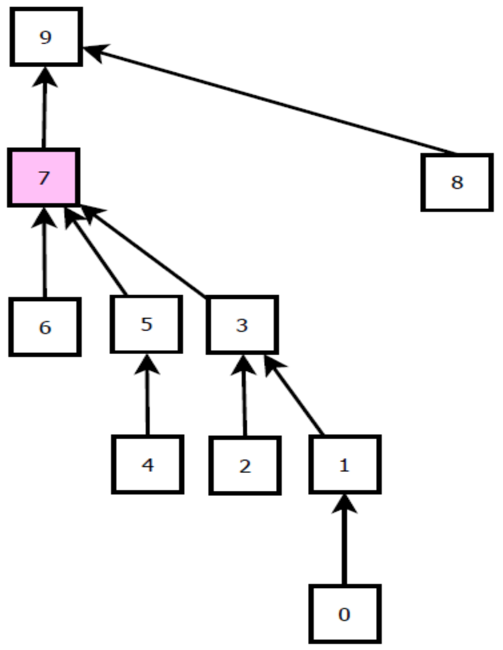
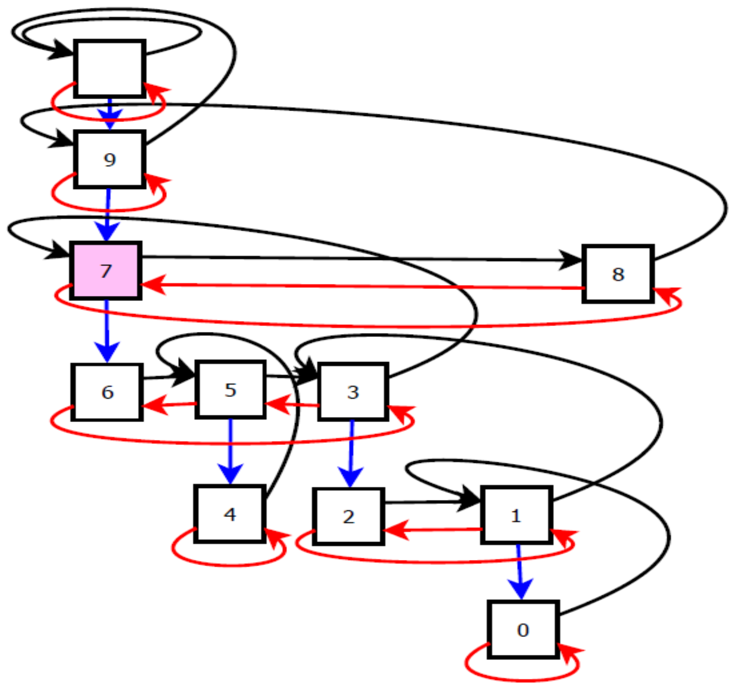
Padovan heaps

Vladan Majerech





Main principle – superexpensive comparisons

- We compare as late as possible (FindMin does almost all work).
- We never forget results of comparisons.
- Organization of comparisons should not take asymptotically more time than comparisons.

Insert

- Insert new element to the list of minimum candidates.
- O(1).
- We should accumulate O(1) more time for future work – we will see later.

FindMin

- Compare candidates for minimum (element for which we don’t know smaller one)
- We remember results of comparisons (denoted by edge directed to smaller element), representation of the directed forest would be discussed later.
- At the end we have just one candidate for minimum.
- Account Φ_0 equal to the number of candidates for minimum can pay for comparisons and increments cost of Insert by O(1).

DeleteMin

- We suppose FindMin preceded, we can call it to be sure.
- We remove minimum, its predecessors (children in its tree) become current candidates for minimum.
- It could be implemented in O(1) time, but we should add number of predecessors to Φ_0 for our amortized analysis.
- We need narrow trees to keep cost of DeleteMin small.

Invariant of c-q narrow trees

There will be rank defined for a heap element v denoted $rank_v$. (New element gets rank 0).

- For c>0 and $1<q\leq 2$ holds:

Subtree of ancestors of an element of rank k has size at least cq^k.

Rank will not be equal to the element indegree (number of children). To be able to pay to Φ_0 during DeleteMin, we need account for situations when indegree exceeds rank. We define

$$\Phi_1 = \sum \deg\_v - rank\_v$$

Summing just positive differences.

DeleteMin again

- Suppose c-q narrow trees invariant holds, than ranks are bounded by $\log(n/c)/\log q$, so after removal of the minimum m, number of candidates become $\deg\_m \ = \ rank\_m + (\deg\_m - rank\_m)$, so except the transaction from $\Phi_1$to Φ_0, we have to increase Φ_0 by O(log n) only.
- Even the worst case for DeleteMin could be O(1), it has amortized cost O(log n) in our analysis.

Maintaining c-q narrow trees invariant

- We cannot compare minimum candidates randomly during FindMin, as that would generate wide trees.

1. It’s fine to compare two elements of the same rank k as we get resulting tree size $2cq^k \geq cq^{\{k+1\}}$, so we can increase resulting tree root v rank and difference $\deg_v - rank_v$ does not change.
2. Just in the case at most one candidate per rank remains, we can compare them arbitrary. We pair the candidates and compare the pairs, than we create new pairs … We want to avoid comparison of one candidate with all others. We don’t change ranks during 2nd phase, we increment Φ_1 instead.

FindMin’s cost

- First phase is paid from Φ_0.
- We didn’t accumulated enough time to be able to pay to Φ_1 during 2nd phase.
- Number of candidates for minimum at the start of 2nd phase is at most (1+maximal achievable rank)∈O(log n). At the same time their number cannot increase during the 1st phase. Let us introduce new account Φ_2 equal to minimum of maximal achievable rank + 1 and the number of candidates for minimum.
- Cost of some Inserts are increased by paying O(1) to Φ_2, DeleteMin pays at most O(log n) to Φ_2.

Implementation details

- We maintain candidates for minimum in bidirectional list (acyclic right and cyclic left). We maintain pointer to the leftmost element what allows us to insert to both ends and remove any member in O(1).
- Exactly the same way we maintain predecessors of all elements (children in the corresponding tree).

FindMin’s implementation details

- We have synchronization place for each rank (empty).
- During the 1st phase we traverse list of candidates right and update pointers in synchronization places to traversed candidates of corresponding ranks. Whenever synchronization place points to anther candidate we compare them move higher of them to right end of predecessors of the smaller one. We increase rank of smaller one at the same time, therefore we should empty the original rank synchronization place and we have to reprocess the new root (paid from Φ_0).
- As the list of candidates is just updated, we can traverse it once again to clean up synchronization places between 1st and 2nd phase (paid from Φ_2).
- During 2nd phase we repeat moving left and comparing current candidate with candidate to the left. We move higher candidate to the start of predecessors (children) of the smaller one. We end when only one candidate remains (paid from Φ_2).

Decrement (so far we have Binomial heaps)

- After an element decrement we cannot rely on edge leaving it so we have to remove it (all other edges remain valid).
- Would c-q narrow trees invariant still hold?
- If we decrement rank of all it's descendants c-q narrownes would be restored, but the update size would equal the element depth, what could be up to Ω(n).
- Tarjan – Fredman shown, propagation of each second rank decrement suffices for maintaining the invariant. That lead to introduction of white and black colors.

Decrement analysis

- Elements would be created white. First rank decrement gives them black color.
- FindMin during 1st phase elements whose are removed from candidates list are colored white. During 2nd phase elements whose are removed from candidates list are colored red.
- Let Φ_3=number of all black successors.
- During Decrement just one element becomes black. We could color a black vertex red and propagate rank update. This propagation is fully paid by decrement of Φ_3.
- DeleteMin worst case cost grows to Φ_0 increase so $\theta(n)$ as we need successor(parent) pointers updated.

Fibonacci heaps

- Let M_k be minimal possible size of ancestors tree of an element of rank k.
- Following recurrence holds:
$$M_k = 1 + 1 + M_0 + M_1 + \cdots + M_{k-2}$$
- So $M_{k+1} = M_k + M_{k-1}$ and we got Fibonacci sequence and c-q narrowness for $q = \frac{1+\sqrt{5}}{2}$.

This finishes Fibonacci heaps analysis.

P.S.: Analysis would work even when using white color instead of red.

We just should prevent decrements of ranks below 0.

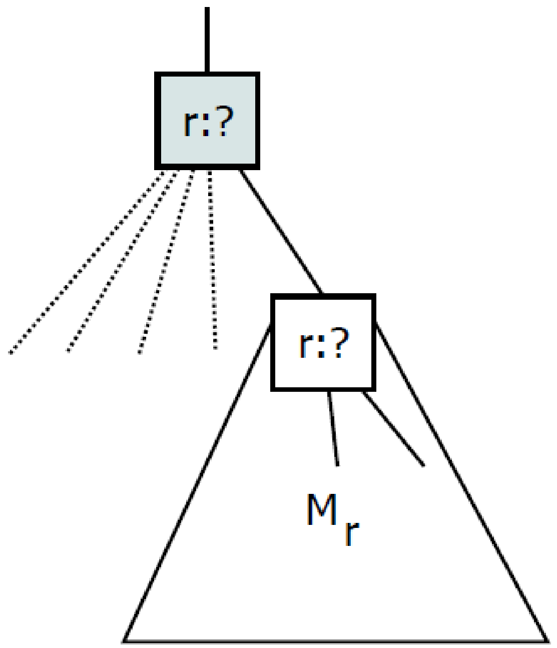
Saving one pointer per element

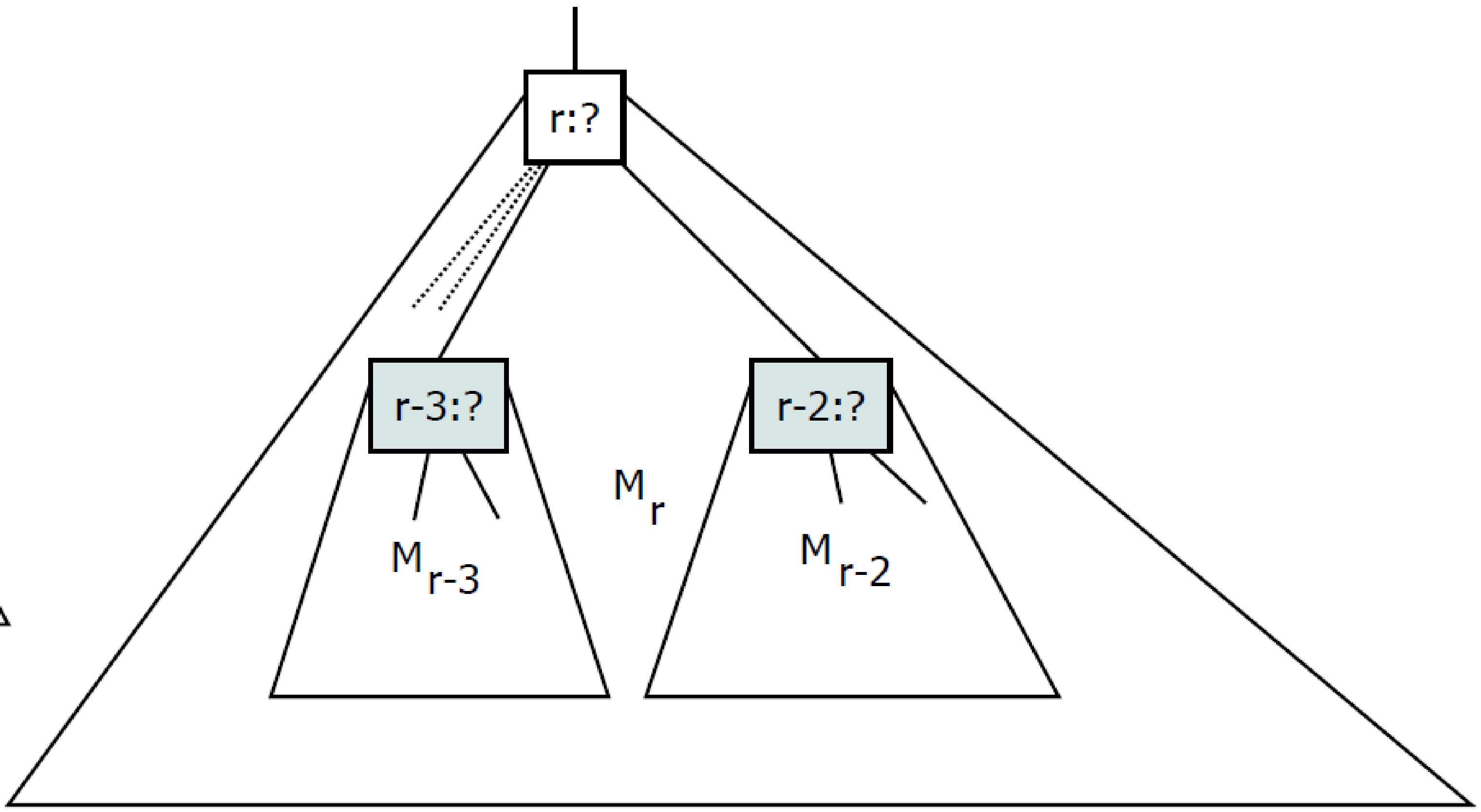
- Decrement in Fibonacci heaps require pointer to successor (parent) to allow rank decrement propagation.
- We save the space by replacing nil pointers at the right ends of lists by pointer to successor (parent). We would not maintain pointers to successors (parents) on other places.
- In that case we cannot propagate rank decrements except at the right end of the list where we can access the successor.
- We would propagate rank recomputation only at the rightmost two elements (elements of highest rank among white and black).
- Worst case time for DeleteMin will be O(1) again.

$rank\_v$ and $wrank\_v$

- Let us define $wrank = 1 + rank$ for a black element, and $wrank = \ rank$ for a white one. Let $wrank = -1$ for $null$ pointers (missing vertices).
- Predecessor (children) lists have all red on left, than $wrank$ increasing (when defined).
- Take among black and white predecessors last two … w_0, w_1 from right (they could be missing).
  - S. Safe: $wrank_{w_1} + 1 = wrank_{w_0}$: $rank_v = wrank_{w_0} + 1$.
  - D. Dangerous: $wrank_{w_1} + 1 < wrank_{w_0}$ and w_0 is white: $rank_v = wrank_{w_0}$. (v cannot be white)
  - F. Forbidden: $wrank_{w_1} + 1 < wrank_{w_0}$ and w_0 is black: color w_0 yellow and recompute.

Minimal size recurrence





Narrowness

- Let M_k be minimal size of an ancestors tree of an element of rank k
- Following recurrence holds:
$$M_k = 1 + M_{k-2} + M_{k-3}$$
- This is closely related to Padovan sequence and c-q narrowness holds for q=$\sqrt[3]{\left(\frac{1}{2}\left(1+\sqrt{\frac{23}{27}}\right)\right)} + \sqrt[3]{\left(\frac{1}{2}\left(1-\sqrt{\frac{23}{27}}\right)\right)} \approx 1.324718$
- $(q^3 = q + 1)$

Decrement fully colored

- We cut an element and if it was one of last two elements of a list we call cascading rank consolidation on it’s successor (parent).
- During cascade consolidation we start by computation of current rank. If there is no change, we are done.
- If rank of a white element drops by 1, it is colored black.
- If rank of a black element drops, it is colored yellow as well as if rank of a white element drops by at least 2.
- If an element is among last two of the list we continue the rank consolidation at the successor (parent).
- During rank calculation of an element we move all yellow predecessors(children) among right two elements to the left end. We color them red during it. If there is a red element among rightmost two, we know there is no more white and black element to the left in the list.

Decrement

- Decrement analysis still does not guarantee O(1).
- Rank computation can lead to drop by more than 1 and propagation continues even for originally white vertex. Such propagation cannot be paid from Φ_3.
- We need another account

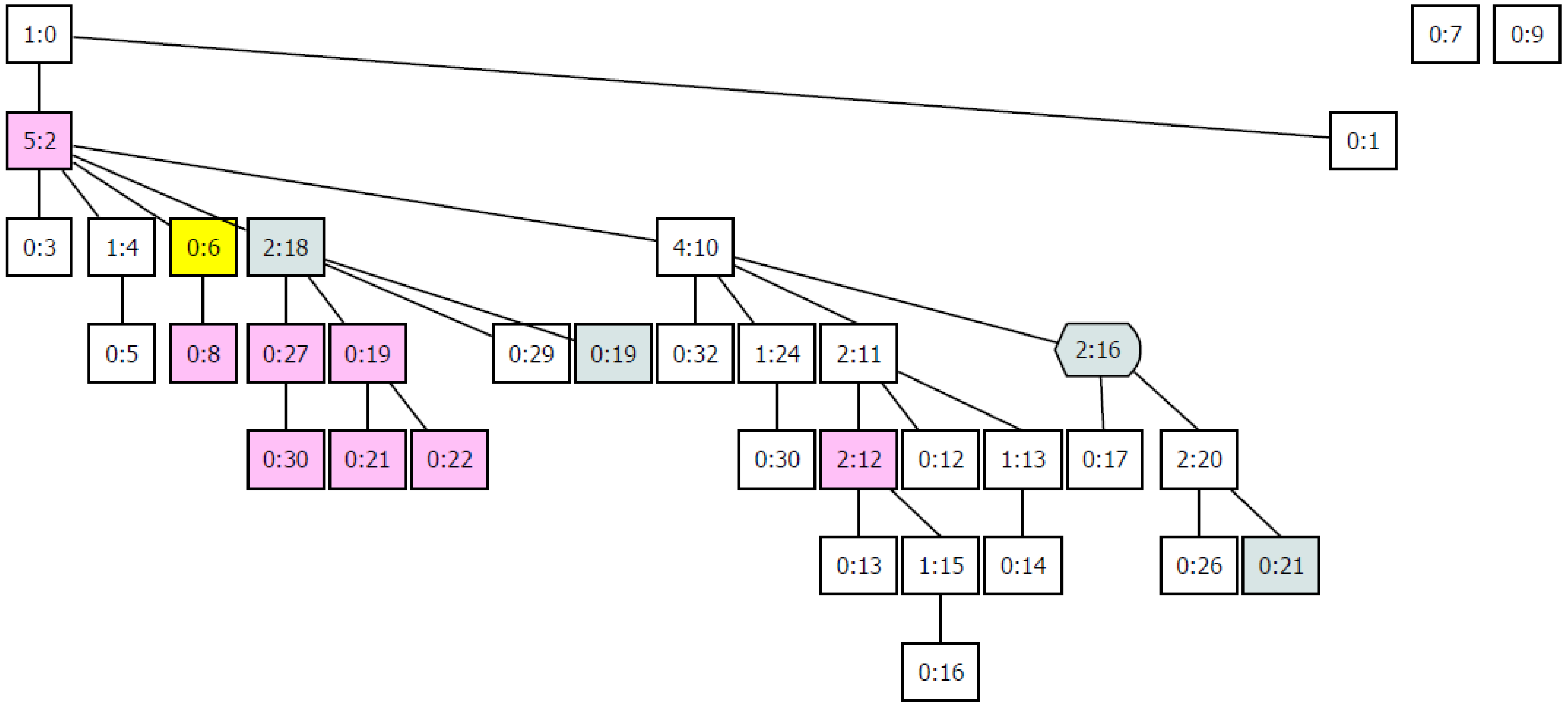
$\Phi_4 = \sum rank_v$ $-$ number ot predecessors of v colored either white or black.

- Removal of white or black predecessor accumulates time for future (higher) rank drop. Join of these during FindMin does not change Φ_4. DeleteMin could decrement Φ_4 without using it. The only method increasing and using Φ_4 is Decrement.
- Now rank update propagation is paid alternatively by either Φ_3 or Φ_4. Except that (and moves of yellow elements) we spent only constant cost.

Red, Yellow and Dangerous

- Due to delaying move of yellow elements to left end of the list we need account Φ_5 = number of all yellow successors.
- DeleteMin does not need $\sum \deg_v - rank_v$. It suffices Φ_1=number of all red successors. As $\deg\_v =$ number of red + number of yellow + number of (white or black), where number of white and black $\leq rank$, so we pay to Φ_0 from Φ_5, Φ_1 and at most $O(\log n)$.
- We need to prevent white dangerous vertices, therefore we make dangerous vertices safe at the start of the FindMin. This is why we introduce account Φ_6 = number of all dangerous vertices.

Picture with all colors (negated keys)



Summary

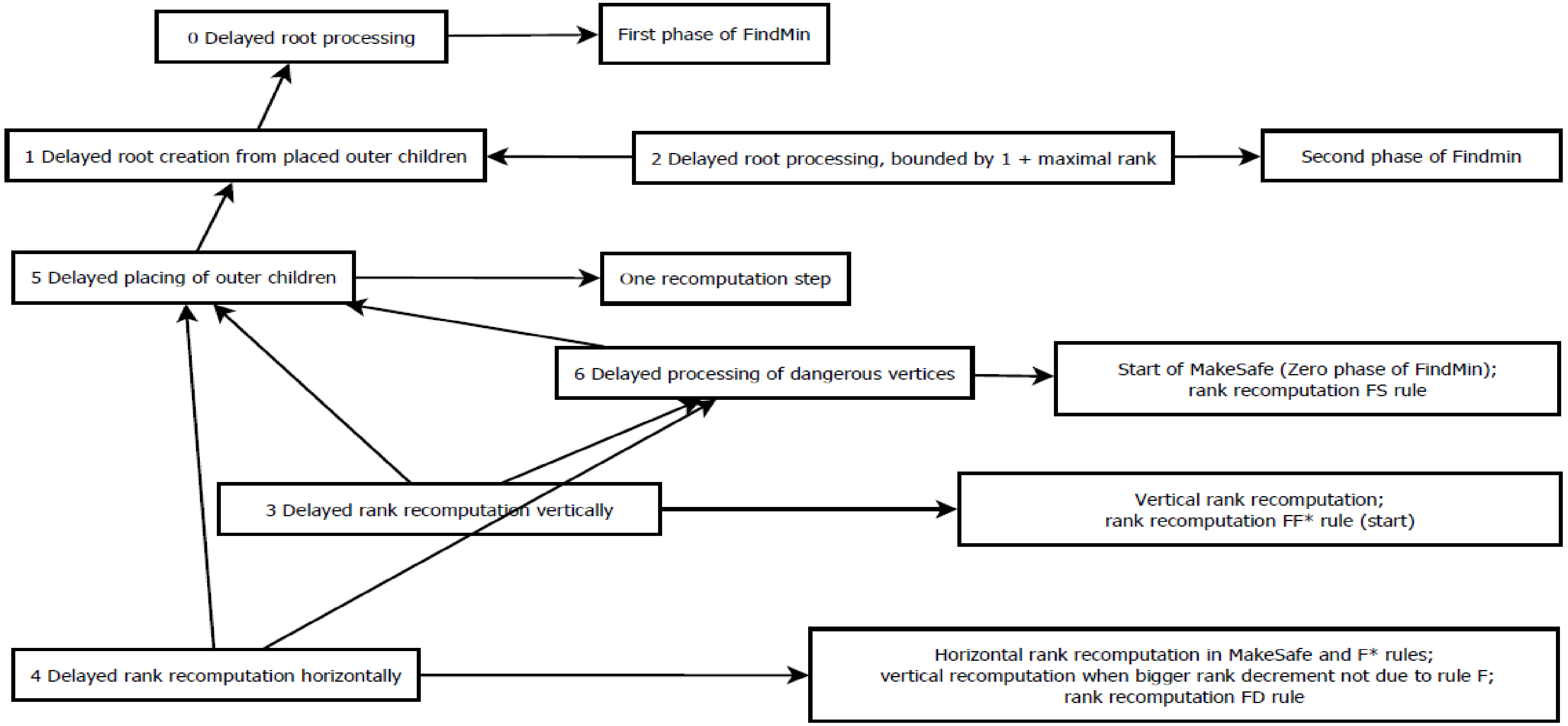
- Potential used in analysis is

$$\Phi_0 t_0 + \Phi_1 t_1 + \Phi_2 t_2 + 2\Phi_3 t_3 + \Phi_4 t_4 + \Phi_5 t_5 + \Phi_6 t_6,$$

where $t_0 \leq t_1 < t_2, t_1 < t_5 < t_6, t_5 + t_6 < t_3$, and $t_5 + t_6 < t_4$ are appropriate constants

- Φ_0 is number of candidates for minimum.
- Φ_2 is number of candidates for minimum, bounded by highest achievable rank+1.
- Φ_1, Φ_3, resp. Φ_5 is number of all red, black, resp. yellow descendants (children).
- Φ_4 is sum of differences between rank and number of white and black descendants (children) of an element.
- Φ_6 is number of dangerous vertices

The payment schema



0 Delayed root processing
First phase of FindMin
1 Delayed root creation from placed outer children
2 Delayed root processing, bounded by 1 + maximal rank
Second phase of Findmin
5 Delayed placing of outer children
One recomputation step
6 Delayed processing of dangerous vertices
Start of MakeSafe (Zero phase of FindMin); rank recomputation FS rule
3 Delayed rank recomputation vertically
Vertical rank recomputation; rank recomputation FF* rule (start)
4 Delayed rank recomputation horizontally
Horizontal rank recomputation in MakeSafe and F* rules; vertical recomputation when bigger rank decrement not due to rule F; rank recomputation FD rule

Data structure’s competition

- Supporter of one structure generates sequence of method calls, both structures invoke the methods and ratio of total times is the gain. The game is repeated with roles changed.
- Heaps according superexpensive comparison principle will won against standard implementation by $\theta(\log n)$ using prefix of a sequence

  i=0;Repeat (Insert(-i), Insert(-(i+1)), FindMin, DeleteMin, i++),

  because maximal rank achieved would be 4.
- Standard heaps could gain at most $\theta(1)$ in revenge.