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.

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

Relational Databases

Attributes

Branch

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

Relation — Cardinality — Degree

Primary key — Foreign key

Staff

staffNo	fName	lName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

Relation

	Implementation					
ADT	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 effeciently implement other ADTs

2. Table Concepts

3. Table Storage

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		
	`add`	`contains` `lookup`	**Space Complexity**
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

```
class LookupTableInt
{
 public:
  LookupTableInt();

  void add( int v );
  bool contains( int v );

 private:
  bool m_tbl[8];
};

LookupTableInt::
  LookupTableInt()
{
  for (int i=0; i<8; i++)
    m_tbl[i] = false;
}
```

```
void LookupTableInt::add( int v ) {
```

```
bool LookupTableInt::
  contains( int v ) {
```

Draw the table resulting from this code sequence:

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

```
class LookupTableStr
{
 public:
  LookupTableStr();

  void add( String v );
  bool contains( String v );

 private:
  int idx( String v );
  bool m_tbl[5];
};

LookupTableStr::
  LookupTableStr()
{
  for (int i=0; i<5; i++)
    m_tbl[i] = false;
}
```

```
void LookupTableStr::add( String v ) {
```

```
bool LookupTableStr::
  contains( String v ) {
```

```
int LookupTableStr::idx( String v )
{
  if      ( v == "apple"  ) return 0;
  else if ( v == "banana" ) return 1;
  else if ( v == "cherry" ) return 2;
  else if ( v == "grape"  ) return 3;
  else if ( v == "kiwi"   ) return 4;
  assert( false );
}
```

Draw the table resulting from this code sequence:

```
LookupTableStr tbl;
tbl.add("cherry");
tbl.add("banana");
tbl.add("apple");
tbl.add("cherry");
```

5. Hash Tables

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

	Time Complexity		
	`add`	`contains` `lookup`	**Space Complexity**
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 `int`s 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

```
int hash( int v ) {
  return (v < 0) ? -v : v;
}

int hash( String v ) {
  int h = 0;
  for ( int i = 0; i < v.size(); i++ )
    h = h + (int) v[i];
  return h;
}

int hash( float v ) {
  return (int) ((v < 0) ? -v : v); // truncate to integer
}

int hash( const Vector<int>& v ) {
  int sum = 0;
  for ( int e : v )
    sum += e;
  return (sum < 0) ? -sum : sum;
}
```

```
class HashTableInt
{
 public:
  HashTableInt();

  void add( int v );
  bool contains( int v );

 private:
  int hash( int v );
  int idx( int v );
  bool m_tbl[4];
};

HashTableInt::
  HashTableInt()
{
  for (size_t i=0; i<4; i++)
    m_tbl[i] = false;
}
```

```
void HashTableInt::add( int v ) {
```

```
bool HashTableInt::contains( int v ) {
```

```
int HashTableInt::hash( int v ) {
  return (v < 0) ? -v : v;
}

int HashTableInt::idx( int v ) {
  return hash(v) % 4;
}
```

Draw the table resulting from this code sequence:

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

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

```
class HashTableInt
{
 public:
  HashTableInt();

  void add( int v );
  bool contains( int v );

 private:
  int hash( int v );
  int idx( int v );
  List<int> m_tbl[4];
};

HashTableInt::
  HashTableInt()
{ }
```

```
void HashTableInt::add( int v ) {
```

```
bool HashTableInt::contains( int v ) {
```

```
int HashTableInt::hash( int v ) {
  return (v < 0) ? -v : v;
}

int HashTableInt::idx( int v ) {
  return hash(v) % 4;
}
```

Draw the table resulting from this code sequence:

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

What is the time complexity for add?

```
class HashTableInt
{
 public:
  HashTableInt();

  void add( int v );
  bool contains( int v );

 private:
  int hash( int v );
  int idx( int v );
  int m_size;
  Vector<List<int>> m_tbl;
};

HashTableInt::HashTableInt()
{
  m_size = 0;
  for ( int i=0; i<4; i++ )
    m_tbl.push_back(List<int>());
}
```

```
bool HashTableInt::contains( int v )
{
  for ( int x : m_tbl[idx(v)] )
    if ( x == v )
      return true;
  return false;
}

int HashTableInt::hash( int v )
{
  return (v < 0) ? -v : v;
}

int HashTableInt::idx( int v )
{
  return hash(v) % m_tbl.size();
}
```

```
void HashTableInt::add( int v )
{
  if ( !contains(v) ) {
    m_tbl[idx(v)].push_back(v);
    m_size++;
  }

  if ( (m_size/(1.0*m_tbl.size())) > 0.5 ) {

    int new_size = 2*m_tbl.size();
    Vector<List<int>> new_tbl;
    for ( int i = 0; i < new_size; i++ )
      new_tbl.push_back( List<int>() );

    for ( int i = 0; i < m_tbl.size(); i++ ) {
      for ( int x : m_tbl[i] )
        new_tbl[hash(x) % new_size].push_back(x);
    }

    m_tbl = new_tbl;
  }
}
```

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

Hash Function for Strings

```
int HashTableStr::hash( String v ) {
  int h = 0;
  for ( int i = 0; i < v.size(); i++ )
    h = h + (int) v[i];
  return h;
}

int HashTableStr::idx( String v ) {
  return hash(v) % m_tbl.size();
}
```

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

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

assume `m_tbl.size()` is 1024

Good Hash Function for Strings

```
int HashTableStr::hash( String v ) {
  int h = 0;
  for ( int i = 0; i < v.size(); i++ )
    h = (29 * h) + (int) v[i];
  return h;
}

int HashTableStr::idx( String v ) {
  return hash(v) % m_tbl.size();
}
```

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

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

assume `m_tbl.size()` is 1024