Hashing & Hash Tables
Overview

- **Hash Table Data Structure: Purpose**
  - To support insertion, deletion and search in average-case constant time
    - Assumption: Order of elements irrelevant
    - \( \Rightarrow \) data structure *not* useful for if you want to maintain and retrieve some kind of an order of the elements

- **Hash function**
  - \( \text{Hash}[ \text{"string key"}] \Rightarrow \text{integer value} \)

- **Hash table ADT**
  - Implementations, Analysis, Applications
Hash table: Main components

Hash table
(implemented as a vector)

TableSize

Hash index

key

value

H(key)

How to determine ... ?

Cpt S 223. School of EECS, WSU
Hash Table

- Hash table is an array of fixed size TableSize
- Array elements indexed by a key, which is mapped to an array index (0...TableSize-1)
- Mapping (hash function) h from key to index
  - E.g., h(“john”) = 3
### Hash Table Operations

- **Insert**
  - \( T[h("john")] = \langle "john", 25000 \rangle \)

- **Delete**
  - \( T[h("john")] = \text{NULL} \)

- **Search**
  - \( T[h("john")] \) returns the element hashed for "john"

What happens if \( h("john") == h("joe") \)?

"collision"
Factors affecting Hash Table Design

- Hash function
- Table size
  - Usually fixed at the start
- Collision handling scheme
Hash Function

- A hash function is one which maps an element’s key into a valid hash table index
  - \( h(\text{key}) \Rightarrow \text{hash table index} \)

Note that this is (slightly) different from saying:
- \( h(\text{string}) \Rightarrow \text{int} \)

- Because the key can be of any type
  - E.g., “\( h(\text{int}) \Rightarrow \text{int} \)” is also a hash function!
- But also note that any type can be converted into an equivalent string form
Hash Function Properties

- A hash function maps key to integer
  - Constraint: Integer should be between $[0, \text{TableSize}-1]$  
- A hash function can result in a many-to-one mapping (causing collision)
  - Collision occurs when hash function maps two or more keys to same array index
- Collisions cannot be avoided but its chances can be reduced using a “good” hash function
Hash Function Properties

- A “good” hash function should have the properties:
  1. Reduced chance of collision
     - Different keys should ideally map to different indices
     - Distribute keys uniformly over table
  2. Should be fast to compute
Hash Function - Effective use of table size

- Simple hash function (assume integer keys)
  - $h(\text{Key}) = \text{Key} \mod \text{TableSize}$

- For random keys, $h()$ distributes keys evenly over table
  - What if TableSize = 100 and keys are ALL multiples of 10?
  - Better if TableSize is a prime number
Different Ways to Design a Hash Function for String Keys

A very simple function to map strings to integers:
- Add up character ASCII values (0-255) to produce integer keys
  - E.g., “abcd” = 97+98+99+100 = 394
  - ==> h(“abcd”) = 394 % TableSize

Potential problems:
- Anagrams will map to the same index
  - h(“abcd”) == h(“dbac”)
- Small strings may not use all of table
  - Strlen(S) * 255 < TableSize
- Time proportional to length of the string
Different Ways to Design a Hash Function for String Keys

- **Approach 2**
  - Treat first 3 characters of string as base-27 integer (26 letters plus space)
    - Key = $S[0] + (27 \times S[1]) + (27^2 \times S[2])$
  - Better than approach 1 because ... ?

**Potential problems:**
- Assumes first 3 characters randomly distributed
  - Not true of English:
    - Apple
    - Apply
    - Appointment
    - Apricot
    - Collision
Different Ways to Design a Hash Function for String Keys

- **Approach 3**
  - Use all $N$ characters of string as an $N$-digit base-$K$ number
  - Choose $K$ to be prime number larger than number of different digits (characters)
    - I.e., $K = 29, 31, 37$
  - If $L = \text{length of string } S$, then
    $$h(S) = \left[ \sum_{i=0}^{L-1} S[L-i-1] \times 37^i \right] \mod \text{TableSize}$$
  - Use Horner’s rule to compute $h(S)$
  - Limit $L$ for long strings

```
1 /**
2 * A hash routine for string objects.
3 */
4 int hash( const string &key, int tableSize )
5 {
6   int hashVal = 0;
7   for( int i = 0; i < key.length(); i++ )
8     hashVal = 37 * hashVal + key[ i ];
9   hashVal %= tableSize;
10  if( hashVal < 0 )
11    hashVal += tableSize;
12  return hashVal;
13 }
```

Problems:
- potential overflow
- larger runtime

Cpt S 223. School of EECS, WSU
Techniques to Deal with Collisions

- Chaining
- Open addressing
- Double hashing
- Etc.
**Resolving Collisions**

- What happens when \( h(k_1) = h(k_2) \)?
  - \( \Rightarrow \) collision!

- Collision resolution strategies
  - *Chaining*
    - Store colliding keys in a linked list at the same hash table index
  - *Open addressing*
    - Store colliding keys elsewhere in the table
Chaining

Collision resolution technique #1
Chaining strategy: maintains a linked list at every hash index for collided elements

- Hash table T is a vector of linked lists
  - Insert element at the head (as shown here) or at the tail
- Key k is stored in list at T[h(k)]
- E.g., TableSize = 10
  - h(k) = k mod 10
  - Insert first 10 perfect squares

Insertion sequence: { 0 1 4 9 16 25 36 49 64 81 }
Implementation of Chaining Hash Table

```
template <typename HashedObj>
class HashTable
{
    public:
        explicit HashTable( int size = 101 );
        bool contains( const HashedObj & x ) const;
        void makeEmpty( );
        void insert( const HashedObj & x );
        void remove( const HashedObj & x );

    private:
        vector<list<HashedObj> > theLists; // The array of Lists
        int currentSize;

        void rehash( );
        int myhash( const HashedObj & x ) const;
    };

    int hash( const string & key );
    int hash( int key );
};
```
Implementation of Chaining Hash Table

```cpp
int myhash( const HashedObj & x ) const
{
    int hashVal = hash(x);
    hashVal %= theLists.size();
    if( hashVal < 0 )
        hashVal += theLists.size();
    return hashVal;
}
```

This is the hashtable’s current capacity (aka. “table size”)

This is the hash table index for the element x
bool insert( const HashedObj & x )
{
    list<HashedObj> & whichList = theLists[ myhash( x ) ];
    if( find( whichList.begin( ), whichList.end( ), x ) != whichList.end( ) )
        return false;
    whichList.push_back( x );

    // Rehash; see Section 5.5
    if( ++currentSize > theLists.size( ) )
        rehash( );

    return true;
}
void makeEmpty() {
    for (int i = 0; i < theLists.size(); i++)
        theLists[i].clear();
}

bool contains( const HashedObj & x ) const
{
    const list<HashedObj> & whichList = theLists[ myhash( x ) ];
    return find( whichList.begin(), whichList.end(), x ) != whichList.end();
}

bool remove( const HashedObj & x )
{
    list<HashedObj> & whichList = theLists[ myhash( x ) ];
    list<HashedObj>::iterator itr = find( whichList.begin(), whichList.end(), x );
    if (itr == whichList.end())
        return false;
    whichList.erase( itr );
    --currentSize;
    return true;
}
All hash objects must define \( == \) and \( != \) operators.

Hash function to handle Employee object type
Collision Resolution by Chaining: Analysis

- **Load factor** $\lambda$ of a hash table $T$ is defined as follows:
  - $N =$ number of elements in $T$ ("current size")
  - $M =$ size of $T$ ("table size")
  - $\lambda = \frac{N}{M}$ ("load factor")
  - i.e., $\lambda$ is the average length of a chain

- **Unsuccessful search time:** $O(\lambda)$
  - Same for insert time

- **Successful search time:** $O(\lambda/2)$
  - Ideally, want $\lambda \leq 1$ (not a function of $N$)
Potential disadvantages of Chaining

Linked lists could get long
- Especially when N approaches M
- Longer linked lists could negatively impact performance

More memory because of pointers

Absolute worst-case (even if N << M):
- All N elements in one linked list!
- Typically the result of a bad hash function
Open Addressing

Collision resolution technique #2
Collision Resolution by Open Addressing

When a collision occurs, look elsewhere in the table for an empty slot

- Advantages over chaining
  - No need for list structures
  - No need to allocate/deallocate memory during insertion/deletion (slow)

- Disadvantages
  - Slower insertion – May need several attempts to find an empty slot
  - Table needs to be bigger (than chaining-based table) to achieve average-case constant-time performance
    - Load factor $\lambda \approx 0.5$
Collision Resolution by Open Addressing

- A “Probe sequence” is a sequence of slots in hash table while searching for an element x
  - \( h_0(x), h_1(x), h_2(x), ... \)
  - Needs to visit each slot exactly once
  - Needs to be repeatable (so we can find/delete what we’ve inserted)

- Hash function
  - \( h_i(x) = (h(x) + f(i)) \mod \text{TableSize} \)
  - \( f(0) = 0 \) => position for the 0th probe
  - \( f(i) \) is “the distance to be traveled relative to the 0th probe position, during the \( i \)th probe”.
Linear Probing

- \( f(i) = \) is a linear function of \( i \),

E.g., \( f(i) = i \)

\( h_i(x) = (h(x) + i) \mod \text{TableSize} \)

**Probe sequence:** +0, +1, +2, +3, +4, …

Continue until an empty slot is found

\#failed probes is a measure of performance
Linear Probing

- \( f(i) = \) is a linear function of \( i \), e.g., \( f(i) = i \)
  - \( h_i(x) = (h(x) + i) \mod \text{TableSize} \)

- **Probe sequence**: +0, +1, +2, +3, +4, ...

- **Example**: \( h(x) = x \mod \text{TableSize} \)
  - \( h_0(89) = (h(89)+f(0)) \mod 10 = 9 \)
  - \( h_0(18) = (h(18)+f(0)) \mod 10 = 8 \)
  - \( h_0(49) = (h(49)+f(0)) \mod 10 = 9 \ (X) \)
  - \( h_1(49) = (h(49)+f(1)) \mod 10 \)
    \[ = (h(49)+1) \mod 10 = 0 \]
## Linear Probing Example

### Insert sequence: 89, 18, 49, 58, 69

<table>
<thead>
<tr>
<th>Empty Table</th>
<th>After 89</th>
<th>After 18</th>
<th>After 49</th>
<th>After 58</th>
<th>After 69</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>49</td>
<td>49</td>
<td>49</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>58</td>
<td>58</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>69</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
</tr>
</tbody>
</table>

### #unsuccessful probes:

| time | 0 | 0 | 1 | 3 | 3 | 7 | 30 |

Cpt S 223. School of EECS, WSU
Linear Probing: Issues

Probe sequences can get longer with time

*Primary clustering*

- Keys tend to cluster in one part of table
- Keys that hash into cluster will be added to the end of the cluster (making it even bigger)
- Side effect: Other keys could also get affected if mapping to a crowded neighborhood
Linear Probing: Analysis

- Expected number of probes for insertion or unsuccessful search
  \[
  \frac{1}{2} \left(1 + \frac{1}{(1 - \lambda)^2}\right)
  \]

- Expected number of probes for successful search
  \[
  \frac{1}{2} \left(1 + \frac{1}{1 - \lambda}\right)
  \]

- Example (\(\lambda = 0.5\))
  - Insert / unsuccessful search
    - 2.5 probes
  - Successful search
    - 1.5 probes

- Example (\(\lambda = 0.9\))
  - Insert / unsuccessful search
    - 50.5 probes
  - Successful search
    - 5.5 probes
Random Probing: Analysis

- Random probing does not suffer from clustering

- Expected number of probes for insertion or unsuccessful search:
  \[ \frac{1}{\lambda} \ln \frac{1}{1 - \lambda} \]

- Example
  - \( \lambda = 0.5 \): 1.4 probes
  - \( \lambda = 0.9 \): 2.6 probes
Linear vs. Random Probing

- Linear probing
- Random probing

Load factor $\lambda$

$U$ - unsuccessful search
$S$ - successful search
$I$ - insert

Cpt S 223. School of EECS, WSU
Quadratic Probing

- Avoids primary clustering
- \( f(i) \) is quadratic in \( i \)
  
  e.g., \( f(i) = i^2 \)

  \[ h_i(x) = (h(x) + i^2) \mod \text{TableSize} \]

- **Probe sequence:**
  
  +0, +1, +4, +9, +16, ...

Continue until an empty slot is found

\#failed probes is a measure of performance
Quadratic Probing

- Avoids primary clustering
- \( f(i) \) is quadratic in \( I \), e.g., \( f(i) = i^2 \)
  - \( h_i(x) = (h(x) + i^2) \mod TableSize \)
- Probe sequence: +0, +1, +4, +9, +16, ...

Example:
- \( h_0(58) = (h(58)+f(0)) \mod 10 = 8 \ (X) \)
- \( h_1(58) = (h(58)+f(1)) \mod 10 = 9 \ (X) \)
- \( h_2(58) = (h(58)+f(2)) \mod 10 = 2 \)
Q) Delete(49), Find(69) - is there a problem?

Quadratic Probing Example

Insert sequence: 89, 18, 49, 58, 69

<table>
<thead>
<tr>
<th>Empty Table</th>
<th>After 89</th>
<th>After 18</th>
<th>After 49</th>
<th>After 58</th>
<th>After 69</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>49</td>
<td>+1²</td>
<td>49</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>58</td>
<td>+2²</td>
<td>58</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>89</td>
<td>+0²</td>
<td>18</td>
<td>+0²</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>89</td>
<td></td>
<td>89</td>
<td></td>
<td>89</td>
</tr>
</tbody>
</table>

#unsuccessful probes:

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

Cpt S 223. School of EECS, WSU
Quadratic Probing: Analysis

- Difficult to analyze
- Theorem 5.1
  - New element can always be inserted into a table that is at least half empty and TableSize is prime
- Otherwise, may never find an empty slot, even if one exists
- Ensure table never gets half full
  - If close, then expand it
Quadratic Probing

- May cause “secondary clustering”

Deletion

- Emptying slots can break probe sequence and could cause find stop prematurely

- Lazy deletion
  - Differentiate between empty and deleted slot
  - When finding skip and continue beyond deleted slots
    - If you hit a non-deleted empty slot, then stop find procedure returning “not found”

- May need compaction at some time
Quadratic Probing: Implementation

```cpp
template <typename HashedObj>
class HashTable
{
public:
    explicit HashTable( int size = 101 );

    bool contains( const HashedObj & x ) const;

    void makeEmpty( );

    bool insert( const HashedObj & x );

    bool remove( const HashedObj & x );
};
```
Quadratic Probing: Implementation

```cpp
enum EntryType { ACTIVE, EMPTY,Deleted }; 

private:
struct HashEntry
{
    HashedObj element;
    EntryType info;

    HashEntry(const HashedObj & e = HashedObj(), EntryType i = EMPTY) :
            element(e), info(i) {} 
};

vector<HashEntry> array;
int currentSize;

bool isActive(int currentPos) const;
int findPos(const HashedObj & x) const;
void rehash();
int myhash(const HashedObj & x) const;
};
```

Lazy deletion
Quadratic Probing: Implementation

1 explicit HashTable( int size = 101 ) : array( nextPrime( size ) )
2     { makeEmpty( ); }
3
4 void makeEmpty( )
5 {
6     currSize = 0;
7     for( int i = 0; i < array.size( ); i++ )
8         array[ i ].info = EMPTY;
9 }

Ensure table size is prime
Quadratic Probing: Implementation

```cpp
bool contains( const HashedObj & x ) const
    { return isActive( findPos( x ) ); };

int findPos( const HashedObj & x ) const
    {
        int offset = 1;
        int currentPos = myhash( x );
        
        while( array[ currentPos ].info != EMPTY &&
            array[ currentPos ].element != x )
        {
            currentPos += offset; // Compute ith probe
            offset += 2;
            if( currentPos >= array.size( ) )
                currentPos -= array.size( );
        }
        
        return currentPos;
    }

bool isActive( int currentPos ) const
    { return array[ currentPos ].info == ACTIVE; }
```

Find

Skip DELETED; No duplicates

Quadratic probe sequence (really)
Quadratic Probing: Implementation

```cpp
bool insert( const HashedObj & x )
{
    // Insert x as active
    int currentPos = findPos( x );
    if( isActive( currentPos ) )
        return false;

    array[ currentPos ] = HashEntry( x, ACTIVE );

    // Rehash; see Section 5.5
    if( ++currentSize > array.size() / 2 )
        rehash();

    return true;
}

bool remove( const HashedObj & x )
{
    int currentPos = findPos( x );
    if( !isActive( currentPos ) )
        return false;

    array[ currentPos ].info = DELETED;
    return true;
}
```

Insert:
No duplicates
Remove:
No deallocation needed
Double Hashing: keep two hash functions $h_1$ and $h_2$

- Use a second hash function for all tries $i$ other than 0:  
  \[ f(i) = i \times h_2(x) \]

- Good choices for $h_2(x)$?
  - Should never evaluate to 0
  - $h_2(x) = R - (x \mod R)$
    - $R$ is prime number less than TableSize

- Previous example with $R=7$
  - $h_0(49) = (h(49)+f(0)) \mod 10 = 9$ (X)
  - $h_1(49) = (h(49)+1*(7 - 49 \mod 7)) \mod 10 = 6$
## Double Hashing Example

<table>
<thead>
<tr>
<th>Empty Table</th>
<th>After 89</th>
<th>After 18</th>
<th>After 49</th>
<th>After 58</th>
<th>After 69</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>69</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>58</td>
<td>58</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>49</td>
<td>49</td>
<td>49</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>49</td>
<td>49</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Double Hashing: Analysis

- Imperative that TableSize is prime
  - E.g., insert 23 into previous table
- Empirical tests show double hashing close to random hashing
- Extra hash function takes extra time to compute
Probing Techniques - review

Linear probing:

Quadratic probing:

Double hashing*:

*(determined by a second hash function)
Rehashing

- Increases the size of the hash table when load factor becomes “too high” (defined by a cutoff)
  - Anticipating that prob(collisions) would become higher
- Typically expand the table to twice its size (but still prime)
- Need to reinsert all existing elements into new hash table
Rehashing Example

\[ h(x) = x \mod 7 \]
\[ \lambda = 0.57 \]

\[ h(x) = x \mod 17 \]
\[ \lambda = 0.29 \]

Insert 23

Rehashing

\[ \lambda = 0.71 \]
Rehashing Analysis

- Rehashing takes time to do N insertions
- Therefore should do it infrequently
- Specifically
  - Must have been N/2 insertions since last rehash
  - Amortizing the O(N) cost over the N/2 prior insertions yields only constant additional time per insertion
Rehashing Implementation

- When to rehash
  - When load factor reaches some threshold (e.g., $\lambda \geq 0.5$), OR
  - When an insertion fails

- Applies across collision handling schemes
Rehashing for Chaining

```cpp
/**
 * Rehashing for separate chaining hash table.
 */
void rehash()
{
    vector<list<HashedObj>> oldLists = theLists;

    // Create new double-sized, empty table
    theLists.resize( nextPrime( 2 * theLists.size( ) ) );
    for( int j = 0; j < theLists.size( ); j++ )
        theLists[ j ].clear( );

    // Copy table over
    currentSize = 0;
    for( int i = 0; i < oldLists.size( ); i++ )
    {
        list<HashedObj>::iterator itr = oldLists[ i ].begin( );
        while( itr != oldLists[ i ].end( ) )
            insert( *itr++ );
    }
```
Rehashing for Quadratic Probing

```cpp
/**
 * Rehashing for quadratic probing hash table.
 */

void rehash()
{
    vector<HashEntry> oldArray = array;

    // Create new double-sized, empty table
    array.resize( nextPrime( 2 * oldArray.size() ) );
    for( int j = 0; j < array.size(); j++ )
        array[ j ].info = EMPTY;

    // Copy table over
    currentSize = 0;
    for( int i = 0; i < oldArray.size(); i++ )
        if( oldArray[ i ].info == ACTIVE )
            insert( oldArray[ i ].element );
}
```
Hash Tables in C++ STL

- Hash tables not part of the C++ Standard Library
- Some implementations of STL have hash tables (e.g., SGI’s STL)
  - `hash_set`
  - `hash_map`
# Hash Set in STL

```cpp
#include <hash_set>

struct eqstr
{
    bool operator()(const char* s1, const char* s2) const
    {
        return strcmp(s1, s2) == 0;
    }
};

void lookup(const hash_set<const char*, hash<const char*>, eqstr>& Set,
            const char* word)
{
    hash_set<const char*, hash<const char*>, eqstr>::const_iterator it
        = Set.find(word);
    cout << word << ": " << (it != Set.end() ? "present" : "not present") << endl;
}

int main()
{
    hash_set<const char*, hash<const char*>, eqstr> Set;
    Set.insert("kiwi");
    lookup(Set, "kiwi");
}
```

Key
Hash fn
Key equality test
# Hash Map in STL

```cpp
#include <hash_map>

struct eqstr
{
    bool operator() (const char* s1, const char* s2) const
    {
        return strcmp(s1, s2) == 0;
    }
};

int main()
{
    hash_map<const char*, int, hash<const char*>, eqstr> months;
    months["january"] = 31;
    months["february"] = 28;
    ...  
    months["december"] = 31;
    cout << "january -> " << months["january"] << endl;
}
```
Problem with Large Tables

- What if hash table is too large to store in main memory?

- Solution: Store hash table on disk
  - Minimize disk accesses

- But...
  - Collisions require disk accesses
  - Rehashing requires a lot of disk accesses

Solution: Extendible Hashing
Hash Table Applications

- **Symbol table** in compilers
- Accessing tree or graph nodes by name
  - E.g., city names in Google maps
- Maintaining a **transposition table** in games
  - Remember previous game situations and the move taken (avoid re-computation)
- Dictionary lookups
  - Spelling checkers
  - Natural language understanding (word sense)
- Heavily used in text processing languages
  - E.g., Perl, Python, etc.
Summary

- Hash tables support fast insert and search
  - $O(1)$ average case performance
  - Deletion possible, but degrades performance
- Not suited if ordering of elements is important
- Many applications
Points to remember - Hash tables

- Table size prime
- Table size much larger than number of inputs (to maintain $\lambda$ closer to 0 or < 0.5)
- Tradeoffs between chaining vs. probing
- Collision chances decrease in this order: linear probing $=>$ quadratic probing $=>$ \{random probing, double hashing\}
- Rehashing required to resize hash table at a time when $\lambda$ exceeds 0.5
- Good for searching. Not good if there is some order implied by data.