

# ECE 2400 Computer Systems Programming

## Fall 2021

### Topic 18: Tables

School of Electrical and Computer Engineering  
Cornell University

revision: 2021-08-29-22-39

1	Table Abstract Data Type	2
2	Table Concepts	4
3	Table Storage	4
4	Lookup Tables	5
5	Hash Tables	8

**zyBooks** The zyBooks logo is used to indicate additional material included in the course zyBook which will not be discussed in detail in lecture. Students are responsible for all material covered in lecture and in the course zyBook.

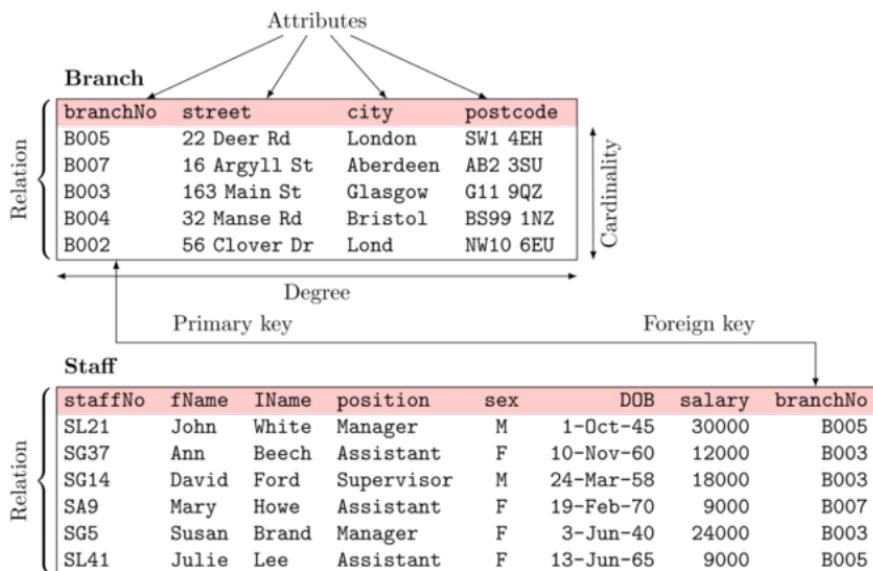
Copyright © 2021 Christopher Batten. All rights reserved. This handout was prepared by Prof. Christopher Batten at Cornell University for ECE 2400 / ENGRD 2140 Computer Systems Programming. Download and use of this handout is permitted for individual educational non-commercial purposes only. Redistribution either in part or in whole via both commercial or non-commercial means requires written permission.

# 1. Table Abstract Data Type

- insert new row in table
- insert new column in table
- modify cell in table
- sort rows/columns in table
- iterate across table

branchNo	street	city	postcode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QZ
B004	32 Manse Rd	Bristol	BS99 1NZ
B002	56 Clover Dr	Lond	NW10 6EU

## Relational Databases



ADT	Implementation					
	List	Vector	Binary Search Tree	Binary Heap Tree	Lookup Table	Hash Table
Indexed Seq	✓	★				
Iterable Seq	★	★				
Stack	★	★				
Queue	★	★				
Priority Queue	✓	✓			★	
Set	✓	✓	★		★	★
Map	✓	✓	★		★	★

While tables can be used on their own as an ADT, in this course we will focus on using tables to efficiently implement other ADTs

## 2. Table Concepts

Concept	Description
Table	A collection of data organized into rows and columns.
Row	A horizontal row of data in a table.
Column	A vertical column of data in a table.
Cell	The intersection of a row and a column, containing a single data value.
Primary Key	A unique identifier for each row in a table.
Foreign Key	A reference to a primary key in another table.
Index	A data structure that speeds up data retrieval.
Constraint	A rule that defines the validity of data in a table.
Normalization	The process of organizing data into tables to reduce redundancy and inconsistency.

## 3. Table Storage

Storage Type	Description
In-Memory Storage	Stores data in memory, providing fast access but requiring power to persist.
Disk-Based Storage	Stores data on disk, providing persistent storage but slower access times.
Relational Database Management Systems (RDBMS)	Systems designed to store, retrieve, and manage relational data.
NoSQL Databases	Systems designed to handle large volumes of unstructured or semi-structured data.
Big Data Storage	Systems designed to handle extremely large datasets and complex processing requirements.
Cloud Storage	Storage services provided over the internet, often using distributed architectures.
Object Storage	A type of storage where data is represented as objects, each with a unique identifier and associated metadata.
File Storage	A type of storage where data is organized into files and directories.
Block Storage	A type of storage where data is organized into blocks, which can be addressed individually.

## 4. Lookup Tables

- Recall that sets provide add and contains member functions
  - Recall that maps provide add and lookup member functions
  - Consider implementing a set/map with a list, vector, or tree
- 

	Time Complexity		Space Complexity
	add	contains lookup	
list			
vector (sorted)			
binary search tree			
lookup table			

- A **lookup table** is a table where the value is *directly* used to index into the table
- Focus on object-oriented array-based lookup tables for storing positive ints or Strings to implement sets
  - Could apply same approach to implementing a map
  - Could use object-oriented programming and dynamic polymorphism
  - Could use generic programming and static polymorphism
  - Could use functional programming to make hash function generic
  - Could use concurrent programming to analyze table in parallel

## 4. Lookup Tables

---

```
1 class LookupTableInt
2 {
3     public:
4         LookupTableInt();
5
6     void add( int v );
7     bool contains( int v );
8
9     private:
10    bool m_tbl[8];
11 };
12
13 LookupTableInt::
14     LookupTableInt()
15 {
16     for (int i=0; i<8; i++)
17         m_tbl[i] = false;
18 }
```

19 void LookupTableInt::add( int v ) {  
20  
21 bool LookupTableInt::  
22 contains( int v ) {  
23  
24 }  
25 }

Draw the table resulting from this code sequence:

```
1 LookupTableInt tbl;
2 tbl.add(3);
3 tbl.add(2);
4 tbl.add(3);
5 tbl.add(5);
6 tbl.add(6);
```

## 4. Lookup Tables

```
1 class LookupTableStr
2 {
3     public:
4         LookupTableStr();
5
6         void add( String v );
7         bool contains( String v );
8
9     private:
10        int idx( String v );
11        bool m_tbl[5];
12    };
13
14 LookupTableStr::
15     LookupTableStr()
16    {
17        for (int i=0; i<5; i++)
18            m_tbl[i] = false;
19    }
20
21     void LookupTableStr::add( String v ) {
22
23     bool LookupTableStr::
24     contains( String v ) {
25
26     int LookupTableStr::idx( String v )
27     {
28
29         if      ( v == "apple" ) return 0;
30         else if ( v == "banana" ) return 1;
31         else if ( v == "cherry" ) return 2;
32         else if ( v == "grape" ) return 3;
33         else if ( v == "kiwi"   ) return 4;
34         assert( false );
35     }
36 }
```

Draw the table resulting from this code sequence:

```
1 LookupTableStr tbl;  
2   tbl.add("cherry");  
3   tbl.add("banana");  
4   tbl.add("apple");  
5   tbl.add("cherry");
```

## 5. Hash Tables

- How can we maintain advantages of lookup table while mitigating the disadvantages?
- 

	Time Complexity		Space Complexity
	add	contains lookup	
list			
vector (sorted)			
binary search tree			
lookup table			
hash table			

---

- A **hash table** is a table where the value is used as input to a *hash function* which returns a positive integer which is then used to index into the table (with a mod (%) operation)
- Focus on object-oriented array-based hash table storing `ints` to implement a set
  - Could apply same approach to implementing a map
  - Could use object-oriented programming and dynamic polymorphism
  - Could use generic programming and static polymorphism
  - Could use functional programming to make hash function generic
  - Could use concurrent programming to analyze table in parallel

## Good Hash Functions

- What makes a hash function a “good” hash function?
- Property 1: We want a *valid* hash function
  - Returns the same value on subsequent calls to the same item
  - For any equivalent objects  $a == b$ , their hashes are also equal
- Property 2: We want a hash function that provides *uniformity*
  - Maps the expected inputs as evenly as possible over the output range
  - Specifically, the hash result should not be a value (e.g., 100) more often
- Property 3: We want a hash function with  $O(1)$  time complexity

## Example Hash Functions

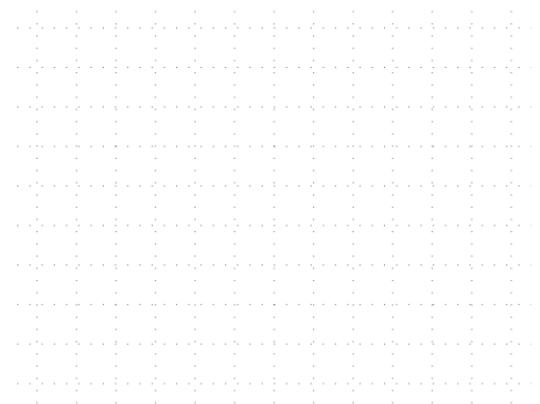
```
1 int hash( int v ) {  
2     return (v < 0) ? -v : v;  
3 }  
4  
5 int hash( String v ) {  
6     int h = 0;  
7     for ( int i = 0; i < v.size(); i++ )  
8         h = h + (int) v[i];  
9     return h;  
10 }  
11  
12 int hash( float v ) {  
13     return (int) ((v < 0) ? -v : v); // truncate to integer  
14 }  
15  
16 int hash( const Vector<int>& v ) {  
17     int sum = 0;  
18     for ( int e : v )  
19         sum += e;  
20     return (sum < 0) ? -sum : sum;  
21 }
```

```
1 class HashTableInt
2 {
3     public:
4         HashTableInt();
5
6     void add( int v );
7     bool contains( int v );
8
9     private:
10    int hash( int v );
11    int idx( int v );
12    bool m_tbl[4];
13 };
14
15 HashTableInt::
16     HashTableInt()
17 {
18     for (size_t i=0; i<4; i++)
19         m_tbl[i] = false;
20 }
```

```
21     void HashTableInt::add( int v ) {
22
23     bool HashTableInt::contains( int v ) {
24
25     int HashTableInt::hash( int v ) {
26         return (v < 0) ? -v : v;
27     }
28     int HashTableInt::idx( int v ) {
29         return hash(v) % 4;
30     }
31
32     if (contains(idx(hash(v)))) {
33         cout << "Collision at index " << idx(hash(v)) << endl;
34     }
35
36     m_tbl[idx(hash(v))] = true;
37 }
```

Draw the table resulting from this code sequence:

```
1 HashTableInt tbl;
2 tbl.add(3);
3 tbl.add(2);
4 tbl.add(3);
5 tbl.add(5);
6 tbl.add(6);
7 tbl.add(1);
```



- Two common approaches for handling collisions
  - Separate chaining (usually with linked lists)
  - Open addressing (usually with linear probing) → **zyBooks**

```
1 class HashTableInt
2 {
3     public:
4         HashTableInt();
5
6     void add( int v );
7     bool contains( int v );
8
9     private:
10    int hash( int v );
11    int idx( int v );
12    List<int> m_tbl[4];
13 };
14
15 HashTableInt::
16     HashTableInt()
17 { }
```

```
18 void HashTableInt::add( int v ) {
19     bool HashTableInt::contains( int v ) {
20     int HashTableInt::hash( int v ) {
21         return (v < 0) ? -v : v;
22     }
23
24     int HashTableInt::idx( int v ) {
25         return hash(v) % 4;
26     }
```

Draw the table resulting from this code sequence:

```
1 HashTableInt tbl;
2 tbl.add(3);
3 tbl.add(2);
4 tbl.add(3);
5 tbl.add(5);
6 tbl.add(6);
7 tbl.add(1);
```

What is the time complexity for add?

```
1 class HashTableInt
2 {
3     public:
4     HashTableInt();
5
6     void add( int v );
7     bool contains( int v );
8
9     private:
10    int hash( int v );
11    int idx( int v );
12    int m_size;
13    Vector<List<int>> m_tbl;
14 };
15
16 HashTableInt::HashTableInt()
17 {
18     m_size = 0;
19     for ( int i=0; i<4; i++ )
20         m_tbl.push_back(List<int>());
21 }
```

```
39 void HashTableInt::add( int v )
40 {
41     if ( !contains(v) ) {
42         m_tbl[idx(v)].push_back(v);
43         m_size++;
44     }
45
46     if ( (m_size/(1.0*m_tbl.size())) > 0.5 ) {
47
48         int new_size = 2*m_tbl.size();
49         Vector<List<int>> new_tbl;
50         for ( int i = 0; i < new_size; i++ )
51             new_tbl.push_back( List<int>() );
52
53         for ( int i = 0; i < m_tbl.size(); i++ ) {
54             for ( int x : m_tbl[i] )
55                 new_tbl[hash(x) % new_size].push_back(x);
56         }
57
58         m_tbl = new_tbl;
59     }
60 }
```

<https://repl.it/@cbatten/ece2400-T18-ex1>  
<https://repl.it/@cbatten/ece2400-T18-ex2>

## Hash Function for Strings

```
1 int HashTableStr::hash( String v ) {  
2     int h = 0;  
3     for ( int i = 0; i < v.size(); i++ )  
4         h = h + (int) v[i];  
5     return h;  
6 }  
7  
8 int HashTableStr::idx( String v ) {  
9     return hash(v) % m_tbl.size();  
10 }
```

40	(	50	<b>2</b>	60	<	70	<b>F</b>	80	<b>P</b>	90	<b>Z</b>	100	<b>d</b>	110	<b>n</b>
41	)	51	<b>3</b>	61	=	71	<b>G</b>	81	<b>Q</b>	91	[	101	<b>e</b>	111	<b>o</b>
42	*	52	<b>4</b>	62	>	72	<b>H</b>	82	<b>R</b>	92	\	102	<b>f</b>	112	<b>p</b>
43	+	53	<b>5</b>	63	?	73	<b>I</b>	83	<b>S</b>	93	]	103	<b>g</b>	113	<b>q</b>
44	,	54	<b>6</b>	64	@	74	<b>J</b>	84	<b>T</b>	94	^	104	<b>h</b>	114	<b>r</b>
45	-	55	<b>7</b>	65	<b>A</b>	75	<b>K</b>	85	<b>U</b>	95	-	105	<b>i</b>	115	<b>s</b>
46	.	56	<b>8</b>	66	<b>B</b>	76	<b>L</b>	86	<b>V</b>	96	_	106	<b>j</b>	116	<b>t</b>
47	/	57	<b>9</b>	67	<b>C</b>	77	<b>M</b>	87	<b>W</b>	97	<b>a</b>	107	<b>k</b>	117	<b>u</b>
48	<b>0</b>	58	:	68	<b>D</b>	78	<b>N</b>	88	<b>X</b>	98	<b>b</b>	108	<b>l</b>	118	<b>v</b>
49	<b>1</b>	59	;	69	<b>E</b>	79	<b>O</b>	89	<b>Y</b>	99	<b>c</b>	109	<b>m</b>	119	<b>w</b>

---

String	hash	idx
"bat"		
"tab"		
"elf"		
"ago"		

---

assume m\_tbl.size() is 1024

## Good Hash Function for Strings

```
1 int HashTableStr::hash( String v ) {  
2     int h = 0;  
3     for ( int i = 0; i < v.size(); i++ )  
4         h = (29 * h) + (int) v[i];  
5     return h;  
6 }  
7  
8 int HashTableStr::idx( String v ) {  
9     return hash(v) % m_tbl.size();  
10 }
```

40	(	50	<b>2</b>	60	<	70	<b>F</b>	80	<b>P</b>	90	<b>Z</b>	100	<b>d</b>	110	<b>n</b>
41	)	51	<b>3</b>	61	=	71	<b>G</b>	81	<b>Q</b>	91	[	101	<b>e</b>	111	<b>o</b>
42	*	52	<b>4</b>	62	>	72	<b>H</b>	82	<b>R</b>	92	\	102	<b>f</b>	112	<b>p</b>
43	+	53	<b>5</b>	63	?	73	<b>I</b>	83	<b>S</b>	93	]	103	<b>g</b>	113	<b>q</b>
44	,	54	<b>6</b>	64	@	74	<b>J</b>	84	<b>T</b>	94	^	104	<b>h</b>	114	<b>r</b>
45	-	55	<b>7</b>	65	<b>A</b>	75	<b>K</b>	85	<b>U</b>	95	-	105	<b>i</b>	115	<b>s</b>
46	.	56	<b>8</b>	66	<b>B</b>	76	<b>L</b>	86	<b>V</b>	96	_	106	<b>j</b>	116	<b>t</b>
47	/	57	<b>9</b>	67	<b>C</b>	77	<b>M</b>	87	<b>W</b>	97	<b>a</b>	107	<b>k</b>	117	<b>u</b>
48	<b>0</b>	58	:	68	<b>D</b>	78	<b>N</b>	88	<b>X</b>	98	<b>b</b>	108	<b>l</b>	118	<b>v</b>
49	<b>1</b>	59	;	69	<b>E</b>	79	<b>O</b>	89	<b>Y</b>	99	<b>c</b>	109	<b>m</b>	119	<b>w</b>

---

String	hash	idx
"bat"		
"tab"		
"elf"		
"ago"		

---

assume m\_tbl.size() is 1024