Goals:

- hashing
  - dictionary operations
  - general idea of hashing
  - hash functions
  - chaining
  - closed hashing
Dictionary operations

• search
• insert
• delete

Applications:

• employees in a company
• books in a library
• web pages (e.g. in web searching)
• geometric shapes in a graphics application
Dictionary operations

- search
- insert
- delete

<table>
<thead>
<tr>
<th></th>
<th>ARRAY</th>
<th>LINKED LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sorted</td>
<td>unsorted</td>
</tr>
<tr>
<td>Search</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Insert</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>delete</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

Exercise: Create a similar table separately for data movements and for comparisons.
Performance goal for dictionary operations:

\( O(n) \) is too inefficient.

Goal is to achieve each of the operations

(a) in \( O(\log n) \) on average

(b) worst-case \( O(\log n) \)

(c) constant time \( O(1) \) on average.

Data structure that achieve these goals:

(a) binary search tree

(b) balanced binary search tree (AVL tree)

(c) hashing. (but worst-case is \( O(n) \))
Hashing

- An important and widely useful technique for implementing dictionaries
- Constant time per operation (on the average)
- Worst case time proportional to the size of the set for each operation (just like array and linked list implementation)
General idea

\[ U = \text{Set of all possible keys: (e.g. 9 digit SS #)} \]

If \( n = |U| \) is not very large, a simple way to support dictionary operations is:

map each key \( e \) in \( U \) to a unique integer \( h(e) \) in the range \( 0 \ldots n - 1 \).

Boolean array \( H[0 \ldots n - 1] \) to store keys.
Figure 11.1  Implementing a dynamic set by a direct-address table $T$. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded, contain NIL.
Ideal case not realistic

- U, the set of all possible keys is usually very large so we can’t create an array of size $n = |U|$.
- Create an array $H$ of size $m$ much smaller than $n$.

- Actual keys present at any time will usually be smaller than $n$.
- Mapping from $U \rightarrow \{0, 1, \ldots, m - 1\}$ is called hash function.

Example: $D =$ students currently enrolled in courses, $U =$ set of all SS #’s, hash table of size $= 1000$

Hash function $h(x) =$ last three digits.
Example (continued)

Insert Student “Dan”  SS# = 1238769871
h(1238769871) = 871
Example (continued)

Insert Student “Tim” SS# = 1872769871

\( h(1238769871) = 871 \), same as that of Dan.

Collision

```
NULL
```

```
0 1 2 3
```

```
hash table
```

```
Dan
```

```
871
```

```
999
```

```
buckets
```

...
Hash Functions

If $h(k_1) = \beta = h(k_2)$: $k_1$ and $k_2$ have collision at slot $\beta$.

There are two approaches to resolve collisions.
Collision Resolution Policies

Two ways to resolve:
(1) Open hashing, also known as separate chaining
(2) Closed hashing, a.k.a. open addressing

Chaining: keys that collide are stored in a linked list.
Previous Example:

Insert Student “Tim” SS# = 1872769871
h(1238769871) = 871, same as that of Dan.

Collision

hash table

NULL

buckets

0 1 2 3

Dan

Tim

871

999
Open Hashing

The hash table is a pointer to the head of a linked list.

All elements that hash to a particular bucket are placed on that bucket's linked list.

Records within a bucket can be ordered in several ways:

- by order of insertion,
- by key value order,
- or by frequency of access order.
Open Hashing Data Organization
Implementation of open hashing - search

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

Find is a function in the STL class algorithm. Code for find is described below:

```cpp
template<class InputIterator, class T>
InputIterator find ( InputIterator first, InputIterator last,
                     const T& value ) { 
    for ( ;first!=last; first++)
        if ( *first==value ) break;
    return first; } 
```
Implementation of open hashing - insert

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 );
    return true;
}

The new key is inserted at the end of the list.
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;
}
Choice of hash function

A good hash function should:

• be easy to compute

• distribute the keys uniformly to the buckets

• use all the fields of the key object.
Example: key is a string over \{a, ..., z, 0, ... 9, _ \} 
Suppose hash table size is \( n = 10007 \).

(Choose table size to be a prime number.)

Good hash function: interpret the string as a number to base 37 and compute mod 10007.

\[
h(\text{“word”}) = \left(23 \times 37 + 15 \times 37 + 18 \times 37 + 4\right) \mod 10007
\]
Computing hash function for a string

Homer’s rule:

```c
int hash( const string & key )
{
    int hashVal = 0;

    for( int i = 0; i < key.length( ); i++ )
        hashVal = 37 * hashVal + key[ i ];

    return hashVal;
}
```