Midterm 2 Review

Disclaimer

- Topics covered in this review may not appear in the exam.
- Topics not covered in this review may appear in the exam.

Hash Table

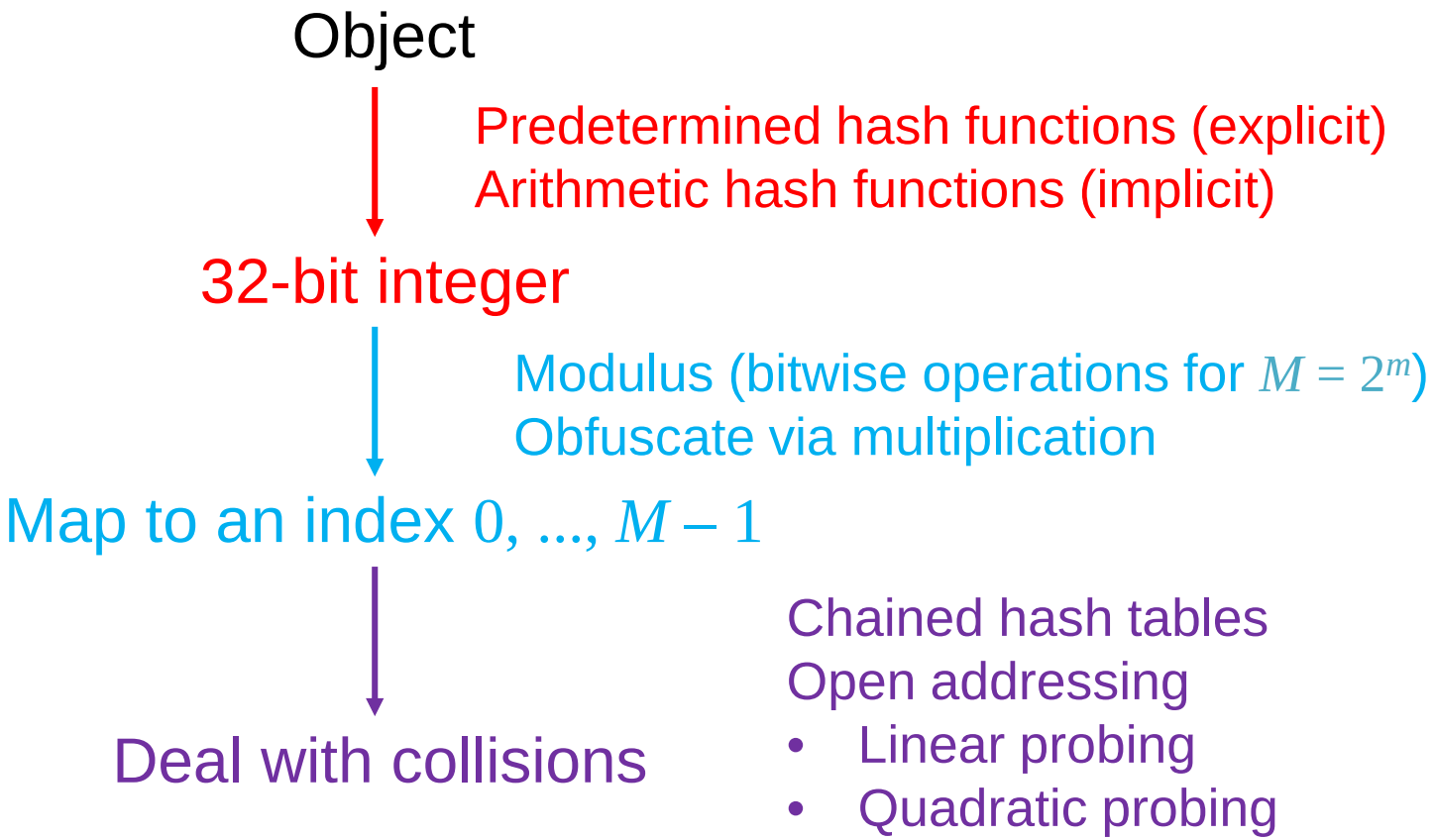
Textbook Ch 11

Goal

Our goal:

- Store data so that all operations are $\Theta(1)$ time
- The memory requirement should be $\Theta(n)$

Summary


Object
Predetermined hash functions (explicit)
Arithmetic hash functions (implicit)
32-bit integer
Modulus (bitwise operations for $M = 2^m$)
Obfuscate via multiplication
Map to an index $0, ..., M - 1$
Deal with collisions
Chained hash tables
Open addressing
• Linear probing
• Quadratic probing

Hash Function: Properties

Necessary properties of such a hash function h are:

1a. Should be fast: ideally $\Theta(1)$

1b. The hash value must be *deterministic*

- It must always return the same 32-bit integer each time

1c. Equal objects hash to equal values

- $x = y \Rightarrow h(x) = h(y)$

1d. If two objects are randomly chosen, there should be only a one-in-2^{32} chance that they have the same hash value

Arithmetic hash functions

Let the individual member variables represent the coefficients of a polynomial in x:

$$p(x) = c_0 x^{n-1} + c_1 x^{n-2} + \cdots + c_{n-3} x^2 + c_{n-2} x + c_{n-1}$$

Use Horner's rule to evaluate this polynomial at a prime number, *e.g.*, hashing a string with $x = 12347$:

```
unsigned int hash( string const &str ) {
    unsigned int hash_value = 0;

    for ( int k = 0; k < str.length(); ++k ) {
        hash_value = 12347*hash_value + str[k];
    }

    return hash_value;
}
```

Mapping Down: Properties

Necessary properties of this mapping function h_M are:

2a. Must be fast: $\Theta(1)$

2b. The hash value must be *deterministic*

- Given n and M, $h_M(n)$ must always return the same value

2c. If two objects are randomly chosen, there should be only a one-in-M chance that they have the same value from 0 to $M-1$

Modulus operator

Easiest method: return the value modulus M

```
unsigned int hash_M( unsigned int n, unsigned int M ) {
    return n % M;
}
```

Unfortunately, calculating the modulus (or remainder) is expensive

If $M = 2^m$, we can simplify the calculation by bitwise operations

- left and right shift and bit-wise and

Obfuscation: the multiplicative method

Multiplying by a fixed constant is a reasonable method

- Take the middle m bits of Cn:

```
unsigned int const C = 581869333;  // some number

unsigned int hash_M( unsigned int n, unsigned int m ) {
    unsigned int shift = (32 – m)/2;
    return ((C*n) >> shift) & ((1 << m) – 1);
}
```

Chained hash table

Associating each bin with a linked list.

For any object assigned to the bin by the hash function, finding, inserting, and erasing the object is done on the linked list.

Load Factor

To describe the length of the linked lists, we define the *load factor* of the hash table:

$$\lambda = \frac{n}{M}$$

This is the average number of objects per bin

- This assumes an even distribution

Linear Probing

Probe the bins of the hash table by searching forward linearly

Assume we are inserting into bin k:

- If bin k is empty, we occupy it
- Otherwise, check bin $k + 1$, $k + 2$, and so on, until an empty bin is found
  - If we reach the end of the array, we start at the front (bin 0)

Quadratic Probing

Quadratic probing suggests moving forward by quadratic amounts

If we ensure $M = 2^m$ then:

```
int initial = hash_M( x.hash(), M );

for ( int k = 0; k < M; ++k ) {
    bin = (initial + (k + k*k)/2) % M;
}
```

Open Addressing

- Erasing: two approaches
  - Trying to fill the hole (not applicable to quadratic probing)
  - Lazy erasing

- Common weakness: clustering
  - Linear probing: primary clustering
  - Quadratic probing: secondary clustering (less severe than primary clustering)

Disjoint Sets

Textbook Ch 21

Disjoint Sets

Definition: a set of elements partitioned into a number of disjoint subsets

There are two operations we would like to perform on disjoint sets:

- Determine if two elements are in the same disjoint set, and
- Take the union of two disjoint sets creating a single set

We will determine if two objects are in the same disjoint set by defining a **`find`** function

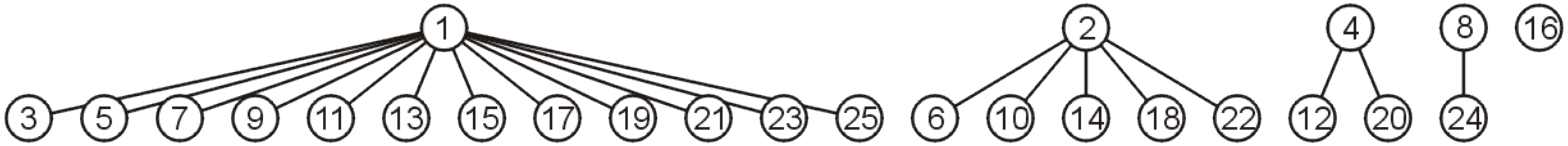
- **`find(a)`**: find the representative object of the disjoint set that **`a`** belongs to
- Given two elements **`a`** and **`b`**, they are in the same set iff.

```
find( a ) == find( b )
```

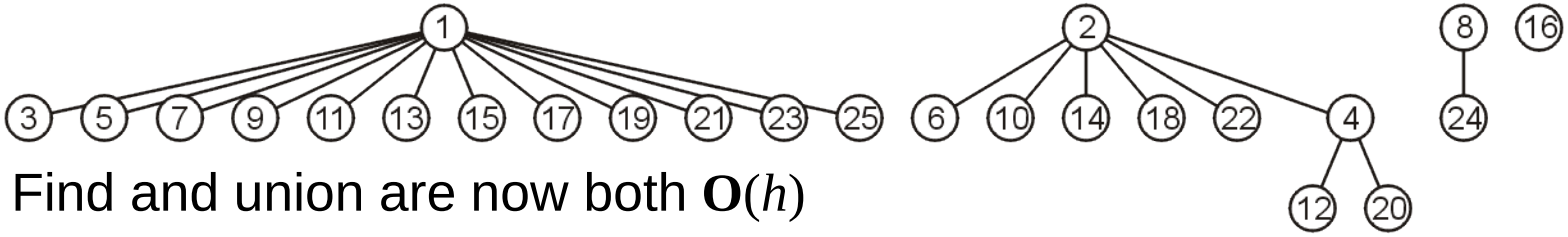
Implementation

Let each disjoint set be represented by a general tree

- The root of the tree is the representative object



To take the union of two such sets, we will simply attach one tree to the root of the other



Find and union are now both $\mathbf{O}(h)$

Optimizations

To optimize both `find` and `set_union`, we must minimize the height of the tree

- Point the root of the shorter tree to the root of the taller tree
  - The height of the taller will increase if and only if the trees are equal in height
  - The height and average depth of the worst case are $\mathbf{O}(\ln(n))$
- (Path Compression) whenever find is called, update the object to point to the root

Graphs

Textbook Ch B.4, B.5.1, 22.1

Concepts

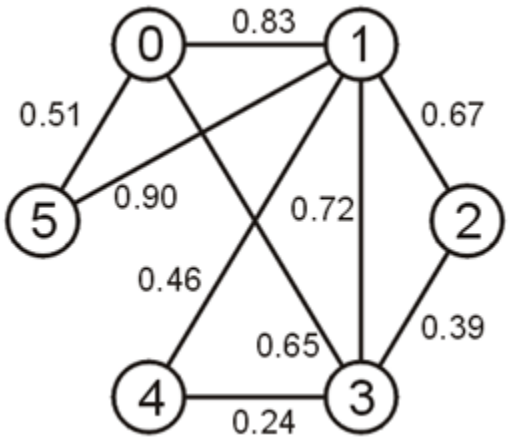
- Definitions
  - Undirected graphs
  - Directed graph
  - Vertex, edge, degree
  - Path, simple path, cycles
  - Connectedness
  - Weight
  - Tree, DAG

Adjacency Matrix

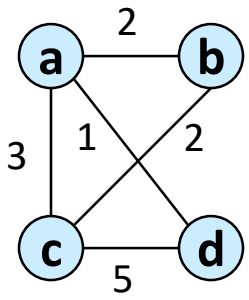
Define an $n \times n$ matrix $\mathbf{A} = (a_{ij})$ and if the vertices v_i and v_j are connected with weight w, then set $a_{ij} = w$ and $a_{ji} = w$

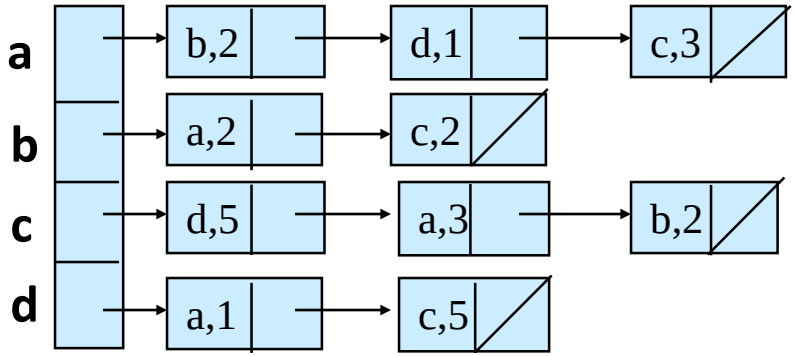
That is, the matrix is symmetric, *e.g.*,

	0	1	2	3	4	5
0		0.83		0.65		0.51
1	0.83		0.67	0.72	0.46	0.90
2		0.67		0.39		
3	0.65	0.72	0.39		0.24	
4		0.46		0.24		
5	0.51	0.90				



Adjacency list





Graph traversal

Textbook Ch 22.2/3/5

Graph Traversal

Different from tree traversal: there may be multiple paths between two vertices.

To avoid visiting a vertex for multiple times, we have to track which vertices have already been visited

The time complexity of graph traversal is $\Theta(|V| + |E|)$

Breadth-first traversal

Breadth-first traversal on a graph:

- Choose any vertex, mark it as visited and push it onto queue
- While the queue is not empty:
  - Pop the top vertex v from the queue
  - For each vertex adjacent to v that has not been visited:
    - Mark it visited, and
    - Push it onto the queue

This continues until the queue is empty

- If there are no unvisited vertices, the graph is connected

The size of the queue is $O(|V|)$

Depth-first traversal

Depth-first traversal on a graph:

- Choose any vertex, mark it as visited
- From that vertex:
  - If there is another adjacent vertex not yet visited, go to it
  - Otherwise, go back to the previous vertex
- Continue until no visited vertices have unvisited adjacent vertices

Two implementations:

- Recursive
- Use a stack

Connectedness

Let us determine whether one vertex is connected to another

- v_j is connected to v_k if there is a path from the first to the second

Strategy:

- Perform a breadth-first traversal starting at v_j
- If the vertex v_k is ever found during the traversal, return true
- Otherwise, return false

Connected Components

Suppose we want to partition the vertices into connected sub-graphs

- While there are unvisited vertices in the tree:
  - Select an unvisited vertex and perform a traversal on that vertex
  - Each vertex that is visited in that traversal is added to the set initially containing the initial unvisited vertex
- Continue until all vertices are visited

Minimum spanning tree

Textbook Ch 23

Spanning trees

Given a connected graph with n vertices, a spanning tree is defined as a subgraph that is a tree and includes all the n vertices

- It has $n - 1$ edges

The weight of a spanning tree is the sum of the weights on all the edges which comprise the spanning tree

The spanning tree that minimizes the weight is termed a *minimum spanning tree*

Minimum Spanning Trees

Simplifying assumption:

- All edge weights are distinct

This guarantees that given a graph, there is a unique minimum spanning tree.

Cut property: let S be any subset of nodes, and let e be the least weight edge with exactly one endpoint in S. Then the MST T^* contains e.

Prim's Algorithm

Prim's algorithm for finding the minimum spanning tree states:

- Start with an arbitrary vertex to form a minimum spanning tree on one vertex
- At each step, add the edge with least weight that connects the current minimum spanning tree to a new vertex
- Continue until we have $n - 1$ edges and n vertices

Implementation and analysis

Use a priority queue to find the closest vertex

We have two run times when using

- A binary heap: $O(|E| \ln(|V|))$
- A Fibonacci heap: $O(|E| + |V| \ln(|V|))$

Kruskal's Algorithm

- Sort the edges by weight
- Go through the edges from least weight to greatest weight
  - add the edges to the spanning tree so long as the addition does not create a cycle
- Repeatedly add more edges until:
  - $|V| - 1$ edges have been added, then we have a minimum spanning tree
  - Otherwise, if we have gone through all the edges, then we have a forest of minimum spanning trees on all connected sub-graphs

Analysis

Sort the edges

- $O(|E| \ln(|E|)) = O(|E| \ln(|V|))$

Determining if adding an edge creates a cycle

- Check if the two vertices of the edge are already connected using disjoint sets
- Effectively constant time