RCL - March 13, 2019
Jonathan Crawford – June 3, 2019

Homework 2 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.)

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 $d \leq \log_2 k$. (Hint: Use mathematical induction on d. For the inductive step, examine the tree of rank d with the fewest number of nodes.)

Problem 3. Consider the family of rectangular mazes described in Disjoint Sets Appendix: Maze Technology.

(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*.)
(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.)
(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?

Context for Problem 1

```
// standard traversal of HashTable t:
for (HashTable::Iterator i = t.Begin(); i != t.End(); ++i)

// HashTable and HashTableIterator context:
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
}
```

1. Using aggregate analysis to count the runtime for the standard traversal of the HashTable shown is the aggregate of the operations listed in red above. The implementation is a vector of lists so that entails iterating through a bucketVector structure and then sequentially iterating through the nested lists within the vector. For the first operation of Begin() the iteration is from the first bucket to the last in search of a non-empty bucket to begin nested traversal. Therefore, Begin() is bounded by the number of buckets $\Theta(b)$. Begin() can also call End(), but End() is included in the original cost since it is the last bucket. Then, operator++() has to iterate through the list of the non-empty bucket using ++li, and then when reaching the end of that list move to the next bucket using ++vi and then Begin() to iterate through it again using ++li and so on until it reaches the End(). This means that operator++() is bounded by both the amount of buckets and the amount of elements. This is equivalent to $\Theta(b+n)$. Then finally it reaches End() which is included in the b for operator++() as well, it's the last bucket in the index. This equals $\Theta(b)$ + Θ(b+n) and since they are both the same number of buckets b possible to be bound by in terms of complexity the cost is aggregated to $\Theta(b+n)$.

2. Prove by Mathematical Induction that any tree T in the Partition forest with rank d and number of elements k is $d \leq \log_2 k$.

    a. Basis: Consider a tree T of lowest rank d = 0, this is just the root with no vertices. The root is one element therefore k = 1. Log2 of 1 = 0. So then,
    $d \leq \log_2 k$
    $0 \leq \log_2 1$
    $0 \leq 0$, This is clear.

    b. Inductive Hypothesis: Assume any tree T in the forest with rank d and number of elements k is $d \leq \log_2 k$ for all integers (cannot have a partial rank or elements).

    c. Inductive Step: Consider a forest F with trees T,U,V with T having the lowest number of nodes among them. Since all the trees in F have rank ≤ d, it follows that U,V would have rank < d. Consider the total number of elements n in the forest F to be the sum of kT + kU + kV. So then $\log_2$kT + $\log_2$kU + $\log_2$kV = $\log_2$n. Consider the case when kT or kU is larger than one another implying k+1, this would imply that d ≤ $\log_2$kV +1 for example. Since the k's have a mathematical quantity that must add up despite their rearrangement or difference in size then d is ≤ $\log_2$kU regardless of this variability, and since d < the smallest log2k amongst them, it follows that d will always be less than or equal to any $\log_2 k$ for any tree in F.

3. Consider the family of rectangular mazes described in Disjoint Sets Appendix: Maze Technology.

    a. This entails taking the numrows and numcols portion from (RCL2016) and instead of using the bits that represent the faces use vertices and edges so that you can create an adjacency matrix where you would put an edge E where the Vx and Vy of matrix were true, and then there would inherently be

isomorphism proven by the matrix because whether x,y is true or y,x is true there is an edge. You also know where everything is in the maze because it will have a common edge side that is clearly defined by the matrix. This logic is given in the graph notes from (RCL 2018):
In the adjacency matrix representation we define an n × n matrix M by
M(i,j) = (G has an edge from vi to vj ? 1 : 0)

b. You can use the same thing as above but there will just be more faces since they can be in any direction. But since they are a defined shape as with the tile floor example their faces are definable in some pattern, f1,f2.f3…fk rather than thinking of them as North, South, etc. The adjacency matrix would tell you when to put an edge based on if the x,y were true it would just have a larger amount of rows and columns. This maintains the property of an isomorphic graph as defined in discrete math due to the x,y to y,x locality. It is noted in the question that they have to be polygons because a squiggly line would lose the preservation of locality. Also, as noted in Wikipedia*: In graph theory, an isomorphism of graphs G and H is a bijection between the vertex sets of G and H such that any two vertices u and v of G are adjacent in G if and only if f(u) and f(v) are adjacent in H. This kind of bijection is commonly described as "edge-preserving bijection", in accordance with the general notion of isomorphism being a structure-preserving bijection.*
Graph Isomorphism: https://en.wikipedia.org/wiki/Graph_isomorphism

c. The 3d cubes would just need to build on technology already in place in the maze technology using the z-axis for up or down From(RCL 2016). Since length is arbitrary the 2d portion can be derived as being walls with z height already, then you just need a top and bottom because their sizes do not really matter just that it's a cube. Isomorphism is maintained due to the directional nature, of eg. North wall, or South face etc. representing locality:

```
const unsigned char NORTH = 0x01; // byte (00000001) cell with north wall only
const unsigned char EAST = 0x02; // byte (00000010) cell with east wall only
const unsigned char SOUTH = 0x04; // byte (00000100) cell with south wall only
const unsigned char WEST = 0x08; // byte (00001000) cell with west wall only
const unsigned char TOP = 0x10;
const unsigned char BOTTOM = 0x20;
```

The bottom is 0x20 for example because they're bits as above so you follow the pattern of moving the bit over one space. This is because you need the prior bits in order to tell when they are all in use at once to represent different amounts of closure of the box.
111111_2 is a fully closed box then.

```
cell &= ~EAST; // remove east wall
cell |= SOUTH; // add south wall
(cell & WEST == 0) // cell has no west wall
(cell & TOP == 0) // a cell with no roof on it like a basket
```