Theory of Trinomial Heaps

Tadao Takaoka

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

Abstract. We design a new data structure, called a trinomial heap, which supports a decrease-key in $O(1)$ time, and an insert operation and delete-min operation in $O(\log n)$ time, both in the worst case, where n is the size of the heap. The merit of the trinomial heap is that it is conceptually simpler and easier to implement than the previously invented relaxed heap. The relaxed heap is based on binary linking, while the trinomial heap is based on ternary linking.

1 Introduction

The Fibonacci heap was invented by Fredman and Tarjan [3] in 1987. Since then, there have been two alternatives that can support n insert and delete-min operations, and m decrease-key operations in $O(m + n \log n)$ time. The relaxed heaps by Driscoll, et. al [2] have the same overall complexity with decrease-key with $O(1)$ worst case time, but are difficult to implement. The other alternative is the 2-3 heap invented by the author [5], which supports the same set of operations with the same time complexity. Although the 2-3 heap is simpler and slightly more efficient than the Fibonacci heap, the $O(1)$ time for decrease-key is in the amortized sense, meaning that the time from one decrease key to the next can not be smooth. 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 [1] and Prim's [4] algorithms with those data structures will solve these two problems in $O(m + n \log n)$ time, where n and m are the numbers of vertices and edges of the given graph. Logarithm here is with base 2, unless otherwise specified.

A Fibonacci heap is a generalization of a binomial queue invented by Vuillemin [6]. 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 in the Fibonacci and 2-3 heap. This removal of a subtree may cause a chain effect to keep structural properties of those heaps, resulting in worst case time greater than $O(1)$, although it is $O(1)$ amortized time. As an alternative to the Fibonacci heap, Driscoll, et. al. proposed a data structure called a relaxed heap, whose shape is the same as that of a binomial queue. The requirement of heap order is relaxed in the relaxed heap; a certain number of nodes are allowed to have smaller key values than those of their parents. Those nodes are called bad children in [2] and inconsistent nodes in this paper. On the other hand, the 2-3 heaps is proposed as

another alternative to the Fibonacci heap. While the Fibonacci heap is based on binary linking, 2-3 heaps are based 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, taking time more than $O(1)$.

In this paper, we combine the ideas of the relaxed heap and 2-3 heap; we use ternary linking and allow a certain number of inconsistent nodes. Ternary linking gives us more flexibility to keep the number of inconsistent nodes under control. The new data structure is called a trinomial heap, and is simpler and easier to implement than the relaxed heap. In the relaxed heap which is based on binary linking, we must keep each bad child at the rightmost branch of its parent, causing difficult book-keeping. The trinomial 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 dimension. This general description of r-ary trees is given in Section 2. The definition of trinomial heaps and their operations are given in Section 3. In Section 4, we implement decrease-key in $O(1)$ amortized time. In Section 5, we implement it with $O(1)$ worst case time. In Section 6, we give several practical considerations for implementation. Section 7 concludes the paper.

In this paper we analyze the number of key comarisons for computing time as the times for other operations are proportional to it. We mainly deal with decrease-key and delete-min in this paper since the main application areas are the single source shortest path problem and the minimum cost spanning tree problem and we can build up the heap of size n at the beginning. The nodes not adjacent with the source can be inserted with infinite key values at the beginning. As shown in Section 2, the time for building the heap can be $O(n)$. Thus we can concentrate our discussion on decrease-key and delete-min after the heap is built.

2 Polynomial queues and their analysis

We borrow some materials from [5] in this section, as the trinomal heap and the 2-3 heap share the same basic structure. We define algebraic operations on rooted trees as the basis for priority queues. 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 liner 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. Since the operation of product is associative, we use the notation of $\mathbf{r}^i$ for the products of i $\mathbf{r}$'s.

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

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

The whole operation in (1) 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 (1) is called the j-th subtree on the trunk counting from $j = 0$. The root of such a j-th subtree is said to be of dimension i. The root of the 0-th subtree is called the head node of the trunk. The dimension of the head node is i or higher, depending on further linking. If v is of dimension i, we write as $dim(v) = i$.

The dimension of tree T, $dim(T)$, is defined by $dim(root(T))$. A sum of two trees S and T, denoted by $S+T$, is just the collection of two trees S and T. Let $\mathbf{a}_i\mathbf{r}^i$ be defined similarly to (1) by linking a_i trees of $\mathbf{r}^i$. An r-ary polynomial of trees of degree $k-1$, P, is defined by

$$P = \mathbf{a}_{k-1}\mathbf{r}^{k-1} + ... + \mathbf{a}_1\mathbf{r} + \mathbf{a}_0 \text{ } (2)$$

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 $|P| = a_{k-1}r^{k-1} + ... + a_1 r + a_0$. 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 dimension i. The path created by linking a_i trees of $\mathbf{r}^i$ is called the main trunk of the tree corresponding to this term. Each term $\mathbf{a}_i\mathbf{r}^i$ in form (2) is a tree of dimension $i+1$ if $a_i > 1$, and i if $a_i = 1$. A polynomial of trees is regarded as a collection of trees of dimensions up to k.

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. We use "label" and "key" interchangeably.

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.

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

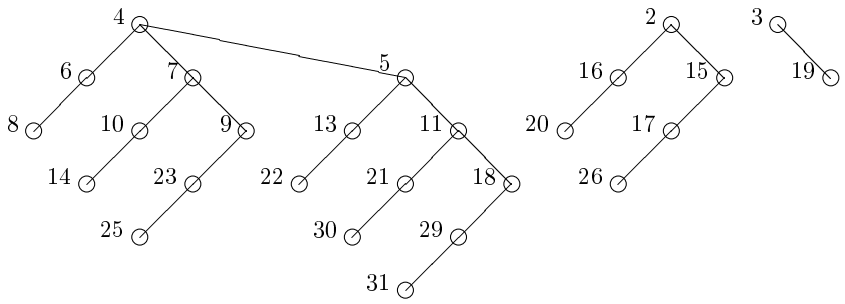

Fig. 1. Polynomial of trees with $r = 3$

the 0-th term. Two i-th terms from both queues 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. The time for n insertions to form a polynomial queue P looks like taking $O(nr \log_r n)$ time. Actually the time is $O(rn)$, as shown below. After the heap is made, 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.

Amortized analysis for insert: Let the deficit function Φ_i after the i-th insert be defined by $r-1$ times the number of trees in the heap. We use the word deficit rather than potential because the number of roots gives a negative aspect of the heap. Let t_i and a_i be the actual time and the amortized time for the i-th insert operation. Then we have $a_i = t_i + \Phi_i - \Phi_{i-1}$. When we insert a single node into the heap, we spend r-1 comparisons by scanning the main trunk, whenever we make a carry to the next position. We decrease the number of trees by one as a result. We terminate the carry propagation either by making a new complete tree or inserting a carry into a main trunk. Thus we have $a_i \leq r - 1$. Note that $\Sigma a_i = \Sigma t_i + \Phi_n - \Phi_0$, $\Phi_0 = 0$, and $\Phi_n = O(\log n)$.

3 Trinomial heaps

We linked r trees in heap order in form (1). We relax this condition in the following way. A node is said to be inconsistent if its key value is smaller than that of its head node. Otherwise the node is said to be consistent. An active node is either an inconsistent node or a node which was once inconsistent. The latter case occurs when a node becomes inconsistent and then the key value of its head node is decreased, and the node becomes consistent again. Since it is

expensive to check whether the descendants of a node become consistent when the key value of the node is decreased, we keep the children intact. Note that we need to look at the roots and active nodes to find the minimum key in the heap. If the number of active nodes in the given r-nomial queue is bounded by t, the queue is said to be an r-nomial heap with tolerance t. We further require that the nodes except for the head node on each trunk are sorted in non-decreasing order of their key values, and that there are no active nodes in each main trunk, that is, main trunks are sorted. Note that an r-nomial queue in the previous section is an r-nomial heap with tolerance 0.

When $r = 3$, we call it a trinomial heap. In a trinomial heap we refer to the three nodes on a trunk of dimension i as the head node, the first child and the second child. Note that each node of dimension $i(> 0)$ is connected downwards with first children of dimension 1, ..., $i - 1$. It is further connected downward and/or upward with trees of dimension i or higher. We refer to the trees in (2) 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 trinomial heaps P and Q is defined similarly to that for polynomial queues. Note that those sum operations involve computational process based on merging. We set $t = \lceil \log_3(n + 1) \rceil - 1$ as the default tolerance.

Example 2. A trinomial heap with tolerance 3 with an underlying polynomial of trees $P = \mathbf{2} * \mathbf{3}^2 + \mathbf{2} * \mathbf{3} + \mathbf{2}$ is given below. Active nodes are shown by black circles. We use $t = 3$ just for explanation purposes, although the default is 2.

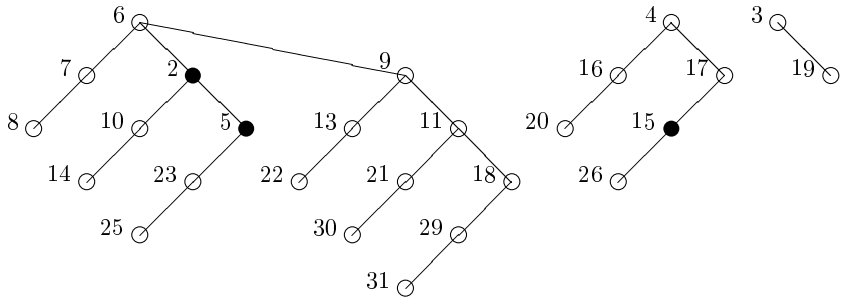

Fig. 2. Trinomial heap with $t = 3$

We describe delete-min, insertion, and decrease-key operations for a trinomial heap. If we make active nodes non-active by spending key comparisons, we say we clean them up.

Let the dimension of node v be i, that is, the trunk of the highest dimension on which node v stands is the i-th. Let $tree_i(v)$ be the complete tree $\mathbf{3}^i$ rooted at v. To prepare for delete-min, we first define the break-up operation. A similar break-up operation for a binomial queue is given in [6]

Break-up: The break-up operation caused by deleting node v is defined in the

following way. Let v_p, v_{p-1}, ..., v_1 be the head nodes on the path from the root ($= v_p$) to the head node ($= v_1$) of $v = (v_0)$ in the tree $\mathbf{a}_i\mathbf{3}^i$ in which v exists. We cut off all subtrees $tree_{j_l}(v_l)$ where $j_l = dim(v_{l-1})$ and the corresponding links to their first children of $dim(v_{l-1}) + 1$, ..., $dim(v_l)$ for $l = p, p-1, ..., 1$ in this order. Then we delete v, and cut the links to all the first children of v. The trunks from which subtrees are removed are shortened. This operation will create i trees of the form $\mathbf{2} * \mathbf{3}^j$ for $j = 0, ..., i-1$, and $\mathbf{3}^i$ if $\mathbf{a}_i = \mathbf{2}$. See Example 3.

Delete-min: Perform break-up after deleting the node with the minimum key. Swap the two subtrees on the main trunk of each resulting tree to have sorted order if necessary. Then merge the trees with the remaining trees at the top level. The time taken is obviously $O(\log n)$. The nodes on the main trunks of the new trees are all cleaned up, and thus the number of active nodes is decreased accordingly.

Reordering: When we move a node v on a trunk of dimension i in the following, we mean we move $tree_{i-1}(v)$. Let v and w be the first and second children and active nodes on the same trunk of dimension i. Then reordering is to reorder the positions of u, v and w after comparing $label(u)$ with $label(w)$. If $label(u) \leq label(w)$, we make w non-active, and do no more. Otherwise we move u to the bottom. When we move u, we move $tree_{i-1}(u)$, although $dim(u)$ may be higher. See Fig. 3. We can decrease the number of active nodes. Note that this operation may cause an effect equivalent to decrease-key at dimension $i+1$.

Rearrangement: Let v and w be active first children of dimension i on different trunks. Suppose v' and w' are the non-active second children on the same trunks. Now we compare keys of v' and w', and v and w, spending two comparisons. Without loss of generality assume $label(v') \leq label(w')$ and $label(v) \geq label(w)$. Then arrange v' and w', and w and v respectively to be the first and second children on each trunk. We call this operation rearrangement. Note that this operation is done within the same dimension of v. See Fig. 4. Since w and v are active on the same trunk, we call reordering described above.

Decrease-key: Suppose we decreased the key value of node v such that $dim(v) = i$. Let its head node and the other child on the same trunk be u and w. If v is the second child and $label(w) \leq label(v)$, do nothing. Otherwise swap w and v and go to the next step. If v is a first child and $label(u) \leq label(v)$, clean v if v was active. We have spent one or two comparisons. If we did not increase active nodes, we can finish decrease-key. Otherwise go to the next step. Assume at this stage v is the first child, $label(u) > label(v)$, and v has been turned active. We perform rearrangement and/or reordering a certain number of times to keep the number of active nodes under control. There are two control mechanisms as shown later.

Insert: To insert one node, merge a tree of a single node with the right most tree of the heap. A possible carry may propagate to the left. Thus the worst case time is $O(\log n)$. Note that main trunks are sorted and there are no active nodes on them, meaning that there is no substantial difference between polynomial queues with r=3 and trinomial heaps. As shown in Section 2, the amortized

time for insert is $O(1)$ with $r = 3$.

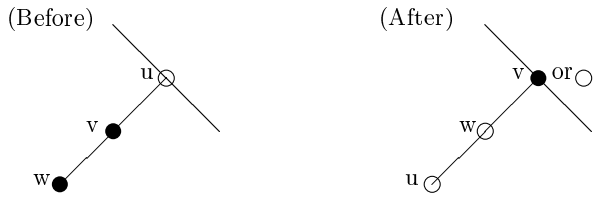

Fig. 3. reordering

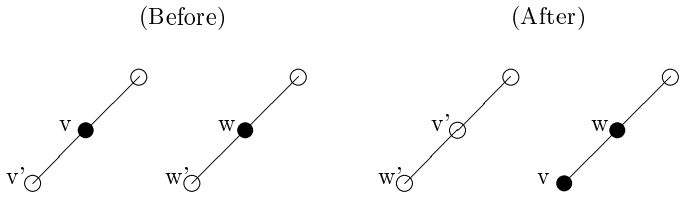

Fig. 4. Rearrangement

Example 3. Let us perform delete-min on the trinomial heap in Fig. 2. Then the biggest tree is broken up as in Fig. 5. After the subtrees rooted at nodes with key 6 and 5 are swapped, the resulting trees and the trees at the top level will be merged.

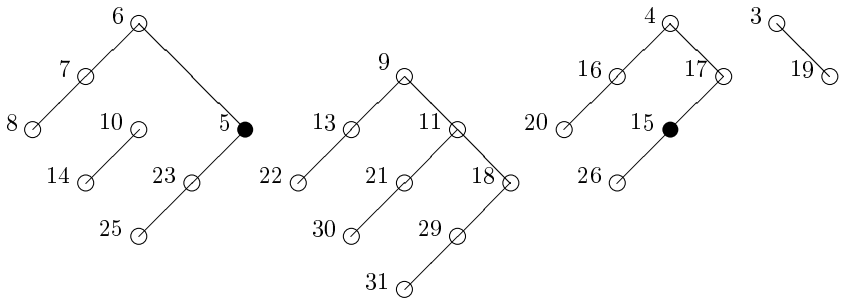

Fig. 5. Trinomial heap with $k = 3$

Example 4. We name a node with key x by $node(x)$. Suppose key 21 is decreased to 1 in Fig. 2. We connect $node(30)$ to $node(26)$, and $node(15)$ to $node(1)$. The result is shown in Fig. 6.

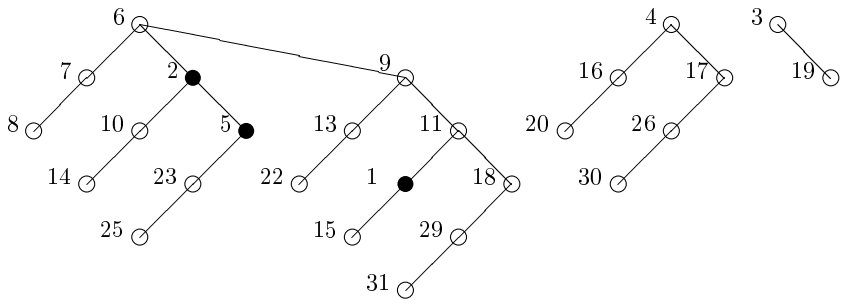

Fig. 6. Trinomial heap with $k = 3$

There are two ways to keep the number of active nodes within tolerance. The first is to allow at most one active node for each dimension. If v becomes active after we perform decrease-key on v, and there is another active w of the same dimension, we can perform rearrangement and/or reordering. The reordering may cause two active nodes at the next higher dimension, causing a possible chain effect of adjustments.

The second is to keep a counter for the number of active nodes. If the counter exceeds the tolerance bound t, we can perform rearrangement and/or reordering on two active nodes of the same dimension, and decrease the counter by one. Note that if the number of active nodes exceeds the tolerance, there must be two active nodes of the same dimension by the pigeon hole principle. There is no chain effect in this case thanks to the global information given by the counter. We allow more than one active nodes of the same dimension, even on the same trunk.

4 $O(1)$ amortized time for decrease-key

We prepare an array, called *active*, of size $t(= k - 1)$, whose elements are active nodes of dimension i $(i = 1, ..., k - 1)$. That is, we have at most one active node for each dimension greater than 0. If there is no active node of dimension i, we denote $active[i] = \phi$. Suppose we perform decrease-key on node v of dimension i, which is now active. If $active[i] = \phi$, we set $active[i] = v$ and finish. If $active[i] = u$, we perform rearrangement and/or reordering on u and v, and adjust *active* accordingly. If we perform reordering, rearrangement and/or reordering will proceed to higher dimensions. This is like carry propagation over array *active*, so the worst case time is $O(\log n)$. Similarly to the insert operation, we can easily show that the amortized time for decrease-key is $O(1)$. As mentioned in Introduction, we can perform m decrease-key operations, and n insert and delete-min operations in $O(m + n \log n)$ time. This implementation is more efficient for applications to shortest paths, etc. than the implementation in the next section, and can be on a par with Fibonacci and 2-3 heaps.

Amortized analysis:. After the key value of a node is decreased, several operations are performed, including checking inconsistencies. The total maximum

number of key comparisons leading to a reordering is five. Thus if we define a deficit function Ψ_i after the i-th decrease-key by five times the number of active nodes, we can show that the amortized time for one decrease-key is $O(1)$. If we have inserts in a series of operations, we need to define the deficit function after the i-th operation by $\Phi_i+\Psi_i$.

5 $O(1)$ worst case time for decrease-key

In this section, we implement trinomial heaps with $O(1)$ worst case time for decrease-key. We prepare a one-dimensional array *point* of pointers to the list of active nodes for each dimension, an array *count*, and *counter*. The variable *counter* is to count the total number of active nodes. The size of the arrays is bounded by the number of dimensions in the heap. The list of nodes for dimension i contains the current active nodes of dimension i. The element $count[i]$ shows the size of the i-th list. To avoid scanning arrays *point* and *count*, we prepare another linked list called *candidates*, which maintains a first-in-first-out structure for candidate dimensions for the clean-up operation. With these additional data structures, we can perform decrease-key with a constant number of operations. Note that we need to modify these data structures on delete-min as well, details of which are omitted.

Example 5. In this example, we have $t = 7$. If we create an active node of dimension 1, we go beyond the tolerance bound, and we perform clean-up using nodes a and b. Suppose this is a rearrangement not followed by reordering. Then a or b is removed from the 3rd list and the first item in *candidates* is removed. Next let us perform reordering using e and f. First we remove e and f from the 6-th list. Suppose $label(e) \leq label(f)$, and e becomes active in the 7-th dimension. Then we append e to the 7-th list, and remove 6 from and append 7 to the list of *candidates*. See Fig. 7. This is a simplified picture. To remove an active node from a list of active nodes in $O(1)$ time, those lists need to be doubly linked.

6 Practical considerations

For the data structure of a trinomial heap, we need to implement it by pointers. For the top level trees, we prepare an array of pointers of size $d = \lceil \log_3(n+1) \rceil$, each of which points to the tree of the corresponding term. Let the dimension of node v be i. 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 d, whose elements are pairs $(first, second)$. The $first$ of the j-th element of the array points to the first node on the j-th trunk of v. The *second* is to the second node. If we prepare the fixed size d for all the nodes, we would need $O(n \log n)$ space. By implementing arrays by pointers, we can implement our algorithm with $O(n)$ space, although this version will be less time-efficient.

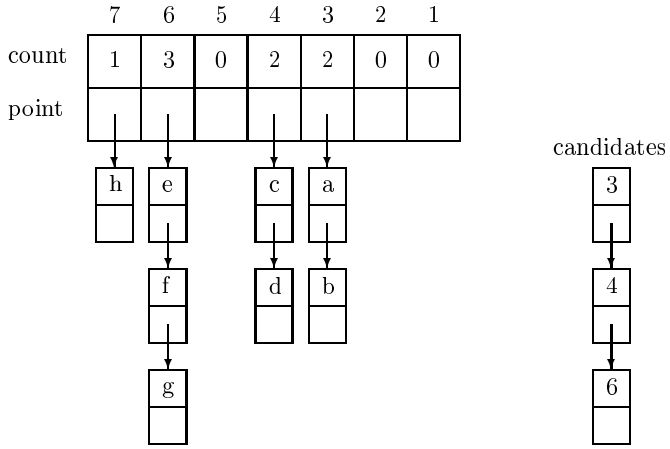

Fig. 7. Auxiliary data structure for $O(1)$ worst case time

7 Concluding remarks

Measuring the complexity of n delete-min's, n insert's, and m decrease-key's by the number of comparisons, we showed it to be $O(m + n \log n)$ in a trinomial heap by two implementations. The first implementation performs decrease-key in $O(1)$ amortized time, whereas the second achieves $O(1)$ worst case time for decrease-key. Although due to less overhead time the first is more efficient for applications such as shortest paths where the total time is concerned, the second has an advantage when the data structure is used on-line to respond to updates and queries from outside, as it can give smoother responses.

The clean-up mechanism is not unique. For example, we could make a node active without key comparison after decrease-key. We will need experiments to see if this is better.

References

1. Dijkstra, E.W., A note on two problems in connexion with graphs, Numer. Math. 1 (1959) 269-271.
2. 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.
3. Fredman, M.L. and R,E, Tarjan, Fibonacci heaps and their uses in inproved network optimization algorithms, Jour. ACM 34 (1987) 596-615
4. Prim, R.C., Shortest connection networks and some generalizations, Bell Sys. Tech. Jour. 36 (1957) 1389-1401.
5. Takaoka, T., Theory of 2-3 Heaps, COCOON 99, Lecture Notes of Computer Science (1999) 41-50
6. Vuillemin, J., A data structure for manipulating priority queues, Comm. ACM 21 (1978) 309-314.