Hashing

R. J. Renka

Department of Computer Science & Engineering
University of North Texas

04/06/2010
Problem: Given data values, such as employee records, associated with keys, such as names, store the data in such a way that key searches are fast. Assume that no ordering information is required. It is not necessary, for example, to find the smallest value. A $\log(N)$ search for the key, as in a binary search tree, is too slow.

Ideally the search key is an index into an array of data values, and search or insert takes constant average time.

*Hashing* is a technique for performing insert, delete, and find in constant average time.

A *hash table* is a fixed-size array containing $n = \text{Tablesize}$ items (key/data pairs).
A hash function maps the set of possible keys to the indices 0 to \(n - 1\). The ideal hash function is cheap to compute and distributes the keys uniformly over the indices so that every cell in the table is equally likely to be selected. Note that the number of possible keys may be much larger than \(n\) (e.g., \(26^m\) for \(m\)-character strings of lowercase letters), and we cannot therefore ensure different cells for different keys. A collision occurs when two keys hash to the same value.

If keys are random integers, Key \(\mod n\), with \(n\) prime, is generally a good hash function. If keys are \(m\)-character strings, some possible hash functions are as follows.

1. \((\sum_{i=0}^{m-1} \text{Key}[i]) \mod n\) (Figure 5.2).
2. \((\text{Key}[0] + 27\times\text{Key}[1] + 27^2\times\text{Key}[2]) \mod n\), assuming \(m \geq 3\) (Figure 5.3)
3. \((\sum_{i=0}^{m-1} \text{Key}[m - 1 - i] \times 37^i) \mod n\) (Figure 5.4)
The first option, the sum of ASCII values modulo $n$, is fast but not a good distribution unless $n$ is small. If all keys have at most 8 characters, the largest hash value would be $8 \times 127 = 1016$.

The idea behind the second option is that 27 is the number of letters, plus one for a blank space. It would be good if the first three characters were actually random, but in practice they are not. While $26^3 = 17,576$, the number of 3-character words in a typical English dictionary is only 2851. If $n = 10,000$ only 28.5% of the table is usable.

The third option is simple, reasonably fast unless keys are long (in which case we could use some random subset of the characters), and generally distributes well.
Figures 5.2-5.3

```c
1  int hash( const string & key, int tableSize )
2  {
3       int hashVal = 0;
4
5       for( int i = 0; i < key.length( ); i++ )
6           hashVal += key[ i ];
7
8       return hashVal % tableSize;
9  }
```

**Figure 5.2** A simple hash function

```c
1  int hash( const string & key, int tableSize )
2  {
3       return ( key[ 0 ] + 27 * key[ 1 ] + 729 * key[ 2 ] ) % tableSize;
4  }
```

**Figure 5.3** Another possible hash function — not too good
A good hash function

The polynomial is computed by Horner’s method. It is adjusted to a positive value in case of overflow.

```c
/**
 * A hash routine for string objects.
 */
int hash( const string & key, int tableSize )
{
    int hashVal = 0;
    for( int i = 0; i < key.length(); i++ )
        hashVal = 37 * hashVal + key[ i ];
    hashVal %= tableSize;
    if( hashVal < 0 )
        hashVal += tableSize;
    return hashVal;
}
```

**Figure 5.4** A good hash function
Collison resolution is required when we insert an element and it hashes to the same cell as a previously inserted element.

Separate chaining involves an array of linked lists. We keep a list of all elements that hash to the same value. We could use a binary search tree or another hash table for each list, but if the table is large and the hash function is good, then all lists should be short, and the simplest method is best. A good choice is the STL list.

The idea is illustrated in Figure 5.5 for the simple case of \( n = 10 \), \( \text{hash}(x) = x \mod 10 \), and keys \( 0, 1, 2^2, 3^2, \ldots, 9^2 \).

Figure 5.6 displays a class template for a separate chaining hash table. \texttt{HashedObj} must provide a hash function and equality operators \texttt{operator==} and/or \texttt{operator!=}. Note the required space between ’s in \texttt{vector\<\texttt{list\<\texttt{HashedObj}\>\> theLists;}
**Figure 5.5** A separate chaining hash table

Insertions are at the front of the list since recently inserted items are often the most likely to be accessed in the near future.
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);

Figure 5.6 Type declaration for separate chaining hash table
Figure 5.7  

```c++
1 int myhash( const HashedObj & x ) const 
2 {
3   int hashVal = hash( x );
4
5   hashVal %= theLists.size( );
6   if( hashVal < 0 )
7       hashVal += theLists.size( );
8
9   return hashVal;
10 }
```

**Figure 5.7** myHash member function for hash tables

myhash is called by insert and calls the overloaded function hash associated with HashedObj.
Figure 5.8 Example of a class that can be used as a HashedObj
```c++
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;
}
```

**Figure 5.9** makeEmpty, contains, and remove routines for separate chaining hash table
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;
}

Figure 5.10 insert routines for separate chaining hash table
The load factor $\lambda$ of a hash table of size $n$ with $k$ elements is $k/n$. This is the average list length (with 0-length lists included).

The cost of a search is the time required to evaluate the hash function plus the list traversal time — $\lambda$ if unsuccessful, or approximately $1 + \lambda/2$ if successful, on average.

The general rule is to make $n$ close to the number of elements expected so that $\lambda \approx 1$. We also make $n$ prime to ensure a good distribution.

The insert function of Figure 5.10 keeps $\lambda \leq 1$. 
An alternative to collision resolution with a list (which requires additional memory allocation time) is to try alternative cells until an empty one is found: \( h_0(x), h_1(x), h_2(x), \ldots, \) where

\[
h_i(x) = (\text{hash}(x) + f(i)) \mod n
\]

for collision strategy \( f \) with \( f(0) = 0 \). This is referred to as open addressing or a probing hash table and generally requires \( \lambda < 0.5 \).

**Linear Probing:** \( f \) is linear; typically \( f(i) = i \). This implies sequential search. Figure 5.11 depicts the result of inserting keys 89, 18, 49, 58, 69 with \( \text{hash}(x) = x \mod 10 \).
<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>18</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>

**Figure 5.11** Hash table with linear probing, after each insertion
5.4.1 Linear Probing

A free cell can be found if it exists, but it can take a long time to find it. Worse, even when the table is relatively empty, blocks of occupied cells form — primary clustering: any key that hashes into a cluster takes several steps to resolve the collision and then adds to the cluster. (Note that the cluster elements are not contiguous if \( f(i) = k \cdot i \) for \( k > 1 \).)

It can be shown that the expected number of probes is roughly \((1/2)[1 + 1/(1 - \lambda)^2]\) for insertions and unsuccessful searches, and \((1/2)[1 + 1/(1 - \lambda)]\) for successful searches. These values are 2.5 and 1.5 for \( \lambda = 0.5 \); \( \infty \) for \( \lambda = 1 \); 1 for \( \lambda = 0 \). If \( 0 \leq \lambda < 1 \) then \( 1/(1 - \lambda) \geq 1 \) and increases with \( \lambda \). In a random collision resolution strategy each probe is independent of the previous probes. The expected number of probes in an unsuccessful search is the number of probes to find an empty cell — \( 1/(1 - \lambda) \) since \( 1 - \lambda \) is the probability of hitting an empty cell. (\( \lambda \) is the fraction of nonempty cells when each cell has 0 or 1 items; \( 1 - \lambda \) empty cells per probe \( \Rightarrow \quad 1/(1 - \lambda) \) probes per empty cell.)
Linear Probing versus Random Strategy

The number of probes in a successful search for $x$ is the same as the number of probes that were required to insert $x$ (following an unsuccessful search). We estimate the number of probes for a successful search by averaging the unsuccessful search costs over the sequence of insertions starting with $\lambda = 0$:

$$
\frac{1}{\lambda} \int_{0}^{\lambda} \frac{1}{1-x} \, dx = \frac{1}{\lambda} \left[ -\ln(1-x) \right]_{0}^{\lambda} = \frac{1}{\lambda} \left[ \ln \left( \frac{1}{1-x} \right) \right]_{0}^{\lambda}
$$

$$
= \frac{1}{\lambda} \ln \left( \frac{1}{1-\lambda} \right)
$$

since

$$
E(f) = \frac{1}{b-a} \int_{a}^{b} f(x) \, dx \approx \frac{1}{N\Delta x} \sum_{i=1}^{N} f_i \Delta x = \frac{1}{N} \sum_{i=1}^{N} f_i.
$$
Figure 5.12 Number of probes plotted against load factor for linear probing (dashed) and random strategy (S is successful search, U is unsuccessful search, and I is insertion)
5.4.2 Quadratic Probing

The quadratic collision strategy $f(i) = i^2$ eliminates primary clustering but has secondary clustering: elements that hash to the same position probe the same alternative cells and increase the cluster size.

<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></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>58</td>
<td>58</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>69</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>18</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>

**Figure 5.13** Hash table with quadratic probing, after each insertion
**Problem**: There is no guarantee of finding an empty cell when the table is more than half full (or even before that if \( n \) is not prime).

**Theorem**: If quadratic probing is used, and the table size \( n \) is prime, then a new element can always be inserted if the table is at least half empty.

**proof**: Suppose \( n \geq 3 \) and \( n \) is prime. It suffices to prove that the first \( \lceil n/2 \rceil \) alternative locations \( h_0, h_0 + 1, h0 + 2^2, \ldots, h_0 + \lceil n/2 \rceil^2 \mod n \) are distinct because fewer than that many locations are occupied. Suppose the result is false; i.e., suppose \( i \neq j \) and \( 1 \leq i, j \leq n/2 \) but

\[
h(x) + i^2 = h(x) + j^2 \mod n.
\]

Then \( i^2 = j^2 \mod n \), and \( i^2 - j^2 = (i - j)(i + j) = 0 \mod n \). This implies that \( i - j \) or \( i + j \) is a multiple of \( n \) because \( n \) is prime. But \( i \neq j \) and \( i, j < n \) \( \Rightarrow \) \( i - j \neq 0 \mod n \), and \( i \neq j, i, j \leq n/2 \) \( \Rightarrow \) \( i + j \neq 0 \mod n \) — a contradiction. \( \square \)
Quadratic Probing continued

The proof shows that the first ⌈n/2⌉ hash function values (cell indices) are distinct. In fact, only the first ⌈n/2⌉ values are distinct. The sequence is then repeated in reverse order. The sequence for \( n = 13 \), for example, is 0, 1, 4, 9, 3, 12, 10, 10, 12, 3, 9, 4, 1, 0, 1, 4, ... In particular, \( i + j = 0 \mod n \) for \( i = ⌊n/2⌋ \), \( j = ⌈n/2⌉ \) \( ⇒ i + j = n \). Thus, an insertion can fail if the table is more than half full.

The number of distinct hash function values is even smaller if \( n \) is not prime; e.g., \( n = 16 \) \( ⇒ 0, 1, 4, 9, 0, 9, 4, 1, 0, 1, ... \)

If \( n = 4k + 3 \) and \( n \) is prime, and \( f(i) = ±i^2 \), then the entire table can be probed; all \( n \) hash values are generated.

Probing hash tables require lazy deletion because an insertion followed by a collision followed by a standard deletion of the originally inserted item causes a search for the second item to be unsuccessful (in error).
Figure 5.14 Class interface for hash tables using probing strategies, including the nested HashEntry class

```cpp
1 template<typename HashedObj>
2 class HashTable {
3   public:
4     explicit HashTable( int size = 101 );
5     bool contains( const HashedObj & x ) const;
6     void makeEmpty( );
7     bool insert( const HashedObj & x );
8     bool remove( const HashedObj & x );
9   
10    enum EntryType { ACTIVE, EMPTY, DELETED };
11   
12   private:
13     struct HashEntry {
14       HashedObj element;
15       EntryType info;
16     
17       HashEntry( const HashedObj & e = HashedObj( ), EntryType i = EMPTY )
18          : element( e ), info( i ) { }
19     
20     vector<HashEntry> array;
21     int currentSize;
22   
23     bool isActive( int currentPos ) const;
24     int findPos( const HashedObj & x ) const;
25     void rehash( );
26     int myhash( const HashedObj & x ) const;
27   
28 };  
```
Figure 5.15

```cpp
explicit HashTable( int size = 101 ) : array( nextPrime( size ) )
{
    makeEmpty( );
}

void makeEmpty( )
{
    currentSize = 0;
    for( int i = 0; i < array.size( ); i++ )
        array[ i ].info = EMPTY;
}
```

**Figure 5.15** Routines to initialize quadratic probing hash table
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; }

Figure 5.16 contains routine (and private helpers) for hashing with quadratic probing
```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;
}
```

**Figure 5.17** insert and remove routines for hash tables with quadratic probing
Function `findPos` in Figure 5.16 (used by `contains`, `insert`, and `remove`) returns the position of the cell containing `HashedObj x` (which may have `info == DELETED`) if `x` was ever inserted, or the position of an empty cell into which `x` could be inserted otherwise.

Figure 5.16 lines 12-15 implement the quadratic collision resolution strategy using the fact that, for \( f(i) = i^2 \), \( f(i) = f(i - 1) + 2i - 1 \), so that the offset is \( 2i - 1 \) — initially 1 for \( i = 1 \), and doubled for each subsequent probe.

Note that insertion followed by deletion followed by reinsertion of `x` increments `currentSize` twice. This can cause the table to fill prematurely and makes `currentSize` incorrect (which is not a problem because `currentSize` is only used to flag a call to `rehash` when \( \lambda \) exceeds 0.5). Member function `insert` should test for `array[currentPos].info == DELETED` at line 7 and just change it to `ACTIVE` if true instead of invoking the `HashEntry` constructor (but `currentSize` would still be incorrect.)
5.4.3 Double Hashing

The cost of secondary clustering has been experimentally determined to be less than half a probe per search. It can be eliminated at the cost of a second evaluation of a hash function — *double hashing*:

\[ f(i) = i \times \text{hash}_2(x), \quad \text{hash}_2(x) \neq 0. \]

We may have \( \text{hash}(y) = \text{hash}(x) \), but \( \text{hash}(y) + \text{hash}_2(y) \) is not likely to coincide with \( \text{hash}(x) + \text{hash}_2(x) \). We cannot use \( \text{hash}_2(x) = x \mod k \) because that would result in \( f(i) = 0 \) for all \( i \) when \( x \mod k = 0 \). A good choice for which all cells can be probed is

\[ \text{hash}_2(x) = R - x \mod R \quad \text{for} \quad R \text{ prime and } R < n. \]

The values are 1, 2, ... \( R \).
Simulations show that the expected number of probes is almost the same as for a random collision resolution strategy. However, quadratic probing is simpler and faster in practice, especially for keys like strings whose hash functions are expensive to compute.

If \( n \) is not prime we can run out of alternate locations prematurely; e.g., \( n = 2k \) and \( \text{hash}_2(x) = k \) results in the probe sequence

\[
(\text{hash}(x) + i \times k) \mod 2k
\]

which has only two distinct values: \( \text{hash}(x) \mod 2k \) and \( (\text{hash}(x) + k) \mod 2k \) since \( (a + n) \mod n = a \mod n \).

Figure 5.18 displays a table associated with \( R = 7 \). Insertion of 60 into the table would generate the sequence of probes 0, 3, 6, 9, 2 because \( \text{hash}_2(60) = 7 - 4 = 3 \).
<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>69</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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></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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 5.18** Hash table with double hashing after each insertion
Collision resolution strategy: $f : \mathbb{Z} \rightarrow \mathbb{Z}, f(0) = 0$.

Probe sequence: $h_i(x) = (\text{hash}(x) + f(i)) \mod n$, $(i = 0, 1, \ldots)$

table size $n$ prime.

Load factor: $\lambda < 0.5$

Offset: $\Delta \equiv f(i + 1) - f(i)$.

- **Linear probing**: $f(i) = ai$, constant $\Delta = a$, primary clustering
- **Quadratic probing**: $f(i) = i^2$, non-constant $\Delta = 2i + 1$, secondary clustering
- **Double hashing**: $f(i) = i \times \text{hash}_2(x)$, where $\text{hash}_2(x) \neq 0$;
  $\text{hash}_2(x) = k - x \mod k$ for $k$ prime, $\Delta = \text{hash}_2(x)$, no clustering
Rehashing is necessary when there is not enough space to simply use a very large table. We build another table, about twice as large, with a new hash function (for the new $n$ value) and scan the original hash table, computing the new hash value for each (non-deleted) element and inserting it in the new table.

Figure 5.20 displays a hash table with linear probing using the hash function $\text{hash}(x) = x \mod 7$ after the insertion sequence 13, 15, 24, 6, 23. Figure 5.21 depicts the result of scan-order insertion into a new table with $n = 17$.

The cost of rehashing is $O(n)$ but it only occurs after $n/2$ insertions. Hence the cost per insertion is $O(1)$.

The implementations for separate chaining and quadratic probing are in Figure 5.22.
Figures 5.20-5.21

**Figure 5.20** Hash table with linear probing after 23 is inserted

**Figure 5.21** Hash table after rehashing
/* 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);

    /* 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++);
        }
    }

Figure 5.22 Rehashing for both separate chaining hash tables and probing hash tables
5.6 Hash Tables in the Standard Library
Many compilers provide `hash_set` and `hash_map` with the same member functions as the `set` and `map` containers. The appropriate include directive and namespace are compiler dependent. Also required are type parameters for the type of key and type of value, the hash function, and an equality operator.

5.7 Extendible Hashing
When a table is too large to fit in memory, the critical cost is the number of disk accesses required for retrieval. Collisions could require several disk accesses per search even for a well-distributed hash table, and rehashing is very expensive. *Extendible hashing* allows a search to be performed in two disk accesses by storing the array as a B-tree of depth 1 with the root in memory.
A **symbol table** is used by a compiler to associate variables with addresses. Identifiers are generally short so that the hash function can be evaluated quickly. It is not necessary to order (alphabetize) the variables.

In a graph theory problem in which the nodes have names rather than numbers, the nodes are assigned integers 1, 2, ... as they are read in. Since ordered input sequences are likely, an AVL tree or hash table is needed.

Game playing programs map positions to moves with a hash function so that when a position recurs it is not necessary to recompute the move. This general feature is the **transposition table**.

On-line spelling checkers use pre-hashed dictionaries to allow words to be checked for correctness (presence in the dictionary) in constant time. (The alphabetical ordering is not needed.)
Tradeoffs between a hash table and a binary search tree include the following.

- **find and insert** require constant expected time with hashing; $O(\log N)$ in a BST. Since the evaluation of the hash function can be expensive, the constant cost might be higher than the $O(\log N)$ cost.

- The BST is necessary if order is needed; e.g., `findMin`, `findMax`.

- If bad insertion sequences (ordered keys) can occur, then hashing is better because balancing is expensive.