Fibonacci Heap and Soft Heap

Zhuang Zeming, 2021232134

Guo Jinhan, 2020232158

I. Fibonacci Heap

A. Data Structure analysis

The Fibonacci heap is a collection of a series of small root heaps, that is, if each node has a parent node, its value is not less than the value of its parent node. Each node class contains seven member variables, which are a pointer to the parent node, a pointer to one of the child nodes, two pointers to the left and right brothers, an integer value, an integer rank, and a Boolean variable. The specific node structure and the schematic diagram of the heap structure are shown in Figures 1 and 2.

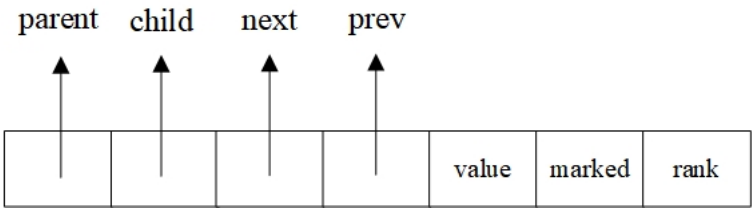

Fig. 1. Fibonacci heap node class member variables

B. operations

1) insert(int $value$): The new node is inserted into the circular linked list composed of the root node. The value of the newly generated node is determined by the input. Both $next$ and $prev$ pointers point to themselves, the value of $rank$ is 1, and both $parent$ and $child$ point to $NULL$.

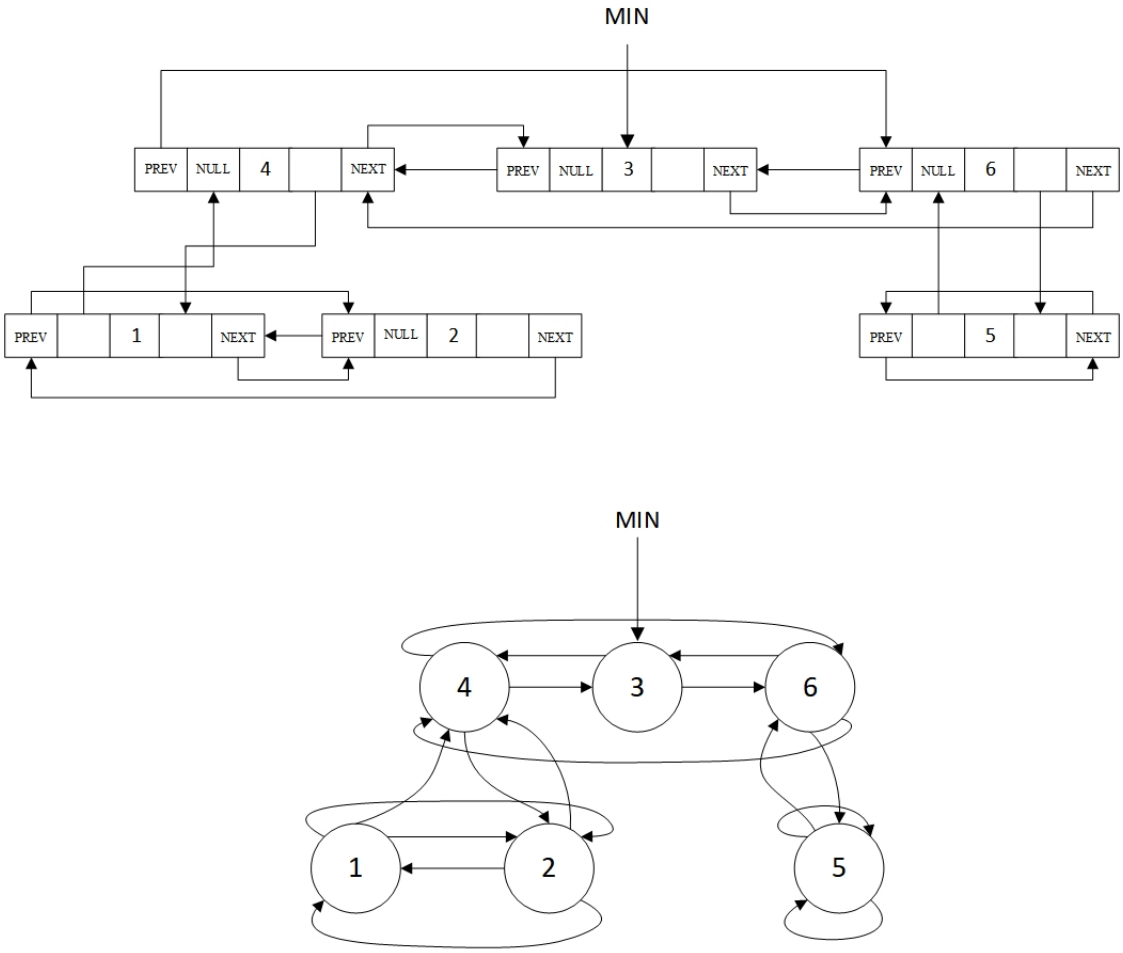

Fig. 2. The specific structure of the internal nodes of the Fibonacci heap

2) pop(): The effect of the pop operation is to delete the node with the smallest value in the heap from the heap and return the smallest value. The heap needs to be maintained after popping the minimum node. Before the minimum value node is deleted, all child nodes of the minimum value node need to be separated from the minimum value node and added to the circular linked list composed of the root node. After the minimum value node is popped, the $rank$ of each sub-tree in the heap cannot be the same, so all trees with the same $rank$ in the circular linked list composed of the root node need to be merged together. The process is shown in Figure 3.

3) Delete value: The operation of deleting a node is divided into two steps to complete: decrease value and pop. If we want to delete a node whose value is key, we first call _find(key) to find the node and modify its value to $MIN-1$, and then perform heap maintenance. Obviously, after maintenance, the minimum value in the heap has become the node to be deleted, so we

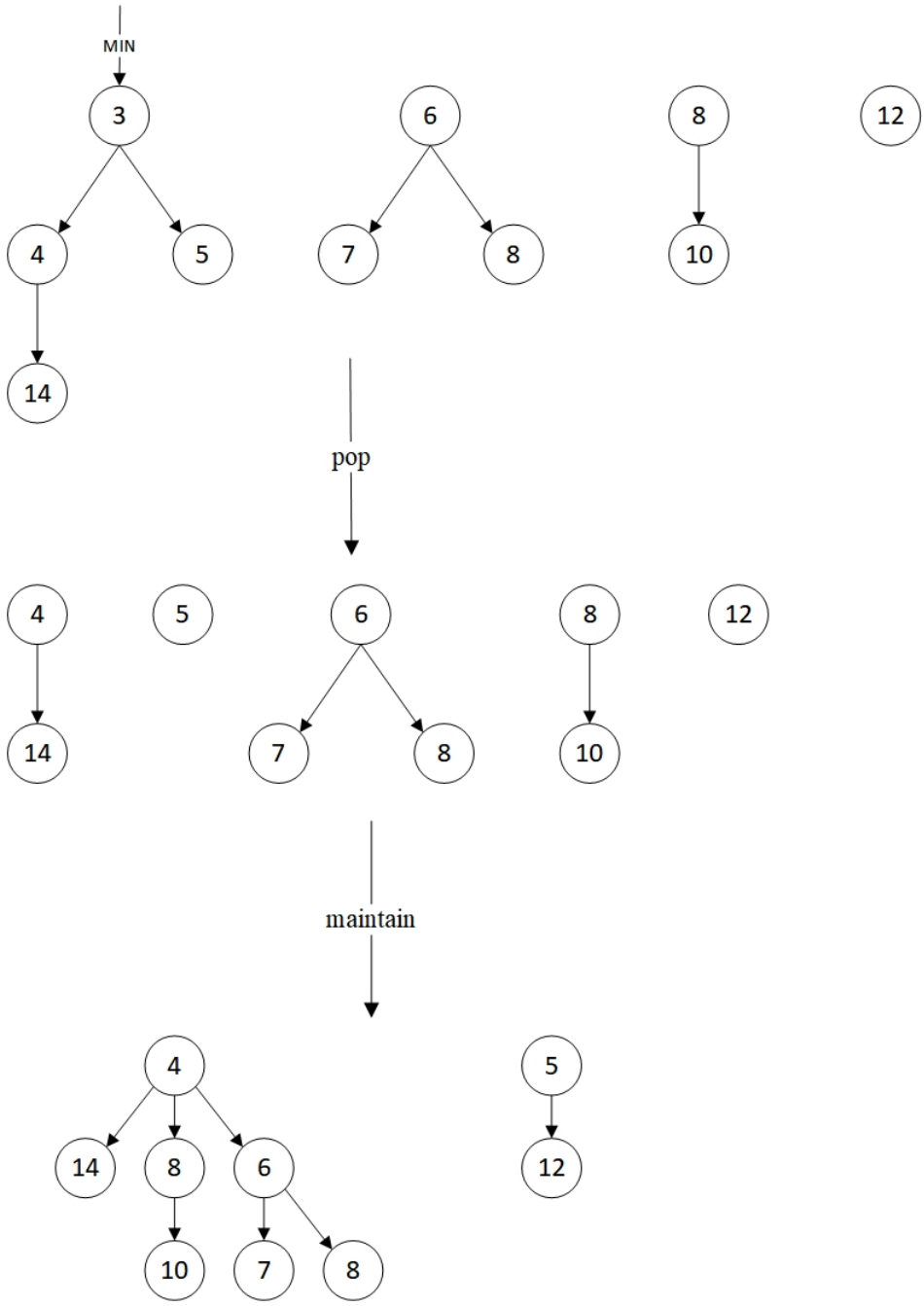

Fig. 3. pop() operation process

only need to perform another pop operation to delete the node from the heap.

C. Operation complexity

In theory, the operation of deleting any node of the Fibonacci heap can be completed within $O(logn)$ amortized time complexity, and the rest of the operations can be completed within $O(1)$ amortized time complexity. Compared with ordinary binary heaps, Fibonacci heaps can complete the operation of inserting nodes more efficiently, but the operation of deleting nodes requires a higher time.

TABLE I

TABLE 1: COMPLEXITY COMPARISON OF FIBONACCI HEAP AND PRIORITY QUEUE

operations	Priority queue(binary heap)	Fibonacci heap
insert	$O(log(n))$ (worst)	$O(1)$ (amortized)
pop	$O(log(n))$ (worst)	$O(log(n))$ (amortized)

D. user instructions

When the program starts to run, enter the integer 1 or 2 to select the function. Function 1 is to show the running time comparison results of the insert operation and pop operation of the Fibonacci heap. Function 2 is to perform the insert, pop and delete operations of the Fibonacci heap. The operation instructions are as follows.

1) insert: Input format: "insert n", n is the positive integer you want to insert.

2) pop: Input format: "pop". Pop up the minimum node in the Fibonacci heap.

3) delete: Input format: "delete n", n is the positive integer to be deleted.

4) print all: The output result is the BFS traversal result of the Fibonacci heap.

E. demo results

TABLE II

TABLE 2: EXPERIMENTAL RESULT COMPARISON

operations	Data	Priority(ms)	F-Heap(ms)
insert	1×10^5 random int	13	7
pop	1.5×10^4 random int	7	9

1) result of function 1: Note: The random number generated during each run is different, and the result may be different

2) result of function 2: The test inputs are: insert 1, insert 2, insert 3, insert 4, insert 5 print_all, pop, print_all, delete 5, print_all. The results are as follows:

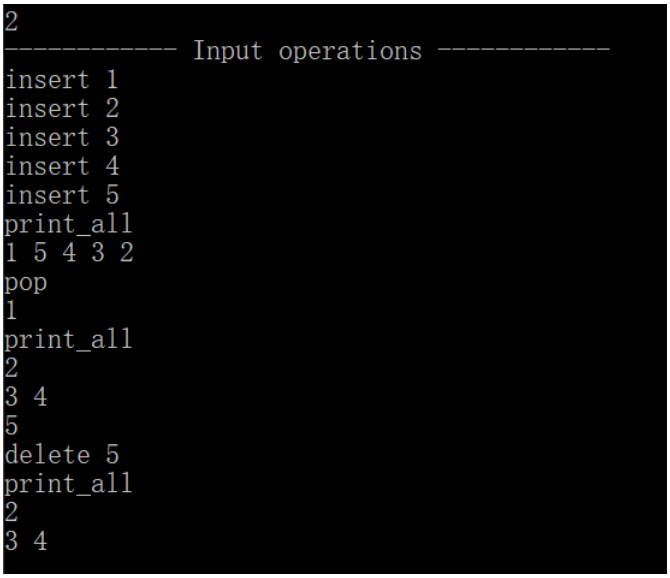

Fig. 4. test results

II. SOFT HEAP

A. Data structure analysis

A min-heap is a tree-based data structure that stores values in a certain way so that the heap satisfies the heap property, which can be represented that if v is a parent of u, then the key in v must be smaller than that in u. Soft heap [2] is a min-heap which has a significant speed up in popping out the minimum element. The regular min-heap deletion callbacks a maintenance, that is, one has to move up another item to the root. This takes $O(\log n)$ time. To achieve this feature, some keys stored in soft heap are corrupted by replacing them with a larger key. The findmin operation returns the current minimum key, which might or might not be corrupted. The benefit is speed: during heap updates, items travel together in packets in a form of ”carpooling”, in order to save time. Chazelle has proved a theorem that for any $0 < \varepsilon \leq 1/2$, a soft heap with error rate supports each operation in constant amortized time, except for insert, which takes

$O(\log 1/\varepsilon)$ time and the data structure never contains more than εn corrupted items at any given time.

The soft heap can be considered as a forest of binomial trees with one head (stored in a doubly linked list) pointing to the root node of every tree.

A binomial heap [3] is implemented as a set of binomial trees (compare with a binary heap, which has a shape of a single binary tree), which are defined recursively as follows:

1) A binomial tree of order 0 is a single node
2) A binomial tree of order k has a root node whose children are roots of binomial trees of orders $k-1, k-2, ..., 2, 1, 0$ (in this order).

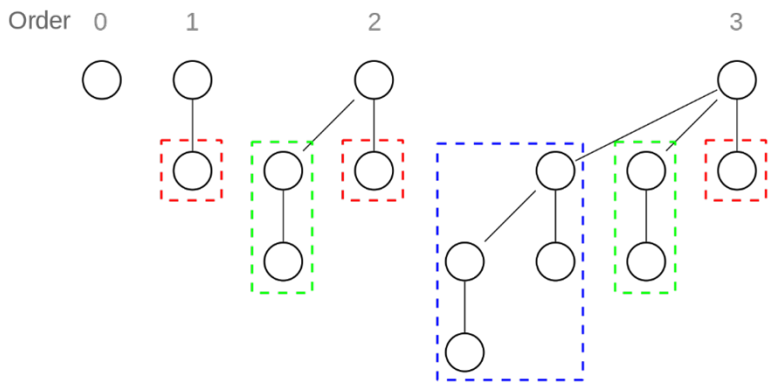

Fig. 5. Examples of binomial trees

The binomial trees used in soft heap (which are called soft queues) are derived from a master-tree (regular binomial tree). But they are slightly modified that the rank of a node will not change when deleting its child node. The rank of a node only depends on the number of its child in the master-tree, and the rank of a soft queue is that of the root node of that queue. For convenient operations, the soft queue is stored in a left child right sibling binary tree, which one node has its child and its next (sibling).

The soft heap has two invariants:

1) Rank-invariant: the number of children of the root should be no smaller than $rank(root)/2$.
2) Next-invariant: we always store the child with the smaller ckey as next.

B. Operations

The soft heap we implemented supports 5 operations:

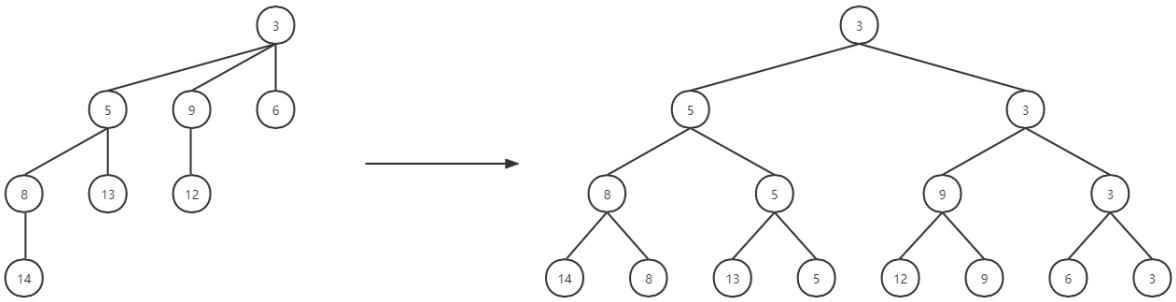

Fig. 6. Reforming binomial tree into LCRS binary tree

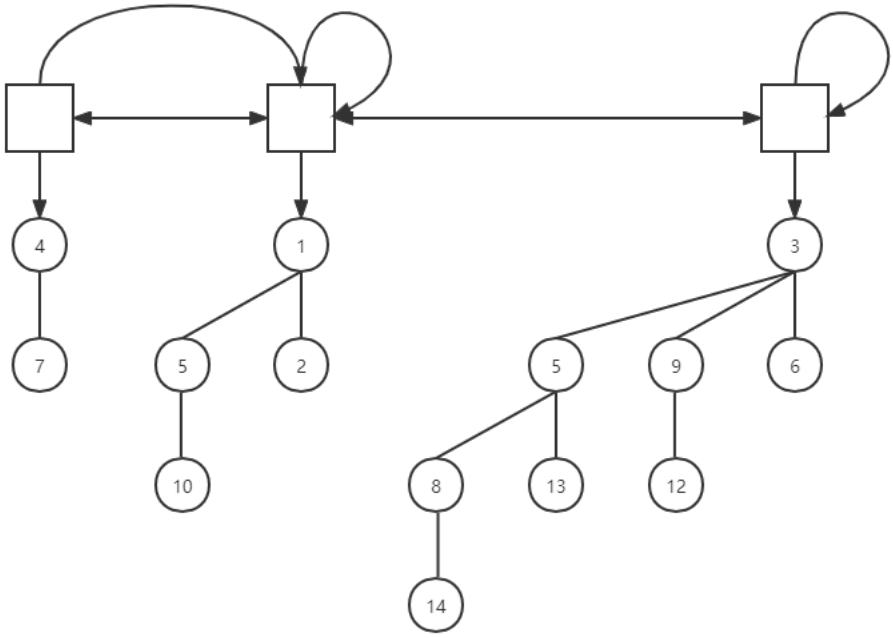

Fig. 7. An example of a soft heap

1) Insert: When inserting a new key into the heap, we simply create a new soft queue with only one node of that ckey and meld it into the heap.

2) Meld: When melding two heap H_1, H_2 together, for simplicity, we assume that H_1 has fewer queues (heads) than H_2. Since one soft can only have one rank-k queue, so when melding a rank-k queue q_1 in H_1, we need to find the first queue q_2 in H_2 that rank(q_2)$\geq k$. If rank(q_2)$> k$, then just add one head pointing to q_1 before q_2's head. Otherwise, it needs to merge two rank-k

trees. Let the binomial tree with larger ckey of root be the child of another tree to create a rank-$(k+1)$ tree. Then repeat the merge operation if there is already a rank-$(k+1)$ tree in H_2. Finally fix the suffix_min pointer in heads.

3) Deletemin: When deleting the minimum, we first find the head to which suffix_min points. Then if the item-list of the root of that queue contains only one key x, then return x and delete it. Or we return the first key and delete it. It may occurs that key x is corrupted, and that's why soft heap has a probability of returning an error key. However, the item-list may be empty. Then we need to refill (rearrange) the item-list of this subtree, which is denoted as sift operation.

The idea of sifting is to start at the root of the binary tree and recursively go down to a leaf and collect the items of the lower nodes on the way back up. If the nodes on the recursion path fulfill certain conditions, we recursively perform the operation again so that the recursion tree is branching and we move up even more items. First we empty the items-list(v) as it will be refilled by new items from further down the tree.

If this node is a leaf, we set its ckey(v) to ∞, indicating that we visited this leaf. Nodes whose ckeys are set to ∞ will be deleted during the clean-up procedure. The leaf's initial ckey is not lost when we set it to ∞ - its initial value is stored in its parent node. If this is not a leaf, we recursively sift its "next" child. After sifting the child, we need to check the next-invariant and item-list of v is set to item-list of $v \to next$ and v's ckey is replaced by $v \to next$'s ckey. After sifting, we execute clean-up. If both ckey of v's children are ∞, then we delete both children and v becomes a leaf. If only ckey of v's child is ∞, we set $v \to child = v \to next \to child$ and $v \to next = v \to next \to next$. Recall that each queue has an underlying master-tree. The changes in the soft queue will affect the master-tree. Thus, the ranks of the nodes are not changed during this process.

4) Findmin: Finding minimum is to do the same as deleting minimum except for deletion.

5) Delete: When deleting a specific key k in heap (which is very NOT recommended for a heap data structure), we traverse every soft queue in the heap by deep first search (DFS). If k is found in a item-list, then just delete it if there exists some other keys in the item-list. Or we need to delete the node as well. If k is found in a ckey of a node without item-list ($rank < r$), just delete the node by setting its ckey to INF and then sift the whole tree.

C. Experiment result

Input format:

TABLE III

TABLE 1: RESULTS OF $N = 100000, r = 1$

operations	Priority queue(binary heap)	soft heap
insert	11ms	22ms
pop	56ms	21ms

The first line reads r, then each line indicates one of four operation as below:

1) "i x" means insert a key x
2) "d x" means delete a key x
3) "p" means pop a minimum key
4) "e" means exit with summary

Input sample:

```
1
i 36
i 2
i 1
i 10
i 2
i 5
i 3
i 0
i 14
d 14
d 0
d 14
d 1
i 10
d 2
p
p
p
p
p
p
p
e
```

Output sample:

```
r = 1
1: insert 36
2: insert 2
3: insert 1
4: insert 10
5: insert 2
6: insert 5
7: insert 3
8: insert 0
9: insert 14
10: delete 14
11: delete 0
12: delete 14 fail! Not found!
13: delete 1
14: insert 10
15: delete 2
16: pop 2
17: pop 3
18: pop 5
19: pop 10
20: pop 10
21: pop 36
22: pop fail! Empty heap!
23: exit
insert: 10
delete: 4
pop: 6
```

III. WORKLOAD

Zhuang Zeming: Soft heap analysis, code writing, testing and report writing

Guo Jinhan: Fibonacci heap analysis, code writing, testing and report writing

REFERENCES

[1] Fredman M L, Tarjan R E. Fibonacci heaps and their uses in improved network optimization algorithms[J]. Journal of the ACM (JACM), 1987, 34(3): 596-615.

[2] Chazelle, Bernard. ”The soft heap: an approximate priority queue with optimal error rate.” Journal of the ACM (JACM) 47.6 (2000): 1012-1027.

[3] Wikipedia contributors. ”Binomial heap.” Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 22 Dec. 2021. Web. 27 Dec. 2021.

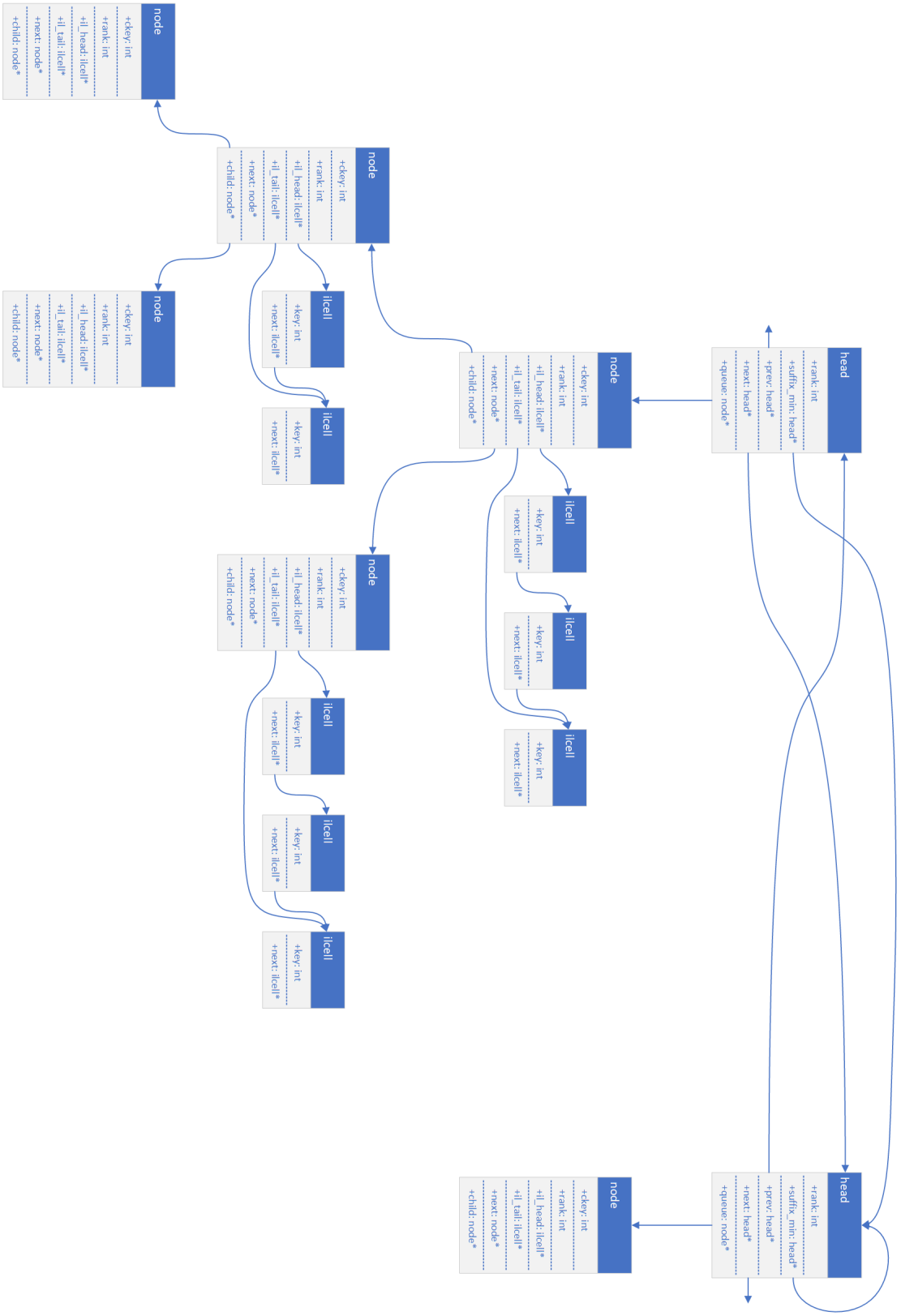

Fig. 8. An complete example of a soft heap

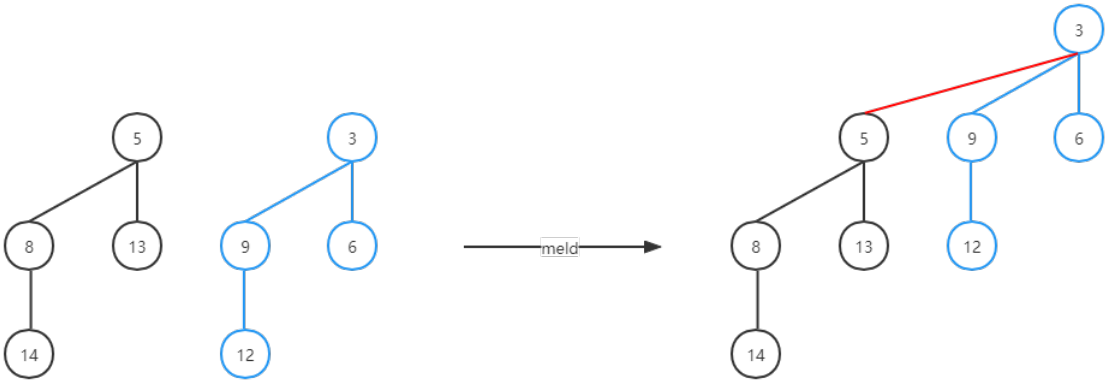

Fig. 9. Melding

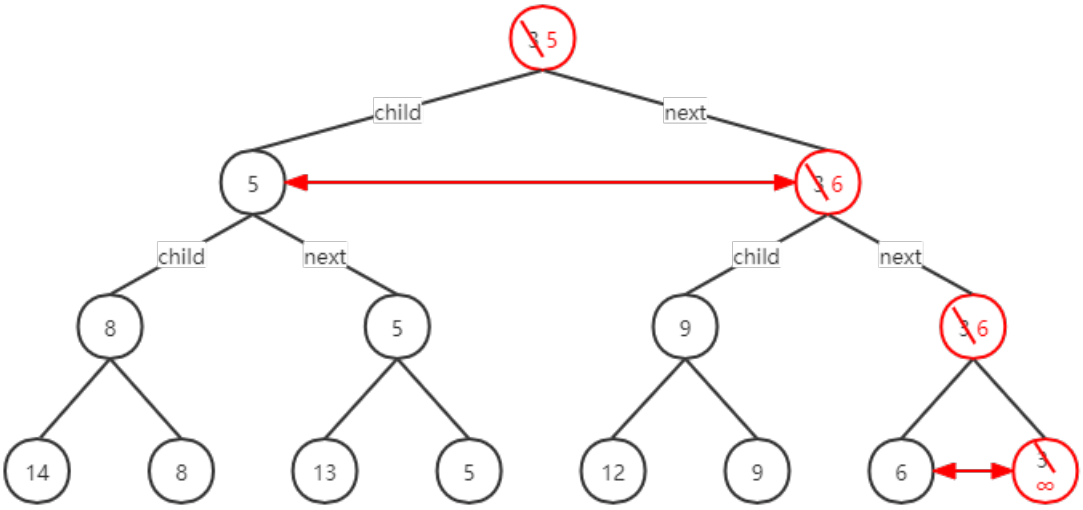

Fig. 10. Sifting

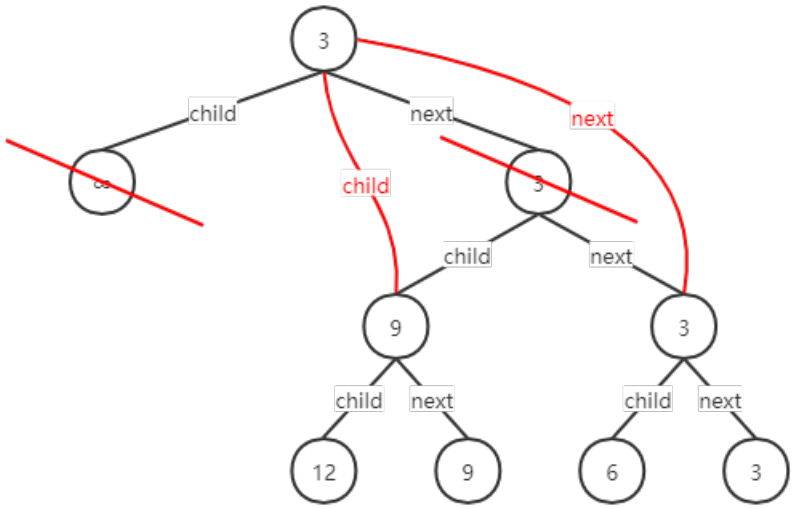

Fig. 11. Cleaning-up for one INF child