Pontifícia Universidade Católica do Rio de Janeiro
INF1010 - Estruturas de Dados Avançadas

Lucas Demarco Cambraia Lemos - 2110013
Jayme Augusto Avelino de Paiva - 2210289

Tarefa 3
Tabelas Hash

Rio de Janeiro
2023.1

1. Objetivo do projeto

O projeto tem o objetivo de implementar uma tabela *hash* para armazenar placas de automóveis no formato “CCCIIII”, ou seja 3 caracteres maiúsculos e 4 inteiros. Para o algoritmo de inclusão desenvolvido, é importante controlar as colisões, ou seja, contá-las e armazenar a placa conflitante no próximo espaço vago do vetor, representando o endereçamento interno. As placas foram fornecidas via arquivo texto e usadas para elaborar os algoritmos de inserção, exclusão e busca

2. Pseudocódigo e código fonte dos algoritmos implementados

2.1) Estrutura de Dados

Inicialmente, é importante mostrar qual foi a estrutura usada para o desenvolvimento do programa. Os campos da estrutura implementada são:

- *int ocu*: número de posições ocupadas da Hash Table
- *int col*: número de colisões
- *char** hash*: vetor de strings que representa a Hash Table

```
 9  typedef struct hashTable HashTable;
10
11  struct hashTable{
12      int col;
13      int ocu;
14      char** hash;
15  };
```

Estrutura utilizada no programa

2.1) Leitura das Placas

Função *readPlacas* (linhas: vetor de string (*char**))

Abre o arquivo *placas.txt* para leitura e armazena o ponteiro para o arquivo na variável *file*

Define uma variável inteira *i* como 0 → contador

Enquanto (houver uma linha a ser lida do arquivo)

Ler uma linha do arquivo e armazenar na variável 'line'

Usar a função *strcpy* de *string.h* para copiar o conteúdo de *line* para a posição apontada por *(*linhas*+*i*)

Adiciona um caractere nulo ('\0') na posição 7 da string apontada por *(*(linhas+i)+7)

Adiciona o valor 1 a variável *i* → contador +=1

Fecha o arquivo 'file'

Fim da função

```
17  void readPlacas(char** linhas){
18      FILE* file = fopen("placas.txt","r");
19      char line[50];
20      int i = 0;
21
22      while (fgets(line,50,file)){
23          strcpy(*(linhas+i),line);
24          *(*(linhas+i)+7) = '\0';
25          i++;
26      }
27  }
```

Código fonte da função *readPlacas*

2.2) Cálculo dos tempos de inserção

Função *calcAllTimesIns* (*hashTable*: ponteiro para a tabela, *placas*: vetor de strings, representando as placas)

Define variável *k* como 1 → contador

Enquanto (*k* < número total de placas (1000))

Insere a quantidade *k* de placas na tabela, chamando a função *calcTimesInsertion*

Fim da função

```
139 void calcAllTimesIns(HashTable* hashTable, char** placas){
140     for (int k = 1; k <= 1000; k++){
141         calcTimesInsertion(k, hashTable, placas);
142     }
143 }
```

Código fonte da função *CalcAllTimesIns*

Função *calcTimesInsertion* (*k*: inteiro, *hashTable*: ponteiro para a tabela, *placas*: vetor de strings, representando as placas)

Define variável *timeIni* → tempo inicial de execução, vem da biblioteca *time.h*

Define variável *i* como 0 → contador

Enquanto (*i* < *k*)

Insere a placa *(*placas*+*i*) na tabela, chamando a função *insertHash*

Soma o valor 1 a *i* → contador +=1

Define variável *timeFim* → tempo final de execução, vem da biblioteca *time.h*

Define variável *time* como *timeFin* - *timeIni*, em segundos

Atribui novo valor inicial para a variável *timeIni*

Define variável *i* como 0 → contador

Enquanto (*i* < *k*)

busca a placa *(*placas*+*i*) na tabela, chamando a função *buscaHash*

Soma o valor 1 a *i* → contador +=1

Define variável *timeFim* → tempo final de execução, vem da biblioteca *time.h*

Define variável *timeBus* como *timeFin* - *timeIni*, em segundos

Imprime o número de inserções (*k*) e o tempo das inserções (*time*), número de colisões (*hashTable->col*), número de buscas (*k*) e o tempo das buscas (*timeBus*)

Limpa a tabela hash *hashTable*, chamando a função *cleanList*

Fim da função

```
120 void calcTimesInsertion(int k, HashTable* hashTable, char** placas){
121     long int timeIni = clock();
122     for (int i = 0; i < k; i++){
123         insertHash(hashTable, *(placas+i));
124     }
125     long int timeFim = clock();
126     double time = (double)(timeFim - timeIni)/CLOCKS_PER_SEC;
127     //printf("%d\n", hashTable->col);
128 
129     timeIni = clock();
130     for (int i = 0; i < k; i++){
131         buscaHash(hashTable, *(placas+i));
132     }
133     timeFim = clock();
134     double timeBus = (double)(timeFim - timeIni)/CLOCKS_PER_SEC;
135     printf("%d inserções: %lfs\n%d colisões\n%d buscas: %lfs\n\n",k, time,hashTable->col,k,timeBus);
136     cleanList(hashTable);
137 }
```

Código fonte da função *calcTimesInsertion*

2.3) Inserção das placas na Tabela Hash

Função *insertHash* (*hashTable*: ponteiro para a tabela, *key*: string que representa a placa)

Define variável *inseriu* como 0

Define variável *k* como 0

Se (*hashTable->ocu* = 1031) → todos os espaços da tabela estão ocupados

Imprime mensagem de falta de memória

Retorna *hashTable->hash*

Enquanto (não inseriu e *k* < 1031)

Define a variável *pos*, sendo a posição de inserção calculada para a *key*, chamando a função *mkKey*

Se (a posição *pos* na *hashTable->hash* estiver vazia ou preenchida pela string ~~########~~)

Aloca memória para esta posição no vetor com tamanho de 50 caracteres

Copia a chave *key* para a posição *hashTable->hash*, usando *strcpy*

Atribui valor 1 para *inseriu*

Adiciona 1 a *hashTable->ocu*, representando que mais uma posição da tabela está ocupada

Senão

Adiciona 1 a *hashTable->col*, ou seja, ocorreu mais uma colisão

Retorna o ponteiro para a tabela Hash *hashTable->hash*

Fim da função

```
51  char** insertHash(HashTable* hashTable, char* key){
52      int inseriu = 0;
53      int k = 0;
54      if (hashTable->ocu == 1031){
55          printf("Error: Falta de espaço\n");
56          return hashTable->hash;
57      }
58      while(!inseriu && k < 1031){
59          int pos = mkKey(key, k);
60          if (*(hashTable->hash + pos) == NULL || !strcmp(*(hashTable->hash + pos),"#######")){
61              *(hashTable->hash+pos) = (char*)(malloc(sizeof(char)*50));
62              strcpy(*(hashTable->hash + pos), key);
63              inseriu = 1;
64              hashTable->ocu++;
65          }else{
66              hashTable->col++;
67          }
68          k++;
69      }
70      return hashTable->hash;
71  }
```

Código fonte da função *insertHash*

2.4) Busca das placas na Tabela Hash

Função *buscaHash* (*hashTable*: ponteiro para a tabela, *value:* string com a placa)

Define variável *tentativa* como 0

Enquanto (tentativa < 1031) → percorreu toda a tabela

Define a variável *key* como o retorno da função *mkKey*, para a placa *value*, representando a chave calculada para essa placa

Se (posição *key* da Tabela Hash for *NULL*)

Exibe mensagem que o elemento não está na tabela

Retorna *NULL*

Se (string na posição *key* da tabela foi igual a *value*) → encontrou a chave

Retorna um ponteiro para essa posição

Adiciona 1 a *tentativa*

Exibe mensagem que o elemento não está na tabela

Retorna *NULL*

Fim da função

```
74  char* buscaHash(HashTable* hashTable, char* value){
75      int key;
76      int tentativa = 0;
77      while (tentativa < 1031){
78          key = mkKey(value, tentativa);
79          if (hashTable->hash[key] == NULL){
80              printf("O elemento não está na lista\n");
81              return NULL;
82          }
83
84          if (!strcmp(hashTable->hash[key],value))
85              return hashTable->hash[key];
86          tentativa++;
87      }
88      printf("O elemento não está na lista\n");
89      return NULL;
90  }
```

Código fonte da função *buscaHash*

2.5) Limpeza dos elementos da tabela

Função *cleanList* (*hashTable*: ponteiro para a tabela)

Define variável *i* como 0

Enquanto (*i* < tamanho da tabela (1031))

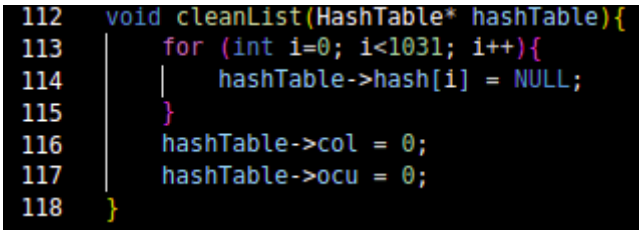
Atribui o valor NULL para cada posição *i* da tabela

Adiciona 1 a *i*

Atribui o valor 0 para o campo *ocu* da *hashTable*

Atribui o valor 0 para o campo *col* da *hashTable*

Fim da função



```
112  void cleanList(HashTable* hashTable){
113      for (int i=0; i<1031; i++){
114          hashTable->hash[i] = NULL;
115      }
116      hashTable->col = 0;
117      hashTable->ocu = 0;
118  }
```

Código fonte da função *cleanList*

2.6) Limpeza dos elementos da tabela

Função *mkKey* (*placa*: string com a placa, *k*: número da tentativa de inserção)

Define variável *king* como 0, representando a soma do valor dos bytes da placa

Define variável *val2* como 0

Define variável *p2*, ponteiro para *val2*

Define variável *val* como 0

Define variável *p*, ponteiro para *val*

Define variável *i* como 0

Enquanto (*i* < quantidade de caracteres da placa (7))

Define variável *uc* como o valor do byte da placa na posição *i*

Atribui *uc* ao byte da posição *i* de *val*

Atribui *uc* ao byte de posição 7 - *i* de *val2*

Adiciona *uc* a *king*

Adiciona 1 a *i*

Define variável *newVal* como *val***val2*

Define variável *key* como o resto da divisão de *newVal* + *val2***k* + *val***k***k* + *k* por 1031

Se (resto da divisão de *king* por 89 for igual a 1)

Atribui a *key* o valor de -*key*

Se (*key* menor que 0)

Atribui a *key* o valor da soma de *key* por 1031

Retorna *key*

Fim da função

```
29  int mkKey(char* placa, int k){
30      unsigned long king = 0;
31      unsigned long val2 = 0;
32      unsigned char* p2 = (unsigned char*)(&val2);
33      unsigned long val = 0;
34      unsigned char* p = (unsigned char*)(&val);
35      for (int i = 0; i<7; i++){
36          unsigned char uc = *(placa+i);
37          *(p+i) = uc;
38          *(p2+7-i) = uc;
39          king += uc;
40      }
41      unsigned long newVal = val*val2;
42      int key = (int)(newVal + (val2)*k + (val)*k*k + k)%(1031);
43      key = (key)%1031;
44      if (king%89 == 1)
45          key = key*(-1);
46      key = key < 0 ? key + 1031 : key;
47      return key;
48  }
```

Código fonte da função *mkKey*

2.7) Apaga a Tabela Hash

Função *deleteHash* (*hashTable*: ponteiro para a tabela, *value:* string com a placa)

Define a variável *tentativa* com o valor 0

Enquanto (*tentativa* < o tamanho total da tabela (1031))

Define *key* como a variável que guarda a chave calculada pela função *mkKey*, para a placa *value*

Se (posição *key* da tabela aponta para *NULL*)

Exibe mensagem que o elemento não está na tabela

Retorna um ponteiro para a Tabela Hash

Se (string na posição *key* da tabela foi igual a *value*) → encontrou a chave

Copia a string ####### para a posição *key* da tabela

Subtrai 1 de *hashTable->ocu*, pois esta posição não está mais ocupada

Retorna um ponteiro para a Tabela Hash

Adiciona 1 a *tentativa*

Exibe mensagem que o elemento não está na tabela

Retorna um ponteiro para a Tabela Hash

Fim da função

```
92  char** deleteHash(HashTable* hashTable, char* value){
93      int key;
94      int tentativa = 0;
95      while (tentativa < 1031){
96          key = mkKey(value, tentativa);
97          if (hashTable->hash[key] == NULL){
98              printf("O elemento não está na lista\n");
99              return hashTable->hash;
100         }
101         if (!strcmp(hashTable->hash[key],value)){
102             strcpy(hashTable->hash[key],"#######");
103             hashTable->ocu--;
104             return hashTable->hash;
105         }
106         tentativa++;
107     }
108     printf("O elemento não está na lista");
109     return hashTable->hash;
110 }
```

Código fonte da função *deleteHash*

3. Resultados e conclusões

Para avaliar a eficiência da função *hash* elaborada, foram elaborados dois gráficos para visualizar as saídas do programa. Em ambos os casos, o tamanho de entrada variou de 1 até 1000, para obter um gráfico com a maior quantidade de informações possíveis. A visualização abaixo mostra o tamanho de entrada pelo tempo de execução de inserção de cada um.

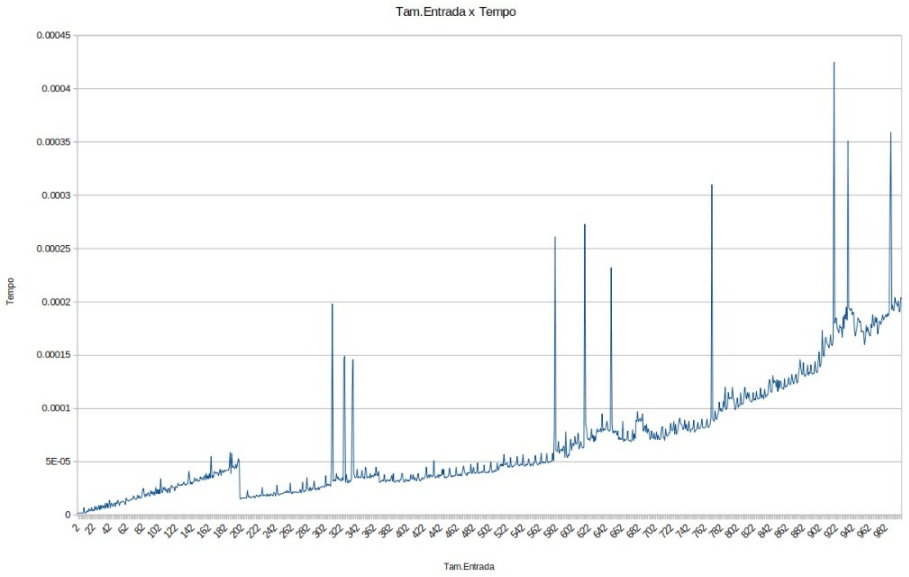


Gráfico Tempo de inserção x Tamanho de Entrada

O outro gráfico mostra a relação entre o tamanho de entrada, variando de 1 até 1000, e a quantidade de colisões do programa.

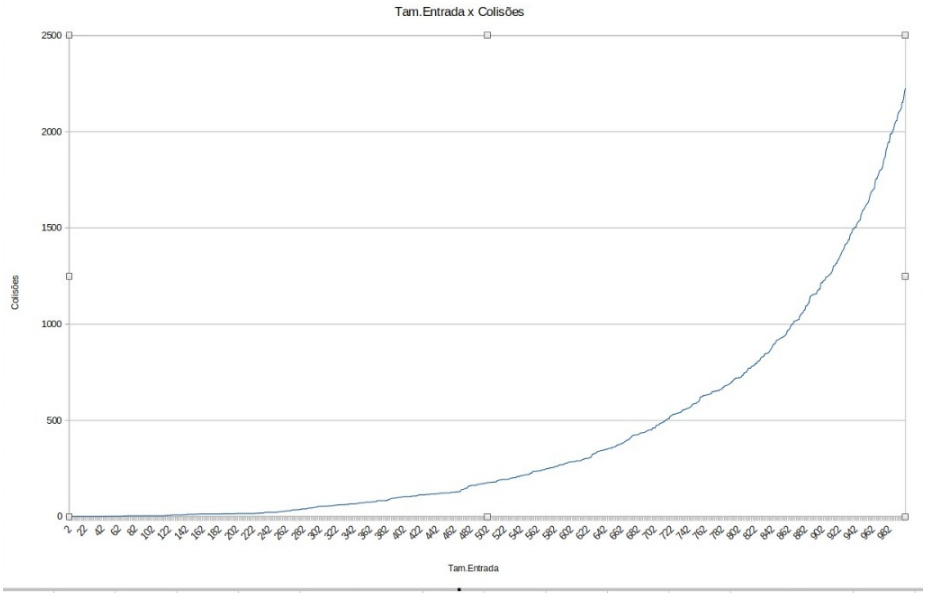


Gráfico Colisões x Tamanho de Entrada

Além disso, o programa emite uma saída com os resultados observados no programa. Neste relatório, observa-se que o tempo total para 1000 inserções é de 0,000623s, enquanto o tempo de total para a busca das 1000 chaves é 0,000587s. Além da visualização disponibilizada pelo gráfico Colisões x Tamanho de Entrada, a saída do programa mostra que a quantidade de colisões para a inserção das 1000 placas é 2225.

O programa exibe um relatório do tamanho de entrada (variando de 10 em 10, de 1 até 1000), o número de colisões e os tempos de inserção e busca. Na tabela abaixo, é possível visualizar esses dados resumidos, com os tamanhos de entrada variando de 100 em 100.

Tamanho de entrada	**Colisões**	**Tempo de inserção**	**Tempo de busca**
100	4	0.000026s	0.000017s
200	16	0.000011s	0.000008s
300	53	0.000017s	0.000013s
400	104	0.000023s	0.000018s
500	176	0.000033s	0.000025s

600	284	0.000041s	0.000036s
700	461	0.000054s	0.000047s
800	721	0.000070s	0.000063s
900	1216	0.000095s	0.000086s
1000	2225	0.000142s	0.000134s