1 Tables Concepts 2

2 Lookup Tables 3
   2.1. Set of Integers as a Lookup Table 3
   2.2. Set of Strings as a Lookup Table 5
   2.3. Strengths and Weaknesses of Lookup Tables 7

3 Hash Tables 8
1. Tables Concepts

- Tables have rows and columns
- Rows are usually identified by a name (number or even a string)
- Columns are usually identified by a name (number or even a string)
- Rows may have multiple columns of data
- Tables are indexed to access row data at a certain column

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table as a Set ADT

Table as a Map ADT
2. Lookup Tables

- A lookup table is a table in which all potential data is known
- A lookup table is accessed by simple array indexing in O(1) time
- Lookup tables may be multidimensional

2.1. Set of Integers as a Lookup Table

- A set of unsigned integers in the range [0, 4]
- Implemented internally as a simple array of booleans

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Present?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Interface

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

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

private:
    static const unsigned int size = 5;
    bool m_table[size];
};
```
2. Lookup Tables

2.1. Set of Integers as a Lookup Table

Implementation

```cpp
    LookupTableInt::LookupTableInt()
    { }

    void LookupTableInt::add( unsigned int v )
    {
        m_table[v] = true;
    }

    void LookupTableInt::remove( unsigned int v )
    {
        m_table[v] = false;
    }

    bool LookupTableInt::contains( unsigned int v )
    {
        return m_table[v];
    }
```

Main

```cpp
    int main( void )
    {
        LookupTableInt table;

        table.add(3); // Add value 3
        table.remove(3); // Remove value 3

        return 0;
    }
```
2.2. Set of Strings as a Lookup Table

- A set of strings limited to "Apple", "Banana", "Cherry"
- Implemented internally as a simple array of booleans
- Must correspond each possible string to an internal array index

<table>
<thead>
<tr>
<th>Index</th>
<th>Apple</th>
<th>Banana</th>
<th>Cherry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Present?</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Interface

```cpp
class LookupTableString {
    public:
        LookupTableString();
        void add ( std::string v );
        void remove ( std::string v );
        bool contains ( std::string v );

    private:
        unsigned int ind( std::string v );
        static const unsigned int size = 3;
        bool m_table[size];
};
```

Implementation

```cpp
LookupTableString::LookupTableString() {
}
```
void LookupTableString::add( std::string v )
{
    m_table[ ind(v) ] = true;
}

void LookupTableString::remove( std::string v )
{
    m_table[ ind(v) ] = false;
}

bool LookupTableString::contains( std::string v )
{
    return m_table[ ind(v) ];
}

unsigned int LookupTableString::ind( std::string v )
{
    if ( v == "Apple" ) return 0;
    else if ( v == "Banana" ) return 1;
    else if ( v == "Cherry" ) return 2;
    else throw std::out_of_range( "No such fruit!" );
}

Main

int main( void )
{
    LookupTableString table;
    table.add( "Cherry" ); // Add value "Cherry"
    table.remove( "Cherry" ); // Remove value "Cherry"
    return 0;
}
2.3. **Strengths and Weaknesses of Lookup Tables**

- We have implemented a set of integers as a lookup table
- We have implemented a set of strings as a lookup table
- Recall that lists, vectors, and even trees can also implement sets!

<table>
<thead>
<tr>
<th></th>
<th>add</th>
<th>remove</th>
<th>contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>list/vector</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>tree</td>
<td>$O(\log(N))$</td>
<td>$O(\log(N))$</td>
<td>$O(\log(N))$</td>
</tr>
<tr>
<td>lut</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

**Strengths of Lookup Tables**

- Lookup tables have great time complexity!

**Weaknesses of Lookup Tables**

- I want a set for a larger range of numbers (e.g., from 0 to 1,000,000)
  - Lookup tables are ________________
- I want a set of other types (e.g., strings)
  - Lookup tables are ________________
2.4. Mitigating the Weaknesses of Lookup Tables

- Lookup tables have space complexity that is linear with \( k \), the space of all possible unique inputs.

\[
\begin{align*}
\text{k = 4 and N = 2} & \quad \text{k = 1,000,000 and N = 2}
\end{align*}
\]

- How can we make the space complexity instead scale with \( N \), the number of unique elements currently in the set?

Reduction Operations

- Define a `reduce` function that converts a large number into the range accessible by the internal array.
- When the internal array fills up, double the size of the array.

Member function to reduce a large number

```c
unsigned int reduce( unsigned int v ) {
```

### Interface

```cpp
class LookupTableIntOneMillion
{
  public:
    LookupTableIntOneMillion();

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

  private:
    unsigned int reduce ( unsigned int v );
    static const unsigned int size = 10;
    bool m_table[size];
};
```

### Implementation

```cpp
LookupTableIntOneMillion::LookupTableIntOneMillion()
{ }

void LookupTableIntOneMillion::add( unsigned int v )
{
  m_table[ reduce(v) ] = true;
}

void LookupTableIntOneMillion::remove( unsigned int v )
{
  m_table[ reduce(v) ] = false;
}

bool LookupTableIntOneMillion::contains( unsigned int v )
{
  return m_table[ reduce(v) ];
}
```
Implementation of New Member Function

```c
18  unsigned int LookupTableIntOneMillion::reduce( unsigned int v
19  {                                                 
20    return v % size;
21  }
```

Main

```c
1  int main( void )
2  {
3    LookupTableIntOneMillion table;
4
5    table.add( 967 ); // Reduces to index 7
6    table.add( 333 ); // Reduces to index 3
7    table.add( 555 ); // Reduces to index 5
8
9    return 0;
10  }
```

Space Complexity with a Reduction Function

- Ignoring conflicts, we now have space complexity linear with $N$
- Time complexity remains at $O(1)$ access
- Achieving this space complexity cost us...
  - Increased work in order to double the size of the internal array
  - Increasing the work on each access
- Any other issues?
2.5. Mitigating the Weaknesses of Lookup Tables (cont.)

- Lookup tables are *inflexible* in storing non-integer types like strings.
- But strings, like any other type represented by a computer, are actually just numbers in the underlying representation!
- The key idea is to add a new function, a *hash* function, which converts a string or other arbitrary type into an integer.
  - Values of char type are just numbers!
  - The ASCII char ‘a’ is 97, ‘b’ is 98, etc...

**Member function to convert a char array into an unsigned integer**

```c
1  unsigned int hash( char v[] ) {
```
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
2. Lookup Tables

2.5. Mitigating the Weaknesses of Lookup Tables (cont.)

Interface

```cpp
class LookupTableStringHash
{
public:
    LookupTableStringHash();
    void add ( std::string v );
    void remove ( std::string v );
    bool contains ( std::string v );

private:
    unsigned int reduce ( unsigned int v );
    unsigned int hash ( std::string v );
    static const unsigned int size = 10;
    bool m_table[size];
};
```

Implementation

```cpp
LookupTableStringHash::LookupTableStringHash()
{
}

void LookupTableStringHash::add( std::string v )
{
    m_table[ reduce( hash(v) ) ] = true;
}

void LookupTableStringHash::remove( std::string v )
{
    m_table[ reduce( hash(v) ) ] = false;
}

bool LookupTableStringHash::contains( std::string v )
{
    return m_table[ reduce( hash(v) ) ];
}
```
Implementation of New Member Functions

```cpp
unsigned int LookupTableStringHash::reduce( unsigned int v )
{
    return v % size;
}

unsigned int LookupTableStringHash::hash( std::string v )
{
    unsigned int hashnum = 0;
    for ( auto& c : v )
        hashnum += c;
    return hashnum;
}
```

Main

```cpp
int main( void )
{
    LookupTableStringHash table;

    table.add( "c" ); // Hash 99 : Reduced to 9
    table.add( "a" ); // Hash 97 : Reduced to 7
    table.add( "t" ); // Hash 116 : Reduced to 6

    table.add( "cat" ); // Hash: 99 + 97 + 116 = 312
                        // Reduced to 2

    return 0;
}
```
Summary

• We started off trying to mitigate the weaknesses of lookup tables

<table>
<thead>
<tr>
<th></th>
<th>add</th>
<th>remove</th>
<th>contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>list/vector</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>tree</td>
<td>O(log(N))</td>
<td>O(log(N))</td>
<td>O(log(N))</td>
</tr>
<tr>
<td>lut</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>lut w/ hash &amp; reduce</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

Strengths of Lookup Tables with hash and reduce Functions

• Time complexity is still O(1)

Mitigated Weaknesses of Lookup Tables with hash and reduce

• I want a set for a larger range of numbers (e.g., from 0 to 1,000,000)
  – Lookup tables were space-inefficient
  – Lookup tables with hash and reduce have space efficiency of O(N)

• I want a set of other types (e.g., strings)
  – Lookup tables were inflexible
  – Lookup tables with hash and reduce can handle any type as long as it provides a valid hash function

• What do we do about collisions?
3. Hash Tables

- Hash tables are lookup tables with hash and reduce functions that also handle collisions.

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

3.1. Collision Handling with Separate Chaining

- The internal array elements are lists in a separate chaining scheme
- With a good hash function, the N elements are spread evenly over the K buckets.
- Adding a new element to the hash table is now a push back on a list
- What does this do to our time complexity?
class List {
public:

class Itr {
public:
Itr( Node* node_p );
void next();
std::string& get();
bool eq( const Itr& itr ) const;

private:
friend class List;
Node* m_node_p;
};

Itr begin();
Itr end();

List();
~List();
void push_front( std::string& v );

private:

struct Node {
std::string value;
Node* next_p;
};

Node* m_head_p;
};
`List::Itr` `operator++( List::Itr& itr, std::string );`

`List::Itr&` `operator++( List::Itr& itr );`

`std::string&` `operator* ( List::Itr& itr );`

`bool` `operator!=( const List::Itr& itr0, const List::Itr& itr1 );`

- With the above `List` class (provided exactly as discussed in previous lectures), implement the `add` function of a hash table that handles collisions with chaining using linked lists.

- Note that a templated argument `Hasher` is often used to provide a class with a hash function.
  
  - The following class “has-a” `Hasher` named `hash`.
  
  - The `Hasher` has an overloaded call operator (i.e., `operator()`) that does the hash (e.g., calling `hash` with a string returns an unsigned integer).

- Below is the interface of the hash table. Assume that calling the `Hasher` object using `hash()` will create some collisions.

```cpp
template < typename Hasher >
class HashTableChaining
{
  public:
    HashTableChaining();
    void add ( const std::string& v );
    void remove ( const std::string& v );
    bool contains ( const std::string& v );

  private:
    unsigned int reduce( unsigned int v );
    Hasher hash;
    static const unsigned int size = 10;
    List m_table[size];
};
```
3. Hash Tables

3.1. Collision Handling with Separate Chaining

```c++
void add( const std::string& v ) {

```
Summary

- How do hash tables compare to previous implementations of sets?

<table>
<thead>
<tr>
<th></th>
<th>add</th>
<th>remove</th>
<th>contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>list/vector</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>tree</td>
<td>$O(\log(N))$</td>
<td>$O(\log(N))$</td>
<td>$O(\log(N))$</td>
</tr>
<tr>
<td>lut</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>lut w/ hash &amp; reduce</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>hash table</td>
<td>________</td>
<td>________</td>
<td>________</td>
</tr>
</tbody>
</table>