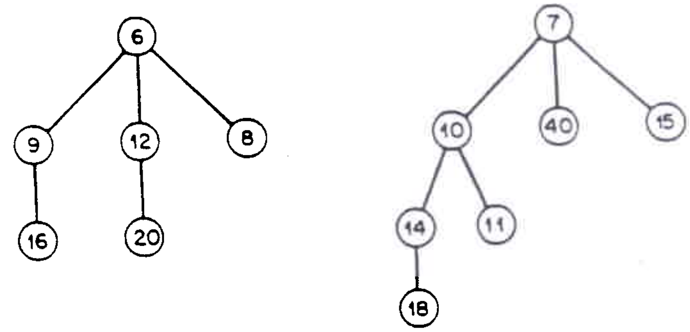
2. *Fibonacci Heaps*

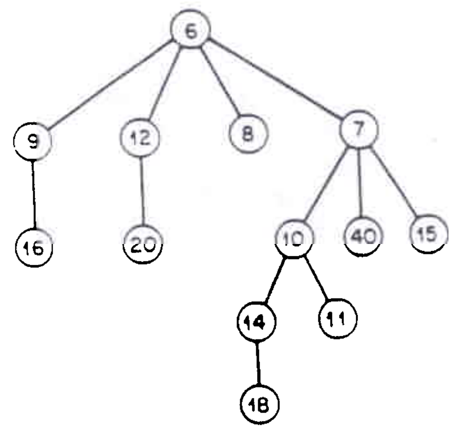
To implement heaps we use heap-ordered trees. A *heap-ordered tree* is a rooted tree containing a set of items, one item in each node, with the items arranged in *heap order*: If x is any node, then the key of the item in x is no less than the key of the item in its parent $p(x)$, provided x has a parent. Thus the tree root contains an item of minimum key. The fundamental operation on heap-ordered trees is *linking*, which combines two item-disjoint trees into one. Given two trees with roots x and y, we link them by comparing the keys of the items in x and y. If the item in x has the smaller key, we make y a child of x; otherwise, we make x a child of y. (See Figure 1.)

A *Fibonacci heap* (*F-heap*) is a collection of item-disjoint heap-ordered trees. We impose no explicit constraints on the number or structure of the trees; the only constraints are implicit in the way the trees are manipulated. We call the number of children of a node x its *rank* $r(x)$. There is no constant upper bound on the rank of a node, although we shall see that the rank of a node with n descendants is $O(\log n)$. Each node is either *marked* or *unmarked*; we shall discuss the use of marking later.

In order to make the correctness of our claimed time bounds obvious, we assume the following representation of F-heaps: Each node contains a pointer to its parent (or to a special node *null* if it has no parent) and a pointer to one of its children. The children of each node are doubly linked in a circular list. Each node also contains its rank and a bit indicating whether it is marked. The roots of all the trees in the heap are doubly linked in a circular list. We access the heap by a pointer to a root containing an item of minimum key; we call this root the *minimum node* of the heap. A minimum node of null denotes an empty heap. (See Figure 2.) This representation requires space in each node for one item, four pointers, an integer, and a bit. The double linking of the lists of roots and children makes deletion from such a list possible in $O(1)$ time. The circular linking makes concatenation of lists possible in $O(1)$ time.



(a)



(b)

FIG. 1. Linking two heap-ordered trees. (In this and most of the succeeding figures, we do not distinguish between items and their keys.) (a) Two trees. (b) After linking.

We shall postpone a discussion of *decrease key* and *delete* until later in the section. The remaining heap operations we carry out as follows: To perform *make heap*, we return a pointer to null. To perform *find min* (h), we return the item in the minimum node of h. To carry out *insert* (i, h), we create a new heap consisting of one node containing i, and replace h by the meld of h and the new heap. To carry out *meld* (h_1, h_2), we combine the root lists of h_1 and h_2 into a single list, and return as the minimum node of the new heap either the minimum node of h_1 or the minimum node of h_2, whichever contains the item of the smaller key. (In the case of a tie, the choice is arbitrary.) All of these operations take $O(1)$ time.

The most time-consuming operation is *delete min* (h). We begin the deletion by removing the minimum node, say, x, from h. Then we concatenate the list of children of x with the list of roots of h other than x, and repeat the following step until it no longer applies.

Linking Step. Find any two trees whose roots have the same rank, and link them. (The new tree root has rank one greater than the ranks of the old tree roots.)

Once there are no two trees with roots of the same rank, we form a list of the remaining roots, in the process finding a root containing an item of minimum key

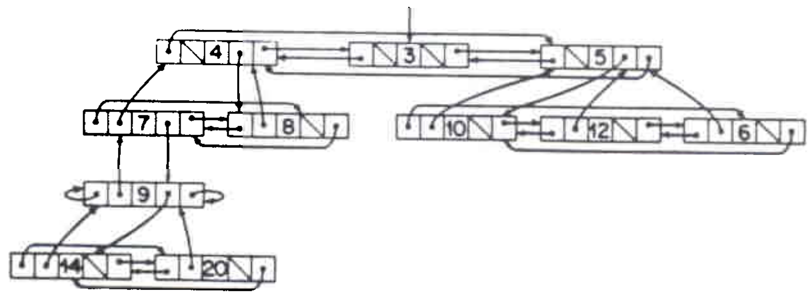


FIG. 2. Pointers representing an F-heap. The four pointers in each node indicate the left sibling, the parent, some child, and the right sibling. The middle field in each node is its key. Ranks and mark bits are not shown.

to serve as the minimum node of the modified heap. We complete the deletion by saving the item in x, destroying x, and returning the saved item. (See Figure 3.)

The *delete min* operation requires finding pairs of tree roots of the same rank to link. To do this we use an array indexed by rank, from zero up to the maximum possible rank. Each array position holds a pointer to a tree root. When performing a *delete min* operation, after deleting the minimum node and forming a list of the new tree roots, we insert the roots one by one into the appropriate array positions. Whenever we attempt to insert a root into an already occupied position, we perform a linking step and attempt to insert the root of the new tree into the next higher position. After successfully inserting all the roots, we scan the array, emptying it. The total time for the *delete min* operation is proportional to the maximum rank of any of the nodes manipulated plus the number of linking steps.

The data structure we have so far described is a "lazy melding" version of binomial queues. If we begin with no heaps and carry out an arbitrary sequence of heap operations (not including *delete* or *decrease key*), then each tree ever created is a *binomial tree*, defined inductively as follows: A binomial tree of rank zero consists of a single node; a binomial tree of rank $k > 0$ is formed by linking two binomial trees of rank $k - 1$. (See Figure 4.) A binomial tree of rank k contains exactly 2^k nodes, and its root has exactly k children. Thus every node in an n-item heap has rank at most $\log n$.[1]

We can analyze the amortized running times of the heap operations by using the "potential" technique of Sleator and Tarjan [19, 25]. We assign to each possible collection of heaps a real number called the *potential* of the heaps. We define the *amortized time* of a heap operation to be its actual running time plus the net increase it causes in the potential. (A decrease in potential counts negatively and thus makes the amortized time less than the actual time.) With this definition, the actual time of a sequence of operations is equal to the total amortized time plus the net decrease in potential over the entire sequence.

To apply this technique, we define the potential of a collection of heaps to be the total number of trees they contain. If we begin with no heaps, the initial potential is zero, and the potential is always nonnegative. Thus the total amortized time of a sequence of operations is an upper bound on the total actual time. The amortized time of a *make heap*, *find min*, *insert*, or *meld* operation is $O(1)$: An insertion increases the number of trees by one; the other operations do not affect

[1] All logarithms in this paper for which a base is not explicitly specified are base 2.

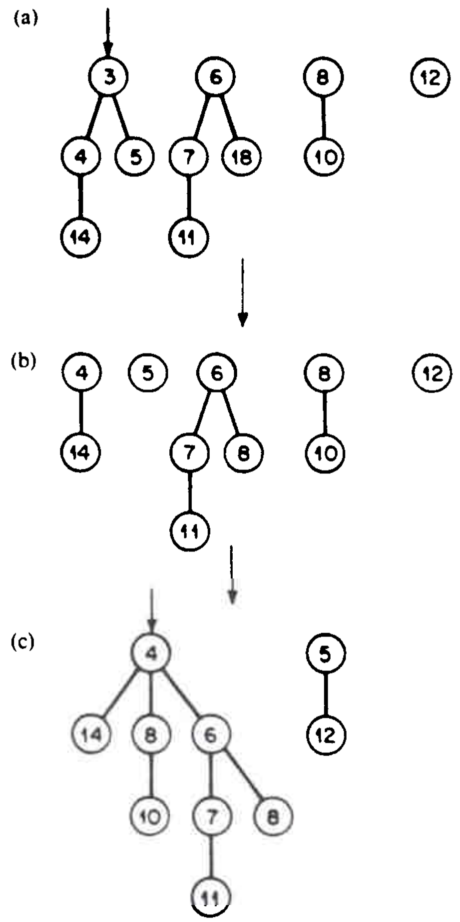


FIG. 3. Effect of a *delete min* operation. (a) Before the deletion. (b) After removal of the minimum node, containing 3. (c) After linking 4 and 8, then 4 and 6, then 5 and 12.

the number of trees. If we charge one unit of time for each linking step, then a *delete min* operation has an amortized time of $O(\log n)$, where n is the number of items in the heap: Deleting the minimum node increases the number of trees by at most $\log n$; each linking step decreases the number of trees by one.

Our goal now is to extend this data structure and its analysis to include the remaining heap operations. We implement *decrease key* and *delete* as follows: To carry out *decrease key* (Δ, i, h), we subtract Δ from the key of i, find the node x containing i, and cut the edge joining x to its parent $p(x)$. This requires removing x from the list of children of $p(x)$ and making the parent pointer of x null. The effect of the cut is to make the subtree rooted at x into a new tree of h, and requires decreasing the rank of $p(x)$ and adding x to the list of roots of h. (See Figure 5.) (If x is originally a root, we carry out *decrease key* (Δ, i, h) merely by subtracting Δ from the key of i.) If the new key of i is smaller than the key of the minimum node, we redefine the minimum node to be x. This method works because Δ is nonnegative; decreasing the key of i preserves heap order within the subtree rooted at x, though it may violate heap order between x and its parent. A *decrease key* operation takes $O(1)$ actual time.

The *delete* operation is similar to *decrease key*. To carry out *delete* (i, h), we find the node x containing i, cut the edge joining x and its parent, form a new list of roots by concatenating the list of children of x with the original list of roots, and

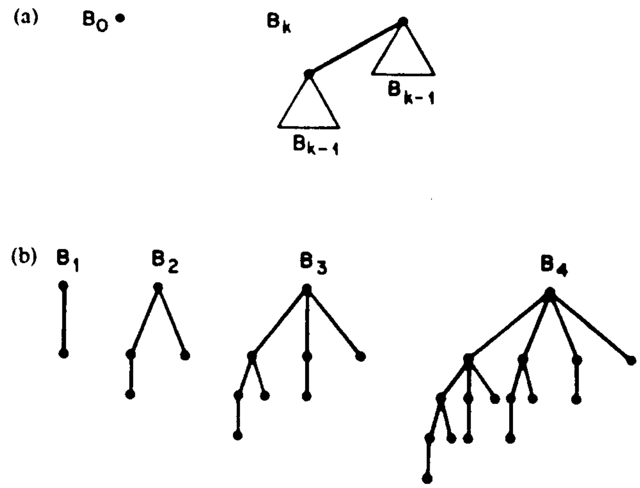


FIG. 4. Binomial trees. (a) Inductive definition. (b) Examples.

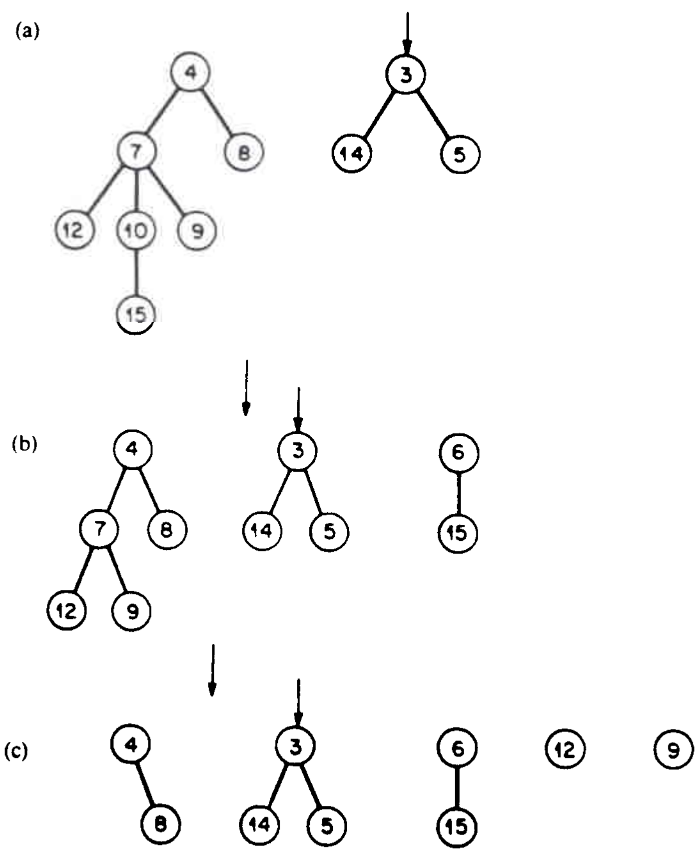


FIG. 5. The *decrease key* and *delete* operations. (a) The original heap. (b) After reducing key 10 to 6. The minimum node is still the node containing 3. (c) After deleting key 7.

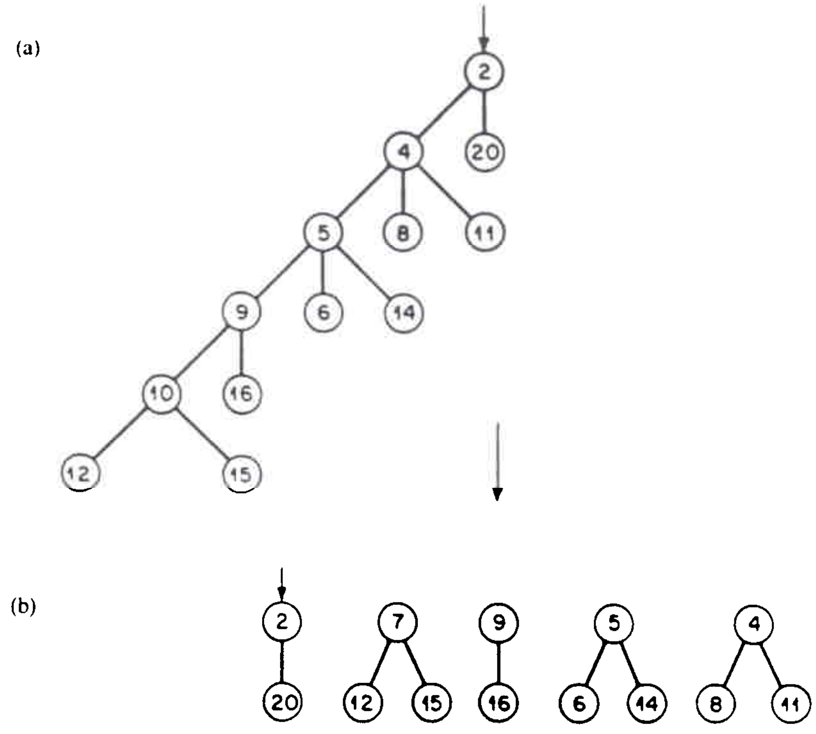


FIG. 6. Cascading cuts. (a) Just before decreasing key 10. Nodes 4, 5, and 9 are assumed to have previously lost a child via a cut. (b) After decreasing key 10 to 7. The original cut separates 10 from 9. Cascading cuts separate 9 from 5, 5 from 4, and 4 from 2.

destroy node x. (See Figure 5.) (If x is originally a root, we remove it from the list of roots rather than removing it from the list of children of its parent; if x is the minimum node of the heap, we proceed as in *delete min.*) A *delete* operation takes $O(1)$ actual time, unless the node destroyed is the minimum node.

There is one additional detail of the implementation that is necessary to obtain the desired time bounds. After a root node x has been made a child of another node by a linking step, as soon as x loses two of its children through cuts, we cut the edge joining x and its parent as well, making x a new root as in *decrease key.* We call such a cut a *cascading cut.* A single *decrease key* or *delete* operation in the middle of a sequence of operations can cause a possibly large number of cascading cuts. (See Figure 6.)

The purpose of marking nodes is to keep track of where to make cascading cuts. When making a root node x a child of another node in a linking step, we unmark x. When cutting the edge joining a node x and its parent $p(x)$, we decrease the rank of $p(x)$ and check whether $p(x)$ is a root. If $p(x)$ is not a root, we mark it if it is unmarked and cut the edge to its parent if it is marked. (The latter case may lead to further cascading cuts.) With this method, each cut takes $O(1)$ time.

This completes the description of F-heaps. Our analysis of F-heaps hinges on two crucial properties: (1) Each tree in an F-heap, even though not necessarily a binomial tree, has a size at least exponential in the rank of its root; and (2) the number of cascading cuts that take place during a sequence of heap operations is bounded by the number of *decrease key* and *delete* operations. Before proving these properties, we remark that cascading cuts are introduced in the manipulation of F-heaps for the purpose of preserving property (1). Moreover, the condition for their occurrence, namely, the "loss of two children" rule, limits the frequency of cascading cuts as described by property (2). The following lemma implies property (1):

LEMMA 1. *Let x be any node in an F-heap. Arrange the children of x in the order they were linked to x, from earliest to latest. Then the ith child of x has a rank of at least $i - 2$.*

PROOF. Let y be the ith child of x, and consider the time when y was linked to x. Just before the linking, x had at least $i - 1$ children (some of which it may have lost after the linking). Since x and y had the same rank just before the linking, they both had a rank of at least $i - 1$ at this time. After the linking, the rank of y could have decreased by at most one without causing y to be cut as a child of x. □

COROLLARY 1. *A node of rank k in an F-heap has at least $F_{k+2} \geq \phi^k$ descendants, including itself, where F_k is the kth Fibonacci number ($F_0 = 0$, $F_1 = 1$, $F_k = F_{k-2} + F_{k-1}$ for $k \geq 2$), and $\phi = (1 + \sqrt{5})/2$ is the golden ratio.* (*See Figure* 7.)

PROOF. Let S_k be the minimum possible number of descendants of a node of rank k. Obviously, $S_0 = 1$, and $S_1 = 2$. Lemma 1 implies that $S_k \geq \sum_{i=0}^{k-2} S_i + 2$ for $k \geq 2$. The Fibonacci numbers satisfy $F_{k+2} = \sum_{i=2}^{k} F_i + 2$ for $k \geq 2$, from which $S_k \geq F_{k+2}$ for $k \geq 0$ follows by induction on k. The inequality $F_{k+2} \geq \phi^k$ is well known [14]. □

Remark. This corollary is the source of the name "Fibonacci heap."

To analyze F-heaps we need to extend our definition of potential. We define the potential of a collection of F-heaps to be the number of trees plus twice the number of marked nonroot nodes. The $O(1)$ amortized time bounds for *make heap*, *find min*, *insert*, and *meld* remain valid, as does the $O(\log n)$ bound for *delete min*; *delete min* (h) increases the potential by at most $1.4404 \log n$ minus the number of linking steps, since, if the minimum node has rank k, then $\phi^k \leq n$ and thus $k \leq \log n/\log \phi \leq 1.4404 \log n$.

Let us charge one unit of time for each cut. A *decrease key* operation causes the potential to increase by at most three minus the number of cascading cuts, since the first cut converts a possibly unmarked nonroot node into a root, each cascading cut converts a marked nonroot node into a root, and the last cut (either first or cascading) can convert a nonroot node from unmarked to marked. It follows that *decrease key* has an $O(1)$ amortized time bound. Combining the analysis of *decrease key* with that of *delete min*, we obtain an $O(\log n)$ amortized time bound for *delete*. Thus we have the following theorem:

THEOREM 1. *If we begin with no F-heaps and perform an arbitrary sequence of F-heap operations, then the total time is at most the total amortized time, where the amortized time is $O(\log n)$ for each delete min or delete operation and $O(1)$ for each of the other operations.*

We close this section with a few remarks on the storage utilization of F-heaps. Our implementation of F-heaps uses four pointers per node, but this can be reduced to three per node by using an appropriate alternative representation [2] and even to two per node by using a more complicated representation, at a cost of a constant factor in running time. Although our implementation uses an array for finding roots of the same rank to link, random-access memory is not actually necessary

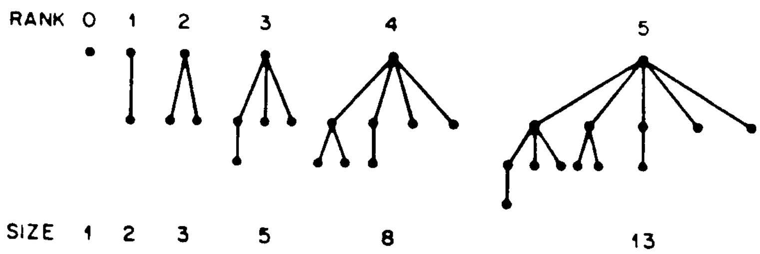


FIG. 7. Trees of minimum possible size for a given rank in a Fibonacci heap.

for this purpose. Instead, we can maintain a doubly linked list of *rank nodes* representing the possible ranks. Each node has a *rank pointer* to the rank node representing its rank. Since the rank of a node is initially zero and only increases or decreases by one, it is easy to maintain the rank pointers. When we need to carry out linking steps, we can use each rank node to hold a pointer to a root of the appropriate rank. Thus the entire data structure can be implemented on a pointer machine [22] with no loss in asymptotic efficiency.

3. *Variants of Fibonacci Heaps*

In this section we consider additional heap operations and four variants of F-heaps designed to accommodate them. We begin with a closer look at deletion of arbitrary items. The $O(\log n)$ time bound for deletion derived in Section 2 can be an overestimate in some situations. For example, we can delete all the items in an n-item heap in $O(n)$ time, merely by starting from the minimum node and traversing all the trees representing the heap, dismantling them as we go. This observation generalizes to a mechanism for "lazy" deletion, due to Cheriton and Tarjan [4]. This idea applied to F-heaps gives our first variant, *F-heaps with vacant nodes*, which we shall now describe.

We perform a *delete min* or *delete* operation merely by removing the item to be deleted from the node containing it, leaving a vacant node in the data structure (which if necessary we can mark vacant). Now deletions take only $O(1)$ time, but we must modify the implementations of *meld* and *find min* since in general the minimum node in a heap may be vacant. When performing *meld*, if one of the heaps to be melded has a vacant minimum node, this node becomes the minimum node of the new heap. To perform *find min* (h) if the minimum node is vacant, we traverse the trees representing the heap top-down, destroying all vacant nodes reached during the traversal and not continuing the traversal below any nonvacant node. This produces a set of trees all of whose roots are nonvacant, which we then link as in the original implementation of *delete min*. (See Figure 8.) The following lemma bounds the amortized time of *find min*.

LEMMA 2. *A find min operation takes $O(l(\log(n/l) + 1))$ amortized time, where l is the number of vacant nodes destroyed and n is the number of nodes in the heap; if $l = 0$, the amortized time is $O(1)$.*

PROOF. If $l = 0$, the lemma is obvious. Thus suppose $l \geq 1$. The amortized time of the *find min* operation is at most a constant times $\log n$ plus the number of new trees created by the destruction of vacant nodes. Let x be any destroyed vacant node, and suppose that x has k nonvacant children before its destruction. By Lemma 1, at least one of these, say, y, has a rank of at least $k - 2$, which means

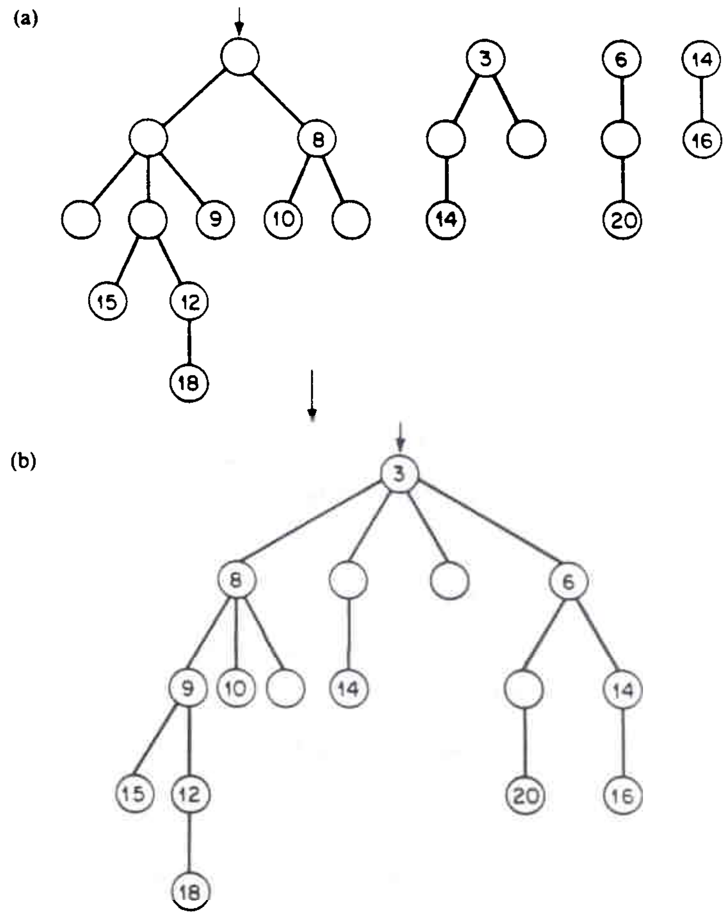


FIG. 8. A *find min* operation on a heap with vacant nodes. (a) The original heap. (b) After *find min*. Subtrees with roots 15, 12, 9, and 8 become trees, and then linking takes place.

that y is the root of a subtree containing at least ϕ^{k-2} nodes. These ϕ^{k-2} nodes can be uniquely associated with x. In other words, if the l destroyed vacant nodes have $k_1, k_2, \ldots, k_l$ nonvacant children, then $\sum_{i=1}^{l} \phi^{k_i-2} \le n$. Subject to this constraint, the sum of the k_is, which counts the number of new trees created, is maximized when all the k_is are equal. This implies that $\sum_{i=1}^{l} k_i = O(l(\log(n/l) + 1))$, giving the lemma. □

Lazy deletion is especially useful for applications in which the deleted items can be identified implicitly, as, for example, if there is a predicate that specifies whether an item is deleted. The main drawback with lazy deletion is that it may use extra space if individual items are inserted and deleted many times. We can avoid this drawback by changing the data structure slightly, giving our second variant of F-heaps, called *F-heaps with good and bad trees.*

In the new variant, we avoid the use of vacant nodes. Instead, we divide the trees comprising the F-heap into two groups: *good trees* and *bad trees.* We maintain the minimum node to be a root of minimum key among the good trees. When inserting an item, the corresponding one-node tree becomes a good tree in the heap. When melding two heaps, we combine their sets of good trees and their sets of bad trees. When performing a *decrease key* operation, all the new trees formed by cascading cuts become good trees. We carry out a *delete* operation as described in Section 2 except that, if the deleted item is the minimum node of the heap, all the subtrees rooted at its children become bad trees; we delay any linking until the next *find min*. (This includes the case of a *delete min* operation.) To carry out a *find min*, we check whether there are any bad trees. If not, we merely return the minimum node. If so, we link trees whose roots have equal rank until there are no two trees of equal rank, make all trees good, update the minimum node, and return it.

With this variant of F-heaps, we obtain the following amortized time bounds (using an analysis similar to that above): $O(1)$ for *find min*, *insert*, *meld*, and *decrease key*; and $O(\log(n/l) + 1)$ per *delete* or *delete min* operation for a sequence of l such operations not separated by a *find min*. The advantage of this variant is that it requires no space for vacant nodes. The disadvantage is that it does not support implicit deletion.

Our third variant of F-heaps uses only a single tree to represent each heap. In this *one-tree variant*, we mark each node that is not a root as either *good* or *bad*; the node type is determined when the linking operation that makes the node a nonroot is done. These marks are in addition to the marks used for cascading cuts. We define the rank of a node to be its number of good children (i.e., we do not count the bad children).

To meld two heaps, we link the corresponding trees, making the root that becomes a child bad. This melding does not change the rank of any node. To perform *find min*, we return the root of the tree representing the heap. To perform *delete min*, we delete the tree root, link pairs of trees whose roots have equal rank until no such linking is possible, making each node that becomes a nonroot good, and then link all the remaining trees in any order, making each node that becomes a nonroot bad. To perform *decrease key*, we perform the appropriate cuts (using mark bits as before), producing a set of trees. We link these trees in any order, making each root that becomes a nonroot bad. To perform an arbitrary deletion, we do the appropriate cuts, discard the node to be deleted, and combine the remaining trees using links as in *delete min*, first combining trees with roots of equal rank and marking the new nonroots good, then combining trees of unequal rank and marking the new nonroots bad.

The one-tree variant of F-heaps has the following properties: Corollary 1 still holds; that is, any node with k good children has at least $F_{k+2} \geq \phi^k$ descendants. The number of bad children created during the running of the algorithm is at most one per *insert* or *meld*, one per cut (and hence $O(1)$ per *decrease key*), and at most $O(\log n)$ per *delete min* or *delete*. Combining this with the previous analysis of F-heaps, we obtain an amortized time bound of $O(1)$ for *insert*, *meld*, *find min*, and *decrease key*, and $O(\log n)$ for *delete min* and *delete*.

Our fourth and last variant of F-heaps, called *F-heaps with implicit keys*, is designed to support the following additional heap operation:

increase all keys (Δ, h): Increase the keys of all items in heap h by the arbitrary real number Δ.

To implement *increase all keys*, we represent the keys of the items implicitly rather than explicitly, using a separate data structure. A suitable variant of compressed trees [23] suffices for this purpose. We maintain a compressed tree for each heap. Each node in the tree contains a value and possibly an item. Any node

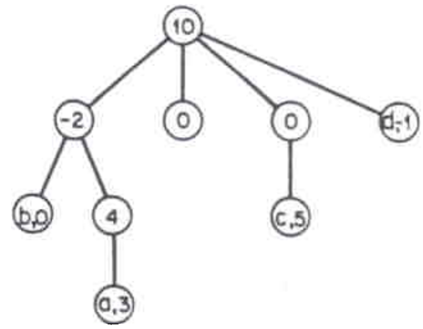


FIG. 9. A compressed tree. Items are a, b, c, and d. The key of item a is $3 + 4 - 2 + 10 = 15$.

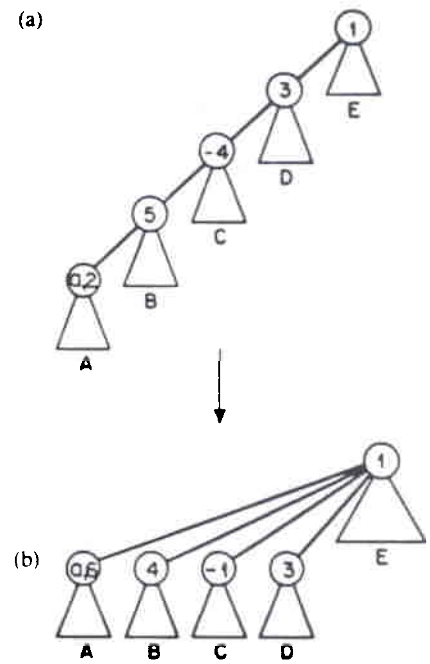


FIG. 10. Path compression. Triangles denote subtrees. (a) The original tree. (b) After evaluating key a.

containing an item has no children. The values represent the keys as follows: If x is any node containing an item i, the sum of the values of the ancestors of x (including x itself) is the key of i. (See Figure 9.)

We manipulate this data structure as follows: When executing *make heap*, we construct a new one-node compressed tree to represent the heap. The new node has value zero. When executing *insert* (i, h), we create a new compressed tree node x containing i, make the root of the compressed tree representing h the parent $p(x)$ of x, and give x a value defined by $value(x) = key(i) - value(p(x))$. When deleting an item i from a heap, we destroy the compressed tree node containing i. When executing *decrease key*(Δ, i, h), we subtract Δ from the value of the compressed tree node containing i. To perform *increase all keys*(Δ, h), we add Δ to the value of the root of the compressed tree representing h.

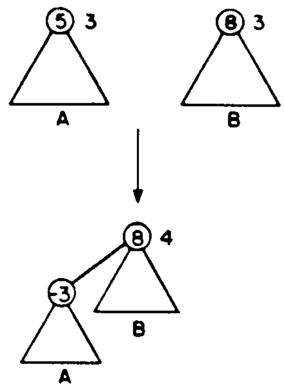


FIG. 11. Combining two compressed trees. Ranks are outside the roots.

Whenever we need to evaluate the key of an item i (as in a *find min* operation or a linking step), we locate the compressed tree node x containing i and follow the path from x through its ancestors up to the tree root. Then we walk back down the path from the root to x, *compressing* it as follows: When we visit a node y that is not a child of the root, we replace $value(y)$ by $value(y) + value(p(y))$ and redefine $p(y)$ to be the root. (See Figure 10.) This compression makes every node along the path a child of the root and preserves the relationship between values and keys. After the compression, we return $value(x) + value\ p(x)$ as the key of i.

The last operation we must consider is melding. (If no melding takes place, then the depth of every compressed tree is at most one, and each operation on a compressed tree takes $O(1)$ time.) To facilitate melding we maintain for each compressed tree root a nonnegative integer *rank*. (This rank should not be confused with the rank of a heap-ordered tree node; it does not count the number of children.) A newly created compressed tree root has rank zero. When executing $meld(h_1, h_2)$, we locate the roots, say, x and y, of the compressed trees representing h_i and h_2. If $rank(x) > rank(y)$, we make x the parent of y and redefine $value(y)$ to be $value(y) - value(x)$. If $rank(x) < rank(y)$, we make y the parent of x and redefine $value(x)$ to be $value(x) - value(y)$. Finally, if $rank(x) = rank(y)$, we increase $rank(x)$ by one and proceed as in the case of $rank(x) > rank(y)$. (See Figure 11.)

To implement this data structure, we need one compressed tree node for each *make heap* operation plus one node per item, with room in each node for an item or rank, a value, and a parent pointer. The total time for compressed tree operations is $O(m + f\alpha(m + f, n))$, where m is the number of heap operations, n is the number of *make heap* and *insert* operations, f is the number of key evaluations, and α is a functional inverse of Ackerman's function [23, 26]. In most applications of heaps requiring use of the *increase all keys* operation, the time for manipulating heap-ordered trees will dominate the time for manipulating compressed trees. Note that the two data structures are entirely separate; we can use compressed trees in combination with any implementation of heaps that is based on key comparison.

4. *Shortest Paths*

In this section we use F-heaps to implement Dijkstra's shortest path algorithm [5] and explore some of the consequences. Our discussion of Dijkstra's algorithm is based on Tarjan's presentation [24]. Let G be a directed graph, one of whose vertices is distinguished as the *source* s, and each of whose edges (v, w) has a