# CS163: Data structures

Topic 2:
“Table” abstract data type

5/2021

# What will be discussed?

- □ Work by “value” rather than “position”
- □ May be implemented using a variety of data structures such as
  - arrays (statically, dynamically allocated)
  - linear linked lists
  - **non-linear linked lists**
- □ Today, we begin our introduction into non-linear linked lists by examining arrays of linked lists!

# Table ADTs

- The ADT's we have learned about so far are appropriate for problems that must manage data by the position of the data (the ADT operations for an Ordered List, Stack, and Queue are all position oriented).
- These operations insert data (at the ith position, the top of stack, or the rear of the queue); they delete data (at the ith position, the top of stack, or the front of the queue); they retrieve data and find out if the list is full or empty.

# Table ADTs

- Tables manage data by its value!
- As with the other ADT's we have talked about, table operations can be implemented using arrays or linked lists.
- Valued Oriented ADTs allow you to:
  - -- insert data containing a certain VALUE into a data structure
  - -- delete a data item containing a certain VALUE from a data structure
  - -- retrieve data and find out if the structure is empty or full.

# Table ADTs

- Applications that use value oriented ADTs are:
  - ...finding the phone number of John Smith
  - ...deleting all information about an employee with an ID # 4432

# Table ADTs

- □ When you think of an ADT table, think of a table of major cities in the world including the city/country/population, a table of To-Do-List items, or a table of addresses including names/addresses/phone number/birthday.
- □ Each entry in the table contains several pieces of information.
- □ It is designed to allow you to look up information **by various search keys**

# Table ADTs

- The basic operations that define an ADT Table are: *(notice, various search keys!!)*
  - Create an empty table (e.g., Create(Table))
  - Insert an item into the table (e.g., Insert(Table,Newdata))
  - Delete an item from the table (e.g., Delete(Table, Key))
  - Retrieve an item from the table (e.g., Retrieve(Table, Key, Returneddata))

# Table ADTs

- But, just like before, you should realize that these operations are only one possible set of table operations.
- Your application might require either a subset of these operations or other operations not listed here.
- Or, it might be better to modify the definitions...to allow for duplicate items in a table.

# Table ADTs

- □ Does anyone see a problem with this approach so far?
- □ What if we wanted to print out all of the items that are in the table? Let's add a traverse.:
- □ Traverse the Table (e.g., Traverse(Table, VisitOrder))
- □ But, because the data is not defined to be in a given order...”sorting” may be required (the client should be unaware of this taking place)

# Data Structures for Tables

- We will look at both array based and pointer based implementations of the ADT Table.
- When we say linear, we mean that our items appear one after another...like a list.
- We can either organize the data in sorted order or not.
- If your application frequently needs a key accessed in sorted order, then they should be stored that way. But, if you access the information in a variety of ways, sorting may not help!

# Data Structures for Tables

- With an unsorted table, we can save information at the end of the list or at the beginning of the list;
- therefore, insert is simple to implement: for both array and pointer-based implementations.
- For an unsorted table, it will take the same amount of time to insert an item regardless of how many items you have in your table.

# Data Structures for Tables

- The only advantage of using a pointer-based implementation is if you are unable to give a good estimate of the maximum possible size of the table.
- Keep in mind that the space requirements for an array based implementation are slightly less than that of a pointer based implementation....because no explicit pointer is stored.

# Data Structures for Tables

- For sorted tables (which is most common), we organize the table in regard to one of the fields in the data's structure.
- Generally this is used when insertion and deletion is rare and the typical operation is traversal (i.e., your data base has already been created and you want to print a list of all of the high priority items). Therefore, the most frequently used operation would be the Traverse operation, sorting on a particular key.

# Data Structures for Tables

- For a sorted list, you need to decide:
  - Whether dynamic memory is needed or whether you can determine the maximum size of your table
  - How quickly do items need to be located given a search key
  - How quickly do you need to insert and delete

# Data Structures for Tables

- So, have you noticed that we have a problem for sorted tables?
- Having a dynamic table requires pointers.
- Having a good means for retrieving items requires arrays.
- Doing a lot of insertion and deletion is a toss up...probably an array is best because of the searching.
- So, what happens if you need to DO ALL of these operations?

# Searching for Data

- Searching is typically a high priority for table abstractions, so let’s spend a moment reviewing searching in general.
- Searching is considered to be "invisible" to the user.
- It doesn't have input and output the user works with.
- Instead, the program gets to the stage where searching is required and it is performed.

# Searching for Data

- □ In many applications, a significant amount of computation time is spent sorting data or searching for data.
- □ Therefore, it is really important that you pick an efficient algorithm that matches the tasks you are trying to perform. Why?
- □ Because some algorithms to sort and search are much slower than others, especially when we are dealing with large collections of data.

# Searching for Data

- When searching, the fields that we search to be able to find a match are called search keys (or, a key...or a target).
- Searching algorithms may be designed to search for any occurrence of a key, the first occurrence of a key, all occurrences of a key, or the last occurrence of a key.
- To begin with, our searching algorithms will assume only one occurrence of a key.

# Searching for Data

- Searching is either done internally or externally.
- Searching internally means that we will search an list of items for a match; this might be searching an array of data items, an array of structures, or a linked list.
- Searching externally means that a file of data items needs to be searched to find a match

# Searching for Data

- Searching algorithms will typically be modularized into their own function(s)...which will have two input arguments:
  - (1) The key to search for (target)
  - (2) The list to search
- and, two output arguments:
  - (1) A boolean indicating success or failure (did we find a match?)
  - (2) The location in the list where the target was found; generally if the search was not successful the location returned is some undefined value and should not be used.

# Searching for Data

- The most obvious and primitive way to search for a given key is to start at the beginning of the list of data items and look at each item in sequence.
- This is called a sequential or linear search.
- The sequential search quits as soon as it finds a copy of the search key in the array. If we are very lucky, the very first key examined may be the one we are looking for. This is the best possible case.
- In the worst case, the algorithm may search the entire search area - from the first to the last key before finding the search value in the last element -- or finding that it isn't present at all.

# Searching for Data

- For a faster way to perform a search, you might instead select the binary search algorithm. This is similar to the way in which we use either a dictionary or a phone book.
- As you should know from CS162, this method is one which divides and conquers. We divide the list of items in two halves and then "conquer" the appropriate half! You continue doing this until you either find a match or determine that the word does not exist!

# Searching for Data

- □ Thinking about binary search, we should notice a few facts:
- □ #1) The binary search is NOT good for searching linked lists. Because it requires jumping back and forth from one end of the list to the middle; this is easy with an array but requires tedious traversal with a linear linked list.
- □ #2) The binary search REQUIRES that your data be arranged in sorted order! Otherwise, it is not applicable.

# Non-Linear Data Structures

- Instead of using an array or a linear linked list for our Table ADT
- We could have used a non-linear data structure, since the client is not aware of the order in which the data is stored
- Our first look at non-linear data structures will be as a hash table, implemented using an array of linear linked lists.

# Hash tables

- For example, we could treat the searching as a “black box”
  - ... like an "address calculator" that takes the item we want to search for and returns to us the exact location in our table where the item is located. Or, if we want to insert something into the table...we give this black box our item and it tells us where we should place it in the table. First of all, we need to learn about hash tables
    - when you see the word “table” in this context, think of just a way of storing data rather than the “adt” itself.

# Hash tables

- So, using this type of approach...to insert would simply be:
  - Step 1: Tell the "black box" (formally called the hashing function) the search key of the item to be inserted
  - Step 2: The "black box" returns to us the location where we should insert the item (e.g., location i)
  - Step 3: `Table[location i] = newitem`

# Hash tables

- For Retrieve:
  - Step 1: Tell the "black box" the search key of the item to be retrieved
  - Step 2: The "black box" returns to us the location where it should be located if it is in the table (e.g., location i)
  - Step 3: Check If our Table[location i] matches the search key....if it does, then return the record at that location with a TRUE success!
    - ...if it does not, then our success is FALSE (no match found).

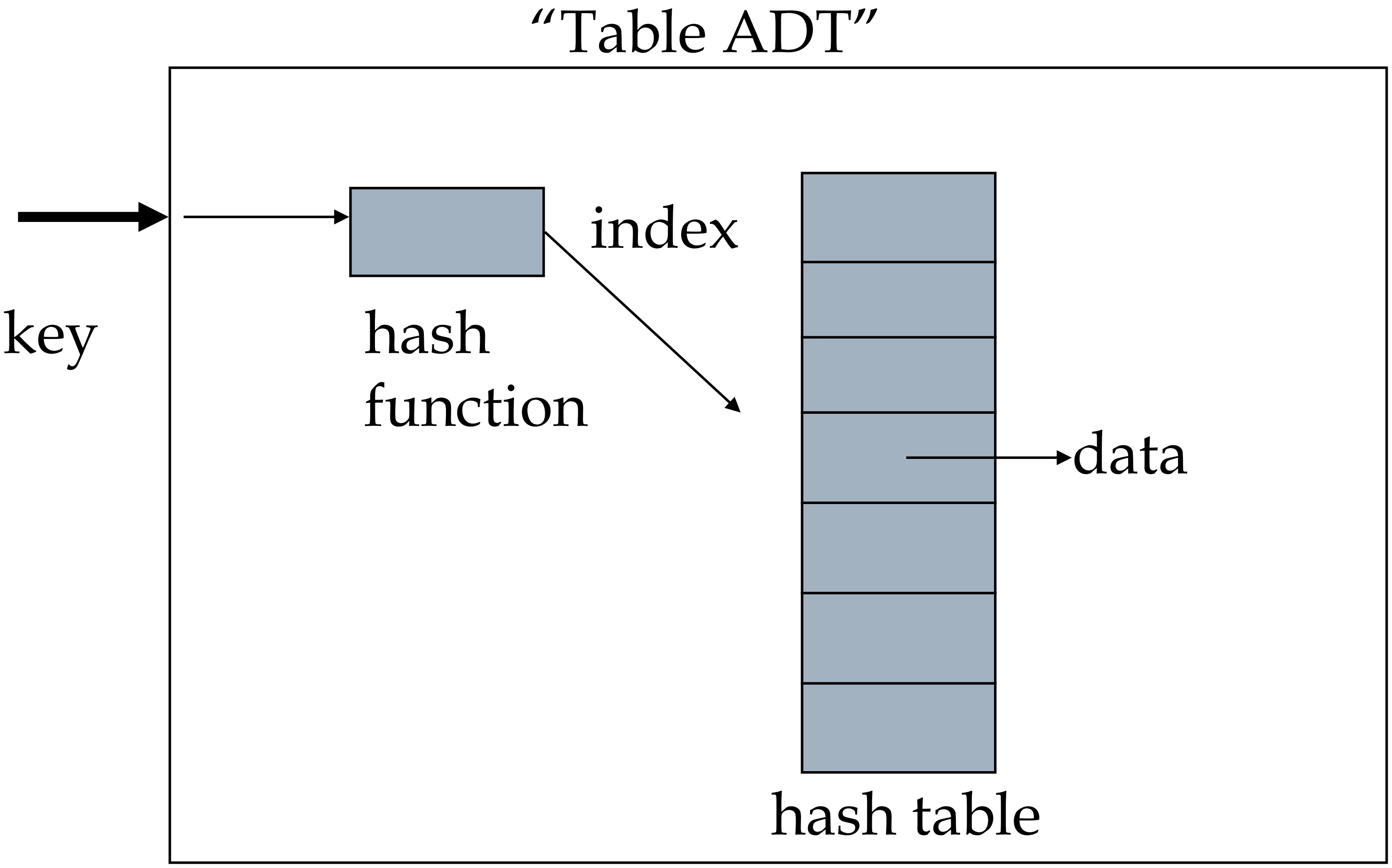
# Hash tables

- Delete would work the same way. Notice we never have to search for an item.
- The amount of time required to carry out the operation depends only on how quickly the black box can perform its computation!
- To implement this approach, we need to be able to construct a black box, which we refer to formally as the hash function. The method we have been describing is called hashing. The Table containing the location of our structures...is called the hash table.

# Hash tables

- Assume we have a Table of structures with locations 0 thru 100.
- Say we are searching for employee id numbers (positive integers).
- We start with a hash function that takes the id number as input and maps it to an integer ranging from 0 to 100.
- So, the ADT operations performs the following:
  - `TableIndex=hashfunction(employee id #)`

# Hash tables



# Hash tables

- Notice Hashing uses a table to store and retrieve items.
- This means that off hand it looks like we are once again limited to a fixed-size implementation.
- The good news is that we will learn in this class about chaining...which will allow us to grow the table dynamically.

# Hash tables

- Ideally we want our hash function to map each search key into a unique index into our table.
- This would be considered a perfect hash function. But, it is only possible to construct perfect hash functions if we can afford to have a unique entry in our table for each search key -- which is not typically the case.
- This means that a typical hash function will end up mapping two or more search keys into the same table index! Obviously this causes a collision.

# Hash tables

- Another way to avoid collisions is to create a hash table which is large enough so that each
- For example, for employee id numbers (###-##-###) ... we'd need to have enough locations to hold numbers from 000-00-000 to 999-99-999.
- Obviously this would require a TREMENDOUS amount of memory!
- And, if we only have 100 employees -- our hash table FAR EXCEEDS the number of items we really need to store. RESERVING SUCH VAST amounts of memory is not practical.

# Collision Resolution

- As an alternative, we need to understand how to implement schemes to resolve collisions.
- Otherwise, hashing isn't a viable searching algorithm.
- When developing a hash functions:
  - try to develop a function that is simple & fast
  - make sure that the function places items evenly throughout the hash table without wasting excess memory
  - make the hash table large enough to be able to accomplish this.

# Hash Functions

- Hash functions typically operate on search keys that are integers.
- It doesn't mean that you can't search for a character or for a string...but what it means is that we need to map our search keys to integers before performing the hashing operation. This keeps it as simple as possible
- But, be careful how you form the hash functions
  - your goal should be to keep the number of collisions to a minimum and avoid clustering of data

# Hash Functions

□ For example, suppose our search key is a 9-digit employee id number. Our hash function could be as simple as selecting the 4th and last digits to obtain an index into the hash table:

□ Therefore, we store the item with a search key 566-99-3411 in our hash table at index 91.

□ In this example, we need to be careful which digits we select. Since the first 3 digits of an ID number are based on geography, if we only selected those digits we would be mapping people from the same state into the same location

# Hash Functions

- Another example would be to "fold" our search keys: adds up the digits.
- Therefore, we store the item with a search key 566-99-3411 in our hash table at index 44.
- Notice that if you add up all of the digits of a 9 digit number...the result will always be between 0 and 9+9+9+9+9+9+9+9+9   which is 81!
- This means that we'd only use indices 0 thru 81 of our hash table. If we have 100 employees...this means we immediately know that there will be some collisions to deal with.

# Hash Functions

- Another example would be to "fold" our search keys a different way. Maybe in groups of 3:
    - 566+993+411 = 1,970
- Obviously with this approach would require a larger hash table. Notice that with a 9 digit number, the result will always be between 0 and 999+999+999 which is 2,997.
- If you are limited to 101 locations in a hash table, a mapping function is required. We would need to map 0 -> 2997 to the range 0 -> 100.

# Hash Functions

- But the most simple AND most effective approach for hashing is to use modulo arithmetic (finding an integer remainder after a division).
- All we need to do is use a hash function like:
  - employee id # % Tablesize
  - 566-99-3411 % 101 results in a remainder of 15
- This type of hash function will always map our numbers to range WITHIN the range of the table. In this case, the results will always range between 0 and 100!

# Hash Functions

- Typically, the larger the table the fewer collisions. We can control the distribution of table items more evenly, and reduce collisions, by choosing a prime number as the Table size. For example, 101 is prime.
- Notice that if the tablesize is a power of a small integer like 2 or 10...then many keys tend to map to the same index. So, even if the hash table is of size 1000...it might be better to choose 997 or 1009.

# Hash Functions

- When two or more items are mapped to the same index after
- using a hash function, a collision occurs. For example, what if our two employee id #s were: 123445678 and 123445779
  - ...using the % operator, with a table of size 101...both of these map to index 44.
  - This means that after we have inserted the first id# at index 44, we can't insert the second id # at index 44 also!

# Linear Probing

- There are many techniques for collision resolution
- We will discuss
  - linear probing
  - chaining
- Linear probing searches for some other empty slot...sequentially...until either we find a empty place to store our new item or determine that the table is full.

# Linear Probing

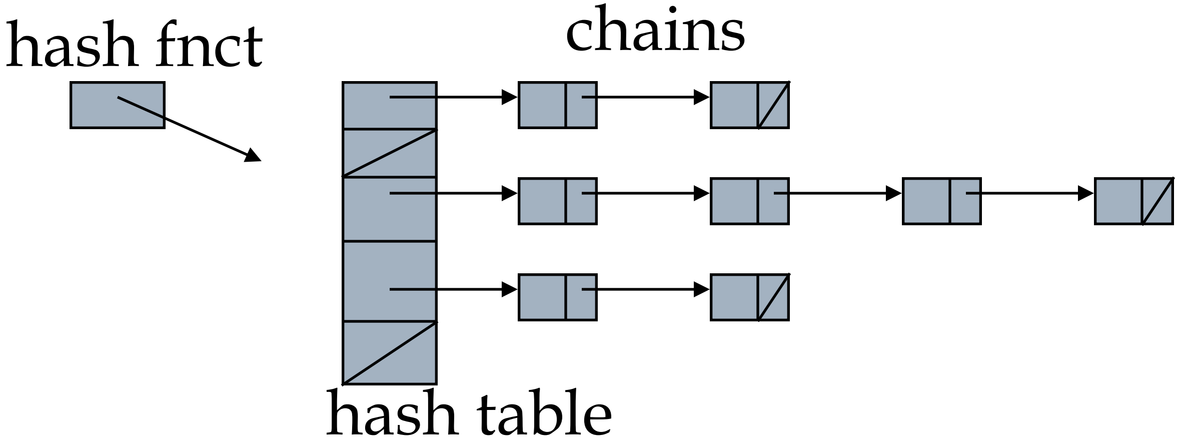
- With this scheme,
  - we search the hash table sequentially, starting at the original hash location...
  - Typically we WRAP AROUND from the last table location to the first table location if we have trouble finding a spot.
  - Notice what this approach does to retrieving and deleting items. We may have to make a number of comparisons before being able to determine where an item is located...or to determine if an item is NOT in the table.

# Linear Probing

- Is this an efficient approach?
- As we use up more and more of the hash table, the chances of collisions increase.
- As collisions increase, the number of probes increase, increasing search time
- Unsuccessful searches will require more time than a successful search.
- Tables end up having items CLUSTER together, at one end/area of the table, affecting overall efficiency.

# Chaining

- A better approach is to design the hash table as an array of linked lists.
- Think of each entry in your table as a chain...or a pointer to a linked list of items that the hash function has mapped into that location.



# Chaining

□ Think of the hash table as an array of head pointers:

```
node * hash_table[101];
```

□ But, what is wrong with this? It is a statically allocated hash table...

```
node ** hash_table;
...
hash_table = new node * [tbl_size];
```

# Chaining

- Do we need to initialize this hash table?
- Yes, each element should be initialized to NULL,
  - since each “head” pointer is null until a chain is established
  - so, your constructor needs a loop setting elements 0 through tbl_size-1 to Null after the memory for the hash table has been allocated

# Chaining

- Then, the algorithm to insert an item into the linked list:
  - Use the hash function to find the hash index
  - Allocate memory for this new item
  - Store the newitem in this new memory
  - Insert this new node between the "head" of this list and the first node in the list
- Why shouldn’t we traverse the chain?
  - Think about the order...

# Chaining

- To retrieve, our algorithm would be:
  - Use the hash function to find the hash index
  - Begin traversing down the linked list searching for a match
  - Terminate the search when either a match is encountered or the end of the linked list is reached.
  - If a match is made...return the item & a flag of SUCCESS!
  - If the end of the list is reached and no match is found, UNSUCCESS

# Chaining

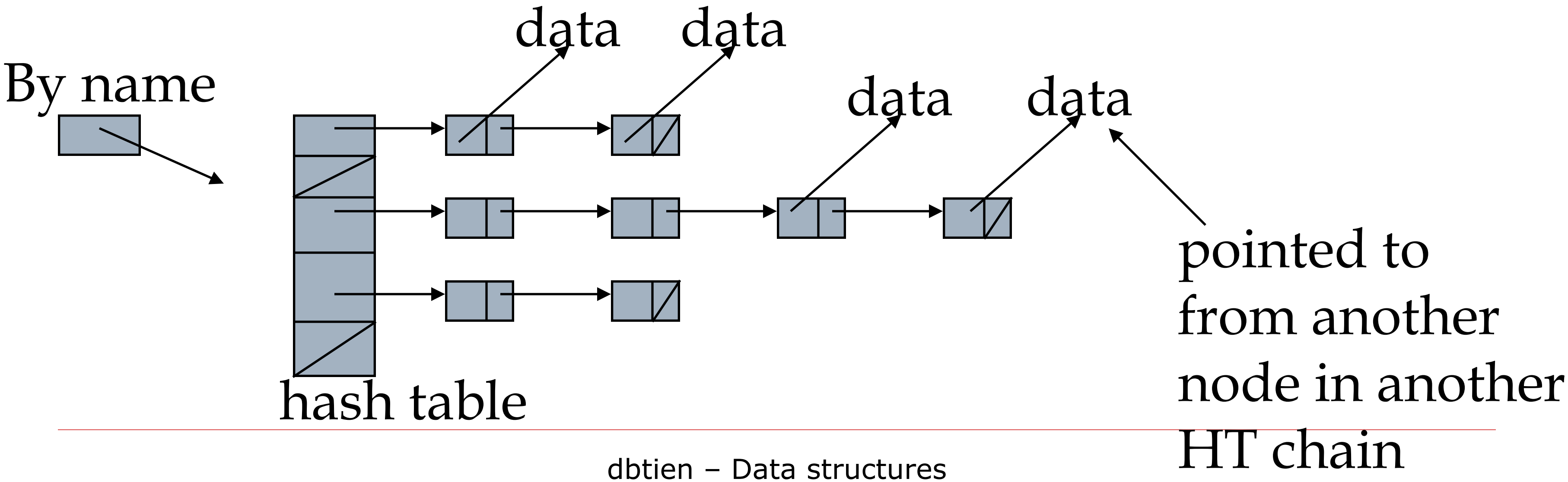
- This approach is beneficial because the total size of the table is now dynamic!
- Each linked list can be as long as necessary!
- And, it still let's our Insert operation be almost instantaneous.
- The problem with efficiency comes with retrieve and delete, where we are required to search a linked list of items that are mapped to the same index in the hash table.

# Chaining

- Therefore, when the linked lists are too long, change the table size and the manner in which the key is converted into an index into the hash table
- Do you notice how we are using a combination of direct access with the array and quick insert/removal with linked lists?
- Notice as well that no data moving happens
- But, what if 2 or more keys are required?

# Chaining

- For 2 or more search keys,
  - you will have 2 or more hash tables (if the performance for each is equally important)
  - where the data within each node actually is a pointer to the physical data rather than an instance



# Chaining

- What if all or most of the items end up hashing to the same location?
- Our linked list(s) could be very large.
- In fact, the worst case is when ALL items are mapped into the same linked list.
- In such cases, you should monitor the number of collisions that happen and change your hash function or increase your table size to reduce the number of collisions and shorten your linked lists.

# In Summary

- For many applications, hashing provides the most efficient way to use an ADT Table.
- But, you should be aware that hashing IS NOT designed to support traversing an entire table of items and have them appear in sorted order.
- Notice that a hash function scatters items randomly throughout the table...so that there is no ordering relationship between search keys and adjacent items in the table

# In Summary

- □ Therefore, to traverse a table and print out items in sorted order would require you to sort the table first.
- □ As we will learn, binary search trees <u>can be</u> far more effective for letting you do both searching and traversal - IN SORTED ORDER than hash tables.

# In Summary

- □ Hash functions should be easy and fast to compute. Most common hash functions require only a single division or multiplication.
- □ Hash functions should evenly scatter the data throughout the hash table.
- □ To achieve the best performance for a chaining method, each chain should be about the same length (Total # items/Total size of table).

# In Summary

- To help ensure this, the calculation of the hash function should INVOLVE THE ENTIRE SEARCH KEY and not just a portion of the key.
- For example, computing a modulo of the entire ID number is much safer than using only its first two digits.
- And, if you do use modulo arithmetic, the Table size should be a prime number.
  - This takes care of cases where search keys are multiples of one another.

# Other Approaches

- □ Another approach is called **quadratic probing**.
- □ This approach significantly reduces the clustering that happens with linear probing: index + $\text{count}^2$
- □ It is a more complicated hash function; it doesn't probe all locations in the table.
  - In fact, if you pick a hash table size that is a power of 2, then there are actually very few positions that get probed w/ many collisions.

# Quadratic probing

- The quadratic probing takes the original hash index and adds successive values of an arbitrary quadratic polynomial until an opened slot is found

Linear probing: H, H+1, H+2, H+3 …, (opened slot)

Quadratic probing: H, H+$1^2$ ,H+$2^2$, H+$3^2$ … (opened slot)

# Other Approaches

- □ We might call a variation of this the Quadratic Residue Search.
- □ This was developed in 1970 as a probe search that starts at the first index using your hash function (probably: key mod hashtablesize).
- □ Then, if there is a collision, take that index and add count$^2$. Then, if there is another collision, take that index and subtract count$^2$.

# Others: Double hashing

- □ Apply the 2nd hash function if the 1st hash function generated the conflict

  (hash1(key) + i*hash2(key))%Table_Size

- □ Keep increasing i until an opened slot found
- □ Hash2 often looks like this

  hash2(key) = Prime – (key%Prime)