Heaps

Combinatorial Optimization

Giovanni Righini

Heaps

In many combinatorial optimization problems it is quite common that a set of values must be kept in a data-structure, so that some operations can be repeatedly done on it. The efficiency of such operations strongly affects the overall efficiency of the optimization algorithm.

Very often, we are not interested in all values, but only in the minimum (or maximum) value at any moment.

Typical operations to be repeatedly done are:

- insert a new element (*Insert*);
- extract the element with minimum value (*ExtractMin*);
- decrease the value of an element (*DecreaseKey*).

Heaps

Let indicate the maximum number of elements in the data-structure with n.

- **List or array**: *Insert* in $O(1)$, *ExtractMin* in $O(n)$.
- **Sorted list**: *Insert* in $O(n)$, *ExtractMin* in $O(1)$.
- **Sorted array**: *Insert* in $O(n)^{(*)}$, *ExtractMin* in $O(1)^{(**)}$.
- **Balanced binary tree**: *Insert* in $O(\log n)$, *ExtractMin* in $O(\log n)$.

$(*)$ Finding the correct insertion position by dichotomic search takes $O(\log n)$ time; shifting the last elements one position to the right to create the empty space in which the new element is to be inserted takes $O(n)$ time.
$(**)$ The elements must be sorted in non-increasing order, so that the minimum is the last one.

Binary heaps

A binary heap is a binary tree in which a *key* is associated with each node and the following two properties hold:

- all levels of the tree are filled from the left to the right and only the last level is allowed to be incomplete;
- the key associated with a parent node is not larger than the keys associated with its two children nodes.

In a binary heap the operations *Insert*, *ExtractMin* and *DecreaseKey* can be implemented to run in $O(\log n)$, being n the number of nodes in the tree.

Binary heap: basic operations

A binary heap can be represented with an array of size n, when we know a priori that no more than n elements are needed (e.g. one for each node of a (di-)graph).

The positions in the heap are numbered in breadth-first order.

Given any element in position p,

- $left(p) = 2p$
- $right(p) = 2p + 1$
- $parent(p) = p \ div \ 2$

where *div* indicates integer division.

Binary heap: basic operations

Every element of the heap is a record with the following fields

- *element*: the corresponding element (e.g. a node of a (di-)graph);
- *cost*: the corresponding dual variable.

We also need:

- an integer L indicating how many elements are in the heap;
- an array *pos* indicating the position of each element in the heap.

$Insert(i, v)$

An element i and its key value v are in input:

1. a new element is placed in the last position of the heap: $O(1)$
2. it is pushed upwards by swapping it with its parent, until the property of the heap is restored: $O(\log n)$

$L \leftarrow L + 1$
$heap[L].node \leftarrow i$
$heap[L].cost \leftarrow v$
$pos[i] \leftarrow L$
$MoveUp(L)$

ExtractMin(i, v)

The root node i and its key value v are in output:

1. the root of the heap is extracted and replaced by the last element: $O(1)$;
2. the new root is moved downwards by swapping it with its minimum cost child, until the property of the heap is restored: $O(\log n)$.

$i \leftarrow heap[1].node$
$v \leftarrow heap[1].cost$
$pos[i] \leftarrow nil$
$heap[1] \leftarrow heap[L]$
$L \leftarrow L - 1$
$MoveDn(1)$

DecreaseKey(i, v)

An element i and a value v are in input:

1. the key value of the node corresponding to element i is updated: $O(1)$;
2. it is pushed upwards by swapping it with its parent, until the property of the heap is restored: $O(\log n)$.

$p \leftarrow pos[i]$
$heap[p].cost \leftarrow v$
$MoveUp(p)$

MoveUp(p)

$stop \leftarrow false$
while $(p \neq 1) \wedge (\neg stop)$ **do**
  $q \leftarrow parent(p)$
  if $heap[q].cost > heap[p].cost$ **then**
    $Swap(p, q)$
  else
    $stop \leftarrow true$
  end if
end while

MoveDn(p)

$stop \leftarrow false$
while $(left(p) \leq L) \wedge (\neg stop)$ **do**
  if $(right(p) > L) \vee (heap[left(p)].cost < heap[right(p)].cost)$ **then**
    $q \leftarrow left(p)$
  else
    $q \leftarrow right(p)$
  end if
  if $heap[p].cost > heap[q].cost$ **then**
    $Swap(p, q)$
  else
    $stop \leftarrow true$
  end if
end while

$Swap(p, q)$

Both *MoveUp* and *MoveDn* use *Swap* as a sub-routine. Its complexity is $O(1)$.
It uses a temporary record r and a temporary integer k.

$r \leftarrow heap[p]$
$heap[p] \leftarrow heap[q]$
$heap[q] \leftarrow r$
$k \leftarrow pos[p]$
$pos[p] \leftarrow pos[q]$
$pos[q] \leftarrow k$
$k \leftarrow p$
$p \leftarrow q$
$q \leftarrow k$

Building a heap

When n values to be partially sorted in a heap are known since the beginning, it is not needed to build the heap with n successive *Insert* operations, which would take $O(n \log n)$ time.

// An array v of elements is in input //
for $i = 1, \ldots, n$ **do**
  $heap[i].node \leftarrow i$
  $heap[i].value \leftarrow v[i]$
  $pos[i] \leftarrow i$
end for
for $k = \lfloor n/2 \rfloor, \ldots, 1$ **do**
  $MoveDn(k)$
end for

Complexity

The height (number of levels) of a heap with n elements is $h = \lceil \log_2 n + 1 \rceil$.

Each leaf is a heap of height 1 and it trivially satisfies both properties of binary heaps (it is a well-formed heap).

Iteratively, from the leaves to the root, well-formed heaps of height h are matched in pairs to form well-formed heaps of height $h + 1$. For each fusion of two heaps into one, if the root does not satisfy the heap property, it is required to move it down to its correct position, with *MoveDn*.

In the worst case, each execution of *MoveDn* implies a number of swaps equal to the height of the two merged heaps.

Complexity

Assume the root is at level 1 and the leaves at level h.

For a node at level k, the maximum number of swap operations when it is checked as a root is $h - k$.

At each level k there are at most 2^{k-1} nodes.

Hence, in the worst case the total number of swap operations is

$$S = \sum_{k=1}^{h-1} (h-k)2^{k-1}.$$

Complexity

Hence

$$S = \sum_{k=1}^{h-1} (h-k)2^{k-1} = h \sum_{k=1}^{h-1} 2^{k-1} - \sum_{k=1}^{h-1} k2^{k-1}.$$

The second term is further split as follows:

$$\sum_{k=1}^{h-1} k2^{k-1} = \sum_{k=0}^{h-2} (k+1)2^k = \sum_{k=0}^{h-2} k2^k + \sum_{k=0}^{h-2} 2^k.$$

Therefore $S = A - B - C$, with

- $A = h \sum_{k=1}^{h-1} 2^{k-1}$;
- $B = \sum_{k=0}^{h-2} k2^k$;
- $C = \sum_{k=0}^{h-2} 2^k$.

Complexity

$$A = h \sum_{k=1}^{h-1} 2^{k-1} = h \sum_{k=0}^{h-2} 2^k = h(2^{h-1} - 1).$$

$$C = \sum_{k=0}^{h-2} 2^k = 2^{h-1} - 1.$$

Complexity

$$B = \sum_{k=0}^{h-2} k2^k = 1\,2^1 + 2\,2^2 + 3\,2^3 + \ldots + (h-2)2^{h-2}.$$

$$2B = \sum_{k=0}^{h-2} k2^{k+1} = 1\,2^2 + 2\,2^3 + 3\,2^4 + \ldots + (h-2)2^{h-1}.$$

$$B = 2B - B = -2^1 - 2^2 - 2^3 - \ldots - 2^{h-2} + (h-2)2^{h-1} =$$

$$= (h-2)2^{h-1} - \sum_{k=1}^{h-2} 2^k = (h-2)2^{h-1} - (\sum_{k=0}^{h-2} 2^k - 2^0) =$$

$$= (h-2)2^{h-1} - (2^{h-1} - 1 - 1) = (h-3)2^{h-1} + 2.$$

Complexity

Finally

$$S = A - B - C = h(2^{h-1} - 1) - [(h-3)2^{h-1} + 2] - [2^{h-1} - 1] =$$
$$= 2\,2^{h-1} - h - 2 + 1 = 2^h - h - 1.$$

Since h grows as $O(\log n)$, then S grows as $O(n)$.

The complexity of *BuildHeap* when n elements are available is $O(n)$.

Find k^{th} smallest element

By-product of the linear complexity of *BuildHeap*:
find the k^{th} smallest element in a set of n elements.

Algorithm 1 (with an array):

- Sort the elements: $O(n \log n)$;
- Access the element in position k: $O(1)$.

Algorithm 2 (with a binary heap):

- *BuildHeap*: $O(n)$;
- For k times *ExtractMin*: $O(k \log n)$.

If k is "small" (i.e. $O(\frac{n}{\log n})$), algorithm 2 is likely to be faster.

Insert vs DecreaseKey

When the key value of an element of the heap decreases:

- *DecreaseKey* of the existing element in the heap;
- *Insert* of a new copy of the element with the new key value, leaving the old copy unchanged.

Both procedures take $O(\log n)$ in the worst case, but....

- *DecreaseKey* keeps the size of the heap constant, but it requires *pos* to access the element;
- *Insert* makes the heap growing, but it does not require *pos* (and the time to update it in *Swap*).

There is a trade-off between the additional time spent because the heap grows and the additional time spent to update *pos*.

d-heaps

A d-heap is a tree in which each element may have up to d children, with $d > 2$.

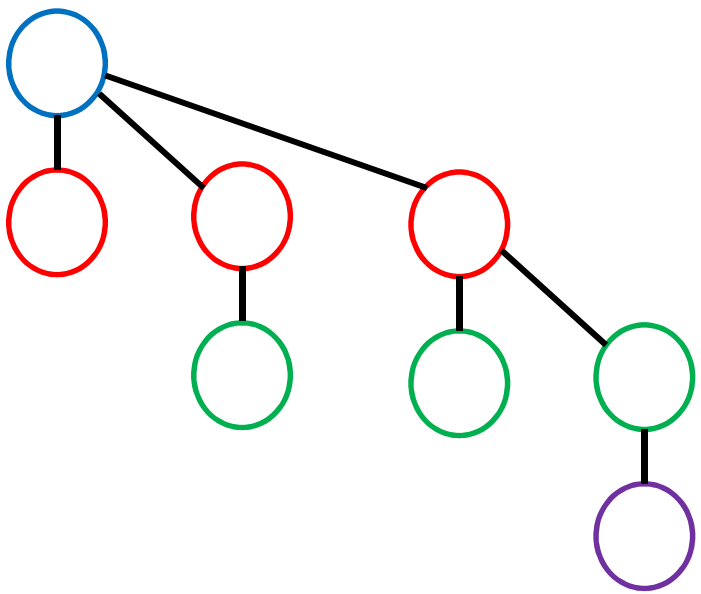
Number of levels: $\lceil \log_d n + 1 \rceil$ instead of $\lceil \log_2 n + 1 \rceil$.

Pro: fewer *Swap* operations are required.
Con: in *MoveDn* the selection of the minimum cost child costs $O(d)$ instead of $O(1)$.

Binomial heaps

A binomial heap is a forest of heap-ordered trees of different size.

Property 1.
A tree of height k has exactly 2^k elements (height = n. edges between the root and the last leaf).
The number of elements at each level $l = 1, \ldots, k$ is $\binom{k}{l}$ (binomial coefficient).



Level	Nodes
0	$1 = \binom{3}{0}$
1	$3 = \binom{3}{1}$
2	$3 = \binom{3}{2}$
3	$1 = \binom{3}{3}$

Binomial heaps

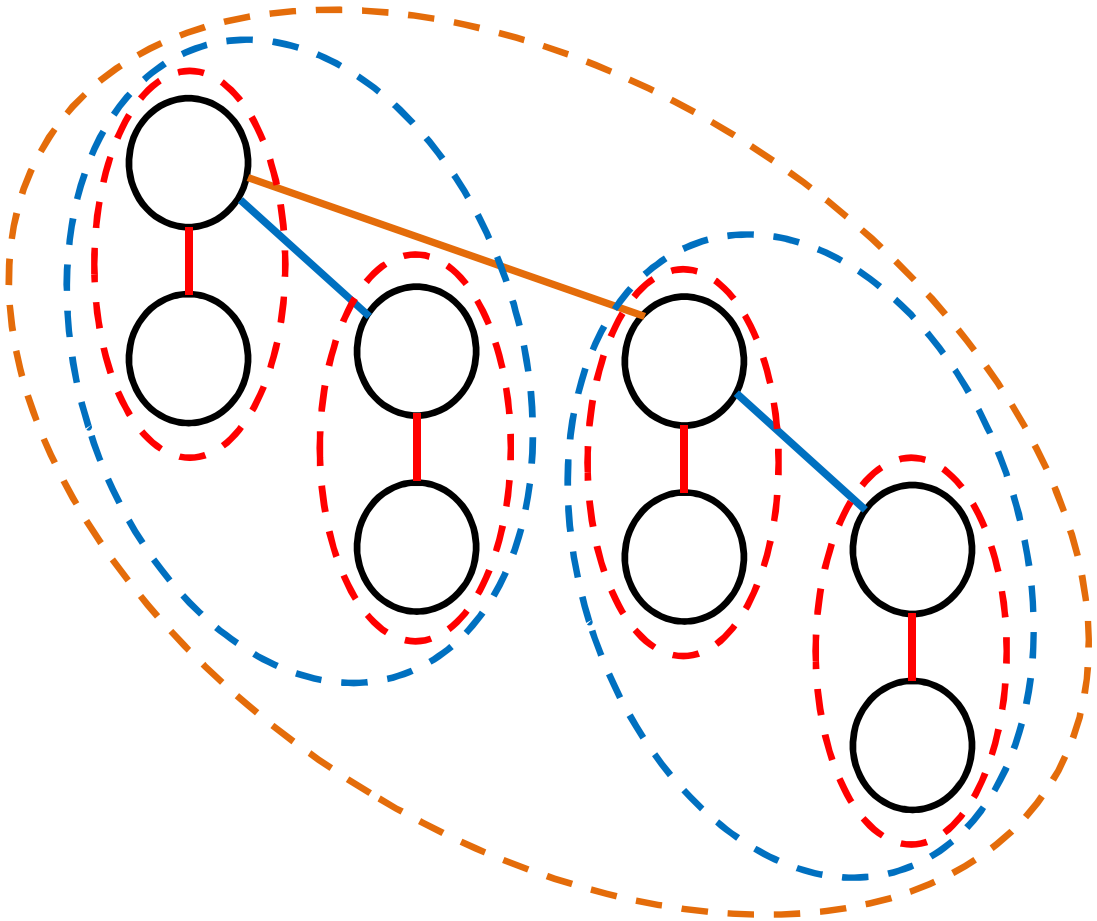
Property 2.
Elements in all trees are partially sorted as in a binary heap: the key of the predecessor is always less than or equal to the keys of its successors.

Property 3.
For each number n of nodes in the heap, there exists a unique corresponding set of heights of the trees:
one-to-one correspondence with the binary encoding of n.

Therefore the number of trees in the heap is $\lceil \log_2 n + 1 \rceil$.

Binomial heaps

Property 4.
Every tree T of height k is composed of two trees T_1 and T_2 of height $k-1$: the root of T_2 is the last successor of the root of T_1.



Data structure

Elements in a binomial heap do not have a predefined number of successors.
A possible implementation uses three pointers for each record:

- a pointer to the predecessor;
- a pointer to the first successor;
- a pointer to the next successor of the predecessor (“sibling”).

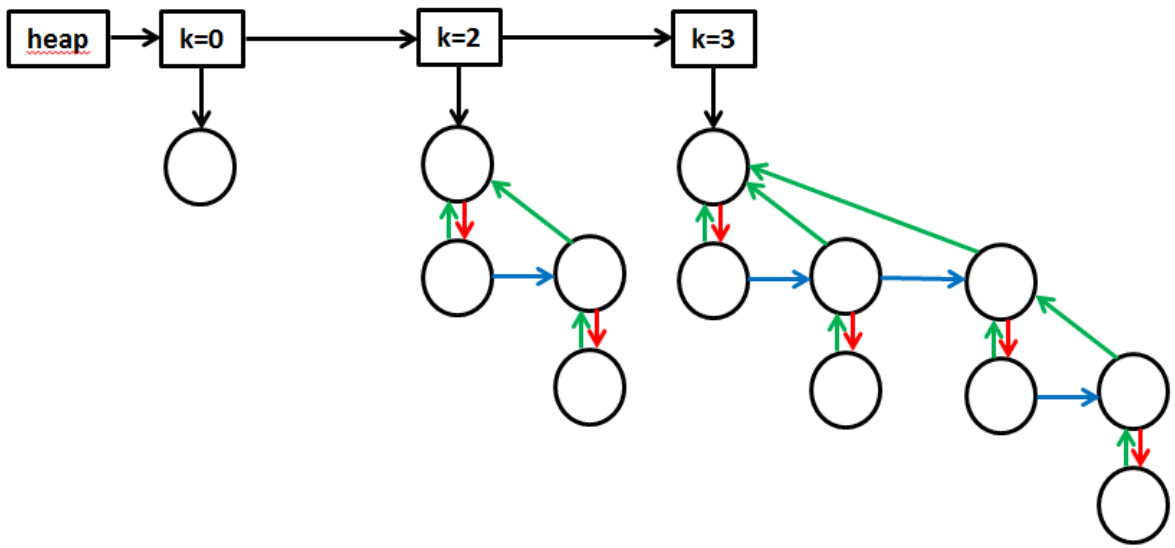


Figure: A sample binomial heap with 13 elements (binary encoding = 1101).

ExtractMin

There are $O(\log n)$ trees in the heap: comparing the keys of their roots and selecting the best one takes $O(\log n)$.

The deletion of the root of a tree T of height k produces:

- a tree T_1 of height $k - 2$;
- a tree T_2 of height $k - 1$.

Then, trees of the same height are merged as binary numbers are added up.

ExtractMin

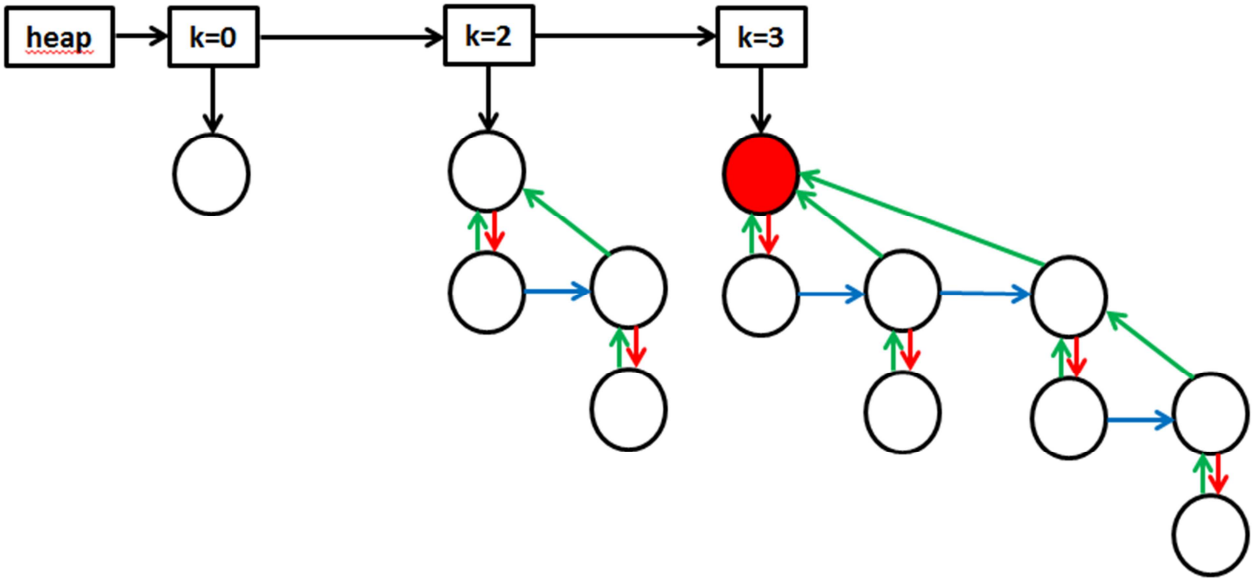


Figure: In this example we assume that the red element has the minimum key and it is extracted. Its tree has height $k^* = 3$.

ExtractMin

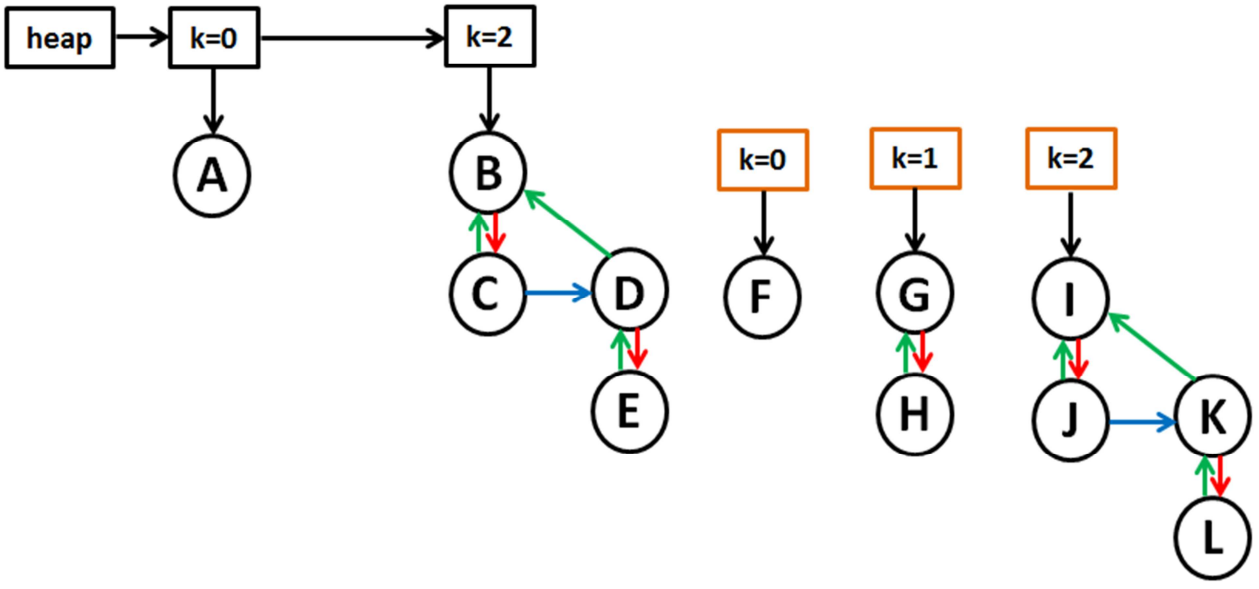


Figure: k^* new trees are generated with height $k = 0, \ldots, k^* - 1$.

ExtractMin

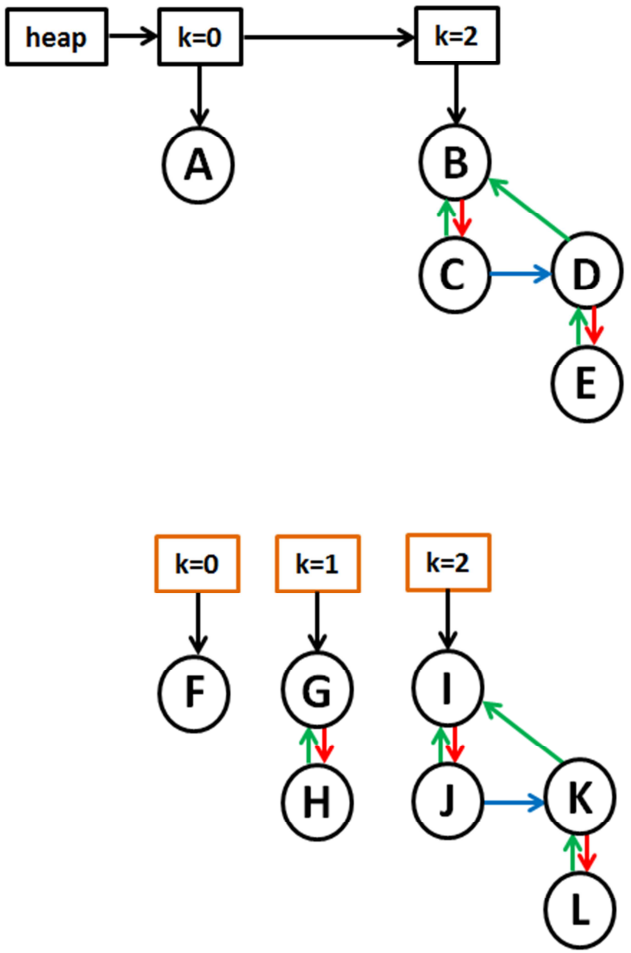


Figure: The trees must merged like binary digits must summed up in an addition.

ExtractMin

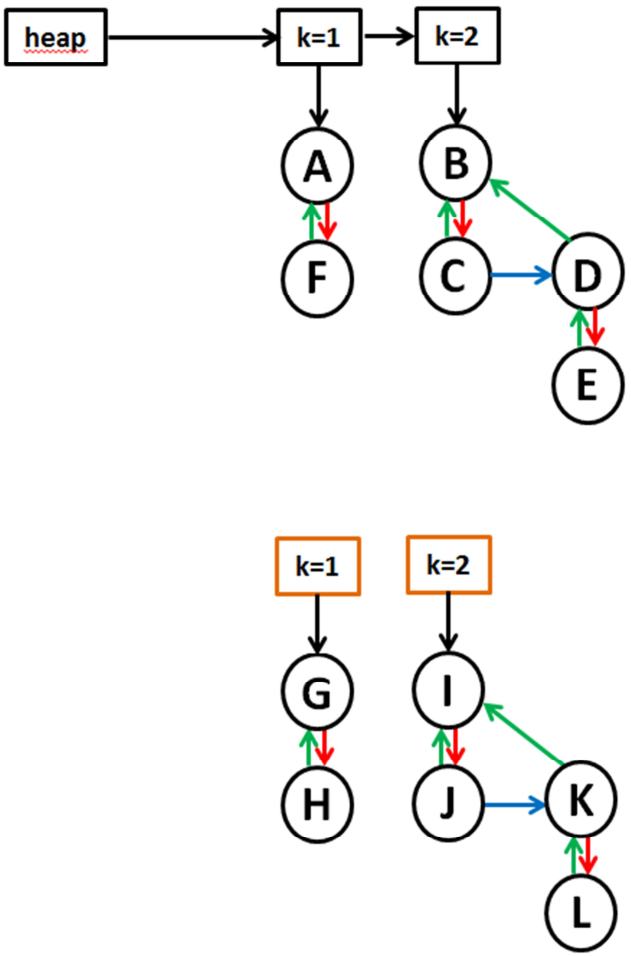


Figure: Two trees of height 0 are merged into a tree of height 1. Its root is the element with the minimum key among the roots of the merged trees.

ExtractMin

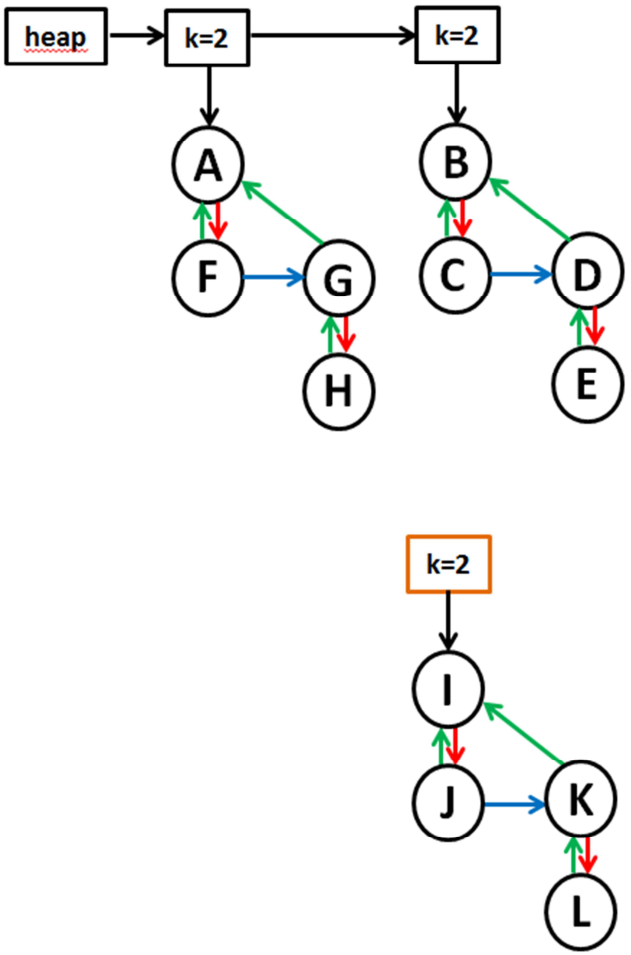


Figure: Now there are three trees of height 2. Any two of them can be merged into a tree of height 3.

ExtractMin

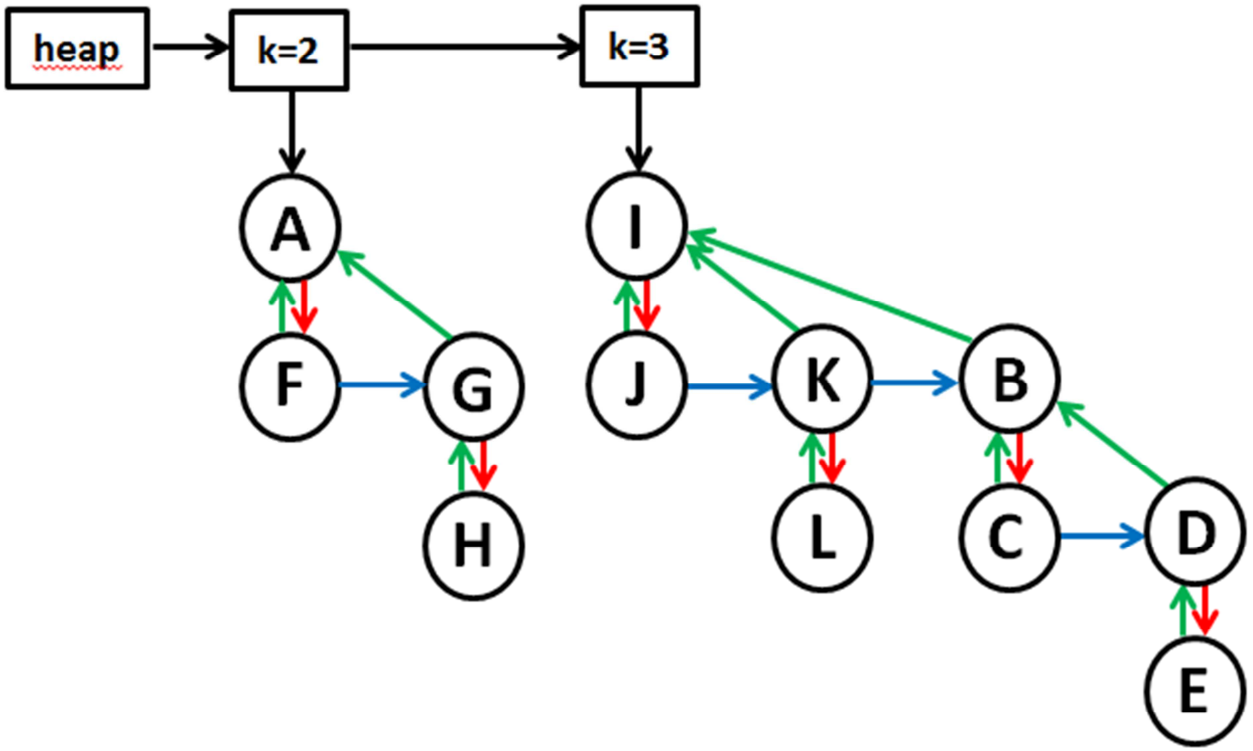


Figure: The final binomial heap corresponds to the binary encoding of $n = 13 - 1 = 12$, i.e. 1 1 0 0.

Insert

The new element is a tree of height 0.

Inserting it into the heap corresponds to increasing a binary number by 1.

The computational complexity of *ExtractMin* and *insert* is $O(\log n)$ in both cases, because it requires up to $\log n$ merge operations and each merge takes $O(1)$ time.

With binomial heaps:

- *Insert* does not require *MoveUp*;
- *ExtractMin* does not require *MoveDn*.

DecreaseKey

In a binomial heap with n elements the height of the trees is significantlty smaller than that of a binary heap with n elements.

Hence *DecreaseKey*, implying a repeated call to *MoveUp*, is faster.

Worst case: the element to be updated is a leaf of the largest tree and it must be moved up to the root.

The maximum height of a tree in a binomial heap with n elements is $\lceil \log_2 n + 1 \rceil - 1$.

Hence *DecreaseKey* has complexity $O(\log n)$.

An array of pointers is required to access each element in $O(1)$. Such an array does not require any update, because the binomial heap is implemented with pointers, not with an array.

Repeated *Insert* and *FindMin*

An uninterrupted sequence of m *Insert* takes only $O(m)$.
Its amortized complexity is $O(1)$.

Finding the minimum without extracting it takes $O(\log n)$.
However, we can keep a pointer to the minimum root, so that *FindMin* takes $O(1)$.
The time needed to update it is:

- $O(1)$ after each *Insert*;
- $O(1)$ after each *DecreaseKey*;
- $O(\log n)$ after each *ExtractMin*.

Fibonacci heaps

With Fibonacci heaps we can do:

- *Insert* in $O(1)$;
- *DecreaseKey* in $O(1)$;
- *ExtractMin* in $O(\log n)$.

The idea is “lazy update”: the data structure is updated (spending time for this) only under special conditions.

The $O(1)$ complexity above is achieved as *amortized complexity*, i.e. considering sequences of operations, not single operations.

Amortized complexity is more realistic because not all operations in a sequence require the worst case time.

Amortized complexity

The amortized complexity of an operation is the **average** worst-case time complexity that is achieved when a sequence of k operations are executed for k large enough.

The amortized complexity of an operation is $O(f(n))$ if it exists a positive integer k such that for a sequence of at least k operations the total time required by the sequence is $O(kf(n))$.

The most common technique to establish the amortized complexity of an operation uses a *potential function*.

A potential function is increased by some operations and decreased by others, so that the maximum number of times an operation can occur can be bounded by the number of times other operations occur.

Structure and implementation

A Fibonacci heap is made by a set of trees.
Their roots are linked in a doubly linked list.

An array *Pred* stores the predecessor of each element.
Successors of the same element are linked in a doubly-linked list, pointed by the predecessor.

Each element has a rank, indicating the number of its successors.

A pointer π^* indicates the root with the minimum key value.
An array *Lost* counts how many successors a non-root node has lost.
An array *Bucket* contains pointers to roots of rank $1, 2, \ldots, \lfloor \log_\Phi n \rfloor$.

Linking and cutting

All operations on a Fibonacci heap use two basic sub-routines:

- $Link(i, j)$: merges two distinct trees into a single tree. The roots i and j of the two trees must have the same rank.
- $Cut(i)$: separates the subtree rooted at i from its tree so that i becomes the root of a new tree.

In $Link(i, j)$, one of the two roots becomes a successor of the other (depending on their key values).

Both operations require $O(1)$ time to update the data structures.

Potential function

Consider a potential function τ defined as the number of trees in the heap.

Every *Link* operation decreases τ by 1 unit.
Every *Cut* operation increases τ by 1 unit.

Therefore, the number of executions of *Link* is bounded above by the number of executions of *Cut* plus the starting value of τ, which is at most n.

Invariant properties

A Fibonacci heap has the following invariant properties.

Property 1.
The key value of each element is less than or equal to the key values of its successors.

Property 2.
Every non-root node may have lost at most one successor after becoming a non-root node.

Property 3.
No two roots have the same rank.

Lower bounding the size of trees

Property 4.
Every (sub-)tree whose root has rank k contains at least $F(k+2)$ elements.

The proof is based on invariant Property 2 and invariant Property 3 and on the Fibonacci numbers sequence.

Proof.
Consider node w with rank k.

Sort its k successors according to the order in which they have been appended to w, from the earliest to the latest.

Consider node y, the i^{th} successor of w, with $1 \leq i \leq k$.

A lemma

Lemma. $rank(y) \geq i - 2$.

- Before linking y to w, w had at least $i - 1$ successors.
- When y had been linked to w the two nodes had the same rank. Therefore, at that moment y had at least $i - 1$ successors.
- After having being linked to w, y may have lost at most one successor.

Therefore y must currently have at least $i - 2$ successors.

Proof of Property 4

The lemma implies that the subtree rooted at w has a number of nodes at least equal to

$$G(k) = 1 + G(0) + G(0) + G(1) + G(2) + \ldots, + G(k-2) = 3 + \sum_{j=1}^{k-2} G(j).$$

Therefore $G(k) - G(k-1) = G(k-2)$ as with Fibonacci numbers. The base of the recursion is slightly different from Fibonacci numbers:

- $G(0) = 1$
- $G(1) = 2$
- $G(2) = 3$

	0	1	2	3	4	5 . . .
G	1	2	3	5	8	13 . . .
F	0	1	1	2	3	5 . . .

$$G(k) = F(k+2).$$

Bounding the rank

Property 5. The rank of all nodes in a Fibonacci heap is bounded by $\lfloor \log_\Phi n \rfloor$.

Lemma. $F(k+2) \geq \Phi^k$, where $\Phi = \frac{1+\sqrt{5}}{2}$.
(The proof is omitted here. It will be given later.)

Since the size of any tree is bounded by n, we have

$$n \geq size(T) \geq F(k+2) \geq \Phi^k$$

where k is rank of the root of tree T. Therefore,

$$k \leq \log_\Phi n.$$

Bounding the number of trees

Property 6. A Fibonacci heap contains at most $\lfloor \log_\Phi n \rfloor$ trees.

Proof. This property follows from:

- Property 3: No two roots have the same rank.
- Property 5: No rank can be larger than $\lfloor \log_\Phi n \rfloor$.

Primary and secondary *Cuts*

To keep invariant property 2 valid, we must ensure that every non-root node does not loose more than one successor after becoming a non-root node.

$Lost(i)$ indicates how many successors node i has already lost.

For the complexity analysis we use a potential function $\mu = \sum_{i=1}^{n} Lost(i)$.

Initially $\mu = 0$ and μ can never become negative.

Primary and secondary *Cuts*

When a $Cut(i)$ operation is done, two cases may occur. Note that $Pred(i)$ certainly exists, because i is not a root.

- $Lost(Pred(i)) = 0$: then $Lost(Pred(i))$ is set to 1 and the procedure stops.
- $Lost(Pred(i)) = 1$: then $Lost(Pred(i))$ is set to 2 but $Cut(Pred(i))$ is executed, to make $Pred(i)$ the root of a new tree. In turn $Cut(Pred(i))$ can trigger the execution of another *Cut* and so on.

We say that $Cut(i)$ is a primary *Cut* operation, while the *Cut* operations triggered by it are secondary *Cut* operations.

A cascade of secondary *Cut* terminates

- when a non-root j is reached with $Lost(j) = 0$;
- when the root of the tree is reached.

Complexity analysis

Consider a primary $Cut(i)$ operation.

- $Lost(i)$ is set to 0: it decreases by 1 unit or remains unchanged.
- $Lost(Pred(i))$ is increased by 1.

Therefore a primary cut does not increase the potential μ by more than 1 unit.

Consider a secondary $Cut(i)$ operation.

- $Lost(i)$ is updated from 2 to 0.
- $Lost(Pred(i))$ is increased by 1.

Therefore a secondary cut decreases the potential μ by 1.

Therefore, the number of secondary cuts cannot exceed the number of primary cuts.

Linking trees

Invariant property 3 requires that no two roots have the same rank.

Bucket(k) points to the root of a tree with rank k, if it exists. Otherwise *Bucket*$(k) = 0$.

When a new tree is generated (by a *Cut* operation for instance), two cases can occur.

- its root j has rank k and *Bucket*$(k) = 0$: then *Bucket*(k) is updated and the procedure stops;
- its root j has rank k and *Bucket*$(k) = i$: then we execute *Link*(i, j), generating a new tree whose root has rank $k + 1$, and the procedure is repeated.

A cascade of *Link* operations terminates when an empty position in the array *Bucket* is reached.

FindMin

This simply uses the pointer π^* to the root with minimum value of the key.

No pointer is modified.

Complexity: $O(1)$.

Insert(*i*)

A new tree with a single element *i* is created and it is inserted in the list of roots in $O(1)$.

The pointer to the best root is possibly updated in $O(1)$.

A sequence of *Link* operations can be triggered to restore invariant property 3.

Complexity: $O(1)$ plus the contribution due to the cascade of *Link* operations.

DecreaseKey(i)

The value of element i is updated in $O(1)$.

If the heap property 1 is violated by i and $Pred(i)$, then $Cut(i)$ is executed and π^* is updated in $O(1)$.

A cascade of secondary *Cut* can be triggered.

All trees generated in this way are temporarily stored in a list.

Then the list is scanned and each element of the list is inserted into the heap, possibly triggering a cascade of *Link* operations.

Complexity: $O(1)$ plus the contribution due to the cascade of *Link* and secondary *Cut* operations.

ExtractMin

The minimum key root element i is identified in $O(1)$.

For each successor j of i, $Cut(j)$ is executed. Overall this takes $O(\log n)$, because of the upper bound on the rank of i.

After each *Cut*, the new tree is immediately inserted into the heap, possibly triggering a cascade of *Link* operations.

At the end, *Bucket* is scanned to find the new root with minimum key value and to update π^*. This takes $O(\log n)$ because of the upper bound on the number of trees in the heap.

Complexity: $O(\log n)$ plus the contribution due to the cascade of *Link* operations.

Amortized complexity

To conclude the analysis we should bound the time taken by the cascades of secondary *Cut* operations and *Link* operations.

But we have proven that

- the number of secondary cuts cannot exceed the number of primary cuts;
- the number of *Link* operations is bounded by n plus the number of *Cut* operations.

Therefore, if we consider a long enough sequence of operations (at least n), the time taken by *Link* operations and by secondary *Cut* operations is bounded by a constant factor times the number of primary *Cut* operations.
Therefore we can establish the amortized complexity just counting primary *Cut* operations only.

Amortized complexity with Fibonacci heaps

Amortized complexity achieved with a Fibonacci heap:

- *FindMin*: $O(1)$
- *Merge*: $O(1)$
- *Insert*: $O(1)$
- *DecreaseKey*: $O(1)$
- *ExtractMin*: $O(\log n)$

One can also prove the following amortized complexity results:

- *IncreaseKey*: $O(\log n)$
- *Extract*: $O(\log n)$

Proving the lemma: $F(k+2) \geq \Phi^k$

Proof for n odd.

We use the following equations:

$$F(n) = \frac{\Phi^n}{\sqrt{5}} - \frac{(1-\Phi)^n}{\sqrt{5}}$$

$$F(n+2) = F(n+1) + F(n)$$

$$1 - \Phi = -\frac{1}{\Phi}$$

Proving the lemma for odd n

$$
\begin{aligned}
F(n+2) &= \frac{\Phi^n}{\sqrt{5}} - \frac{(1-\Phi)^n}{\sqrt{5}} + \frac{\Phi^{n+1}}{\sqrt{5}} - \frac{(1-\Phi)^{n+1}}{\sqrt{5}} = \\
&= \frac{1}{\sqrt{5}}(\Phi^n - (1-\Phi)^n + \Phi^{n+1} - (1-\Phi)^{n+1}) = \\
&= \frac{\Phi^n}{\sqrt{5}}(1 - \frac{1}{\Phi}(-\frac{1}{\Phi})^n + \Phi - \frac{1}{\Phi}(-\frac{1}{\Phi})^{n+1}) = \\
&= \frac{\Phi^n}{\sqrt{5}}(1 + (\frac{1}{\Phi})^{n+1} + \Phi - (\frac{1}{\Phi})^{n+2}) = \\
&= \frac{\Phi^n}{\sqrt{5}}(1 + \Phi + (\frac{1}{\Phi})^{n+1}(1 - \frac{1}{\Phi})) = \\
&= \frac{\Phi^n}{\sqrt{5}}(1 + \Phi + (\frac{1}{\Phi})^{n+1}(2 - \Phi)).
\end{aligned}
$$

Since $(\frac{1}{\Phi})^{n+1}(2-\Phi) > 0$, then $F(n+2) > \frac{\Phi^n}{\sqrt{5}}(1+\Phi)$.

Since $\frac{1+\Phi}{\sqrt{5}} > 1$, then $F(n+2) > \Phi^n$.

Proving the lemma for even n

Proof for n even.

We use the following equations:

$$1 + \frac{1}{\Phi} = \Phi$$

$$F(n+2) = F(n+1) + F(n)$$

$$F(n+2) > \Phi^n \ \forall n \text{ odd}.$$

Assume $F(n) \geq \Phi^{n-2}$ for an even n.

$$F(n+2) = F(n+1) + F(n) \geq \Phi^{n-1} + \Phi^{n-2} =$$

$$= \Phi^{n-1}(1 + \frac{1}{\Phi}) = \Phi^{n-1}\Phi = \Phi^n$$

Inductive step: $F(n) \geq \Phi^{n-2}$ implies $F(n+2) \geq \Phi^n$.
Induction basis: $F(4) = 3 > \Phi^2$.

Therefore $F(n+2) > \Phi^n$ also for all n even.