AN OPTIMAL ALGORITHM FOR SELECTION IN A MIN-HEAP

Greg N. Frederickson

CSD-TR-91-027
April 1991

An Optimal Algorithm for Selection in a Min-Heap

Greg N. Frederickson*

Department of Computer Science
Purdue University
West Lafayette, Indiana 47907

email: gnf@cs.purdue.edu

April 19, 1991

Abstract. An $O(k)$-time algorithm is presented for selecting the k-th smallest element in a binary heap of size $n \gg k$. The result matches the information theoretic lower bound, settles an open problem, and contradicts a recently reported lower bound. Two applications are given for this algorithm.

Key words and phrases. Data structures, heap, partial order, resource allocation, selection.

*This research was supported in part by the National Science Foundation under grants CCR-86202271 and CCR-9001241, and by the Office of Naval Research under contract N00014-86-K-0689.

1. Introduction

A problem of importance in the design of a number of algorithms is that of finding the k-th smallest element in a set of unordered elements [BFPRT], [SPP], [BJ]. A more general version of this problem, and one with additional applications, allows the set of elements to satisfy a relation encoded in some partial order [FJ2], or forbids certain total orders for the set [FJ1], [CSSS]. For example, algorithms with optimal running times have been given in [FJ2] for selecting the k-th smallest element in a collection of sorted matrices (matrices whose entries in any row or column are in sorted order). For the above version of this problem, the upper bounds match information theory lower bounds [FJ2]. A natural question to ask is whether the complexity of finding the k-th smallest element in a set of elements organized in any given partial order coincides with the information theory lower bound. Thus we wish to know if the information in certain partial orders can be extracted efficiently. In addition, we wish to know how to coordinate the acquisition of information from partial orders.

We focus these questions by considering a partial order for a binary min-heap, i.e., a partial order on items x_i, $i = 1, 2, \cdots, n$, where $x_i > x_{\lfloor i/2 \rfloor}$ for $i = 2, \cdots, n$. We shall concentrate on examples in which k is much much smaller than n, so that for all intents and purposes the heap can be viewed as infinite. (This is not a real restriction on the problem, since the heap can be viewed as "padded out" with very large values.) The problem of selecting the k-th smallest element in a heap is challenging because the partial order associated with a heap is rather "bushy", and for $k > 1$ the k-th smallest element can be in any one of $2^k - 2$ positions in the heap. In investigations culminating in [FJ2], this author and Donald Johnson observed that there was a sizable gap between the straightforward lower bound of $\Omega(k)$ and a simple upper bound of $O(k \log k)$.[1]

[1]All logarithms are to the base 2.

The problem of narrowing this gap by raising the lower bound was stated as an open problem in [AK]. Recently, a lower bound of $\Omega(k \log k)$ has been claimed in [WN], but the arguments make unwarranted assumptions about how such a selection algorithm must work.

The straightforward lower bound is based on information theory. The number of binary trees with k nodes is $\frac{1}{k+1}\binom{2k}{k}$, which is greater than $4^k/(k\sqrt{\pi k})$. (See pp. 388–389 of [K]). It follows that the height of any decision tree to select the k-th element in an infinite heap is at least $2k - \frac{3}{2}\log k - 1$.

The straightforward upper bound is achieved by creating an auxiliary heap. The auxiliary heap is initialized as containing a single value equal to that of element x_1. Then the following is done for k steps: The minimum is extracted from the auxiliary heap, and two values are inserted into it. The values inserted are equal to elements x_{2i} and x_{2i+1}, where x_i equals the value just extracted from the auxiliary heap. Clearly, this takes $O(k \log k)$ time. We note that the inverse problem of finding the rank of a given element x in a heap is easy. Just perform an inorder traversal of the subtree induced by all values no greater than x. This clearly takes $O(k)$ time, where k is the rank of x in the heap.

In this paper we present an algorithm to select the k-th smallest element in a heap that is optimal to within a constant factor. We achieve a time of $O(k)$ by using the following ideas. First, we organize appropriate elements into groups, called clans, and handle a representative of each clan in a heap. Second, we form clans on a number of different levels, forming larger clans recursively from smaller clans. We use heaps on each of the resulting $O(\log^* k)$ levels.[2] Third, we redefine clans, choosing a nonobvious rule for inclusion of elements into a clan. Finally we form each clan incrementally, using a number of different clan sizes on any level, and insert the new representative

[2]The function $\log^* k$ is the iterated logarithm of k, defined by $\log^* 1 = \log^* 2 = 1$ and $\log^* k = 1 + \log^* \lceil \log k \rceil$ for $k > 2$.

of an enlarged clan back into a heap.

We discuss two applications for our selection problem. The first application is the enumeration of the k smallest combinatorial objects of some particular type. We focus on enumerating the k smallest spanning trees of a weighted undirected graph, as discussed in [F2]. Let T_i denote the i-th smallest spanning tree of the graph. The $k-1$ spanning trees $T_2, \cdots, T_k$ can be generated one at a time. Each tree T_i with $i > 1$ will be derived from some tree T_j, $j < i$, by a swap (e_i, f_i), in which a tree edge e_i is replaced by a nontree edge f_i. So that no tree is derived more than once, an inclusion-exclusion approach can be used [L1] and [L2, pages 100–104]. It follows that a binary tree describes how the spanning trees are derived from each other. From this binary tree, a min-heap can be derived such that each node in the min-heap is labeled by the cost of a corresponding spanning tree. Our algorithm here can be used in the algorithm of [F2] to identify the cost of k-th smallest spanning tree. The portions of the min-heap accessed by our algorithm here can be generated on the fly by the algorithm in [F2] at only a constant multiplicative factor increase in time. The total time for the algorithm in [F2], exclusive of operations that search the min-heap, is $O(m \log \beta(m,n) + \min\{k^{3/2}, km^{1/2}\})$. Using our algorithm for selecting the k-th smallest element in a min-heap rather than the straightforward $O(k \log k)$-time algorithm gives a faster algorithm whenever k grows faster than $2^{m^{1/2}}$. If the underlying graph is planar, then there is an algorithm in [F2] whose total time, exclusive of operations that search the min-heap, is $O(n + k(\log n)^3)$. Using our algorithm for selecting the k-th smallest element in a heap gives a faster algorithm whenever k grows faster than $2^{(\log n)^3}$.

Our selection problem in a binary heap also has an application in the area of resource allocation problems [IK]. Consider for example a sales region, that can be divided into subregions, which can be subdivided into smaller regions, etc. The manager for the sales region will have an assistant for each subregion, and each assistant will have an assistant for each smaller region, etc. This organization can clearly be modeled by a tree. Assume that any sales region that is divided into two subregions, so that the organization is modeled by a binary tree. Assume that a certain positive benefit x_i accrues to the placement of each manager or assistant in a particular region i, and that the benefit of placing a manager in a region is greater than the benefit obtained by placing any of his assistants. Suppose that the company has k people that it wishes to place, so as to maximize the total benefit obtained. This problem reduces to the problem of selecting the k-th largest element in a max-heap. Our algorithm for selecting the k-th smallest element in a min-heap can be easily adapted and applied, identifying the k people who should be placed, along with which regions should be divided. This is a special case of tree-structured resource problems, which have been discussed in a general context in [IK, sections 9.3 and 9.4], [B1], [B2], [M]. Our algorithm is faster than the algorithms discussed in the above references, for the problem in the form discussed here.

In section 2 we show how to group the elements together into a relatively simple scheme that realizes a time of $O(k \log\log k)$. We then apply this approach recursively to yield an $O(k\, 3^{\log^* k})$-time algorithm. (Note that $k\, 3^{\log^* k}$ is asymptotically a slowly growing function.) In section 3 we elaborate on the additional ideas that bring the time down first to $O(k\, 2^{\log^* k})$, and then to $O(k)$. For simplicity in the exposition, we assume that all elements in the heap are distinct.

A preliminary version of this paper appeared in [F1].

2. Grouping elements into clans

In this section we show how to group elements into subsets in order to reduce the overhead of extracting a minimum in a heap. First, we show how to use a second auxiliary heap to achieve a time of $O(k \log\log k)$. Then we generalize the problem and apply a recursive scheme to achieve a time of $O(k\, 3^{\log^* k})$.

Our first idea is to group elements together into subsets of equal size, called *clans.* We shall choose the clan size initially to be $\lfloor \log k \rfloor$. The largest element in each clan will be called the *representative* of the clan. The representatives of the clans are then inserted into and extracted from the auxiliary heap. If the clans are formed appropriately, then after $\lceil k/\lfloor \log k \rfloor \rceil$ *extractmin* operations, the last element extracted will have rank between k and $2k$ in the original heap. It is then easy to identify all elements in the original heap smaller than this element and select directly in this set, using the algorithm from [BFPRT]. Note that since there are only $\lceil k/\lfloor \log k \rfloor \rceil$ *extractmin* operations in the auxiliary heap, the total time to perform these *extractmin* operations is $O(k)$ and thus is no longer a bottleneck in the algorithm. It follows that the bottleneck in the algorithm is in forming the clans.

The crucial issue is how the elements should be grouped into clans. A logical choice is to group the $\lfloor \log k \rfloor$ smallest elements together to form a first clan C_1. A second clan C_2 can be formed by grouping the next $\lfloor \log k \rfloor$ smallest elements together. To describe the formation of the remaining clans we introduce some terminology. Assume that the clans are formed one at a time. When a clan C_j is formed, those elements that are not in C_j, but are children of elements in the original heap that are in C_j, are called the *offspring* $os(C_j)$ of clan C_j. For $i > 2$, the clan C_i can be formed in one of two ways. First, consider an *extractmin* operation, which extracts the representative of some clan, say C_j. A clan C_i is formed by grouping the $\lfloor \log k \rfloor$ smallest elements from the subheaps rooted at the offspring $os(C_j)$ of C_j.

The second way to form a clan is motivated by the following. Since there may be some offspring of clan C_j not included in C_i, and since clan C_j no longer has a representative in the auxiliary heap, we assign them as a responsibility to the new clan

C_i, and call them the *poor relations* $pr(C_i)$ of clan C_i. Thus if there is a nonempty set of poor relations $pr(C_j)$ of clan C_j, then when the representative of clan C_j is extracted from the auxiliary heap, a second clan C_{i+1} is created, consisting of the $\lfloor \log k \rfloor$ smallest elements from the subheaps of the original heap rooted at the poor relations of clan C_j.

Thus for each clan C_j, $j > 1$, there are two sets associated with it, the offspring $os(C_j)$ and the poor relations $pr(C_j)$. When the representative of C_j is extracted from the auxiliary heap, clans must be formed starting with each of these two sets.

As an example, consider the heap in Figure 1. We can imagine that the heap is actually much larger, and only the first five levels are shown in the figure. Suppose $k = 8$. Then each clan will be of size 3. The first four clans created have been circled. First we find $C_1 = \{1,2,3\}$, with $os(C_1) = \{7,10,12,4\}$ and $pr(C_1) = \emptyset$. The representative of C_1 will be 3. When 3 is extracted from the auxiliary heap, clan C_2 will be created. Clan C_2 is grown from $os(C_1)$. Thus $C_2 = \{4,5,6\}$, $os(C_2) = \{8,11,17,19\}$ and $pr(C_2) = \{7,10,12\}$. The representative of C_2 will be 6. When 6 is extracted from the auxiliary heap, clans C_3 and C_4 will be created. Clan C_3 is grown from $os(C_2)$. Then $C_3 = \{8,11,13\}$, $os(C_3) = \{26,21,79,20,14\}$ and $pr(C_3) = \{17,19\}$. The representative of C_3 will be 13. Clan C_4 is grown from $pr(C_2)$. Then $C_4 = \{7,9,10\}$, $os(C_4) = \{15,16,32,18,28\}$ and $pr(C_4) = \{12\}$. The representative of C_4 will be 10. When 10 is extracted from the auxiliary heap, clans C_5 and C_6 will be created. (We do not identify these here.) Note that 3 values will be extracted from the auxiliary heap. Then the set of values in the original heap that are less than or equal to 10 will be identified, and the k-th smallest value among them will be selected directly.

The maximum size of the offspring and poor relations sets will be $2\lfloor \log k \rfloor$ and $2\lfloor \log k \rfloor - 1$, resp. The bound on the size of the offspring follows, since each element in the offspring set is a child of some element in the clan, and each element has at most two children. The bound on the size of poor relations follows, since all but one element in the offspring can become poor relations when a clan is formed. (Also, a clan can be formed from a poor relations set, but this cannot produce a bound as large as that from forming a clan from offspring.)

A clan can be formed from either an offspring set or a poor relations set by doing the following. Initialize yet another heap to contain the members of the set. Then for $\lfloor \log k \rfloor$ times, extract the minimum from the heap, and insert its two children from the original heap. It will take $O(\log k \log\log k)$ time in total to determine a clan.

We summarize our first algorithm $SEL1$ as follows. Using an auxiliary heap H_2 find the $\lfloor \log k \rfloor$ smallest elements in H_0. Let these be clan C_1, and determine the offspring $os(C_1)$. Set $pr(C_1)$ to the empty set. Initialize an auxiliary heap H_1 with the representative of C_1. Then for $\lceil k/\lfloor \log k \rfloor \rceil$ times, do the following. Perform an *extractmin* operation in H_1. Let C_j be the clan represented by the element extracted. Using heap H_2, find the $\lfloor \log k \rfloor$ smallest elements in the subheaps rooted at members of $os(C_j)$. Let these be clan C_i, where $i-1$ is the number of clans found up to that point. Determine the members of the sets $os(C_i)$ and $pr(C_i)$. If $pr(C_j)$ is not empty, find the $\lfloor \log k \rfloor$ smallest elements in the subheaps rooted at members of $pr(C_j)$. Let these be clan C_{i+1}. Determine the members of the sets $os(C_{i+1})$ and $pr(C_{i+1})$. Insert the representatives of C_i and C_{i+1} into H_1. When the above loop terminates, take the last element extracted from H_1, and identify the set of elements in H_0 less than or equal to this element. Select the k-th smallest element in this set, using the algorithm in [BFPRT]. This completes the description of algorithm $SEL1$.

Lemma 2.1. Algorithm $SEL1$ finds a k-th smallest element in a binary heap in $O(k \log\log k)$ time.

Proof. For correctness, note that the clans are disjoint, and that the representative of a clan is the largest element in the clan. Furthermore, the last element extracted from auxiliary heap H_1 is the largest of $\lceil k/\lfloor\log k\rfloor\rceil$ representatives. Thus the last element extracted is transitively as large as at least k elements. Since algorithm $SEL1$ selects from the set of all elements less than or equal to the last element extracted, all elements less than or equal to the k-th smallest element when be included in this set. Thus the algorithm will return the k-th smallest element.

We analyze the time of algorithm $SEL1$ as follows. As discussed previously, the $O(k/\log k)$ operations in heap H_1 will use $O(k)$ time. Since at most $2\lceil k/\lfloor\log k\rfloor\rceil + 1$ clans are formed, and each clan can be formed in $O(\log k \log\log k)$ time, the total time to form all clans will be $O(k \log\log k)$. It remains to bound the time to form and select in the set of all elements less than or equal to the last element extracted from H_1. We claim that the cardinality of this set is $O(k)$, so that the time for this last step is $O(k)$, and the time for the algorithm as a whole is $O(k \log\log k)$.

We complete the proof by showing that the cardinality of this set is as claimed. Note that for any clan C_j whose representative is in H_1, its representative is smaller than any of its offspring or poor relations. Thus when any clan representative is extracted from H_1, any element smaller than it must be in some clan. Every time a representative of a clan C_j, $j > 1$, is extracted from the auxiliary heap, at most two clans are formed. Thus after the representative of clan C_j, $j > 1$, has been extracted, there are at most $2j$ clans. Thus after the $\lceil k/\lfloor\log k\rfloor\rceil$-th *extractmin* is performed, yielding value x, there are fewer than $2(k + \lfloor\log k\rfloor)$ elements in all clans. Since the last two clans are created as a result of extracting x, all elements in them are greater than x. Thus there are fewer than $2k$ elements less than or equal to x. $\square$

We note that the above scheme creates and uses heaps on two levels. We next wish to extend this scheme recursively. Since the smaller heaps were initialized with more than element, we generalize our problem as follows. Let k be a positive integer, and let $\mathcal{H}$ be a forest of at most $2k$ (infinite) heaps. We wish to find a k-th smallest element in $\mathcal{H}$. Our definition of clans will be essentially the same, except that the basis of the definition will be different. In general there will be more than one clan created before any representative elements are extracted from the corresponding heap.

Our algorithm *SEL2* to select the k-th smallest element in a heap is the following. It calls the recursive procedure *RSEL2* with parameters $\{H_0\}$, 1, and k. In general, procedure *RSEL2*$(\mathcal{H}, l, r)$ returns the r smallest elements, as well as the r-th smallest element in a collection $\mathcal{H}$ of at most $2r$ heaps. The parameter l is a level number. (The level number is not actually used by the procedure, but it is useful in describing the action of the procedure.) Thus the initial call *RSEL2*$(\{H_0\}, 1, k)$ returns the k smallest elements, as well as the k-th smallest element in the original heap.

Procedure *RSEL2*$(\mathcal{H}, l, r)$ is the following. If $r = 1$, return the smallest value among the roots of the heaps in $\mathcal{H}$. Otherwise, do the following. Partition $\mathcal{H}$ into subsets, with each subset except at most one, containing $2\lfloor \log r \rfloor$ heaps, and the remaining at most one subset containing fewer than $2\lfloor \log r \rfloor$ heaps. For each subset $\mathcal{H}_i$ recursively find the $\lfloor \log r \rfloor$ smallest elements in $\mathcal{H}_i$. Designate these elements as clan C_i, and determine the offspring $os(C_i)$. Set $pr(C_i)$ to the set of roots of heaps in $\mathcal{H}_i$ that are not in C_i. Initialize an auxiliary heap H_l with the representatives of the clans C_i. Then for $\lceil r/\lfloor \log r \rfloor \rceil$ times, do the following. Perform an *extractmin* operation in H_l. Let C_j be the clan represented by the element extracted. Recursively at level $l+1$ find the $\lfloor \log r \rfloor$ smallest elements in the subheaps rooted at members of $os(C_j)$. Let these be clan C_i, where $i-1$ is the number of clans found up to that point. Determine the members of the sets $os(C_i)$ and $pr(C_i)$. If $pr(C_j)$ is not empty, recursively find at level $l+1$ the $\lfloor \log r \rfloor$ smallest elements in the subheaps rooted at members of $pr(C_j)$. Let these be clan C_{i+1}. Determine the members of the sets $os(C_{i+1})$ and $pr(C_{i+1})$. Insert the representatives of C_i and C_{i+1} into H_l. When the above loop terminates, take the last element extracted from H_l, and identify the set of elements in $\mathcal{H}$ less than or equal to this element. Select the r-th smallest element in this set. Return the r smallest elements in this set. This completes the description of recursive procedure *RSEL2*.

Lemma 2.2. Algorithm *SEL2* finds a k-th smallest element in a binary heap in $O(k\, 3^{\log^* k})$ time.

Proof. Correctness is established in a fashion similar to that in the proof of Lemma 2.1. We analyze the time for a call to *RSEL2* with parameter r. The time to find the elements with which to initialize heap H_l is dominated by the time to perform at most $\lceil r/\lfloor \log r \rfloor \rceil$ recursive calls on problems of size $\lfloor \log r \rfloor$. The time to set up the heap and perform $\lceil r/\lfloor \log r \rfloor \rceil$ *extractmin* operations and $2\lceil r/\lfloor \log r \rfloor \rceil$ *insert* operations is $O(r)$ time. For each *extractmin* there are either 1 or 2 recursive calls on problems of size $\lfloor \log r \rfloor$, one for the offspring $os(C_j)$ of clan C_j, and the other call for the poor relations $pr(C_j)$, if any, of C_j. Finding the r-th smallest element from among all elements in $\mathcal{H}$ less than or equal to the last representative extracted will take $O(r)$ time. This follows since these elements will be only in the at most $3\lceil r/\lfloor \log r \rfloor \rceil$ clans that are created, and by an argument similar to that in the proof of Lemma 2.1, there will be $O(r)$ such elements. Thus the time to find the r-th smallest element is described by the recurrence:

$$\begin{aligned} T(1) &\leq c \\ T(r) &\leq c\,r + 3\lceil r/\lfloor \log r \rfloor \rceil T(\lfloor \log r \rfloor) \end{aligned}$$

It follows that $T(r)$ is $O(r\, 3^{\log^* r})$. □

3. Redefining the clans and building them incrementally

In this section we identify additional ideas that speed up our basic approach. First, we show how to avoid discarding the structure built up from previous recursive calls.

Instead of forming clans that contain the $\lfloor \log r \rfloor$ smallest elements in a given subset on recursive levels, we form clans from smaller clans that have smallest representatives from a given subset. This idea reduces the time somewhat, and sets the stage for the remaining idea. Second, we reduce the number of elements larger than the r-th smallest element that are examined. This is accomplished by introducing a number of sizes of clans at each level of recursion, and by expanding clans incrementally. When a less than full-size clan is expanded, its new representative is inserted back into the heap. The appropriate combination of these ideas yields an $O(k)$-time algorithm.

We first discuss how to avoid discarding the structure built up from previous recursive calls. Our recursive procedure from the previous section sets up an auxiliary heap H_l on level l, and recursively identifies clans whose representatives are inserted into H_l. It then returns the r smallest elements, as well as the r-th smallest element. Identifying the r-th smallest element ruins the structure, because in general many clans will have only some and not all of their elements less than or equal to the r-th smallest.

To be able to reuse the structure generated within the recursion, we modify the recursive procedure so that it does not identify an r-th smallest element. Instead, the modified procedure will return the last representative extracted from H_l, which will represent a clan at level l containing clans at level $l+1$ represented by elements that were extracted from H_l. Thus we revise the notion of a clan as follows. First, we define a function $h(\cdot)$ that defines the sizes of clans in terms of a target size r. This helps whenever the size of a clan at level $l+1$ does not evenly divide into a target size r. If $r = 1$, then $h(r) = 1$. Otherwise, $h(r) = h(\lfloor \log r \rfloor)\lceil r/h(\lfloor \log r \rfloor)\rceil$. Clearly, $r \leq h(r) < 2r$.

Next, we specify a clan as follows. If $r = 1$, then the clan is the smallest element in the appropriate subset of elements. Otherwise, a clan at level l is the set of elements in the $h(r)/h(\lfloor \log r \rfloor)$ clans at level $l+1$ whose representatives were extracted from heap H_l to form this clan at level l. When a clan is formed, it will have an element and a pointer associated with it. The element will be the largest element, which will be the representative of the clan. The pointer will be to the remaining portion of heap H_l.

We are able to reuse the structure generated within the recursion in the sense that we do not discard the remaining portion of heap H_l, but save it so that we can perform additional *extractmin* operations (and the corresponding *insert* operations too) in it. This alone is not enough, since the repeated straightforward reuse of this heap over the course of time would cause it to grow very large, with a concomitant growth in the cost of performing a heap operation. To prevent this growth, every time when we wish to reuse a heap structure H_l, we split it into two heaps of equal size, and then use each of these two heaps in a recursive call. (The motivation for this splitting is similar to the motivation in *SEL*1 and *SEL*2 for generating two sets $os(C_j)$ and $pr(C_j)$, rather than just one combined set, whenever a clan C_j is generated.) If a heap H_l is handled in this way, then it will never contain more than $2h(r)/h(\lfloor \log r \rfloor)$ elements in it.

Our algorithm *SEL*3 to select the k-th smallest element in a heap is the following. It calls the recursive procedure *RSEL*3 with parameters *nil*, 1 and k. Procedure *RSEL*3(H, l, r) should identify a clan at level l of size $h(r)$ generated starting with the heap pointed to by H, if H is not *nil*, and using the original heap H_0 otherwise. At the time of the call to *RSEL*3, heap H should have at most $h(r)/h(\lfloor \log r \rfloor)$ elements in it. Procedure *RSEL*3(H, l, r) returns the representative element of the clan and a pointer to what remains of the heap used to build the clan. Algorithm *SEL*3 then searches the original heap H_0 to identify all elements in H_0 less than or equal to the representative of the clan, and then selects the k-th element within this set directly.

We next describe procedure *RSEL3*(H, l, r). If $r = 1$, then do the following. If $H = nil$, then take the minimum element x_1 of H_0 as the single element of the clan, and set H to be a heap containing x_2 and x_3 of H_0. Otherwise, if $H \neq nil$, extract the minimum element from heap H to be the single element of the clan, and insert into heap H the children of the clan's element in H_0. (Actually, the minimum element from H may already have been designated as a clan. If so, then it already has a heap H' containing its children in H_0. Thus H' can be merged into H.) In either case, let the clan's representative be its single element, and return the representative and a pointer to the heap.

Otherwise, if $r > 1$, do the following. First we initialize an auxiliary heap H_l as follows. If $H = nil$, then do the following. Call *RSEL3*$(H, l+1, \lfloor \log r \rfloor)$ recursively. Let the items returned by this call be *rep_elt* and *hptr*. Initialize heap H_l with *rep_elt*, and associate *hptr* with this entry. Otherwise, if $H \neq nil$, then initialize H_l to be H. After initializing auxiliary heap H_l, we do the following. For $h(r)/h(\lfloor \log r \rfloor)$ iterations, do the following. Perform an *extractmin* operation in H_l. Let *ext_elt* be the element extracted, and let *hptr* be associated with *ext_elt*. Split the heap pointed to by *hptr* arbitrarily into into two heaps of equal size, pointed to by *hptr*1 and *hptr*2. Call *RSEL3*$(hptr1, l+1, \lfloor \log r \rfloor)$ recursively, and let the items returned by this call be *rep_elt'* and *hptr'*. Insert *rep_elt'* into H_l and associate *hptr'* with this entry. Call *RSEL3*$(hptr2, l+1, \lfloor \log r \rfloor)$ recursively, and let the items returned by this call be *rep_elt''* and *hptr''*. Insert *rep_elt''* into H_l and associate *hptr''* with this entry. When the above loop terminates, let *last_elt* be the last element extracted from H_l. Return *last_elt* and a pointer to H_l. This concludes the description of *RSEL3*.

Lemma 3.1. Algorithm *SEL3* finds a k-th smallest element in a binary heap in $O(k\, 2^{\log^* k})$ time.

Proof. For correctness, we argue the following. For a given level l, any element is extracted from a heap at that level at most once. Thus, by induction on the level numbers, each clan generated at level l contains distinct elements, and any pair of clans at level l are disjoint. Thus the element returned to algorithm *SEL*3 can be no smaller than the k-th smallest element in the original heap H_0. Since algorithm *SEL*3 selects from the set of all elements less than or equal to the last element extracted, all elements less than or equal to the k-th smallest element will be included in this set. Thus the algorithm will return the k-th smallest element.

We analyze the time of algorithm *SEL*3 as follows. Consider a call to *RSEL*3 with parameter $r > 1$. The time to set up a heap and perform $h(r)/h(\lfloor \log r \rfloor)$ *extractmin* operations and $2h(r)/h(\lfloor \log r \rfloor)$ *insert* operations is $O(h(r))$. For each *extractmin* there are 2 recursive calls generating clans of size $h(\lfloor \log r \rfloor)$. Let $T(r)$ be the time used by procedure *RSEL*3 when it is called with last parameter equal to r. Then $T(r)$ is bounded by the recurrence:

$$\begin{aligned} T(1) &\leq c \\ T(r) &\leq c\,h(r) + 2h(r)/h(\lfloor \log r \rfloor) T(\lfloor \log r \rfloor) \end{aligned}$$

It follows that $T(r)$ is $O(h(r)\,2^{\log^* r})$. Since for $r > 1$, $h(r) \leq r + h(\lfloor \log r \rfloor)$, it follows that $T(r)$ is $O(r\,2^{\log^* r})$.

To complete the analysis, we show that the rank of the element returned by procedure *RSEL*3 to algorithm *SEL*3 is $O(k\,2^{\log^* k})$. Clearly this is true if $k = 1$, so consider $k > 1$. First observe that any representative of a clan is smaller than the elements in the associated heap for the clan. Next, observe that any element x in H_0 that is not in any clan created by *RSEL*3 has an ancestor y that is not in a clan but whose parent z comprises a clan of size 1. Element z will be the representative of its clan, and thus be in an associated heap. A recursive application of the first observation establishes that z is larger than the element returned to algorithm *SEL*3. But z is smaller than x, which implies that any element not in a clan created by *RSEL*3 is larger than the element returned to algorithm *SEL*3. The number of elements placed in clans at all levels while finding a clan of size $h(r)$ is at most $T(r)$. Thus the element returned to *SEL*3 has rank that is $O(k\,2^{\log^* k})$. The time to select the k smallest in the set of elements less than or equal to the returned element will be $O(k\,2^{\log^* k})$. □

In Figure 2 we consider a heap, the first five levels of which are shown in Figure 1, with $k = 2^8$. Note that for $r = 2^8 = 256$, $\lfloor \log r \rfloor = 8$, $\lfloor \log\log r \rfloor = 3$, and $\lfloor \log\log\log r \rfloor = 1$. Thus $h(\lfloor \log\log\log r \rfloor) = 1$, $h(\lfloor \log\log r \rfloor) = 3$, $h(\lfloor \log r \rfloor) = 9$, and $h(r) = 9 \cdot \lceil 256/9 \rceil = 9 \cdot 29 = 261$. Thus algorithm *SEL*3 will find a clan at level 1, consisting of 29 clans at level 2, each of which consists of 3 clans at level 3, each of which consists of 3 clans at level 4. The clans at level 4 contain single elements. The actual contents of the clans depend on the particular implementation of heap operations used.

We discuss the formation of the first clan $C_{2,1}$ at level 2. The initial clan at level 3 was $C_{3,1} = \{1, 2, 3\}$. Its associated heap contained $\{7, 10, 12, 4\}$, and its representative was 3. Thus the heap for forming $C_{2,1}$ contained $\{3\}$ at this point. When 3 was extracted from this heap, the associated heap for $C_{3,1}$ was split into $\{7, 10\}$ and $\{12, 4\}$, and clans $C_{3,2} = \{7, 9, 10\}$ and $C_{3,3} = \{4, 5, 6\}$ were formed from these heaps, resp. Their associated heaps contained $\{15, 16, 32, 18, 28\}$ and $\{12, 8, 11, 17, 19\}$, resp., and their representatives, 10 and 6, were inserted into the heap for forming $C_{2,1}$. When 6 was extracted from this heap, the associated heap for $C_{3,3}$ was split into $\{12, 8\}$ and $\{11, 17, 19\}$, and clans $C_{3,4} = \{8, 12, 13\}$ and $C_{3,5} = \{11, 14, 17\}$ were formed from these heaps, resp. Their representatives, 13 and 17, were inserted into the heap for forming $C_{2,1}$. A snapshot at this point in the algorithm is shown in Figure 2, with the clans of level 3 that are already included in $C_{2,1}$ shown in dashed edges, and the remaining three clans of level 3 shown in solid edges.

At the next step in the algorithm, 10 is extracted from the heap for forming $C_{2,1}$. The associated heap for $C_{3,4}$ is split into $\{15, 16\}$ and $\{32, 18, 28\}$, and clans $C_{3,6} = \{15, 22, 16\}$ and $C_{3,7} = \{32, 18, 28\}$ were formed from these heaps, resp. Their representatives, 22 and 32, were inserted into the heap for forming $C_{2,1}$. At this point, the formation of clan $C_{2,1}$ is complete, and consists of the union of clans $C_{3,1}$, $C_{3,2}$ and $C_{3,3}$. Its associated heap contains $\{22, 32, 13, 17\}$. A snapshot at this point in the algorithm is shown in Figure 3, with the clan $C_{2,1}$ shown in bold, and the four remaining clans of level 3 shown in solid edges. To proceed with finding the first clan at level 1, one would split the associated heap for $C_{2,1}$, and recursively find clans $C_{2,2}$ and $C_{2,3}$. Having illustrated at this point the salient features of our algorithm, we carry the example no further.

What keeps the time bound on algorithm *SEL3* comparatively large is that in finding a clan at level l, many clans at level $l+1$ can be created whose elements will not be included in any subsequent clan at level l. We thus reduce the number of elements in such clans at level l+1. To do this we have, for each level, clans of a number of different sizes. The actual number of sizes will be $(\log^* r)^2$, chosen to limit the total number of elements examined to $O(k)$. Let $f(r) = \lfloor (\lceil \log r \rceil / \log^* r)^2 \rfloor$. As in algorithm *SEL3*, we define a function $h(\cdot)$ that defines the sizes of clans in terms of a target size r. Again, this helps whenever the size of a clan at level $l+1$ does not evenly divide into a target size r. If $r = 1$, then $h(r) = 1$. Otherwise, $h(r) = h(f(r)) \lceil r / h(f(r)) \rceil$. The sizes of clans will be $i\, h(f(r))$, for $i = 1, 2, \cdots, (\log^* r)^2$.

Whenever a new clan at level $l+1$ is created, it is created at the smallest size. When a representative of a clan at level $l+1$ is extracted from heap H_l, and the clan is not of full size, the growth process is resumed for that clan, using the associated heap at level $l+1$, up to the next larger size. The new representative of this clan at level $l+1$ is then reinserted into H_l, and the count of elements so far in the clan being built at level l is incremented by $h(f(r))$. Thus when we perform an *extractmin* in H_l, giving element x, we assign to the clan being built at level l the $h(f(r))$ elements less than or equal to x that were most recently assigned to the clan at level $l+1$ whose representative was just extracted. When a representative of a full-size clan is extracted from heap H_l, this element is reported as extracted, and the count of elements so far is incremented by $h(f(r))$.

Our algorithm *SEL4* to select the k-th smallest element in a heap is the following. It calls the recursive procedure *RSEL4* with parameters nil, 1 and k. Procedure *RSEL4*(H, l, r) should identify a clan at level l of size $h(r)$ generated starting with the heap H, if H is not nil, and using the original heap H_0 otherwise. Procedure *RSEL4*(H, l, r) returns the representative element of the clan and a pointer to a heap representing what remains of the heap used to build the clan. After the return from *RSEL4*, algorithm *SEL4* searches the original heap H_0 to identify all elements in H_0 less than or equal to the representative of the clan, and then selects the k-th element within this set directly.

We next describe procedure *RSEL4*(H, l, r). If $r = 1$, then do the following. If $H = nil$, then take the minimum element x_1 of H_0 as the single element of the clan, and set H to be a heap containing x_2 and x_3 of H_0. Otherwise, if $H \neq nil$, extract the minimum element from heap H to be the single element of the clan, and insert into heap H the children of the clan's element in H_0. (Again, the minimum element from H may already have been designated as a clan. If so, then it already has a heap H' containing its children in H_0. Thus H' can be merged into H.) In either case, let the clan's representative be its single element, and return the representative and a pointer to the heap.

Otherwise, if $r > 1$, do the following. First we initialize an auxiliary heap H_l as follows. If $H = nil$, then do the following. Call *RSEL4*$(H, l{+}1, f(r))$ recursively. Let the items returned by this call be *rep_elt* and *hptr*. Initialize heap H_l with *rep_elt*, and associate *hptr* and a clan category of 1 with this entry. Otherwise, if $H \neq nil$, then initialize H_l to be H. After initializing auxiliary heap H_l, we do the following. For $h(r)/h(f(r))$ iterations, do the following. Perform an *extractmin* operation in H_l. Let *ext_elt* be the element extracted, and let *hptr* and *clan_cat* be associated with *ext_elt*. If $clan\_cat < (\log^* r)^2$, then call *RSEL4*$(hptr, l+1, f(r))$ recursively, and let the items returned by this call be *rep_elt'* and *hptr'*. Insert *rep_elt'* into H_l and associate *hptr'* and $1+clan\_cat$ with this entry. Otherwise, when $clan\_cat = (\log^* r)^2$, do the following. Split the heap pointed at by *hptr* arbitrarily into two heaps of approximately equal size pointed at by *hptr*1 and *hptr*2. Call *RSEL4*$(hptr1, l+1, f(r))$ recursively, and let the items returned by this call be *rep_elt'* and *hptr'*. Insert *rep_elt'* into H_l and associate *hptr'* and 1 with this entry. Call *RSEL4*$(hptr2, l+1, f(r))$ recursively, and let the items returned by this call be *rep_elt''* and *hptr''*. Insert *rep_elt''* into H_l and associate *hptr''* and 1 with this entry. When the above loop terminates, let *last_elt* be the last element extracted from H_l. Return *last_elt* and a pointer to H_l. This concludes the description of *RSEL4*.

Theorem 3.2. Algorithm *SEL4* finds a k-th smallest element in a binary heap in $O(k)$ time.

Proof. For correctness, note that $\lceil \log r \rceil \geq \log^* r$ whenever $r > 1$, so that $f(r)$ is positive for $r > 1$. The rest of the discussion concerning correctness is similar to that in the proof of Lemma 3.1.

We analyze the time of algorithm *SEL4* as follows. Consider a call to *RSEL4* with parameter $r > 1$. The time to set up a heap and perform $h(r)/h(f(r))$ *extractmin* and *insert* operations is $O((h(r)/h(f(r))\lceil \log r \rceil)$, which is $O(h(r)(\log^* r)^2/\lceil \log r \rceil)$. Since 2 recursive calls are made only when $clan\_cat = (\log^* r)^2$ and 1 call is made the rest of the time, we charge each *extractmin* with $1 + 1/(\log^* r)^2$ recursive calls. Let $T(r)$

be the time charged to procedure *RSELA* when it is called with last parameter equal to r. Then $T(r)$ is bounded by the recurrence:

$$\begin{aligned} T(1) &\leq c \\ T(r) &\leq c\,h(r)\frac{(\log^* r)^2}{\lceil \log r \rceil} + \left(1 + \frac{1}{(\log^* r)^2}\right)\frac{h(r)}{h(f(r))}\,T(f(r)) \end{aligned}$$

We claim that

$$T(r) \leq c' h(r) \prod_{i=1}^{2\log^* r} (1 + \frac{4}{i^2})$$

for an appropriate constant c'. The proof is by induction on r. The basis is for $r \leq 2$. For $r = 1$, $T(1) \leq c$, so that the claim is satisfied if $c \leq c' \cdot 1(1+4/1)(1+4/4) = 10c'$. For $r = 2$, $T(2) \leq c \cdot 2 \cdot 1/1 + (1 + 1/1)\,2/1 \cdot c = 3c$, so that the claim is satisfied if $3c \leq c' \cdot 2\,(1+4/1)(1+4/4) = 20c'$.

For $r > 2$, assume that the claim is true for $r' < r$. Note that for $r > 2$, it can be verified that $\log^* r > \log^* f(r)$. Then

$$T(r) \leq c\,h(r)\frac{(\log^* r)^2}{\lceil \log r \rceil} + \left(1 + \frac{1}{(\log^* r)^2}\right)\frac{h(r)}{h(f(r))} T(f(r))$$

$$\leq c\,h(r)\frac{(\log^* r)^2}{\lceil \log r \rceil} + \left(1 + \frac{4}{(2\log^* r)^2}\right)\frac{h(r)}{h(f(r))}\,c' h(f(r)) \prod_{i=1}^{2\log^* f(r)} (1 + \frac{4}{i^2})$$

$$\leq c\,h(r)\frac{(\log^* r)^2}{\lceil \log r \rceil} + \left(1 + \frac{4}{(2\log^* r)^2}\right) c' h(r) \prod_{i=1}^{2\log^* r - 2} (1 + \frac{4}{i^2})$$

$$= c\,h(r)\frac{(\log^* r)^2}{\lceil \log r \rceil} + \left(c' h(r) \prod_{i=1}^{2\log^* r} (1 + \frac{4}{i^2})\right) / \left(1 + \frac{4}{(2\log^* r - 1)^2}\right)$$

The claim will hold if

$$c\,h(r)\frac{(\log^* r)^2}{\lceil \log r \rceil} + \left(c' h(r) \prod_{i=1}^{2\log^* r} (1 + \frac{4}{i^2})\right) / \left(1 + \frac{4}{(2\log^* r - 1)^2}\right) \leq c' h(r) \prod_{i=1}^{2\log^* r} (1 + \frac{4}{i^2})$$

which will hold if

$$c\,h(r)\frac{(\log^* r)^2}{\lceil \log r \rceil}\left(1 + \frac{4}{(2\log^* r - 1)^2}\right) \leq \frac{4}{(2\log^* r - 1)^2}\, c' h(r) \prod_{i=1}^{2\log^* r} (1 + \frac{4}{i^2})$$

which will hold if

$$\frac{(\log^* r)^2((2\log^* r - 1)^2 + 4)}{4\lceil \log r\rceil \prod_{i=1}^{2\log^* r}(1+\frac{4}{i^2})}\, c \le c'$$

The lefthand side takes on its maximum in the range $r = 17$ to $r = 32$, for which the above reduces to $1.596\, c \le c'$. The claim then follows by induction, with c' chosen to be 1.596c.

From the claim it follows that $T(r)$ is $O(r)$, since for $r > 1$, $h(r) < 2r$, and $\prod_{i=1}^{\infty}(1+\frac{4}{i^2})$ is a constant.

Upon its return to algorithm *SEL4*, procedure *RSEL4* has identified an element that is no smaller than the k-th smallest element in H_0. By reasoning similar to that in the proof of Lemma 3.1, the number of elements placed in clans at all levels while finding a clan of size $h(r)$ is at most $T(r)$. Thus the element returned to *SEL4* has rank that is $O(k)$. The time to select the k smallest in the set of elements less than or equal to the returned element will be $O(k)$. □

We note that the constant implied by the analysis in the above proof is rather large. We would not be surprised if this value could be reduced substantially by a more careful analysis.

4. Discussion

Rajamani Sundar has pointed out that the information theory bound is not tight for a t-ary heap, if t is assumed to be variable [S]. The number of t-ary trees with k nodes is $\frac{1}{(t-1)k+1}\binom{tk}{k}$. (See exercise 11, page 396 of [K]). The best lower bound that this gives is $\Omega(k \log t)$. However, a simple adversary argument gives a bound of $\Omega(kt)$ for $k > 1$: Assume that the $k-1$ smallest elements are already identified. An adversary can force any algorithm to examine each of the $(t-1)(k-1)+1$ remaining children of the $k-1$ smallest elements. As for the upper bound, our methods can easily be adapted to t-ary heaps, increasing the time for each step by a factor of t.

Acknowledgement.

I would like to thank Rajamani Sundar and a referee for their careful reading of the manuscript and many helpful comments. I would also like to thank Mike Atallah and Ian Munro for bringing some references to my attention.

References

[AK] M. J. Atallah and S. R. Kosaraju. An adversary-based lower bound for sorting. *Info. Proc. Lett.*, 13:55–57, 1981.

[BJ] S. W. Bent and J. John. Finding the median requires $2n$ comparisons. In *Proc. 17th ACM Symp. on Theory of Computing*, pages 213–216, 1985.

[BFPRT] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7:448–461, 1972.

[B1] P. Brucker. Network flows in trees and knapsack problems with nested constraints. In H. J. Schneider and H. Gottler, editors, *Proc. 8th Conf. on Graph Theoretic Concepts in Computer Science*, pages 25–35, Munich, 1982. Hanser.

[B2] P. Brucker. An $O(n \log n)$ algorithm for the minimum cost flow problem in trees. In G. Hammer and D. Pallaschke, editors, *Proc. 8th Symp. on Operations Research: Selected Topics in Operations Research and Mathematical Economics*, pages 299–306, Berlin, 1984. Springer. Lecture Notes in Economics, Vol. 226.

[CSSS] R. Cole, J. S. Salowe, W. L. Steiger, and E. Szemeredi. An optimal-time algorithm for slope selection. *SIAM J. Comput.*, 18:792–810, 1989.

[F1] G. N. Frederickson. The information theory bound is tight for selection in a heap. In *Proc. 22nd ACM Symp. on Theory of Computing*, pages 26–33, 1990.

[F2] G. N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. manuscript, March 1991.

[FJ1] G. N. Frederickson and D. B. Johnson. The complexity of selection and ranking in $X+Y$ and matrices with sorted columns. *J. Comput. Syst. Sci.*, 24:197–208, 1982.

[FJ2] G. N. Frederickson and D. B. Johnson. Generalized selection and ranking: sorted matrices. *SIAM J. on Computing*, 13:14–30, 1984.

[IK] T. Ibaraki and N. Katoh. *Resource Allocation Problems: Algorithmic Approaches.* MIT Press, Cambridge, Massachusetts, 1988.

[K] D. E. Knuth. *The Art of Computer Programming, Vol. 1, second edition.* Addison-Wesley, Reading, Massachusetts, 1973.

[L1] E. L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Sci.*, 18:401–405, 1972.

[L2] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, New York, 1976.

[M] K. M. Mjelde. Discrete resource allocation with tree constraints by an incremental method. *European Journal of Operational Research*, 17:393–400, 1984.

[SPP] A. Schōnhage, M. Paterson, and N. Pippenger. Finding the median. *J. Comput. Syst. Sci.*, 13:184–199, 1976.

[S] R. Sundar. email communication, 1989.

[WN] M. A. Weiss and J. K. Navlakha. The distribution of keys in a binary heap. In J.-R. Sack F. Dehne and N. Santoro, editors, *Proc. Workshop on Algorithms and Data Structures*, pages 213–216. Springer-Verlag, 1989. LNCS, Vol. 382.

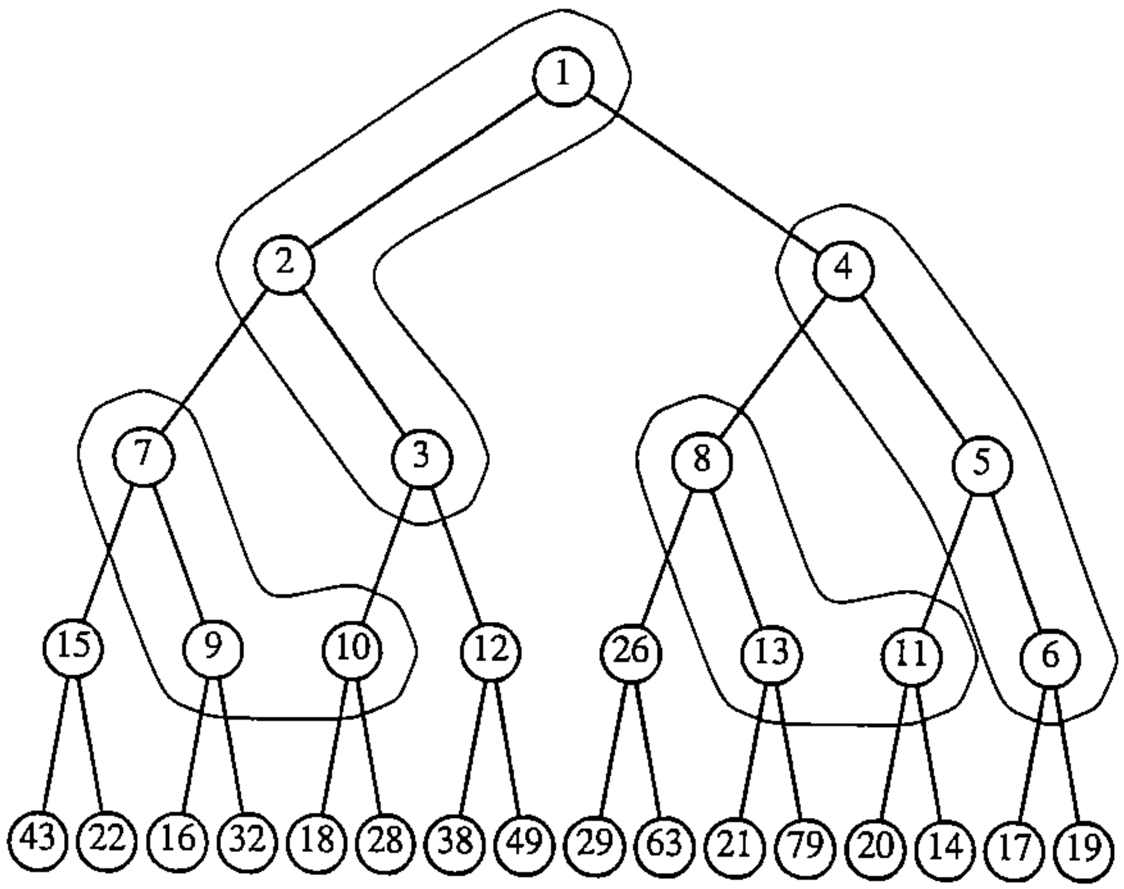

Figure 1. The first five levels of a large heap, plus some clans of size 3.

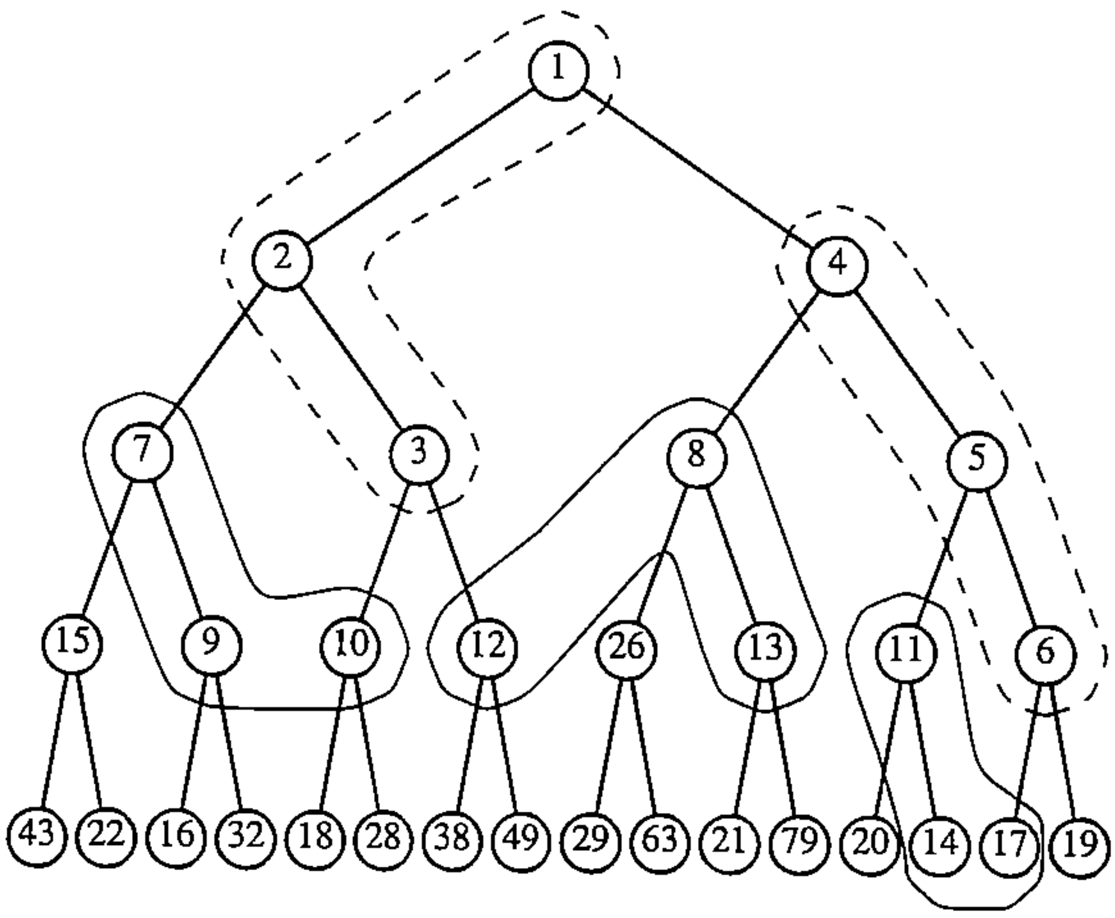

Figure 2. A snapshot just before the first clan at level 2 is generated by *RSEL*3.

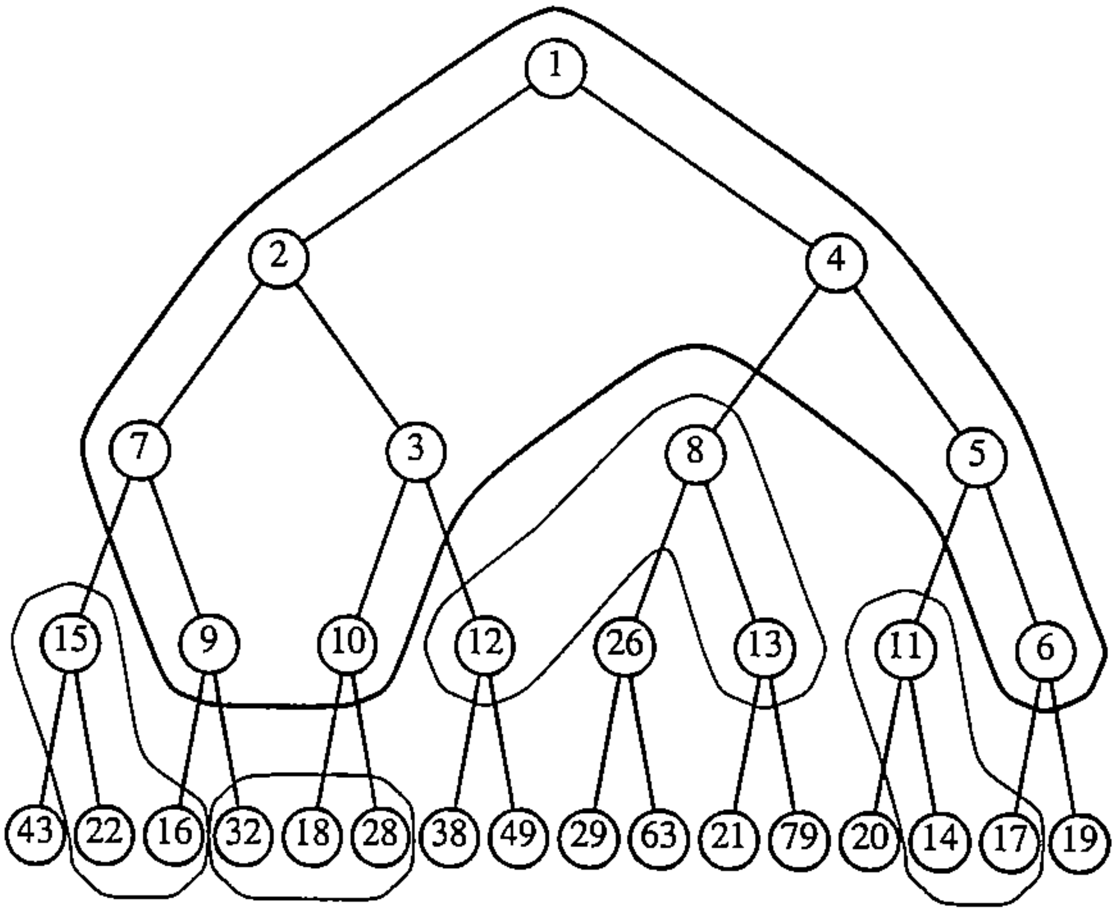

Figure 3. A snapshot just after the first clan at level 2 is generated by *RSEL*3.