Chapter 8 Hashing

- Concept of Hashing
- Static Hashing
- Dynamic Hashing

Concept of Hashing

- In CS, a **hash table**, or a **hash map**, is a data structure that associates keys (names) with values (attributes).
  - Look-Up Table
  - Dictionary
  - Cache
  - Extended Array
Origins of the Term

The term "hash" comes by way of analogy with its standard meaning in the physical world, to "chop and mix." D. Knuth notes that Hans Peter Luhn of IBM appears to have been the first to use the concept, in a memo dated January 1953; the term hash came into use some ten years later.

Example

A small phone book as a hash table.

(Figure is from Wikipedia)
Dictionaries

- Collection of pairs.
  - (key, value)
  - Each pair has a unique key.

- Operations.
  - Get (theKey)
  - Delete (theKey)
  - Insert (theKey, theValue)

Just An Idea

- Hash table:
  - Collection of pairs,
  - Lookup function (Hash function)

- Hash tables are often used to implement associative arrays,
  - Worst-case time for Get, Insert, and Delete is $O(size)$.
  - Expected time is $O(1)$. 

Search vs. Hashing

- Search tree methods: key comparisons
  - Time complexity: $O(\text{size})$ or $O(\log n)$
- Hashing methods: hash functions
  - Expected time: $O(1)$
- Types
  - Static hashing
  - Dynamic hashing

Static Hashing

Key-value pairs are stored in a fixed size table called a hash table.
- A hash table is partitioned into many buckets.
- Each bucket has many slots.
- Each slot holds one record.
- A hash function $f(x)$ transforms the identifier (key) into an address in the hash table.

```
<table>
<thead>
<tr>
<th>b buckets</th>
<th>0</th>
<th>1</th>
<th>s slots</th>
<th>s-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>b-1</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>b</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>
```

C-C Tsai
### Example of Hash table

<table>
<thead>
<tr>
<th></th>
<th>Slot 0</th>
<th>Slot 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>acos</td>
<td>atan</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>char</td>
<td>ceil</td>
</tr>
<tr>
<td>3</td>
<td>define</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>exp</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>float</td>
<td>floor</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**26 buckets**

**2 slots**

### Data Structure for Hash Table

```c
#define MAX_CHAR 10
#define TABLE_SIZE 13
typedef struct {
    char key[MAX_CHAR];
    /* other fields */
} element;

element hash_table[TABLE_SIZE];
```
Other Extensions

Hash List and Hash Tree
(Figure is from Wikipedia)

Formal Definition

- Hash Function
  - In addition, one-to-one / onto
Ideal Hashing

- Uses an array 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 $[0, b-1]$.
- Every dictionary pair (key, element) is stored in its home bucket table[$f[k]$].
Example

- Pairs are: (22, a), (33, c), (3, d), (73, e), (85, f).
- Hash table is $table[0:7]$, $b = 8$.
- Hash function is $key \mod 11$.

Example:

$22 \mod 11 = 2$, $33 \mod 11 = 3$,
$3 \mod 11 = 0$, ...

<table>
<thead>
<tr>
<th></th>
<th>(3, d)</th>
<th>(22, a)</th>
<th>(33, c)</th>
<th>(73, e)</th>
<th>(85, f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
</tr>
</tbody>
</table>

What Can Go Wrong?

- 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.
    - $22 \mod 11 = 2$ and $26 \mod 11 = 2$
- The bucket for (26, g) is already occupied.
Some Issues

- Choice of hash function.
  - Really tricky!
  - To avoid collision (two different pairs are in the same bucket.)
  - Size (number of buckets) of hash table.
- Overflow handling method.
  - Overflow: there is no space in the bucket for the new pair.

Example

<table>
<thead>
<tr>
<th></th>
<th>Slot 0</th>
<th>Slot 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>acos</td>
<td>atan</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>char</td>
<td>ceil</td>
</tr>
<tr>
<td>3</td>
<td>define</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>exp</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>float</td>
<td>floor</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

synonyms: char, ceil, clock

overflow
Choice of Hash Function

- Requirements
  - easy to compute
  - minimal number of collisions
- If a hashing function groups key values together, this is called clustering of the keys.
- A good hashing function distributes the key values uniformly throughout the range.

Some hash functions

- Division
  - $H(x) = \text{return } x \mod k$
- Middle of square
  - $H(x) = \text{return middle digits of } x^2$
- Multiplicative:
  - $H(x) = \text{return the first few digits of the fractional part of } x^k, \text{ where } k \text{ is a fraction. (advocated by D. Knuth in TAOCPP vol. III)}$
- Digit analysis:
  - If all the keys have been known in advance, then we could delete the digits of keys having the most skewed distributions, and use the rest digits as hash address.
Some hash functions (Cont’d)

Folding:
- Partition the identifier x into several parts, and add the parts together to obtain the hash address

Example: if x=12320324111220 and partition x into 
p1=123, p2=203, p3=241, p4=112, p5=20; then return the address
  \[ \Sigma pi = 123+203+241+112+20 = 699 \]
- Shift folding vs. folding at the boundaries
  \[ \Sigma pi = 123+302+241+211+20 = 897 \]
  (reverse p2 and p4)

Hashing By Division
- Domain is all integers.
- For a hash table of size b, the number of integers that get hashed into bucket i is approximately \( 2^{32}/b \).
- The division method results in a uniform hash function that maps approximately the same number of keys into each bucket.
Hashing By Division II

In practice, keys tend to be correlated.

- If divisor is an even number, odd integers hash into odd home buckets and even integers into even home buckets.
  
  Example: $20 \% 14 = 6$, $30 \% 14 = 2$, $8 \% 14 = 8$
  $15 \% 14 = 1$, $3 \% 14 = 3$, $23 \% 14 = 9$

- If divisor is an odd number, odd (even) integers may hash into any home.
  
  Example: $20 \% 15 = 5$, $30 \% 15 = 0$, $8 \% 15 = 8$
  $15 \% 15 = 0$, $3 \% 15 = 3$, $23 \% 15 = 8$

Hashing By Division III

- 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 large prime number $b$. Alternatively, choose $b$ so that it has no prime factors smaller than 20.
### Hash Algorithm via Division

```c
void init_table(element ht[])
{
    int i;
    for (i=0; i<TABLE_SIZE; i++)
        ht[i].key[0]=NULL;
}

int hash(char *key)
{
    return (transform(key) % TABLE_SIZE);
}

int transform(char *key)
{
    int number=0;
    while (*key) number += *key++;
    return number;
}
```

### Criterion of Hash Table

- The **key density** (or identifier density) of a hash table is the ratio $n/T$
  - $n$ is the number of keys in the table
  - $T$ is the number of distinct possible keys
- The **loading density** or **loading factor** of a hash table is $\alpha = n/(sb)$
  - $s$ is the number of slots
  - $b$ is the number of buckets
Example

If \( b = 26, s = 2, n = 10 \)
Then \( \alpha = \frac{n}{sb} = \frac{10}{52} = 0.19 \), \( f(x) = \) the first char of \( x \)

<table>
<thead>
<tr>
<th></th>
<th>Slot 0</th>
<th>Slot 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>acos</td>
<td>atan</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>char</td>
<td>ceil</td>
</tr>
<tr>
<td>3</td>
<td>define</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>exp</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>float</td>
<td>floor</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

- **Open addressing** ensures that all elements are stored directly into the hash table, thus it attempts to resolve collisions using various methods.
- **Linear Probing** resolves collisions by placing the data into the next open slot in the table.

Linear Probing - Get And Insert

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

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

e.g., 6 % 17=6, 12 % 17=12, 34 % 17=0, 29 % 17=12 ...

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>0</td>
<td>45</td>
<td>6</td>
<td>23</td>
</tr>
</tbody>
</table>

C-C Tsai
**Linear Probing – Delete**

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>0</td>
<td>45</td>
<td>6</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>12</td>
<td>29</td>
<td>11</td>
</tr>
</tbody>
</table>

- **Delete(0)**

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>45</td>
<td>6</td>
<td>23</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>12</td>
<td>29</td>
<td>11</td>
</tr>
</tbody>
</table>

Since 45 % 17 = 11 is NOT in right slot, so move it.

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

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>45</td>
<td>6</td>
<td>23</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>12</td>
<td>29</td>
<td>11</td>
</tr>
</tbody>
</table>

---

**Linear Probing – Delete(34)**

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>0</td>
<td>45</td>
<td>6</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>12</td>
<td>29</td>
<td>11</td>
</tr>
</tbody>
</table>

Since 0%17=0 and 45%17=11 are NOT in right slots, so move them.

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

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>6</td>
<td>23</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>12</td>
<td>29</td>
<td>11</td>
</tr>
</tbody>
</table>

---

C-C Tsai
Linear Probing – Delete(29)

```
0 4 8 12 16
34 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
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 45 33
```

Since 45 % 17 = 11 is NOT in right slot, so move it.

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

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

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$S_n$</th>
<th>$U_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.50</td>
<td>1.5</td>
<td>2.5</td>
</tr>
<tr>
<td>0.75</td>
<td>2.5</td>
<td>8.5</td>
</tr>
<tr>
<td>0.90</td>
<td>5.5</td>
<td>50.5</td>
</tr>
</tbody>
</table>

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

The proof refers to D. Knuth's TAOCP vol. III

$\alpha \leq 0.75$ is recommended

Linear Probing

```c
void linear_insert(element item, element ht[]) {
    int i, hash_value;
    i = hash_value = hash(item.key);
    while(strlen(ht[i].key)) {
        if (!strcmp(ht[i].key, item.key)) {
            fprintf(stderr, "Duplicate entry\n"); exit(1);
        }
        i = (i+1)%TABLE_SIZE;
        if (i == hash_value) {
            fprintf(stderr, "The table is full\n"); exit(1);
        }
    }
    ht[i] = item;
}
```
Problem of Linear Probing

- Identifiers tend to cluster together. Adjacent clusters tend to coalesce (or combine) that will increase the search time.

Coalesce Phenomenon

<table>
<thead>
<tr>
<th>bucket</th>
<th>x</th>
<th>bucket searched</th>
<th>bucket</th>
<th>x</th>
<th>bucket searched</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>acos</td>
<td>1</td>
<td>1</td>
<td>atoi</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>char</td>
<td>1</td>
<td>3</td>
<td>define</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>exp</td>
<td>1</td>
<td>5</td>
<td>ceil</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>cos</td>
<td>5</td>
<td>7</td>
<td>float</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>atol</td>
<td>9</td>
<td>9</td>
<td>floor</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>ctime</td>
<td>9</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>25</td>
</tr>
</tbody>
</table>

Average number of buckets examined is $41/11=3.73$ per identifier.

Quadratic Probing

- Linear probing searches buckets $(H(x)+i^2)\% b$.
- Quadratic probing uses a quadratic function of $i$ as the increment.
- Examine buckets $H(x)$,
  
  $(H(x)+i^2) \% b$, for $1 \leq i \leq (b-1)/2$

  $b$ is a prime number of the form $4j+3$, $j$ is an integer.

  - It may suffer the secondary clustering problem.
Double Hashing

Double hashing represents an improvement over linear and quadratic probing in that probe sequence are used. Its performance is more closed to uniform hashing.

\[ h(k,i) = (h_1(k) + i h_2(k)) \mod m \]

Example:

\[ h_1(k) = k \mod m \]
\[ h_2(k) = 1 + (k \mod m) \]

Example of Double Hashing

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>79</td>
<td></td>
<td></td>
<td>69</td>
<td></td>
<td>98</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>72</td>
<td></td>
<td></td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>50</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Insert 14

\[ h_1 = 14 \mod 13 = 1 \]
\[ h_2 = 1 + 14 \mod 11 = 1 + 3 = 4 \]

\[ h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m \]
\[ h(14,0) = (1 + 0 \cdot 4) \mod 13 = 1 \rightarrow 79 \]
\[ h(14,1) = (1 + 1 \cdot 4) \mod 13 = 5 \rightarrow 98 \]
\[ h(14,2) = (1 + 2 \cdot 4) \mod 13 = 9 \rightarrow 14 \]
Random Probing

- Random probing works incorporating with random numbers.
  - $H(x) = (H'(x) + S[i]) \mod b$
  - $S[i]$ is a table with size $b-1$
  - $S[i]$ is a random permutation of integers $[1,b-1]$.

Rehashing

- **Rehashing**: Try $H_1$, $H_2$, ..., $H_m$ in sequence if collision occurs. Here $H_i$ is a hash function.
- **Double hashing** is one of the best methods for dealing with collisions.
  - If the slot is full, then a second hash function is calculated and combined with the first hash function.
  - $H(k, i) = (H_1(k) + i H_2(k)) \mod m$
Summary: Hash Table Design

- Performance requirements are given, determine maximum permissible loading density. Hash functions must usually be custom-designed for the kind of keys used for accessing the hash table.
  - We want a successful search to make no more than 10 comparisons (expected).
    - $S_n \sim \frac{1}{2} (1 + 1/(1 - \alpha))$
    - $\alpha \leq 18/19$
  - We want an unsuccessful search to make no more than 13 comparisons (expected).
    - $U_n \sim \frac{1}{2} (1 + 1/(1 - \alpha)^2)$
    - $\alpha \leq 4/5$
  - So $\alpha \leq \min\{18/19, 4/5\} = 4/5$.

Summary: Hash Table Design II

- 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.
  - Loading density $\leq 4/5 \geq b \geq 5/4 \times 1000 = 1250$.
  - Pick $b$ (equal to divisor) to be a prime number or an odd number with no prime divisors smaller than 20.
Chaining

The idea of **Chaining** is to combine the linked list and hash table to solve the overflow problem.

```
#define MAX_CHAR 10
#define TABLE_SIZE 13
#define IS_FULL(ptr) (!(ptr))
typedef struct {
    char key[MAX_CHAR];
    /* other fields */
} element;
typedef struct list *list_pointer;
typedef struct list {
    element item;
    list_pointer link;
} list_pointer hash_table[TABLE_SIZE];
```
Sorted Chains

- Bucket = key % 17.

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

- e.g., 6 % 17=6
- 12 % 17=12
- 34 % 17=0
- 23 % 17=6

Expected Performance of Chaining

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

Refer to D. Knuth’s TAOCP vol. III for the proof.
Comparison

- Average number of bucket accesses per identifier retrieved.
  - If open addressing is used, then each table slot holds at most one element, therefore, the loading factor $\alpha$ can never be greater than 1.
  - If external chaining is used, then each table slot can hold many elements, therefore, the loading factor may be greater than 1.

<table>
<thead>
<tr>
<th>$\alpha = \frac{n}{b}$</th>
<th>.50</th>
<th>.75</th>
<th>.90</th>
<th>.95</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash Function</td>
<td>Chain</td>
<td>Open</td>
<td>Chain</td>
<td>Open</td>
</tr>
<tr>
<td>mid square</td>
<td>1.26</td>
<td>1.73</td>
<td>1.40</td>
<td>9.75</td>
</tr>
<tr>
<td>division</td>
<td>1.19</td>
<td>4.52</td>
<td>1.31</td>
<td>7.20</td>
</tr>
<tr>
<td>shift fold</td>
<td>1.33</td>
<td>21.75</td>
<td>1.48</td>
<td>65.10</td>
</tr>
<tr>
<td>bound fold</td>
<td>1.39</td>
<td>22.97</td>
<td>1.57</td>
<td>48.70</td>
</tr>
<tr>
<td>digit analysis</td>
<td>1.35</td>
<td>4.55</td>
<td>1.49</td>
<td>30.62</td>
</tr>
<tr>
<td>theoretical</td>
<td>1.25</td>
<td>1.50</td>
<td>1.37</td>
<td>2.50</td>
</tr>
</tbody>
</table>

(Adapted from V. Lum, P. Yuen, and M. Dodd, CACM, 1971, Vol. 14, No. 4)

Conclusion

- The main tradeoffs between these methods are that linear probing has the best cache performance but is most sensitive to clustering, while double hashing has poorer cache performance but exhibits virtually no clustering; quadratic probing falls in between the previous two methods.
Dynamic Hashing

- In this hashing scheme the set of keys can be varied, and the address space is allocated dynamically
  - File $F$: a collection of records
  - Record $R$: a key + data, stored in pages (buckets)
  - space utilization

\[
\text{NumberOfRecord} = \frac{\text{NumberOfPages} \times \text{PageCapacity}}{}
\]

Trie

- Trie: a binary tree in which an identifier is located by its bit sequence
- Key lookup is faster. Looking up a key of length $m$ takes worst case $O(m)$ time.
Dynamic Hashing Using Directories

- Some identifiers requiring 3 bits per character.

**Example:**

M (# of pages) = 4, P (page capacity) = 2

Allocation: lower order two bits

<table>
<thead>
<tr>
<th>Identifiers</th>
<th>Binary representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0</td>
<td>100 000</td>
</tr>
<tr>
<td>a1</td>
<td>100 001</td>
</tr>
<tr>
<td>b0</td>
<td>101 000</td>
</tr>
<tr>
<td>b1</td>
<td>101 001</td>
</tr>
<tr>
<td>c0</td>
<td>110 000</td>
</tr>
<tr>
<td>c1</td>
<td>110 001</td>
</tr>
<tr>
<td>c2</td>
<td>110 010</td>
</tr>
<tr>
<td>c3</td>
<td>110 011</td>
</tr>
</tbody>
</table>

h(a0,2) = 00
h(a1,2) = 01
h(b0,2) = 00
h(b1,2) = ...

A trie to hold identifiers

- Read it in reverse order.

  c5: 110 101
  c1: 110 001
Dynamic Hashing Using Directories II

- We need to consider some issues!
  - Skewed Tree,
  - Access time increased.
- Fagin et. al. proposed **extendible hashing** to solve above problems.

Dynamic Hashing Using Directories III

- A directories is a table of pointer of pages.
- The directory has $k$ bits to index $2^k$ entries.
- We could use a hash function to get the address of entry of directory, and find the page contents at the page.
The directory of the three tries

<table>
<thead>
<tr>
<th>a0</th>
<th>10000</th>
<th>h(a0,2)=00</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>10001</td>
<td>h(a1,2)=01</td>
</tr>
<tr>
<td>b0</td>
<td>10100</td>
<td></td>
</tr>
<tr>
<td>b1</td>
<td>10101</td>
<td></td>
</tr>
<tr>
<td>c0</td>
<td>11000</td>
<td></td>
</tr>
<tr>
<td>c1</td>
<td>11001</td>
<td></td>
</tr>
<tr>
<td>c2</td>
<td>11002</td>
<td></td>
</tr>
<tr>
<td>c3</td>
<td>11003</td>
<td></td>
</tr>
<tr>
<td>c5</td>
<td>11010</td>
<td></td>
</tr>
</tbody>
</table>

(a) 2 bits  (b) 3 bits  (c) 4 bits

Dynamic Hashing Using Directories IV

- It is obvious that the directories will grow very large if the hash function is clustering.
- Therefore, we need to adopt the uniform hash function to translate the bits sequence of keys to the random bits sequence.
- Moreover, we need a family of uniform hash functions, since the directory will grow.
Dynamic Hashing Using Directories IV

- a family of uniform hash functions:
  \[ hash_i: key \rightarrow \{0 \ldots 2^{i-1}\}, 1 \leq i \leq d \]

- If the page overflows, then we use \( hash_i \) to rehash the original page into two pages, and we coalesce two pages into one in reverse case.

- Thus we hope the family holds some properties like hierarchy.

Analysis

1. Only two disk accesses.
2. Space utilization
   - \( \sim 69 \% \)
   - If there are \( k \) records and the page size \( p \) is smaller than \( k \), then we need to distribute the \( k \) records into left page and right page. It should be a symmetric binomial distribution.
Analysis II

If there are $j$ records in the left page, then there are $k-j$ records in the right page. The probability is:

$$L(k) = \frac{1}{2^k} \sum_{i=0}^{k} \binom{k}{j} \{L(j) + L(k-j)\}$$

$$= 2^{1-k} \sum_{j=0}^{k} \binom{k}{j} L(j)$$

Thus the space utilization is

$$\frac{k}{pL(k)} \sim \ln 2 \sim 0.69$$

Overflow pages

To avoid doubling the size of directory, we introduce the idea of overflow pages, i.e.,

- If overflow occurs, then we allocate a new (overflow) page instead of doubling the directory.
- Put the new record into the overflow page, and put the pointer of the overflow page to the original page. (like chaining.)
Overflow pages II

- Obviously, it will improve the storage utilization, but increases the retrieval time.
- Larson et. al. concluded that the size of overflow page is from \( p \) to \( p/2 \) if 80% utilization is enough. (\( p \) is the size of page.)
- For better space utilization, we could monitor
  - Access time
  - Insert time
  - Total space utilization
- Fagin et al. conclude that it performed at least as well or better than B-tree, by simulation.

Extendible Hashing

Directoryless Dynamic Hashing

- If we have a contiguous space that is large enough to hold all the records, we could estimate the directory and leave the memory management mechanism to OS, e.g., paging.


Linear Hashing

- **Drawback** of previous mapping: It wastes space, since we need to double the contiguous space if page overflow occurs.

- How to **improve**: Intuitively, add only one page, and rehash this space!

![Diagram](image)


Eventually, the space is doubled. Begin new phase!
Linear Hashing II.

The suitable family of hashing functions:

\[ h_j(K) = g(K) \mod (N \times 2^j), \quad j = 0, 1, \ldots \]

\[ g(K) = (cK) \mod M, \]

Where \( N \) is the minimum size of hash table, \( c \) is a constant, and \( M \) is a large prime.


Example of Splitting a Bucket


The case that keys is rehashed into new page.
Recall Overflow pages: If overflow occurs, then we allocate a new (overflow) page instead of doubling the directory.

The family of hash function
$$\text{hash}(\text{key}, r) := \text{key} \pmod{2^{r-1}}$$

No new keys be rehashed into new pages.

Analysis

- Space utilization is not good!
  - [Litwin] ~ 60%
  - Litwin suggested to keep overflows until the space utilization exceeds the predefined amount.
  - It can also be solved by open addressing, etc.