Table of Contents

Section outline

1. **Beyond the binary heap**
   - Merging of binary heaps
   - Alternate formulation of heap merging
   - Example of alternate formulation of heap merging
   - Optimised heap merging using NPL
   - Time complexity of NPL guided heap merging
   - Leftist heap
   - Leftist heap operations

Merging of binary heaps

Heap merging can be used to implement heap operations

insert A single element is a heap; merging it with an existing heap leads to an insertion of that element into the heap

delete After the min/max element is removed from the heap, we are left with two heaps; being able to merge these two heaps would allow the deletion to be completed

Efficient merging mechanism is needed

Merging of binary heaps

- Concatenate the two arrays of m and n keys
- Make a new heap in $O(N)$ time, $N = m + n$
- Also possible to add elements from one heap to the other, if there is additional space left over in the array
- Complexity: $N \lg N \geq \lg n + \ldots + \lg(n + m - 1) = \lg\left(\frac{(N-1)!}{(n-1)!}\right)$

Merging of binary heaps

Heap merging can be used to implement heap operations

insert A single element is a heap; merging it with an existing heap leads to an insertion of that element into the heap

delete After the min/max element is removed from the heap, we are left with two heaps; being able to merge these two heaps would allow the deletion to be completed

Efficient merging mechanism is needed

Merging of binary heaps

- Concatenate the two arrays of m and n keys
- Make a new heap in $O(N)$ time, $N = m + n$
- Also possible to add elements from one heap to the other, if there is additional space left over in the array
- Complexity: $N \lg N \geq \lg n + \ldots + \lg(n + m - 1) = \lg \left(\frac{(N-1)!}{(n-1)!} \right)$

Merging of binary heaps

Heap merging can be used to implement heap operations

insert A single element is a heap; merging it with an existing heap leads to an insertion of that element into the heap

delete After the min/max element is removed from the heap, we are left with two heaps; being able to merge these two heaps would allow the deletion to be completed

Efficient merging mechanism is needed

Merging of binary heaps

- Concatenate the two arrays of m and n keys
- Make a new heap in $O(N)$ time, $N = m + n$
- Also possible to add elements from one heap to the other, if there is additional space left over in the array
- Complexity: $N \lg N \geq \lg n + \ldots + \lg(n + m - 1) = \lg\left(\frac{(N-1)!}{(n-1)!}\right)$

Alternate formulation of heap merging

Inputs Let two heaps A and B be given for merging, objective is to merge these two heaps – via heapMerge(A, B)

Base case If either of A or B is empty, return the other

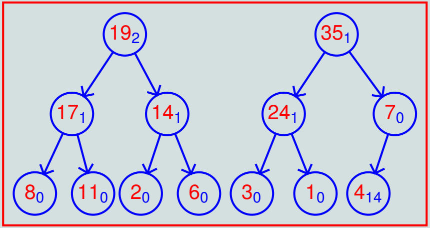
Induction

1. Choose the heap (say A with sub-trees A_L and A_R) containing the larger max element
2. If any sub-tree of A is missing, attach B in its place and return heap rooted at A
3. Detach either of the sub-trees A_L or A_R of as X and replace it with heapMerge(X, B)

Example of alternate formulation of heap merging

Example (Merging of two heaps)

Heaps to merge

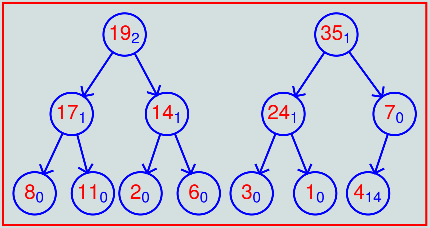


Red box indicates the heap resulting from merging the heaps inside it

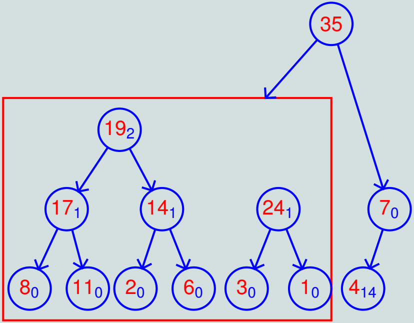
Example of alternate formulation of heap merging

Example (Merging of two heaps)

Heaps to merge



Let the larger key be the root

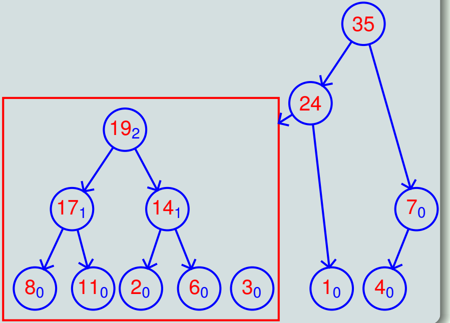


Red box indicates the heap resulting from merging the heaps inside it

Naive merging of heaps

Example (Merging of two heaps (contd.))

Step 3 of merging recursively

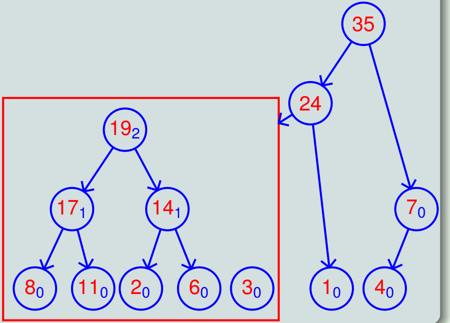


Red box indicates the heap resulting from merging the heaps inside it

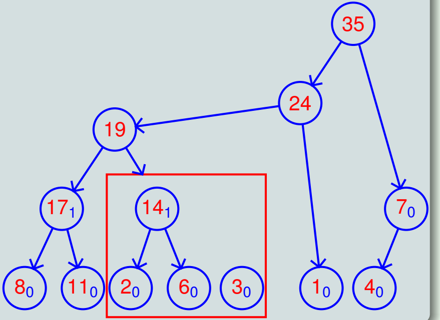
Naive merging of heaps

Example (Merging of two heaps (contd.))

Step 3 of merging recursively



Step 4 of merging recursively

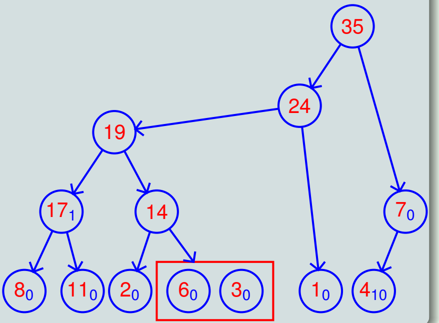


Red box indicates the heap resulting from merging the heaps inside it

Naive merging of heaps (contd.)

Example (Merging of two heaps (contd.))

Step 5 of merging recursively

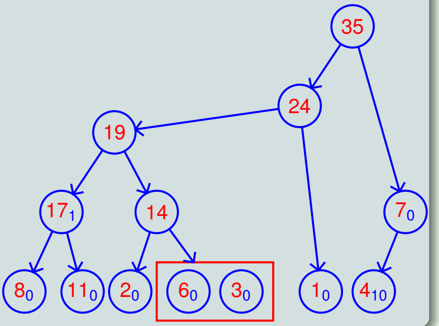


Red box indicates the heap resulting from merging the heaps inside it

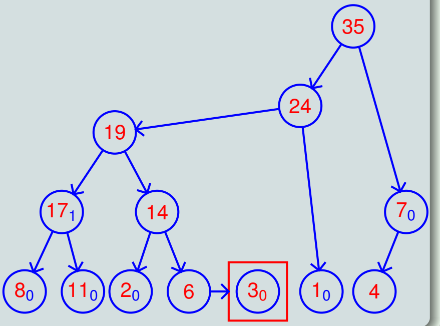
Naive merging of heaps (contd.)

Example (Merging of two heaps (contd.))

Step 5 of merging recursively



Step 6 of merging recursively

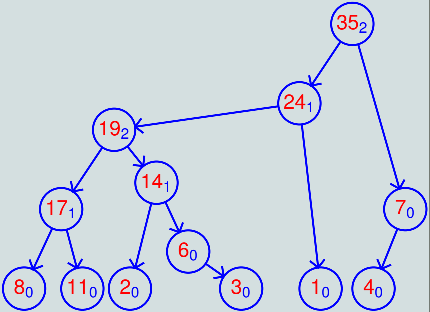


Red box indicates the heap resulting from merging the heaps inside it

Observations on naive merging of heaps

Example (Merging of two heaps (contd.))

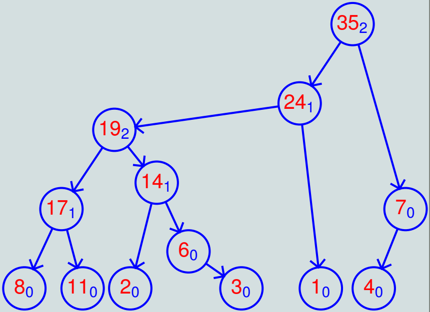
Step 7 of merging recursively



Observations on naive merging of heaps

Example (Merging of two heaps (contd.))

Step 7 of merging recursively



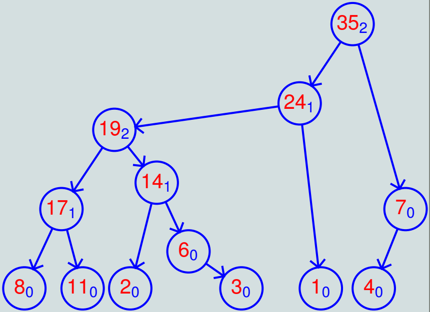
Observations

- Resulting structure no longer a complete binary tree
- Heap ordering is maintained
- Structurally only a binary tree
- Merging proceeds along arbitrary paths of both trees
- Longest path in each tree may be followed
- Each tree may be degenerate
- Complexity: $O(n_1 + n_2)$

Observations on naive merging of heaps

Example (Merging of two heaps (contd.))

Step 7 of merging recursively



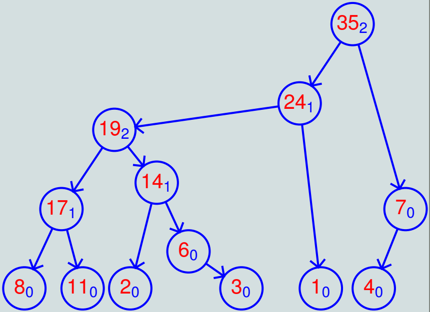
Observations

- Resulting structure no longer a complete binary tree
- Heap ordering is maintained
- Structurally only a binary tree
- Merging proceeds along arbitrary paths of both trees
- Longest path in each tree may be followed
- Each tree may be degenerate
- Complexity: $O(n_1 + n_2)$

Observations on naive merging of heaps

Example (Merging of two heaps (contd.))

Step 7 of merging recursively



Observations

- Resulting structure no longer a complete binary tree
- Heap ordering is maintained
- Structurally only a binary tree
- Merging proceeds along arbitrary paths of both trees
- Longest path in each tree may be followed
- Each tree may be degenerate
- Complexity: $O(n_1 + n_2)$

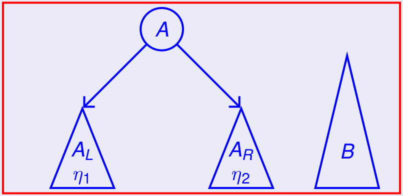
Optimised heap merging using NPL

Key observation

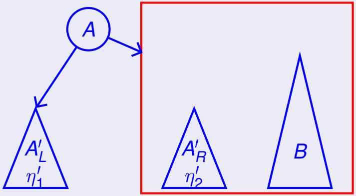
Merging proceeds along arbitrary (possibly longest) paths of both trees

- Can the choice be optimised so that longer paths are avoided?
- Let η denote the shortest distance to a leaf – the null path length (NPL)
- Let A'_R be such that $\eta'_2 = \min(\eta_1, \eta_2)$, so that termination can happen along the shortest available path to a leaf

Heaps to merge



Let the larger key be the root



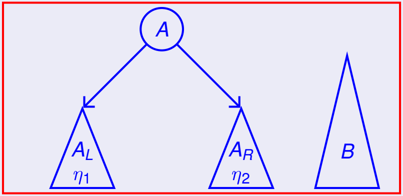
Optimised heap merging using NPL

Key observation

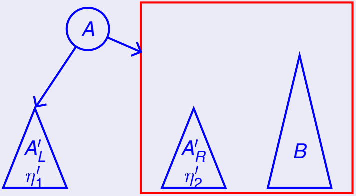
Merging proceeds along arbitrary (possibly longest) paths of both trees

- Can the choice be optimised so that longer paths are avoided?
- Let η denote the shortest distance to a leaf – the null path length (NPL)
- Let A'_R be such that $n'_2 = \min(\eta_1, \eta_2)$, so that termination can happen along the shortest available path to a leaf

Heaps to merge



Let the larger key be the root



Time complexity of NPL guided heap merging

- As merging proceeds, it is necessary to update the NPL of nodes on the affected path as:

```
n->npl = 1 + min(n->lC ? n->lC->npl:-1,
                 n->rC ? n->rC->npl:-1);
```

- NPL is at most lg n (for a binary heap), otherwise less
- Merging is done in $O(2\lg\left(\frac{n}{2}\right)) = O(\lg n)$ time

NPL properties

Let the NPL of a binary tree T be l

- The nodes of T from the root till level l form a perfect binary tree (otherwise, the NPL would have to be shorter)
- The mininum number of nodes in T is $2^{l+1} - 1$

Time complexity of NPL guided heap merging

- As merging proceeds, it is necessary to update the NPL of nodes on the affected path as:

```
n->npl = 1 + min(n->lC ? n->lC->npl:-1,
                 n->rC ? n->rC->npl:-1);
```

- NPL is at most lg n (for a binary heap), otherwise less
- Merging is done in $O(2\lg\left(\frac{n}{2}\right)) = O(\lg n)$ time

NPL properties

Let the NPL of a binary tree T be l

- The nodes of T from the root till level l form a perfect binary tree (otherwise, the NPL would have to be shorter)
- The mininum number of nodes in T is $2^{l+1} - 1$

Time complexity of NPL guided heap merging

- As merging proceeds, it is necessary to update the NPL of nodes on the affected path as:

```
n->npl = 1 + min(n->lC ? n->lC->npl:-1,
                 n->rC ? n->rC->npl:-1);
```

- NPL is at most lg n (for a binary heap), otherwise less
- Merging is done in $O(2\lg\left(\frac{n}{2}\right)) = O(\lg n)$ time

NPL properties

Let the NPL of a binary tree T be l

- The nodes of T from the root till level l form a perfect binary tree (otherwise, the NPL would have to be shorter)
- The mininum number of nodes in T is $2^{l+1} - 1$

Time complexity of NPL guided heap merging

- As merging proceeds, it is necessary to update the NPL of nodes on the affected path as:

```
n->npl = 1 + min(n->lC ? n->lC->npl:-1,
                 n->rC ? n->rC->npl:-1);
```

- NPL is at most $\lg n$ (for a binary heap), otherwise less
- Merging is done in $O(2\lg\left(\frac{n}{2}\right)) = O(\lg n)$ time

NPL properties

Let the NPL of a binary tree T be l

- The nodes of T from the root till level l form a perfect binary tree (otherwise, the NPL would have to be shorter)
- The mininum number of nodes in T is $2^{l+1} - 1$

Leftist heap

Definition (Leftist tree)

A binary tree T is said to be leftist, if for any node u of T with left and right children v_l and v_r, respectively, $\text{npl}(v_l) \geq \text{npl}(v_r)$; it is conventionally assumed that $\text{npl}(\phi) = -1$.

A leftist binary tree satisfying the heap property is a leftist heap, invented by Knuth, 1973

Thus, for every node in the leftist tree, the left subtree is at least as deep as the right subtree.

In a leftist tree, the length of the rightmost path starting from the root node (RPL) matches the NPL

- NPL guided merging does preserves leftist heap property? No!
- Property can be restored by swapping children of nodes violating this property while retreating after completion of recursive merging

Leftist heap

Definition (Leftist tree)

A binary tree T is said to be leftist, if for any node u of T with left and right children v_l and v_r, respectively, $\text{npl}(v_l) \geq \text{npl}(v_r)$; it is conventionally assumed that $\text{npl}(\phi) = -1$.

A leftist binary tree satisfying the heap property is a leftist heap, invented by Knuth, 1973

Thus, for every node in the leftist tree, the left subtree is at least as deep as the right subtree.

In a leftist tree, the length of the rightmost path starting from the root node (RPL) matches the NPL

- NPL guided merging does preserves leftist heap property? No!
- Property can be restored by swapping children of nodes violating this property while retreating after completion of recursive merging

Leftist heap

Definition (Leftist tree)

A binary tree T is said to be leftist, if for any node u of T with left and right children v_l and v_r, respectively, $\text{npl}(v_l) \geq \text{npl}(v_r)$; it is conventionally assumed that $\text{npl}(\phi) = -1$.

A leftist binary tree satisfying the heap property is a leftist heap, invented by Knuth, 1973

Thus, for every node in the leftist tree, the left subtree is at least as deep as the right subtree.

In a leftist tree, the length of the rightmost path starting from the root node (RPL) matches the NPL

- NPL guided merging does preserves leftist heap property? No!
- Property can be restored by swapping children of nodes violating this property while retreating after completion of recursive merging

Leftist heap

Definition (Leftist tree)

A binary tree T is said to be leftist, if for any node u of T with left and right children v_l and v_r, respectively, $\text{npl}(v_l) \geq \text{npl}(v_r)$; it is conventionally assumed that $\text{npl}(\phi) = -1$.

A leftist binary tree satisfying the heap property is a leftist heap, invented by Knuth, 1973

Thus, for every node in the leftist tree, the left subtree is at least as deep as the right subtree.

In a leftist tree, the length of the rightmost path starting from the root node (RPL) matches the NPL

- NPL guided merging does preserves leftist heap property? No!
- Property can be restored by swapping children of nodes violating this property while retreating after completion of recursive merging

Leftist heap

Definition (Leftist tree)

A binary tree T is said to be leftist, if for any node u of T with left and right children v_l and v_r, respectively, $\text{npl}(v_l) \geq \text{npl}(v_r)$; it is conventionally assumed that $\text{npl}(\phi) = -1$.

A leftist binary tree satisfying the heap property is a leftist heap, invented by Knuth, 1973

Thus, for every node in the leftist tree, the left subtree is at least as deep as the right subtree.

In a leftist tree, the length of the rightmost path starting from the root node (RPL) matches the NPL

- NPL guided merging does preserves leftist heap property? No!
- Property can be restored by swapping children of nodes violating this property while retreating after completion of recursive merging

Leftist heap

Definition (Leftist tree)

A binary tree T is said to be leftist, if for any node u of T with left and right children v_l and v_r, respectively, $\text{npl}(v_l) \geq \text{npl}(v_r)$; it is conventionally assumed that $\text{npl}(\phi) = -1$.

A leftist binary tree satisfying the heap property is a leftist heap, invented by Knuth, 1973

Thus, for every node in the leftist tree, the left subtree is at least as deep as the right subtree.

In a leftist tree, the length of the rightmost path starting from the root node (RPL) matches the NPL

- NPL guided merging does preserves leftist heap property? No!
- Property can be restored by swapping children of nodes violating this property while retreating after completion of recursive merging

Leftist heap operations

Leftist heap operations:

1. creating a new heap
2. finding the minimum key
3. merging two leftist heaps
4. inserting a key
5. deleting the root of a tree
6. increasing a key
7. decreasing a key

- Op-1 is trivial, takes $O(1)$ time
- Op-2 key in root node located in $O(1)$ time
- Op-3 in $O(\lg n)$ time, as explained
- Op-4 may be done by merging a single node heap for the key with the existing heap, in $O(\lg n)$ time
- Op-5 would require removing, root node and merging the resulting subtrees in $O(\lg n)$ time
- Op-6 only leads to the percolation keys downwards in the leftist tree in $O(\lg n)$ time
- Op-7 for simple heaps, remove sub-tree, adjust heap and then merge – $O(\lg n)$ time

Does not work for leftist heaps (why?)

Utility of leftist heaps?

Leftist heap operations

Leftist heap operations:

1. creating a new heap
2. finding the minimum key
3. merging two leftist heaps
4. inserting a key
5. deleting the root of a tree
6. increasing a key
7. decreasing a key

Utility of leftist heaps?

- Op-1 is trivial, takes $O(1)$ time
- Op-2 key in root node located in $O(1)$ time
- Op-3 in $O(\lg n)$ time, as explained
- Op-4 may be done by merging a single node heap for the key with the existing heap, in $O(\lg n)$ time
- Op-5 would require removing, root node and merging the resulting subtrees in $O(\lg n)$ time
- Op-6 only leads to the percolation keys downwards in the leftist tree in $O(\lg n)$ time
- Op-7 for simple heaps, remove sub-tree, adjust heap and then merge – $O(\lg n)$ time
  Does not work for leftist heaps (why?)

Leftist heap operations

Leftist heap operations:

1. creating a new heap
2. finding the minimum key
3. merging two leftist heaps
4. inserting a key
5. deleting the root of a tree
6. increasing a key
7. decreasing a key

- Op-1 is trivial, takes $O(1)$ time
- Op-2 key in root node located in $O(1)$ time
- Op-3 in $O(\lg n)$ time, as explained
- Op-4 may be done by merging a single node heap for the key with the existing heap, in $O(\lg n)$ time
- Op-5 would require removing, root node and merging the resulting subtrees in $O(\lg n)$ time
- Op-6 only leads to the percolation keys downwards in the leftist tree in $O(\lg n)$ time
- Op-7 for simple heaps, remove sub-tree, adjust heap and then merge – $O(\lg n)$ time

Does not work for leftist heaps (why?)

Utility of leftist heaps?

Leftist heap operations

Leftist heap operations:

1. creating a new heap
2. finding the minimum key
3. merging two leftist heaps
4. inserting a key
5. deleting the root of a tree
6. increasing a key
7. decreasing a key

- Op-1 is trivial, takes $O(1)$ time
- Op-2 key in root node located in $O(1)$ time
- Op-3 in $O(\lg n)$ time, as explained
- Op-4 may be done by merging a single node heap for the key with the existing heap, in $O(\lg n)$ time
- Op-5 would require removing, root node and merging the resulting subtrees in $O(\lg n)$ time
- Op-6 only leads to the percolation keys downwards in the leftist tree in $O(\lg n)$ time
- Op-7 for simple heaps, remove sub-tree, adjust heap and then merge – $O(\lg n)$ time

Does not work for leftist heaps (why?)

Utility of leftist heaps?

Leftist heap operations

Leftist heap operations:

1. creating a new heap
2. finding the minimum key
3. merging two leftist heaps
4. inserting a key
5. deleting the root of a tree
6. increasing a key
7. decreasing a key

Utility of leftist heaps?

- Op-1 is trivial, takes $O(1)$ time
- Op-2 key in root node located in $O(1)$ time
- Op-3 in $O(\lg n)$ time, as explained
- Op-4 may be done by merging a single node heap for the key with the existing heap, in $O(\lg n)$ time
- Op-5 would require removing, root node and merging the resulting subtrees in $O(\lg n)$ time
- Op-6 only leads to the percolation keys downwards in the leftist tree in $O(\lg n)$ time
- Op-7 for simple heaps, remove sub-tree, adjust heap and then merge – $O(\lg n)$ time

Does not work for leftist heaps (why?)

Leftist heap operations

Leftist heap operations:

1. creating a new heap
2. finding the minimum key
3. merging two leftist heaps
4. inserting a key
5. deleting the root of a tree
6. increasing a key
7. decreasing a key

Utility of leftist heaps?

- Op-1 is trivial, takes $O(1)$ time
- Op-2 key in root node located in $O(1)$ time
- Op-3 in $O(\lg n)$ time, as explained
- Op-4 may be done by merging a single node heap for the key with the existing heap, in $O(\lg n)$ time
- Op-5 would require removing, root node and merging the resulting subtrees in $O(\lg n)$ time
- Op-6 only leads to the percolation keys downwards in the leftist tree in $O(\lg n)$ time
- Op-7 for simple heaps, remove sub-tree, adjust heap and then merge – $O(\lg n)$ time

Does not work for leftist heaps (why?)

Leftist heap operations

Leftist heap operations:

1. creating a new heap
2. finding the minimum key
3. merging two leftist heaps
4. inserting a key
5. deleting the root of a tree
6. increasing a key
7. decreasing a key

- Op-1 is trivial, takes $O(1)$ time
- Op-2 key in root node located in $O(1)$ time
- Op-3 in $O(\lg n)$ time, as explained
- Op-4 may be done by merging a single node heap for the key with the existing heap, in $O(\lg n)$ time
- Op-5 would require removing, root node and merging the resulting subtrees in $O(\lg n)$ time
- Op-6 only leads to the percolation keys downwards in the leftist tree in $O(\lg n)$ time
- Op-7 for simple heaps, remove sub-tree, adjust heap and then merge – $O(\lg n)$ time

  Does not work for leftist heaps (why?)

Utility of leftist heaps?

Leftist heap operations

Leftist heap operations:

1. creating a new heap
2. finding the minimum key
3. merging two leftist heaps
4. inserting a key
5. deleting the root of a tree
6. increasing a key
7. decreasing a key

- Op-1 is trivial, takes $O(1)$ time
- Op-2 key in root node located in $O(1)$ time
- Op-3 in $O(\lg n)$ time, as explained
- Op-4 may be done by merging a single node heap for the key with the existing heap, in $O(\lg n)$ time
- Op-5 would require removing, root node and merging the resulting subtrees in $O(\lg n)$ time
- Op-6 only leads to the percolation keys downwards in the leftist tree in $O(\lg n)$ time
- Op-7 for simple heaps, remove sub-tree, adjust heap and then merge – $O(\lg n)$ time

Does not work for leftist heaps (why?)

Utility of leftist heaps?

Leftist heap operations

Leftist heap operations:

1. creating a new heap
2. finding the minimum key
3. merging two leftist heaps
4. inserting a key
5. deleting the root of a tree
6. increasing a key
7. decreasing a key

Utility of leftist heaps?

- Op-1 is trivial, takes $O(1)$ time
- Op-2 key in root node located in $O(1)$ time
- Op-3 in $O(\lg n)$ time, as explained
- Op-4 may be done by merging a single node heap for the key with the existing heap, in $O(\lg n)$ time
- Op-5 would require removing, root node and merging the resulting subtrees in $O(\lg n)$ time
- Op-6 only leads to the percolation keys downwards in the leftist tree in $O(\lg n)$ time
- Op-7 for simple heaps, remove sub-tree, adjust heap and then merge – $O(\lg n)$ time

Does not work for leftist heaps (why?)

Leftist heap operations

Leftist heap operations:

1. creating a new heap
2. finding the minimum key
3. merging two leftist heaps
4. inserting a key
5. deleting the root of a tree
6. increasing a key
7. decreasing a key

Utility of leftist heaps?

- Op-1 is trivial, takes $O(1)$ time
- Op-2 key in root node located in $O(1)$ time
- Op-3 in $O(\lg n)$ time, as explained
- Op-4 may be done by merging a single node heap for the key with the existing heap, in $O(\lg n)$ time
- Op-5 would require removing, root node and merging the resulting subtrees in $O(\lg n)$ time
- Op-6 only leads to the percolation keys downwards in the leftist tree in $O(\lg n)$ time
- Op-7 for simple heaps, remove sub-tree, adjust heap and then merge – $O(\lg n)$ time

  Does not work for leftist heaps (why?)

Leftist heap operations

Leftist heap operations:

1. creating a new heap
2. finding the minimum key
3. merging two leftist heaps
4. inserting a key
5. deleting the root of a tree
6. increasing a key
7. decreasing a key

Utility of leftist heaps?

- Op-1 is trivial, takes $O(1)$ time
- Op-2 key in root node located in $O(1)$ time
- Op-3 in $O(\lg n)$ time, as explained
- Op-4 may be done by merging a single node heap for the key with the existing heap, in $O(\lg n)$ time
- Op-5 would require removing, root node and merging the resulting subtrees in $O(\lg n)$ time
- Op-6 only leads to the percolation keys downwards in the leftist tree in $O(\lg n)$ time
- Op-7 for simple heaps, remove sub-tree, adjust heap and then merge – $O(\lg n)$ time

  Does not work for leftist heaps (why?)

Section outline

2 **Binomial Heaps**

- Binomial trees
- Binomial heap
- Representation of a binomial heap
- Heap union of two trees of the same order
- Operations on binomial heaps
- Merging two binomial heaps
- Comparison of binary and binomial heaps
- Amortised accounting analysis of Insert

Binomial trees

Definition (Binomial tree)

A binomial tree B_k, having an order k, is an ordered tree defined recursively as:

- B_0 consists of a single node.
- B_k, $k \geq 1$, is a pair of B_{k-1} trees, where the root of one B_{k-1} becomes the leftmost child of the other.

Example (Binomial trees)

Nodes at level ℓ of B_k

$$N_k^\ell = \begin{cases} 1 & \ell = 0 \\ 1 & \ell = k \\ N_{k-1}^\ell + N_{k-1}^{\ell-1} & 0 < \ell < k \end{cases}$$

$N_k^\ell = \binom{k}{\ell}$ or ${}^kC_\ell$

Hence the name binomial tree

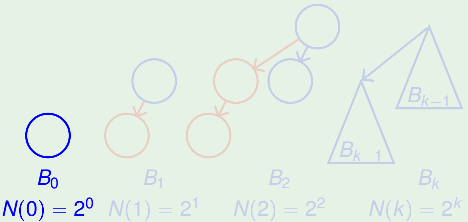
Binomial trees

Definition (Binomial tree)

A binomial tree B_k, having an order k, is an ordered tree defined recursively as:

- B_0 consists of a single node.
- B_k, $k \geq 1$, is a pair of B_{k-1} trees, where the root of one B_{k-1} becomes the leftmost child of the other.

Example (Binomial trees)



Nodes at level ℓ of B_k

$$N_k^\ell = \begin{cases} 1 & \ell = 0 \\ 1 & \ell = k \\ N_{k-1}^\ell + N_{k-1}^{\ell-1} & 0 < \ell < k \end{cases}$$

$N_k^\ell = \binom{k}{\ell}$ or ${}^kC_\ell$

Hence the name binomial tree

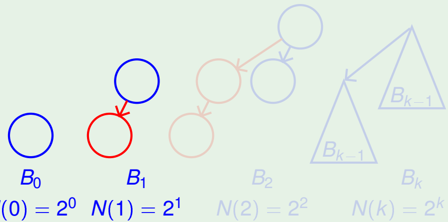
Binomial trees

Definition (Binomial tree)

A binomial tree B_k, having an order k, is an ordered tree defined recursively as:

- B_0 consists of a single node.
- B_k, $k \geq 1$, is a pair of B_{k-1} trees, where the root of one B_{k-1} becomes the leftmost child of the other.

Example (Binomial trees)



Nodes at level ℓ of B_k

$$N_k^\ell = \begin{cases} 1 & \ell = 0 \\ 1 & \ell = k \\ N_{k-1}^\ell + N_{k-1}^{\ell-1} & 0 < \ell < k \end{cases}$$

$N_k^\ell = \binom{k}{\ell}$ or ${}^kC_\ell$

Hence the name binomial tree

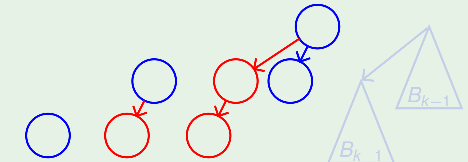
Binomial trees

Definition (Binomial tree)

A binomial tree B_k, having an order k, is an ordered tree defined recursively as:

- B_0 consists of a single node.
- B_k, $k \geq 1$, is a pair of B_{k-1} trees, where the root of one B_{k-1} becomes the leftmost child of the other.

Example (Binomial trees)



Nodes at level ℓ of B_k

$$N_k^\ell = \begin{cases} 1 & \ell = 0 \\ 1 & \ell = k \\ N_{k-1}^\ell + N_{k-1}^{\ell-1} & 0 < \ell < k \end{cases}$$

$N_k^\ell = \binom{k}{\ell}$ or ${}^kC_\ell$

Hence the name binomial tree

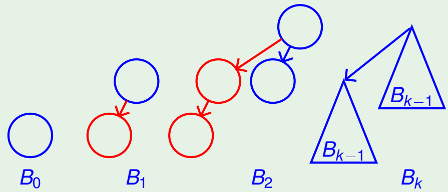
Binomial trees

Definition (Binomial tree)

A binomial tree B_k, having an order k, is an ordered tree defined recursively as:

- B_0 consists of a single node.
- B_k, $k \geq 1$, is a pair of B_{k-1} trees, where the root of one B_{k-1} becomes the leftmost child of the other.

Example (Binomial trees)



$N(0) = 2^0$ $N(1) = 2^1$ $N(2) = 2^2$ $N(k) = 2^k$

Nodes at level ℓ of B_k

$$N_k^\ell = \begin{cases} 1 & \ell = 0 \\ 1 & \ell = k \\ N_{k-1}^\ell + N_{k-1}^{\ell-1} & 0 < \ell < k \end{cases}$$

$N_k^\ell = \binom{k}{\ell}$ or ${}^kC_\ell$

Hence the name binomial tree

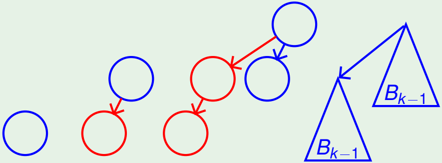
Binomial trees

Definition (Binomial tree)

A binomial tree B_k, having an order k, is an ordered tree defined recursively as:

- B_0 consists of a single node.
- B_k, $k \geq 1$, is a pair of B_{k-1} trees, where the root of one B_{k-1} becomes the leftmost child of the other.

Example (Binomial trees)



B_0 $N(0) = 2^0$ | B_1 $N(1) = 2^1$ | B_2 $N(2) = 2^2$ | B_k $N(k) = 2^k$

Nodes at level ℓ of B_k

$$N_k^\ell = \begin{cases} 1 & \ell = 0 \\ 1 & \ell = k \\ N_{k-1}^\ell + N_{k-1}^{\ell-1} & 0 < \ell < k \end{cases}$$

$N_k^\ell = \binom{k}{\ell}$ or ${}^kC_\ell$

Hence the name binomial tree

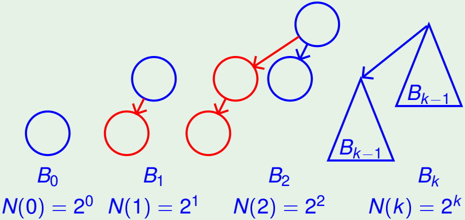
Binomial trees

Definition (Binomial tree)

A binomial tree B_k, having an order k, is an ordered tree defined recursively as:

- B_0 consists of a single node.
- B_k, $k \geq 1$, is a pair of B_{k-1} trees, where the root of one B_{k-1} becomes the leftmost child of the other.

Example (Binomial trees)



Nodes at level ℓ of B_k

$$N_k^\ell = \begin{cases} 1 & \ell = 0 \\ 1 & \ell = k \\ N_{k-1}^\ell + N_{k-1}^{\ell-1} & 0 < \ell < k \end{cases}$$

$N_k^\ell = \binom{k}{\ell}$ or ${}^kC_\ell$

Hence the name binomial tree

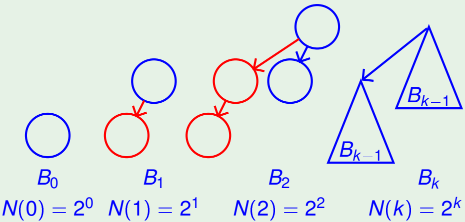
Binomial trees

Definition (Binomial tree)

A binomial tree B_k, having an order k, is an ordered tree defined recursively as:

- B_0 consists of a single node.
- B_k, $k \geq 1$, is a pair of B_{k-1} trees, where the root of one B_{k-1} becomes the leftmost child of the other.

Example (Binomial trees)



Nodes at level ℓ of B_k

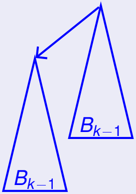
$$N_k^\ell = \begin{cases} 1 & \ell = 0 \\ 1 & \ell = k \\ N_{k-1}^\ell + N_{k-1}^{\ell-1} & 0 < \ell < k \end{cases}$$

$$N_k^\ell = \binom{k}{\ell} \text{ or } {}^kC_\ell$$

Hence the name binomial tree

Structure of a binomial tree

Decomposition of binomial tree B_k



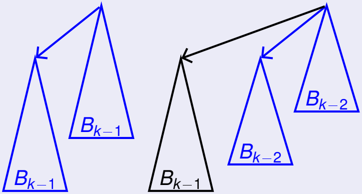
Definition (Alternate definition of binomial tree)

A binomial tree is defined recursively as follows:

- A binomial tree of order 0 is a single node
- A binomial tree of order k has a root node whose children are root nodes of binomial trees of orders $k-1$, $k-2$, . . ., 2, 1, 0 (in order)
- Number of children of the root is the rank (= order) of the tree

Structure of a binomial tree

Decomposition of binomial tree B_k



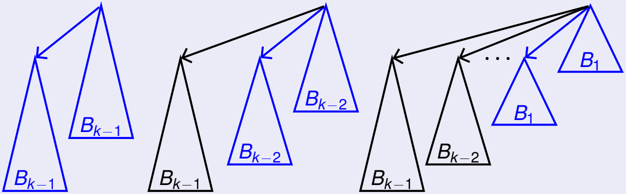
Definition (Alternate definition of binomial tree)

A binomial tree is defined recursively as follows:

- A binomial tree of order 0 is a single node
- A binomial tree of order k has a root node whose children are root nodes of binomial trees of orders $k-1, k-2, \ldots, 2, 1, 0$ (in order)
- Number of children of the root is the rank (= order) of the tree

Structure of a binomial tree

Decomposition of binomial tree B_k



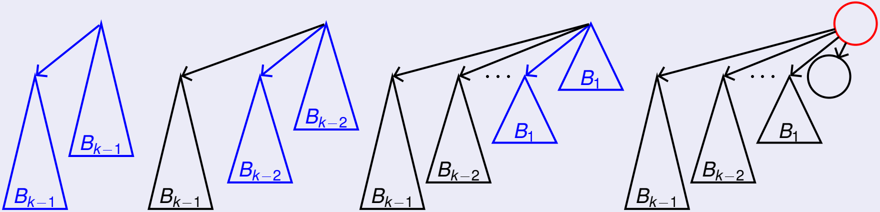
Definition (Alternate definition of binomial tree)

A binomial tree is defined recursively as follows:

- A binomial tree of order 0 is a single node
- A binomial tree of order k has a root node whose children are root nodes of binomial trees of orders $k-1$, $k-2$, ..., 2, 1, 0 (in order)
- Number of children of the root is the rank (= order) of the tree

Structure of a binomial tree

Decomposition of binomial tree B_k



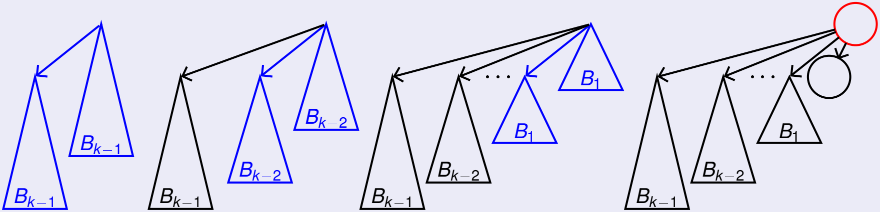
Definition (Alternate definition of binomial tree)

A binomial tree is defined recursively as follows:

- A binomial tree of order 0 is a single node
- A binomial tree of order k has a root node whose children are root nodes of binomial trees of orders $k-1$, $k-2$, . . ., 2, 1, 0 (in order)
- Number of children of the root is the rank (= order) of the tree

Structure of a binomial tree

Decomposition of binomial tree B_k



Definition (Alternate definition of binomial tree)

A binomial tree is defined recursively as follows:

- A binomial tree of order 0 is a single node
- A binomial tree of order k has a root node whose children are root nodes of binomial trees of orders $k-1$, $k-2$, . . ., 2, 1, 0 (in order)
- Number of children of the root is the rank (= order) of the tree

Binomial heap

Definition (Binomial heap)

A binomial heap, invented by Vuillemin, 1978, is a collection of binomial trees that satisfies the following binomial-heap properties:

1. No two binomial trees in the collection have the same size.
2. Each node in each tree has a key.
3. Each binomial tree in the collection is heap-ordered in the sense that each non-root has a key strictly less than the key of its parent.

Some implications

- For all $n \geq 1$ and $k \geq 0$, B_k appears in an n-node binary heap if and only if the $(k+1)^{\text{st}}$ bit of the binary representation of n is a 1
- The number of trees in a binomial heap of n nodes is $O(\lg n)$
- The time to search for the minimum element is $O(\lg n)$

Binomial heap

Definition (Binomial heap)

A binomial heap, invented by Vuillemin, 1978, is a collection of binomial trees that satisfies the following binomial-heap properties:

1. No two binomial trees in the collection have the same size.
2. Each node in each tree has a key.
3. Each binomial tree in the collection is heap-ordered in the sense that each non-root has a key strictly less than the key of its parent.

Some implications

- For all $n \geq 1$ and $k \geq 0$, B_k appears in an n-node binary heap if and only if the $(k+1)^{\text{st}}$ bit of the binary representation of n is a 1
- The number of trees in a binomial heap of n nodes is $O(\lg n)$
- The time to search for the minimum element is $O(\lg n)$

Binomial heap

Definition (Binomial heap)

A binomial heap, invented by Vuillemin, 1978, is a collection of binomial trees that satisfies the following binomial-heap properties:

1. No two binomial trees in the collection have the same size.
2. Each node in each tree has a key.
3. Each binomial tree in the collection is heap-ordered in the sense that each non-root has a key strictly less than the key of its parent.

Some implications

- For all $n \geq 1$ and $k \geq 0$, B_k appears in an n-node binary heap if and only if the $(k+1)^{\text{st}}$ bit of the binary representation of n is a 1
- The number of trees in a binomial heap of n nodes is $O(\lg n)$
- The time to search for the minimum element is $O(\lg n)$

Binomial heap

Definition (Binomial heap)

A binomial heap, invented by Vuillemin, 1978, is a collection of binomial trees that satisfies the following binomial-heap properties:

1. No two binomial trees in the collection have the same size.
2. Each node in each tree has a key.
3. Each binomial tree in the collection is heap-ordered in the sense that each non-root has a key strictly less than the key of its parent.

Some implications

- For all $n \geq 1$ and $k \geq 0$, B_k appears in an n-node binary heap if and only if the $(k+1)^{\text{st}}$ bit of the binary representation of n is a 1
- The number of trees in a binomial heap of n nodes is $O(\lg n)$
- The time to search for the minimum element is $O(\lg n)$

Binomial heap

Definition (Binomial heap)

A binomial heap, invented by Vuillemin, 1978, is a collection of binomial trees that satisfies the following binomial-heap properties:

1. No two binomial trees in the collection have the same size.
2. Each node in each tree has a key.
3. Each binomial tree in the collection is heap-ordered in the sense that each non-root has a key strictly less than the key of its parent.

Some implications

- For all $n \geq 1$ and $k \geq 0$, B_k appears in an n-node binary heap if and only if the $(k+1)^{\text{st}}$ bit of the binary representation of n is a 1
- The number of trees in a binomial heap of n nodes is $O(\lg n)$
- The time to search for the minimum element is $O(\lg n)$

Binomial heap

Definition (Binomial heap)

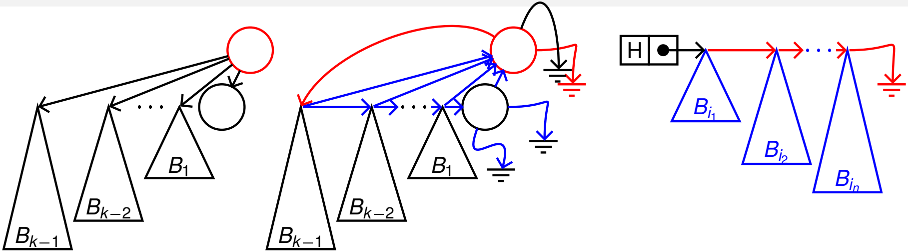
A binomial heap, invented by Vuillemin, 1978, is a collection of binomial trees that satisfies the following binomial-heap properties:

1. No two binomial trees in the collection have the same size.
2. Each node in each tree has a key.
3. Each binomial tree in the collection is heap-ordered in the sense that each non-root has a key strictly less than the key of its parent.

Some implications

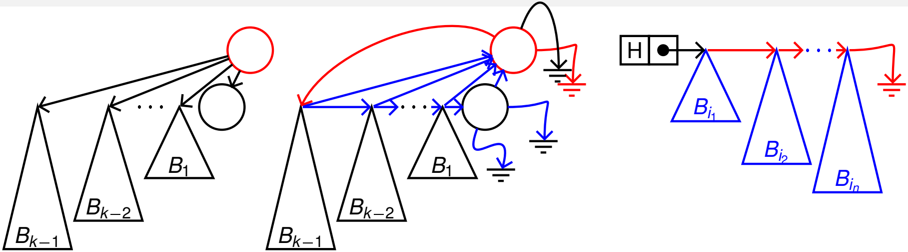
- For all $n \geq 1$ and $k \geq 0$, B_k appears in an n-node binary heap if and only if the $(k+1)^{\text{st}}$ bit of the binary representation of n is a 1
- The number of trees in a binomial heap of n nodes is $O(\lg n)$
- The time to search for the minimum element is $O(\lg n)$

Representation of a binomial heap



1. The following items of information per node are needed:
   - a field **`key`** for its key,
   - a field **`degree`** for the number of children,
   - a pointer **`child`** , which points to the leftmost-child,
   - a pointer **`sibling`**, which points to the right-sibling, and
   - a pointer **`parent`**, which points to the parent
2. The roots of the trees are connected so that the sizes of the connected trees are in decreasing order
3. For a heap **`H`**, **`H.head`** points to the head of the list

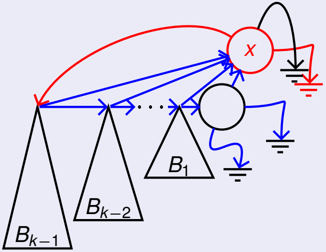
Representation of a binomial heap



1. The following items of information per node are needed:
   - a field **key** for its key,
   - a field **degree** for the number of children,
   - a pointer **child** , which points to the leftmost-child,
   - a pointer **sibling**, which points to the right-sibling, and
   - a pointer **parent**, which points to the parent
2. The roots of the trees are connected so that the sizes of the connected trees are in decreasing order
3. For a heap **H**, **H.head** points to the head of the list

Heap union of two trees of the same order

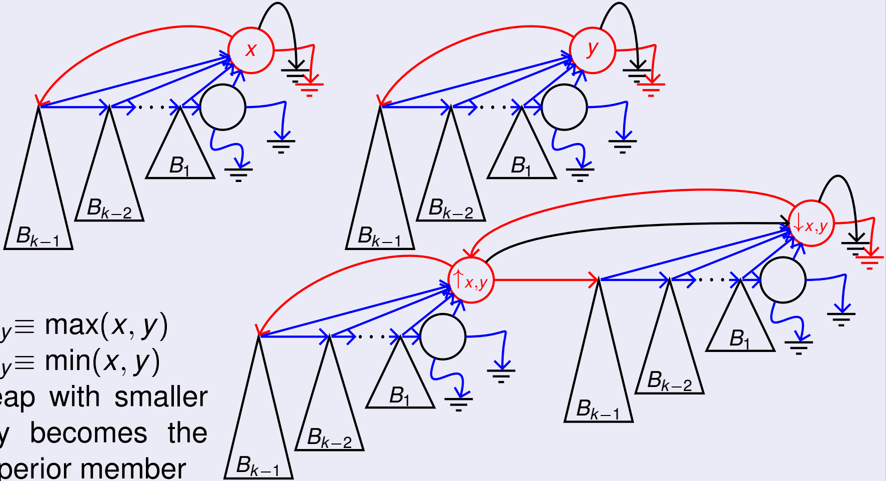
Heap union of two binomial min-heaps of the same order



$\uparrow_{x,y} \equiv \max(x, y)$
$\downarrow_{x,y} \equiv \min(x, y)$
Heap with smaller key becomes the superior member

Heap union of two trees of the same order

Heap union of two binomial min-heaps of the same order



$\uparrow_{x,y} \equiv \max(x, y)$
$\downarrow_{x,y} \equiv \min(x, y)$
Heap with smaller key becomes the superior member

Operations on binomial heaps

The important operations on a binomial heap are:

1. creating a new heap
2. finding the minimum key
3. merging two binomial heaps
4. inserting a key
5. deleting the root of a tree
6. decreasing a key

- Op-1 is trivial, $O(1)$ time
- Op-2 requires traversing through all the binomial trees, takes $O(\lg n)$ time
- Op-4 may be done by merging a single node heap for the key with the existing heap
- Op-5 would require removing, in $O(\lg n)$ time,the binomial tree T having the minimum element from the heap yielding H'; removing the root node of T and reorganising the remaining binomial trees of T, in $O(\lg n)$ time, as a binomial heap and merging this with H'
- Op-6 only leads to percolation of the key the binomial tree containing the affected key in $O(\lg n)$ time
- Op-3 remains to be addressed

Operations on binomial heaps

The important operations on a binomial heap are:

1. creating a new heap
2. finding the minimum key
3. merging two binomial heaps
4. inserting a key
5. deleting the root of a tree
6. decreasing a key

- Op-1 is trivial, $O(1)$ time
- Op-2 requires traversing through all the binomial trees, takes $O(\lg n)$ time
- Op-4 may be done by merging a single node heap for the key with the existing heap
- Op-5 would require removing, in $O(\lg n)$ time,the binomial tree T having the minimum element from the heap yielding H'; removing the root node of T and reorganising the remaining binomial trees of T, in $O(\lg n)$ time, as a binomial heap and merging this with H'
- Op-6 only leads to percolation of the key the binomial tree containing the affected key in $O(\lg n)$ time
- Op-3 remains to be addressed

Operations on binomial heaps

The important operations on a binomial heap are:

1. creating a new heap
2. finding the minimum key
3. merging two binomial heaps
4. inserting a key
5. deleting the root of a tree
6. decreasing a key

- Op-1 is trivial, $O(1)$ time
- Op-2 requires traversing through all the binomial trees, takes $O(\lg n)$ time
- Op-4 may be done by merging a single node heap for the key with the existing heap
- Op-5 would require removing, in $O(\lg n)$ time,the binomial tree T having the minimum element from the heap yielding H'; removing the root node of T and reorganising the remaining binomial trees of T, in $O(\lg n)$ time, as a binomial heap and merging this with H'
- Op-6 only leads to percolation of the key the binomial tree containing the affected key in $O(\lg n)$ time
- Op-3 remains to be addressed

Operations on binomial heaps

The important operations on a binomial heap are:

1. creating a new heap
2. finding the minimum key
3. merging two binomial heaps
4. inserting a key
5. deleting the root of a tree
6. decreasing a key

- Op-1 is trivial, $O(1)$ time
- Op-2 requires traversing through all the binomial trees, takes $O(\lg n)$ time
- Op-4 may be done by merging a single node heap for the key with the existing heap
- Op-5 would require removing, in $O(\lg n)$ time,the binomial tree T having the minimum element from the heap yielding H'; removing the root node of T and reorganising the remaining binomial trees of T, in $O(\lg n)$ time, as a binomial heap and merging this with H'
- Op-6 only leads to percolation of the key the binomial tree containing the affected key in $O(\lg n)$ time
- Op-3 remains to be addressed

Operations on binomial heaps

The important operations on a binomial heap are:

1. creating a new heap
2. finding the minimum key
3. merging two binomial heaps
4. inserting a key
5. deleting the root of a tree
6. decreasing a key

- Op-1 is trivial, $O(1)$ time
- Op-2 requires traversing through all the binomial trees, takes $O(\lg n)$ time
- Op-4 may be done by merging a single node heap for the key with the existing heap
- Op-5 would require removing, in $O(\lg n)$ time,the binomial tree T having the minimum element from the heap yielding H'; removing the root node of T and reorganising the remaining binomial trees of T, in $O(\lg n)$ time, as a binomial heap and merging this with H'
- Op-6 only leads to percolation of the key the binomial tree containing the affected key in $O(\lg n)$ time
- Op-3 remains to be addressed

Operations on binomial heaps

The important operations on a binomial heap are:

1. creating a new heap
2. finding the minimum key
3. merging two binomial heaps
4. inserting a key
5. deleting the root of a tree
6. decreasing a key

- Op-1 is trivial, $O(1)$ time
- Op-2 requires traversing through all the binomial trees, takes $O(\lg n)$ time
- Op-4 may be done by merging a single node heap for the key with the existing heap
- Op-5 would require removing, in $O(\lg n)$ time,the binomial tree T having the minimum element from the heap yielding H'; removing the root node of T and reorganising the remaining binomial trees of T, in $O(\lg n)$ time, as a binomial heap and merging this with H'
- Op-6 only leads to percolation of the key the binomial tree containing the affected key in $O(\lg n)$ time
- Op-3 remains to be addressed

Operations on binomial heaps

The important operations on a binomial heap are:

1. creating a new heap
2. finding the minimum key
3. merging two binomial heaps
4. inserting a key
5. deleting the root of a tree
6. decreasing a key

- Op-1 is trivial, $O(1)$ time
- Op-2 requires traversing through all the binomial trees, takes $O(\lg n)$ time
- Op-4 may be done by merging a single node heap for the key with the existing heap
- Op-5 would require removing, in $O(\lg n)$ time,the binomial tree T having the minimum element from the heap yielding H'; removing the root node of T and reorganising the remaining binomial trees of T, in $O(\lg n)$ time, as a binomial heap and merging this with H'
- Op-6 only leads to percolation of the key the binomial tree containing the affected key in $O(\lg n)$ time
- Op-3 remains to be addressed

Merging two binomial heaps

Example (Analogy of merging binomial heaps to binary addition)

$H_i^{(k)}$ is a certain heap of k items; we wish to merge, say, $H_1^{(7)}$ and $H_2^{(11)}$

	2^0	2^1	2^2	2^3	2^4	B_0	B_1	B_2	B_3	B_4
$H_1^{(7)}$	1	1	1	0		$B_0^{(7)}$	$B_1^{(7)}$	$B_2^{(7)}$		
$H_2^{(11)}$	1	1	0	1		$B_0^{(11)}$	$B_1^{(11)}$		$B_3^{(11)}$	
S	0	1	0	0	1		$B_1^{(S)}$			$B_4^{(S)}$
C		1	1	1	1		$B_1^{(C)}$	$B_2^{(C)}$	$B_3^{(C)}$	$B_4^{(C)}$

NB: The binomial trees are in ascending order in a linked list, tree order is found from the **`degree`** field of the root node (and not positionally)

Merging two binomial heaps

Example (Analogy of merging binomial heaps to binary addition)

$H_i^{(k)}$ is a certain heap of k items; we wish to merge, say, $H_1^{(7)}$ and $H_2^{(11)}$

	2^0	2^1	2^2	2^3	2^4	B_0	B_1	B_2	B_3	B_4
$H_1^{(7)}$	1	1	1	0		$B_0^{(7)}$	$B_1^{(7)}$	$B_2^{(7)}$		
$H_2^{(11)}$	1	1	0	1		$B_0^{(11)}$	$B_1^{(11)}$		$B_3^{(11)}$	
S	0	1	0	0	1		$B_1^{(S)}$			$B_4^{(S)}$
C		1	1	1	1		$B_1^{(C)}$	$B_2^{(C)}$	$B_3^{(C)}$	$B_4^{(C)}$

NB: The binomial trees are in ascending order in a linked list, tree order is found from the `degree` field of the root node (and not positionally)

Merging two binomial heaps

Example (Analogy of merging binomial heaps to binary addition)

$H_i^{(k)}$ is a certain heap of k items; we wish to merge, say, $H_1^{(7)}$ and $H_2^{(11)}$

	2^0	2^1	2^2	2^3	2^4	B_0	B_1	B_2	B_3	B_4
$H_1^{(7)}$	1	1	1	0		$B_0^{(7)}$	$B_1^{(7)}$	$B_2^{(7)}$		
$H_2^{(11)}$	1	1	0	1		$B_0^{(11)}$	$B_1^{(11)}$		$B_3^{(11)}$	
S	0	1	0	0	1		$B_1^{(S)}$			$B_4^{(S)}$
C		1	1	1	1		$B_1^{(C)}$	$B_2^{(C)}$	$B_3^{(C)}$	$B_4^{(C)}$

NB: The binomial trees are in ascending order in a linked list, tree order is found from the **degree** field of the root node (and not positionally)

Merging two binomial heaps

Example (Analogy of merging binomial heaps to binary addition)

$H_i^{(k)}$ is a certain heap of k items; we wish to merge, say, $H_1^{(7)}$ and $H_2^{(11)}$

	2^0	2^1	2^2	2^3	2^4	B_0	B_1	B_2	B_3	B_4
$H_1^{(7)}$	1	1	1	0		$B_0^{(7)}$	$B_1^{(7)}$	$B_2^{(7)}$		
$H_2^{(11)}$	1	1	0	1		$B_0^{(11)}$	$B_1^{(11)}$		$B_3^{(11)}$	
S	0	1	0	0	1		$B_1^{(S)}$			$B_4^{(S)}$
C		1	1	1	1		$B_1^{(C)}$	$B_2^{(C)}$	$B_3^{(C)}$	$B_4^{(C)}$

NB: The binomial trees are in ascending order in a linked list, tree order is found from the `degree` field of the root node (and not positionally)

Merging two binomial heaps

Example (Analogy of merging binomial heaps to binary addition)

$H_i^{(k)}$ is a certain heap of k items; we wish to merge, say, $H_1^{(7)}$ and $H_2^{(11)}$

	2^0	2^1	2^2	2^3	2^4	B_0	B_1	B_2	B_3	B_4
$H_1^{(7)}$	1	1	1	0		$B_0^{(7)}$	$B_1^{(7)}$	$B_2^{(7)}$		
$H_2^{(11)}$	1	1	0	1		$B_0^{(11)}$	$B_1^{(11)}$		$B_3^{(11)}$	
S	0	1	0	0	1		$B_1^{(S)}$			$B_4^{(S)}$
C		1	1	1	1		$B_1^{(C)}$	$B_2^{(C)}$	$B_3^{(C)}$	$B_4^{(C)}$

NB: The binomial trees are in ascending order in a linked list, tree order is found from the `degree` field of the root node (and not positionally)

Merging two binomial heaps

Example (Analogy of merging binomial heaps to binary addition)

$H_i^{(k)}$ is a certain heap of k items; we wish to merge, say, $H_1^{(7)}$ and $H_2^{(11)}$

	2^0	2^1	2^2	2^3	2^4	B_0	B_1	B_2	B_3	B_4
$H_1^{(7)}$	1	1	1	0		$B_0^{(7)}$	$B_1^{(7)}$	$B_2^{(7)}$		
$H_2^{(11)}$	1	1	0	1		$B_0^{(11)}$	$B_1^{(11)}$		$B_3^{(11)}$	
S	0	1	0	0	1		$B_1^{(S)}$			$B_4^{(S)}$
C		1	1	1	1		$B_1^{(C)}$	$B_2^{(C)}$	$B_3^{(C)}$	$B_4^{(C)}$

NB: The binomial trees are in ascending order in a linked list, tree order is found from the `degree` field of the root node (and not positionally)

Merging two binomial heaps (contd.)

- Similar to ripple carry addition of two unsigned binary numbers
- Let H_1 and H_2 represent the two binomial heaps, initially, $H_1 = \left\langle B^1_{i_1}, B^1_{i_2}, \ldots, B^1_{i_m} \right\rangle$ and $H_2 = \left\langle B^2_{j_1}, B^2_{j_2}, \ldots, B^2_{j_n} \right\rangle$
- Let there be a carry over tree B, initially empty; its order is B°
- Let the resulting binomial heap be H, initially empty; $|H| = |H_1| + |H_2|$
- While merging, let $B^1_{i_p}$ and $B^2_{j_q}$ be at the heads of their respective sequences of binomial trees
- Merging proceeds by examining B and $B^1_{i_p}$ and $B^2_{j_q}$
- From time to time a tree is extracted from the head of H_1 or H_2
- Let $\left(B^1_{i_p} \oplus B^2_{j_q}\right)$, $i_p = j_q$, represent the heap union of $B^1_{i_p}$ $B^2_{j_q}$ to form a binomial tree of order $i_p + 1 = j_q + 1$
- If H_1 and H_2 are exhausted
  - If $B \neq \phi$, $H \leftarrow H \| B$, terminate
  - If $B = \phi$, terminate

Merging two binomial heaps (contd.)

- Similar to ripple carry addition of two unsigned binary numbers
- Let H_1 and H_2 represent the two binomial heaps, initially, $H_1 = \left\langle B^1_{i_1}, B^1_{i_2}, \ldots, B^1_{i_m} \right\rangle$ and $H_2 = \left\langle B^2_{j_1}, B^2_{j_2}, \ldots, B^2_{j_n} \right\rangle$
- Let there be a carry over tree B, initially empty; its order is B°
- Let the resulting binomial heap be H, initially empty; $|H| = |H_1| + |H_2|$
- While merging, let $B^1_{i_p}$ and $B^2_{j_q}$ be at the heads of their respective sequences of binomial trees
- Merging proceeds by examining B and $B^1_{i_p}$ and $B^2_{j_q}$
- From time to time a tree is extracted from the head of H_1 or H_2
- Let $\left(B^1_{i_p} \oplus B^2_{j_q}\right)$, $i_p = j_q$, represent the heap union of $B^1_{i_p}$ $B^2_{j_q}$ to form a binomial tree of order $i_p + 1 = j_q + 1$
- If H_1 and H_2 are exhausted
  - If $B \neq \phi$, $H \leftarrow H || B$, terminate
  - If $B = \phi$, terminate

Merging two binomial heaps (contd.)

- Similar to ripple carry addition of two unsigned binary numbers
- Let H_1 and H_2 represent the two binomial heaps, initially, $H_1 = \left\langle B^1_{i_1}, B^1_{i_2}, \ldots, B^1_{i_m} \right\rangle$ and $H_2 = \left\langle B^2_{j_1}, B^2_{j_2}, \ldots, B^2_{j_n} \right\rangle$
- Let there be a carry over tree B, initially empty; its order is B°
- Let the resulting binomial heap be H, initially empty; $|H| = |H_1| + |H_2|$
- While merging, let $B^1_{i_p}$ and $B^2_{j_q}$ be at the heads of their respective sequences of binomial trees
- Merging proceeds by examining B and $B^1_{i_p}$ and $B^2_{j_q}$
- From time to time a tree is extracted from the head of H_1 or H_2
- Let $\left(B^1_{i_p} \oplus B^2_{j_q}\right)$, $i_p = j_q$, represent the heap union of $B^1_{i_p}$ $B^2_{j_q}$ to form a binomial tree of order $i_p + 1 = j_q + 1$
- If H_1 and H_2 are exhausted
  - If $B \neq \phi$, $H \leftarrow H \| B$, terminate
  - If $B = \phi$, terminate

Merging two binomial heaps (contd.)

- Similar to ripple carry addition of two unsigned binary numbers
- Let H_1 and H_2 represent the two binomial heaps, initially, $H_1 = \left\langle B^1_{i_1}, B^1_{i_2}, \ldots, B^1_{i_m} \right\rangle$ and $H_2 = \left\langle B^2_{j_1}, B^2_{j_2}, \ldots, B^2_{j_n} \right\rangle$
- Let there be a carry over tree B, initially empty; its order is B°
- Let the resulting binomial heap be H, initially empty; $|H| = |H_1| + |H_2|$
- While merging, let $B^1_{i_p}$ and $B^2_{j_q}$ be at the heads of their respective sequences of binomial trees
- Merging proceeds by examining B and $B^1_{i_p}$ and $B^2_{j_q}$
- From time to time a tree is extracted from the head of H_1 or H_2
- Let $\left(B^1_{i_p} \oplus B^2_{j_q}\right)$, $i_p = j_q$, represent the heap union of $B^1_{i_p}$ $B^2_{j_q}$ to form a binomial tree of order $i_p + 1 = j_q + 1$
- If H_1 and H_2 are exhausted
  - If $B \neq \phi$, $H \leftarrow H || B$, terminate
  - If $B = \phi$, terminate

Merging two binomial heaps (contd.)

- Similar to ripple carry addition of two unsigned binary numbers
- Let H_1 and H_2 represent the two binomial heaps, initially, $H_1 = \left\langle B^1_{i_1}, B^1_{i_2}, \ldots, B^1_{i_m} \right\rangle$ and $H_2 = \left\langle B^2_{j_1}, B^2_{j_2}, \ldots, B^2_{j_n} \right\rangle$
- Let there be a carry over tree B, initially empty; its order is B°
- Let the resulting binomial heap be H, initially empty; $|H| = |H_1| + |H_2|$
- While merging, let $B^1_{i_p}$ and $B^2_{j_q}$ be at the heads of their respective sequences of binomial trees
- Merging proceeds by examining B and $B^1_{i_p}$ and $B^2_{j_q}$
- From time to time a tree is extracted from the head of H_1 or H_2
- Let $\left(B^1_{i_p} \oplus B^2_{j_q}\right)$, $i_p = j_q$, represent the heap union of $B^1_{i_p}$ $B^2_{j_q}$ to form a binomial tree of order $i_p + 1 = j_q + 1$
- If H_1 and H_2 are exhausted
  - If $B \neq \phi$, $H \leftarrow H \| B$, terminate
  - If $B = \phi$, terminate

Merging two binomial heaps (contd.)

If only H_1 is exhausted

- if $B = \phi$, $H \leftarrow H \| H_2$, terminate
- if $B^\circ < j_q$, $H \leftarrow H \| B$, $B \leftarrow \phi$
- if $B^\circ = j_q$, $B \leftarrow \left(B \oplus B^2_{j_q}\right)$

If only H_2 is exhausted

- if $B = \phi$, $H \leftarrow H \| H_1$, terminate
- if $B^\circ < i_p$, $H \leftarrow H \| B$, $B \leftarrow \phi$
- if $B^\circ = i_p$, $B \leftarrow \left(B \oplus B^1_{i_p}\right)$

Case: *B is empty*

- if $i_p < j_q$, $H \leftarrow H \| B^1_{i_p}$
- if $i_p > j_q$, $H \leftarrow H \| B^2_{j_q}$
- if $i_p = j_q$,
  $B \leftarrow \left(B^1_{i_p} \oplus B^2_{j_q}\right)$

Case: *B is non-empty*

- if $B^\circ = i_p = j_q$, $H \leftarrow H \| B$, $B \leftarrow \left(B^1_{i_p} \oplus B^2_{j_q}\right)$
- if $B^\circ \neq i_p < j_q$, $H \leftarrow H \| B^1_{i_p}$
- if $i_p > j_q \neq B^\circ$, $H \leftarrow H \| B^2_{j_q}$
- if $B^\circ = i_p < j_q$, $B \leftarrow \left(B \oplus B^1_{i_p}\right)$
- if $i_p > j_q = B^\circ$, $B \leftarrow \left(B \oplus B^2_{j_q}\right)$; $B \leftarrow \phi$
- if $B^\circ < i_p = j_q$, $H \leftarrow H \| B$

$O(\lg |H|)$ time, each possible tree position in H processed in $O(1)$ time

Merging two binomial heaps (contd.)

If only H_1 is exhausted

- if $B = \phi$, $H \leftarrow H \| H_2$, terminate
- if $B^\circ < j_q$, $H \leftarrow H \| B$, $B \leftarrow \phi$
- if $B^\circ = j_q$, $B \leftarrow \left(B \oplus B^2_{j_q}\right)$

If only H_2 is exhausted

- if $B = \phi$, $H \leftarrow H \| H_1$, terminate
- if $B^\circ < i_p$, $H \leftarrow H \| B$, $B \leftarrow \phi$
- if $B^\circ = i_p$, $B \leftarrow \left(B \oplus B^1_{i_p}\right)$

Case: *B is empty*

- if $i_p < j_q$, $H \leftarrow H \| B^1_{i_p}$
- if $i_p > j_q$, $H \leftarrow H \| B^2_{j_q}$
- if $i_p = j_q$, $B \leftarrow \left(B^1_{i_p} \oplus B^2_{j_q}\right)$

Case: *B is non-empty*

- if $B^\circ = i_p = j_q$, $H \leftarrow H \| B$, $B \leftarrow \left(B^1_{i_p} \oplus B^2_{j_q}\right)$
- if $B^\circ \neq i_p < j_q$, $H \leftarrow H \| B^1_{i_p}$
- if $i_p > j_q \neq B^\circ$, $H \leftarrow H \| B^2_{j_q}$
- if $B^\circ = i_p < j_q$, $B \leftarrow \left(B \oplus B^1_{i_p}\right)$
- if $i_p > j_q = B^\circ$, $B \leftarrow \left(B \oplus B^2_{j_q}\right)$; $B \leftarrow \phi$
- if $B^\circ < i_p = j_q$, $H \leftarrow H \| B$

$O(\lg |H|)$ time, each possible tree position in H processed in $O(1)$ time

Merging two binomial heaps (contd.)

If only H_1 is exhausted

- if $B = \phi$, $H \leftarrow H \| H_2$, terminate
- if $B^\circ < j_q$, $H \leftarrow H \| B$, $B \leftarrow \phi$
- if $B^\circ = j_q$, $B \leftarrow \left(B \oplus B^2_{j_q}\right)$

If only H_2 is exhausted

- if $B = \phi$, $H \leftarrow H \| H_1$, terminate
- if $B^\circ < i_p$, $H \leftarrow H \| B$, $B \leftarrow \phi$
- if $B^\circ = i_p$, $B \leftarrow \left(B \oplus B^1_{i_p}\right)$

Case: *B is empty*

- if $i_p < j_q$, $H \leftarrow H \| B^1_{i_p}$
- if $i_p > j_q$, $H \leftarrow H \| B^2_{j_q}$
- if $i_p = j_q$,
  $B \leftarrow \left(B^1_{i_p} \oplus B^2_{j_q}\right)$

Case: *B is non-empty*

- if $B^\circ = i_p = j_q$, $H \leftarrow H \| B$, $B \leftarrow \left(B^1_{i_p} \oplus B^2_{j_q}\right)$
- if $B^\circ \neq i_p < j_q$, $H \leftarrow H \| B^1_{i_p}$
- if $i_p > j_q \neq B^\circ$, $H \leftarrow H \| B^2_{j_q}$
- if $B^\circ = i_p < j_q$, $B \leftarrow \left(B \oplus B^1_{i_p}\right)$
- if $i_p > j_q = B^\circ$, $B \leftarrow \left(B \oplus B^2_{j_q}\right)$; $B \leftarrow \phi$
- if $B^\circ < i_p = j_q$, $H \leftarrow H \| B$

$O(\lg |H|)$ time, each possible tree position in H processed in $O(1)$ time

Merging two binomial heaps (contd.)

If only H_1 is exhausted

- if $B = \phi$, $H \leftarrow H \| H_2$, terminate
- if $B^\circ < j_q$, $H \leftarrow H \| B$, $B \leftarrow \phi$
- if $B^\circ = j_q$, $B \leftarrow \left(B \oplus B^2_{j_q}\right)$

If only H_2 is exhausted

- if $B = \phi$, $H \leftarrow H \| H_1$, terminate
- if $B^\circ < i_p$, $H \leftarrow H \| B$, $B \leftarrow \phi$
- if $B^\circ = i_p$, $B \leftarrow \left(B \oplus B^1_{i_p}\right)$

Case: *B is empty*

- if $i_p < j_q$, $H \leftarrow H \| B^1_{i_p}$
- if $i_p > j_q$, $H \leftarrow H \| B^2_{j_q}$
- if $i_p = j_q$,
  $B \leftarrow \left(B^1_{i_p} \oplus B^2_{j_q}\right)$

Case: *B is non-empty*

- if $B^\circ = i_p = j_q$, $H \leftarrow H \| B$, $B \leftarrow \left(B^1_{i_p} \oplus B^2_{j_q}\right)$
- if $B^\circ \neq i_p < j_q$, $H \leftarrow H \| B^1_{i_p}$
- if $i_p > j_q \neq B^\circ$, $H \leftarrow H \| B^2_{j_q}$
- if $B^\circ = i_p < j_q$, $B \leftarrow \left(B \oplus B^1_{i_p}\right)$
- if $i_p > j_q = B^\circ$, $B \leftarrow \left(B \oplus B^2_{j_q}\right)$; $B \leftarrow \phi$
- if $B^\circ < i_p = j_q$, $H \leftarrow H \| B$

$O(\lg |H|)$ time, each possible tree position in H processed in $O(1)$ time

Merging two binomial heaps (contd.)

If only H_1 is exhausted

- if $B = \phi$, $H \leftarrow H \| H_2$, terminate
- if $B^\circ < j_q$, $H \leftarrow H \| B$, $B \leftarrow \phi$
- if $B^\circ = j_q$, $B \leftarrow \left(B \oplus B^2_{j_q}\right)$

If only H_2 is exhausted

- if $B = \phi$, $H \leftarrow H \| H_1$, terminate
- if $B^\circ < i_p$, $H \leftarrow H \| B$, $B \leftarrow \phi$
- if $B^\circ = i_p$, $B \leftarrow \left(B \oplus B^1_{i_p}\right)$

Case: *B is empty*

- if $i_p < j_q$, $H \leftarrow H \| B^1_{i_p}$
- if $i_p > j_q$, $H \leftarrow H \| B^2_{j_q}$
- if $i_p = j_q$,
  $B \leftarrow \left(B^1_{i_p} \oplus B^2_{j_q}\right)$

Case: *B is non-empty*

- if $B^\circ = i_p = j_q$, $H \leftarrow H \| B$, $B \leftarrow \left(B^1_{i_p} \oplus B^2_{j_q}\right)$
- if $B^\circ \neq i_p < j_q$, $H \leftarrow H \| B^1_{i_p}$
- if $i_p > j_q \neq B^\circ$, $H \leftarrow H \| B^2_{j_q}$
- if $B^\circ = i_p < j_q$, $B \leftarrow \left(B \oplus B^1_{i_p}\right)$
- if $i_p > j_q = B^\circ$, $B \leftarrow \left(B \oplus B^2_{j_q}\right)$; $B \leftarrow \phi$
- if $B^\circ < i_p = j_q$, $H \leftarrow H \| B$

$O(\lg |H|)$ time, each possible tree position in H processed in $O(1)$ time

Merging two binomial heaps (contd.)

If only H_1 is exhausted

- if $B = \phi$, $H \leftarrow H \| H_2$, terminate
- if $B^\circ < j_q$, $H \leftarrow H \| B$, $B \leftarrow \phi$
- if $B^\circ = j_q$, $B \leftarrow \left(B \oplus B^2_{j_q}\right)$

If only H_2 is exhausted

- if $B = \phi$, $H \leftarrow H \| H_1$, terminate
- if $B^\circ < i_p$, $H \leftarrow H \| B$, $B \leftarrow \phi$
- if $B^\circ = i_p$, $B \leftarrow \left(B \oplus B^1_{i_p}\right)$

Case: *B is empty*

- if $i_p < j_q$, $H \leftarrow H \| B^1_{i_p}$
- if $i_p > j_q$, $H \leftarrow H \| B^2_{j_q}$
- if $i_p = j_q$,
  $B \leftarrow \left(B^1_{i_p} \oplus B^2_{j_q}\right)$

Case: *B is non-empty*

- if $B^\circ = i_p = j_q$, $H \leftarrow H \| B$, $B \leftarrow \left(B^1_{i_p} \oplus B^2_{j_q}\right)$
- if $B^\circ \neq i_p < j_q$, $H \leftarrow H \| B^1_{i_p}$
- if $i_p > j_q \neq B^\circ$, $H \leftarrow H \| B^2_{j_q}$
- if $B^\circ = i_p < j_q$, $B \leftarrow \left(B \oplus B^1_{i_p}\right)$
- if $i_p > j_q = B^\circ$, $B \leftarrow \left(B \oplus B^2_{j_q}\right)$; $B \leftarrow \phi$
- if $B^\circ < i_p = j_q$, $H \leftarrow H \| B$

$O(\lg |H|)$ time, each possible tree position in H processed in $O(1)$ time

Comparison of binary and binomial heaps

Op	Binary	Binomial
Create	$\Theta(1)$	$\Theta(1)$
Merge	$\Theta(n)$	$O(\lg(n))$
FindMin	$\Theta(1)$	$O(\lg(n))$
DelMin	$O(\lg(n))$	$O(\lg(n))$
Insert	$O(\lg(n))$	$O(\lg(n))$
DecKey	$O(\lg(n))$	$O(\lg(n))$

Can FindMin for binomial heaps be improved?

- Yes, with a modification
- Keep track of the tree with the minimum element
- Update on Insert, DelMin, DecKey
- Cost: $\Theta(1)$

A closer look at Insert

- What is the total cost of inserting $n = 2^k$ elements?
- Each element is first inserted as B_0s – cost: 2^k
- Pairs of B_0s are combined as B_1s – cost: $\frac{2^k}{2}$
- Pairs of B_1s are combined as B_2s – cost: $\frac{2^k}{4}$
- Total cost: $2^k + 2^{k-1} + 2^{k-2} + \ldots + 1 = 2^{k+1} - 1 = 2n - 1 \in \Theta(n)$
- Amortised cost (avg cost over n insertions by the aggregate method): $\Theta(1)$

Comparison of binary and binomial heaps

Op	Binary	Binomial
Create	$\Theta(1)$	$\Theta(1)$
Merge	$\Theta(n)$	$O(\lg(n))$
FindMin	$\Theta(1)$	$O(\lg(n))$
DelMin	$O(\lg(n))$	$O(\lg(n))$
Insert	$O(\lg(n))$	$O(\lg(n))$
DecKey	$O(\lg(n))$	$O(\lg(n))$

Can FindMin for binomial heaps be improved?

- Yes, with a modification
- Keep track of the tree with the minimum element
- Update on Insert, DelMin, DecKey
- Cost: $\Theta(1)$

A closer look at Insert

- What is the total cost of inserting $n = 2^k$ elements?
- Each element is first inserted as B_0s – cost: 2^k
- Pairs of B_0s are combined as B_1s – cost: $\frac{2^k}{2}$
- Pairs of B_1s are combined as B_2s – cost: $\frac{2^k}{4}$
- Total cost: $2^k + 2^{k-1} + 2^{k-2} + \ldots + 1 = 2^{k+1} - 1 = 2n - 1 \in \Theta(n)$
- Amortised cost (avg cost over n insertions by the aggregate method): $\Theta(1)$

Comparison of binary and binomial heaps

Op	Binary	Binomial
Create	$\Theta(1)$	$\Theta(1)$
Merge	$\Theta(n)$	$O(\lg(n))$
FindMin	$\Theta(1)$	$O(\lg(n))$
DelMin	$O(\lg(n))$	$O(\lg(n))$
Insert	$O(\lg(n))$	$O(\lg(n))$
DecKey	$O(\lg(n))$	$O(\lg(n))$

Can FindMin for binomial heaps be improved?

- Yes, with a modification
- Keep track of the tree with the minimum element
- Update on Insert, DelMin, DecKey
- Cost: $\Theta(1)$

A closer look at Insert

- What is the total cost of inserting $n = 2^k$ elements?
- Each element is first inserted as B_0s – cost: 2^k
- Pairs of B_0s are combined as B_1s – cost: $\frac{2^k}{2}$
- Pairs of B_1s are combined as B_2s – cost: $\frac{2^k}{4}$
- Total cost: $2^k + 2^{k-1} + 2^{k-2} + \ldots + 1 = 2^{k+1} - 1 = 2n - 1 \in \Theta(n)$
- Amortised cost (avg cost over n insertions by the aggregate method): $\Theta(1)$

Comparison of binary and binomial heaps

Op	Binary	Binomial
Create	$\Theta(1)$	$\Theta(1)$
Merge	$\Theta(n)$	$O(\lg(n))$
FindMin	$\Theta(1)$	$O(\lg(n))$
DelMin	$O(\lg(n))$	$O(\lg(n))$
Insert	$O(\lg(n))$	$O(\lg(n))$
DecKey	$O(\lg(n))$	$O(\lg(n))$

Can FindMin for binomial heaps be improved?

- Yes, with a modification
- Keep track of the tree with the minimum element
- Update on Insert, DelMin, DecKey
- Cost: $\Theta(1)$

A closer look at Insert

- What is the total cost of inserting $n = 2^k$ elements?
- Each element is first inserted as B_0s – cost: 2^k
- Pairs of B_0s are combined as B_1s – cost: $\frac{2^k}{2}$
- Pairs of B_1s are combined as B_2s – cost: $\frac{2^k}{4}$
- Total cost: $2^k + 2^{k-1} + 2^{k-2} + \ldots + 1 = 2^{k+1} - 1 = 2n - 1 \in \Theta(n)$
- Amortised cost (avg cost over n insertions by the aggregate method): $\Theta(1)$

Comparison of binary and binomial heaps

Op	Binary	Binomial
Create	$\Theta(1)$	$\Theta(1)$
Merge	$\Theta(n)$	$O(\lg(n))$
FindMin	$\Theta(1)$	$O(\lg(n))$
DelMin	$O(\lg(n))$	$O(\lg(n))$
Insert	$O(\lg(n))$	$O(\lg(n))$
DecKey	$O(\lg(n))$	$O(\lg(n))$

Can FindMin for binomial heaps be improved?

- Yes, with a modification
- Keep track of the tree with the minimum element
- Update on Insert, DelMin, DecKey
- Cost: $\Theta(1)$

A closer look at Insert

- What is the total cost of inserting $n = 2^k$ elements?
- Each element is first inserted as B_0s – cost: 2^k
- Pairs of B_0s are combined as B_1s – cost: $\frac{2^k}{2}$
- Pairs of B_1s are combined as B_2s – cost: $\frac{2^k}{4}$
- Total cost: $2^k + 2^{k-1} + 2^{k-2} + \ldots + 1 = 2^{k+1} - 1 = 2n - 1 \in \Theta(n)$
- Amortised cost (avg cost over n insertions by the aggregate method): $\Theta(1)$

Comparison of binary and binomial heaps

Op	Binary	Binomial
Create	$\Theta(1)$	$\Theta(1)$
Merge	$\Theta(n)$	$O(\lg(n))$
FindMin	$\Theta(1)$	$O(\lg(n))$
DelMin	$O(\lg(n))$	$O(\lg(n))$
Insert	$O(\lg(n))$	$O(\lg(n))$
DecKey	$O(\lg(n))$	$O(\lg(n))$

Can FindMin for binomial heaps be improved?

- Yes, with a modification
- Keep track of the tree with the minimum element
- Update on Insert, DelMin, DecKey
- Cost: $\Theta(1)$

A closer look at Insert

- What is the total cost of inserting $n = 2^k$ elements?
- Each element is first inserted as B_0s – cost: 2^k
- Pairs of B_0s are combined as B_1s – cost: $\frac{2^k}{2}$
- Pairs of B_1s are combined as B_2s – cost: $\frac{2^k}{4}$
- Total cost: $2^k + 2^{k-1} + 2^{k-2} + \ldots + 1 = 2^{k+1} - 1 = 2n - 1 \in \Theta(n)$
- Amortised cost (avg cost over n insertions by the aggregate method): $\Theta(1)$

Comparison of binary and binomial heaps

Op	Binary	Binomial
Create	$\Theta(1)$	$\Theta(1)$
Merge	$\Theta(n)$	$O(\lg(n))$
FindMin	$\Theta(1)$	$O(\lg(n))$
DelMin	$O(\lg(n))$	$O(\lg(n))$
Insert	$O(\lg(n))$	$O(\lg(n))$
DecKey	$O(\lg(n))$	$O(\lg(n))$

Can FindMin for binomial heaps be improved?

- Yes, with a modification
- Keep track of the tree with the minimum element
- Update on Insert, DelMin, DecKey
- Cost: $\Theta(1)$

A closer look at Insert

- What is the total cost of inserting $n = 2^k$ elements?
- Each element is first inserted as B_0s – cost: 2^k
- Pairs of B_0s are combined as B_1s – cost: $\frac{2^k}{2}$
- Pairs of B_1s are combined as B_2s – cost: $\frac{2^k}{4}$
- Total cost: $2^k + 2^{k-1} + 2^{k-2} + \ldots + 1 = 2^{k+1} - 1 = 2n - 1 \in \Theta(n)$
- Amortised cost (avg cost over n insertions by the aggregate method): $\Theta(1)$

Comparison of binary and binomial heaps

Op	Binary	Binomial
Create	$\Theta(1)$	$\Theta(1)$
Merge	$\Theta(n)$	$O(\lg(n))$
FindMin	$\Theta(1)$	$O(\lg(n))$
DelMin	$O(\lg(n))$	$O(\lg(n))$
Insert	$O(\lg(n))$	$O(\lg(n))$
DecKey	$O(\lg(n))$	$O(\lg(n))$

Can FindMin for binomial heaps be improved?

- Yes, with a modification
- Keep track of the tree with the minimum element
- Update on Insert, DelMin, DecKey
- Cost: $\Theta(1)$

A closer look at Insert

- What is the total cost of inserting $n = 2^k$ elements?
- Each element is first inserted as B_0s – cost: 2^k
- Pairs of B_0s are combined as B_1s – cost: $\frac{2^k}{2}$
- Pairs of B_1s are combined as B_2s – cost: $\frac{2^k}{4}$
- Total cost: $2^k + 2^{k-1} + 2^{k-2} + \ldots + 1 = 2^{k+1} - 1 = 2n - 1 \in \Theta(n)$
- Amortised cost (avg cost over n insertions by the aggregate method): $\Theta(1)$

Amortised accounting analysis of Insert

- Charge each item two units for insertion
- One unit is used immediately to insert key as a B_0 tree in the list of binomial trees
- The other unit is saved as a credit
- At times binomial trees of the same degree/order need to be merged
- Assume B_{i_1} and B_{i_2} each have one saved credit
- One unit of credit is used up to merge them and the other stays with $\left(B_{i_1} \oplus B_{i_2}\right)$
- Thus, the trees never run of credit through the process of merging
- Hence, insertion is done with $\Theta(1)$ amortised cost

Amortised accounting analysis of Insert

- Charge each item two units for insertion
- One unit is used immediately to insert key as a B_0 tree in the list of binomial trees
- The other unit is saved as a credit
- At times binomial trees of the same degree/order need to be merged
- Assume B_{i_1} and B_{i_2} each have one saved credit
- One unit of credit is used up to merge them and the other stays with $\left(B_{i_1} \oplus B_{i_2}\right)$
- Thus, the trees never run of credit through the process of merging
- Hence, insertion is done with $\Theta(1)$ amortised cost

Amortised accounting analysis of Insert

- Charge each item two units for insertion
- One unit is used immediately to insert key as a B_0 tree in the list of binomial trees
- The other unit is saved as a credit
- At times binomial trees of the same degree/order need to be merged
- Assume B_{i_1} and B_{i_2} each have one saved credit
- One unit of credit is used up to merge them and the other stays with $\left(B_{i_1} \oplus B_{i_2}\right)$
- Thus, the trees never run of credit through the process of merging
- Hence, insertion is done with $\Theta(1)$ amortised cost

Section outline

3 **Lazy binomial heaps**
- Lazy merge of binomial heaps
- Coalescing trees
- Example of LBH operations
- Cost of deleting minimum element
- Summary of lazy binomial heap operation costs

Lazy merge of binomial heaps

- The binomial trees may be linked together in a doubly linked list (*why, really necessary*?)
- Merge is performed by just stitching the two linked lists together – easily done with doubly linked lists (*what about just linked lists*?)
- Merging of trees of identical rank/order is *not immediately* done – hence lazy
- Each heap has its min-pointer, the new list has as its min-pointer the minimum of the two values at the min-pointers of the consituent trees
- Cost: $\Theta(1)$
- Note that if the minimum element is deleted, it will be necessary to traverse through the entire list of trees to identify the new minimum
- Number of trees in the heap grows with insert, merge and delete
- After coalescing the number of trees are back to $O(\lg n)$

Lazy merge of binomial heaps

- The binomial trees may be linked together in a doubly linked list (*why, really necessary*?)
- Merge is performed by just stitching the two linked lists together – easily done with doubly linked lists (*what about just linked lists*?)
- Merging of trees of identical rank/order is *not immediately* done – hence lazy
- Each heap has its min-pointer, the new list has as its min-pointer the minimum of the two values at the min-pointers of the consituent trees
- Cost: $\Theta(1)$
- Note that if the minimum element is deleted, it will be necessary to traverse through the entire list of trees to identify the new minimum
- Number of trees in the heap grows with insert, merge and delete
- After coalescing the number of trees are back to $O(\lg n)$

Lazy merge of binomial heaps

- The binomial trees may be linked together in a doubly linked list (*why, really necessary*?)
- Merge is performed by just stitching the two linked lists together – easily done with doubly linked lists (*what about just linked lists*?)
- Merging of trees of identical rank/order is *not immediately* done – hence lazy
- Each heap has its min-pointer, the new list has as its min-pointer the minimum of the two values at the min-pointers of the consituent trees
- Cost: $\Theta(1)$
- Note that if the minimum element is deleted, it will be necessary to traverse through the entire list of trees to identify the new minimum
- Number of trees in the heap grows with insert, merge and delete
- After coalescing the number of trees are back to $O(\lg n)$

Lazy merge of binomial heaps

- The binomial trees may be linked together in a doubly linked list (*why, really necessary*?)
- Merge is performed by just stitching the two linked lists together – easily done with doubly linked lists (*what about just linked lists*?)
- Merging of trees of identical rank/order is *not immediately* done – hence lazy
- Each heap has its min-pointer, the new list has as its min-pointer the minimum of the two values at the min-pointers of the consituent trees
- Cost: $\Theta(1)$
- Note that if the minimum element is deleted, it will be necessary to traverse through the entire list of trees to identify the new minimum
- Number of trees in the heap grows with insert, merge and delete
- After coalescing the number of trees are back to $O(\lg n)$

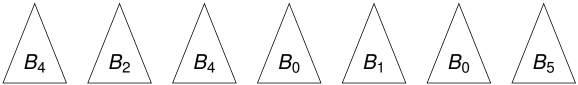
Lazy merge of binomial heaps

- The binomial trees may be linked together in a doubly linked list (*why, really necessary*?)
- Merge is performed by just stitching the two linked lists together – easily done with doubly linked lists (*what about just linked lists*?)
- Merging of trees of identical rank/order is *not immediately* done – hence lazy
- Each heap has its min-pointer, the new list has as its min-pointer the minimum of the two values at the min-pointers of the consituent trees
- Cost: $\Theta(1)$
- Note that if the minimum element is deleted, it will be necessary to traverse through the entire list of trees to identify the new minimum
- Number of trees in the heap grows with insert, merge and delete
- After coalescing the number of trees are back to $O(\lg n)$

Coalescing trees

- The following sequence of heaps do not satisfy the required order for binomial heaps
- They are formed by a sequence of lazy merge operations (lazy insert, lazy merge)
- Coalescing may be done after a DelMax or DelMin operation because the list of heaps will have to be traversed to identify the new min/max element

H:

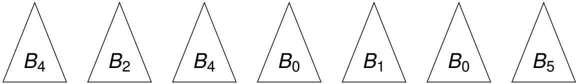


BinomT order	0	1	2	3	4	5	6
Coalesced trees	B_0	B_1	B_2	B_3	B_4	B_5	B_6

Coalescing trees

- The following sequence of heaps do not satisfy the required order for binomial heaps
- They are formed by a sequence of lazy merge operations (lazy insert, lazy merge)
- Coalescing may be done after a DelMax or DelMin operation because the list of heaps will have to be traversed to identify the new min/max element

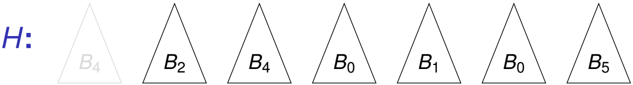
H:



BinomT order	0	1	2	3	4	5	6
Coalesced trees	B_0	B_1	B_2	B_3	B_4	B_5	B_6

Coalescing trees

- The following sequence of heaps do not satisfy the required order for binomial heaps
- They are formed by a sequence of lazy merge operations (lazy insert, lazy merge)
- Coalescing may be done after a DelMax or DelMin operation because the list of heaps will have to be traversed to identify the new min/max element

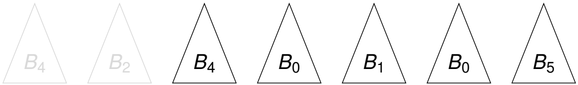


BinomT order	0	1	2	3	4	5	6
Coalesced trees	B_0	B_1	B_2	B_3	B_4	B_5	B_6

Coalescing trees

- The following sequence of heaps do not satisfy the required order for binomial heaps
- They are formed by a sequence of lazy merge operations (lazy insert, lazy merge)
- Coalescing may be done after a DelMax or DelMin operation because the list of heaps will have to be traversed to identify the new min/max element

H:

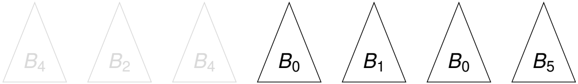


BinomT order	0	1	2	3	4	5	6
Coalesced trees	B_0	B_1	B_2	B_3	B_4	B_5	B_6

Coalescing trees

- The following sequence of heaps do not satisfy the required order for binomial heaps
- They are formed by a sequence of lazy merge operations (lazy insert, lazy merge)
- Coalescing may be done after a DelMax or DelMin operation because the list of heaps will have to be traversed to identify the new min/max element

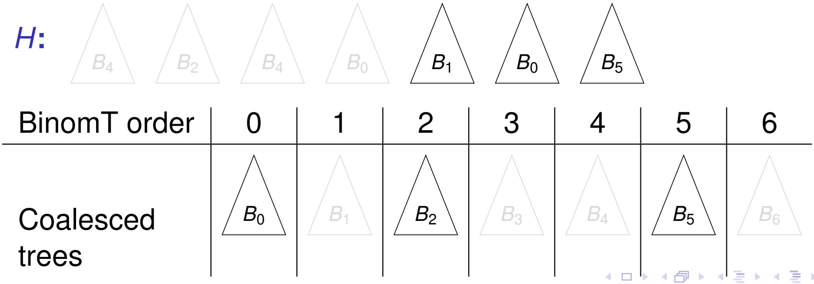
H:



BinomT order	0	1	2	3	4	5	6
Coalesced trees	B_0	B_1	B_2	B_3	B_4	B_5	B_6

Coalescing trees

- The following sequence of heaps do not satisfy the required order for binomial heaps
- They are formed by a sequence of lazy merge operations (lazy insert, lazy merge)
- Coalescing may be done after a DelMax or DelMin operation because the list of heaps will have to be traversed to identify the new min/max element



Coalescing trees

- The following sequence of heaps do not satisfy the required order for binomial heaps
- They are formed by a sequence of lazy merge operations (lazy insert, lazy merge)
- Coalescing may be done after a DelMax or DelMin operation because the list of heaps will have to be traversed to identify the new min/max element

H:

BinomT order	0	1	2	3	4	5	6
Coalesced trees	B_0	B_1	B_2	B_3	B_4	B_5	B_6

Coalescing trees

- The following sequence of heaps do not satisfy the required order for binomial heaps
- They are formed by a sequence of lazy merge operations (lazy insert, lazy merge)
- Coalescing may be done after a DelMax or DelMin operation because the list of heaps will have to be traversed to identify the new min/max element

H:

BinomT order	0	1	2	3	4	5	6
Coalesced trees	B_0	B_1	B_2	B_3	B_4	B_5	B_6

Coalescing trees

- The following sequence of heaps do not satisfy the required order for binomial heaps
- They are formed by a sequence of lazy merge operations (lazy insert, lazy merge)
- Coalescing may be done after a DelMax or DelMin operation because the list of heaps will have to be traversed to identify the new min/max element

H:

BinomT order	0	1	2	3	4	5	6
Coalesced trees	B_0	B_1	B_2	B_3	B_4	B_5	B_6

Coalescing trees (contd.)

- A binomial heap with n keys needs $(1 + \lg n) = m$ binomial trees
- With lazy merging the trees in the heap are not of unique rank and also ordered
- Maintain an m-place vector $\mathbf{V}$ for trees $B_0, B_1, \ldots, B_m$
- While handling T in the list, check $\mathbf{V}[T°]$
  - if ϕ, $\mathbf{V}[T°] \leftarrow T$
  - otherwise, $T \leftarrow (T \oplus \mathbf{V}[T°])$, $\mathbf{V}[T° = 1] \leftarrow \phi$ and continue
- Finally, stitch the trees in $\mathbf{V}$ in the linked list
- Needs to be done only for deleting the minimum element, worst case time $O(n)$, as all preceeding operations could be only inserts

Coalescing trees (contd.)

- A binomial heap with n keys needs $(1 + \lg n) = m$ binomial trees
- With lazy merging the trees in the heap are not of unique rank and also ordered
- Maintain an m-place vector $\mathbf{V}$ for trees $B_0, B_1, \ldots, B_m$
- While handling T in the list, check $\mathbf{V}[T^\circ]$
  - if ϕ, $\mathbf{V}[T^\circ] \leftarrow T$
  - otherwise, $T \leftarrow (T \oplus \mathbf{V}[T^\circ])$, $\mathbf{V}[T^\circ = 1] \leftarrow \phi$ and continue
- Finally, stitch the trees in $\mathbf{V}$ in the linked list
- Needs to be done only for deleting the minimum element, worst case time $O(n)$, as all preceeding operations could be only inserts

Coalescing trees (contd.)

- A binomial heap with n keys needs $(1 + \lg n) = m$ binomial trees
- With lazy merging the trees in the heap are not of unique rank and also ordered
- Maintain an m-place vector $\mathbf{V}$ for trees $B_0, B_1, \ldots, B_m$
- While handling T in the list, check $\mathbf{V}[T^\circ]$
  - if ϕ, $\mathbf{V}[T^\circ] \leftarrow T$
  - otherwise, $T \leftarrow (T \oplus \mathbf{V}[T^\circ])$, $\mathbf{V}[T^\circ = 1] \leftarrow \phi$ and continue
- Finally, stitch the trees in $\mathbf{V}$ in the linked list
- Needs to be done only for deleting the minimum element, worst case time $O(n)$, as all preceeding operations could be only inserts

Example of LBH operations

Example (LBH operations)

Carry out the following operations on a min-lazy binomial heap:

insert 2, 5, 8, 11, 9, 7, 4, 3

delMin give count of the operations performed

decKey 9 $\rightarrow$ 1, show details of the affected tree and the operations performed

insert 13 18

delMin details of affected trees and op count

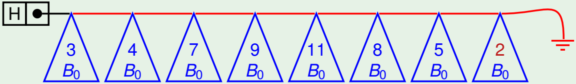
decKey 19 $\rightarrow$ 6, details of affected trees and op count

Example of LBH operations

Example (LBH operations)

Carry out the following operations on a min-lazy binomial heap:

insert 2, 5, 8, 11, 9, 7, 4, 3



delMin give count of the operations performed

decKey $9 \rightarrow 1$, show details of the affected tree and the operations performed

insert 13 18

delMin details of affected trees and op count

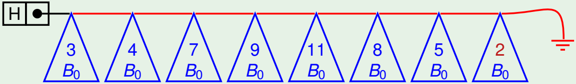
decKey $19 \rightarrow 6$, details of affected trees and op count

Example of LBH operations

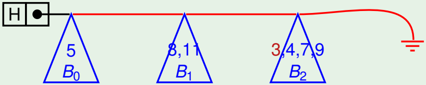
Example (LBH operations)

Carry out the following operations on a min-lazy binomial heap:

insert 2, 5, 8, 11, 9, 7, 4, 3



delMin give count of the operations performed



decKey 9 → 1, show details of the affected tree and the operations performed

insert 13 18

delMin details of affected trees and op count

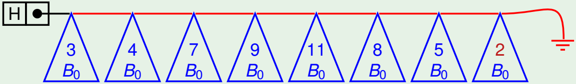
decKey 19 → 6, details of affected trees and op count

Example of LBH operations

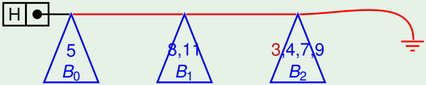
Example (LBH operations)

Carry out the following operations on a min-lazy binomial heap:

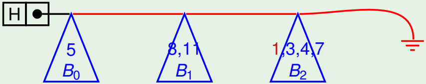
insert 2, 5, 8, 11, 9, 7, 4, 3



delMin give count of the operations performed



decKey 9 $\to$ 1, show details of the affected tree and the operations performed



insert 13 18

delMin details of affected trees and op count

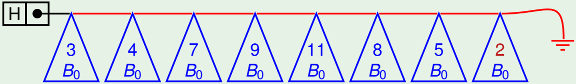
decKey 19 $\to$ 6, details of affected trees and op count

Example of LBH operations

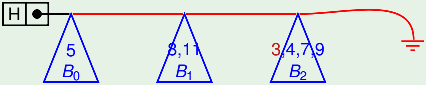
Example (LBH operations)

Carry out the following operations on a min-lazy binomial heap:

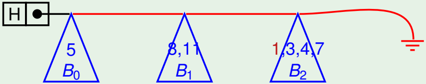
insert 2, 5, 8, 11, 9, 7, 4, 3



delMin give count of the operations performed



decKey 9 → 1, show details of the affected tree and the operations performed



insert 13 18

delMin details of affected trees and op count

decKey 19 → 6, details of affected trees and op count

Cost of deleting minimum element

- A binomial heap with lazy merge has these worst-case time bounds:
  - Insert: $O(1)$
  - Merge: $O(1)$
  - FindMin: $O(1)$
  - DelMin: $O(n)$
  - DecKey: $O(\lg n)$
- These are worst-case time bounds
- Intuitively, DelMin does not have to do badly all the time!

  The coalescing activity of Insert has been transferred to DelMin!

- The worst case time complexity for DelMin for regular binomial heaps was $O(\lg n)$
- The amortised cost of $\Theta(1)$ of Insert has been added to the amortised cost of DelMin
- The amortised cost of DelMin is $O(\Theta(1) + \lg n) \in O(\lg n)$

Cost of deleting minimum element

- A binomial heap with lazy merge has these worst-case time bounds:
  - Insert: $O(1)$
  - Merge: $O(1)$
  - FindMin: $O(1)$
  - DelMin: $O(n)$
  - DecKey: $O(\lg n)$
- These are worst-case time bounds
- Intuitively, DelMin does not have to do badly all the time!

The coalescing activity of Insert has been transferred to DelMin!

- The worst case time complexity for DelMin for regular binomial heaps was $O(\lg n)$
- The amortised cost of $\Theta(1)$ of Insert has been added to the amortised cost of DelMin
- The amortised cost of DelMin is $O(\Theta(1) + \lg n) \in O(\lg n)$

Cost of deleting minimum element

- A binomial heap with lazy merge has these worst-case time bounds:
  - Insert: $O(1)$
  - Merge: $O(1)$
  - FindMin: $O(1)$
  - DelMin: $O(n)$
  - DecKey: $O(\lg n)$
- These are worst-case time bounds
- Intuitively, DelMin does not have to do badly all the time!

The coalescing activity of Insert has been transferred to DelMin!

- The worst case time complexity for DelMin for regular binomial heaps was $O(\lg n)$
- The amortised cost of $\Theta(1)$ of Insert has been added to the amortised cost of DelMin
- The amortised cost of DelMin is $O(\Theta(1) + \lg n) \in O(\lg n)$

Cost of deleting minimum element

- A binomial heap with lazy merge has these worst-case time bounds:
  - Insert: $O(1)$
  - Merge: $O(1)$
  - FindMin: $O(1)$
  - DelMin: $O(n)$
  - DecKey: $O(\lg n)$
- These are worst-case time bounds
- Intuitively, DelMin does not have to do badly all the time!

The coalescing activity of Insert has been transferred to DelMin!

- The worst case time complexity for DelMin for regular binomial heaps was $O(\lg n)$
- The amortised cost of $\Theta(1)$ of Insert has been added to the amortised cost of DelMin
- The amortised cost of DelMin is $O(\Theta(1) + \lg n) \in O(\lg n)$

Summary of lazy binomial heap operation costs

- The amortised costs of the operations on a lazy binomial heap are as follows:
  - Insert: $O(1)$
  - Merge: $O(1)$
  - FindMin: $O(1)$
  - DelMin: $O(\lg n)$
  - DecKey: $O(\lg n)$
- Any series of e insert operations mixed with d DelMin operations will take time $O(e + d \lg e)$
- Can anything be done about DecKey?

  Can the cost of DecKey be suitably amortised and pushed on to DelMin – as was done for Insert?
- What is special about DecKey that must be avoided?

  Percolation! For that would entail $O(\lg n)$ steps (in the worst case)

Summary of lazy binomial heap operation costs

- The amortised costs of the operations on a lazy binomial heap are as follows:
  - Insert: $O(1)$
  - Merge: $O(1)$
  - FindMin: $O(1)$
  - DelMin: $O(\lg n)$
  - DecKey: $O(\lg n)$
- Any series of e insert operations mixed with d DelMin operations will take time $O(e + d \lg e)$
- Can anything be done about DecKey?

  Can the cost of DecKey be suitably amortised and pushed on to DelMin – as was done for Insert?
- What is special about DecKey that must be avoided?

  Percolation! For that would entail $O(\lg n)$ steps (in the worst case)

Summary of lazy binomial heap operation costs

- The amortised costs of the operations on a lazy binomial heap are as follows:
  - Insert: $O(1)$
  - Merge: $O(1)$
  - FindMin: $O(1)$
  - DelMin: $O(\lg n)$
  - DecKey: $O(\lg n)$
- Any series of e insert operations mixed with d DelMin operations will take time $O(e + d \lg e)$
- Can anything be done about DecKey?

Can the cost of DecKey be suitably amortised and pushed on to DelMin – as was done for Insert?

- What is special about DecKey that must be avoided?

Percolation! For that would entail $O(\lg n)$ steps (in the worst case)

Summary of lazy binomial heap operation costs

- The amortised costs of the operations on a lazy binomial heap are as follows:
  - Insert: $O(1)$
  - Merge: $O(1)$
  - FindMin: $O(1)$
  - DelMin: $O(\lg n)$
  - DecKey: $O(\lg n)$
- Any series of e insert operations mixed with d DelMin operations will take time $O(e + d \lg e)$
- Can anything be done about DecKey?

Can the cost of DecKey be suitably amortised and pushed on to DelMin – as was done for Insert?

- What is special about DecKey that must be avoided?

Percolation! For that would entail $O(\lg n)$ steps (in the worst case)

Summary of lazy binomial heap operation costs

- The amortised costs of the operations on a lazy binomial heap are as follows:
  - Insert: $O(1)$
  - Merge: $O(1)$
  - FindMin: $O(1)$
  - DelMin: $O(\lg n)$
  - DecKey: $O(\lg n)$
- Any series of e insert operations mixed with d DelMin operations will take time $O(e + d \lg e)$
- Can anything be done about DecKey?

Can the cost of DecKey be suitably amortised and pushed on to DelMin – as was done for Insert?

- What is special about DecKey that must be avoided?

Percolation! For that would entail $O(\lg n)$ steps (in the worst case)

Section outline

4 **Fibonacci heaps**

- Relation with binomial heaps
- Restricting excessive damage through DecKey
- Minimum rank of a node in a Fibonacci heap
- Max damage to binomial trees in Fibonacci heap
- Maximally damaged trees in Fibonacci heaps
- Example of FH operations
- Costing time taken for DecKey
- Costing time taken for DecKey with DelMin
- Charges and invariants for Fibonacci heaps
- Comparison of heaps
- Representation of a Fibonacci heap

Relation with binomial heaps

- Fibonacci heaps, developed by Fredman and Tarjan in 1986, are very similar to lazy binomial heaps
- If the reduction of the key does not violate the heap property, then nothing needs to be done
- Otherwise, there is a radical departure for DecKey

  In order to avoid the $O(\lg n)$ cost entailed by percolation, the node is simply cut out of the tree and entered into the list of trees!

- There are consequences
  - The trees in a Fibonacci heap may not be binomial trees
  - There is risk of too many nodes getting deleted
  - If nodes are deleted arbitrarily, the height of the a tree may no longer be logarithmic in the number of nodes in the tree
  - So, some damage control mechanism is needed

Relation with binomial heaps

- Fibonacci heaps, developed by Fredman and Tarjan in 1986, are very similar to lazy binomial heaps
- If the reduction of the key does not violate the heap property, then nothing needs to be done
- Otherwise, there is a radical departure for DecKey

In order to avoid the $O(\lg n)$ cost entailed by percolation, the node is simply cut out of the tree and entered into the list of trees!

- There are consequences
  - The trees in a Fibonacci heap may not be binomial trees
  - There is risk of too many nodes getting deleted
  - If nodes are deleted arbitrarily, the height of the a tree may no longer be logarithmic in the number of nodes in the tree
  - So, some damage control mechanism is needed

Relation with binomial heaps

- Fibonacci heaps, developed by Fredman and Tarjan in 1986, are very similar to lazy binomial heaps
- If the reduction of the key does not violate the heap property, then nothing needs to be done
- Otherwise, there is a radical departure for DecKey

  In order to avoid the $O(\lg n)$ cost entailed by percolation, the node is simply cut out of the tree and entered into the list of trees!

- There are consequences
  - The trees in a Fibonacci heap may not be binomial trees
  - There is risk of too many nodes getting deleted
  - If nodes are deleted arbitrarily, the height of the a tree may no longer be logarithmic in the number of nodes in the tree
  - So, some damage control mechanism is needed

Relation with binomial heaps

- Fibonacci heaps, developed by Fredman and Tarjan in 1986, are very similar to lazy binomial heaps
- If the reduction of the key does not violate the heap property, then nothing needs to be done
- Otherwise, there is a radical departure for DecKey

  In order to avoid the $O(\lg n)$ cost entailed by percolation, the node is simply cut out of the tree and entered into the list of trees!

- There are consequences
  - The trees in a Fibonacci heap may not be binomial trees
  - There is risk of too many nodes getting deleted
  - If nodes are deleted arbitrarily, the height of the a tree may no longer be logarithmic in the number of nodes in the tree
  - So, some damage control mechanism is needed

Relation with binomial heaps

- Fibonacci heaps, developed by Fredman and Tarjan in 1986, are very similar to lazy binomial heaps
- If the reduction of the key does not violate the heap property, then nothing needs to be done
- Otherwise, there is a radical departure for DecKey

In order to avoid the $O(\lg n)$ cost entailed by percolation, the node is simply cut out of the tree and entered into the list of trees!

- There are consequences
  - The trees in a Fibonacci heap may not be binomial trees
  - There is risk of too many nodes getting deleted
  - If nodes are deleted arbitrarily, the height of the a tree may no longer be logarithmic in the number of nodes in the tree
  - So, some damage control mechanism is needed

Restricting excessive damage through DecKey

- Mark the (non-root) parent of the deleted node, if not marked
- If the parent is already marked, indicating that it has already lost a child, it is also removed along with its subtree and added to the root list and unmarked
- These cuts can be cascading, as ancestor nodes could also be marked earlier
- This measure ensures that a node having lost more that a single child does not remain within the tree

How is this supposed to help?

- Coalescing of trees ensure that no two trees in the heap have the same rank/order
- If the number of nodes in each tree is shown to be exponential in its rank/order, then the number of trees in the heap will be logarithmic in the number of nodes in the heap
- That helps to ensure that heap merging is done in lg n time

Restricting excessive damage through DecKey

- Mark the (non-root) parent of the deleted node, if not marked
- If the parent is already marked, indicating that it has already lost a child, it is also removed along with its subtree and added to the root list and unmarked
- These cuts can be cascading, as ancestor nodes could also be marked earlier
- This measure ensures that a node having lost more that a single child does not remain within the tree

How is this supposed to help?

- Coalescing of trees ensure that no two trees in the heap have the same rank/order
- If the number of nodes in each tree is shown to be exponential in its rank/order, then the number of trees in the heap will be logarithmic in the number of nodes in the heap
- That helps to ensure that heap merging is done in lg n time

Restricting excessive damage through DecKey

- Mark the (non-root) parent of the deleted node, if not marked
- If the parent is already marked, indicating that it has already lost a child, it is also removed along with its subtree and added to the root list and unmarked
- These cuts can be cascading, as ancestor nodes could also be marked earlier
- This measure ensures that a node having lost more that a single child does not remain within the tree

How is this supposed to help?

- Coalescing of trees ensure that no two trees in the heap have the same rank/order
- If the number of nodes in each tree is shown to be exponential in its rank/order, then the number of trees in the heap will be logarithmic in the number of nodes in the heap
- That helps to ensure that heap merging is done in lg n time

Restricting excessive damage through DecKey

- Mark the (non-root) parent of the deleted node, if not marked
- If the parent is already marked, indicating that it has already lost a child, it is also removed along with its subtree and added to the root list and unmarked
- These cuts can be cascading, as ancestor nodes could also be marked earlier
- This measure ensures that a node having lost more that a single child does not remain within the tree

How is this supposed to help?

- Coalescing of trees ensure that no two trees in the heap have the same rank/order
- If the number of nodes in each tree is shown to be exponential in its rank/order, then the number of trees in the heap will be logarithmic in the number of nodes in the heap
- That helps to ensure that heap merging is done in lg n time

Steps for DecKey

- Let y be the parent of x.
- After decreasing key[x], if key[x]<key[y], mark x
- Repeat the following step until x is unmarked:
  - Insert x to the root list.
  - Unmark x if x is marked.
  - Adjust min[H] if key[min[H]]>key[x]
  - Eliminate x from the list of children; decrease degree[y] by 1
  - If y is marked, then set x to y, set y to parent[x], otherwise, if y is not a root, then mark y

Minimum rank of a node in a Fibonacci heap

Lemma

Let X be any node in the tree of a Fibonacci heap. Let C be the i^{th} youngest child of X, at the time of linking to X, then the rank of C is at least $i - 2$

Proof.

- At the time of linking C to X as the i^{th} child, $i - 1$ earlier children would have been present
- Rank of X, at the time of linking C would be $i - 1$
- Rank of C, at the time of linking C would be also be $i - 1$ (why ?)

  – because only trees of the same rank are linked
- C could lose at most one child in the future, until it is cut off from X
- Its rank is at at least $i - 2$

□

Minimum rank of a node in a Fibonacci heap

Lemma

Let X be any node in the tree of a Fibonacci heap. Let C be the i^{th} youngest child of X, at the time of linking to X, then the rank of C is at least $i - 2$

Proof.

- At the time of linking C to X as the i^{th} child, $i - 1$ earlier children would have been present
- Rank of X, at the time of linking C would be $i - 1$
- Rank of C, at the time of linking C would be also be $i - 1$ (why ?)

  – because only trees of the same rank are linked
- C could lose at most one child in the future, until it is cut off from X
- Its rank is at at least $i - 2$

Minimum rank of a node in a Fibonacci heap

Lemma

Let X be any node in the tree of a Fibonacci heap. Let C be the i^{th} youngest child of X, at the time of linking to X, then the rank of C is at least $i-2$

Proof.

- At the time of linking C to X as the i^{th} child, $i-1$ earlier children would have been present
- Rank of X, at the time of linking C would be $i-1$
- Rank of C, at the time of linking C would be also be $i-1$ (why ?)
  – because only trees of the same rank are linked
- C could lose at most one child in the future, until it is cut off from X
- Its rank is at at least $i-2$

Minimum rank of a node in a Fibonacci heap

Lemma

Let X be any node in the tree of a Fibonacci heap. Let C be the i^{th} youngest child of X, at the time of linking to X, then the rank of C is at least $i - 2$

Proof.

- At the time of linking C to X as the i^{th} child, $i - 1$ earlier children would have been present
- Rank of X, at the time of linking C would be $i - 1$
- Rank of C, at the time of linking C would be also be $i - 1$ (why ?)

  – because only trees of the same rank are linked
- C could lose at most one child in the future, until it is cut off from X
- Its rank is at at least $i - 2$

□

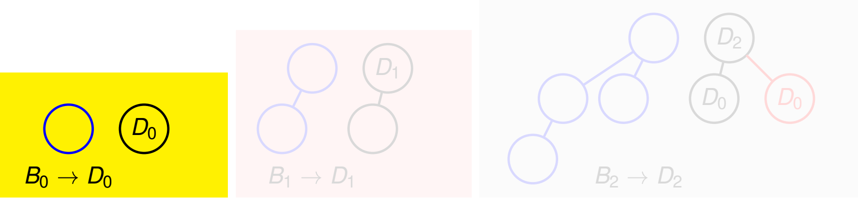
Evaluating the damage done by Deckey

- If DecKey is never done, the Fibonacci heap remains structurally identical to a binomial heap
- Each tree in the heap is a binomial tree
- Each tree of rank/order k has 2^k nodes in it
- Maximum rank of a tree in a such heap of n nodes is $O(\lg n)$
- On the other hand, suppose that all trees in the binomial heap have lost the maximum possible number of nodes
- In that case, how many nodes will each such maximally damaged tree have?

Evaluating the damage done by Deckey

- If DecKey is never done, the Fibonacci heap remains structurally identical to a binomial heap
- Each tree in the heap is a binomial tree
- Each tree of rank/order k has 2^k nodes in it
- Maximum rank of a tree in a such heap of n nodes is $O(\lg n)$
- On the other hand, suppose that all trees in the binomial heap have lost the maximum possible number of nodes
- In that case, how many nodes will each such maximally damaged tree have?

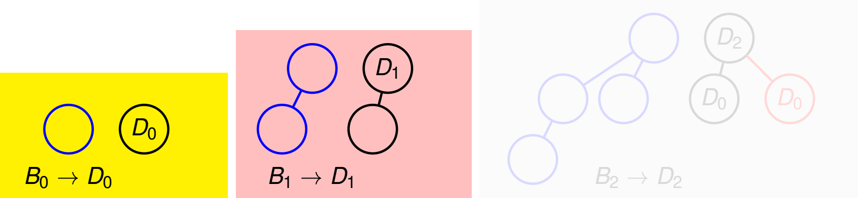
Max damage to binomial trees in Fibonacci heap



During damage, children of root of B_k retained to obtain D_k

- Inferior $B_2 \equiv C_3$ can lose only one child (why?)
- By lemma, 3rd child of B_3, $B_2 \equiv C_3 \xrightarrow{\text{damage}} D_{3-2}$ or D_1
- For B_k with $C_1, C_2, \ldots, C_k$ as children D_k has $D_0, D_{2-2}, D_{3-2}, \ldots, D_{k-2}$ or $D_0, D_0, D_1, \ldots, D_{k-2}$ as children

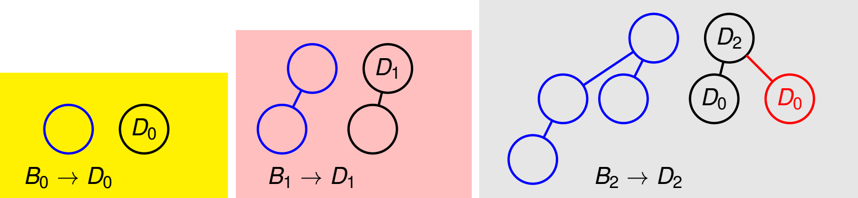
Max damage to binomial trees in Fibonacci heap



During damage, children of root of B_k retained to obtain D_k

- Inferior $B_2 \equiv C_3$ can lose only one child (why?)
- By lemma, 3rd child of B_3, $B_2 \equiv C_3 \xrightarrow{\text{damage}} D_{3-2}$ or D_1
- For B_k with $C_1, C_2, \ldots, C_k$ as children D_k has D_0, $D_{2-2}, D_{3-2}, \ldots, D_{k-2}$ or $D_0, D_0, D_1, \ldots, D_{k-2}$ as children

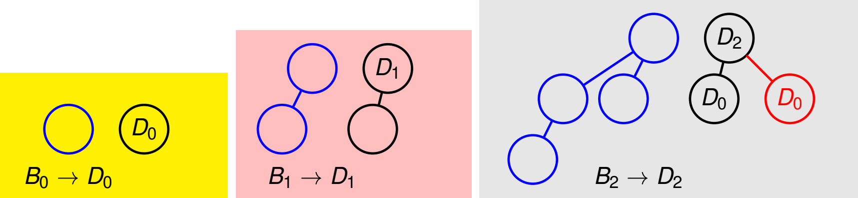
Max damage to binomial trees in Fibonacci heap



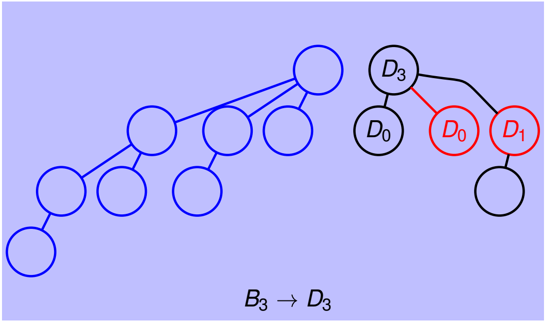
During damage, children of root of B_k retained to obtain D_k

- Inferior $B_2 \equiv C_3$ can lose only one child (why?)
- By lemma, 3[rd] child of B_3, $B_2 \equiv C_3 \xrightarrow{\text{damage}} D_{3-2}$ or D_1
- For B_k with $C_1, C_2, \ldots, C_k$ as children D_k has D_0, $D_{2-2}, D_{3-2}, \ldots, D_{k-2}$ or $D_0, D_0, D_1, \ldots, D_{k-2}$ as children

Max damage to binomial trees in Fibonacci heap

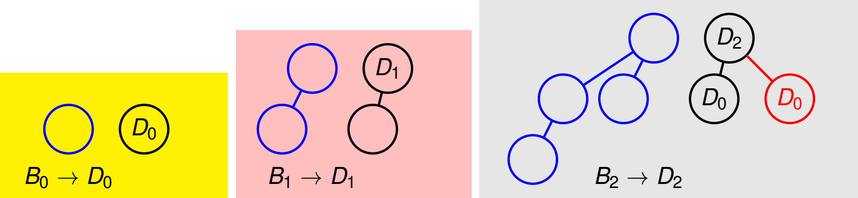


During damage, children of root of B_k retained to obtain D_k

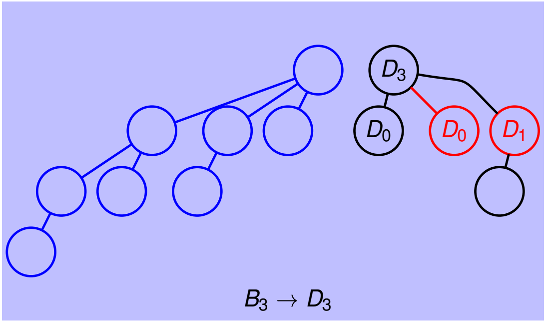


- Inferior $B_2 \equiv C_3$ can lose only one child (why?)
- By lemma, 3rd child of B_3, $B_2 \equiv C_3 \xrightarrow{\text{damage}} D_{3-2}$ or D_1
- For B_k with $C_1, C_2, \ldots, C_k$ as children D_k has D_0, $D_{2-2}, D_{3-2}, \ldots, D_{k-2}$ or $D_0, D_0, D_1, \ldots, D_{k-2}$ as children

Max damage to binomial trees in Fibonacci heap

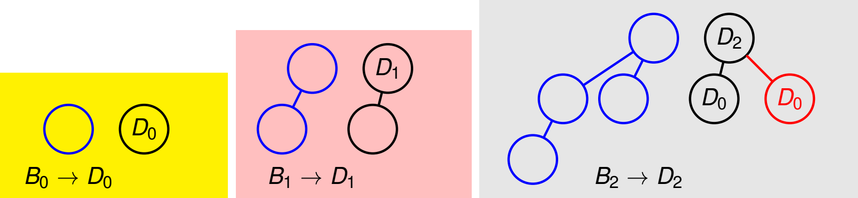


During damage, children of root of B_k retained to obtain D_k

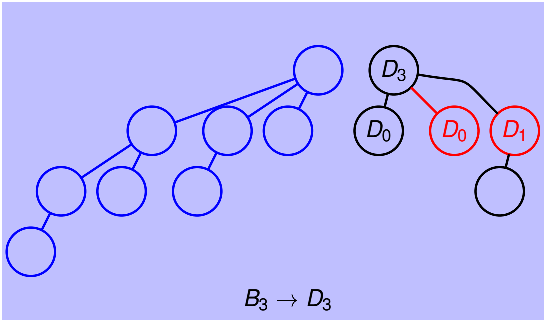


- Inferior $B_2 \equiv C_3$ can lose only one child (why?)
- By lemma, 3rd child of B_3, $B_2 \equiv C_3 \xrightarrow{\text{damage}} D_{3-2}$ or D_1
- For B_k with $C_1, C_2, \ldots, C_k$ as children D_k has D_0, $D_{2-2}, D_{3-2}, \ldots, D_{k-2}$ or $D_0, D_0, D_1, \ldots, D_{k-2}$ as children

Max damage to binomial trees in Fibonacci heap

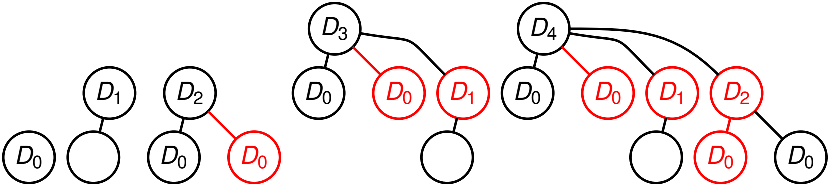


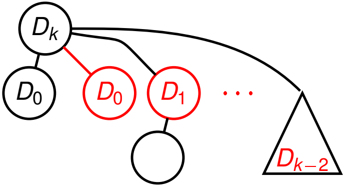
During damage, children of root of B_k retained to obtain D_k



- Inferior $B_2 \equiv C_3$ can lose only one child (why?)
- By lemma, 3rd child of B_3, $B_2 \equiv C_3 \xrightarrow{\text{damage}} D_{3-2}$ or D_1
- For B_k with $C_1, C_2, \ldots, C_k$ as children D_k has D_0, $D_{2-2}, D_{3-2}, \ldots, D_{k-2}$ or $D_0, D_0, D_1, \ldots, D_{k-2}$ as children

Maximally damaged trees in Fibonacci heaps





Maximally damaged tree in a Fibonacci heap

A maximally-damaged tree of order/rank k is rooted at a node whose children are maximally-damaged trees of orders/ranks $0, 0, 1, 2, 3, \ldots, k-2$

$|D_0| = 1, |D_1| = 2, |D_2| = 3, |D_3| = 5, |D_4| = 8, |D_k| = |D_{k-1}| + |D_{k-2}|$

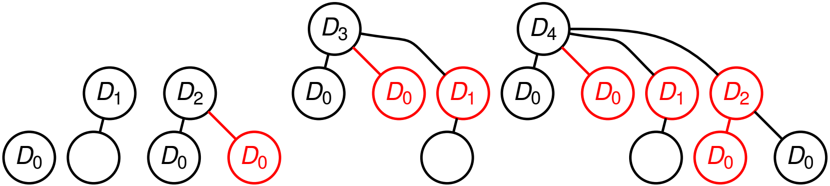
Recalling Fibonacci numbers

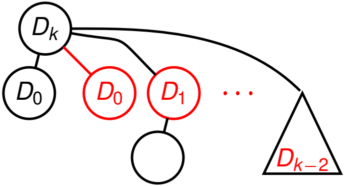
$F_0 = 0, F_1 = 1, F_2 = 1,$
$F_k = F_{k-1} + F_{k-2}, k > 1$

Relating $|D_k|$ to Fibonacci numbers

$|D_0| = F_2 = 1, |D_1| = F_3 = 2,$
$|D_k| = F_{k+2}, k > 1$

Maximally damaged trees in Fibonacci heaps





Maximally damaged tree in a Fibonacci heap

A maximally-damaged tree of order/rank k is rooted at a node whose children are maximally-damaged trees of orders/ranks $0, 0, 1, 2, 3, \ldots, k-2$

$$|D_0| = 1, |D_1| = 2, |D_2| = 3, |D_3| = 5, |D_4| = 8, |D_k| = |D_{k-1}| + |D_{k-2}|$$

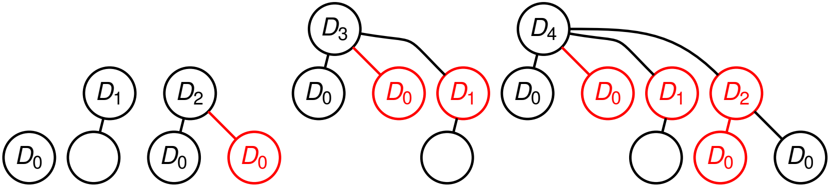
Recalling Fibonacci numbers

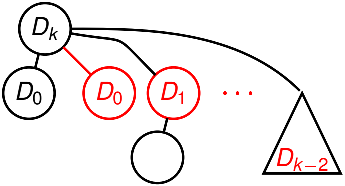
$F_0 = 0, F_1 = 1, F_2 = 1,$
$F_k = F_{k-1} + F_{k-2}, k > 1$

Relating $|D_k|$ to Fibonacci numbers

$|D_0| = F_2 = 1, |D_1| = F_3 = 2,$
$|D_k| = F_{k+2}, k > 1$

Maximally damaged trees in Fibonacci heaps





Maximally damaged tree in a Fibonacci heap

A maximally-damaged tree of order/rank k is rooted at a node whose children are maximally-damaged trees of orders/ranks $0, 0, 1, 2, 3, \ldots, k-2$

$$|D_0| = 1, |D_1| = 2, |D_2| = 3, |D_3| = 5, |D_4| = 8, |D_k| = |D_{k-1}| + |D_{k-2}|$$

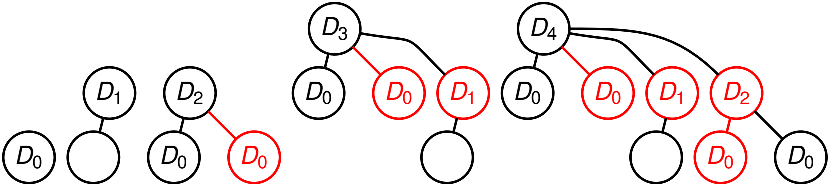
Recalling Fibonacci numbers

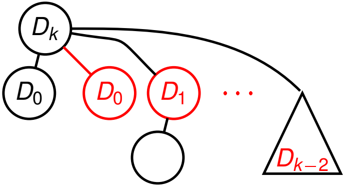
$F_0 = 0, F_1 = 1, F_2 = 1,$
$F_k = F_{k-1} + F_{k-2}, k > 1$

Relating $|D_k|$ to Fibonacci numbers

$|D_0| = F_2 = 1, |D_1| = F_3 = 2,$
$|D_k| = F_{k+2}, k > 1$

Maximally damaged trees in Fibonacci heaps





Maximally damaged tree in a Fibonacci heap

A maximally-damaged tree of order/rank k is rooted at a node whose children are maximally-damaged trees of orders/ranks $0, 0, 1, 2, 3, \ldots, k-2$

$|D_0| = 1, |D_1| = 2, |D_2| = 3, |D_3| = 5, |D_4| = 8, |D_k| = |D_{k-1}| + |D_{k-2}|$

Recalling Fibonacci numbers

$F_0 = 0, F_1 = 1, F_2 = 1,$
$F_k = F_{k-1} + F_{k-2}, k > 1$

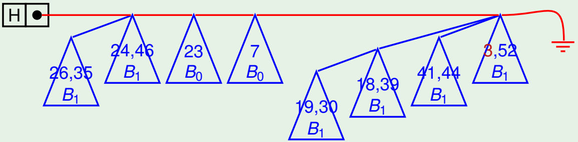
Relating $|D_k|$ to Fibonacci numbers

$|D_0| = F_2 = 1, |D_1| = F_3 = 2,$
$|D_k| = F_{k+2}, k > 1$

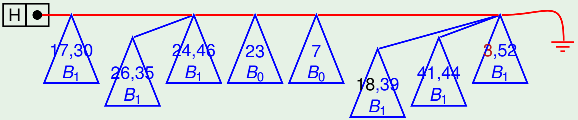
Example of FH operations

Example (FH operations)

Carry out the following operations on this Fibonacci heap:



decKey 19 → 17 Note that 18 will get marked while loosing its inferior B_1 tree to the root list



insert 21 just a lazy addition to the root list

delMin lots of changes will happen, depict the details

Costing time taken for DecKey

- Most of the acitivities bounded by constant time
  - Decreasing the key value, comparing with with parent key value
  - Possibly cutting node, transferring to root list and marking parent
- Cascading cuts: worst case is $O(\lg n)$, but amortised cost?

Amortised cost of DecKey (by the accounting method)

- Let there be a charge of 2 units for reducing the key value
- One unit is used up immediately for fixed cost operations
- The other unit is given to the marked parent as credit
- A marked parent acquires 2 credits when its (second) child is cut
- One unit is used immediately for the constant time bounded operations to cut the parent node and transfer it to the root list
- The other credit of 1 unit is passed to its parent
- Thus, a sequence of DecKeys are fully supported by the constant cost charged for each operation, so amortised cost is $\Theta(1)$ time

Costing time taken for DecKey

- Most of the acitivities bounded by constant time
  - Decreasing the key value, comparing with with parent key value
  - Possibly cutting node, transferring to root list and marking parent
- Cascading cuts: worst case is $O(\lg n)$, but amortised cost?

Amortised cost of DecKey (by the accounting method)

- Let there be a charge of 2 units for reducing the key value
- One unit is used up immediately for fixed cost operations
- The other unit is given to the marked parent as credit
- A marked parent acquires 2 credits when its (second) child is cut
- One unit is used immediately for the constant time bounded operations to cut the parent node and transfer it to the root list
- The other credit of 1 unit is passed to its parent
- Thus, a sequence of DecKeys are fully supported by the constant cost charged for each operation, so amortised cost is $\Theta(1)$ time

Costing time taken for DecKey

- Most of the acitivities bounded by constant time
  - Decreasing the key value, comparing with with parent key value
  - Possibly cutting node, transferring to root list and marking parent
- Cascading cuts: worst case is $O(\lg n)$, but amortised cost?

Amortised cost of DecKey (by the accounting method)

- Let there be a charge of 2 units for reducing the key value
- One unit is used up immediately for fixed cost operations
- The other unit is given to the marked parent as credit
- A marked parent acquires 2 credits when its (second) child is cut
- One unit is used immediately for the constant time bounded operations to cut the parent node and transfer it to the root list
- The other credit of 1 unit is passed to its parent
- Thus, a sequence of DecKeys are fully supported by the constant cost charged for each operation, so amortised cost is $\Theta(1)$ time

Costing time taken for DecKey

- Most of the acitivities bounded by constant time
  - Decreasing the key value, comparing with with parent key value
  - Possibly cutting node, transferring to root list and marking parent
- Cascading cuts: worst case is $O(\lg n)$, but amortised cost?

Amortised cost of DecKey (by the accounting method)

- Let there be a charge of 2 units for reducing the key value
- One unit is used up immediately for fixed cost operations
- The other unit is given to the marked parent as credit
- A marked parent acquires 2 credits when its (second) child is cut
- One unit is used immediately for the constant time bounded operations to cut the parent node and transfer it to the root list
- The other credit of 1 unit is passed to its parent
- Thus, a sequence of DecKeys are fully supported by the constant cost charged for each operation, so amortised cost is $\Theta(1)$ time

Costing time taken for DecKey

- Most of the acitivities bounded by constant time
  - Decreasing the key value, comparing with with parent key value
  - Possibly cutting node, transferring to root list and marking parent
- Cascading cuts: worst case is $O(\lg n)$, but amortised cost?

Amortised cost of DecKey (by the accounting method)

- Let there be a charge of 2 units for reducing the key value
- One unit is used up immediately for fixed cost operations
- The other unit is given to the marked parent as credit
- A marked parent acquires 2 credits when its (second) child is cut
- One unit is used immediately for the constant time bounded operations to cut the parent node and transfer it to the root list
- The other credit of 1 unit is passed to its parent
- Thus, a sequence of DecKeys are fully supported by the constant cost charged for each operation, so amortised cost is $\Theta(1)$ time

Costing time taken for DecKey with DelMin

- It was noted that the effort of coalescing trees resulting from Insert operations was pushed to the DelMin while being fully costed through the charge imposed for each Insert operation
- It may be noted that the charge of 2 units imposed for DecKey does not leave any spare credit for supporting coalescing of the trees transferred to the root list
- Fortunately, the problem is easily rectified, by increasing the charge to 3 units – how does this help?

  The extra charge of one unit is saved as credit with the tree (root) transferred to the root list
- At the time of coalescing the trees during DelMin this credit held by each tree (root) introduced through the DecKey operation completely covers the costs incurred
- Amortised cost of DecKey continues to be $\Theta(1)$

Costing time taken for DecKey with DelMin

- It was noted that the effort of coalescing trees resulting from Insert operations was pushed to the DelMin while being fully costed through the charge imposed for each Insert operation
- It may be noted that the charge of 2 units imposed for DecKey does not leave any spare credit for supporting coalescing of the trees transferred to the root list
- Fortunately, the problem is easily rectified, by increasing the charge to 3 units – how does this help?

  The extra charge of one unit is saved as credit with the tree (root) transferred to the root list

- At the time of coalescing the trees during DelMin this credit held by each tree (root) introduced through the DecKey operation completely covers the costs incurred
- Amortised cost of DecKey continues to be $\Theta(1)$

Costing time taken for DecKey with DelMin

- It was noted that the effort of coalescing trees resulting from Insert operations was pushed to the DelMin while being fully costed through the charge imposed for each Insert operation
- It may be noted that the charge of 2 units imposed for DecKey does not leave any spare credit for supporting coalescing of the trees transferred to the root list
- Fortunately, the problem is easily rectified, by increasing the charge to 3 units – how does this help?

  The extra charge of one unit is saved as credit with the tree (root) transferred to the root list

- At the time of coalescing the trees during DelMin this credit held by each tree (root) introduced through the DecKey operation completely covers the costs incurred
- Amortised cost of DecKey continues to be $\Theta(1)$

Charges and invariants for Fibonacci heaps

Charge for insertion 2 units are charged

Usage 1 unit for constant time operations for insertion,

Further usage 1 unit retained as credit with the resulting B_0

Invariant Each tree in the heap always has 1 saved credit

Usage of credit Coalescing of trees after DelMin at no extra cost

---

Charge for DecKey 3 units are charged

Usage (in case heap order is violated) 1 unit is used for constant time operations for detaching node (with subtree) from tree and adding to the root list and marking parent

Further usage 1 unit is transferred as credit to tree added to root list

Further usage 1 unit is retainied a credit with (last) marked parent

Invariant Each marked node has 1 saved credit

Usage of credit For cascaded detachment of marked nodes (along with the subtree) at no extra cost

Charges and invariants for Fibonacci heaps

Charge for insertion 2 units are charged

Usage 1 unit for constant time operations for insertion,

Further usage 1 unit retained as credit with the resulting B_0

Invariant Each tree in the heap always has 1 saved credit

Usage of credit Coalescing of trees after DelMin at no extra cost

---

Charge for DecKey 3 units are charged

Usage (in case heap order is violated) 1 unit is used for constant time operations for detaching node (with subtree) from tree and adding to the root list and marking parent

Further usage 1 unit is transferred as credit to tree added to root list

Further usage 1 unit is retainied a credit with (last) marked parent

Invariant Each marked node has 1 saved credit

Usage of credit For cascaded detachment of marked nodes (along with the subtree) at no extra cost

Charges and invariants for Fibonacci heaps

Charge for insertion 2 units are charged

Usage 1 unit for constant time operations for insertion,

Further usage 1 unit retained as credit with the resulting B_0

Invariant Each tree in the heap always has 1 saved credit

Usage of credit Coalescing of trees after DelMin at no extra cost

---

Charge for DecKey 3 units are charged

Usage (in case heap order is violated) 1 unit is used for constant time operations for detaching node (with subtree) from tree and adding to the root list and marking parent

Further usage 1 unit is transferred as credit to tree added to root list

Further usage 1 unit is retainied a credit with (last) marked parent

Invariant Each marked node has 1 saved credit

Usage of credit For cascaded detachment of marked nodes (along with the subtree) at no extra cost

Charges and invariants for Fibonacci heaps

Charge for insertion 2 units are charged

Usage 1 unit for constant time operations for insertion,

Further usage 1 unit retained as credit with the resulting B_0

Invariant Each tree in the heap always has 1 saved credit

Usage of credit Coalescing of trees after DelMin at no extra cost

---

Charge for DecKey 3 units are charged

Usage (in case heap order is violated) 1 unit is used for constant time operations for detaching node (with subtree) from tree and adding to the root list and marking parent

Further usage 1 unit is transferred as credit to tree added to root list

Further usage 1 unit is retainied a credit with (last) marked parent

Invariant Each marked node has 1 saved credit

Usage of credit For cascaded detachment of marked nodes (along with the subtree) at no extra cost

Charges and invariants for Fibonacci heaps

Charge for insertion 2 units are charged

Usage 1 unit for constant time operations for insertion,

Further usage 1 unit retained as credit with the resulting B_0

Invariant Each tree in the heap always has 1 saved credit

Usage of credit Coalescing of trees after DelMin at no extra cost

---

Charge for DecKey 3 units are charged

Usage (in case heap order is violated) 1 unit is used for constant time operations for detaching node (with subtree) from tree and adding to the root list and marking parent

Further usage 1 unit is transferred as credit to tree added to root list

Further usage 1 unit is retainied a credit with (last) marked parent

Invariant Each marked node has 1 saved credit

Usage of credit For cascaded detachment of marked nodes (along with the subtree) at no extra cost

Charges and invariants for Fibonacci heaps

Charge for insertion 2 units are charged

Usage 1 unit for constant time operations for insertion,

Further usage 1 unit retained as credit with the resulting B_0

Invariant Each tree in the heap always has 1 saved credit

Usage of credit Coalescing of trees after DelMin at no extra cost

---

Charge for DecKey 3 units are charged

Usage (in case heap order is violated) 1 unit is used for constant time operations for detaching node (with subtree) from tree and adding to the root list and marking parent

Further usage 1 unit is transferred as credit to tree added to root list

Further usage 1 unit is retainied a credit with (last) marked parent

Invariant Each marked node has 1 saved credit

Usage of credit For cascaded detachment of marked nodes (along with the subtree) at no extra cost

Charges and invariants for Fibonacci heaps

Charge for insertion 2 units are charged

Usage 1 unit for constant time operations for insertion,

Further usage 1 unit retained as credit with the resulting B_0

Invariant Each tree in the heap always has 1 saved credit

Usage of credit Coalescing of trees after DelMin at no extra cost

---

Charge for DecKey 3 units are charged

Usage (in case heap order is violated) 1 unit is used for constant time operations for detaching node (with subtree) from tree and adding to the root list and marking parent

Further usage 1 unit is transferred as credit to tree added to root list

Further usage 1 unit is retainied a credit with (last) marked parent

Invariant Each marked node has 1 saved credit

Usage of credit For cascaded detachment of marked nodes (along with the subtree) at no extra cost

Charges and invariants for Fibonacci heaps

Charge for insertion 2 units are charged

Usage 1 unit for constant time operations for insertion,

Further usage 1 unit retained as credit with the resulting B_0

Invariant Each tree in the heap always has 1 saved credit

Usage of credit Coalescing of trees after DelMin at no extra cost

---

Charge for DecKey 3 units are charged

Usage (in case heap order is violated) 1 unit is used for constant time operations for detaching node (with subtree) from tree and adding to the root list and marking parent

Further usage 1 unit is transferred as credit to tree added to root list

Further usage 1 unit is retainied a credit with (last) marked parent

Invariant Each marked node has 1 saved credit

Usage of credit For cascaded detachment of marked nodes (along with the subtree) at no extra cost

Charges and invariants for Fibonacci heaps

Charge for insertion 2 units are charged

Usage 1 unit for constant time operations for insertion,

Further usage 1 unit retained as credit with the resulting B_0

Invariant Each tree in the heap always has 1 saved credit

Usage of credit Coalescing of trees after DelMin at no extra cost

---

Charge for DecKey 3 units are charged

Usage (in case heap order is violated) 1 unit is used for constant time operations for detaching node (with subtree) from tree and adding to the root list and marking parent

Further usage 1 unit is transferred as credit to tree added to root list

Further usage 1 unit is retainied a credit with (last) marked parent

Invariant Each marked node has 1 saved credit

Usage of credit For cascaded detachment of marked nodes (along with the subtree) at no extra cost

Charges and invariants for Fibonacci heaps

Charge for insertion 2 units are charged

Usage 1 unit for constant time operations for insertion,

Further usage 1 unit retained as credit with the resulting B_0

Invariant Each tree in the heap always has 1 saved credit

Usage of credit Coalescing of trees after DelMin at no extra cost

---

Charge for DecKey 3 units are charged

Usage (in case heap order is violated) 1 unit is used for constant time operations for detaching node (with subtree) from tree and adding to the root list and marking parent

Further usage 1 unit is transferred as credit to tree added to root list

Further usage 1 unit is retainied a credit with (last) marked parent

Invariant Each marked node has 1 saved credit

Usage of credit For cascaded detachment of marked nodes (along with the subtree) at no extra cost

Charges and invariants for Fibonacci heaps

Charge for insertion 2 units are charged

Usage 1 unit for constant time operations for insertion,

Further usage 1 unit retained as credit with the resulting B_0

Invariant Each tree in the heap always has 1 saved credit

Usage of credit Coalescing of trees after DelMin at no extra cost

---

Charge for DecKey 3 units are charged

Usage (in case heap order is violated) 1 unit is used for constant time operations for detaching node (with subtree) from tree and adding to the root list and marking parent

Further usage 1 unit is transferred as credit to tree added to root list

Further usage 1 unit is retainied a credit with (last) marked parent

Invariant Each marked node has 1 saved credit

Usage of credit For cascaded detachment of marked nodes (along with the subtree) at no extra cost

Comparison of running times of heap operations

Operation	Binary	Binomial	Lazy Binomial[a]	Fibonacci[a]
Create	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Merge	$\Theta(n)$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
FindMin	$\Theta(1)$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
DelMin	$O(\lg(n))$	$O(\lg(n))$	$O(\lg n)$	$O(\lg n)$
Insert	$O(\lg(n))$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
DecKey	$O(\lg(n))$	$O(\lg(n))$	$O(\lg n)$	$\Theta(1)$

[a] Amortised cost

Comparison of tree sizes

Binary $[2^h, 2^{h+1} - 1]$ for tree height of h

Binomial 2^k for tree of rank k

Lazy bino same a binomial

Fibonacci $[2^k, F_{k+2}]$ for tree of rank k

$F_{k+2} = \frac{\varphi^{k+2} - (-\varphi)^{k+2}}{\sqrt{5}}$, $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.61803$

Tree heights

- $h(D_0) = 0, h(D_1) = 1$
- $h(D_k) = 1 + h(D_{k-2}), k > 1$
- $h(D_k) = \left\lceil \frac{k}{2} \right\rceil, k \geq 0$
- $h(B_k) = k, k \geq 0$

Comparison of running times of heap operations

Operation	Binary	Binomial	Lazy Binomial[a]	Fibonacci[a]
Create	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Merge	$\Theta(n)$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
FindMin	$\Theta(1)$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
DelMin	$O(\lg(n))$	$O(\lg(n))$	$O(\lg n)$	$O(\lg n)$
Insert	$O(\lg(n))$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
DecKey	$O(\lg(n))$	$O(\lg(n))$	$O(\lg n)$	$\Theta(1)$

[a] Amortised cost

Comparison of tree sizes

Binary $[2^h, 2^{h+1}-1]$ for tree height of h

Binomial 2^k for tree of rank k

Lazy bino same a binomial

Fibonacci $[2^k, F_{k+2}]$ for tree of rank k

$F_{k+2} = \frac{\varphi^{k+2}-(-\varphi)^{k+2}}{\sqrt{5}}, \varphi = \frac{1+\sqrt{5}}{2} \approx 1.61803$

Tree heights

- $h(D_0) = 0, h(D_1) = 1$
- $h(D_k) = 1 + h(D_{k-2}), k > 1$
- $h(D_k) = \left\lceil \frac{k}{2} \right\rceil, k \geq 0$
- $h(B_k) = k, k \geq 0$

Comparison of running times of heap operations

Operation	Binary	Binomial	Lazy Binomial[a]	Fibonacci[a]
Create	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Merge	$\Theta(n)$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
FindMin	$\Theta(1)$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
DelMin	$O(\lg(n))$	$O(\lg(n))$	$O(\lg n)$	$O(\lg n)$
Insert	$O(\lg(n))$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
DecKey	$O(\lg(n))$	$O(\lg(n))$	$O(\lg n)$	$\Theta(1)$

[a] Amortised cost

Comparison of tree sizes

Binary $[2^h, 2^{h+1} - 1]$ for tree height of h

Binomial 2^k for tree of rank k

Lazy bino same a binomial

Fibonacci $[2^k, F_{k+2}]$ for tree of rank k

$F_{k+2} = \frac{\varphi^{k+2} - (-\varphi)^{k+2}}{\sqrt{5}}$, $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.61803$

Tree heights

- $h(D_0) = 0, h(D_1) = 1$
- $h(D_k) = 1 + h(D_{k-2}), k > 1$
- $h(D_k) = \left\lceil \frac{k}{2} \right\rceil, k \geq 0$
- $h(B_k) = k, k \geq 0$

Comparison of running times of heap operations

Operation	Binary	Binomial	Lazy Binomial[a]	Fibonacci[a]
Create	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Merge	$\Theta(n)$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
FindMin	$\Theta(1)$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
DelMin	$O(\lg(n))$	$O(\lg(n))$	$O(\lg n)$	$O(\lg n)$
Insert	$O(\lg(n))$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
DecKey	$O(\lg(n))$	$O(\lg(n))$	$O(\lg n)$	$\Theta(1)$

[a] Amortised cost

Comparison of tree sizes

Binary $[2^h, 2^{h+1} - 1]$ for tree height of h

Binomial 2^k for tree of rank k

Lazy bino same a binomial

Fibonacci $[2^k, F_{k+2}]$ for tree of rank k

$F_{k+2} = \frac{\varphi^{k+2} - (-\varphi)^{k+2}}{\sqrt{5}}, \varphi = \frac{1+\sqrt{5}}{2} \approx 1.61803$

Tree heights

- $h(D_0) = 0, h(D_1) = 1$
- $h(D_k) = 1 + h(D_{k-2}), k > 1$
- $h(D_k) = \left\lceil \frac{k}{2} \right\rceil, k \geq 0$
- $h(B_k) = k, k \geq 0$

Comparison of running times of heap operations

Operation	Binary	Binomial	Lazy Binomial[a]	Fibonacci[a]
Create	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Merge	$\Theta(n)$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
FindMin	$\Theta(1)$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
DelMin	$O(\lg(n))$	$O(\lg(n))$	$O(\lg n)$	$O(\lg n)$
Insert	$O(\lg(n))$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
DecKey	$O(\lg(n))$	$O(\lg(n))$	$O(\lg n)$	$\Theta(1)$

[a] Amortised cost

Comparison of tree sizes

Binary $[2^h, 2^{h+1} - 1]$ for tree height of h

Binomial 2^k for tree of rank k

Lazy bino same a binomial

Fibonacci $[2^k, F_{k+2}]$ for tree of rank k

$$F_{k+2} = \frac{\varphi^{k+2} - (-\varphi)^{k+2}}{\sqrt{5}}, \varphi = \frac{1+\sqrt{5}}{2} \approx 1.61803$$

Tree heights

- $h(D_0) = 0, h(D_1) = 1$
- $h(D_k) = 1 + h(D_{k-2}), k > 1$
- $h(D_k) = \left\lceil \frac{k}{2} \right\rceil, k \geq 0$
- $h(B_k) = k, k \geq 0$

Comparison of running times of heap operations

Operation	Binary	Binomial	Lazy Binomial[a]	Fibonacci[a]
Create	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Merge	$\Theta(n)$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
FindMin	$\Theta(1)$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
DelMin	$O(\lg(n))$	$O(\lg(n))$	$O(\lg n)$	$O(\lg n)$
Insert	$O(\lg(n))$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
DecKey	$O(\lg(n))$	$O(\lg(n))$	$O(\lg n)$	$\Theta(1)$

[a] Amortised cost

Comparison of tree sizes

Binary $[2^h, 2^{h+1}-1]$ for tree height of h

Binomial 2^k for tree of rank k

Lazy bino same a binomial

Fibonacci $[2^k, F_{k+2}]$ for tree of rank k

$$F_{k+2} = \frac{\varphi^{k+2} - (-\varphi)^{k+2}}{\sqrt{5}}, \varphi = \frac{1+\sqrt{5}}{2} \approx 1.61803$$

Tree heights

- $h(D_0) = 0, h(D_1) = 1$
- $h(D_k) = 1 + h(D_{k-2}), k > 1$
- $h(D_k) = \left\lceil \frac{k}{2} \right\rceil, k \geq 0$
- $h(B_k) = k, k \geq 0$

Comparison of running times of heap operations

Operation	Binary	Binomial	Lazy Binomial[a]	Fibonacci[a]
Create	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Merge	$\Theta(n)$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
FindMin	$\Theta(1)$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
DelMin	$O(\lg(n))$	$O(\lg(n))$	$O(\lg n)$	$O(\lg n)$
Insert	$O(\lg(n))$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
DecKey	$O(\lg(n))$	$O(\lg(n))$	$O(\lg n)$	$\Theta(1)$

[a] Amortised cost

Comparison of tree sizes

Binary $[2^h, 2^{h+1} - 1]$ for tree height of h

Binomial 2^k for tree of rank k

Lazy bino same a binomial

Fibonacci $[2^k, F_{k+2}]$ for tree of rank k

$$F_{k+2} = \frac{\varphi^{k+2} - (-\varphi)^{k+2}}{\sqrt{5}}, \varphi = \frac{1+\sqrt{5}}{2} \approx 1.61803$$

Tree heights

- $h(D_0) = 0, h(D_1) = 1$
- $h(D_k) =$
  $1 + h(D_{k-2}), k > 1$
- $h(D_k) = \left\lceil \frac{k}{2} \right\rceil, k \geq 0$
- $h(B_k) = k, k \geq 0$

Comparison of running times of heap operations

Operation	Binary	Binomial	Lazy Binomial[a]	Fibonacci[a]
Create	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Merge	$\Theta(n)$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
FindMin	$\Theta(1)$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
DelMin	$O(\lg(n))$	$O(\lg(n))$	$O(\lg n)$	$O(\lg n)$
Insert	$O(\lg(n))$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
DecKey	$O(\lg(n))$	$O(\lg(n))$	$O(\lg n)$	$\Theta(1)$

[a] Amortised cost

Comparison of tree sizes

Binary $[2^h, 2^{h+1} - 1]$ for tree height of h

Binomial 2^k for tree of rank k

Lazy bino same a binomial

Fibonacci $[2^k, F_{k+2}]$ for tree of rank k

$$F_{k+2} = \frac{\varphi^{k+2} - (-\varphi)^{k+2}}{\sqrt{5}}, \varphi = \frac{1+\sqrt{5}}{2} \approx 1.61803$$

Tree heights

- $h(D_0) = 0, h(D_1) = 1$
- $h(D_k) = 1 + h(D_{k-2}), k > 1$
- $h(D_k) = \left\lceil \frac{k}{2} \right\rceil, k \geq 0$
- $h(B_k) = k, k \geq 0$

Comparison of running times of heap operations

Operation	Binary	Binomial	Lazy Binomial[a]	Fibonacci[a]
Create	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Merge	$\Theta(n)$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
FindMin	$\Theta(1)$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
DelMin	$O(\lg(n))$	$O(\lg(n))$	$O(\lg n)$	$O(\lg n)$
Insert	$O(\lg(n))$	$O(\lg(n))$	$\Theta(1)$	$\Theta(1)$
DecKey	$O(\lg(n))$	$O(\lg(n))$	$O(\lg n)$	$\Theta(1)$

[a] Amortised cost

Comparison of tree sizes

Binary $[2^h, 2^{h+1} - 1]$ for tree height of h

Binomial 2^k for tree of rank k

Lazy bino same a binomial

Fibonacci $[2^k, F_{k+2}]$ for tree of rank k

$$F_{k+2} = \frac{\varphi^{k+2} - (-\varphi)^{k+2}}{\sqrt{5}}, \varphi = \frac{1+\sqrt{5}}{2} \approx 1.61803$$

Tree heights

- $h(D_0) = 0, h(D_1) = 1$
- $h(D_k) = 1 + h(D_{k-2}), k > 1$
- $h(D_k) = \left\lceil \frac{k}{2} \right\rceil, k \geq 0$
- $h(B_k) = k, k \geq 0$

Representation of a Fibonacci heap

1. The following items of information per node are needed:
   - a field **key** for its key,
   - a field **degree** for the number of children,
   - a pointer **child** , which points to the leftmost-child,
   - a pointer **sibLeft**, which points to the leftt-sibling,
   - a pointer **sibRight**, which points to the right-sibling, and
   - a pointer **parent**, which points to the parent
2. The roots of the trees are connected in a circular doubly connected linked list
3. Ranks of the connected trees can be in any order, need not be unique unless coalasced
4. For a heap **H**, **H.head** points to the head of the list; **H.min** points to the min root in the list