Hashing

Letícia Rodrigues Bueno

Federal University of ABC (UFABC)

Hashing: Introduction

- By using a function $h(x)$, key x is used to compute an index into an array of slots;

Hashing: Introduction

- By using a function $h(x)$, key x is used to compute an index into an array of slots;
- **Ideally**: assigning each key to a unique slot;

Hashing: Introduction

- By using a function $h(x)$, key x is used to compute an index into an array of slots;
- **Ideally**: assigning each key to a unique slot;
- Average complexity of operations: $O(1)$;

Hashing: Introduction

- By using a function $h(x)$, key x is used to compute an index into an array of slots;
- **Ideally**: assigning each key to a unique slot;
- Average complexity of operations: $O(1)$;
- Worst-case complexity: $O(n)$;

Hashing: Definition

Hashing: Definition

- n keys stored in a sequential table T of dimension m in the interval $[0, m-1]$;

Hashing: Definition

- n keys stored in a sequential table T of dimension m in the interval $[0, m-1]$;
- Two steps:

Hashing: Definition

- n keys stored in a sequential table T of dimension m in the interval $[0, m-1]$;
- Two steps:
  1. Compute the value of **hash function**;

Hashing: Definition

- n keys stored in a sequential table T of dimension m in the interval $[0, m-1]$;
- Two steps:
  1. Compute the value of **hash function**;
  2. **Collisions resolution**;

Hash Function

Operations over keys are arithmetic, therefore:

Hash Function

Operations over keys are arithmetic, therefore:

First step: convert non-numerical keys into numbers:

Hash Function

Operations over keys are arithmetic, therefore:

First step: convert non-numerical keys into numbers:

$$\sum_{i=0}^{n-1} key[i] \times p[i]$$

Hash Function

Operations over keys are arithmetic, therefore:

First step: convert non-numerical keys into numbers:

$$\sum_{i=0}^{n-1} key[i] \times p[i]$$

where $key[i]$ is in ASCII or Unicode, and $p[i]$ is a weight.

Hash Function

In C/C++:

Hash Function

In C/C++:

```
void generateWeights(){
    for (int i = 0; i < n; i++)
        weights[i] = rand() % m + 1;
}
```

Hash Function

In C/C++:

```
void generateWeights(){
    for (int i = 0; i < n; i++)
        weights[i] = rand() % m + 1;
}
```

Objective: different keys with same characters have different weights to give different results for the hash function.

Hash Function

In C/C++:

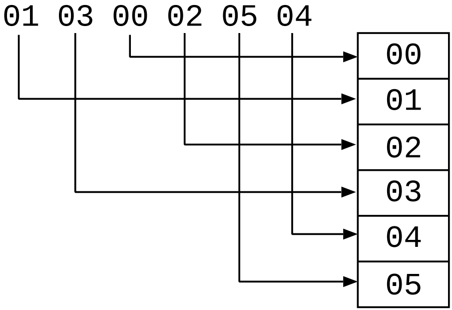
Hash Function

In C/C++:

```
int h(char key[ ]){
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += (unsigned int)key[i] * weights[i];
    return sum % m;
}
```

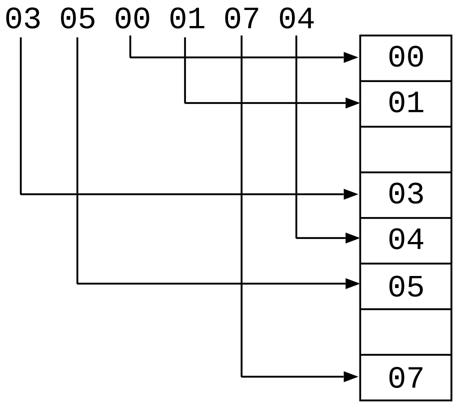
Trivial Hash Function

Key x is stored in the bucket x:



Trivial Hash Function

Key x is stored in the bucket x:



It can be used when $n = m$ or $n < m$ but $m - n$ is small.

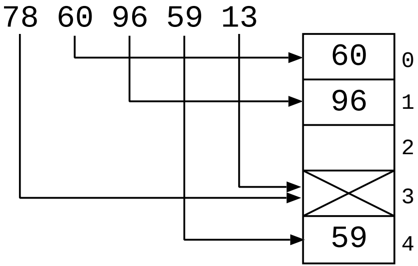
Collision resolution

Collision resolution

For keys $x \neq y$, we get $h(x) = h(y)$:

Collision resolution

For keys $x \neq y$, we get $h(x) = h(y)$:



h(x)=x mod 5

Hash Function

Ideally, a hash function:

Hash Function

Ideally, a hash function:

- generates small number of collisions;

Hash Function

Ideally, a hash function:

- generates small number of collisions;
- is easily computable: in time $O(1)$;

Hash Function

Ideally, a hash function:

- generates small number of collisions;
- is easily computable: in time $O(1)$;
- is uniform;

Hash Function

Ideally, a hash function:

- generates small number of collisions;
- is easily computable: in time $O(1)$;
- is uniform;

Uniform function: same probability for all slots, i.e., $\frac{1}{m}$ of getting $h(x)$.

Hash Function: Division Method

Hash Function: Division Method

Function: $h(x) = x \pmod{m}$

Hash Function: Division Method

Function: $h(x) = x \pmod{m}$
About choosing m:

Hash Function: Division Method

Function: $h(x) = x \pmod{m}$
About choosing m:

- **If m is even:**

Hash Function: Division Method

Function: $h(x) = x \pmod{m}$
About choosing m:

- **If m is even:** x even

Hash Function: Division Method

Function: $h(x) = x \pmod{m}$

About choosing m:

- **If m is even:** x even $\Rightarrow h(x)$ even;

Hash Function: Division Method

Function: $h(x) = x \pmod{m}$

About choosing m:

- **If m is even:** x even $\Rightarrow h(x)$ even;
- **If m is odd:**

Hash Function: Division Method

Function: $h(x) = x \pmod{m}$

About choosing m:

- **If m is even:** x even $\Rightarrow h(x)$ even;
- **If m is odd:** x odd

Hash Function: Division Method

Function: $h(x) = x \pmod{m}$

About choosing m:

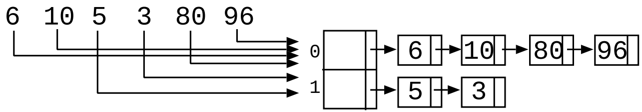
- **If m is even:** x even $\Rightarrow h(x)$ even;
- **If m is odd:** x odd $\Rightarrow h(x)$ odd;

Hash Function: Division Method

Function: $h(x) = x \pmod{m}$

About choosing m:

- **If m is even:** x even $\Rightarrow h(x)$ even;
- **If m is odd:** x odd $\Rightarrow h(x)$ odd;



h(x)=x mod 2

Hash Function: guidelines to choose m

Hash Function: guidelines to choose m

- m is prime number not close to a power of 2;

Hash Function: guidelines to choose *m*

- *m* is prime number not close to a power of 2;
- *m* does not have prime dividers smaller than 20;

Hash Function: guidelines to choose m

- m is prime number not close to a power of 2;
- m does not have prime dividers smaller than 20;
- if $m = 2^j$: $h(x)$ is defined only by the last j digits of x;

Hash Function: guidelines to choose m

- m is prime number not close to a power of 2;
- m does not have prime dividers smaller than 20;
- if $m = 2^j$: $h(x)$ is defined only by the last j digits of x;

Integer	Binary	Remainder from $2^3 = 8$
0	00000	0
1	00001	1
2	00010	2
3	00011	3
4	00100	4
5	00101	5
6	00110	6
7	00111	7
8	01000	0
9	01001	1
10	01010	2
11	01011	3
12	01100	4
13	01101	5
14	01110	6
15	01111	7
16	10000	0

Hash Function: guidelines to choose m

- m is prime number not close to a power of 2;
- m does not have prime dividers smaller than 20;
- if $m = 2^j$: $h(x)$ is defined only by the last j digits of x;

Integer	Binary	Remainder from $2^3 = 8$
0	00**000**	0
1	00**001**	1
2	00**010**	2
3	00**011**	3
4	00**100**	4
5	00**101**	5
6	00**110**	6
7	00**111**	7
8	01**000**	0
9	01**001**	1
10	01**010**	2
11	01**011**	3
12	01**100**	4
13	01**101**	5
14	01**110**	6
15	01**111**	7
16	10**000**	0

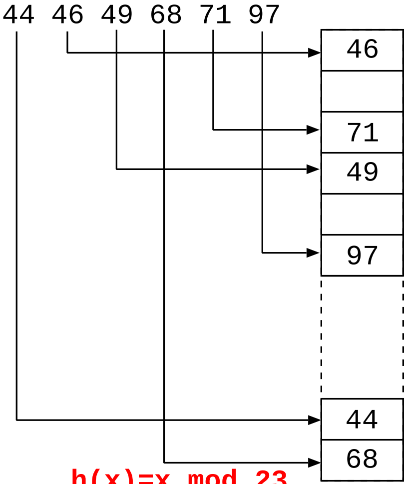
Hash Function: guidelines to choose m

Hash Function: guidelines to choose m

m is prime:

Hash Function: guidelines to choose *m*

m is prime:



h(x)=x mod 23

Collision resolution

Collision resolution

- Collision resolution is a critical procedure and should be carefully performed.

Collision resolution

- Collision resolution is a critical procedure and should be carefully performed.
- **Load factor:** $\alpha = \frac{n}{m}$.

Collision resolution

- Collision resolution is a critical procedure and should be carefully performed.
- **Load factor:** $\alpha = \frac{n}{m}$.
- The lower the load factor, the fewer collisions.

Collision resolution

- Collision resolution is a critical procedure and should be carefully performed.
- **Load factor:** $\alpha = \frac{n}{m}$.
- The lower the load factor, the fewer collisions.
- A small load factor does not assure absence of collisions.

Collision resolution

- Collision resolution is a critical procedure and should be carefully performed.
- **Load factor:** $\alpha = \frac{n}{m}$.
- The lower the load factor, the fewer collisions.
- A small load factor does not assure absence of collisions.
- **Birthday Problem (Feller):** in a group with randomly chosen 23 or more people, there is a probability greater than 50% of two people to have birthdays in the same day.

Collision resolution

- Collision resolution is a critical procedure and should be carefully performed.
- **Load factor:** $\alpha = \frac{n}{m}$.
- The lower the load factor, the fewer collisions.
- A small load factor does not assure absence of collisions.
- **Birthday Problem (Feller):** in a group with randomly chosen 23 or more people, there is a probability greater than 50% of two people to have birthdays in the same day.
- In a Hash Table, this means: $\alpha = \frac{23}{365} = 0,063$.

Collision resolution

- Collision resolution is a critical procedure and should be carefully performed.
- **Load factor:** $\alpha = \frac{n}{m}$.
- The lower the load factor, the fewer collisions.
- A small load factor does not assure absence of collisions.
- **Birthday Problem (Feller):** in a group with randomly chosen 23 or more people, there is a probability greater than 50% of two people to have birthdays in the same day.
- In a Hash Table, this means: $\alpha = \frac{23}{365} = 0,063$.
- Even so, there are 50% chance of at least one collision.

Collision resolution

Collision resolution

Some strategies:

Collision resolution

Some strategies:

1. Separate chaining:

Collision resolution

Some strategies:

1. Separate chaining:
   1.1 with linked lists;

Collision resolution

Some strategies:

1. Separate chaining:
   1.1 with linked lists;
   1.2 with interior lists;

Collision resolution

Some strategies:

1. Separate chaining:
    1.1 with linked lists;
    1.2 with interior lists;
2. Collision resolution by Open Addressing:

Collision resolution

Some strategies:

1. Separate chaining:
   1.1 with linked lists;
   1.2 with interior lists;
2. Collision resolution by Open Addressing:
   2.1 Linear Probing;

Collision resolution

Some strategies:

1. Separate chaining:
   1.1 with linked lists;
   1.2 with interior lists;
2. Collision resolution by Open Addressing:
   2.1 Linear Probing;
   2.2 Quadratic probing;

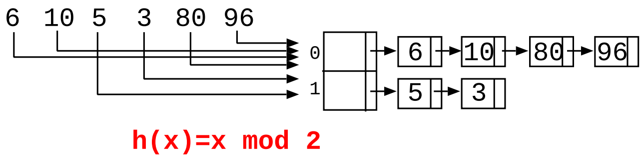
Collision resolution: Separate Chaining with linked lists

Collision resolution: Separate Chaining with linked lists

- Each bucket has a linked list;

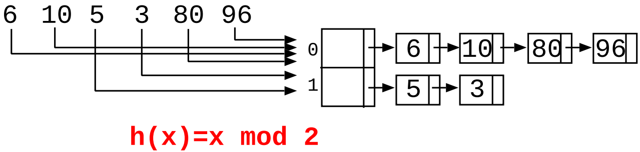
Collision resolution: Separate Chaining with linked lists

- Each bucket has a linked list;



Collision resolution: Separate Chaining with linked lists

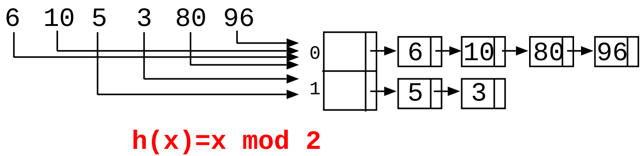
- Each bucket has a linked list;



- keys are added into the end of the list, which must be scanned to assure the key does not already exist;

Collision resolution: Separate Chaining with linked lists

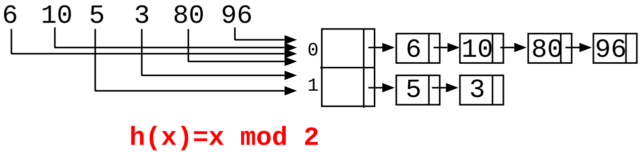
- Each bucket has a linked list;



- keys are added into the end of the list, which must be scanned to assure the key does not already exist;
- if hash function is uniform, searching has time $O(1)$;

Collision resolution: Separate Chaining with linked lists

- Each bucket has a linked list;



- keys are added into the end of the list, which must be scanned to assure the key does not already exist;
- if hash function is uniform, searching has time $O(1)$;
- linked list can be replace by an AVL tree for optimization;

Collision resolution: Separate chaining with interior lists

Collision resolution: Separate chaining with interior lists

- **Two spaces in the table with $m = p + s$:**

Collision resolution: Separate chaining with interior lists

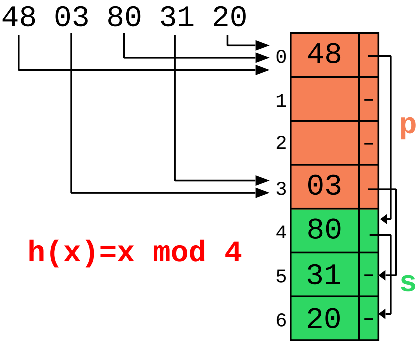
- **Two spaces in the table with $m = p + s$:**
    1. one space with size p for common addresses;

Collision resolution: Separate chaining with interior lists

- **Two spaces in the table with $m = p + s$:**
  1. one space with size p for common addresses;
  2. one space with size s for the synonym keys;

Collision resolution: Separate chaining with interior lists

- **Two spaces in the table with $m = p + s$:**
  1. one space with size p for common addresses;
  2. one space with size s for the synonym keys;



Collision resolution: Separate chaining with interior lists

Collision resolution: Separate chaining with interior lists

Problems:

Collision resolution: Separate chaining with interior lists

Problems:

- **possibility of false overflow**: space for synonym keys is full, but space for addresses has empty buckets;

Collision resolution: Separate chaining with interior lists

Problems:

- **possibility of false overflow**: space for synonym keys is full, but space for addresses has empty buckets;
- **To avoid false overflow:** increase collisions space;

Collision resolution: Separate chaining with interior lists

Problems:

- **possibility of false overflow**: space for synonym keys is full, but space for addresses has empty buckets;
- **To avoid false overflow:** increase collisions space;
- **Increasing collision space:** decreases efficiency of table;

Collision resolution: Separate chaining with interior lists

Problems:

- **possibility of false overflow**: space for synonym keys is full, but space for addresses has empty buckets;
- **To avoid false overflow:** increase collisions space;
- **Increasing collision space:** decreases efficiency of table;
- $p = 1$ and $s = m - 1 \Rightarrow$

Collision resolution: Separate chaining with interior lists

Problems:

- **possibility of false overflow**: space for synonym keys is full, but space for addresses has empty buckets;
- **To avoid false overflow:** increase collisions space;
- **Increasing collision space:** decreases efficiency of table;
- $p = 1$ and $s = m - 1 \Rightarrow$ **linked list!!!**

Collision resolution: Separate chaining with interior lists

Option: do not make difference between two spaces in table

Collision resolution: Separate chaining with interior lists

Option: do not make difference between two spaces in table

- any bucket can be for an address or for a synonym key;

Collision resolution: Separate chaining with interior lists

Option: do not make difference between two spaces in table

- any bucket can be for an address or for a synonym key;
- **Problem:** secondary collisions;

Collision resolution: Separate chaining with interior lists

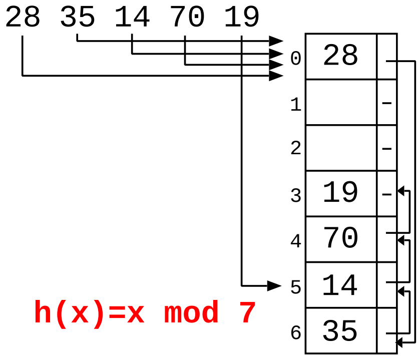
Option: do not make difference between two spaces in table

- any bucket can be for an address or for a synonym key;
- **Problem:** secondary collisions;
- **Mixing different lists:** decrease of efficiency;

Collision resolution: Separate chaining with interior lists

Option: do not make difference between two spaces in table

- any bucket can be for an address or for a synonym key;
- **Problem:** secondary collisions;
- **Mixing different lists:** decrease of efficiency;



h(x)=x mod 7

Collision resolution: Open Addressing

Collision resolution: Open Addressing

- Idea: storing synonym keys in the table;

Collision resolution: Open Addressing

- Idea: storing synonym keys in the table;
- No pointers;

Collision resolution: Open Addressing

- Idea: storing synonym keys in the table;
- No pointers;
- Calculates next address to be verified;

Collision resolution: Open Addressing

- Idea: storing synonym keys in the table;
- No pointers;
- Calculates next address to be verified;
- **Successful search:** it ends when finds the key;

Collision resolution: Open Addressing

- Idea: storing synonym keys in the table;
- No pointers;
- Calculates next address to be verified;
- **Successful search:** it ends when finds the key;
- **Unsuccessful search:** it ends when finds an empty bucket or reaches the end of table;

Collision resolution: Open Addressing

- Idea: storing synonym keys in the table;
- No pointers;
- Calculates next address to be verified;
- **Successful search:** it ends when finds the key;
- **Unsuccessful search:** it ends when finds an empty bucket or reaches the end of table;
- **Hash function:** provides m address $h(x, k)$ for key x and $k = 0, \ldots, m-1$ trials;

Collision resolution: Open Addressing

- Idea: storing synonym keys in the table;
- No pointers;
- Calculates next address to be verified;
- **Successful search:** it ends when finds the key;
- **Unsuccessful search:** it ends when finds an empty bucket or reaches the end of table;
- **Hash function:** provides m address $h(x, k)$ for key x and $k = 0, \ldots, m - 1$ trials;
- **Address:** $h(x, 0)$;

Collision resolution: Open Addressing

- Idea: storing synonym keys in the table;
- No pointers;
- Calculates next address to be verified;
- **Successful search:** it ends when finds the key;
- **Unsuccessful search:** it ends when finds an empty bucket or reaches the end of table;
- **Hash function:** provides m address $h(x, k)$ for key x and $k = 0, \ldots, m - 1$ trials;
- **Address:** $h(x, 0)$;
- **If collision:** $h(x, 0)$, $h(x, 1)$, $h(x, 2), \ldots, h(x, m - 1)$;

Collision resolution: Open Addressing

Collision resolution: Open Addressing

Pseudocode for searching in Open Addressing:

Collision resolution: Open Addressing

Pseudocode for searching in Open Addressing:

```
searchOpenAd(x, end, a):
    a ← 3; k ← 0;
    while k < m do
        end ← h(x, k);
        if (T[end].key = x) then
            a ← 1;
            k ← m;
        else if (T[end].key = null) then
            a ← 2;
            k ← m;
        else k ← k + 1;
```

where $a = 1$ means the key was found, $a = 2$ or 3 means not found key. If $a = 2$, *end* gives an empty position.

Open Addressing: Linear Probing

Open Addressing: Linear Probing

Linear Probing:

Open Addressing: Linear Probing

Linear Probing:

- Suppose address of key x is $h'(x)$, i.e., $h'(x) = h(x, 0)$;
- $h(x, k) = (h'(x) + k) \pmod{m}, 0 \leq k \leq m - 1$

Open Addressing: Linear Probing

Linear Probing:

- Suppose address of key x is $h'(x)$, i.e., $h'(x) = h(x, 0)$;
- $h(x, k) = (h'(x) + k) \pmod{m}, 0 \leq k \leq m - 1$

which is the same that:

Open Addressing: Linear Probing

Linear Probing:

- Suppose address of key x is $h'(x)$, i.e., $h'(x) = h(x, 0)$;
- $h(x, k) = (h'(x) + k) \pmod m, 0 \leq k \leq m - 1$

which is the same that:

- $h(x, k) = (h(x, k - 1) + 1) \pmod m, 0 \leq k \leq m - 1.$

Linear Probing:

- Suppose address of key x is $h'(x)$, i.e., $h'(x) = h(x, 0)$;
- $h(x, k) = (h'(x) + k) \pmod{m}, 0 \leq k \leq m - 1$

which is the same that:

- $h(x, k) = (h(x, k - 1) + 1) \pmod{m}, 0 \leq k \leq m - 1.$

Example:

Open Addressing: Linear Probing

Linear Probing:

- Suppose address of key x is $h'(x)$, i.e., $h'(x) = h(x, 0)$;
- $h(x, k) = (h'(x) + k) \pmod{m}, 0 \leq k \leq m - 1$

which is the same that:

- $h(x, k) = (h(x, k - 1) + 1) \pmod{m}, 0 \leq k \leq m - 1.$

Example:

$$
\begin{array}{lllcl}
h(x,0) & = 1 & & = & 1 \\
h(x,1) & = (h(x,0)+1) \pmod{m} & & = & 2 \\
h(x,2) & = (h(x,1)+1) \pmod{m} & & = & 3 \\
h(x,3) & = (h(x,2)+1) \pmod{m} & & = & 4 \\
\vdots & = & \vdots & = & \vdots
\end{array}
$$

Open Addressing: Linear Probing

Open Addressing: Linear Probing

Linear Probing:

Open Addressing: Linear Probing

Linear Probing:

- **Disadvantage:** primary grouping;

Open Addressing: Linear Probing

Linear Probing:

- **Disadvantage:** primary grouping;
- **Primary grouping:** long consecutive part of memory occupied;

Open Addressing: Linear Probing

Linear Probing:

- **Disadvantage:** primary grouping;
- **Primary grouping:** long consecutive part of memory occupied;
- the larger the primary grouping, the more likely it is to increase it more;

Open Addressing: Quadratic Probing

Open Addressing: Quadratic Probing

Quadratic Probing:

Open Addressing: Quadratic Probing

Quadratic Probing:

- Suppose address of key x is $h'(x)$, i.e., $h'(x) = h(x, 0)$;
- $h(x, k) = (h(x, k-1) + k) \pmod{m}, 0 < k < m.$

Open Addressing: Quadratic Probing

Quadratic Probing:

- Suppose address of key x is $h'(x)$, i.e., $h'(x) = h(x, 0)$;
- $h(x, k) = (h(x, k-1) + k) \pmod{m}, 0 < k < m.$

Example:

Open Addressing: Quadratic Probing

Quadratic Probing:

- Suppose address of key x is $h'(x)$, i.e., $h'(x) = h(x, 0)$;
- $h(x, k) = (h(x, k-1) + k) \pmod{m}, 0 < k < m.$

Example:

$$
\begin{array}{lllcc}
h(x,0) & = 1 & & = & 1 \\
h(x,1) & = (h(x,0)+1) & \pmod{m} & = & 2 \\
h(x,2) & = (h(x,1)+2) & \pmod{m} & = & 4 \\
h(x,3) & = (h(x,2)+3) & \pmod{m} & = & 7 \\
h(x,4) & = (h(x,3)+4) & \pmod{m} & = & 11 \\
h(x,5) & = (h(x,4)+5) & \pmod{m} & = & 16 \\
\vdots & = & \vdots & = & \vdots
\end{array}
$$

Open Addressing: Quadratic Probing

Open Addressing: Quadratic Probing

Quadratic Probing:

Open Addressing: Quadratic Probing

Quadratic Probing:

- avoids primary groupings;

Open Addressing: Quadratic Probing

Quadratic Probing:

- avoids primary groupings;
- but can generate secondary groupings;

Open Addressing: Quadratic Probing

Quadratic Probing:

- avoids primary groupings;
- but can generate secondary groupings;
- degradation produced by secondary groupings is still smaller than the one produced by primary groupings;

Conclusion

Conclusion

- Average complexity by operation: $O(1)$;

Conclusion

- Average complexity by operation: $O(1)$;
- Hash table is data structure which does not allow storing repeated elements;

Conclusion

- Average complexity by operation: $O(1)$;
- Hash table is data structure which does not allow storing repeated elements;
- No retrieval of elements in sequential order (sorted elements);

Conclusion

- Average complexity by operation: $O(1)$;
- Hash table is data structure which does not allow storing repeated elements;
- No retrieval of elements in sequential order (sorted elements);
- No retrieval of the predecessor or successor of an element;

Conclusion

- Average complexity by operation: $O(1)$;
- Hash table is data structure which does not allow storing repeated elements;
- No retrieval of elements in sequential order (sorted elements);
- No retrieval of the predecessor or successor of an element;
- For optimization of hash function, it's essential to know the nature of keys;

Conclusion

- Average complexity by operation: $O(1)$;
- Hash table is data structure which does not allow storing repeated elements;
- No retrieval of elements in sequential order (sorted elements);
- No retrieval of the predecessor or successor of an element;
- For optimization of hash function, it's essential to know the nature of keys;
- In the worst case, the order of operations can be $O(n)$ when all added elements collide.

Applications

Applications

- **search of elements in a database:**

Applications

- **search of elements in a database:**
    - data structures in memory;

Applications

- **search of elements in a database:**
  - data structures in memory;
  - databases;

Applications

- **search of elements in a database:**
  - data structures in memory;
  - databases;
  - search engines on the Internet;

Applications

- **search of elements in a database:**
  - data structures in memory;
  - databases;
  - search engines on the Internet;
- **verification of integrity of large amounts of data:**

Applications

- **search of elements in a database:**
  - data structures in memory;
  - databases;
  - search engines on the Internet;
- **verification of integrity of large amounts of data:**
  1. sending data with the result of hash function;

Applications

- **search of elements in a database:**
    - data structures in memory;
    - databases;
    - search engines on the Internet;
- **verification of integrity of large amounts of data:**
    1. sending data with the result of hash function;
    2. receptor calculates hash function over received data and gets a new result;

Applications

- **search of elements in a database:**
  - data structures in memory;
  - databases;
  - search engines on the Internet;
- **verification of integrity of large amounts of data:**
  1. sending data with the result of hash function;
  2. receptor calculates hash function over received data and gets a new result;
  3. if they are the same, data is ok (not corrupted);

Applications

- **search of elements in a database:**
  - data structures in memory;
  - databases;
  - search engines on the Internet;
- **verification of integrity of large amounts of data:**
  1. sending data with the result of hash function;
  2. receptor calculates hash function over received data and gets a new result;
  3. if they are the same, data is ok (not corrupted);
  4. if they are different, a new download must be done;

Applications

- **search of elements in a database:**
    - data structures in memory;
    - databases;
    - search engines on the Internet;
- **verification of integrity of large amounts of data:**
    1. sending data with the result of hash function;
    2. receptor calculates hash function over received data and gets a new result;
    3. if they are the same, data is ok (not corrupted);
    4. if they are different, a new download must be done;

    Example: download of a Linux image;

Applications

- **search of elements in a database:**
  - data structures in memory;
  - databases;
  - search engines on the Internet;
- **verification of integrity of large amounts of data:**
  1. sending data with the result of hash function;
  2. receptor calculates hash function over received data and gets a new result;
  3. if they are the same, data is ok (not corrupted);
  4. if they are different, a new download must be done;

  Example: download of a Linux image;
- **securely storing of passwords:** only the result of a hash function is stored in the server;

Applications

- **search of elements in a database:**
  - data structures in memory;
  - databases;
  - search engines on the Internet;
- **verification of integrity of large amounts of data:**
  1. sending data with the result of hash function;
  2. receptor calculates hash function over received data and gets a new result;
  3. if they are the same, data is ok (not corrupted);
  4. if they are different, a new download must be done; **Example:** download of a Linux image;
- **securely storing of passwords:** only the result of a hash function is stored in the server;
- **Examples of cryptographic hash function):** $MD5$ and family SHA ($SHA - 2$, $SHA - 256$ and $SHA - 512$);

Applications: search engines on the Internet

Applications: search engines on the Internet

1. ***spiders***: construct lists of words from Websites;

Applications: search engines on the Internet

1. ***spiders***: construct lists of words from Websites;
2. *spiders* start searching in the most used servers and popular sites;

Applications: search engines on the Internet

1. ***spiders***: construct lists of words from Websites;
2. *spiders* start searching in the most used servers and popular sites;
3. *spider* starts searching in popular sites, indexes words (and where they were found) and follows by hyperlinks;

Applications: search engines on the Internet

1. ***spiders***: construct lists of words from Websites;
2. *spiders* start searching in the most used servers and popular sites;
3. *spider* starts searching in popular sites, indexes words (and where they were found) and follows by hyperlinks;
4. **special attention:** meta tags, titles, subtitles;

Applications: search engines on the Internet

1. ***spiders***: construct lists of words from Websites;
2. *spiders* start searching in the most used servers and popular sites;
3. *spider* starts searching in popular sites, indexes words (and where they were found) and follows by hyperlinks;
4. **special attention:** meta tags, titles, subtitles;
5. **Google:** disregards articles (“a”, “an”, “the”);

Applications: search engines on the Internet

1. ***spiders***: construct lists of words from Websites;
2. *spiders* start searching in the most used servers and popular sites;
3. *spider* starts searching in popular sites, indexes words (and where they were found) and follows by hyperlinks;
4. **special attention:** meta tags, titles, subtitles;
5. **Google:** disregards articles (“a”, “an”, “the”);
6. **Construction of index:** hash table;

Applications: search engines on the Internet

1. ***spiders***: construct lists of words from Websites;
2. *spiders* start searching in the most used servers and popular sites;
3. *spider* starts searching in popular sites, indexes words (and where they were found) and follows by hyperlinks;
4. **special attention:** meta tags, titles, subtitles;
5. **Google:** disregards articles (“a”, “an”, “the”);
6. **Construction of index:** hash table;
7. **Use of weight:** to express number of occurrences of word, where it appears (title, meta tags, etc), if capital letters, type of font, etc;

Applications: search engines on the Internet

1. ***spiders***: construct lists of words from Websites;
2. *spiders* start searching in the most used servers and popular sites;
3. *spider* starts searching in popular sites, indexes words (and where they were found) and follows by hyperlinks;
4. **special attention:** meta tags, titles, subtitles;
5. **Google:** disregards articles (“a”, “an”, “the”);
6. **Construction of index:** hash table;
7. **Use of weight:** to express number of occurrences of word, where it appears (title, meta tags, etc), if capital letters, type of font, etc;

Combination of

Applications: search engines on the Internet

1. ***spiders***: construct lists of words from Websites;
2. *spiders* start searching in the most used servers and popular sites;
3. *spider* starts searching in popular sites, indexes words (and where they were found) and follows by hyperlinks;
4. **special attention:** meta tags, titles, subtitles;
5. **Google:** disregards articles (“a”, “an”, “the”);
6. **Construction of index:** hash table;
7. **Use of weight:** to express number of occurrences of word, where it appears (title, meta tags, etc), if capital letters, type of font, etc;

Combination of **efficient indexing**

Applications: search engines on the Internet

1. ***spiders***: construct lists of words from Websites;
2. *spiders* start searching in the most used servers and popular sites;
3. *spider* starts searching in popular sites, indexes words (and where they were found) and follows by hyperlinks;
4. **special attention:** meta tags, titles, subtitles;
5. **Google:** disregards articles (“a”, “an”, “the”);
6. **Construction of index:** hash table;
7. **Use of weight:** to express number of occurrences of word, where it appears (title, meta tags, etc), if capital letters, type of font, etc;

Combination of **efficient indexing** $+$

Applications: search engines on the Internet

1. ***spiders***: construct lists of words from Websites;
2. *spiders* start searching in the most used servers and popular sites;
3. *spider* starts searching in popular sites, indexes words (and where they were found) and follows by hyperlinks;
4. **special attention:** meta tags, titles, subtitles;
5. **Google:** disregards articles (“a”, “an”, “the”);
6. **Construction of index:** hash table;
7. **Use of weight:** to express number of occurrences of word, where it appears (title, meta tags, etc), if capital letters, type of font, etc;

Combination of **efficient indexing** + **effective storing**;

Exercises

Exercises

1. Consider the algorithms of sequential search, binary search and search in hash tables:

Exercises

1. Consider the algorithms of sequential search, binary search and search in hash tables:
   1.1 Describe the advantages and disadvantages for each one of these techniques, indicating in which situations you would use each one.

Exercises

1. Consider the algorithms of sequential search, binary search and search in hash tables:
   1.1 Describe the advantages and disadvantages for each one of these techniques, indicating in which situations you would use each one.
   1.2 What is the efficiency of memory use (relation between the needed space for data and the total space) for each method?

Exercises

1. Consider the algorithms of sequential search, binary search and search in hash tables:
   1.1 Describe the advantages and disadvantages for each one of these techniques, indicating in which situations you would use each one.
   1.2 What is the efficiency of memory use (relation between the needed space for data and the total space) for each method?
2. What are some properties of a good hash function?

Exercises

1. Consider the algorithms of sequential search, binary search and search in hash tables:
   1.1 Describe the advantages and disadvantages for each one of these techniques, indicating in which situations you would use each one.
   1.2 What is the efficiency of memory use (relation between the needed space for data and the total space) for each method?
2. What are some properties of a good hash function?
3. In which situations must a hash table be used?

Exercises

4. Describe two different mechanisms to solve the collisions problem for many keys in a same position of the table, highlighting advantages and disadvantages of each one.

Exercises

4. Describe two different mechanisms to solve the collisions problem for many keys in a same position of the table, highlighting advantages and disadvantages of each one.
5. For a set of n keys formed by the n first multiples of 7, how many collisions are produced when using the following hash functions, where x is a key?

Exercises

4. Describe two different mechanisms to solve the collisions problem for many keys in a same position of the table, highlighting advantages and disadvantages of each one.
5. For a set of n keys formed by the n first multiples of 7, how many collisions are produced when using the following hash functions, where x is a key?
   5.1 $x \pmod 7$

Exercises

4. Describe two different mechanisms to solve the collisions problem for many keys in a same position of the table, highlighting advantages and disadvantages of each one.
5. For a set of n keys formed by the n first multiples of 7, how many collisions are produced when using the following hash functions, where x is a key?
   - **5.1** $x \pmod 7$
   - **5.2** $x \pmod{14}$

Exercises

4. Describe two different mechanisms to solve the collisions problem for many keys in a same position of the table, highlighting advantages and disadvantages of each one.
5. For a set of n keys formed by the n first multiples of 7, how many collisions are produced when using the following hash functions, where x is a key?
   5.1 $x \pmod 7$
   5.2 $x \pmod{14}$
   5.3 $x \pmod 5$

Exercises

4. Describe two different mechanisms to solve the collisions problem for many keys in a same position of the table, highlighting advantages and disadvantages of each one.
5. For a set of n keys formed by the n first multiples of 7, how many collisions are produced when using the following hash functions, where x is a key?
   - 5.1 $x \pmod 7$
   - 5.2 $x \pmod{14}$
   - 5.3 $x \pmod 5$
6. Describe search, insertion and deletion algorithms for a hash table with collision resolution by separate chaining with linked lists.

Exercises

4. Describe two different mechanisms to solve the collisions problem for many keys in a same position of the table, highlighting advantages and disadvantages of each one.
5. For a set of n keys formed by the n first multiples of 7, how many collisions are produced when using the following hash functions, where x is a key?
   5.1 $x \pmod 7$
   5.2 $x \pmod{14}$
   5.3 $x \pmod 5$
6. Describe search, insertion and deletion algorithms for a hash table with collision resolution by separate chaining with linked lists.
7. Describe search and insertion algorithms for a hash table with collision resolution by separate chaining with linked lists, assuming there are no deletions.

Bibliography

FRANKLIN, Curt. *“How Internet Search Engines Work”*. 27 September 2000. HowStuffWorks.com. < *http://computer.howstuffworks.com/internet/basics/search-engine.htm* > 14 October 2012.

SZWARCFITER, J. L. and MARKENZON, L. Estruturas de Dados e seus Algoritmos, LTC, 1994. (in Portuguese)

ZIVIANI, N. Projeto de Algoritmos: com implementações em Java e C++, Cengage Learning, 2009. (in Portuguese)

Questions?