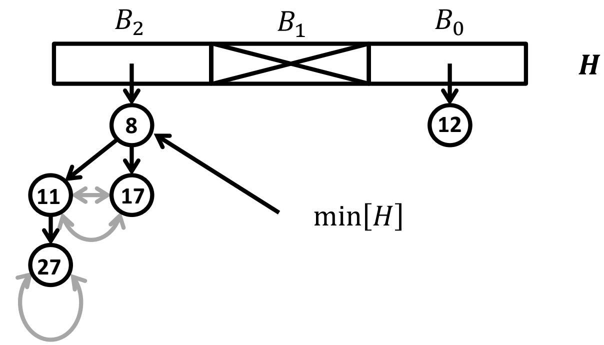
Step 1: $H' \leftarrow$ MAKE-HEAP(x)



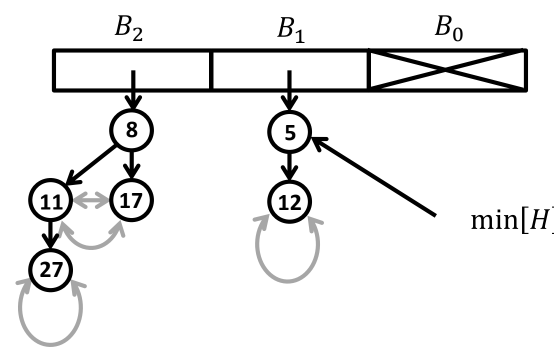
Takes $\Theta(1)$ time.

Step 2: $H \leftarrow$ UNION(H, H')

(in-place at H)



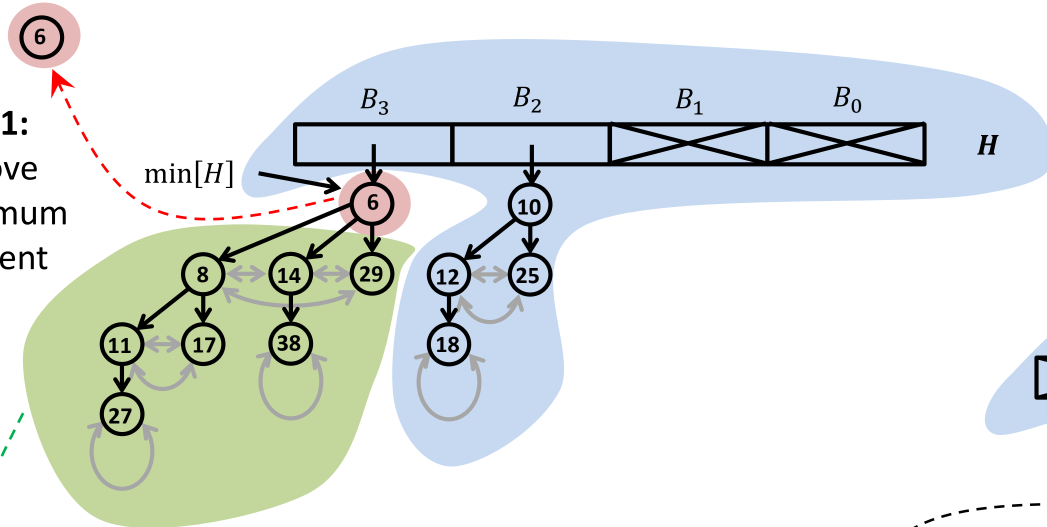
Takes $\mathrm{O}(\log n)$ time, where n is the number of nodes in H.



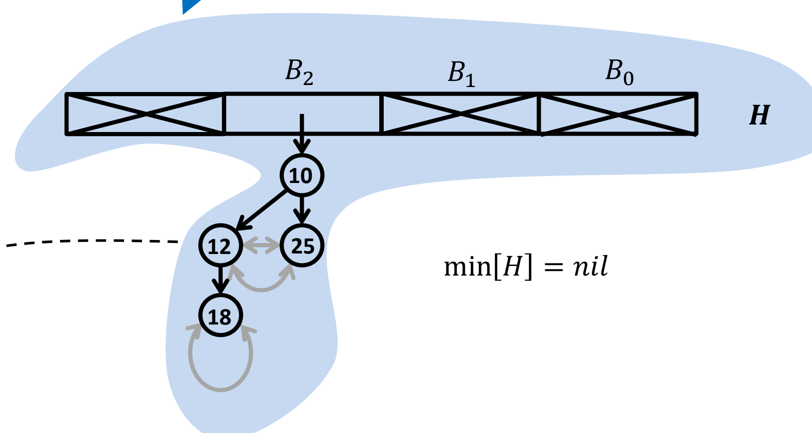
Thus the worst-case cost of INSERT(H, x) is $\mathrm{O}(\log n)$, where n is the number of items already in the heap.

Binomial Heap Operations: EXTRACT-MIN(H)

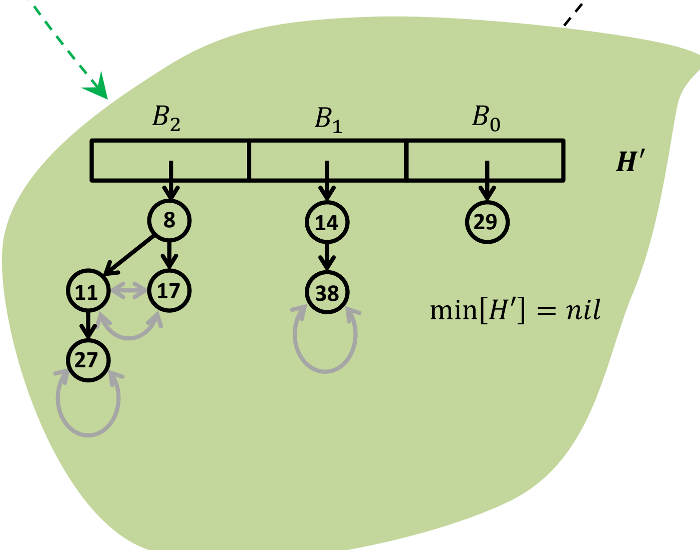
Step 1: remove minimum element



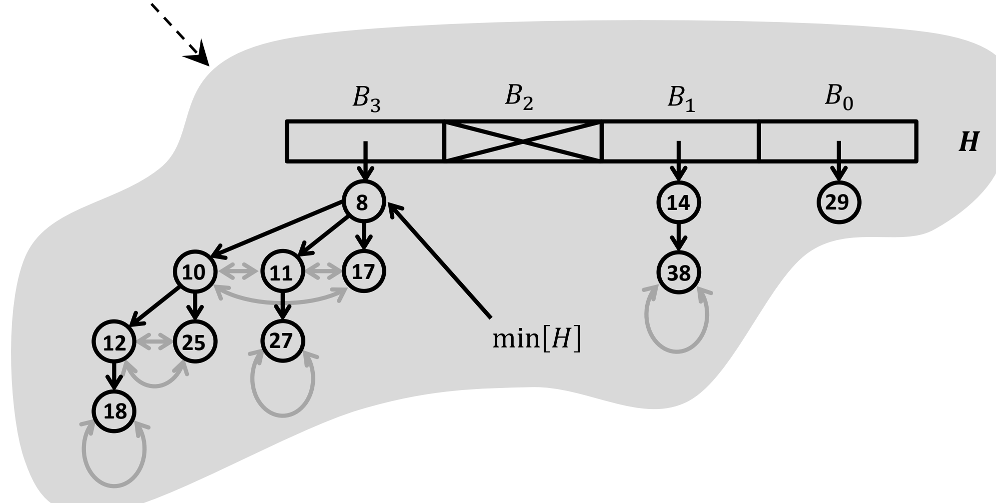
Step 2: remove the binomial tree with the smallest root from the input heap



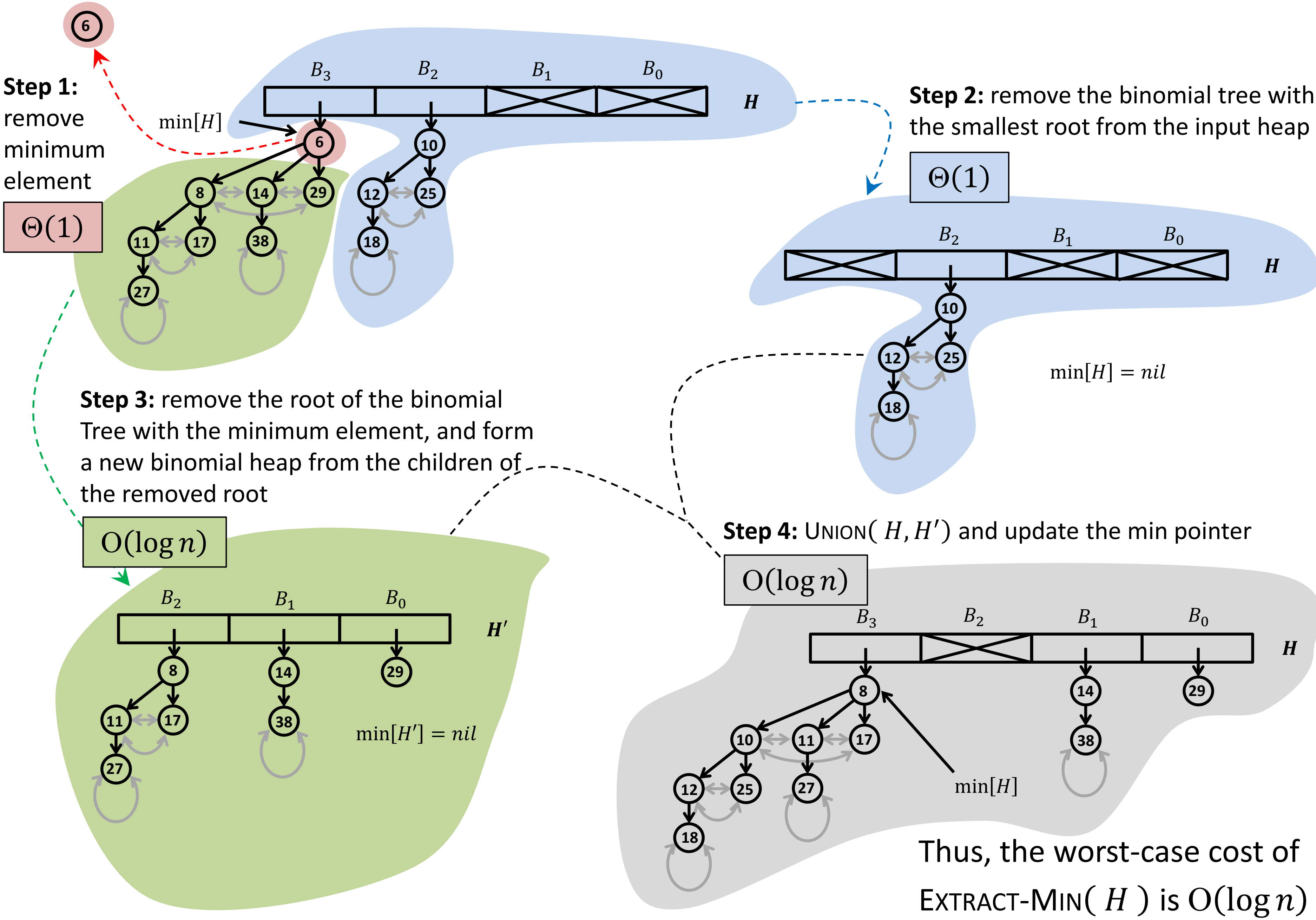
Step 3: remove the root of the binomial tree with the minimum element, and form a new binomial heap from the children of the removed root



Step 4: UNION(H, H') and update the min pointer



Binomial Heap Operations: EXTRACT-MIN(H)



Binomial Heap Operations

Heap Operation	Worst-case
MAKE-HEAP	$\Theta(1)$
INSERT	$O(\log n)$
MINIMUM	$\Theta(1)$
EXTRACT-MIN	$O(\log n)$
UNION	$O(\log n)$

Amortized Analysis (Accounting Method)

We maintain a credit account for every tree in the heap, and always maintain the following invariant:

$$\bigwedge_{B_j \in H} credit(B_j) = 1$$

MAKE-HEAP(x):

actual cost, $c_i = 1$ (for creating the singleton heap)

extra charge, $\delta_i = 1$ (for storing in the credit account of the new tree)

amortized cost, $\hat{c}_i = c_i + \delta_i = 2 = \Theta(1)$

Amortized Analysis (Accounting Method)

We maintain a credit account for every tree in the heap, and always maintain the following invariant:

$$\boxed{\bigwedge_{B_j \in H} credit(B_j) = 1}$$

LINK($\boldsymbol{B_k^{(1)}}, \boldsymbol{B_k^{(2)}}$):

actual cost, $c_i = 1$ (for linking the two trees)

We use $credit\left(B_k^{(1)}\right)$ pay for this actual work.

Let B_{k+1} be the newly created tree. We restore the credit invariant by transferring $credit\left(B_k^{(2)}\right)$ to $credit(B_{k+1})$.

Hence, amortized cost, $\hat{c}_i = c_i + \delta_i = 1 - 1 = 0$

Amortized Analysis (Accounting Method)

We maintain a credit account for every tree in the heap, and always maintain the following invariant:

$$\boxed{\bigwedge_{B_j \in H} credit(B_j) = 1}$$

INSERT($\boldsymbol{H}, \boldsymbol{x}$):

Amortized cost of MAKE-HEAP(x) is $= 2$

Then UNION(H, H') is simply a sequence of free LINK operations with only a constant amount of additional work that do not create any new trees. Thus the credit invariant is maintained, and the amortized cost of this step is $= 1$.

Hence, amortized cost of INSERT, $\hat{c}_i = 2 + 1 = 3 = \Theta(1)$

Amortized Analysis (Accounting Method)

We maintain a credit account for every tree in the heap, and always maintain the following invariant:

$$\boxed{\bigwedge_{B_j \in H} credit(B_j) = 1}$$

UNION(H_1, H_2):

UNION(H_1, H_2) includes a sequence of free LINK operations that maintain the credit invariant.

But it also includes $\mathrm{O}(\log n)$ other operations that are not free (e.g., consider melding a heap with $n = 2^k$ elements with one containing $n - 1$ elements). These operations do not create new trees (and so do not violate the credit invariant), and each cost $\Theta(1)$.

Hence, amortized cost of UNION, $\hat{c}_i = \mathrm{O}(\log n)$

Amortized Analysis (Accounting Method)

We maintain a credit account for every tree in the heap, and always maintain the following invariant:

$$\boxed{\bigwedge_{B_j \in H} credit(B_j) = 1}$$

EXTRACT-MIN(H):

Steps 1 & 2: The $\Theta(1)$ actual cost is paid for by the credit released by the deleted tree.

Step 3: Exposes $\mathrm{O}(\log n)$ new trees, and we charge 1 unit of extra credit for storing in the credit account of each such tree.

Step 4: Performs a UNION that has $\mathrm{O}(\log n)$ amortized cost.

Hence, amortized cost of EXTRACT-MIN, $\hat{c}_i = \mathrm{O}(\log n)$

Amortized Analysis (Potential Method)

Potential Function,

$\boxed{\Phi(D_i) = c \times \text{(\#trees in the data structure after the } i\text{-th operation),}}$

where c is a constant.

Clearly, $\Phi(D_0) = 0$ (no trees in the data structure initially)

and for all $i > 0$, $\Phi(D_i) \geq 0$ (#trees cannot be negative)

MAKE-HEAP(x):

actual cost, $c_i = 1$ (for creating the singleton heap)

potential change, $\Delta_i = \Phi(D_i) - \Phi(D_{i-1}) = c$

(as #trees increases by 1)

amortized cost, $\hat{c}_i = c_i + \Delta_i = 1 + c = \Theta(1)$

Amortized Analysis (Potential Method)

Potential Function,

$\Phi(D_i) = c \times$ (#trees in the data structure after the i-th operation),

where c is a constant.

INSERT(H, x):

The number of trees increases by 1 initially.

Then the operation scans $k > 0$ (say) locations of the array of tree pointers. Observe that we use tree linking $(k-1)$ times each of which reduces the number of trees by 1.

actual cost, $c_i = 1 + k$

potential change, $\Delta_i = \Phi(D_i) - \Phi(D_{i-1}) = c(1 - (k-1))$

$$= c - c(k-1)$$

amortized cost, $\hat{c}_i = c_i + \Delta_i = 2 + c - (c-1)(k-1)$

For $c \geq 1$, we have, $\hat{c}_i \leq 2 + c = \Theta(1)$

Amortized Analysis (Potential Method)

Potential Function,

$\boxed{\Phi(D_i) = c \times (\text{ \#trees in the data structure after the } i\text{-th operation }),}$

where c is a constant.

UNION($\boldsymbol{H_1, H_2}$):

Suppose the operation scans $k > 0$ locations of the array of tree pointers, and uses the link operation l times. Observe that $k > l \geq 0$. Each link reduces the number of trees by 1.

actual cost, $c_i = k$

potential change, $\Delta_i = \Phi(D_i) - \Phi(D_{i-1}) = -c \times l$

amortized cost, $\hat{c}_i = c_i + \Delta_i = k - c \times l$

Since $k = \mathrm{O}(\log n)$ and $l = \mathrm{O}(\log n)$, we have,

$\hat{c}_i = \mathrm{O}(\log n)$ for any c.

Amortized Analysis (Potential Method)

Potential Function,

$\Phi(D_i) = c \times$ (#trees in the data structure after the i-th operation),

where c is a constant.

EXTRACT-MIN(H):

Let in Step 1: $r =$ rank of the tree with the smallest key

and in Step 4: $k =$ #locations of pointer array scanned during UNION

$l =$ #link operations during UNION

$t =$ #trees in the heap after the UNION

Then actual cost, $c_i = 1 \text{ (step 1)} + 1 \text{ (step 2)} + r \text{ (step 3)}$

$+ k \text{ (step 4: union)} + t \text{ (step 4: update } min \text{ ptr)}$

$= 2 + k + t + r$

Amortized Analysis (Potential Method)

Potential Function,

$\Phi(D_i) = c \times$ (#trees in the data structure after the i-th operation),

where c is a constant.

EXTRACT-MIN(H):

Let in Step 1: $r =$ rank of the tree with the smallest key

and in Step 4: $k =$ #locations of pointer array scanned during UNION

$l =$ #link operations during UNION

$t =$ #trees in the heap after the UNION

potential change, $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

$= c \times (r - 1)$ (removing min element in step 1 removes 1 tree but creates r new ones)

$-c \times l$ (linkings in step 4 reduces #trees by l)

Amortized Analysis (Potential Method)

Potential Function,

$\Phi(D_i) = c \times$ (#trees in the data structure after the i-th operation),

where c is a constant.

EXTRACT-MIN(H):

Let in Step 1: $r =$ rank of the tree with the smallest key

and in Step 4: $k =$ #locations of pointer array scanned during UNION

$l =$ #link operations during UNION

$t =$ #trees in the heap after the UNION

actual cost, $c_i = 2 + k + t + r$

potential change, $\Delta_i = \Phi(D_i) - \Phi(D_{i-1}) = c \times (r - l - 1)$

Then amortized cost, $\hat{c}_i = c_i + \Delta_i = 2 + k + t + r + c \times (r - l - 1)$

Since $k = \mathrm{O}(\log n)$, $l = \mathrm{O}(\log n)$, $t = \mathrm{O}(\log n)$ & $r = \mathrm{O}(\log n)$,

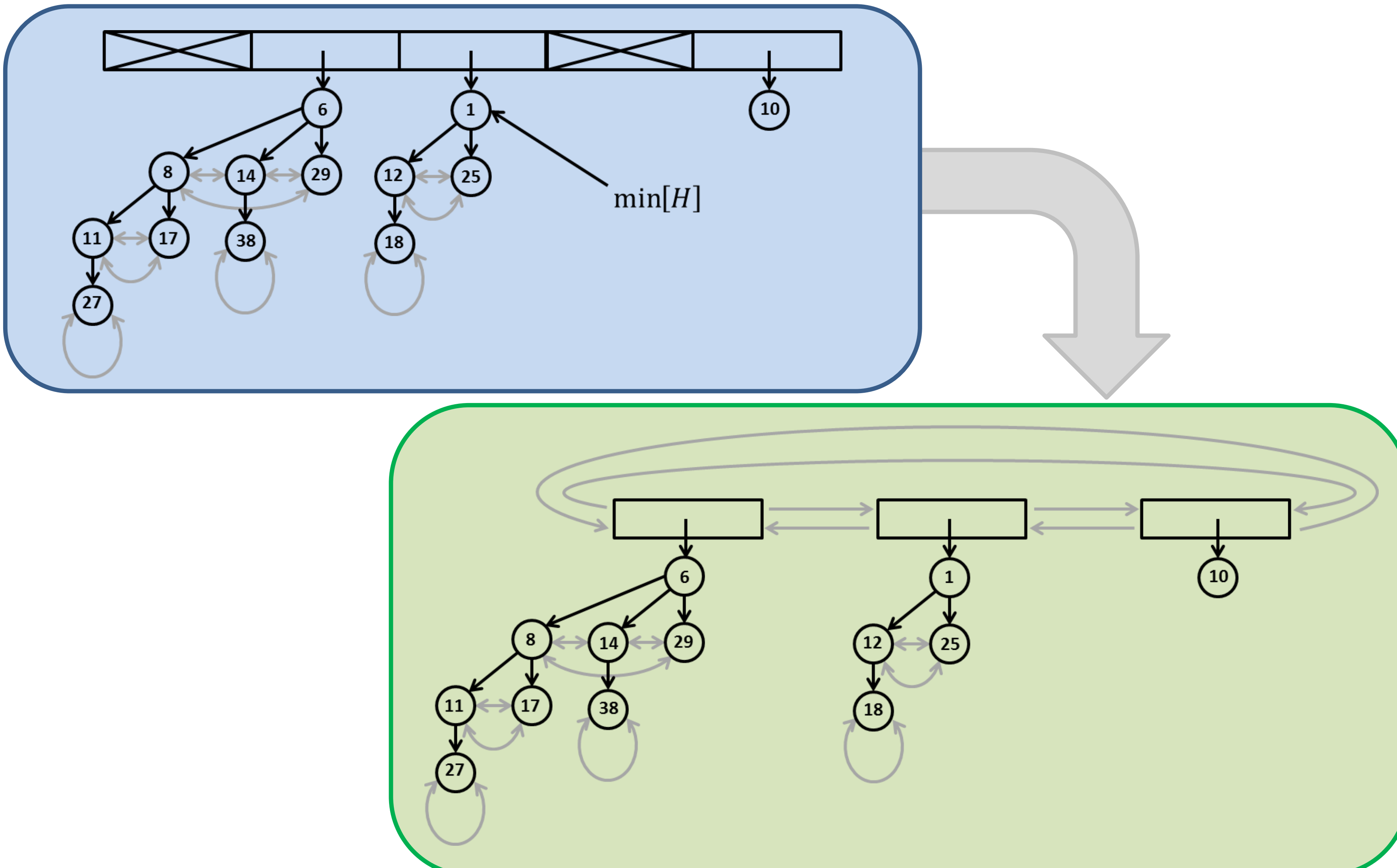
we have, $\hat{c}_i = \mathrm{O}(\log n)$ for any c.

Binomial Heap Operations

Heap Operation	Worst-case	Amortized
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$O(\log n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$O(\log n)$	$O(\log n)$
UNION	$O(\log n)$	$O(\log n)$

Binomial Heaps with Lazy Union

We maintain pointers to the trees in a doubly linked circular list (instead of an array), but do not maintain a min pointer.



Binomial Heap Operations with Lazy Union

We maintain the following invariant: $\boxed{\bigwedge_{B_j \in H} credit(B_j) = 2}$

Make-Heap(x): Create a singleton heap as before. Hence, amortized cost $= \Theta(1)$.

Link($B_k^{(1)}, B_k^{(2)}$): The two input trees have 4 units of saved credits of which 1 unit will be used to pay for the actual cost of linking, and 2 units will be saved as credit for the newly created tree. So, linking is still free, and it has 1 unused credit that can be used to pay for additional work if necessary.

Union(H_1, H_2): Simply concatenate the two root lists into one, and update the min pointer. Clearly, amortized cost $= \Theta(1)$.

Insert(H, x): This is Make-Heap followed by a Union. Hence, amortized cost $= \Theta(1)$.

Binomial Heap Operations with Lazy Union

We maintain the following invariant: $$\bigwedge_{B_j \in H} credit(B_j) = 2$$

EXTRACT-MIN(H): Unlike in the array version, in this case we may have several trees of the same rank.

We create an array of length $\lfloor \log_2 n \rfloor + 1$ with each location containing a nil pointer. We use this array to transform the linked list version to array version.

We go through the list of trees of H, inserting them one by one into the array, and linking and carrying if necessary so that finally we have at most one tree of each rank. We also create a min pointer.

We now perform EXTRACT-MIN as in the array case.

Finally, we collect the nonempty trees from the array into a doubly linked list, and return.

Binomial Heap Operations with Lazy Union

We maintain the following invariant: $\boxed{\bigwedge_{B_j \in H} credit(B_j) = 2}$

EXTRACT-MIN(H): We only need to show that converting from linked list version to array version takes $\mathrm{O}(\log n)$ amortized time.

Suppose we start with t trees, and perform l links. So, we spend $\mathrm{O}(t + l)$ time overall.

As each link decreases the number of trees by 1, after l links we end up with $t - l$ trees. Since at that point we have at most one tree of each rank, we have $t - l \le \lfloor \log_2 n \rfloor + 1$.

Thus $t + l = 2l + (t - l) = \mathrm{O}(l + \log n)$.

The $\mathrm{O}(l)$ part can be paid for by the l extra credits from l links.

We only charge the $\mathrm{O}(\log n)$ part to EXTRACT-MIN.

Binomial Heap Operations with Lazy Union

We use exactly the same potential function as in the previous version,

$\Phi(D_i) = c \times$ (#trees in the data structure after the i-th operation),

where c is a constant.

As before, clearly, $\Phi(D_0) = 0$

and for all $i > 0$, $\Phi(D_i) \geq 0$

MAKE-HEAP(x):

actual cost, $c_i = 1$ (for creating the singleton heap)

potential change, $\Delta_i = \Phi(D_i) - \Phi(D_{i-1}) = c$

(as #trees increases by 1)

amortized cost, $\hat{c}_i = c_i + \Delta_i = 1 + c = \Theta(1)$

Binomial Heap Operations with Lazy Union

We use exactly the same potential function as in the previous version,

$\Phi(D_i) = c \times$ (#trees in the data structure after the i-th operation),

where c is a constant.

UNION($\boldsymbol{H_1}, \boldsymbol{H_2}$):

actual cost, $c_i = 1$ (for merging the two doubly linked lists)

potential change, $\Delta_i = \Phi(D_i) - \Phi(D_{i-1}) = 0$

(no new tree is created or destroyed)

amortized cost, $\hat{c}_i = c_i + \Delta_i = 1 = \Theta(1)$

Binomial Heap Operations with Lazy Union

We use exactly the same potential function as in the previous version,

$\Phi(D_i) = c \times$ (#trees in the data structure after the i-th operation),

where c is a constant.

INSERT(H, x):

Constant amount of work is done by MAKE-HEAP and UNION, and MAKE-HEAP creates a new tree.

actual cost, $c_i = 1 + 1 = 2$

potential change, $\Delta_i = \Phi(D_i) - \Phi(D_{i-1}) = c$

amortized cost, $\hat{c}_i = c_i + \Delta_i = 2 + c = \Theta(1)$

Binomial Heap Operations with Lazy Union

We use exactly the same potential function as in the previous version,

$\Phi(D_i) = c \times$ (#trees in the data structure after the i-th operation),

where c is a constant.

EXTRACT-MIN(H):

Cost of creating the array of pointers is $\lfloor \log_2 n \rfloor + 1$.

Suppose we start with t trees in the doubly linked list, and perform l link operations during the conversion from linked list to array version. So we perform $t + l$ work, and end up with $t - l$ trees.

Cost of converting to the linked list version is $t - l$.

actual cost, $c_i = \lfloor \log_2 n \rfloor + 1 + (t + l) + (t - l) = 2t + \lfloor \log_2 n \rfloor + 1$

potential change, $\Delta_i = \Phi(D_i) - \Phi(D_{i-1}) = -c \times l$

Binomial Heap Operations with Lazy Union

We use exactly the same potential function as in the previous version,

$$\Phi(D_i) = c \times (\text{ \#trees in the data structure after the } i\text{-th operation }),$$

where c is a constant.

EXTRACT-MIN(H):

actual cost, $c_i = \lfloor \log_2 n \rfloor + 1 + (t + l) + (t - l) = 2t + \lfloor \log_2 n \rfloor + 1$

potential change, $\Delta_i = \Phi(D_i) - \Phi(D_{i-1}) = -c \times l$

amortized cost, $\hat{c}_i = c_i + \Delta_i = 2(t - l) + \lfloor \log_2 n \rfloor + 1 - (c - 2) \times l$

But $t - l \leq \lfloor \log_2 n \rfloor + 1$ (as we have at most one tree of each rank)

So, $\hat{c}_i \leq 3\lfloor \log_2 n \rfloor + 3 - (c - 2) \times l$

$\leq 3\lfloor \log_2 n \rfloor + 3$ (assuming $c \geq 2$)

$= O(\log n)$

Binomial Heap Operations

Heap Operation	Worst-case	Amortized (Eager Union)	Amortized (Lazy Union)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$O(\log n)$	$\Theta(1)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$O(\log n)$	$O(\log n)$	$O(\log n)$
UNION	$O(\log n)$	$O(\log n)$	$\Theta(1)$