Data Structure II

Merge-find set, split and join based AVL tree

Section 1

Merge-find Set

Introduction

Suppose there is a ground set S with $|S| = n$, and we are to maintain a collection C of subsets of S, such that any two different elements in C are disjoint. Moreover, we shall support the following.

- Union. Given $x, y \in S$, update $C := C \backslash \{X, Y\} \cup \{X \cup Y\}$, where X, Y are the sets contain x, y respectively.
- Find. Given $x \in S$, return $X \in C$ such that $x \in X$.
- Test. Given $x, y \in S$, test whether they are in the same set X for some $X \in C$.

A well known data structure called merge-find set can achieve the above in near constant armotized time.

Review

We first review the merge-find set. W.l.o.g, we assume $S = \{1, 2, \dots, n\}$. Define P to be an integer array of length n. For each element $i \in S$, we use $P[i]$ to define its parent. If $P[i] = i$, then i has no parent. Observe that P defines a forest, and we use $F(P)$ to denote it. We let $P[i] = i$ for all i initially.

We maintain the invariant that $x, y \in S$ are in the same tree of $F(P)$, if and only if x, y are in a same set in C.

Naive Implementation

For Find(x), we test if $P[x] = x$, if it is then return x; otherwise, update $x := P[x]$ and repeat.

Union(x, y) is implemented by updating $P[\text{Find}(x)] := \text{Find}(y)$.

Test(x, y) simply compares Find(x) and Find(y).

This approach is linear, and hence not practical.

Union by rank and path compression[1]

There are two powerful optimazations. Apply either one of them alone gives $O(\log n)$ per-operation. If we apply them together, we get nearly armotized $O(1)$ per-operation.

[1] See also
`https://en.wikipedia.org/wiki/Disjoint-set_data_structure`

Path Compression

When doing the Find operation, we do two passes.
The first pass is the same as in the naive approach, and we denote the result of it as y. Then for all x that is on the path to y, we change $P[x] = y$. This way, when we do Find(x) later, it takes only $O(1)$.

Union by Rank

For each element x, we define a value $R[x]$ to denote its *rank*. Initially, every element has rank 0. In Union(x, y), define $x' :=$ Find(x) and $y' :=$ Find(y). Then, we let the smaller rank one in x', y' to be the child of the other. If the two happen to have the same rank, we let $P[x'] := y'$ and increase $R[y']$ by one.

Maintaining the Difference[2]

An important extension of merge-find set is to maintain the different of the elements.

Specifically, for each element i, there is an underlying but *unknown* integer $V[i]$. Then information of the form $V[j] - V[i] = k$ comes. We are asked to do the following.

1. Test whehter this is consistent with the previously known information.
2. If it is consistent, then record such an information.

Moreover, we are also asked to answer whether $V[j] - V[i]$ is uniquely determined, and answer the value of it if it is.

[2] `http://acm.hdu.edu.cn/showproblem.php?pid=3038`

Difference Variable

We use another array D to represent the difference. Formaly, $D[i] := V[i] - V[P[i]]$, with $D[i] = 0$ initially. We maintain this as an invariant. We first show that the Find and Union procedure can be modified to maintain this additional invariant.

The Find procedure still takes x and return its root. Observe that for some x, summing up the D value of the points in the path from x to the root r gives $V[x] - V[r]$. Hence we can update $D[x]$ accordingly in the Find procedure for all the relavent x.

The Union procedure now extends to take (i, j, k) such that $V[j] - V[i] = k$ is provided.
Suppose $x' = \text{Find}(i)$, and $y' = \text{Find}(j)$, and suppose we are doing $P[x'] = y'$. Then we update $D[x'] := D[j] - D[i] - k$.

Apply The Data Structure

When we are provided $V[j] - V[i] = k$, we try to $\text{Union}(i, j, k)$. When we are asked to answer $V[j] - V[i]$, we first check if $\text{Find}(j) = \text{Find}(i)$, and it is true if and only if $V[j] - V[i]$ is determined uniquely by the previous information. Also, if it is true, we return $D[j] - D[i]$ as the answer.

Extend to XOR

http://acm.hdu.edu.cn/showproblem.php?pid=3234.

Section 2

Split and Join Based AVL Tree

Join And Split

Instead of implementing the balanced tree with insertion, deleting and searching as primitives, we introduce a way to implement the balanced tree using *Join* and Split procedure as primitives.

Join. The Join proceudre takes T_1 and T_2 as input, such that T_1 and T_2 are AVL trees, and any keyword in T_1 is less than any keyword in T_2. It returns a AVL tree T that is the union of T_1 and T_2. We claim that this can be done in $O(|\mathrm{ht}(T_1) - \mathrm{ht}(T_2)|)$, where ht denotes the height.

Split. The Split procedure takes T and k as inputs, such that T is an AVL tree, and k is a keyword of T. It returns AVL trees T_1, T_2, such that T_1 contains the elements that is less than k in T, and T_2 contains those larger than k. We claim that this can be done in $O(\mathrm{ht}(T))$.

Implementing Dictionary Operations

We observe that the searching operation is as before.

Insertion. Suppose we are going to insert a keyword x to T. Let $[T_1, T_2] :=$ Split(T, x). Then, return Join$(T_1, \{x\}, T_2\}$.

Deletion. Suppose we are going to delete a keyword x from T. Let $[T_1, T_2] :=$ Split(T, x). Then, return Join(T_1, T_2).

Advanced Operations

Interval selection. Suppose we are going to return an AVL tree T' that consists of elements $e \in T$ such that $k_1 \leq e \leq k_2$, where T is an AVL tree.
Interval cut. Cut out an interval from T.
Link and cut. Cut out an interval from T and link it to another place of it.
Maintaining statistics. Return some statistics such as min / max / sum for an AVL tree.

Section 3

Persistent Data Structures

Introduction

Suppose we have a data structure D_0. Whenever we modify D_i, it will create another D_{i+1} that is the data structure after the modification, and D_i is not modified.

Suppose we have done n operations and generated $D_0, D_1, \ldots, D_n$. We call D_i to be the i-th history version of D for $0 \leq i \leq n$. It is also desired to access any history version efficiently.

This kind of data structure D is always called persistent data structures, and are often the backend of functional programming languages and VCS.

Persistent AVL

We are going to make AVL persistent. Of course, it is undesired to copy a new version of AVL when modified, since it takes linear space and time. Instead, we can actually achieve $O(\log n)$ space and time for each modification.

The idea is very simple. Observe that AVL is a link based data structure, and the modification operations are exactly Join and Split. For each of them, only $O(\log n)$ nodes are possibly visited and modified, and others keep unchanged.

Whenever a node is going to be modified, we create a copy of it, and modify the copy.

Observe that root is modified for sure. We return the new root as the root of the new version. The new version can then be accessed via the new root. This way, we reuse most other nodes, while only create copies for modified nodes.

Version Controlled Editor

https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=3983

Application: k-th smallest

`http://acm.timus.ru/problem.aspx?space=1&num=1521`

Application: Dynamic Subtree

http://acm.sgu.ru/problem.php?contest=0&problem=550