Lecture #14

- Hash Tables
  - The Modulus Operator
  - Closed hash tables
  - Open hash tables
  - Hash table efficiency and "load factor"
  - Hashing non-numeric values
  - unordered_map: A hash-based STL map class
- (Database) Tables

The Hash Function

How can we write a hashFunc that converts our large ID# into a bucket # that falls within our 100,000 element array?

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

This line takes an input value idNum and returns an output value between 0 and ARRAY_SIZE - 1. (0 to 99,999)

RIGHT! The C++ % operator (aka the modulus division operator) does exactly what we want!!!

And this corresponding value can be used to pick a bucket in our 100,000 element array!

So now for each input ID# we can compute a corresponding value between 0-99,999!

Closed Hash Table with Linear Probing: Insertion

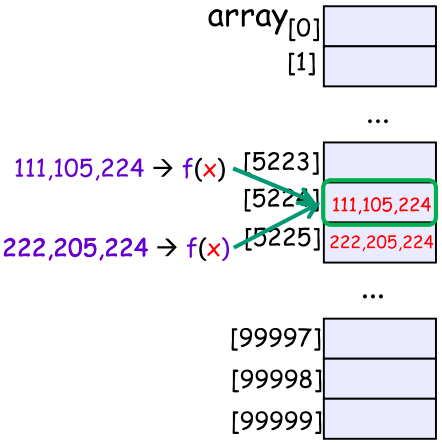
Linear Probing Insertion:

As before, we use our hash function to locate the right bucket in our array.

If the target bucket is empty, we can store our value there.

However, instead of storing true in the bucket, we store our full original value - this prevents ambiguity!

If the bucket is occupied, scan down from that bucket until we hit the first open bucket. Put the new value there.



Closed Hash Table with Linear Probing: Insertion

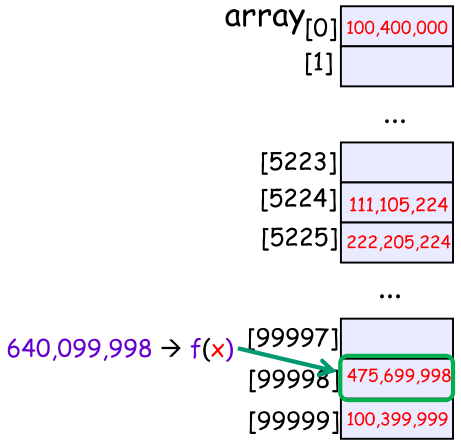
Linear Probing Insertion:

Sometimes, you'll need to insert an item near the end of the table...

For instance, let's say we want to insert a new value of 640,099,998 into our hash table.

If you run into a collision on the last bucket, and go past the end...

You simply wrap back around the top!



Closed Hash Table with Linear Probing: Searching

Linear Probing Searching:

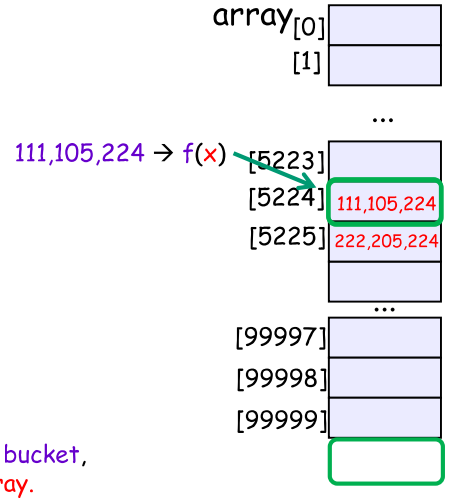
To search our hash table, we use a similar approach.

We compute a target bucket number with our hash function.

We then look in that bucket for our value. If we find it, great!

If we don't find our value, we probe linearly down the array until we either find our value or hit an empty bucket.

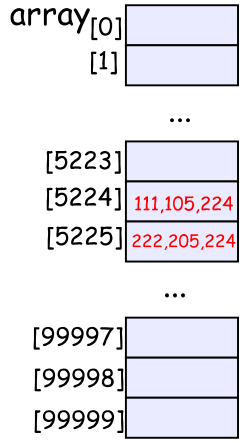
If while probing, you run into an empty bucket, it means: your value isn't in the array.



Closed Hash Table with Linear Probing

This approach addresses collisions by putting each value as close as possible to its intended bucket.

Since we store every original value (e.g., 111,105,224) in the array, there is no chance of ambiguity.



Closed Hash Table with Linear Probing

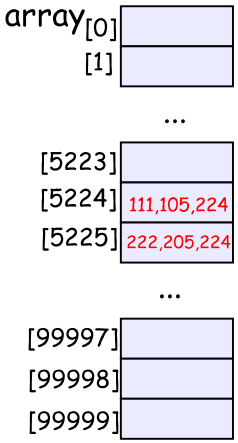
So why do we call this a "Closed" hash table???

Since our data is stored in a fixed-size array, there are a fixed (closed) number of buckets for us to put values.

Once we run out of empty buckets, we can't add new values...

Linked lists and binary search trees don't have this problem!

Ok, let's see the C++ code now!



Linear Probing Hash Table: The Details

In a Linear Probing Hash Table, each bucket in the array is just a C++ struct.

Each bucket holds two 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.

```
struct BUCKET
{
   // a bucket stores a value (e.g. an ID#)
   int idNum;

   bool          used;  // is bucket in-use?
};
```

If this field is false, it means that this Bucket in the array is empty. If the field is true, then it means this Bucket is already filled with valid data.

25

```
#define NUM_BUCK    10

class HashTable
{
public:
  void insert(int idNum)
  {
     int bucket = hashFunc(idNum);

     for (int tries=0;tries<NUM_BUCK;tries++)
     {
        if (m_buckets[bucket].used == false)
        {
           m_buckets[bucket].idNum = idNum;
           m_buckets[bucket].used = true;
           return;
        }
        bucket = (bucket + 1) % NUM_BUCK;
     }
     // no room left in hash table!!!
  }
private:
   int hashFunc(int idNum) const
       {  return idNum % NUM_BUCK;  }

   BUCKET m_buckets[NUM_BUCK];
};
```

First we 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'll store our new item in the first unused bucket that we find, starting with the bucket selected by our hash 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).

Here's our hash function. As before, we compute our bucket number by dividing the ID number by the total # of buckets and then taking the remainder (%).

Our hash table has 10 slots, aka "buckets."

33

What Can you Store in your Hash Table?

Oh, and if you like, you can include additional associated values (e.g., a name, GPA) in each bucket!

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

You can do that!

Now when you look up a student by their ID# you can ALSO get their name and GPA!

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

```
bool search(int id, string &name, float &GPA)
{
  int bucket = hashFunc(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;// not in the hash table
```

34

Linear Probing: Deleting?

So far, we've seen how to insert items into our Linear Probe hash table.

What if we want to delete a value from our hash table?

Let's take a naïve approach and see what happens…

To delete the value, let's just zero out our value and set the used field to false…

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.

So, as you can see, if we simply delete an item from our hash table, we have problems!

There are ways to solve this problem with a Linear Probing hash table, but they're not recommended!

0	idNum: 79	used: T
1	idNum:	used: f
2	idNum:	used: f
3	idNum:	used: f

So, in summary, only use Closed/Linear Probing hash tables when you don't intend to delete items from your hash table.

Like if you're building a hash table that holds words for a dictionary…

You'll just add words, never delete any, right?

36

The "Open" Hash Table

How about searching our Open hash table?

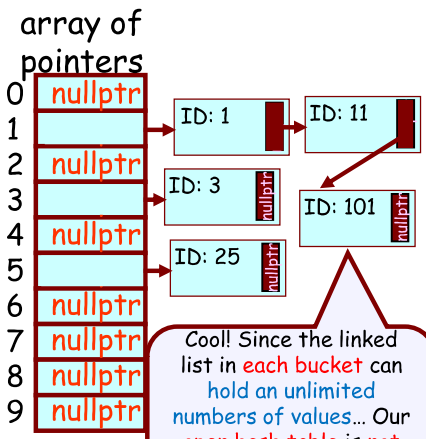
Idea: Instead of storing our values directly in the array, each array bucket points to a linked list of values.

To search for an item:

1. As before, compute a bucket # with your hash function:

   ```
   bucket = hashFunc(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!



Cool! Since the linked list in each bucket can hold an unlimited numbers of values… Our open hash table is not size-limited like our closed one!

Insert the following values: 1, 3, 11, 25, 101

The "Open" Hash Table: Deletions

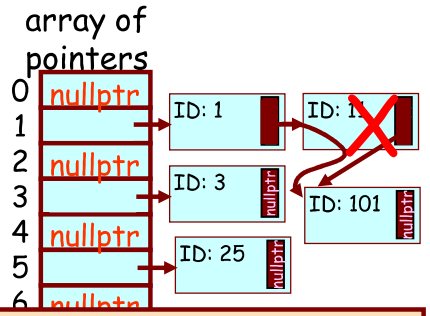
Question:
How do you delete an item from an open hash table?

Answer:
You just remove the value from the linked list.

Let's delete the student with ID=11 and see what happens…

Cool! Unlike a closed hash table, you can easily delete items from an open hash table!

Oh – and there's no reason why we have to use a linked-list to deal with collisions…



If you plan to repeatedly insert and delete values into the hash table, then the Open table is your best bet!

Also, you can insert more than N items into your table and still have great performance!

Hash Table Efficiency

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

Answer:

It depends upon:

(a) The type of hash table (e.g., closed vs. open),

(b) how full your hash table is, and

(c) how many collisions you have in the hash table.

Hash Table Efficiency

Let's assume we have a completely (or nearly) empty hash table…

What's the maximum number of steps required to insert a new value ?

Right! There's zero chance of collision, so we can add our new value in one step!

And finding an item in a nearly-empty hash table is just as fast!

We have no collisions so either we find an item right away or we know it's not in the hash table…

0	idNum: -1 Name:	GPA: etc…
1	idNum: -1 Name:	GPA: etc…
2	idNum: -1 Name:	GPA: etc…
3	idNum: -1 Name:	GPA: etc…
4	idNum: -1 Name:	GPA: etc…
5	idNum: -1 Name:	GPA: etc…
6	idNum: -1 Name:	GPA: etc…
7	idNum: -1 Name:	GPA: etc…
8	idNum: -1 Name:	GPA: etc…
9	idNum: -1 Name:	GPA: etc…

Hash Table Efficiency

Ok, but what if our hash table is nearly full?

What's the maximum number of steps required to insert a new value ?

Right! It could take up to N steps!

And searching can take just as long in the worst case…

So technically, a hash table can be up to O(N) when it's nearly full!

So how big must we make our hash table so it runs quickly? To figure this out, we first need to learn about the "load" concept…

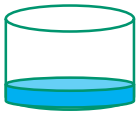
0	idNum: 89 Name: Tad	GPA: 3.87 etc…
1	idNum: 21 Name: Abe	GPA: 4.0 etc…
2	idNum: 12 Name: Ben	GPA: 3.2 etc…
3	idNum: 42 Name: Liz	GPA: 3.9 etc…
4	idNum: 34 Name: Al	GPA: 1.10 etc…
5	idNum: Name:	GPA: etc…
6	idNum: 06 Name: Jill	GPA: 3.89 etc…
7	idNum: 67 Name: Hoa	GPA: 3.4 etc…
8	idNum: 78 Name: Bill	GPA: 1.7 etc…
9	idNum: 29 Name: Nat	GPA: 2.1 etc…

Hash Table Efficiency: The Load Factor

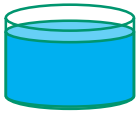
The "load" of a hash table is the maximum number of values you intend to add divided by the number of buckets in the array.

$$L = \frac{\text{Max \# of values to insert}}{\text{Total buckets in the array}}$$

Example: A load of L=.1 means your 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 w/Linear Probing Efficiency

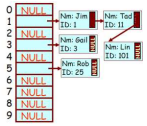
Given a particular load L for a Closed Hash Table w LP, it's easy to compute the average # of tries it'll take you to insert/find an item:

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

So, if your closed hash table has a

load factor of	your search will take
.10 (your array is 10x bigger than required)	~1.05 searches
.20 (your array is 5x bigger than required)	~1.12 searches
.30 (your array is 3x bigger than required)	~1.21 searches
...	
.70 (your array is 30% bigger than required)	~2.16 searches
.80 (your array is 20% bigger than required)	~3.00 searches
.90 (your array is 10% bigger than required)	~5.50 searches

Open Hash Table Efficiency



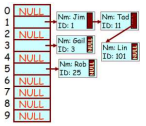
Given a particular load L for an Open Hash Table, it's also easy to compute the average # of tries to insert/find an item:

Average # of Checks = 1 + L/2

So, if your open hash table has a

load factor of	your search will take
.10 (your array is 10x bigger than required)	~1.05 searches
.20 (your array is 5x bigger than required)	~1.10 searches
.30 (your array is 3x bigger than required)	~1.15 searches
...	
.70 (your array is 30% bigger than required)	~1.35 searches
.80 (your array is 20% bigger than required)	~1.40 searches
.90 (your array is 10% bigger than required)	~1.45 searches

Closed vs. Open Hash Table



Closed Hash w/L.P.		Open Hash	
Load	Avg Steps	Load	Avg Steps
.10	~1.05 searches	.10	~1.05 searches
.20	~1.12 searches	.20	~1.10 searches
.30	~1.21 searches	.30	~1.15 searches
	...		...
.70	~2.16 searches	.70	~1.35 searches
.80	~3.00 searches	.80	~1.40 searches
.90	~5.50 searches	.90	~1.45 searches

Moral: Open hash tables are almost ALWAYS more efficient than Closed hash tables!

Sizing your Hash Table

Challenge:

If you want to store up to 1000 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?

Remember: Expected # of Checks = 1 + L/2

If our hash table has 2000 buckets and we're inserting a maximum of 1000 values, we are guaranteed to have an average of 1.25 steps per insert/search!

This result means: "If you want to be able to find/insert items into your open hash table in an average of 1.25 steps, you need a load of .5, or roughly 2x more buckets than the maximum number of values you'll put into your table."

Answer:

Part 1: Set the equation above equal to 1.25 and solve for L:

1.25 = → .25 = L/2 → .5 = L

Part 2: Use the load formula to solve for "Required size":

$$L = \frac{\text{\# of items to insert}}{\text{Required hash table size}} \rightarrow .5 = \frac{1000}{\text{Required hash table size}} \rightarrow \text{Required hash table size} = \frac{1000}{.5}$$

= 2000 buckets

So basically it's a tradeoff!

You could always use a really big hash table with way-too-many buckets and ensure really fast searches…

But then you'll end up wasting lots of memory…

On the other hand, if you have a really small hash table (with just barely enough room), it'll be slower.

Finally, when choosing the exact size of your hash table (the number of buckets)…

Always try to choose a prime number of buckets…

Instead of 2000 buckets, give your hash table 2017 buckets.

This causes more even distribution and fewer collisions!

What Happens If…

What happens if we want to allow the user to search by the student's name instead of their ID number?

Well, our original hash function function won't quite work:

```
int hashFunc(int ID)
{
   return(ID % 100000)
}
```

```
int hashFunc(string &name)
{
   // what do we do?
}
```

Now we need a hash function that can convert from a string of letters to a number between 0 and N-1.

A Hash Function for Strings

Here's one possibility for a hash function that can convert a string into a number between 0 and N-1.

```
int hashFunc(string &name)
{
  int i, total=0;

  for (i=0;i<name.length(); i++)
        total = total + name[i];

  total = total % HASH_TABLE_SIZE;

  return(total);
}
```

Hint:

What happens if we hash "BAT"?

What happens if we hash "TAB"?

But this hash function isn't so good. Why not?

How can we fix it?

A Better Hash Function for Strings

Here's better version of our string hashing function – while not perfect, it disperses items more uniformly in the table.

```
int hashFunc(string &name)
{
   int i, total=0;

   for (i=0;i<name.length(); i++)
         total = total + (i+1) * name[i];

   total = total % HASH_TABLE_SIZE;

   return(total);
}
```

Now "BAT" and "TAB" hash to different slots in our array since this version takes character position into account.

A GREAT Hash Function for Strings

Rather than write your own hash function from scratch, why not use one written by the pros?

C++ provides a great string hashing function:

Make sure to #include<functional> to use C++'s hash function!

We'll define our own hash function, but leverage C++'s algorithm under the hood.

First you define a C++ string hashing object.

This returns a hash value between 0 and 4 billion.

Then you use the object to hash your input string.

Finally, you apply your own modulo function and return a bucket # that fits into your hash table's array.

```
#include <functional>

unsigned int yourHashFunction(const std::string &hashMe)
{
   std::hash<std::string> str_hash;                  // creates a string hasher!
   unsigned int hashValue = str_hash(hashMe);        // now hash our string!
   ...hen just add your own modulo
   unsigned int bucketNum = hashValue % NUM_BUCKETS;
   return bucketNum;
}
```


Writing Your Own Hash Function

Great! But what if you need to write a hash function for some non-standard data type?

```
unsigned int yourHashFunction(const SomeCrazyTypeOfData &hashMe)
{
                ????
}
```

Like hashing...

Geospatial coordinates
An array of N numbers
The contents of a data file

This is a non-trivial exercise!

You really need to understand the "nature" of the data you're hashing...

Then design your algorithm, analyze it, and iterate.

The unordered_map: A hash-based version of a map

```
#include <unordered_map>
#include <iostream>
#include <string>
using namespace std::tr1; // required for a hash-based map
using namespace std;

int main( )
{
   unordered_map <string,int> hm;              // define a new U_M
   unordered_map <string,int>::iterator iter; // define an iterator for a U_M
   hm["Carey"] = 10;                          // insert a new item into the U_M
   hm["David"] = 20;
   iter = hm.find("Carey");                   // find Carey in the hash map
   if (iter == hm.end())                      // did we find Carey or not?
      cout << "Carey was not found!";         // couldn't find "Carey" in the hash map
   else
   {
       cout << "When we look up " << iter->first;      // "When we look up Carey"
       cout << " we find " << iter->second;           // "we find 10"
   }
}
```

Hash Tables vs. Binary Search Trees

	Hash Tables	Binary Search Trees
Speed	O(1) regardless of # of items	$O(\log_2 N)$
Simplicity	Easy to implement	More complex to implement
Max Size	Closed: Limited by array size Open: Not limited, but high load impacts performance	Unlimited size
Space Efficiency	Wastes a lot of space if you have a large hash table holding few items	Only uses as much memory as needed (one node per item inserted)
Ordering	No ordering (random)	Alphabetical ordering

"Tables"

Let's say you want to want to write a program to keep track of all your BFFs…

Of course, you want to remember all the important dirt about each BFF:

And you want to quickly be able to search for a BFF in one or more ways…

" Find all the dirt on my BFF 'David Johansen' "

" Find all the dirt on the BFF whose number is 867-5309 "

Name: Carey Nash
Phone number: 867-5309
Birthday: July 28
iPhone or 'droid: iPhone
Social Security #: 111222333
Favorite food: …

"Tables"

In CS lingo, a group of related data is called a "record."

Each record has a bunch of "fields" like Name, Phone #, Birthday, etc. that can be filled in with values.

If we have a bunch of records, we call this a "table." Simple!

While you may have many records with the same Name field value (e.g., John Smith) or the same Birthday field value (e.g., Jan 1st)…

Some fields, like Social Security Number, will have unique values across all records - this type of field is useful for searching and finding a unique record!

A BFF Record

Name: Carey Nash
Phone number: 867-5309
Birthday: July 28
iPhone or 'droid: iPhone
Social Security #: 111222333
Favorite food: …

Table of BFF Records

Name: David Small

Name: John Rohr
Phone number: 999-9191
Birthday: Jan 1
iPhone or 'droid: Droid
Social Security #: 47372727
Favorite food: …

A field (like the SSN) that has unique values across all records is called a "key field."

Implementing Tables

How could you create a record in C++?

Answer: Just use a struct or class to represent a record of data!

How can you create a table in C++?

Answer: You can simply create an array or vector of your struct!

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

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

```
// algorithm to search by the name field
int SearchByName(vector<Student> &table, string &findName)
{
  for (int s = 0; s < table.size(); s++ )
    if (findName == table[ s ].name)
          return( s );// the student you're looking for is in slot s
  return( -1 );                    // didn't find that student in your table
}
```

```
// algorithm to search by the phone field
int SearchByPhone(vector<Student> &table, string &findPhone)
{
  for (int s = 0; s < table.size(); s++ )
    if (findPhone == table[ s ].phone)
          return( s );// the student you're looking for is in slot s
  return( -1 );                    // didn't find that student in your table
}
```

Implementing Tables

Heck, why not just create a whole C++ class for our table?

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

```
class TableOfStudents
{
public:
   TableOfStudents();      // construct a new table
   ~TableOfStudents();     // destruct our table
   void addStudent(Student &stud);  // add a new Student
   Student getStudent(int s);   // retrieve Students from slot s
   int searchByName(string &name);  // name is a searchable field
   int searchByPhone(int phone);  // phone is a searchable field
   ...
private:

   vector<Student> m_students;
};
```


Tables

In the TableOfStudents class, we used a vector to hold our table and a linear search to find Students by their name or phone.

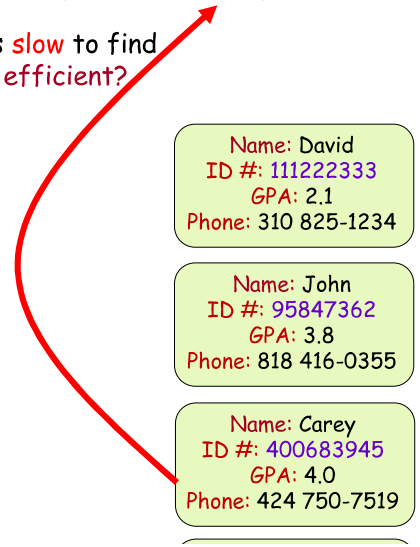
This is a perfectly valid table – but it's slow to find a student! How can we make it more efficient?

Well, we could alphabetically sort our vector of records by their names…

Then we could use a binary search to quickly locate a record based on a person's name.

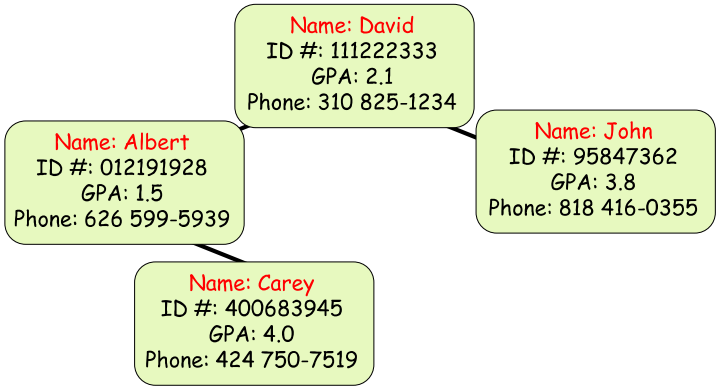
But then every time we add a new record, we have to re-sort the whole table. Yuck!

And if we sort by name, we can't search efficiently by other fields like phone # or ID #!



Tables

Hmmm… What if we stored our records in a binary search tree (e.g., a map) organized by name? Would that fix things?

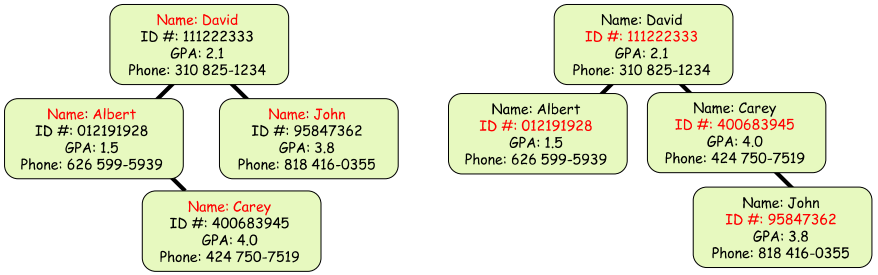


Well, now we can search the table efficiently by name…

But we still can't search efficiently by ID# or Phone #….

Tables

Hmmm… What if we create two tables, ordering the first by name and the second by ID#?



That works… Now I can quickly find people by name or ID#!

But now we have two copies of every record, one in each tree! If the records are big, that's a waste of space!

So what can we do? Let's see!

Making an Efficient Table

1. We'll still use a vector to store all of our records…
2. Let's also add a data structure that lets us associate each person's name with their slot # in the vector…
3. And we can add another data structure to associate each person's ID # with their slot # too!

m_students

0 name: Alex GPA: 2.05 ID: 7124 …

1 name: Linda GPA: 3.99 ID: 0003 …

2 name: Jason GPA: 1.55

```
class TableOfStudents
{
public:
   TableOfStudents();
   ~TableOfStudents();
   void addStudent(Student &stud);
   Student getStudent(int s);
   int searchByName(string &name);
   int searchByPhone(int phone);
private:
   vector<Student> m_students;
   map<string,int>  m_nameToSlot;
   map<int,int>     m_idToSlot;
   map<int,int>     m_phoneToSlot;
};
```

Our second data structure lets us quickly look up a name and find out which slot in the vector holds the related record.

Our third data structure lets us quickly look up an ID# and find out which slot in the vector holds the related record.

These secondary data structures are called "indexes." Each index lets us efficiently find a record based on a particular field. We may have as many indexes as we need for our application.

Making an Efficient Table

So what does method lo

Well, we have to add our new student record to our vector just like before.

But now, every time we add a record, we've also got to add the name to slot # mapping to our first map!

```
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;   // maps name to slot #
   m_idToSlot[stud.IDNum] = slot;    // maps ID# to slot #
}
```

Finally, every time we add a record, we've also got to add the ID# to slot # mapping to our second map!

Using Hashing to Speed Up Tables

Can we use hash tables to index our data instead of binary search trees?

Of course!

Now we can have O(1) searches by name! Cool! But in that case why not just always use hash tables to index all of our key fields?

Answer: Because hash tables store the data in an essentially random order.

While a BST is slower, it does order the key fields in alphabetical order…

For instance, what if we want to be able to print out all students alphabetically by their name.

If our index data structure is a binary search tree, that's easy!

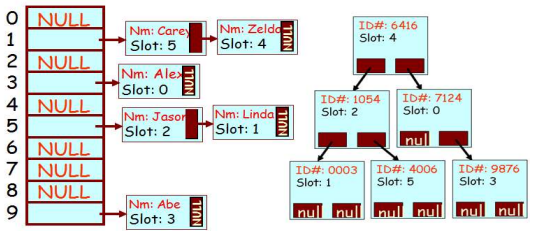
If we indexed with a hash table, we'd have to do a lot more work to do the same thing…

0 name: Alex GPA: 2.05 ID: 7124 …

1 name: Linda GPA: 3.99 ID: 0003 …

2 name: Jason GPA: 1.55 ID: 1054 …

3 name: Abe GPA: 4.00 ID: 9876



Moral: You need to understand how your table will be used to determine how to best index each field.

For example:

I'd use a BST for the name field so I can print people's names in alphabetical order. But I'd use a hash table for the phone field, cause I just need to search quickly but I don't need to order records by their phone #.