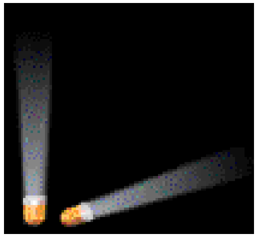
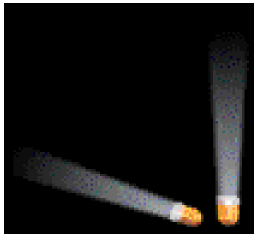
# Chapter Eleven

## Hashing

# Dictionaries Again

- Collection of pairs.
  - (key, element)
  - Pairs have different keys.
- Operations.
  - Get(theKey)
  - Delete(theKey)
  - Insert(theKey, theElement)

# Hash Tables

- Worst-case time for Get, Insert, and Delete is O(size).
- Expected time is O(1).

# Ideal Hashing

- Uses a 1D array (or table) table[0:b-1].
  - Each position of this array is a bucket.
  - A bucket can normally hold only one dictionary pair.
- Uses a hash function f that converts each key k into an index in the range [0, b-1].
  - f(k) is the home bucket for key k.
- Every dictionary pair (key, element) is stored in its home bucket table[f[key]].

# Ideal Hashing Example

- Pairs are: (22,a), (33,c), (3,d), (73,e), (85,f).
- Hash table is table[0:7], b = 8.
- Hash function is key/11.
- Pairs are stored in table as below:

| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] |
|---|---|---|---|---|---|---|---|
| (3,d) | | (22,a) | (33,c) | | | (73,e) | (85,f) |

- Get, Insert, and Delete take O(1) time.

# What Can Go Wrong?

| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] |
|---|---|---|---|---|---|---|---|
| (3,d) | | (22,a) | (33,c) | | | (73,e) | (85,f) |

- Where does (26,g) go?
- Keys that have the same home bucket are synonyms.
  - 22 and 26 are synonyms with respect to the hash function that is in use.
- The home bucket for (26,g) is already occupied.

# What Can Go Wrong?

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| (3,d) | | (22,a) | (33,c) | | | (73,e) | (85,f) |

- A collision occurs when the home bucket for a new pair is occupied by a pair with a different key.
- An overflow occurs when there is no space in the home bucket for the new pair.
- When a bucket can hold only one pair, collisions and overflows occur together.
- Need a method to handle overflows.

# Hash Table Issues

- Choice of hash function.
- Overflow handling method.
- Size (number of buckets) of hash table.

# Hash Functions

- Two parts:
  - Convert key into a nonnegative integer in case the key is not an integer.
    - Done by the function hash().
- Map an integer into a home bucket.
  - f(k) is an integer in the range [0, b-1], where b is the number of buckets in the table.

# String To Integer

- Each character is 1 byte long.
- An int is 4 bytes.
- A 2 character string s may be converted into a unique 4 byte non-negative int using the code:

```
int answer = s.at(0);
answer = (answer << 8) + s.at(1);
```

- Strings that are longer than 3 characters do not have a unique non-negative int representation.

# String To Nonnegative Integer

```
template<>
class hash<string>
{
  public:
  size_t operator()(const string theKey) const
  {// Convert theKey to a nonnegative integer.
      unsigned long hashValue = 0;
      int length = (int) theKey.length();
      for (int i = 0; i < length; i++)
          hashValue = 5 * hashValue +
                      theKey.at(i);

      return size_t(hashValue);
   }
};
```

# Map Into A Home Bucket

| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] |
|---|---|---|---|---|---|---|---|
| (3,d) | | (22,a) | (33,c) | | | (73,e) | (85,f) |

- Most common method is by division.

homeBucket = hash(theKey) % divisor;

- divisor equals number of buckets b.
- 0 <= homeBucket < divisor = b

# Uniform Hash Function

| (3,d) | | (22,a) | (33,c) | | | (73,e) | (85,f) |
|---|---|---|---|---|---|---|---|
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] |

- Let keySpace be the set of all possible keys.
- A uniform hash function maps the keys in keySpace into buckets such that approximately the same number of keys get mapped into each bucket.

# Uniform Hash Function

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| (3,d) | | (22,a) | (33,c) | | | (73,e) | (85,f) |
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] |

- Equivalently, the probability that a randomly selected key has bucket $i$ as its home bucket is $1/b$, $0 <= i < b$.

- A uniform hash function minimizes the likelihood of an overflow when keys are selected at random.

# Hashing By Division

- keySpace = all ints.
- For every b, the number of ints that get mapped (hashed) into bucket i is approximately $2^{32}/b$.
- Therefore, the division method results in a uniform hash function when keySpace = all ints.
- In practice, keys tend to be correlated.
- So, the choice of the divisor b affects the distribution of home buckets.

# Selecting The Divisor

- Because of this correlation, applications tend to have a bias towards keys that map into odd integers (or into even ones).
- When the divisor is an even number, odd integers hash into odd home buckets and even integers into even home buckets.
  - 20%14 = 6, 30%14 = 2, 8%14 = 8
  - 15%14 = 1, 3%14 = 3, 23%14 = 9
- The bias in the keys results in a bias toward either the odd or even home buckets.

# Selecting The Divisor

- When the divisor is an odd number, odd (even) integers may hash into any home.
  - 20%15 = 5, 30%15 = 0, 8%15 = 8
  - 15%15 = 0, 3%15 = 3, 23%15 = 8
- The bias in the keys does not result in a bias toward either the odd or even home buckets.
- Better chance of uniformly distributed home buckets.
- So do not use an even divisor.

# Selecting The Divisor

- Similar biased distribution of home buckets is seen, in practice, when the divisor is a multiple of prime numbers such as 3, 5, 7, …
- The effect of each prime divisor p of b decreases as p gets larger.
- Ideally, choose b so that it is a prime number.
- Alternatively, choose b so that it has no prime factor smaller than 20.

# STL hash_map

- Simply uses a divisor that is an odd number.
- This simplifies implementation because we must be able to resize the hash table as more pairs are put into the dictionary.
  - Array doubling, for example, requires you to go from a 1D array table whose length is b (which is odd) to an array whose length is 2b+1 (which is also odd).

# Overflow Handling

- An overflow occurs when the home bucket for a new pair (key, element) is full.
- We may handle overflows by:
  - Search the hash table in some systematic fashion for a bucket that is not full.
    - Linear probing (linear open addressing).
    - Quadratic probing.
    - Random probing.
  - Eliminate overflows by permitting each bucket to keep a list of all pairs for which it is the home bucket.
    - Array linear list.
    - Chain.

# Linear Probing – Get And Insert

- divisor = b (number of buckets) = 17.
- Home bucket = key % 17.

| 0 | | | | 4 | | | | 8 | | | | 12 | | | | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 0 | 45 | | | | 6 | 23 | 7 | | | 28 | 12 | 29 | 11 | 30 | 33 |

- Insert pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45

# Linear Probing – Delete

| 0 | | | | 4 | | | | 8 | | | | 12 | | | | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 0 | 45 | | | | 6 | 23 | 7 | | | 28 | 12 | 29 | 11 | 30 | 33 |

- Delete(0)

| 0 | | | | 4 | | | | 8 | | | | 12 | | | | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | | 45 | | | | 6 | 23 | 7 | | | 28 | 12 | 29 | 11 | 30 | 33 |

- Search cluster for pair (if any) to fill vacated bucket.

| 0 | | | | 4 | | | | 8 | | | | 12 | | | | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 45 | | | | | 6 | 23 | 7 | | | 28 | 12 | 29 | 11 | 30 | 33 |

# Linear Probing – Delete(34)

| 0 | | | | 4 | | | | 8 | | | | 12 | | | | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 0 | 45 | | | | 6 | 23 | 7 | | | 28 | 12 | 29 | 11 | 30 | 33 |

| 0 | | | | 4 | | | | 8 | | | | 12 | | | | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 45 | | | | 6 | 23 | 7 | | | 28 | 12 | 29 | 11 | 30 | 33 |

- Search cluster for pair (if any) to fill vacated bucket.

| 0 | | | | 4 | | | | 8 | | | | 12 | | | | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | | 45 | | | | 6 | 23 | 7 | | | 28 | 12 | 29 | 11 | 30 | 33 |

| 0 | | | | 4 | | | | 8 | | | | 12 | | | | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 45 | | | | | 6 | 23 | 7 | | | 28 | 12 | 29 | 11 | 30 | 33 |

# Linear Probing – Delete(29)

| 0 | | | | 4 | | | | 8 | | | | 12 | | | | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 0 | 45 | | | | 6 | 23 | 7 | | | 28 | 12 | 29 | 11 | 30 | 33 |

| 0 | | | | 4 | | | | 8 | | | | 12 | | | | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 0 | 45 | | | | 6 | 23 | 7 | | | 28 | 12 | | 11 | 30 | 33 |

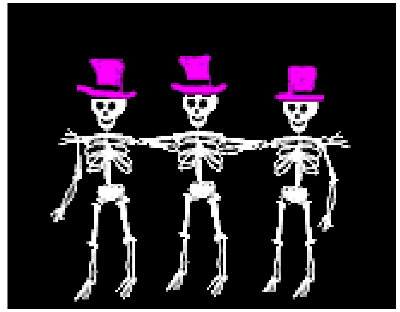
- Search cluster for pair (if any) to fill vacated bucket.

| 0 | | | | 4 | | | | 8 | | | | 12 | | | | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 0 | 45 | | | | 6 | 23 | 7 | | | 28 | 12 | 11 | | 30 | 33 |

| 0 | | | | 4 | | | | 8 | | | | 12 | | | | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 0 | 45 | | | | 6 | 23 | 7 | | | 28 | 12 | 11 | 30 | | 33 |

| 0 | | | | 4 | | | | 8 | | | | 12 | | | | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 0 | | | | | 6 | 23 | 7 | | | 28 | 12 | 11 | 30 | 45 | 33 |

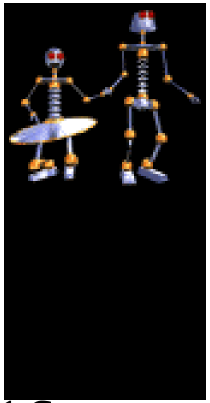
# Performance Of Linear Probing



| 0 | | | | 4 | | | | 8 | | | | 12 | | | | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 0 | 45 | | | | 6 | 23 | 7 | | | 28 | 12 | 29 | 11 | 30 | 33 |

- Worst-case find/insert/erase time is $\Theta(n)$, where $n$ is the number of pairs in the table.
- This happens when all pairs are in the same cluster.

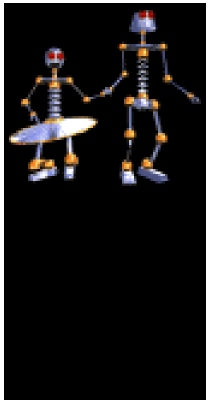
# Expected Performance



| 0 | | | | 4 | | | | 8 | | | | 12 | | | | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 34 | 0 | 45 | | | | 6 | 23 | 7 | | | 28 | 12 | 29 | 11 | 30 | 33 |

- $\alpha$ = loading density = (number of pairs)/b.
  - $\alpha = 12/17$.
- $S_n$ = expected number of buckets examined in a successful search when $n$ is large
- $U_n$ = expected number of buckets examined in a unsuccessful search when $n$ is large
- Time to put and remove governed by $U_n$.

# Expected Performance



- $S_n \sim \frac{1}{2}(1 + 1/(1 - \alpha))$
- $U_n \sim \frac{1}{2}(1 + 1/(1 - \alpha)^2)$
- Note that $0 <= \alpha <= 1$.

| ***alpha*** | ***$S_n$*** | ***$U_n$*** |
|---|---|---|
| ***0.50*** | 1.5 | 2.5 |
| ***0.75*** | 2.5 | 8.5 |
| ***0.90*** | 5.5 | 50.5 |

$\alpha <= 0.75$ is recommended.

# Hash Table Design

- Performance requirements are given, determine maximum permissible loading density.
- We want a successful search to make no more than 10 compares (expected).
  - $S_n \sim ½(1 + 1/(1 – \alpha))$
  - $\alpha <= 18/19$
- We want an unsuccessful search to make no more than 13 compares (expected).
  - $U_n \sim ½(1 + 1/(1 – \alpha)^2)$
  - $\alpha <= 4/5$
- So $\alpha <= \min\{18/19, 4/5\} = 4/5$.

# Hash Table Design

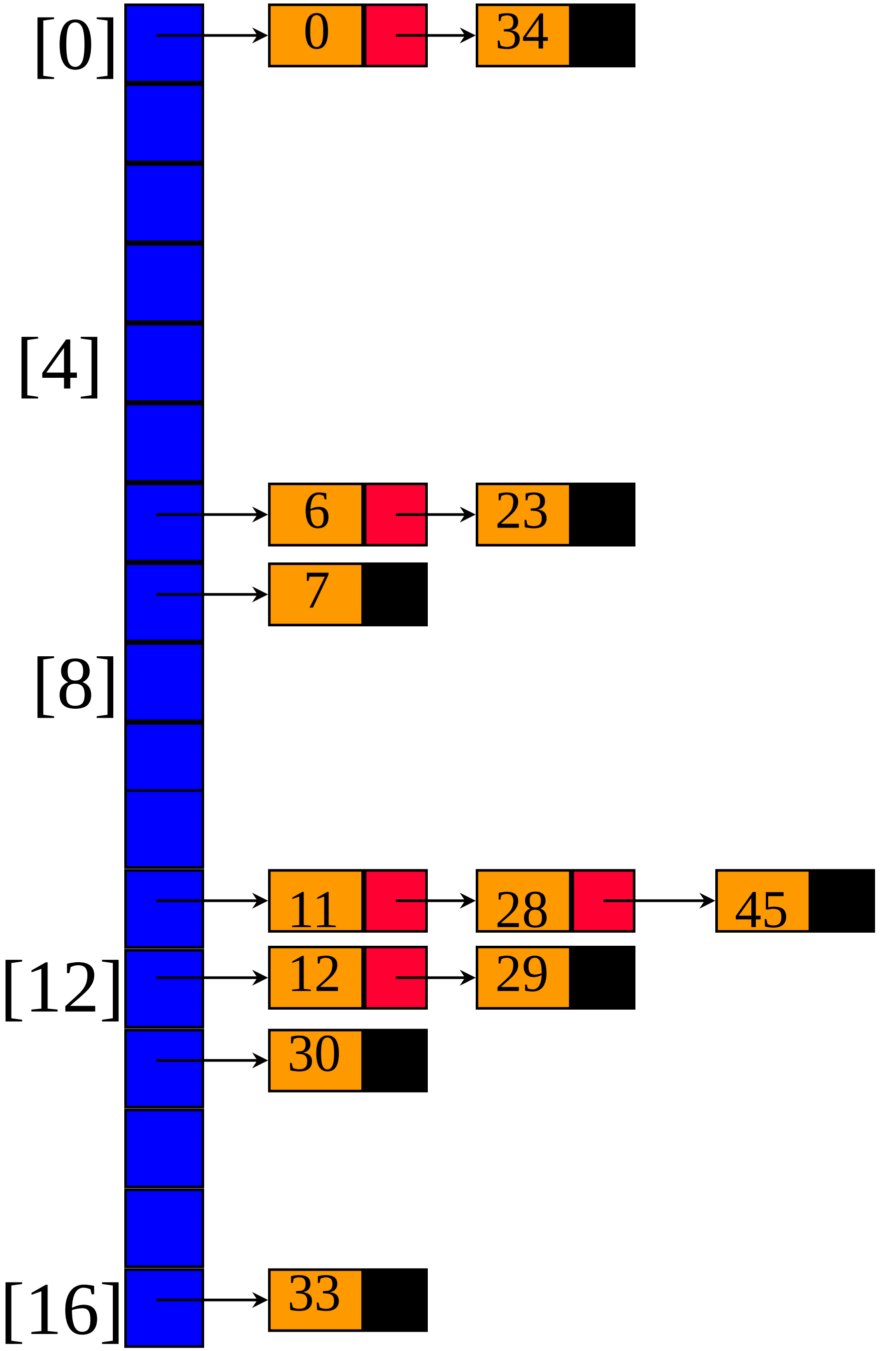
- Dynamic resizing of table.
  - Whenever loading density exceeds threshold (4/5 in our example), rehash into a table of approximately twice the current size.
- Fixed table size.
  - Know maximum number of pairs.
  - No more than 1000 pairs.
  - Loading density $<= 4/5 => b >= 5/4*1000 = 1250$.
  - Pick b (equal to divisor) to be a prime number or an odd number with no prime divisors smaller than 20.

# Linear List Of Synonyms

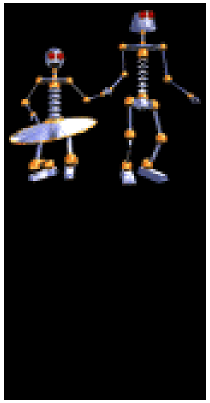
- Each bucket keeps a linear list of all pairs for which it is the home bucket.
- The linear list may or may not be sorted by key.
- The linear list may be an array linear list or a chain.

# Sorted Chains

- Put in pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45
- Home bucket = key % 17.



# Expected Performance



- Note that $\alpha >= 0$.
- Expected chain length is $\alpha$.
- $S_n \sim 1 + \alpha/2$.
- $U_n \sim \alpha$