ECE 2400 Computer Systems Programming Spring 2025

Topic 16: Tables

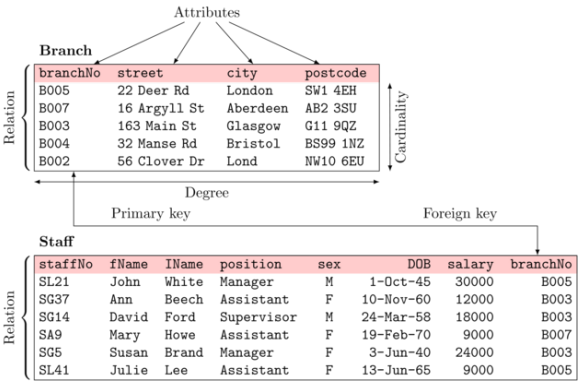
School of Electrical and Computer Engineering
Cornell University

revision: 2025-04-23-10-16

zyBooks logo indicates readings and coding labs 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 Basics

Common use case: Relational Databases



Tables can be ADTs with operations `insert row|column`, `modify cell`, `sort rows|columns`, etc. However, in this class, we use **tables as efficient implementations of other ADTs**:

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

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` (*no duplicates: must do contains first!)*	`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

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

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);
```

```
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` (*no duplicates: must do contains first!*)	`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 ( int i=0; i<4; i++ )
19     m_tbl[i] = false;
20 }
```

```
21 void HashTableInt::add( int v ) {
22 bool HashTableInt::contains( int v ) {
23 int HashTableInt::hash( int v ) {
24   return (v < 0) ? -v : v;
25 }
26
27 int HashTableInt::idx( int v ) {
28   return hash(v) % 4;
29 }
```

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

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

```
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>());
}
```

```
 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;
   }
 }
```

```
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();
}
```

See zyBook section 16.1 for runnable code.

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	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

```
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	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