July 23, 2019

Homework 2 Solutions
50 Points

Problem 1. Consider hash tables with collision resolved by chaining, implemented as vector-of-lists, as in `fsu::HashTable<K,D,H>`. Show that the standard traversal has runtime $\Theta(b+n)$, where b is the number of buckets and n is the size of the table. Use the context and notation established below. (Hint: use aggregate analysis.)

Solution. The standard traversal calls two `HashTable::` processes: `Begin()` and `Iterator::++()`, each of which is implemented with a loop of calls to the counter increment `++i.vi`. Because `i.vi` is an unsigned integer that is initialized to 0 and limited in range to be less than b, and `i.vi` is always advanced (never decremented), the aggregate number of calls to `++i.vi` is $b = \Theta(b)$.

In addition, each call to `HashTable::Iterator::operator++` calls `ListIterator::operator++` and possibly other List methods when the list is empty or we reach the end of the list. All of the list operations have constant runtime. The aggregate number of such calls is $\Theta(n+b)$: n from the list increment calls and b from the list Empty/Begin calls.

Therefore the aggregate number of calls to any of the primitive operations in a traversal is $\Theta(b) + \Theta(n+b) = \Theta(n+b)$. □

```
// context for analysis:
for (HashTable::Iterator i = t.Begin(); i != t.End(); ++i) {}


class HashTable
{
public:
  typedef HashTableIterator Iterator;

  Iterator Begin();
  Iterator End();

  ...
private:
  Vector<List> v;   // vector of lists (bucket vector)
};

class HashTableIterator
{
public:
  typedef HashTableIterator Iterator;

  Iterator& operator++();

  ...
private:
  Table *      pt;  // pointer to table object
  unsigned     vi;  // vector index
```

```
  ListIterator li;  // bucket iterator
};

HashTableIterator HashTable::Begin()
{
  Iterator i;
  i.pt = this;
  i.vi = 0;                    // start at 0th bucket
  while (i.vi < v.Size() && v[i.vi].Empty())     // while bucket is empty
    ++i.vi;                      // go to next bucket
  if (i.vi == v.Size())        // no non-empty bucket found
    return End();
  i.li = v[i.vi].Begin();      // start at beginning of this bucket
  return i;                    // NOTE: Begin() == End() for an empty bucket
}

HashTableIterator HashTable::End()
{
  Iterator i;
  i.pt = this;
  i.vi = v.Size() - 1;         // last bucket
  i.li = v[i.vi].End();        // end of last bucket
  return i;                    // NOTE: Begin() == End() for an empty bucket
}

HashTableIterator& HashTableIterator::operator++()
{
  ++li;                        // go to next item in bucket
  if (li == v[vi].End())       // if at end of bucket
  {
    do
      ++vi;                    // go to next bucket
    while (vi < pt->v.Size() && v[vi].Empty());    // until bucket is not empty
    if (vi == pt -> v.Size())
      *this = pt -> End();
    else
      li = v[vi].Begin();      // start at beginning of this bucket
  }
  return *this;                // NOTE: Begin() == End() for an empty bucket
}
```

Problem 2. Consider the `Partition` data structure implementing the Union/Find disjoint sets algorithms. Let T be any tree in the forest, and denote the rank of T by d and the number of elements of the set represented by T by k. Show that $d \leq \log_2 k$.

Solution. We apply the principle of mathematical induction on the variable d = rank. The case $d = 0$ reflects the fact that a tree of height 0 has $\log 2 = 1$ node. For the inductive step, assume the result is true for trees in the Partition forest with rank $< d$ and prove the result for trees with rank d.

Let T be a tree of smallest size among all trees of rank d. Let T_1 and T_2 be the two trees that formed T in a Link operation. Denote the sizes of T, T_1, T_2 by k, k_1, k_2, respectively.

Because T_1 has fewer nodes than T, T_1 has rank $< d$, by our choice of T having minimal size among all trees of rank d. Similarly T_2 has rank $< d$. Therefore by the induction hypothesis $d - 1 \leq \log k_1$ and $d - 1 \leq \log k_2$, so

$$d \leq \log k_1 + 1 = \log k_1 + \log 2 = \log 2k_1$$

and similarly

$$d \leq \log 2k_2$$

Now, $k = k_1 + k_2$, and either $k_1 \leq k_2$ or $k_2 \leq k_1$, so either $2k_1 \leq k$ or $2k_2 \leq k$. Therefore $d \leq \log k$.

For any other tree T' of rank d and size k', $k \leq k'$ so $d \leq \log k'$. □

Problem 3(a). Devise an algorithm that translates a 2-D maze of square cells into a graph whose characteristics reflect all properties of the maze. For example, a path in the graph would correspond to a path in the maze. (We'll refer to this translation as an *isomorphism*.)

Solution. Suppose the maze has p rows and q columns, and that the cells are numbered consequtively $c_0, c_1, \ldots, c_{n-1}$, where $n = pq$. Define the graph $G = (V, E)$ by

$$V = \{v_0, v_1, \ldots, v_{n-1}\}$$
$$E = \{(v_i, v_j) | \quad c_i \text{ and } c_j \text{ share a face with no wall}\}$$

In other words, G has n vertices and an edge connecting v_i to v_j iff c_i and c_j share a common face with no wall up between them. The mapping that sends a cell c_i to the vertex v_i preserves the notion of adjacency from the maze context to the graph context, in the sense that:

c_i is a maze-neighbor of c_j iff v_i is a graph-neighbor of v_j.

This is the notion of isomorphism. □

Problem 3(b). Describe in more general terms how the isomorphism would generalize to 2-D mazes of cells of other shapes, such as hexagonal, or variable shape as long as the shapes are polygons. (E.g., any tile floor would do.)

Solution. The description for part (a) works equally well for any polygonal shape, as long as the notion of "share a common face" is reasonably defined:

We are assuming that the cells have polygonal boundaries (consisting of a finite number of straight line segments) and that when cells share any portion of a boundary that portion consists entirely of one of the straight line segments from the boundary of each cell. We refer to this as "sharing a common face". We also say that two cells sharing a common face either have a wall along that face or no wall along that face, so that passage from one cell to the other is either possible (common face has no wall) or not possible (comman face has a wall). When two cells share a common face with no wall, we define these cells to be *adjacent* cells in the maze.

With those ground rules, we can map a maze to a graph as follows: Number the cells consequtively as $\{c_0 \dots c_{n-1}\}$ and an equal number or vertices $\{v_0 \dots v_{n-1}\}$. Moreover, put an edge $e_{i,j} = [v_i, v_j]$ in the graph if and only if c_i and c_j are adjacent. Then the map

$$c_i \to v_i$$

defines an isomorphism from the maze to the graph as constructed. □

Problem 3(c). Based on the technology for 2-D mazes of square cells, invent maze technology for describing 3-D mazes of cubical cells. How would the isomorphism generalize to this case?

Solution.

```
const unsigned char Right = 0x01; // byte 00000001 = 1
const unsigned char Left  = 0x02; // byte 00000010 = 2
const unsigned char Up    = 0x04; // byte 00000100 = 4
const unsigned char Down  = 0x08; // byte 00001000 = 8
const unsigned char Front = 0x10; // byte 00010000 = 16
const unsigned char Back  = 0x20  // byte 00100000 = 32

cell |= Up;          // add wall on up face of cell
cell &= ~Right;      // remove wall on right face of cell
(cell & Front == 0)  // front face of cell is open
cell = Right|Left|Up|Down|Front|Back; // decimal 63 - closed box
```

An isomorphism to graphs works the same as before if we define "share a comman face" appropriately. □