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?

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

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

▶ Write the remainder hash function for strings.

▶ Write the multiplicative hash function (you can assume that the remainder hash function is already implemented as `remainder_hash`, and just use an undefined constant A as the multiplicative constant).

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

▶ How does the collision resolution method *open addressing* work, and what are the three probe sequences we discussed.

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

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

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

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

▶ What is the problem with linear probing?

2 More trees: Binary Heaps and Disjoint Sets

Unless stated otherwise, *heap* means max-heap.

▶ 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
```

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

▶ 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
```

▶ 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.

```
void emp_heap::fix_up(int i)
{
    // Your code here
```

► 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.

► 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.

► 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?