

Theory of 2-3 Heaps

Tadao Takaoka

Department of Computer Science, University of Canterbury
Christchurch, New Zealand
E-mail: tad@cosc.canterbury.ac.nz

Abstract. As an alternative to the Fibonacci heap, we design a new data structure called a 2-3 heap, which supports n insert, n delete-min, and m decrease-key operations in $O(m + n \log n)$ time. Our experiments show the 2-3 heap is more efficient. The new data structure will have a wide application in graph algorithms.

1 Introduction

Since Fredman and Tarjan [7] published Fibonacci heaps in 1987, there has not been an easy alternative that can support n insert, n delete-min, and m decrease-key operations in $O(m + n \log n)$ time. The relaxed heaps by Driscoll, et al. [6] have the same overall complexity with decrease-key in $O(1)$ worst case time, but are difficult to implement. Logarithm here is with base 2, unless otherwise specified. Two representative application areas for these operations will be the single source shortest path problem and the minimum cost spanning tree problem. Direct use of these operations in Dijkstra's [5] and Prim's [8] algorithms with a Fibonacci heap will solve these two problems in $O(m + n \log n)$ time. The Fibonacci heap is a generalization of a binomial queue invented by Vuillemin [9]. When the key value of a node v is decreased, the subtree rooted at v is removed and linked to another tree at the root level. If we perform this operation many times, the shape of a tree may become shallow, that is, the number of children from a node may become too many due to linkings without adjustment. If this happens at the root level, we will face a difficulty, when the node with the minimum is deleted and we need to find the next minimum. To prevent this situation, they allow loss of at most one child from any node for it to stay at the current position. If one more loss is required, it will cause what is called cascading cut. This tolerance bound will prevent the number of children of any node in the heap from getting more than $1.44 \log n$. The constant is the golden ratio derived from the Fibonacci sequence. Since this property will keep the number of children, which we call the degree, in some bound, let us call this "horizontal balancing".

In the area of search trees, there are two well-known balanced tree schemes; the AVL tree [1] and the 2-3 tree [2]. When we insert or delete items into or from a binary search tree, we may lose the height balance of the tree. To prevent this situation, we restore the balance by modifying the shape of the tree. As we control the path lengths, we can view this adjustment as "vertical balancing".

The AVL tree can maintain the tree height to be $1.44 \log n$ whereas the 2-3 tree will keep this to be $\log n$. As an alternative to the Fibonacci heap, we propose a new data structure called a 2-3 heap, the idea of which is borrowed from the 2-3 tree. It has a structure based on dimensions, which is more rigid than that of the Fibonacci heap. The degree of any node in the 2-3 heap is bounded by $\log n$, better than the Fibonacci heap by a constant factor. While the Fibonacci heap is based on binary linking, we base our 2-3 heap on ternary linking; we link three roots of three trees in increasing order according to the key values. We call this path of three nodes a trunk. We allow a trunk to shrink by one. If there is requirement of further shrink, we make adjustment by moving a few subtrees from nearby positions. This adjustment may propagate, prompting the need for amortized analysis. The word “potential” used in [7] is used in a reverse meaning. It counts the number of nodes that lost one child. This number reflects some deficiency of the tree, not potential. We use the word “potential” in amortized analysis to describe the real potential of a tree. We define the potential of a trunk with two nodes to be 1, and that of a trunk with three nodes to be 3. We define the potential of the 2-3 heap to be the sum of those potentials. Amortized time for one decrease-key or insert is shown to be $O(1)$, and that for delete-min to be $O(\log n)$.

The concept of r -ary linking is similar to the product of graphs, but different in connections. When we make the product $G \times H$ of graphs G and H , we substitute H for every vertex in G and connect corresponding vertices of H if an edge exists in G . See Bondy and Murty [4], for example, for the definition. In the product of trees, only corresponding roots are connected. The 2-3 heap is constructed by ternary linking of trees repeatedly, that is, repeating the process of making the product of a linear tree and a tree of lower degree, where the degree of a tree is the number of children of the root. This general description of r -ary trees is given in Section 2. The precise definition of 2-3 heaps is given in Section 3. The description of operations on 2-3 heaps is given in Section 4. Section 5 gives amortized analysis of those operations. In Section 6, we consider several problems in implementation, and also some practical considerations for further speed up. Section 7 concludes this paper. Note that our computational model is comparison-based. If we can use special properties of key values, there are efficient data structures, such as Radix-heaps [3].

2 Polynomial of trees

We define algebraic operations on trees. We deal with rooted trees in the following. A tree consists of nodes and branches, each branch connecting two nodes. The root of tree T is denoted by $root(T)$. A linear tree of size r is a linear list of r nodes such that its first element is regarded as the root and a branch exists from a node to the next. The linear tree of size r is expressed by bold face \mathbf{r} . Thus a single node is denoted by $\mathbf{1}$, which is an identity in our tree algebra. The empty tree is denoted by $\mathbf{0}$, which serves as the zero element. A product of two trees S and T , $P = ST$, is defined in such a way that every node of S is replaced

by T and every branch in S connecting two nodes u and v now connects the roots of the trees substituted for u and v in S . Note that $\mathbf{2} * \mathbf{2} \neq \mathbf{4}$, for example, and also that $ST \neq TS$ in general. The symbol “ $*$ ” is used to avoid ambiguity.

The number of children of node v is called the degree of v and denoted by $deg(v)$. The degree of tree T , $deg(T)$, is defined by $deg(root(T))$. The sum of two trees S and T , denoted by $S + T$, is just the collection of two trees S and T . A polynomial of trees is defined next. Since the operation of product is associative, we use the notation of \mathbf{r}^i for the products of i \mathbf{r} 's. Note that $deg(\mathbf{r}^i) = i$. An r -ary polynomial of trees of degree $k - 1$, P , is defined by

$$P = \mathbf{a}_{k-1}\mathbf{r}^{k-1} + \dots + \mathbf{a}_1\mathbf{r} + \mathbf{a}_0 \quad (1)$$

where \mathbf{a}_i is a linear tree of size a_i and called a coefficient in the polynomial. Let $|P|$ be the number of nodes in P and $|\mathbf{a}_i| = a_i$. Then we have $|P| = a_{k-1}r^{k-1} + \dots + a_1r + a_0$. We choose a_i to be $0 \leq a_i \leq r - 1$, so that n nodes can be expressed by the above polynomial of trees uniquely, as the k digit radix- r expression of n is unique with $k = \lceil \log_r(n + 1) \rceil$. The term $\mathbf{a}_i\mathbf{r}^i$ is called the i -th term. We call \mathbf{r}^i the complete tree of degree i . Let the operation “ \bullet ” be defined by the tree $L = S \bullet T$ for trees S and T . The tree L is made by linking S and T in such a way that $root(T)$ is connected as a child of $root(S)$. Then the product $\mathbf{r}^i = \mathbf{r}\mathbf{r}^{i-1}$ is expressed by

$$\mathbf{r}^i = \mathbf{r}^{i-1} \bullet \dots \bullet \mathbf{r}^{i-1} \quad (r - 1 \bullet\text{'s are evaluated right to left}) \quad (2)$$

The whole operation in (2) is to link r trees, called an i -th r -ary linking. The path of length $r - 1$ created by the r -ary linking is called the i -th trunk of the tree \mathbf{r}^i , which defines the i -th dimension of the tree in a geometrical sense. The j -th \mathbf{r}^{i-1} in (2) is called the j -th subtree on the trunk. The path created by linking a_i trees of \mathbf{r}^i in form (1) is called the main trunk of the tree corresponding to this term. A polynomial of trees is regarded as a collection of trees of distinct degrees connected by their main trunks. We next define a polynomial queue. An r -nomial queue is an r -ary polynomial of trees with a label $label(v)$ attached to each node v such that if u is a parent of v , $label(u) \leq label(v)$. A binomial queue is a 2-nomial queue.

Example 1. A polynomial queue with an underlying polynomial of trees $P = \mathbf{2} * \mathbf{3}^2 + \mathbf{2} * \mathbf{3} + \mathbf{2}$ is given in Fig.1.

Each term $\mathbf{a}_i\mathbf{r}^i$ in form (1) is a tree of degree $i + 1$ if $a_i > 1$. One additional degree is caused by the coefficient. The merging of two linear trees \mathbf{r} and \mathbf{s} is to merge the two lists by their labels. The result is denoted by the sum $\mathbf{r} + \mathbf{s}$. The merging of two terms $\mathbf{a}_i\mathbf{r}^i$ and $\mathbf{a}'_i\mathbf{r}^i$ is to merge the main trunks of the two trees by their labels. When the roots are merged, the trees underneath are moved accordingly. If $a_i + a'_i < r$, we have the merged tree with coefficient $\mathbf{a}_i + \mathbf{a}'_i$. Otherwise we have a carry tree \mathbf{r}^{i+1} and the remaining tree with the main trunk of length $a_i + a'_i - r$. The sum of two polynomial queues P and Q is made by merging two polynomial queues in a very similar way to the addition of two radix- r numbers. We start from the 0-th term. Two i -th terms from both queues

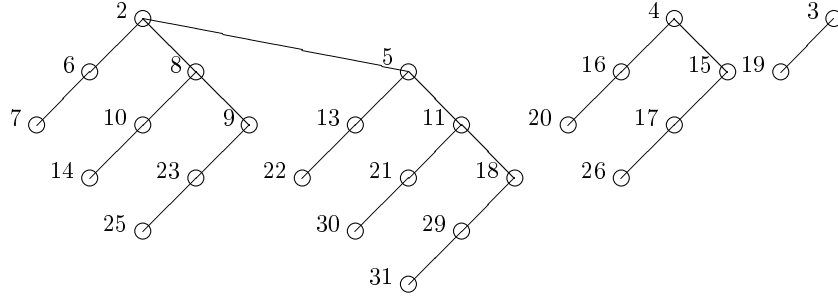


Fig. 1. A polynomial queue with $r = 3$

are merged, causing a possible carry to the $(i + 1)$ -th terms. Then we proceed to the $(i + 1)$ -th terms with the possible carry.

An insertion of a key into a polynomial queue is to merge a single node with the label of the key into the 0-th term, taking $O(r \log_r n)$ time for possible propagation of carries to higher terms. Thus n insertions will form a polynomial queue P in $O(nr \log_r n)$ time. The value of k in form (1) is $O(\log_r n)$ when we have n nodes in the polynomial of trees. We can take n successive minima from the queue by deleting the minimum in some tree T , adding the resulting polynomial queue Q to $P - T$, and repeating this process. This will sort the n numbers in $O(nr \log_r n)$ time after the queue is made. Thus the total time for sorting is $O(nr \log_r n)$. In the sorting process, we do not change key values. If the labels are updated frequently, however, this structure of polynomial queue is not flexible, prompting the need for a more flexible structure in the next section.

3 2-3 heaps

We linked r trees in form (2). We relax this condition in such a way that the number of trees linked is from l to r . Specifically an (l, r) -tree $T(i)$ of degree i is formed recursively by

$$T(0) = \text{a single node}$$

$$T(i) = T_1(i-1) \bullet \dots \bullet T_m(i-1) \quad (s \text{ is between } l \text{ and } r) \quad (3)$$

Note that s varies for different linkings. The subscripts of $T(i)$ are given to indicate different trees of degree $i - 1$ are used. Note that the shape of an (l, r) -tree $T(i)$ is not unique for n such that $|T(i)| = n$. We say $root(T_1(i-1))$ is the head node of this trunk. The dimension of non-head nodes is $i - 1$ and that of the head node is i . We omit a subscript for $T(i)$; $T(i)$ sits for an (l, r) -tree of degree i . In this context, we sometimes refer to $T(i)$ as a type of tree. Then an extended polynomial of trees, P , is defined by

$$P = \mathbf{a}_{k-1}T(k-1) + \dots + \mathbf{a}_1T(1) + \mathbf{a}_0 \quad (4)$$

The main trunk and the i -th trunk in each term of (4) are similarly defined to those for (1). So is the j -th subtree in the trunk. Let us assign labels with nodes

in the tree in a similar way to the last section. That is, when the s -ary linking is made, the roots are connected in non-decreasing order of labels. Then the resulting polynomial is called an (l, r) -heap. When $l = 2$ and $r = 3$, we call it a 2-3 heap. We refer to the trees in (4) and their roots as those at the top level, as we often deal with subtrees at lower levels and need to distinguish them from the top level trees. The sum $P + Q$ of the two 2-3 heaps P and Q is defined similarly to that for polynomial queues. Note that those sum operations involve computational process based on merging.

Example 2. $P = 2T(2) + 2T(1) + 2T(0)$.

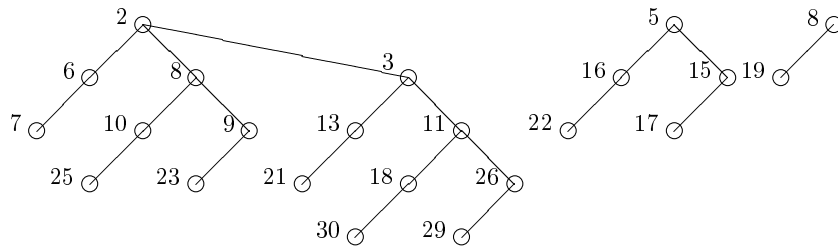


Fig. 2. A 2-3 heap

Lemma 1. *If we maintain n nodes in a 2-3 heap in form (4), the upper bound for k is given when $2^{k-1} = n$.*

From this we see $k \leq \lceil \log(n + 1) \rceil$. We informally describe delete-min, insertion, and decrease-key operations for a 2-3 heap. Precise definitions will be given in the next section. Let the dimension of node v be $i - 1$, that is, the trunk of the highest dimension on which node v stands is the i -th and let v be not on a main trunk. We define the work space for v to be a collection of nodes on the i -th trunk on which v stands, the $(i + 1)$ -th trunk of the head node of v , and the other i -th trunks whose head nodes are on the $(i + 1)$ -th trunk. The work space has 4 to 9 nodes. Let the head node of the work space be the node at the first position of the $(i + 1)$ -th trunk. We also define $tree(u)$ for u in the above defined work space to be the tree of type $T(i - 1)$ rooted at u . In this paper the words “key” and “label” are used interchangeably.

Example 3. Let us denote the node with label x by $node(x)$ in Example 2. The dimensions of $node(6)$ and $node(7)$ are 0. That of $node(8)$ is 1 and that of $node(3)$ is 2. Under $node(2)$, we have three trunks of dimension 1, 2, and 3. The work space for $node(18)$ is $\{node(3), node(13), node(21), node(11), node(18), node(30), node(26), node(29)\}$. The head node of this work space is $node(3)$. The work space for $node(9)$ is in higher dimensions and given by $\{node(2), node(8), node(9), node(3), node(11), node(26)\}$. The head node is $node(2)$.

Delete-min: Find the minimum by scanning the roots of the trees. Let $T(i)$ have the minimum at the root and Q be the polynomial resulting from $T(i)$ by removing the root. Then merge $P - T(i)$ and Q , i.e., $(P - T(i)) + Q$.

Insertion: Perform $T + v$, where we insert v with its key into the 2-3 tree T . Here v is regarded as a 2-3 tree with only a term of type $T(0)$.

Removal of a tree: We do not remove $tree(v)$ for a node v if it is a root at the top level. Suppose we remove $tree(v)$ of type $T(i - 1)$ for a node v . Consider the two cases where the size of the work space is greater than 4, and it is equal to 4. We start with the first case. Let the i -th trunk of node v be (u, v, w) or (u, w, v) . Then remove $tree(v)$ and shrink the trunk. If the trunk is (u, v) , remove $tree(v)$. And we would lose this trunk. To prevent this loss, we move a few trees of type $T(i - 1)$ within the work space.

Let the size of the work space be 4. Remove $tree(v)$ and rearrange the remaining three nodes in such a way that two will come under the head node of the work space to form the i -th trunk. Then we recover the i -th trunk to be of length 2, that is, of three nodes, but lose the $(i + 1)$ -th trunk. Our work proceeds to a higher dimension, if it exists. Otherwise stop. Note that during the above modifications of the heap, we need to compare key values of nodes to find out the correct positions for placing trees.

Decrease-key: Suppose the key of node v has been decreased. If v is not at the top level, remove $tree(v)$, and insert it to the j -th term at the top level with the new key, where j is the dimension of v . If v is at the top level, we do nothing after decreasing the key.

Example 4. In Example 2, we name the node with label x by $node(x)$. Suppose we decrease labels 6 to 4, and 29 to 14. Then we have the following $T = 1T(3) + 1T(0)$.

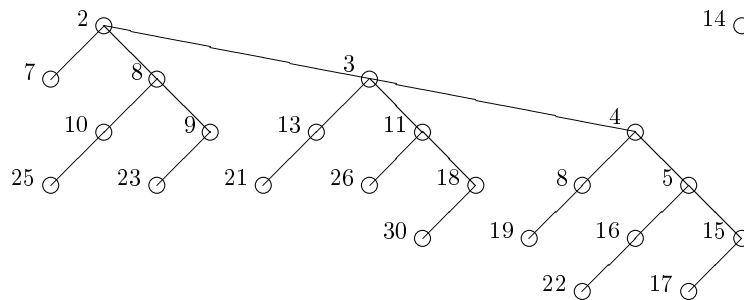


Fig. 3. The 2-3 heap after two decrease-key operations

We first remove $node(6)$, causing the move of $node(7)$ to the child position of $node(2)$. The new node $node(4)$ is inserted into $2T(0)$, resulting in $T(1)$ with $node(8)$ and $node(19)$ and carrying over to $2T(1)$ to cause insertion. Then the newly formed $T(2)$ will be carried to $2T(2)$, resulting in $T(3)$. For the second

decrease-key, we have a new node $node(14)$. Since the trunk can not shrink further, the two links from $node(11)$ to $node(18)$ and $node(26)$ are swapped.

4 Detailed description of operations

Decrease-key. We first describe the decrease-key operation in detail. Suppose we perform decrease-key on node v of dimension $i - 1$. After decreasing the key value of v , we perform removal of $tree(v)$, and then insertion of $tree(v)$ at the top level. For removal of a tree, let us classify the situation using parts of the tree structure. In the following figures, only work space is shown. Each node can be regarded as a tree of the same degree for general considerations. The left-hand side and the right-hand side are before and after conversion. The trunks going left-down are trunks of i -th dimension (i -th trunk for short), on one of which v stands, and that going right-down is a trunk of $(i+1)$ -th dimension. We have two or three trunks of i -th dimension in the following figures. By removing a node and shrinking a trunk, we create a vacant position, which may or may not be adjusted. By checking those trunks, we classify the situation into several cases depending on the size w of the work space. In the following figures the i -th trunks are arranged in non-decreasing order of lengths for simplicity of explanation. We call this standard arrangement. Other cases are similar. The potential of a work space is the sum of the potentials of all trunks in the work space.

Case 1. $w = 9$. The removal of any of the six black nodes will bring the same new shape within standard arrangement. We decrease the potential of the work space from 12 to 10, that is, by two, and spend no comparison. See Fig. 4.

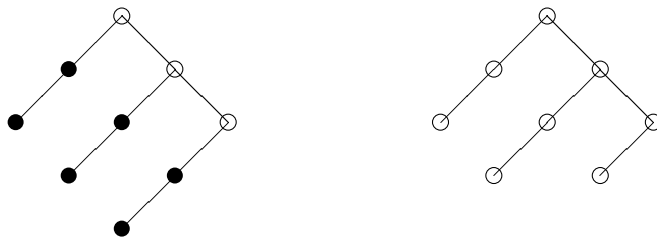


Fig. 4. The case of $w = 9$

Case 2. $w = 8$. The first case of $w = 8$ is similar to case 1. We spend no comparison and decrease the potential by two. See Fig. 5.

Case 3. $w = 8$. A node on a trunk with potential one is removed. In this case, we can rearrange the heap within the work space with no comparison and decrease the potential by two. To visualize the process, labels a and b are provided in the figures. In some of the following cases, similar rearrangements are done. See Fig. 6.

Case 4. $w = 7$. We decrease the potential by two and spend no comparison. See Fig. 7.



Fig. 5. A case of $w = 8$

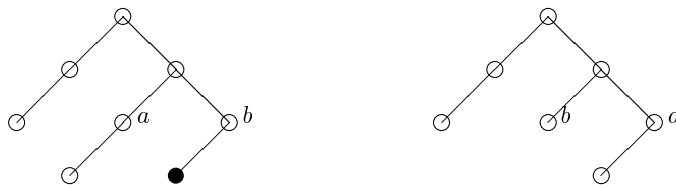


Fig. 6. The other case of $w = 8$

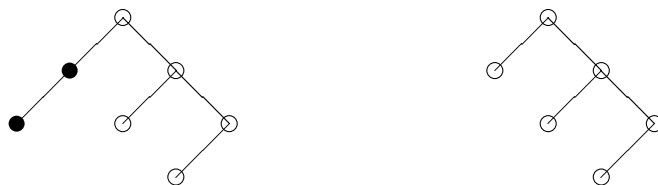


Fig. 7. A case of $w = 7$

Case 5. $w = 7$ We decrease the potential by one and spend at most one comparison. See Fig. 8.

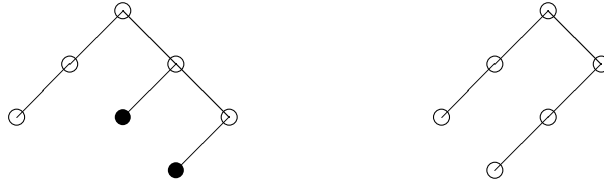


Fig. 8. The other case of $w = 7$

Case 6. $w = 6$. We spend at most one comparison and decrease the potential by one. See Fig. 9.



Fig. 9. A case of $w = 6$

Case 7. $w = 6$. We decrease the potential by two and spend no comparison. See Fig. 10.



Fig. 10. The other case of $w = 6$

Case 8. $w = 5$. We decrease the potential by two, and spend no comparison. See Fig. 11.

Case 9. $w = 4$. All the trunks of the work space defined by the i -th and $(i+1)$ -th dimensions have potential one. We make a trunk of the i -th with potential 3 for the head node and the situation becomes the loss of a node at the $(i+1)$ -th trunk. Make-up will be made in the work space defined by the $(i+1)$ -th trunk and the $(i+2)$ -th trunk. The make-up process may repeat several times, and stops in one of cases 1-8, or if no higher trunk exists. The $(i+2)$ -th trunk is



Fig. 11. Another case of $w = 5$

drawn by a dotted line, and the by-gone $(i + 1)$ -th trunk is drawn by a broken line at the right-hand side. Let us leave the removed node at the end of the broken line for the accounting purposes. We increase the potential by one and spend at most one comparison. See Fig. 12.



Fig. 12. The case of $w = 4$

Example 5. In Fig.13, the removed node is given by a black circle in the first picture. We have two cases of $w = 4$ followed by a case of $w = 6$, resulting in the second picture.

Next we describe top level insertions. Suppose we insert a tree of type $T(i)$ into the term $\mathbf{a}_i T(i)$ at the top level. We have three cases.

Case A. $\mathbf{a}_i = \mathbf{0}$. We simply put the tree in the correct position.

Case B. $\mathbf{a}_i = \mathbf{1}$. We can form a new $\mathbf{2}T(i)$ with one comparison, and increase the potential by one.

Case C. $\mathbf{a}_i = \mathbf{2}$. We make a carry of $T(i + 1)$ with two comparisons and increase the potential by two. Then proceed to the insertion at $\mathbf{a}_{i+1} T(i + 1)$.

Insert. Insertion is covered by the above three cases of A, B, and C with insertion of type $T(0)$ tree.

Delete-min. Delete-min needs some description. After we delete the root of tree $\mathbf{a}_i T(i)$, which has the minimum key, we have the tree broken apart into trees $\mathbf{b}_0 T(0)$, ..., $\mathbf{b}_i T(i)$, where each \mathbf{b}_j is $\mathbf{1}$ or $\mathbf{2}$ for $j = 0, \dots, i - 1$, and $\mathbf{b}_i = \mathbf{1}$ if $\mathbf{a}_i = \mathbf{2}$, and $\mathbf{b}_i = \mathbf{0}$ if $\mathbf{a}_i = \mathbf{1}$. We merge these trees into the remaining trees in the heap for $j = 0, \dots, i$. This merging process is very similar to addition of two ternary numbers. In general we merge $\mathbf{a}_j T(j)$, $\mathbf{b}_j T(j)$, and $\mathbf{c}_j T(j)$, where $\mathbf{c}_j T(j)$ is the carry from the $(j - 1)$ -th position. If $\mathbf{c}_j = \mathbf{1}$, there is a carry, and if $\mathbf{c}_j = \mathbf{0}$, there is no carry. There are 18 combinations of $(\mathbf{a}_j, \mathbf{b}_j, \mathbf{c}_j)$. Excluding symmetry, we consider the following cases.

Case $(\mathbf{1}, \mathbf{0}, \mathbf{0})$. Covered by Case A above.

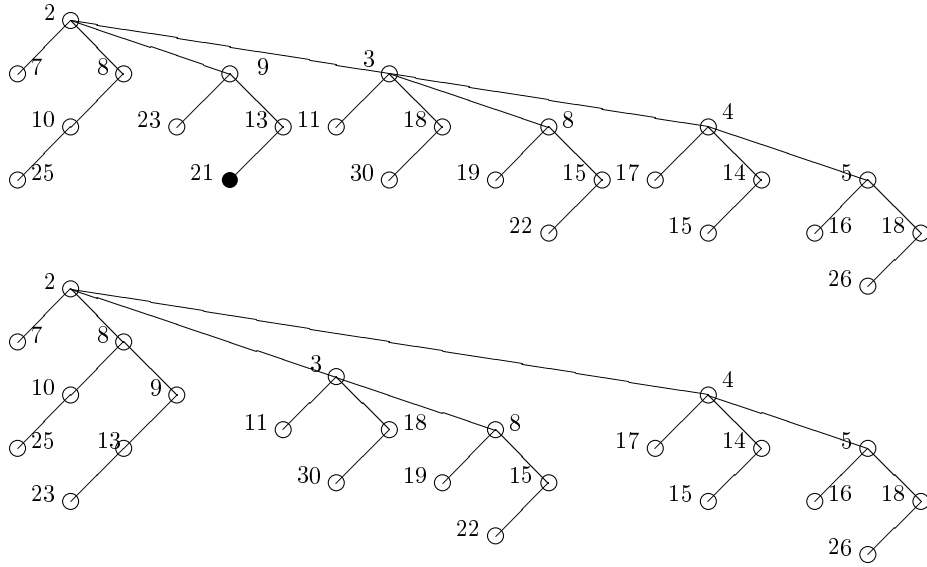


Fig. 13. The change of a 2-3 heap with case 4

Case $(1, 1, 0)$. Covered by Case B above.

Case $(1, 1, 1)$. By spending three comparisons, we can make a carry of $T(j+1)$ with the increase of 3 in the potential.

Case $(2, 1, 0)$. Covered by Case C above.

Case $(2, 1, 1)$. Covered by Case C above.

Case $(2, 2, 1)$. Covered by Case C above.

Case $(2, 2, 0)$. By comparing the keys of the two roots and putting the smaller root on top of the other, we can make a tree of type $T(j+1)$, which is a carry to the next position. We spent one comparison and increased the potential by one.

5 Analysis

We analyze the computing time by the number of comparisons between key values, based on amortized analysis. Other times such as pointer manipulations and search in the work space are proportional to it, or absorbed into a term proportional to $m + n \log n$. Define the potential of a 2-3 heap by the sum of potentials of all trunks. The potential of the empty 2-3 heap is defined to be zero. We regard the potential as the saving of comparisons. The actual cost of an operation is the number of comparisons performed. The amortized cost of an operation is defined to be the actual cost minus the gain in potential. Suppose we perform n insert, n delete-min, and m decrease-key operations. The actual cost and amortized cost for the i -th operation are denoted by t_i and a_i , where $a_i = t_i - (\Phi_i - \Phi_{i-1})$, and Φ_i is the potential after the i -th operation. Let the total number of operations be N . Since $\sum a_i = \sum t_i - \Phi_N + \Phi_0$, and the potential

is assumed to be zero at the beginning and end, the total actual cost is equal to the total amortized cost.

Amortized cost for delete-min: It takes at most $\lceil \log(n+1) \rceil$ comparisons to find the minimum. After that we break apart the subtrees under the root with the minimum. This causes loss of potential by at most $2\lceil \log(n+1) \rceil$. The merging process for those subtrees at the top level takes $O(\log n)$ actual time. The amortized cost of this process is 0. Thus one delete-min operation's amortized cost is bounded by $3\lceil \log(n+1) \rceil$. The actual time is $O(\log n)$.

Amortized cost for decrease-key: It takes $O(1)$ time for decreasing the value of the key. After that, we perform various operations described above, the costs of all of which are bounded by 2. The amortized costs for case 9 and cases A,B, and C above are 0. Thus the amortized cost for this part is 2.

Amortized cost for insertion: We insert a new node to the 0-th term of the top level. Thus the amortized cost is 0, that is, $O(1)$. The actual time is $O(\log n)$.

If we perform n insert, n delete-min, and m decrease-key operations, the total time is now bounded by $O(m + n \log n)$. In terms of the number of comparisons, we have $2m + 3n \log n$ through the amortized analysis described above. Thus we can solve the single source shortest path problem and the minimum cost spanning tree problem in $O(m + n \log n)$ time in a similar setting to [7].

6 Practical considerations

For the data structure of a 2-3 heap, we need to implement it by pointers. For the top level trees, we prepare an array of pointers of size $k = \lceil \log(n+1) \rceil$, which we call the top level array. The i -th element of the array points to the root of the tree of the i -th term for $i = 0, \dots, k-1$, if the term exists. Otherwise it is *nil*. Let a node v 's highest trunk be the i -th. The data structure for node v consists of integer variables *key* and $dim = i$, a pointer to the head node of the i -th trunk, and an array of size k , whose elements are pairs (*second*, *third*). We call this array the node array. The *second* of the j -th element of the array points to the root of the second node on the j -th trunk of v . The *third* is to the third node. While *second* is not *nil*, *third* can be *nil*, in which case there is no third node on this trunk. With this data structure, we can explore the work space for v . Suppose we perform decrease-key on node v of dimension i . By the pointer to the head node, we go to the head node. By exploring the i -th and $(i+1)$ -th trunks with the aid of the array, we can know which case the work space falls into. If we prepare the fixed size k for the arrays of all the nodes, we would need $O(n \log n)$ space.

We can implement our data structure with $O(n)$ space, as is done in the Fibonacci heap, although this version will be less time-efficient. We use the same top level array. A node is slightly different. Specifically a node v of dimension i consists of variables similar to the above version except for the node array. It is now replaced by a pointer to the second node on the trunk of the highest dimension i . The second node and the third node, if any, are pointing to each other. The second nodes have parent pointers to v . From each third node, we

can go to its head node indirectly through the second node. The second nodes are formed by a doubly linked circular structure, which ensures $O(1)$ time for insertion and deletion, and also for going from the j -th trunk to the $(j + 1)$ -th trunk for $1 \leq j \leq i - 1$ in $O(1)$ time. A general picture is given in Fig. 14, in which the symbols “*” are connected. We actually implemented this version to compare with the Fibonacci heap.

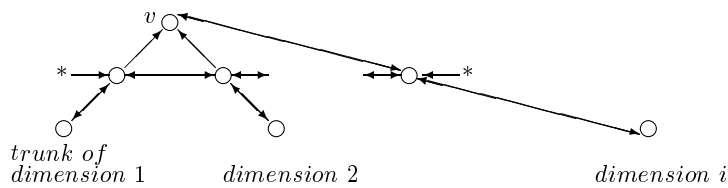


Fig. 14. The 2-3 heap after two decrease-key operations

7 Concluding remarks

Measuring the complexity of n delete-min's, n inserts's, and m decrease-key's by the number of comparisons, we showed it to be $O(m + n \log n)$ in a 2-3 heap. Our analysis shows the number of comparisons is bounded by $2m + 3n \log n$. To compare the 2-3 tree and the Fibonacci heap, we can modify the latter in the same framework of the former, that is, restricting operations to delete-min, insert, and decrease-key. In the original paper [7], the number of trees in the heap can be up to n . When the delete-min operation is performed, trees of the same rank (degree in this paper) are joined together by comparing the roots. In this process, the minimum is also found. To modify the Fibonacci heap, we can restrict the number of roots to $1.44 \log n$, and to find the minimum, we can scan through the roots. In this version, we can show that the number of comparisons is bounded approximately by $2m + 2.88n \log n$ through amortized analysis, where we charge one unit of cost for a marked node. In [7], they charge 2 units for a marked node to allow for the cost of cascade cut. This would bring the term $3m$ in stead of $2m$ in the above complexity. After all, the Fibonacci heap and the 2-3 heap are almost on a par in the worst case analysis of the number of comparisons. Our actual implementation of both of those heaps for Dijkstra's algorithm for a certain class of random graphs, however, shows the 2-3 heap is better than the Fibonacci heap by about 20% in both the number of comparisons and CPU time. This is because we have several cases where the amortized cost for decrease-key is 0 or 1, not 2. Also at delete-min, we decrease potentials of the trunks under the minimum node, but the loss is 1 or 2, not always 2, on each of those trunks.

We note that we can support all operations for Fibonacci heaps with 2-3 heaps with the same asymptotic complexities. We focused on the above mentioned operations in this paper.

The method for modifying the heap when a node is removed is not unique. The method in Section 4 is designed based on adjustment with neighbors. To say this is the best method, we will need further study. We can define similar heaps, such as 2-4 heaps, 3-4 heaps, etc., and their operations. It is an open question whether the performance of those heaps is better than 2-3 heaps.

acknowledgement. The author expresses thanks to the referees whose constructive comments improved the technical quality of the paper to a great extent. He is also grateful to Shane Saunders who implemented the 2-3 heap and did extensive experiments on Dijkstra's algorithm with the heap.

References

1. Adel'son-Vel'skii, G.M, and Y.M. Landis, An algorithm for the organization of information, Soviet Math. Dokl. 3 (1962) 1259-1262.
2. Aho, A.V., J.E. Hopcroft, and J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley (1974).
3. Ahuja, K., K. Melhorn, J.B. Orlin, and R.E. Tarjan, Faster algorithms for the shortest path problem, Jour. ACM, 37 (1990) 213-223.
4. Bondy, J.A. and U.S.R. Murty, Graph Theory with Applications, Macmillan Press (1976).
5. Dijkstra, E.W., A note on two problems in connexion with graphs, Numer. Math. 1 (1959) 269-271.
6. Driscoll, J.R., H.N. Gabow, R. Shrairman, and R.E. Tarjan, An alternative to Fibonacci heaps with application to parallel computation, Comm. ACM, 31(11) (1988) 1343-1345.
7. Fredman, M.L. and R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, Jour. ACM 34 (1987) 596-615
8. Prim, R.C., Shortest connection networks and some generalizations, Bell Sys. Tech. Jour. 36 (1957) 1389-1401.
9. Vuillemin, J., A data structure for manipulating priority queues, Comm. ACM 21 (1978) 309-314.