SIT221: DATA STRUCTURES & ALGORITHMS

LECTURE #5: HASH TABLES & DICTIONARIES

8th August, 2016

Recap

- What is the best thing about Linked Lists? Insert & Delete
- What is the worst thing about Linked Lists? Linear access
- What is the best thing about array? Random access
  - What is the search key? Array index
  - What is the array index type? Integer
  - What if we want to use an index of other data types? Not possible
- What is the worst thing about arrays? Memory & insert/delete

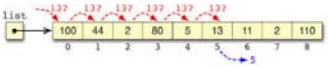
Motivation

- Imagine you have a customer database of 1,000,000 elements, and you want to build a search by **NAME** capability in your program.
- What options you can think of?

1. Linear search

4

- How this works?
  - If you have 10 items in your array?
  - If you have 100 items in your array?
  - What's the general worst case running time? What does this mean?
- What does this mean for the customer dataset? How long does it take to search for a customer?

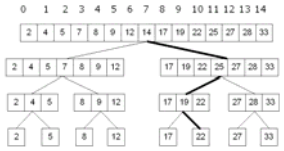


- linear search is not very efficient - O(n)

2. Binary search

5

- How this works?
  - Your data MUST be sorted
  - If you have 10 items in your array?
  - If you have 100 items in your array?
  - What's the worst case running time?
  - What does this mean for the customer dataset?
- How long does it take to find a customer by name?



- Generally, binary search is OK but Sorting - O(log(n)) + Sorting O(nlogn(n))

Can we do better?

6

- What do we need? A data structure that supports
  - Random access in near constant time – we call it O(1) [Like arrays]
  - Search (find element) in constant time
  - Insert in constant time
  - Delete in constant time
  - Index does not need to be integer [problem with arrays]

Dictionaries / Hashtables

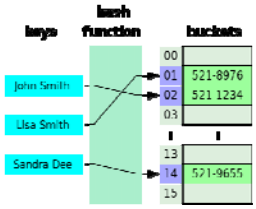
7

- What is the promise of Dictionaries / Hashtables
  - Insert(key, value) in constant time
  - Delete(key) in constant time
  - Contains(key) in constant time
  - ValueAt(key) in constant time
- There are many practical applications. Examples include
  - Symbol table of a compiler.
  - Memory-management tables in operating systems.
  - Large-scale distributed systems.

Hash table data structure

8

- Represents a collection of key/value pairs that are organized based on the hash code of the key.
- A Hashtable object consists of buckets that contain the elements of the collection. A bucket is a virtual subgroup of elements within the Hashtable, which makes searching and retrieving easier and faster than in most collections.
- Each bucket is associated with a hash code, which is generated using a hash function and is based on the key of the element.



Phone book using hashtable

Hash function

- Maps a key k into one of the m slots by taking the remainder of k divided by m. That is,

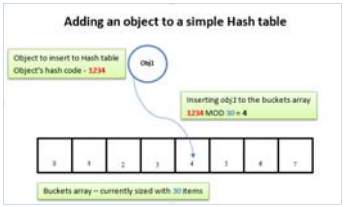
$$h(k) = k \bmod m$$

- Example: $m = 31$ and $k = 78 \Rightarrow h(k) = 16$.
- A hash function must always return the same hash code for the same key.
- **Advantage:** Fast, since it requires just one division operation.
- **Disadvantage:** Have to avoid certain values of m.
  - Don't pick certain values, such as $m=2^p$
  - Or hash won't depend on all bits of k.
- **Good choice for m:**
  - Primes, not too close to power of 2 (or 10) are good.

Hash function – Magic function

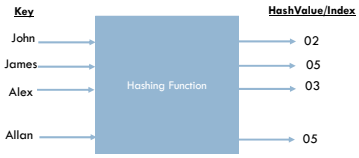
10

- Map key (string or integer value or whatever) to array index (intger) using the array size.
  - index = f(key) % size
- In the example
  - index = f(1234) % 30



Hashing function - example customers database

11



00	Jun, Jun data
01	Dave, Dave data
02	John, John data
03	Alex, Alex data
04	
05	James data Or Allan data ??? **Collision!!!**
06	
07	

- Is it possible to get same index for two different keys? YES
  - It depends on the hash function (good or bad function) and array size.
- What makes a good hashing function?
  - Number of collisions? Uniform distribution of hashcodes? Can easily scale its range

How to write a hashing function

12

- **Modulo-division hashing**
  - Restrict the range of hashcode to a certain size: any number x modulo n (remainder) will be less than n 123%10? **3**
- **Digit extraction hashing**
  - Choose certain digits in your key as the hashcode: 2000123456 → 216
- **Folding method – fold shift**
  - Divide key into equal segments whose size/length matches the target size
  - 2000123456 → 002 + 000 + 123 + 456 → 581
- **Many more techniques available…**

Adding elements to hash table

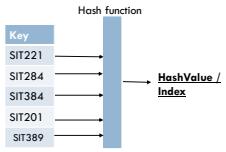
13

- Hashtable() - Initializes a new, empty instance of the Hashtable class using the default initial capacity, load factor, hash code provider, and comparer.
- Elements are added to the Hashtable using the **Hashtable.Add()** method.
- When adding an element to the Hashtable, you must provide
  - the element to be added,
  - unique key by which the element is accessed.
- Both the unique key and the element can be of any type.

Adding elements to hash table

14

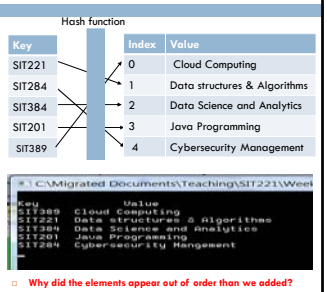
```
static void addHashTable (Hashtable course)
{
    try  {
      course.Add("SIT221", "Data structures & Algorithms");
      course.Add("SIT284", "Cybersecurity Management");
      course.Add("SIT384", "Data Science and Analytics");
      course.Add("SIT201", "Java Programming");
      course.Add("SIT389", "Cloud Computing");
} catch   {
        Console.WriteLine("The same key already exists.");
     }
}
```



Adding elements to a hash table

15

```
static void Main(string[] args)
{
  // Create a new hash table.
    Hashtable course = new Hashtable();
    addHashTable(course);
    Console.WriteLine();
    Console.WriteLine("Key          Value ");
    foreach (DictionaryEntry sit in course)
    {
      Console.WriteLine("{0}  {1}", sit.Key, sit.Value);
    }
   Console.ReadLine();
}
```

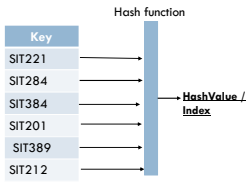


- Why did the elements appear out of order than we added?

Adding elements with exiting 'value' to a hash table

16

```
static void addHashTable (Hashtable course)
{
    try {
      course.Add("SIT221", "Data structures & Algorithms");
      course.Add("SIT284", "Cybersecurity Management");
      course.Add("SIT384", "Data Science and Analytics");
      course.Add("SIT201", "Java Programming");
      course.Add("SIT389", "Cloud Computing");
      course.Add("SIT212", "Data Science and Analytics");
} catch  {
        Console.WriteLine("The same key already exists.");
    }
}
```

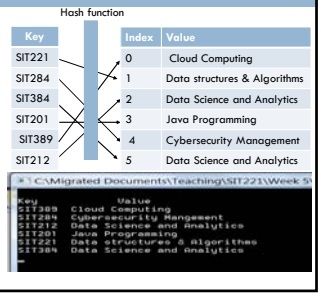


Graphical representation of a hash function

Adding elements with exiting 'value' to a hash table

17

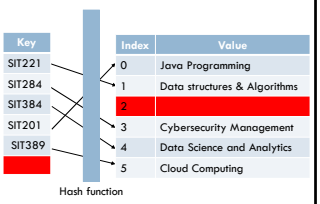
```
static void Main(string[] args)
{
  // Create a new hash table.
    Hashtable course = new Hashtable();
    addHashTable(course);
    Console.WriteLine();
    Console.WriteLine("Key          Value ");
    foreach (DictionaryEntry sit in course)
    {
      Console.WriteLine("{0}\t {1}", sit.Key, sit.Value);
    }
   Console.ReadLine();
}
```



Adding elements with exiting 'key' to a hash table

18

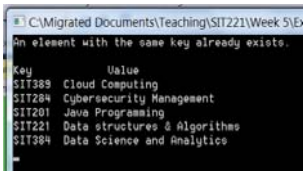
```
static void addHashTable (Hashtable course)
{
    try {
      course.Add("SIT221", "Data structures & Algorithms");
      course.Add("SIT284", "Cybersecurity Management");
      course.Add("SIT384", "Data Science and Analytics");
      course.Add("SIT201", "Java Programming");
      course.Add("SIT389", "Cloud Computing");
      course.Add("SIT221", "Network Security");
} catch  {
        Console.WriteLine("The same key already exists.");
    }
}
```



Adding elements with exiting 'key' to a hash table

19

```
static void Main(string[] args)
{
  // Create a new hash table.
    Hashtable course = new Hashtable();
    addHashTable(course);
    Console.WriteLine();
    Console.WriteLine("Key          Value ");
    foreach (DictionaryEntry sit in course)
    {
      Console.WriteLine("{0}\t {1}", sit.Key, sit.Value);
    }
   Console.ReadLine();
}
```

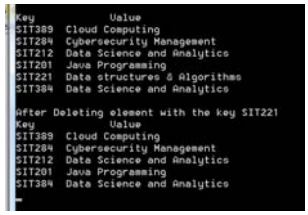


Deleting elements for the hash table

20

Removes the element with the key SIT221.

```
course.Remove("SIT221");
foreach (DictionaryEntry sit in course)
{
  Console.WriteLine("{0}\t{1}", sit.Key, sit.Value);
}
```



Hash table lookup

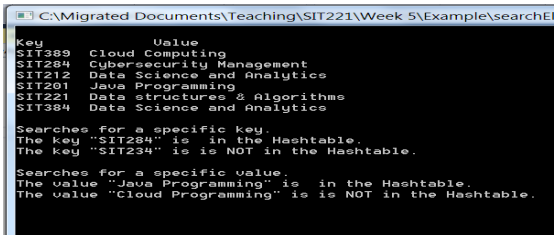
21

- For searching, we use the ContainsKey() / ContainsValue() method, which returns a Boolean indicating whether or not a specified key/value was found in the Hashtable.

```
static void searchingHashKey(Hashtable course, string myKey)
{
    string found = course.ContainsKey(myKey) ? " in the Hashtable" : "is NOT in the Hashtable";
    Console.WriteLine("The key \"{0}\" is {1}.", myKey, found);
}
static void searchingHashValue(Hashtable course, string myValue)
{
     string found = course.ContainsValue(myValue) ? " in the Hashtable" : "is NOT in the Hashtable";
     Console.WriteLine("The value\"{0}\" is {1}.", myValue, found);
}
```

Hash table lookup …

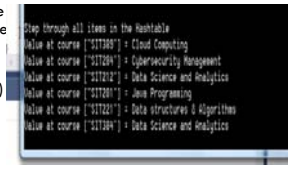
22



Enumerating hash table with Keys property

23

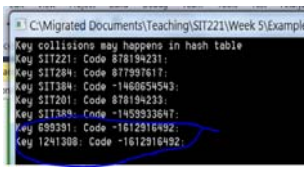
- The Hashtable class contains a Keys property that returns a collection of the keys used in the Hashtable. You can index the Hashtable by the key, just like you would index an array by an ordinal value.




```
static void iteratinHashValue(Hashtable course)
{
   foreach (string key in course.Keys)
      Console.WriteLine("Value at course [\""
         + key + "\"] = " +
            course[key].ToString());
}
```

Issues with Hashing: Issue #1

- Multiple keys can hash to the same slot – collisions are possible.
- For example, the strings "699391" and "1241308" produce the same hashcode.
- We can design hash functions such that collisions are minimized.
- Collision is an inherent property of hash function and thus avoiding collisions is virtually impossible.
- The only solution is to design collision-resolution techniques.



How are collisions handled?

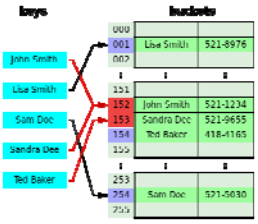
25

- The basic task is to find an alternative location in the hash table to place the object when collision occurs.
- There are a number of *collision resolution* strategies that can be employed.
  - Linear probe method
  - Rehashing technique
  - Chaining approach

Linear probe collision resolution method

26

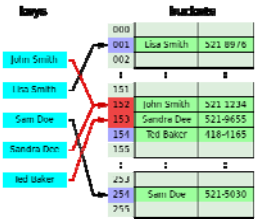
- Linear probe is one of the easiest methods and works as follows:
  - Suppose a new item is to be inserted into the hash table at spot 'i'.
  - If spot 'i' in the hash table is already occupied, check if the next spot at (i + 1) is empty. If it is empty insert it at that spot.
  - If spot at (i + 1) is also taken, check i + 2, and so on, until an available spot is located.



Linear probe collision resolution method

27

- Inserting an item (T, x)
  - Worst-case complexity – proportional to number of checks for empty slots.
- Deleting an item (T, x)
  - Worst-case complexity – proportional to number of checks for empty slots.
- Search for an item (T, k)
  - Worst-case complexity – proportional to number of checks for empty slots.



Rehashing collision resolution technique

28

- Rehashing (also known as double hashing) is used by C# Hashtable class.
- Rehashing uses a set of different hash functions, $H_1, H_2, \cdots, H_n$ that differ only by a multiplicative factor. In general, the hash function H_k $(1 \leq k \leq n)$ is defined as:

  H_k(key) = [GetHash(key) + **k** * (1 + (((GetHash(key) >> 5) + 1) % (size – 1)))] % size
  - Size is the hash size.
- It works as follows: try H_1 (key) →Collison; try H_2(key) → collision, …., try H_n(key) until you can insert the element into the hash table.
- For example, the strings "699391" and "1241308" produce the same hashcode.
  - H_1 ("699391") → OK
  - H_1 ("1241308") → collision; H_2 ("1241308") → collision; H_3 ("1241308") → OK.

Rehashing collision resolution technique

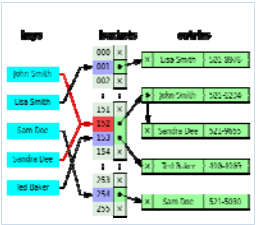
29

- Inserting an item (T, x)
  - Worst-case complexity – proportional to number of hash functions are executed.
- Deleting an item (T, x)
  - Worst-case complexity – proportional to number of hash functions are executed.
- Search for an item (T, k)
  - Worst-case complexity – proportional to number of hash functions are executed.

Chaining collision avoidance method

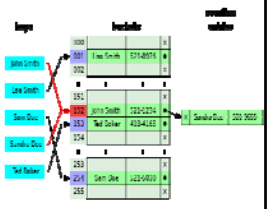
30

- Value will be maintained in a linked list
- The hash table slot will store a pointer to the head of the linked list.
- Store all elements that hash to the same slot in a linked list.
- Inserting an item (T, x) at the head of list T[h(key[x])].
  - Worst-case complexity – O(1).
- Deleting an item (T, x) from the list T[h(key[x])].
  - Worst-case complexity – proportional to length of list with singly-linked lists. O(1) with doubly-linked lists.
- Search for an item (T, k) with key k in list T[h(k)].
  - Worst-case complexity – proportional to length of list.



Chaining collision avoidance method

- The hash table value filed has a pointer to a list.
- The first element is stored in the hashtable and elements that hash to the same slot stored in a linked list.
- Inserting an item (T, x) at the slot or head of list T[h(key[x])].
  - Worst-case complexity – O(1).
- Deleting an item (T, x) from the slot or list T[h(key[x])].
  - Worst-case complexity – proportional to length of list with singly-linked lists.
- Search for an item (T, k) with key k in list T[h(k)].
  - Worst-case complexity – proportional to length of list.



Issues with Hashing: Issue #2

32

- Hash function collisions also present a problem when searching a hash table.
- In the example just discussed, the strings "699391" and "1241308" will result in a collision. Therefore, we expect to have the following cases
  - H_1(" 699391") → OK
  - H_1("1241308") → Collision; therefore, H_2("1241308") will have to be used to insert it in the hash table
- **Question**
  - When I call course["1241308"], how does the hashtable knows that it has to use H_2("1241308") ?

Dictionary data structure

33

- A dictionary is collection of KeyValuePair<KEY, VALUE>
  - Value store data
  - Each value has a key associated with it, which is used for insertion and retrieval purposes.
  - A functions that map keys to buckets of the hash table.
- Note that we can add keys and values of any type to the Hashtable.
- In contrast, the Dictionary class requires the data types for both the key and value to be explicitly specified:
  - Dictionary<keyType, valueType> variableName = new Dictionary<keyType, valueType>();
  - **Dictionary myDictionary = new Dictionary<int, Part>();**

Adding items to dictionary

34

```
myDictionary.Add(1, new Part { PartId = 1, PartName = “part 1”});
myDictionary.Add(2, new Part { PartId = 2, PartName = “part 2”});
myDictionary.Add(3, new Part { PartId = 3, PartName = “part 3”});
myDictionary.Add(20, new Part { PartId = 20, PartName = “part 20”});
myDictionary.Add(30, new Part { PartId = 30, PartName = “part 30”});
myDictionary.Add(200, new Part { PartId = 200, PartName = “part 200”});
myDictionary.Add(300, new Part { PartId = 300, PartName = “part 300”});
myDictionary.Add(2, new Part { PartId = 2, PartName = “part 2”});
```

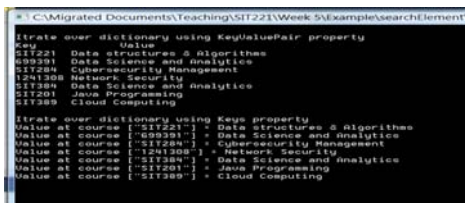
Dictionary data structure

35

```
static void Main(string[] args)
{
    Dictionary<string, string> course = new Dictionary<string, string>(); // Create a new hash table.
     addDictionary(course);  //populate the disctionary
     Console.WriteLine("Itrate over dictionary using KeyValuePair property");
     Console.WriteLine("Key          Value ");
     foreach (KeyValuePair<string, string> sit in course)
       Console.WriteLine("{0}\t{1}", sit.Key, sit.Value);
    Console.WriteLine("Itrate over dictionary using Keys property");
    foreach (string key in course.Keys)
        Console.WriteLine("Value at course [\"“  + key + "\"] = " +   course[key].ToString());
}
```

Dictionary data structure…

36



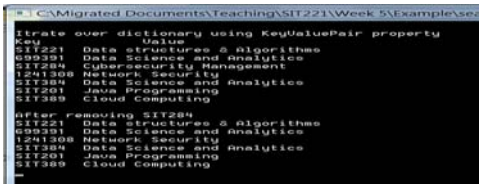
Deleting element from a dictionary

37

```
static void Main(string[] args)
{
 // Create a new hash table.
    Dictionary<string, string> course = new Dictionary<string, string>();
     addDictionary(course);
    Console.WriteLine("Itrate over dictionary using KeyValuePair property");
    Console.WriteLine("Key            Value ");
    foreach (KeyValuePair<string, string> sit in course)
      Console.WriteLine("{0}\t{1}", sit.Key, sit.Value);
    course.Remove("SIT284");
    Console.WriteLine("After removing {0}", "SIT284");
    foreach (KeyValuePair<string, string> sit in course)
      Console.WriteLine("{0}\t{1}", sit.Key, sit.Value);
}
```

Deleting element from a dictionary

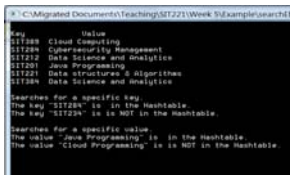
38



Let's put all together…Find/Remove/Iterate

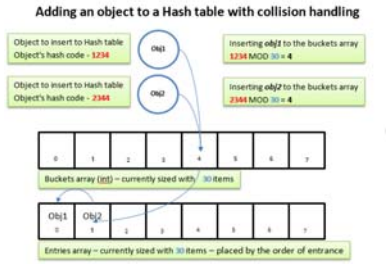
39

- Random access
  - Part myPart = myDictionary[20]; //indexer on key type
- Contains:
  - myDictionary.ContainsKey(myKey); // returns true;
  - myDictionary.ContainsValue(myValue); //returns true



.NET Dictionary handles collision using chaining

40



Lecture Summary

41

- When inserting an item into or retrieving an item from a hash table, a collision can occur.
  - When inserting an element into a hash table, an available slot must be found.
  - When retrieving an item from a hash table, the actual item must be found if it is not in the expected location.
- Collison avoidance methods
  - With rehashing, the hash is recomputed in the event of a collision, and the new slot corresponding to a hash is tried.
  - With chaining, a secondary data structure (linked list) is used to store any collisions. Specifically, each slot in the Dictionary has an array of elements that map to that bucket.
  - With linear prob, each slot from the point of collision is checked.

	Average	Worst case
Space	$O(n)$[1]	$O(n)$
Search	$O(1)$	$O(n)$
Insert	$O(1)$	$O(n)$
Delete	$O(1)$	$O(n)$