ĐẠI HỌC QUỐC GIA HÀ NỘI

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

BÁO CÁO

BẢNG BĂM VÀ ỨNG DỤNG

Học phần : **Cấu trúc dữ liệu & thuật toán**

Mã học phần : **MAT3514**

Nhóm thực hiện : **Nhóm G07**

Lớp chuyên ngành : **K66A2 - Toán tin**

Giảng viên hướng dẫn : **Nguyễn Thị Hồng Minh**

HÀ NỘI – 2023

LỜI NÓI ĐẦU

Cấu trúc dữ liệu & thuật toán là một học phần đóng vai trò quan trọng trong lĩnh vực khoa học máy tính và công nghệ thông tin. Học phần này cung cấp những kiến thức cơ bản nhất về các cấu trúc dữ liệu thông dụng và các thuật toán căn bản giúp sinh viên có thể tiếp cận, ứng dụng để giải quyết các bài toán được đặt ra trong thực tế. Tuy nhiên, để có thể giải quyết được các bài toán thực tiễn đòi hỏi sinh viên phải hiểu rõ các cấu trúc dữ liệu và nắm vững các thuật toán. Chính vì vậy, nhóm em quyết định thực hiện bài tập lớn cuối học phần MAT3514 về "Bảng băm và ứng dụng"; nhằm có cái nhìn tổng quan và sâu sắc về bảng băm và các thuật toán xung quanh, từ đó có thêm một trải nghiệm và góc nhìn mới về môn học này.

Tuy nhiên, do hạn chế về mặt kiến thức nên bài báo cáo sẽ không tránh khỏi những thiếu sót. Nhóm em luôn sẵn lòng đón nhận những nhận xét, đánh giá và chân thành mong nhận được những ý kiến đóng góp của thầy, cô và các bạn để có thể cải thiện và hoàn thiện hơn. Chúng em xin được gửi lời cảm ơn tới thầy cô đã giúp chúng em có được những kiến thức nền tảng vững chắc trong môn học này. Đồng thời nhóm em muốn bày tỏ lòng biết ơn sâu sắc tới cô Nguyễn Thị Hồng Minh, người đã quan tâm và dẫn dắt chúng em có thể hoàn thiện bài báo cáo đúng hướng và hiệu quả nhất.

Xin chân thành cảm ơn!

Hà Nội, ngày 26 tháng 12 năm 2023

Nhóm thực hiện

THÀNH VIÊN NHÓM

STT	Họ và tên	Mã sinh viên	Lớp
1	Nguyễn Ngọc Anh	21000663	K66A2
2	Lê Thị Nhung	21000697	K66A2
3	Kiều Thị Ngọc Linh	21000687	K66A2

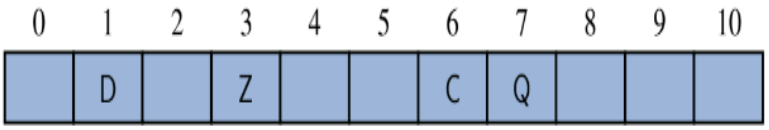
MỤC LỤC

CHƯƠNG I : KỸ THUẬT BĂM (HASHING)

1. Đặt vấn đề:

Giả sử có một đối tượng và muốn gán cho nó một khóa (key) để giúp tìm kiếm dễ dàng hơn.
Để lưu giữ cặp <key, value>, có thể sử dụng mảng bình thường để làm việc này.

- i = key
- a[i] = value



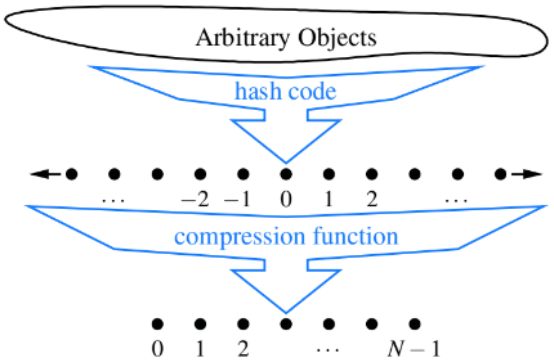
Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014b). Data structures and Algorithms in Java. John Wiley & Sons. p.410

Tuy nhiên, trong trường hợp phạm vi của khóa quá lớn và giá trị của key không phải là 1 dãy số nguyên liên tiếp, việc sử dụng mảng sẽ không hiệu quả.
=> Khi đó sẽ cần sử dụng "băm" (Hashing).

2. Băm là gì:

Băm (Hashing): là phương pháp chuyển đổi, lưu trữ dữ liệu thành giá trị số để dễ thao tác.
Ví dụ: Trong trường đại học, mỗi sinh viên được chỉ định một mã sinh viên không giống nhau và qua mã sinh viên đó có thể truy xuất các thông tin của sinh viên đó.
Băm (Hashing) được thực hiện qua 2 bước:

- Chuyển đổi giá trị của key thành giá nguyên (Hash code)
- Nén giá trị nguyên đó thành 1 số trong phạm vi [0, N - 1], N là kích thước của mảng/ bảng cần tạo (Compression function)



Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014b). *Data structures and Algorithms in Java*. John Wiley & Sons. p411

Nhận xét:

- Mã băm độc lập với kích thước bảng: được tính toán 1 lần và sử dụng cho nhiều bảng băm kích thước khác nhau
- Mã băm chung cho mọi đối tượng: không cần điều chỉnh theo kích thước của bảng
- Chỉ hàm nén (compression function) phụ thuộc vào kích thước bảng
- Dễ dàng thay đổi kích thước bảng: dựa trên số lượng đang lưu trữ => tối ưu về mặt không gian và thời gian
- Linh hoạt và thích ứng: khi chia thành 2 phần có thể bảo toàn tính linh hoạt trong quá trình thay đổi cấu trúc của bảng băm

CHƯƠNG II: HÀM BĂM (HASH FUNCTION)

Hàm băm được chia thành 2 phần riêng biệt: Mã băm (hash code) và hàm nén (Compression Functions)

1. Hash code:

1.1. Khái niệm:

Mã băm là một số nguyên được tạo ra từ khóa key và sử dụng để định vị một vị trí trong bảng băm với mục tiêu các khóa khác nhau sẽ có mã băm khác nhau, giảm thiểu xung đột hay.
Hàm băm "tốt" nếu tạo ra mã băm một cách dự đoán được và phủ đều trên không gian mã băm. (phân phối đều và một chiều)

1.2. Cách thiết kế mã băm trong Java:

1.2.1. Xử lý biểu diễn bit như một số nguyên (Treating the Bit Representation as an Integer):

Xử lý biểu diễn bit của một kiểu dữ liệu nhất định (X) như một số nguyên theo quy tắc sau:

- Xử lý kiểu dữ liệu không dài hơn 32 bit
  Xét các kiểu dữ liệu byte, short, int, char và float.
  - Với kiểu dữ liệu cơ bản như byte, short, int và char, có thể sử dụng giá trị trực tiếp của chúng làm hashcode bằng cách ép kiểu chúng về int.
  - Với kiểu dữ liệu float, có thể chuyển đổi giá trị thành một số nguyên sử dụng phương thức **Float.floatToIntBits(x)** sau đó sử dụng số nguyên này làm hash code của x

Một vài cài đặt hashCode() có sẵn trong Java:

- Boolean:

```
244     public static int hashCode(boolean value) {
245         return value ? 1231 : 1237;
246     }
```

- Integer

```
1213    public static int hashCode(int value) {
1214        return value;
1215    }
```

- Long

```
1437    public static int hashCode(long value) {
1438        return (int)(value ^ (value >>> 32));
1439    }
```

- Short

```
468    public static int hashCode(short value) {
469        return (int)value;
470    }
```

- Byte

```
463    public static int hashCode(byte value) {
464        return (int)value;
465    }
```

- Char

```
8903    public static int hashCode(char value) {
8904        return (int)value;
8905    }
```

- Xử lý kiểu dữ liệu dài hơn 32 bit
  Đối với kiểu dữ liệu biểu diễn bit dài hơn 32 (long, double) không thể tiếp cận trực tiếp như trên, vì vậy phải có giải pháp khác cho chúng. Có hai cách tiếp cận:
  - Chỉ sử dụng 32 bit cao (hoặc 32 bit thấp) của giá trị. Tuy nhiên cách này bỏ qua một nửa số thông tin trong khóa gốc và có thể dẫn đến va chạm (collision) nếu khóa chỉ khác nhau ở những bit này.
  - Kết hợp 32 bit cao và thấp tạo thành hash codes 32 bit, giữ lại tất cả các bit gốc bằng cách sử dụng bằng cách cộng 2 số 32 bits (bỏ qua tràn số) hoặc sử dụng phép XOR.
    Ta có biểu diễn theo công thức như sau:
    x biểu diễn nhị phân bởi n-tuple $(x_0, x_1, \ldots, x_{n-1})$
    - biểu diễn qua tổng: $\sum_{i=0}^{n-1} x_i$
    - Biểu diễn qua XOR: $x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1}$

1.2.2. *Mã băm đa thức (Polynomial Hash Codes):*

Hash code tổng và hash code XOR không phải là lựa chọn tốt cho chuỗi ký tự hoặc các đối tượng có chiều dài thay đổi có thể biểu diễn dưới

dạng (x_0, x_1, …, x_{n-1}), x_i xác định. Ví dụ: hash code 16 bit của chuỗi ký tự s là tổng của giá trị Unicode từng ký tự trong s. Tồn tại rất nhiều va chạm nếu xét các chuỗi "stop", "tops", "pots", and "spot".

Để giải quyết vấn đề này, xét đến vị trí của thành phần x_i, hash codes được biểu diễn theo công thức: $x_0 a^{n-1} + x_1 a^{n-2} + \ldots + x_{n-2} a + x_{n-1}$

1.2.3. *Mã băm dịch chu kỳ (Cyclic-Shift Hash Codes):*

Một biến thể của mã hash đa thức thay thế phép nhân với a bằng một phép dịch chu kỳ: dịch một phần tổng qua một số bit nhất định:
Ví dụ cho phép dịch bit 5:

```
1   package dsa_end.hashcode;
2
3   public class HashcodeCyclesShifting {
4
5       // triển khai mã băm chu kỳ với String
6       static int hashCode(String s) {
7           int h = 0;
8           for (int i = 0; i < s.length(); i++) {
9               h = (h << 5) | (h >>> 27);
10              h += (int) s.charAt(i);
11          }
12          return h;
13      }
        Run | Debug
14      public static void main(String[] args) {
15          String s = "Hello";
16          System.out.println(hashCode(s)); // 78921199
17      }
18  }
```

Giải thích:

- Xét dòng 9 và dòng 10:
  - với h là một số nguyên 32 bit, thực hiện phép dịch chu kỳ 5 bit:
    h = (h << 5) | (h >>> 27);
  - Dòng 10: Cập nhật giá trị h:
    $h \; += \;$ *giá trị nguyên của ký tự tại vị trí i của chuỗi s*
    h += (int) s.charAt(i);
- Đoạn mã trên có thể viết lại một cách chi tiết như sau:

```
14              int h0 = h;
15              int h1 = h << 5;
16              int h2 = h >>> 27;
17              int h3 = h1 | h2;
18              int h4 = (int) s.charAt(i);
19              int h5 = h3 + h4;
20              h = h5;
```

- Ngoài ra, cần sử dụng phương thức phụ trợ printBin() để dễ dàng quan sát quá trình dịch bit và biến đổi của h

```
27
28  static String printBin (int h) {
29      return (String.format(format:"%32s",
30              Integer.toBinaryString(h)).replace (oldChar:' ',newChar:'0'));
31  }
32
33  static void printBin (char c, int h0, int h1, int h2, int h3, int h4, int h5, int h) {
34      System.out.println("char: " + c);
35      System.out.println("h0: " + printBin(h0));
36      System.out.println("h1: " + printBin(h1));
37      System.out.println("h2: " + printBin(h2));
38      System.out.println("h3: " + printBin(h3));
39      System.out.println("h4: " + printBin(h4));
40      System.out.println("h5: " + printBin(h5));
41      System.out.println("h: " + printBin(h));
42      System.out.println("h (int): " + h);
43  }
```

- Xét String s = "IEllo! Whut's cp?"; tại vị trí ký tự ‘c’, phép dịch bit trên sẽ biến đổi như sau:

  char: c
  h0: 00011111010010001010110100101101
  h1: 11101001000101011010010110100000
  h2: 00000000000000000000000000000011
  h3: 11101001000101011010010110100011
  h4: 00000000000000000000000001100011
  h5: 11101001000101011010011000000110
  h: 11101001000101011010011000000110
  h (int): -384457210

  Trong h = (h << 5) | (h >>> 27); phép toán dịch bit được chuyển dịch theo chu kỳ 5. Tại h1 = (h << 5)5 bit phải cùng của h1 có giá trị bằng 0, điều đó gây nên tình trạng mất dữ liệu của 5 bit trái cùng đã dịch, dó đó h2 = (h >>> 27) được sử dụng để lưu các bit trái cùng sau khi dịch. Toán tử OR (|) h3 = h1 | h2 giúp lưu trữ lại những giá trị = 1 tại các vị trí cũng như tạo ra một giá trị h3 được “dịch vòng tròn” một cách hoàn chỉnh.
- Nếu không sử dụng toán tử OR mà chỉ sử dụng h = (h << 5) rất dễ gây ra xung đột:

```java
    // triển khai mã băm chu kỳ với String
    static int hashCode(String s) {
        int h = 0;
        for (int i = 0; i < s.length(); i++) {
            h = (h << 5);//| (h >>> 27);
            h += (int) s.charAt(i);
        }
        return h;
    }
```

```java
public static void main(String[] args) {
    String s1 = "IEllo! Whut's ap?";
    String s2 = "HeloeddddddeqP! Whut's ap?";

    System.out.println(hashCode(s1));
    System.out.println(hashCode(s2));
    System.out.println(hashCode(s1) == hashCode(s2));
    System.out.println(s1 == s2);
}
```

Chương trình cho ra kết quả như sau:

1430360639
1430360639
true
false

Tuy nhiên khi h = (h << 5) | (h >>> 27);(dòng 9), chương trình đưa ra kết quả:

1452810723
314976204
false
false

- Để xem thêm về va chạm của trường hợp này xin hãy tham khảo thêm tệp hashcode\circle-shift-collision-explanation.txt

1.3. Mã băm cài đặt trong Java:

Mã băm là một phần không thể thiếu trong ngôn ngữ lập trình Java. Trong đó, lớp Object, lớp cao nhất của các loại đối tượng, bao gồm một phương thức hashCode() mặc định trả về một số nguyên 32 bit kiểu int, được sử dụng làm hash codes của đối tượng. phiên bản mặc định của hashCode() được cung cấp bởi Object thường chỉ là một biểu diễn số nguyên được tạo ra từ địa chỉ bộ nhớ của đối tượng.

Tuy nhiên, cần phải thận trọng khi sử dụng phương thức hashCode() mặc định khi triển khai một lớp. Khi không cung cấp triển khai cụ thể, phiên bản mặc định sẽ được cung cấp bởi Object và chỉ sử dụng địa chỉ bộ nhớ của đối tượng để tạo hash codes, dẫn tới không đảm bảo độ tin cậy của hash code.

Ngoài ra, nếu hai khóa bằng nhau thì chúng phải có hash codes bằng nhau:

x.equals(y) => x.hashCode() == y.hashCode()

Ví dụ với lớp SinglyLinkedList:

- Định nghĩa: hai danh sách liên kết bằng nhau nếu chúng có chung các phần tử với số lượng và thứ tự như nhau.
- Sử dụng phương thức hashCode() của từng phần tử trong danh sách và thực hiện một phép toán XOR (exclusive-or) với mã băm hiện tại, sau đó thực hiện dịch chuyển chu kỳ (cyclic shift) 5 bit để tạo mã băm cuối cùng:

```
28      public int hashCode() {
29          int h = 0;
30      for (Node walk = head; walk != null; walk = walk.getNext()) {
31          h ^= walk.getElement().hashCode();
32          h = (h << 5) | (h >>> 27);
33      }
34          return h;
35      }
```

- Sau đó tiến hành kiểm tra với các danh sách liên kết:

```
37      public static void main(String[] args) {
38          SinglyLinkedList list = new SinglyLinkedList();
39          list.addFirst(e:"Hello");
40          list.addFirst(e:"World");
41          System.out.println(list.hashCode()); // 2087247443
42
43          SinglyLinkedList list2 = new SinglyLinkedList();
44          list2.addFirst(e:"Hello");
45          list2.addFirst(e:"World");
46          System.out.println(list.equals(list2)); // true
47          System.out.println(list2.hashCode()); //2087247443
48
49          SinglyLinkedList list3 = new SinglyLinkedList();
50          list3.addFirst(e:"World");
51          list3.addFirst(e:"Hello");
52          System.out.println(list.equals(list3)); // false
53          System.out.println(list3.hashCode()); // 124233296
54      }
```

- Nhận xét: có thể thấy list và list2 bằng nhau (được kiểm tra bằng phương thức equals() tại dòng 47) có hash code bằng nhau, tuy nhiên list và list3 khác nhau nên có hash code khác nhau.

2. Hàm nén (Compression function):

Mã băm cho một khóa k có thể nguyên âm hoặc vượt quá giới hạn của mảng bucket và thường không phù hợp để sử dụng ngay lập tức. Do đó, khi đã xác định một mã băm nguyên cho một đối tượng khóa k, cần phải ánh xạ số nguyên đó vào phạm vi [0,

N - 1], N là kích thước của mảng bucket. Phép tính này được gọi là hàm nén. Một hàm nén tốt là một hàm giảm thiểu tối đa va chạm cho tập hợp các mã băm phân biệt nhất định.

2.1. Phương pháp chia (The Division Method):

Phương pháp này sử dụng ánh xạ :

$i \to i \ mod \ N$ (N: kích thước mảng chứa)

Trong đó:

- N thường là số nguyên tố để tránh xung đột và giúp phân tán giá trị của bảng băm
- Ví dụ: Thêm các khóa với mã băm {200, 205, 210,...600} vào một mảng bucket có kích thước N:
  - Với N = 100: mỗi mã băm sẽ va chạm với ba mã băm khác.
  - Với N = 101: không có va chạm.

Một hàm băm "tốt" đảm bảo xác suất hai khóa khác nhau vào cùng một bucket là $1/N$. Tuy nhiên, N là một số nguyên tố có thể chưa đủ, đặc biệt trong một tập lặp lại liên tục mã băm có dạng $pN + q$ (p, q là số nguyên bất kỳ), trong trường hợp này sẽ xuất hiện va chạm.

Ví dụ:

- $N = 11$
- H = {13, 24, 35, 46, 57, 68} là tập các hash code.

Dễ thấy khi sử dụng phương pháp chia sẽ gây ra va chạm tại 2.

2.2. Phương pháp MAD (Multiply-Add-and-Divide):

Ánh xạ:

i -> [(ai + b) mod p] mod N

N: kích thước mảng chứa

p: số nguyên tố lớn hơn N

a,b : số nguyên được chọn ngẫu nhiên trong [0, p - 1]

Hàm nén này được chọn để loại bỏ các mẫu lặp lại trong tập hợp của các mã băm và đưa gần hơn tới một hàm băm "tốt" (xác suất của 2 khóa khác nhau bị va chạm là $1/N$)

Ví dụ:

- $N = 11$
- H = {13, 24, 35, 46, 57, 68} là tập các hash code.
- chọn a = 3, b = 7, p = 13
- Ta có:
  - $13 \to ((3 \times 13 + 7) \ mod \ 13) \ mod \ 11 = 7$
  - $24 \to ((3 \times 24 + 7) \ mod \ 13) \ mod \ 11 = 1$
  - $35 \to ((3 \times 35 + 7) \ mod \ 13) \ mod \ 11 = 8$

- $46 \rightarrow ((3 \times 46 + 7) \, mod \, 13) \, mod \, 11 = 2$
- $57 \rightarrow ((3 \times 57 + 7) \, mod \, 13) \, mod \, 11 = 9$
- $68 \rightarrow ((3 \times 68 + 7) \, mod \, 13) \, mod \, 11 = 3$

Nhận xét: Phương pháp MAD đã giải quyết vấn đề va chạm tại 2 của tập hash codes này trong phương pháp chia.

3. Va chạm (Hash Collision) và phương pháp xử lý va chạm:

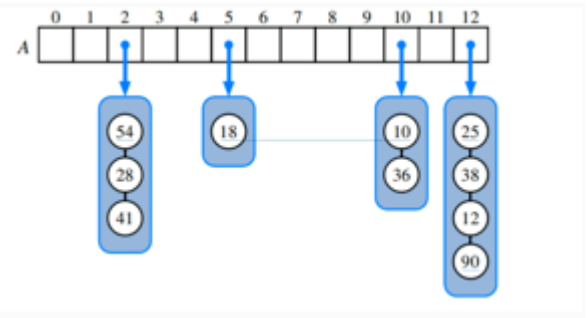
Va chạm xảy ra khi tồn tại hai khóa k_1, k_2 phân biệt sao cho sau khi được ánh xạ qua hàm băm chúng có cùng một giá trị.

Xét h là hàm băm:

Va chạm xảy ra $\Leftrightarrow \exists \, k_1 \neq k_2 : \; h(k_1) = h(k_2)$

3.1. Sử dụng chuỗi tách biệt (Separate Chaining) :

Một cách đơn giản để xử lý va chạm là để mỗi bucket A[j] lưu trữ container phụ của riêng nó, chứa tất cả các mục đầu vào (k, v) sao cho h(k) = j.



Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014b). *Data structures and Algorithms in Java*. John Wiley & Sons. p.417

Trường hợp xấu nhất khi các mục đầu vào đều được lưu trữ trong cùng 1 bucket, khi đó bảng băm chuyển về dạng mảng không sắp xếp với độ phức tạp của thuật toán tìm kiếm quay về O(n).

3.2. Open Addressing (Kỹ thuật định địa chỉ mở):

Băm theo chuỗi riêng biệt có nhiều tính chất tốt, chẳng hạn như cung cấp các triển khai đơn giản của các thao tác trên bảng, nhưng nó vẫn có một nhược điểm nhỏ: Nó yêu cầu sử dụng một cấu trúc dữ liệu phụ trợ để giữ các mục với các khóa xung đột (các Linkedlist - container).

Nếu không gian là một vấn đề quan trọng thì chúng ta có thể sử dụng cách tiếp cận thay thế của việc lưu trữ mỗi mục trực tiếp trong một chỉ mục nào đó của bảng, dựa vào các mục trống trong bảng để giúp giải quyết xung đột. Cách tiếp cận này tiết kiệm không gian vì không có cấu trúc phụ trợ nào được sử dụng, nhưng nó yêu cầu một chút phức tạp hơn để xử lý đúng các xung đột.

Có một số biến thể của cách tiếp cận này, được gọi chung là các biến thể của **địa chỉ mở**, mà chúng ta sẽ thảo luận tiếp theo. Địa chỉ mở yêu cầu rằng hệ số tải n/N luôn luôn tối đa là 1 và các mục nhập vào được lưu trữ trực tiếp trong các ô của mảng thùng.

Cài đặt:

```
17      public void insert(Key key, Value value) {
18          if (size + deleted == capacity) {
19              resize(capacity * 2);
20          }
21          int index = hash(key);
22          int i = 1;
23          while (keys[index] != null) {
24              index = (index + 1) % capacity;
25              i++;
26          }
27          keys[index] = key;
28          values[index] = value;
29          size++;
30      }
32      public void delete(Key key) {
33          int index = hash(key);
34          int i = 1;
35          while (keys[index] != key) {
36              index = (index + 1) % capacity;
37              i++;
38          }
39          keys[index] = null;
40          values[index] = null;
41          size--;
42          deleted++;
43
44          if (size + deleted == capacity / 4) {
45              resize(capacity / 2);
46          }
47      }
49      public Value search(Key key) {
50          int index = hash(key);
51          int i = 1;
52          while (keys[index] != key && keys[index] != null) {
53              index = (index + 1) % capacity;
54              i++;
55          }
56          return values[index];
57      }
```

3.3. Linear Probing (Phương pháp thăm dò tuyến tính) và các biến thể:

3.3.1. Linear Probing (Phương pháp thăm dò tuyến tính):

Quá trình chèn trong Linear Probing sẽ tiếp tục cho đến khi tìm thấy một ô trống tại vị trí mới được tính bằng công thức:

A[(j + i) mod N]

- j: vị trí ban đầu được xác định bởi hàm băm h(k)
- i: số lần "probing" (đang xem xét ô kế tiếp)
- N: kích thước của bảng băm

Nếu ô đó đã bị chiếm, ta tăng i và thử lại đến khi tìm được một ô trống.

Mã giả:

```
int i = 0;
while (position(j + i) is not insertable)
        i++;
insert at (j + i)
```

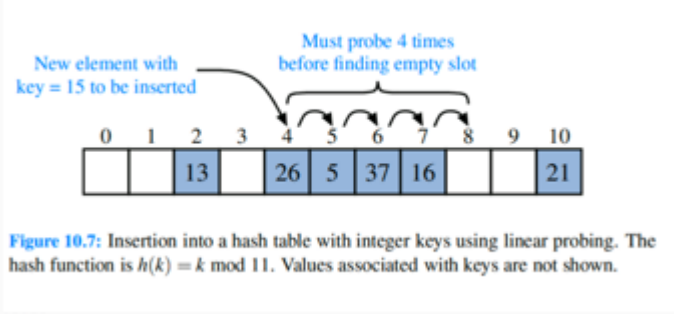


Figure 10.7: Insertion into a hash table with integer keys using linear probing. The hash function is $h(k) = k \bmod 11$. Values associated with keys are not shown.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014b). *Data structures and Algorithms in Java*. John Wiley & Sons. p.418

3.3.2. *Quadratic Probing (Thăm dò bậc hai)*:

Tương tự như Linear Probing, quá trình chèn trong Quadratic Probing sẽ được tiếp tục cho đến khi tìm thấy 1 ô trống nhưng với công thức:

A[(h(k)+ f(i)) mod N]

Trong đó:

$f(i) = i^2, \quad i = 1, 2, 3, \ldots$

h(k): hàm băm

N: kích thước của bảng băm

Cài đặt:

```
18      public void insert(Key key, Value value) {
19          if (size + deleted == capacity) {
20              resize(capacity * 2);
21          }
22          int index = hash(key);
23          int i = 1;
24          while (keys[index] != null) {
25              index = (index + i * i) % capacity;
26              i++;
27          }
28          keys[index] = key;
29          values[index] = value;
30          size++;
31      }
33      public void delete(Key key) {
34          int index = hash(key);
35          int i = 1;
36          while (keys[index] != key) {
37              index = (index + i * i) % capacity;
38              i++;
39          }
40          keys[index] = null;
41          values[index] = null;
42          size--;
43          deleted++;
44
45          if (size + deleted == capacity / 4) {
46              resize(capacity / 2);
47          }
48      }
50      public Value search(Key key) {
51          int index = hash(key);
52          int i = 1;
53          while (keys[index] != key && keys[index] != null) {
54              index = (index + i * i) % capacity;
55              i++;
56          }
57          return values[index];
58      }
```

3.3.3. *Double Hashing (Kỹ thuật ánh số đôi):*

Chọn 1 hàm băm phụ h. Khi A[h(k)] đã bị chiếm, thì ta lần lượt thử các ô:

A[(h(k) + f(i)) mod N]

Với f(i) = i * h(k), i = 1, 2, 3,...

Tiếp tục tăng giá trị của i cho đến khi tìm thấy một ô trống

Hàm băm phụ không được phép đánh giá bằng không, một lựa chọn phổ biến là: **h(k) = q - (k mod q)**, với q là số nguyên tố nhỏ hơn N. Ngoài ra, N nên là số nguyên tố.

Cài đặt:

```
22      private int hash2(Key key) {
23          return Math.abs(7 - (hash(key) % 7));
24      }
46      public void insert(Key key, Value value) {
47          if (size + deleted == capacity) {
48              resize(capacity * 2);
49          }
50          int index = hash(key);
51          int i = 1;
52          while (keys[index] != null) {
53              index = (index + i * hash2(key)) % capacity;
54              i++;
55          }
56          keys[index] = key;
57          values[index] = value;
58          size++;
59      }
61      public void delete(Key key) {
62          int index = hash(key);
63          int i = 1;
64          while (keys[index] != key) {
65              index = (index + i * hash2(key)) % capacity;
66              i++;
67          }
68          keys[index] = null;
69          values[index] = null;
70          size--;
71          deleted++;
72
73          if (size + deleted == capacity / 4) {
74              resize(capacity / 2);
75          }
76      }
```

```java
78      public Value search(Key key) {
79          int index = hash(key);
80          int i = 1;
81          while (keys[index] != key && keys[index] != null) {
82              index = (index + i * hash2(key)) % capacity;
83              i++;
84          }
85          return values[index];
86      }
```

4. Độ phức tạp tính toán:

Method	Unsorted List	Hash Table	
		expected	worst case
get	O(N)	O(1)	O(N)
put	O(N)	O(1)	O(N)
remove	O(N)	O(1)	O(N)
size, isEmpty	O(1)	O(1)	O(1)
entrySet, keySet, values	O(N)	O(N)	O(N)
search	O(N)	O(1)	O(N)

CHƯƠNG III: BẢNG BĂM (HASH TABLES)

Trong phần này nội dung chủ yếu về phần cài đặt bảng băm Separate Chaining và Linear Probing, những bảng băm còn lại đã được trình bày ở phần trên và cài đặt tương tự.

- Khởi tạo bảng băm
- Cài đặt các phương thức cơ bản như insert, delete, search, hash, resize và print

- Hàm băm được sử dụng là hàm lấy modulo sau khi đã gọi phương thức hashCode() mặc định của Object.

```
78      private int hash(Key key) {
79          return key.hashCode() % capacity & 0x7fffffff;
80      }
```

1. Separate Chaining:

Khởi tạo Separate Chaining:

```
1   package dsa_end.hashtables;
2
3   import java.util.*;
4
5   public class SeparateChainingHash<Key, Value> {
6
7       private class Node {···
16
17      private LinkedList<Node>[] table;
18      private int size;
19      private int capacity;
20      private int deleted;
21
22      public SeparateChainingHash(int capacity) {···
28
29      public void insert(Key key, Value value) {···
46
47      public void delete(Key key) {···
64
65      public Value search(Key key) {···
77
78      private int hash(Key key) {···
81
82      private void resize(int capacity) {···
98
99      public void print() {···
111 }
```

Phương thức insert:

```
29      public void insert(Key key, Value value) {
30          if (size + deleted == capacity) {
31              resize(capacity * 2);
32          }
33          int index = hash(key);
34          if (table[index] == null) {
35              table[index] = new LinkedList<>();
36          }
37          for (Node node : table[index]) {
38              if (node.key.equals(key)) {
39                  node.value = value;
40                  return;
41              }
42          }
43          table[index].add(new Node(key, value));
44          size++;
45      }
```

Phương thức delete:

```
47      public void delete(Key key) {
48          int index = hash(key);
49          if (table[index] == null) {
50              return;
51          }
52          for (Node node : table[index]) {
53              if (node.key.equals(key)) {
54                  table[index].remove(node);
55                  size--;
56                  deleted++;
57                  return;
58              }
59          }
60          if (size + deleted == capacity / 4) {
61              resize(capacity / 2);
62          }
63      }
```

Phương thức search:

```
65      public Value search(Key key) {
66          int index = hash(key);
67          if (table[index] == null) {
68              return null;
69          }
70          for (Node node : table[index]) {
71              if (node.key.equals(key)) {
72                  return node.value;
73              }
74          }
75          return null;
76      }
```

Từ dòng 70 đến dòng 74, phương thức tiến hành tìm kiếm tuần tự trong chuỗi có mã băm của key.

2. Linear Probing:

Cài đặt bảng băm LinearProbingHash:

```
1   package dsa_end.hashtables;
2
3   public class LinearProbingHash <Key, Value>{
4       private Key[] keys;
5       private Value[] values;
6       private int size;
7       private int capacity;
8       private int deleted;
9
10      public LinearProbingHash(int capacity) {···
17
18      public void insert(Key key, Value value) {···
30
31      public void delete(Key key) {···
45
46      public Value search(Key key) {···
53
54      private int hash(Key key) {···
57
58      private void resize(int capacity) {···
75
76      public void print() {···
82  }
```

Tương tự với các cài đặt bảng băm còn lại, các phương thức đặc trưng như insert, delete, search được thể hiện như sau:

```
18      public void insert(Key key, Value value) {
19          if (size + deleted == capacity) {
20              resize(capacity * 2);
21          }
22          int index = hash(key);
23          while (keys[index] != null) {
24              index = (index + 1) % capacity;
25          }
26          keys[index] = key;
27          values[index] = value;
28          size++;
29      }
```

```
31      public void delete(Key key) {
32          int index = hash(key);
33          while (keys[index] != key) {
34              index = (index + 1) % capacity;
35          }
36          keys[index] = null;
37          values[index] = null;
38          size--;
39          deleted++;
40
41          if (size + deleted == capacity / 4) {
42              resize(capacity / 2);
43          }
44      }
```

```
46      public Value search(Key key) {
47          int index = hash(key);
48          while (keys[index] != key && keys[index] != null) {
49              index = (index + 1) % capacity;
50          }
51          return values[index];
52      }
```

CHƯƠNG IV: ỨNG DỤNG CỦA BẢNG BĂM

1. Ứng dụng trong thực tiễn:

Với bài toán thông thường: chỉ cần sử dụng cấu trúc dữ liệu được cài đặt sẵn: map, set trong Java, C/C++,..
Với bài toán đặc thù: tự viết hàm băm và xây dựng cấu trúc dữ liệu bảng băm cho phù hợp

- Biểu diễn các đối tượng
- Lập chỉ mục cơ sở dữ liệu
- Thiết lập cơ chế Caches…

2. Ứng dụng trong giải quyết bài toán cụ thể:

Trong phần này, Hash Table được tích hợp với giao diện Map (HashMap) để giải quyết bài toán tìm kiếm từ vựng trong từ điển. Trước Java8, HashMap xử lý va chạm bằng Linked List, tuy nhiên, việc tìm kiếm trên Linked List có độ phức tạp của một hàm tuyến tính (O(n)), Java8 đã giải quyết vấn đề này bằng cách sử dụng cây cân bằng nếu phần tử của list vượt quá ngưỡng nào đó để giải quyết vấn đề này. Do đó độ phức tạp trong việc tìm kiếm chuyển từ O(n) thành O(log n).
Cho key là các từ vựng tiếng Anh và tiếng Trung, value là phiên âm và nghĩa bằng tiếng Việt được lưu trữ trong text file (có định dạng .txt). Mặc định dữ liệu được lưu trữ trong file theo mỗi dòng dưới dạng <từ vựng> : <phiên âm và nghĩa>.

```
3694    zero: number /’ziərou/ số không
3695    zone: n. /zoun/ khu vực, miền, vùng
3696    爱情: /ài qíng/ - tình yêu
3697    安排: /ān pái/ - sắp xếp
3698    安全: /ān quán/ - an toàn
```

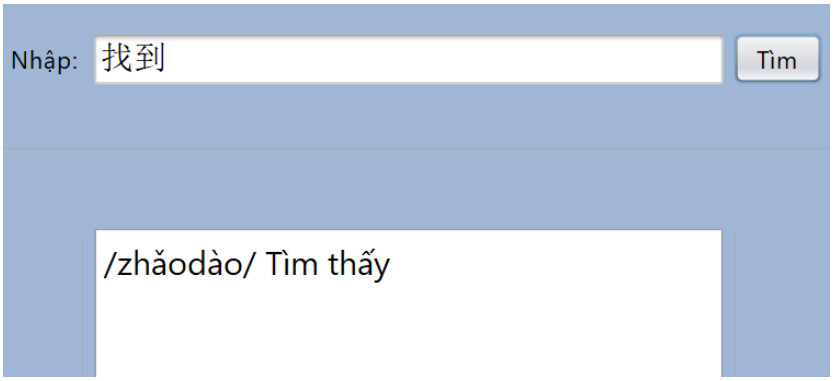
Đầu tiên, tiến hành lấy dữ liệu từ file và đưa vào bảng băm, sử dụng put(key, value) mặc định của HashMap:

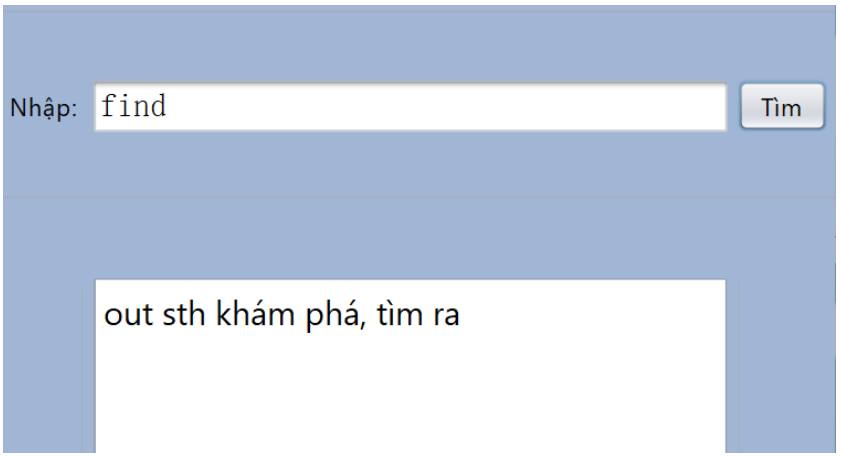
```
266     private static void getDataFromFile() {
267         try {
268             File file = new File(pathname:"test\\TestFile.txt");
269             FileReader fr = new FileReader(file);
270             BufferedReader br = new BufferedReader(fr);
271             String line;
272             while ((line = br.readLine()) != null) {
273                 String[] parts = line.split(regex:": ",limit:2);
274                 if (parts.length == 2) {
275                     String word = parts[0].trim();
276                     String meaning = parts[1].trim();
277                     words.put(word, meaning);
278                 } else {
279                     System.out.println("Invalid line format: " + line);
280                 }
281             }
282             br.close();
283             fr.close();
284         } catch (Exception e) {
285             System.out.println("Error: " + e);
286         }
287     }
```

Sau đó, tiến hành nhập từ muốn tìm kiếm. Từ cần tìm kiếm sẽ được chỉnh sửa cơ bản như chuyển về dạng chữ thường (nếu viết hoa), bỏ dấu cách ở đầu hoặc cuối. Tiến hành tìm kiếm từ bằng containKey() và đưa ra nghĩa của từ bằng get():

```
191
192     private void findWordMouseClicked(java.awt.event.MouseEvent evt) {
193         String word = findWordField.getText().trim();
194         word = word.toLowerCase();
195         String meaning = "";
196         if (words.containsKey(word)) {
197             meaning = words.get(word);
198         } else {
199             meaning = "Không tìm thấy từ này";
200         }
201         jTextArea1.setText(meaning);
202     }//GEN-LAST:event_findWordMouseClicked
203
```

Sau đây là ví dụ khi tìm kiếm từ:





KẾT LUẬN

Sau một thời gian chúng em tìm hiểu, nghiên cứu, đồng thời nhận được sự góp ý của thầy cô và các bạn, chúng em đã hoàn thiện bài báo cáo "Bảng băm và ứng dụng của bảng băm" với 4 nội dung cơ bản sau:

1.Kỹ thuật băm

2.Hàm băm

3.Bảng băm

4.Ứng dụng

Mong rằng bài báo cáo này sẽ giúp mọi người có thể nắm được những kiến thức cơ bản về bảng băm và có thể ứng dụng bảng băm để giải quyết các bài toán sau này.

Chúng em xin chân thành cảm ơn.

TÀI LIỆU THAM KHẢO

1. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and Algorithms in Java. John Wiley & Sons.
2. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). Introduction to algorithms. MIT Press. (bản dịch: Ngọc Anh Thư (2002) Giáo trình thuật toán - Lý thuyết và bài tập. Nhà xuất bản thống kê)
3. Sedgewick, R., & Wayne, K. (2011). Algorithms. Addison-Wesley Professional.