Hash Tables

Introduction

Hash tables offer faster searching than a BST, but they don't order their elements alphabetically

General Idea

Create an array with N slots to hold your values

Define a func `f()` that takes a value V as an input and produces a lost number from `0` to `N-1` as its output

- e.g. for people, `f()` might add the person's birthday digits mm+dd+yyyy to compute a slot number

Use `f()` to find the right slot number to either add a new item, or search for an existing item

Occasionally, `f()` will map several items to the same slot. You can either place the second item in the next open slot or have a linked list of items in each slot

---

The Modulus Operator

The `%` operator is used to divide two numbers and obtain the remainder

Example: Divide a bunch of numbers by 5

- All of the remainders are less than 5 (between 0-4)
- If you divide a bunch of numbers by 100,000, the remainders would all be less than 100,000, between 0-99,999

> When you divide by a given value N, all of your remainders are guaranteed to be between 0 and N-1

The "Hash Table"

So far the most efficient ADT to insert and search is BST (O(log2N))

Example: Create an ADT where we can insert 9-digit student ID#s for all 50,000 UCLA student and then find if our ADT holds a given ID# in just one step (**O(1)**)

Idea 1:

- Create a really large array with 1 billion slots
  - One slot for each valid ID#
- To add a new ID# with a value of N, we'll simply set `array[N]` to true
- To determine if our array holds a prev-added value Q, simply check if `array[Q]` is true

```
class AlmostHashTable
{
    public:
        void addItem(int n)
        {
            m_array[n] = true;
        }
        bool holdsItem(int q)
        {
            return m_array[q] == true;
        }
    private:
        bool m_array[1000000000];
};

int main()
{
    AlmostHashTable x;
    x.addItem(400683948);
    if (x.holdsItem(1234) != true)
        cout << "Couldn't find it!";
}
```

- Problem: Really inefficient
    - Our array has 1 billion slots, but there are only 50,000 UCLA student IDs
    - Wasting 999,950,000 other slots
    - Better if could use same algorithm but with smaller array
        - Like one with **100,000** slots instead

Idea 2:

Keep track of 50,000 ID#s in an array with 100,000 slots

- If just try to use 9-digit number to index the array (like above), cannot index in 100,000 slot array
- Need function that takes in a 9-digit ID# and converts it to a unique slot number between 0 and 99,999

> f(x) is a **mapping function**

```
class AlmostHashTable2
{
    public:
        void addItem(int n)
        {
            int slot = mapFunc(n);
            m_array[slot] = true;
        }
    private:
        int mapFunc(int idNum)
            {/* ??? */}
        bool m_array[100000];
};
```

- To add new item:
    - `mapFunc` converts 9-digit ID to slot number in array
    - track our ID# in that slot (or **bucket**) by setting it to `true`

The Mapping Function

How can we write a `mapFunc` that converts our large ID# into a bucket# that falls within 100,000 element array? **Modulus Operator**

```
int mapFunc(int idNum)
{
    const int ARRAY_SIZE = 100000;
    int bucket = idNum % ARRAY_SIZE;
    return bucket;
}
```

```
int mapFunc(int idNum)
{
    return idNum % 100000;
}
```

```
int main()
{
    AlmostHashTable2 x;
    x.addItem(400683948);
    x.addItem(111105224); //results in the same slot
    x.addItem(222205224); //results in the same slot
}
```

What if search by value other than int?

We can't just use the modulus operator on a string etc.

1. First compute a unique numeric value for our string using a "hash" function
    - hash function takes an arbitary input (string etc.) and produces an integer output
2. Use modulo to compute a bucket number

```
int mapFunc(string &name)
{
    int h = hash(name);
    return h % 100000;
}
```

Hash Functions For Strings

Possibility 1

```
int hash(const string &name)
{
    int i, total = 0;
    for (i = 0; i < name.length(); i++)
        total = total + name[i];
    return total;
}
```

- Not good because yields same result for "BAT" and "TAB"

Possibility 2

```
int hash(const string &name)
{
    int i, total = 0;
    for (i = 0; i < name.length(); i++)
        total = total + (i+1)*name[i];
    return total;
}
```

Possibility 3 (C++ Hash Function): THIS ONE IS BEST

```
#include <functional>

unsigned int yourMapFunction(const std::string &hashMe)
{
    std::hash<std::string> str_hash; //creates a string hasher

    unsigned int hashValue = str_hash(hashMe); //hash string

    //then just add your own modulo
    unsigned int bucketNum = hashValue % NUM_BUCKETS;
    return bucketNum;
}
```

- First define a C++ string hashing object
- Then use object to hash your input string, returns a hash value between 0 and 4 billiion

Writing Own Hash Function for Non-Standard Data Type

1. The hash function must always give us the same output value for a given input value
2. Hash function should disperse items throughout the hash array as randomly as possible
3. Always measure how well it disperses items

Collision

> A collision is a condition where two or more values both map to the same bucket in the array

This causes ambiguity, and we can't tell what value was actually stored in array

---

Closed Hash Table with **Linear Probing**

Addresses collisions by putting each value as close as possible to its intended buckets

Since store every original value in the array, there is no chance of ambiguity

Linear Probing Insertion

1. Use mapping function to locate the right bucket in array
2. If target bucket is empty
   - Store value there
   - Instead of storing `true` in the bucket, store **full original value** to prevent ambiguity
3. If the bucket is occupied
   - Scan down from that bucket until we hit the **first open bucket**
   - Put the new value in the first open slot

   Insert an item near the end of the table

   - When you run into a collsion on the last bucket, and go past the end
   - **Wrap back around the top**

Linear Probing Searching

1. Compute a target bucket number with our mapping function
2. Then look in that bucket for our value
   - If find it, then good

    - If don't find it, **probe linearly** down until we either find our value or hit an empty bucket
        - If while probing, run into an empty bucket->your value is not in the array
    - If end up searching past the end, just wrap back up to the top

Closed Hash Table:

- Since data is stored in a fixed-sized array, there are a fixed (closed) # of buckets to put values
- Once **run out** of empty buckets, can't add new values
- Linked lists and BST do not have this problem

Code and Details

Each bucket in the array:

- Holds 2 items
    1. A variable to hold your value (e.g. an int for an ID#)
    2. A "used" field that indicates if this bucket in the hash table has been filled or not
        - If field `false`: this `Bucket` in the array is empty
        - If `true`: `Bucket` is already filled with valid data

```
struct BUCKET
{
    int idNum;
    bool used;
}
```

Insertion

```
const int NUM_BUCK = 10;

class HashTable
{
    public:
        void insert(int idNum)
        {
            int bucket = mapFunc(idNum); //COMPUTATION

            for (int tries = 0; tries < NUM_BACK; tries++)
            {
                if (m_buckets[bucket].used == false)
                {
                    m_buckets[bucket].idNum = idNum;
                    m_buckets[bucket].used = true;
                    return;
                }
                bucket = (bucket + 1) % NUM_BACK;
            }

            //no room left in hash table!!!
        }
    private:
        int mapFunc(int idNum) const
            { return idNum % NUM_BACK; }

        BUCKET m_buckets[NUM_BACK];
};
```

- Our hash table has 10 slots (buckets)
- Our mapping function uses modulus operator
- First, compute the starting bucket number
- Since our array has 10 slots, we will loop up to **10 times looking for an empty space**
  - If we don't find an empty space after 10 tries, our table is full!
  - We store our new item in the first unused bucket that we find, starting with the bucket selected by our mapping function
  - If the current bucket is already occupied by an item, advance to the next bucket (wrapping around from slot 9 back to slot 0 when we hit the end)

    ```
    bucket = (bucket + 1) % NUM_BACK;

    //is the same as

    bucket = bucket + 1;
    if (bucket == NUM_BACK)
        bucket = 0;
    ```

```
main()
{
    HashTable ht;
    ht.insert(29);
    ht.insert(65);
    ht.insert(79);
}
```

- When we construct our hash table, all of our buckets have their used field initialized to false
  - Indicates all empty

Searching

```
const int NUM_BUCK = 10;

class HashTable
{
    public:
        bool search(int idNum)
        {
            int bucket = mapFunc(idNum);

            for (int tries = 0; tries < NUM_BUCK; tries++)
            {
                if (m_buckets[bucket].used == false)
                    return false;
                if (m_buckets[buckets].idNum == idNum)
                    return true;
                bucket = (bucket + 1) % NUM_BUCK;
            }

            return false; //not in the hash table
        }
    private:
        ...
}
```

- Compute the starting bucket where we expect to find our item
- Since we may have collisions, in the worst case, we may need to check the entire table (10 slots)
    - Every single inserted item collided
- If we reach an empty bucket and haven't yet found our item-> item is not in table, return false
- Otherwise, the bucket is in-use
    - If it also holds ID# to searched for, then found and done
- If did not find our item, advance to the next bucket in the search
    - Wrap around when reach the end of array
- If we went through every bucket and did not find our item, then it is not in the hash table

```
main()
{
    HashTable ht;
    ...
    bool x;
    x = ht.search(29);
    x = ht.search(175);
    x = ht.search(20);
}
```

Included Additional Associated Values in Hash Table

For instance, what if I want to store the student's name and GPA in each bucket along with their ID#?

```
struct Bucket
{
    int idNum;
    string name;
    float GPA:
    bool used;
}
```

```
void insert(int id, string &name, float GPA)
{
    int bucket = mapFunc(idNum);
    for (int tries = 0; tries < NUM_BUCK; tries++)
    {
        if (m_buckets[bucket].used == false)
        {
            m_buckets[bucket].idNum = id;
            m_buckets[bucket].name = name;
            m_buckets[bucket].GPA = GPA:
            m_buckets[bucket].used = true;
            return;
        }
        bucket = (bucket + 1) % NUM_BUCK;
    }
}
```

- Even thought we chood bucket# based on the ID#, can also add other values

When you look up a student by ID#, can also get their name and GPA

```
bool search(int id, string &name, float GPA)
{
    int bucket = mapFunc(idNum);
    for (int tries = 0; tries < NUM_BUCK; tries++)
    {
        if (m_buckets[bucket.used] == false)
            return false;
        if (m_buckets[bucket].idNum == idNum)
        {
            name = m_buckets[bucket].name;
            GPA = m_buckets[bucket].GPA;
            return true;
        }
        bucket = (bucket + 1) % NUM_BUCK;
    }
    return false;
}
```

Deletion

Naive Approach: Delete value of 65 from hash table

- Zero out value and set teh `used` field to false
- But now search for value 15
    - Hash function yields same bucket slot for 15 and 65 in this 9 element array
    - But this bucket is empty now
        - If our value of 15 were in the ahsh table, we would have found it before hitting an empty slot
            - hence search will conclude that 15 is not in the hash table, because we've deleted 65
        - But in fact 15 was next down from where 65 was

> If we delete a value where a collision happened, when we try to search again, we may prematurely abort our search, failing to find the sought-for value

Attempts to solve this problem are NOT recommended

> Only use Closed/Linear Probing hash tables when you don't intend to delete items from your hash table

- e.g. build a hash tables that holds words for a dictionary, will just add words not delete

---

The "Open Hash Table"

Avoid problems:

- Difficulty to delete items
- Cap on the number of items it can hold

Idea:

- Instead of storing our values directly in the array, each array bucket points to a linked list of values
    - Linked list can hold an unlimited numbers of values

To insert a new item:

1. As before, compute a bucket # with mapping function

   ```
   bucket = mapFunc(idNum);
   ```

2. Add your new value to the linked list at array[bucket]

3. DONE!

To search for an item:

1. As before, compute a bucket # with mapping function

   ```
   bucket = mapFunc(idNum);
   ```

2. Search the linked list at array[bucket] for your item

3. If we reach the end of the list without finding our item, it's not in the table

Deletion:

- Remove the value from the linked list

> If you plan to repeatedly insert and delete values into the hash table, then the Open Table is better

- Also can insert more than N items into your table and still have great performance

---

Hash Table Efficiency

How efficient is the hash table ADT? How long does it take to locate an tiem and to insert an item?

- Depends on:

  1. the type of hash table (closed, open)

  2. how full your hash table is

  3. how many collision you have in the table

The Load Factor

The load of a hash table is the **maximum number of values you intend to add** divided by **the number of buckets in the array**

> **L= (Max # of values to insert) / (Total buckets in the array)**

Example: A load of L = .1 means you array has 10X more buckets than you need

- You'll only fill 10% of the buckets

Example: A load of L = .9 means your array has 10% more buckets than you need

- You'll fill 90% of the buckets

Closed Hash Table Efficiency

If Hash Table is **Empty** (or nearly)

- Insertion: zero chance of collision
  - Add value in 1 step: **O(1)**
- Search: **O(1)**
  - Just as fast, we find an item right away or it's not in table

If Hash Table is nearly **Full**

- Insertion: could take up to N steps
  - Every single bucket is filled except for one, and we keep probing until that last one-> **O(N)**
- Searching: **O(N)**

Overall Efficiency

Given a particular load `L` for a Closed hash Table with Linear Probing: compute the average # of tries it'll take you to insert/find an item

- Average # of Tries = $1/2(1+1/(1-L))$ for L < 1.0

If Closed Hash Table has a load factor of:

.10: ~1.05 searches .20 (array is 5x bigger than required): ~1.12 searches .90: ~5.50 searches

Open Hash Table Efficiency

Given a particular load `L` for an Open Hash Table: compute the average # of tries it'll take you to insert/find an item

- Average # of Checks = $1+L/2$

If Open Hash Table has a load factor of .10: ~1.05 searches .20: ~1.15 searches .90: ~1.45 searches

> Open hash tables are almost always more efficient than Close Hash Tables

Question: If you want to store up to 100 items in an Open Hash Table and be able to find any item in roughly 1.25 searches, how many buckets must your hash table have?

- Expected # of Checks = 1 + L/2
  - 1.25 = 1 + L/2
  - L = .5
- L = # of items of insert/ required hash table size
  - .5 = 1000 / required size
  - size = 2000 buckets

Tradeoff

- Use a really big hash table and ensure really fast searches
    - But waste lots of memory
- Really small hash table, slower search

> When choosing the exact size of your hash table (the # of buckets) -> always try to choose a prime number of buckets

- Instead of 2000, choose 2017
- Causes more even distribution and fewer collisions

The Unordered_Map: Hash-Based Version of a Map

```
#include <unordered_nap>
#include <iostream>
#include <string>

using namespace std;

int main()
{
    unordered_map<string, int> hm; //define new u_m
    unordered_map<string, int>::iterator iter;

    hm["Carey"] = 10; //insert new item
    hm["David"] = 20;

    iter = hm.find("Carey"); //find Carey in the hash map

    if (iter == hm.end()) //cannot find Carey
        cout << "carey was not found!";
    else
    {
        cout << "When we look up " << iter->first;;
        cout << " We find " << iter->second;
        //when we look up Carey we find 10
    }
}
```

Hash Tables vs. BST

```
                Hash Tables             BST
Speed            O(1)                 O(log2N)
                regardless of N
```

```
Simplicity      Easy to implement    More complex

Max Size        Closed: limited        Unlimited
                by array size
                Open: not limited,
                but high load impacts
                performance

Space           Wastes a lot of space   Uses as much
Efficiency      if you have a large     memory as needed
                hash table holding few  (One node per
                items                   item inserted)

Ordering        Random                  Alphabetical
```

- For closed hash table: if you want to expand hash table size
    - basically have to create a whole new one
        1. allocate a whole new array with more buckets
        2. rehash every value from the original table into new table
        3. free original table

Tables

A table is an ADT that stores a bunch of "records" (like student info) and then provides multiple ways to search for an item

E.g. you could search for students by phone number, full name, and or student ID

A simple ADT like a hash table or BST only enable searching efficently by a single field type

In a table ADT you store the full record (name, number, ID, GPA) just once, then create multiple light "indexes" (e.g. using BSTs/hash tables) to speed up searching by different field types

- BSTs/Hash Tables store the name/ID as key and slot number of table as value
- "Record": a group of related data
    - Each record has a bunch of "fields" like Name, Phone#, Birthday etc. that can be filled with values
    - A bunch of records-> table
    - Some fields, like SSN will have unique values across all records -> useful for searching a unique record

Create a Record and Table

Record

```
struct Student
{
    string name;
    int IDNum;
    float GPA;
    string phone;
    ...
}
```

- a struct or class to represent a record of data

Table

```
vector<Student> table;
```

- array or vector

Search for a Record with a Particular Field Value

Write a search function that iterates through the array/vector

```
int SearchByName(vector<Student> &table, string &findName)
{
    for (int s = 0; s < table.size(); s++)
        if (findName == table[s].name)
            return s;
    return -1; //if not found
}
```

Create a whole class for table

```
struct Student
{
    string name;
    int IDNum;
    float GPA;
    string phone;
};

class TableOfStudent
{
    public:
        TableOfStudents(); //construct a new table
        ~TableOfStudents(); //destruct our table
        void addStudent(Student &stud); //add
        void TableOfStudents::addStudent(Student &record)
        {
            m_students.push_back(record);
        }
        Student getStudent(int s);
        int searchByName(string &name);
        int searchByPhone(int phone);
    private:
        vector<Student> m_students;
};
```

- Linear search like this is inefficient
- If we sort, have to resort everytime after insertion, and can only sort one field
- If put in BST organized by name
    - Name can be searched efficiently, but not ID or Phone
- If create two tables, first order by name, second by ID
    - Efficient search, BUT 2 copies of every record, waste of space

Make an Efficient Table

1. Still use a vector to store all of our records
2. Also add a data structure that lets us associate each person's name with their slot number in the vector

```
private:
vector<Student> m_students;
map<string, int> m_nameToSlot;
```

- Map lets us quickly loop up a name and find out which slot in the vector holds the related record

3. Add another data structure to associate each person's ID with the slot # too

```
map<int,int> m_idToSlot;
```

- Lets us quickly look up an ID and find out which slot in the vector holds the related record

> these secondary data structre are called "indexes"

Add Student

```
void addStudent(Student &stud)
{
    m_students.push_back(stud);
    int slot = m_students.size() - 1; //get slot# of new record
```

```
        m_nameToSlot[stud.name] = slot;
        m_idToSlot[stud.IDNum] = slot;
    }
```

1. Add our new record to the end of our vector
2. Update our first index to point to our new record
    - Put into BST under the map
3. Update our second Index

Deletion and Update (A record's searchable fields)

- Also have to update indexes

Use Hashing to Speed Up Tables

Use hash tables to index data instead of bst

- Now we can have O(1) searches by name
- However, hash tables store data in random random
    - BST is slower but orders key fields in alphabetical order
        - If want to print out all students alphabetically by their name, BST is good
            - If indexed with hash table, more work...
    - Would use a hash table for the phone field, because just need quick search, don't need ordering

Big-O of Traversing All Elements of Hash Table