Disjoint set data structure (C)

From LiteratePrograms

A disjoint set data structure (http://en.wikipedia.org/wiki/Disjoint-set_data_structure) , also known as a union-find set, is a data structure that keeps track of a partitioning of a set of elements over time as certain operations are done. The three main operations are:

- **MakeSet**: Create a new partition containing a single given element.
- **Find**: Figure out which partition a given element is in.
- **Merge/Union**: Merge two partitions into a single partition.

There are several solutions to this problem. The most efficient known and most commonly used is disjoint set forests with path compression and the union rank heuristic, which has a performance of O(α(*n*)) amortized time per operation, where α(*n*) is a very slowly growing function that for all practical purposes never exceeds 5.

Disjoint set forests

In a *disjoint set forest*, we represent each partition using a tree (not necessarily a binary tree). The nodes of the tree have no child pointers, but do have parent pointers. We use void pointers for the values to allow any value to be placed in the data structure (in other languages one would use generics here):

<<forest node simple>>=
```
typedef struct forest_node_simple_t {
    void* value;
    struct forest_node_simple_t* parent;
} forest_node_simple;
```

When using this data structure in a real application, another option is to embed the forest node properties directly into the structure representing each element (this is known as *internal storage*).

Given any node in the tree, we can reach the root by following parent pointers; we let this root node represent the partition as a whole. This gives us our initial find operation:

<<simple find operation>>=
```
forest_node_simple* FindSimple(forest_node_simple* node) {
    while (node->parent != NULL) {
        node = node->parent;
    }
    return node;
}
```

Merging two trees into one tree is also simple; we just find the root of one of them and attach it to the root of the other by setting its parent pointer:

<<simple union operation>>=
```
/* Given the root elements of two trees, merge the trees into one tree */
void UnionSimple(forest_node_simple* node1, forest_node_simple* node2) {
    node2->parent = node1; /* or node1->parent = node2; */
}
```

Finally, MakeSet allocates a new node with no parent and places the given value in it:

<<simple MakeSet operation>>=
```
forest_node_simple* MakeSetSimple(void* value) {
    forest_node_simple* node = malloc(sizeof(forest_node_simple));
    node->value = value;
    node->parent = NULL;
    return node;
}
```

The `malloc()` function requires stdlib.h:

<<header files>>=
```
#include <stdlib.h>
```

Here's an example of how it might be used:

<<union_find_simple.c>>=
```
header files
#include <assert.h>

forest node simple
simple MakeSet operation
simple find operation
simple union operation

int main() {
    int i1=1, i2=2, i3=3;
    forest_node_simple* s1=MakeSetSimple(&i1);
    forest_node_simple* s2=MakeSetSimple(&i2);
    forest_node_simple* s3=MakeSetSimple(&i3);

    assert(FindSimple(s1) == s1);
    UnionSimple(s1, s2);
    assert(FindSimple(s1) == FindSimple(s2));
    assert(FindSimple(s1) != FindSimple(s3));
    UnionSimple(s2, s3);
    assert(FindSimple(s1) == FindSimple(s2) &&
           FindSimple(s1) == FindSimple(s3));

    return 0;
}
```

Enhancements

Although simple, the algorithm described so far has the problem that the tree can, over many merge operations, develop a large height. Since the find operation takes time proportional to tree height, this can lead to linear worst-case time. Recalling that we can attach either root to the other during a merge, we would ideally always attach the tree of smaller height to the root of the tree of larger height. Unfortunately, height is expensive to calculate, especially without child pointers. Instead, we'll use a rough estimate of height called the *rank*, stored in the tree's root node:

<<forest node>>=
```
typedef struct forest_node_t {
    void* value;
    struct forest_node_t* parent;
    int rank;
} forest_node;
```

Rank is defined as follows:

- MakeSet always produces a tree of rank 0:

<<declarations>>=
```
forest_node* MakeSet(void* value);
```
<<MakeSet operation>>=
```
forest_node* MakeSet(void* value) {
    forest_node* node = malloc(sizeof(forest_node));
    node->value = value;
    node->parent = NULL;
    node->rank = 0;
    return node;
}
```

- If rank(s) ≠ rank(t), then rank(Union(s,t)) is the larger of rank(s) and rank(t). In this case, we attach the tree with smaller rank to the root of the tree with larger rank.
- If rank(s) = rank(t), then rank(Union(s,t)) = rank(s) + 1 = rank(t) + 1.

<<declarations>>=
```
void Union(forest_node* node1, forest_node* node2);
```
<<union operation>>=
```
void Union(forest_node* node1, forest_node* node2) {
    if (node1->rank > node2->rank) {
        node2->parent = node1;
    } else if (node2->rank > node1->rank) {
        node1->parent = node2;
    } else { /* they are equal */
        node2->parent = node1;
        node1->rank++;
    }
}
```

This technique, known as the *union find heuristic*, guarantees logarithmic height, which is reasonably efficient. But we can do even better by using a technique called *path compression*. The idea is that all nodes we visit when traversing up from a node to the root are in the same tree, and so they might as well be pointing directly at the root. We make a second pass and update their parent pointers, speeding up future searches involving these nodes dramatically:

<<declarations>>=
```
forest_node* Find(forest_node* node);
```
<<find operation>>=
```
forest_node* Find(forest_node* node) {
    forest_node* temp;
    /* Find the root */
    forest_node* root = node;
    while (root->parent != NULL) {
        root = root->parent;
    }
    /* Update the parent pointers */
    while (node->parent != NULL) {
        temp = node->parent;
        node->parent = root;
        node = temp;
    }
    return root;
}
```

There are more efficient approaches that don't require two passes, but these are more complicated and we won't consider them here.

We can wrap all this up in a source and header files for reuse:

<<union_find.h>>=
```
#ifndef _UNION_FIND_H_
#define _UNION_FIND_H_

forest node
```

```
declarations

#endif /* #ifndef _UNION_FIND_H_ */
```

<<union_find.c>>=

```
header files
#include "union_find.h"

MakeSet operation
union operation
find operation
```

Here's some sample usage much like our previous example:

<<union_find_example.c>>=

```
#include <assert.h>
#include "union_find.h"

int main() {
    int i1=1, i2=2, i3=3;
    forest_node *s1=MakeSet(&i1), *s2=MakeSet(&i2), *s3=MakeSet(&i3);
    assert(Find(s1) == s1);
    Union(s1, s2);
    assert(Find(s1) == Find(s2));
    assert(Find(s1) != Find(s3));
    Union(s2, s3);
    assert(Find(s1) == Find(s2) &&
           Find(s1) == Find(s3));
    return 0;
}
```

This completes the implementation.

Download code (http://en.literateprograms.org/index.php?title=Special:Downloadcode/Disjoint_set_data_structure_(C)&oldid=18651)