Sorting and Searching II

Rana Barua

Visiting Scientist, IAI, TCG CREST Kolkata

1 Search

Before we embark onto some basic search algorithms, let us consider the following fundamental operations on sets

1.1 Fundamental Operations on Sets

We shall consider the following fundanmental operations on sets.

1. MEMBER(a.S): Determine whether a is a member of the set S; if so print "ÿes", else print "no".
2. INSERT(a, S): Replace the set S by $S \cup \{a\}$.
3. DELETE(a, S): Replace the set S by $S - \{a\}$.
4. UNION(S_1, S_2, S_3): Replace sets S_1 and S_2 by $S_3 = S_1 \cup S_2$. We assume that S_1 and S_2 are disjoint when this operation is performed.
5. FIND(a): Print the name of the set of which a is currently a member. If A is in more than one set then it's undefined.
6. SPLIT(a, S): Here we assume that S is linearly ordered. This operation partitions the set S into two sets S_1 and S_2 such that
$$S_1 = \{b \in S : b \leq a\},$$
$$S_2 = \{b \in S : b > a\}.$$
7. MIN(S): Prints the smallest element of the set S.

1.2 Search Trees and Algorithms

Binary Search: We are given a set S with n elements drawn from a large universal set. We are to process a sequence σ consisting only of MEMBER instructions. If there is a linear order$\leq$ on S, a solution is to use binary search.

Here we store the elements of S in an array A. Next we sort the array so that

$$A[1] < A[2] < \ldots < A[n].$$

Then to determine whether an element a is in S, we compare a with b stored in location $\lfloor \frac{n+1}{2} \rfloor$. If $a = b$, we halt and answer "ÿes". Otherwise, we repeat the process on the first half of the array if $a < b$, or on the last half if $a > b$. By repeatedly splitting the search area in half, we need not make more than $\lceil \log(n+1) \rceil$ comparisons to find a or to determine it is not in S.

Procedure: $SEARCH(a, f, l)$

Looks for an element a in locations $f, f+1, \ldots, l$ of the array A. To determine wheter a is in S, we call $SEARCH(a, 1, n)$.

if $f > l$ **then return** "no";
else
  if $a = A[\lfloor \frac{f+l}{2} \rfloor]$ **then return** "yes";
  else
    if $a < A[\lfloor \frac{f+l}{2} \rfloor]$ **then return** $SEARCH(a, f, \lfloor \frac{l+f}{2} \rfloor - 1)$;
    else return $SEARCH(a, \lfloor \frac{f+l}{2} \rfloor + 1, l)$.

Exercise 1. Show that the procedure $SEARCH$ makes at most $\lceil \log(n+1) \rceil$ comparisons for a set with n elements.

Binary Search Trees. Suppose we have a set S in which elements are being inserted and from which elements are being deleted. From time to time we may want to know the smallest element currently in S. We assume that the elements being added to S comes from a large universal set that is linearly ordered. This problem can be abstracted as processing a sequence of INSERT, DELETE, MEMBER and MIN instructions. A data structure that is suitable for all four instructions is *the binary search tree.*

Definition 1. *A binary search tree for a set S is a labeled binary tree in which each vertex v is labeled by an element $l(v) \in S$ such that*

1. *For each vertex u in the left subtree of v, $l(u) < l(v)$.*
2. *For each vertex u in the right subtree of v, $l(u) > l(v)$.*
3. *For each element $a \in S$, there is a unique vertex v such that $l(v) = a$.*

Searching a Binary Tree: To determine whether an element a is in a set S represented by a binary search tree T, we first compare a with the label of the root. If they are equal, then clearly $a \in S$. If a is less than the label of the root, then we search the left subtree of the root, if it exists. If a is greater than the label of the root then we search the right subtree of the root. If a is in the tree, then eventually a will be located. Otherwise, the process will terminate when we will have to search a non-existing subtree.

Procedure $SEARCH(a, v)$
Input: An element a and the root of the search tree for a set S.
Output: "yes" if $a \in S$; elese "no".

```
1.      if a = l(v) then return "yes";
        else
2.          if a < l(v) then;
3.              if v has a left son w then return SEARCH(a, w);
4.              else return "no";
            else
5.              if v has a right child w then return SEARCH(a, w);
6.              else return "no".
```

Executing the Fundamental Operations:

1. $MEMBER(a, S)$: Let v be the root of the search tree representing S. Then call $SEARCH(a, S)$.
2. $INSERT(a, S)$.
   - If the tree is empty, we create a root with label a.
   - If the tree is non-empty and the lement to be inserted is not found in the tree, then the procedure $SEARCH$ fails to find a child either at line 3 or at line 5. Instaed of returning "no" at line 4 or 6, respectively, a new vertex with label a is attached where the missing child belongs.

- $DELETE(a, S)$: Suppose the element a to be deleted is found at vertex v. The following three cases can occur.
  (a) Vertex v is a leaf. In this case remove vertex v from the tree.
  (b) Vertex v has exactly one child. In this case, make the parent of v the parent of the child. If v is the root, then make the child of v the new root.
  (c) Vertex v has two children.
  Find the largest element b in the left subtree. Recursively remove the vertex containing b from the subtree. Then set the label of v to be b.

3. MIN : The smallest element in a binary search tree T is found by the following path $v_0, v_1, \ldots, v_p$, where v_0 is the root, v_i is the left child of v_{i-1} and v_p has no left child. The label on v_p is the smallest element of T.

1.3 The Union-Find Problem

We shall present a data structure consisting of a forest of trees to represent a collection of sets. This data structure will allow the processing of $O(n)UNION$ and FIND intruction s in *alomost* linear time.

Each set A is represented by a rooted tree T_A, where the elements of A correspond to the vertices of T_A. The name of the set is attached to the root of the tree. An intruction of the form $UNION(A, B, C)$ can be executed by making the root of T_A a child of the root of T_B and changing the root of T_B to C. An instruction of the form $FIND(i)$ can be executed by locating the vertex representing the element i in some tree T in the forest and then traversing the path from this vertex to the root of the tree T, where we find the name of the set containing i.

With such a scheme, the cost of merging two trees is a constant. The cost of a $FIND(i)$ is of the order of the length of the path from vertex i to its root. Such a path can have length $n-1$. Thus the cost of executing $n-1UNION$ instructions followed by $nFIND$ intructions could be as high as $O(n^2)$. For example, consider the cost of the following sequence.

$$UNION(1, 2, 2),$$

$$UNION(2, 3, 3),$$

$$UNION(3, 4, 4),$$

$$\vdots$$

$$UNION(n-1, n, n),$$

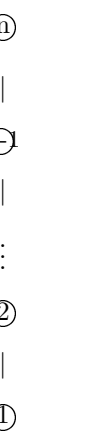
$$FIND(1),$$

$$FIND(2),$$

$$\vdots$$

$$FIND(n).$$

Executing the FIND instructions gives us the following tree.



Thus the cost of $FIND(1)$ is $n-1$, that of $FIND(2)$ is $n-2$ and so on. So the total cost is

$$\sum_{i=1}^{n-1} i = O(n^2).$$

The cost can be reduced if the tree can be kept balanced. To achieve this we keeep a count of the number of vertices in each tree and, while merging two trees, always to attach the smaller tree to the root of the larger tree.

Lemma 1. *If in executing each UNION instruction, the root of the tree with fewer vertices is made a child of the root of the larger tree, then a tree in the forest with height at least h will have at least* 2^h *vertices.*

Proof. By induction on h.
Let T be a tree with *fewest* number of vertices. Then T must have been obtained by merging two trees T_1 and T_2, where T_1 has height $h-1$ and has no more vertices than T_2. By induction hypothesis, T_1 has at leat 2^{h-1} vertices and hence, T_2 has at least 2^{h-1} vertices. This implies that T has at least $2^{h-1} + 2^{h-1} = 2^h$ vertices.

Complexity: Consider the worst-case execution time for a sequence of $nUNION$ and FIND instructions. Using the forest data structure, with the modification that in aUNION the root of the smaller tree becomes a child of the root of the larger one. By Lemma 1, no tree can have height greater than $\log n$.(Why?) Hence the execution of $O(n)$ UNION and FIND instructions takes at most $O(n \log n)$ time.

Remark 1. This bound is tight and can not be improved.

Further modification: We introduce another modification to the algorithm called *path compression*. We shall try to reduce the cost of FIND intructions. Each time a $FIND(i)$ instruction is executed, we traverse the path from vertex i to its root r. Let $i, v_1, v_2, \ldots, v_k, r$ be the vertices on this path. We then make each of $i, v_1, v_2, \ldots, v_{k-1}$ a child of the root r.

The complete tree-merging algorithm for the UNION-FIND problem, including path compression is the following.

Fast Disjoint-Set Union Algorithm

Input: A sequence σ of UNION and FIND instructions on a co;;ection of sets whose elements are integers in $[1, n]$. Theb set names are also integers from 1b to n. Initially, element i is by itself in a set named i.

Output: The sequence of responses to the FIND instructions in σ. The response to each FIND is to be produced before looking at the next instruction in σ.

*Method:*The algorithm consists of three parts;

1. the initialization,
2. the response to a FIND, and
3. the response to a UNION.

1. **Initilization:** For each element $i, 1 \leq i \leq n$, we create a vertex v_i. We set $COUNT[v_i] = 1, NAME[v_i] = i$ and $FATHER[v_i] = 0$. Initially, each vertex is a tree by itself. In order to locate the root of set i, we create an array ROOT with $ROOT[i]$ pointing to v_i. To locate the vertex for element i, we create an array ELEMENT, initially with $ELEMENT[i] = v_i$.
2. **Executing FIND(i):**
   Starting at $ELEMENT[i]$, we follow the path from the root of the tree, making a list of alll the vertices encountered. At the root the name of the set is printed, and each vertex on the path traversed is made a child of the root.

begin
make $LIST$ empty;
$v \leftarrow ELEMENT[i]$;
while $FATHER[v] \neq 0$ **do**
begin
add v to $LIST$;
$v \leftarrow FATHER[v]$
end
Comment: v is now the root;
PRINT $NAME[v]$;
for each w on $LIST$ **do** $FATHER[w] \leftarrow v$
end

3. **Executing UNION(i,j,k)**:
Via the array ROOT, we find the roots of the trees representing sets i and j. We then make the root of the smaller tree a child of the root of the larger tree.

begin
wlg assume $COUNT[ROOT[i]] \leq COUNT[ROOT[j]]$
otherwise interchange i and j;
begin
$LARGE \leftarrow ROOT[j]$;
$SMALL \leftarrow ROOT[i]$;
$FATHER[SMALL] \leftarrow LARGE$;
$COUNT[LARGE] \leftarrow COUNT[LSARGE] + COUNT[SMALL]$;
$NAME[LARGE] \leftarrow k$;
$ROOT[k] \leftarrow LARGE$
end
end

Complexity: We shall show that path compression speeds up the algorithm considerably. We introduce two funcytions F and G as follows.

$$F(0) = 1,$$

$$F(i) = 2^{F(i-1)}, i > 0.$$

$F(1) = 2, F(2) = 2^2 = 4, F(3) = 2^4 = 16, F(4) = 2^{16} = 65536, F(5) = 2^{65536}$.
$G(n) =$ smallest k such that $F(k) \geq n$.
Thus $G(0) = G(1) = 1, G(2) = 1, G(3) = G(4) = 2$. In fact, for $F(k-1) < n \leq F(k), G(n) = k$. Thus, for $n \leq 2^{65536}, G(n) \leq 5$. Thus for all practical purposes, $nG(n)$ is linear.

We now claim that the above algorithm will execute a sequence σ of $cnUNION$ and $FIND$ insrructions in at most $c'nG(n)$ time, where c' is a constant depending on c.

Without loss of generality we assume that the execution of a $UNION$ instruction takes one "time unit" and the execution of $FIND($ takes a number of time units proportion to the number of vertices on the path from i to the root of the tree containing this vertex.

Definition 2. *We define rank of a vetex with respect to σ as follows.*

1. *Delete the FIND instructions from σ.*
2. *Execute the resulting sequence σ' of UNION instructions.*
3. *The rank of a vetex is the height of v in the resulting forest.*

Lemma 2. *There are atmost $n/2^r$ vertices of rank r.*

Proof. Let R be the number of vertices of rank r. Assume $R > n/2^r$.

Now, by Lemma 1, each vertex of rank r has at least 2^r descendants. Also any two distinct vertices of rank r in a forest has disjoint sets of descendants. Hence the number of vertices is at least $R.2^r > n$. a contradiction. Hence $R \leq n/2^r$. □

Corollary 1. *No vertex has rank greater that* $\log n$.

Proof. The number of vertices of rank $\log n + 1 \leq \frac{n}{2^{\log n+1}} = \frac{1}{2}$. Hence there is no vertex of rank grater than $\log n$.

Lemma 3. *If at some point in the execution of σ, w is a proper descendant of v, then rank of w is less than the rank of v.*

Proof. If at point during the execution of σ, w is made a decendant of v, then w will remain a descendant v in the forest obtained from the execution of σ'. Hence the height of w must be less than that of v. So rank of w kis less than rank of v. □

We now partion the ranks onto groups. We put rank r in the group $G(r)$. Thus if $F(k-1) < r \leq F(k)$, then r is put in group k. For $n > 1$, the largest possible rank, $\lfloor \log n \rfloor$, is put in group $G(\lfloor \log n \rfloor) \leq G(n) - 1$.

Computing the cost:

Consider the cost of executing a sequence σ of $cnUNION$ and FIND instructions. Since each $UNION$ instruction can be executed at the cost of one time unit, all UNION instructions in σ can be executed in $O(n)$ time. We now bound the cost of executing all FIND instructions. The costb of executing a single FIND is allocated between the FIND instructions itself and certain vertices on the path in the forest data structure which are actually moved. The total cost is computed by summing over all FIND insructions, the cost allocated to them; and then summing the cost assigned to the vertices, over all vertices in the forest.

We charge for the instruction $FIND(i)$ as follows. Let v be a vertex on the path fom i to the root of the tree containing i.

1. If v is the root or if $FATHER[v]$ is in different rank group from v, then charge one time unit to the FIND instuction itself.
2. If both v and its father are in the same rank group, then charge one time unit to v

By Lemma 3, vertices going up the path are monotonically increasing in rank, and since there are $G(n)$ rank group, no FIND instruction is charged more than $G(n)$ time unit under Rule 1.. If Rule 2 applies, verex v will be moved and made a child of a higher rank than its previous father If vertex v is in rank group $g > 0$, then v can be moved and charged at most $F(g) - F(g-1)$ times before it acquires a father in a different rank group.

To obtain an upper bound on the charges made to the vertices themselves, we multiply the maximum charge to any vertex in a rank group by the number of vertices in that group and sum over all rank groups.

Let $N(g)$ be the number of vertices in rank group $g > 0$. Then by Lemma 2,

$$N(g) \leq \sum_{r=F(g-1)+1}^{F(g)} n/2^r$$
$$\leq \frac{n}{2^{F(g-1)+1}}\{1 + \frac{1}{2} + \ldots\}$$
$$\leq \frac{n}{2^{F(g-1)}} = \frac{n}{F(g)}.$$

Thus the maximun charge to a rank group $g = N(g)(F(g) - F(g-1)) \leq n$. Hence the maximun charge to all vertices is $nG(n)$.

Hence the total time required to process cn $FIND$ instructions is at most $cnG(n)$ charged to the FIND insructions and at most $nG(n)$ charged to the vertices. Thus we have,

Theorem 1. *Let c be a constant. Then there exist another constant c' depending on c such that the above algorithm will execute a sequence σ of cn $UNION$ and FIND instructions on n elements in at most $c'nG(n)$ time units.*

2 Application of Union-Find:

2.1 Kruskal's Algorithm

Definition 3. *Let $G = (V, E)$ be a connected graph. A subgraph $S = (V, T)$ is called a spanning tree for G if T is a tree. Let $c(,)$ be a cost function on the edges E of G. Then S is called a minimum-cost spanning treee if the cost of S i.e. the sum of costs on its edges T is as small as possible.*

To describe Kruskal's algorithm, we need the following Lemma.

Lemma 4. *Let $G = (V, E)$ be a connected, undirected graph and $c(,)$ a cost function on its edges. Let $(V_1, T_1), \ldots, (V_k, T_k)$ be any spanning forest with $k > 1$. Let $T = \cup_{i=1}^{k}$. Suppose $e = v, w$ is an edge of lowest cost in $E - T$ such that $v \in V_1$ and $w \notin V_1$. Then there is a spanning tree which includes $T \cup \{e\}$ and is of as low a cost as any spanning tree that includes T.*

Proof. Suppose the result does not hold. Then there is a spanning tree $S' = (V, T')$ for G such that T' includes T but not e and that cost of S' is lower than any spanning tree that includes $T \cup \{e\}$.

Now addition of e to S' forms a unique cycle. Since $v \in V_1$ and $w \notin V_1$, that cycle must include an edge $e' = (v', w') \neq e$ such that $v' \in V_1$ and $w' \notin V_1$. Now by hypothesis,

$$c(e) \leq c(e').$$

Now consider the graph S formed by adding e to S' and deleting e' from S'. Clearly S has no cycle, since e' has been removed from the only cycle. Thus S is a spanning tree. Since $c(e) \leq c(e')$, S is no more costly than S'. But S containg $T \cup \{e\}$ This contradicts the minimality of S'. □

Kruskal's Algorithm: We now describe Kruskal's algorithm. The algorithm maintains a collection VS of disjoint sets of vertices. Each set W in VS represents a connected set of vertices that forms a tree in the spanning forest represented by VS. Edges are chosen from E in order of increasing cost. We consider each edge $\{v, w\}$ in turn. If v and s are already in the same set in VS then we discard the edge. If v and s are in distinct sets W_1 and W_2, we merge W_1 and W_2 into a single set and add $\{v, w\}$ to T, the set of edges in the final spanning tree. By Lemma 4, at least one minimum cost spanning tree for G will contain this edge.

Minimum-cost Spanning Tree:
Input: An undirected graph $G = (V, E)$ with cost function $c(,)$ on its edges.
Output $S = (V, T)$, a minimum-cost spanning tree for G.

```
        begin
1.          T ← φ
2.          VS ← φ
3.          construct a priority queque Q containing all edges of E
4.       for each vertex v ∈ V do add {v} to VS
5.       while |VS| > 1 do
             begin
6.             choose {v, w}, an edge in Q of lowest cost
7.             delete {v, w} from Q
8.             if v and w are in different sets W1 and W2 in VS then
                 begin
9.                   replace W1 and W2 in VS by W1 ∪ W2;
10.                  add {v, w} to T.
                 end
           end
        end
```

Remark 2. One can use the above fast disjoit set union algorithm for Lines 8 and 9 which, for all practical purposes, can be executed in linear time. Thus the cost of Kruskal' algorithm depends on the cost of choosing an edge of lowest cost at Line 6 and the number of times the while loop is executed.

REFERENCES

1. A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms.
2. E. Horowitz, S. Sahni, Computer Algorithms
3. U. Manber, Introduction to Algorithms–A Creative Approach.