ESO211 (Data Structures and Algorithms) Lectures 32

Shashank K Mehta

In this lecture we will discuss a datastructure which stores a partition of a fixed set U also called universal set, i.e., it will store all the subsets in a partition of U. It will support two operations (i) Find: given an element, it will tell which set does it belong to, (ii) Union or Disjoint-Union: given any two subsets, it will compute the union and save in the datastructure and delete the two parent sets, hence structure will continue to store a partition.

It may be noted that the subsets will be identified by some local name. Hence the result of a Find operation may not be significant (as it will return an internal name of the set) but Find can be useful in deciding whether given two elements belong to the same set or not.

We will describe the datastructure assuming that the universal set is $U = \{1, 2, \dots, n\}$. The datastructure has three parts: Element array E, a 2D array S called set-array, and linked lists (one for each subset). There will be one node for each element of the universal set. The node of element j will contain j in *Element* field, the identity/index/internal-name of the subset (which contains j) in *Set* field, and will store a pointer to the next node in the *Next* field. So the node (record) of each element is like

```
type
        elementNode = record
                Element: int
                Set:int
                Next:pointer to elementNode
        end
```

Array E will store pointers. $E[j]$ will point to the node of element j. The nodes of the elements in set i will be stored in a linked list, in no particular order. $S[i, 0]$ will point to the first node of that linked list and $S[i, 1]$ will be integer-type and it will store the cardinality of the subset/linked list. To summarize, if element j is in set i, then the node pointed by the pointer $E[j]$ will be in the list pointed by $S[i, 0]$; $E[j] \cdot Element = j$; and $E[j] \cdot Set = i$.

Find Operation

The find operation, $Find(j)$, will return $Element[j] \cdot Set$. It will cost $O(1)$ time.

Union Operation

Given two sets i_1 and i_2 the union operation involves (i) change the set identity of each element of the smaller linked list and make it same as that of the larger set, (ii) join the linked lists into one, (iii) make the smaller set pointer null and the larger size set pointer point to the combined linked list, (iv) correct the set size entries. Following code described the union operation.

The cost of a single union operation can be $O(n)$ in the worst case.

```
if $S[i_1, 1] \leq S[i_2, 1]$ then
    $small := i_1$;
    $large := i_2$;
else
    $small := i_2$;
    $large := i_1$;
end
$p := S[small, 0]$;
if $S[small, 1] > 0$ then
    while $p \neq null$ do
        $p \cdot Set := large$;
        $n := p$;
        $p := p \cdot Next$;
    end
    $n \cdot Next := S[large, 0]$;
    $S[large, 0] := S[small, 0]$;
    $S[large, 1] := S[large, 1] + S[small, 1]$;
    $S[small, 0] := null$;
    $S[small, 1] := 0$;
end
```

Algorithm 1: $Union i_1, i_2$

Amortized Analysis

Suppose in a programme execution α Find operations and β Union operations are performed on this datastructure. Clearly the Finds will cost $O(\alpha)$. But we will do an amortized analysis for the union operations. Observe that even if initially all sets are singletons, there cannot be more than $n-1$ disjoint-unions. To find an upperbound for the combined cost of all the unions, observe that each union operation has two components: cost of changing the set identity of the elements of the smaller sets, cost of relinking, correcting the set size parameters, and setting the Set pointers. The cost of all parts except the first part will be $O(1)$ for a union operation. Hence total cost will be $O(\beta)$ not including the cost of all the set identity change.

Suppose each time the set identity of an element changes, it gives away Re 1 to some central pool. Clearly the total number of rupees collected at the end of the programme execution will be the cost of changing all the set identities of all elements. Observe that after the set identity of a node changes, the set to which it belongs after the change is at least twice as large as that of the previous set to which it belonged. Hence the set identity of any element can be changed at most $\log n$ times because the largest set to which an element can belong is the whole of U and in each step the set size at least doubles. Hence the total cost of changing the set identity will be at most $n \log n$.

Hence the total cost will be $O(\alpha + n \log n)$ because $\beta < n$.