Midterm 3 practice problems

CS 133

July 1, 2022

1 Hash functions and hash tables

- What are the two *good* hash methods we discussed, and how do they work?

  - Remainder: use Horner's method to treat the string as a (big) base-256 number, and then take that modulo m, the hash-table size.
  - Multiplicative: multiply the output of Horner's method by A, a floating-point constant, take the fractional part, multiply by m, and finally round down.

- What are the properties that a hash function should have?

  - Deterministic
  - Uniform distribution
  - Avalanche Effect
  - Low probability of collision

- Why is using string length, or the first character of a string, bad choices for hash functions?

Because it will result in a lot of collisions and is highly non-uniform. All strings with the same first char, same length, will hash to the same value.

- Write the remainder hash function for strings, using a hash size of m.

```
string s = ...;

int h = 0;
for(int i = 0; i < s.length(); ++i)
    h = (256 * h + s[i]) % m;
```

► Write the multiplicative hash function, using an arbitrary predefined constant A and a hash size of m.

```
string s = ...;
float fh = 0;
for(int i = 0; i < s.length(); ++i)
    fh = fmod(A * fh * 256 + A * s[i], 1);

int h = int(fh * m);
```

► How does the collision resolution method *chaining* work?

By storing a linked-list in each hash-table entry. Collisions result in new elements being added to the front of the list.

► How does the collision resolution method *probing* work, and what are the three probe sequences we discussed.

By searching for another empty location in the table when a collision occurs. The three probe sequences are

- Linear: search the next location (i.e., probe sequence is $\text{hash}(s)+i$ where i is incremented every time we hit a full location).
- Quadratic: search $\text{hash}(s) + a * i + b * i^2$ for some constants a and b (not all constants work!).
- Double-hashing: use two hash functions to get $\text{hash}_1(s) + \text{hash}_2(s) * i$. The "best" open addressing method.

► Assuming remainder hashing with $m = 9$, insert the following values into a hash table using chaining:

$$19, 28, 38, 47, 83$$

0	
1	~~2~~7,19
2	83,4~~7~~,38
3	
4	
5	
6	
~~7~~	
8	

► Assuming remainder hashing with $m = 9$, insert the previous values into a hash table using open addressing, with linear probing.

0	
1	19
2	28
3	38
4	~~47~~
5	83
6	
~~7~~	
8	

► What is the load factor α of the above table, after inserting the values?

$\alpha = \frac{5}{9}$

► What is the problem with linear probing?

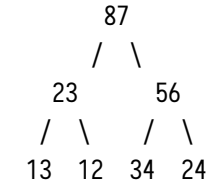
It leads to *clustering*: groups of full table-entries. Any value that hashes into the cluster will have to search all the way to the end of the cluster to find an empty space (slow) and will also grow the cluster, making future collisions that much worse.

2 More trees: Binary Heaps and Disjoint Sets

► Draw the heap that would result from inserting the following values, using the standard `insert(x)` heap function:

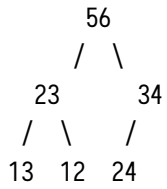
34 13 56 23 12 87 24

Solution:



► Perform one `extract_max()` operation on the heap resulting from the previous problem and draw the result.

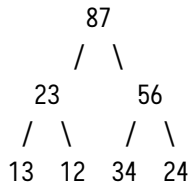
Solution:



► Draw the heap that would result from using the `BuildHeap` algorithm to build a heap out of the following values:

```
34  13  56  23  12  87  24
```

Solution:



In this case, the result is the same as for successive `insert`s, but this is not always the case.

► Suppose we want to build a heap for employee data, where the heap is organized around *employee years of service* (i.e., employees who have worked for the company longer have higher priority).

```
class emp_heap {
  public:
    struct employee
    {
        string name;
        string dept;
        int years;
    };
     ⋮
  private:
    void fix_up(int i);

    vector<employee> heap;
};
```

Write the implementation of `fix_up` for this heap class.

Solution:

```
void emp_heap::fix_up(int i)
{
    while(i > 1 and heap[i].years > heap[i/2].years) {
        swap(heap[i], heap[i/2]);
        i /= 2;
    }
}
```

► In an optimized disjoint set, path compression is performed in the `rep()` function. What if, instead, we performed path compression on all nodes at once? Recall that *path compression* means replacing a node's parent with the root of its tree, so that all a root node's descendants become direct children.

This class uses a `vector<int> parents` to store the parents of each node (nodes don't actually exist). I.e., `parents[i]` records the index of i's parent, or -1 if i is a root.

```
class disjoint_set {
  public:
    disjoint_set(int n)
    {
        parents.resize(n);
        for(int i = 0; i < n; ++i)
            parents[i] = -1; // Everything is a root
    }

    void compress_all();

  private:
    vector<int> parents;
};
```

Write the definition of the `compress_all` function, which should perform path compression on *all* nodes in the disjoint set.

Solution:

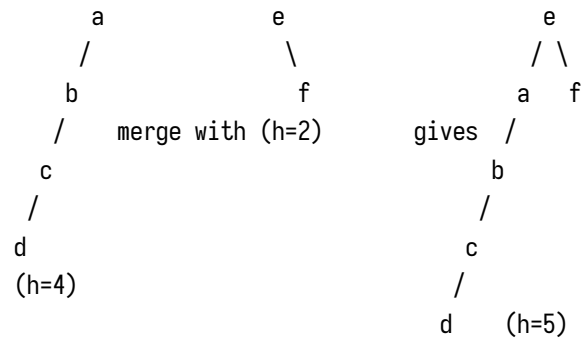
```
void disjoint_set::compress_all()
{
    // For each element i...
    for(int i = 0; i < n; ++i) {
        // Find i's root
        int r = i;
        while(parents[r] != -1)
            r = parents[r];
```

```
        // Compress i
        parents[i] = r;
    }
}
```

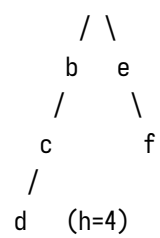
► In a disjoint set with merge-by-rank, when merging two trees, we make the tree with the smaller *rank* an child of the larger-ranked tree (where *rank* is an approximation of the size/height of the tree). Why? Why is it better to make the larger tree the root, and the smaller the child? Give an example of two trees where merge-by-size produces a better outcome than the opposite.

Solution: The complexity of `rep` is based on the height of the tree in question, so merge-by-rank is an attempt to control the *heights* of the resulting trees. (Why use the approximation "rank" instead of actual tree height? Because with path compression, the tree heights are constantly changing, and would need to be contintually updated.) The height of a tree is the maximum of the heights of all of its children, plus 1, so if we use the *taller* tree as a child, it will potentially be larger than the largest existing child, increasing the overall height.

Example:



if we use the smaller tree as the parent, but if we use the larger, we get



Hence, merge-by-height (rank) produces shorter trees.

► For a disjoint set *without* path compression or merge-by-rank, what is the worst-case big-O complexity of `rep` and `merge`, where $n =$ the number of elements in the disjoint set?

Solution: If all elements are in the same set in a single "chain" (degenerate tree), then the complexity of both will be $O(n)$.