TAREFA 3 – EAD

SINGULAR:

Nome: Enzo Tres Mediano

Matrícula: 2211731

//ENUNCIADO:

Estruturas de Dados – Trabalho sobre Tabelas Hash – 2023_1 Implementar algoritmos de busca, inclusão e exclusão em vetor de X posições (onde X é o maior número primo após 1024, ou 210), usando hashing com tratamento de colisão para armazenar placas de automóveis no formato “CCCNNNN” onde C é um caractere maiúsculo e N um numeral. A rotina de tratamento de colisão deve armazenar a nova placa no próprio vetor (endereçamento interno). Não devem ser usadas outras estruturas de dados. A função hash deve ser definida individualmente e utilizar endereçamento aberto com tentativa linear, quadrática ou dispersão dupla.

Elaborar programa contendo a implementação dos algoritmos de inserção, exclusão e busca, em linguagem C. Os algoritmos devem prever uma entrada de dados via arquivo texto contendo placas de automóveis geradas aleatoriamente (em cada linha do arquivo texto deverá vir uma placa seguida pelos caracteres). O programa deve prever ainda a geração de um relatório indicando o total de colisões geradas e os tempos de inclusão e de busca a todos os elementos do vetor (a busca deve ser feita após o término da inclusão de todas as placas).

Deverá ser entregue ainda um relatório contendo o pseudocódigo dos algoritmos utilizados, os programas fonte, os tempos tomados para cada algoritmo e a geração do gráfico de tamanho da entrada X tempo de execução de cada um, para diferentes valores de entrada por exemplo: 128, 256 e 512 placas (complexidade prática).

Todos os aspectos considerados deverão ser avaliados.

A avaliação vai ser feita em um mesmo equipamento para todos os grupos. A maior nota caberá ao grupo que fizer o trabalho completo, com menor número de colisões e menor tempo de execução dos algoritmos. Para a avaliação o professor fornecerá um arquivo de placas geradas aleatoriamente para servir de comparação entre os trabalhos (número de colisões e tempo de execução).

//PseudoCódigo dos Funcoes

```
int insertHash(HashTable* HashTable, char* liscensePlate);
```

inicializar valor

hashNum

escumando

inserir valores

-> Funcao para inserir a placa no hash correta incluindo se tem colisao e para verificar se o hashtable precisava ser maior.

```
unsigned int hash(char* plate, int sizeHT);
```

definir valor inicial

Forloop

hashfuncao

retorno

-> Funcao do hash para achar qual hash a placa deveria ser colocado.

```
void initHashTable(HashTable* ht, int size);
```

malloc

definir valores

definir placa e check valores em loop

-> Inicializa o HashTable;

```
int Search(HashTable* ht, char* lic);
```

hashNum

ForLoop

procurar

comparar strings

retorne se verdadeiro

procure se flase

-> A função retornaria o hash de onde a placa do carro foi encontrada

```
void deletePlate(PlateList* Hashlist, char* liscensePlate);
```

hashNum

ForLoop

procurar

comparar strings

definir valores se verdadeiro

procure se falso

-> Um funcao que deletava a placa usando hash para achar o lugar necessário e checando cada Node dentro daquele hash

```
char** loadLicensePlates(sizeOfHashTable);
```

criar variável de arquivo

ler arquivo

criar array de strings

ForLoop

arquivo fscanf até \n

definir valor fscanf na array

fechar arquivo

retorno array

-> Esse funcao abria o arquivo criado pelo “createFile” e colocava cada placa dentro de um array de placas para ser usado com mais eficiência pela programa.

```
void freeHashTable(HashTable* HashList);
```

ForLoop

free placa

free matriz

-> Esse funcao dava o free do HashTable todo.

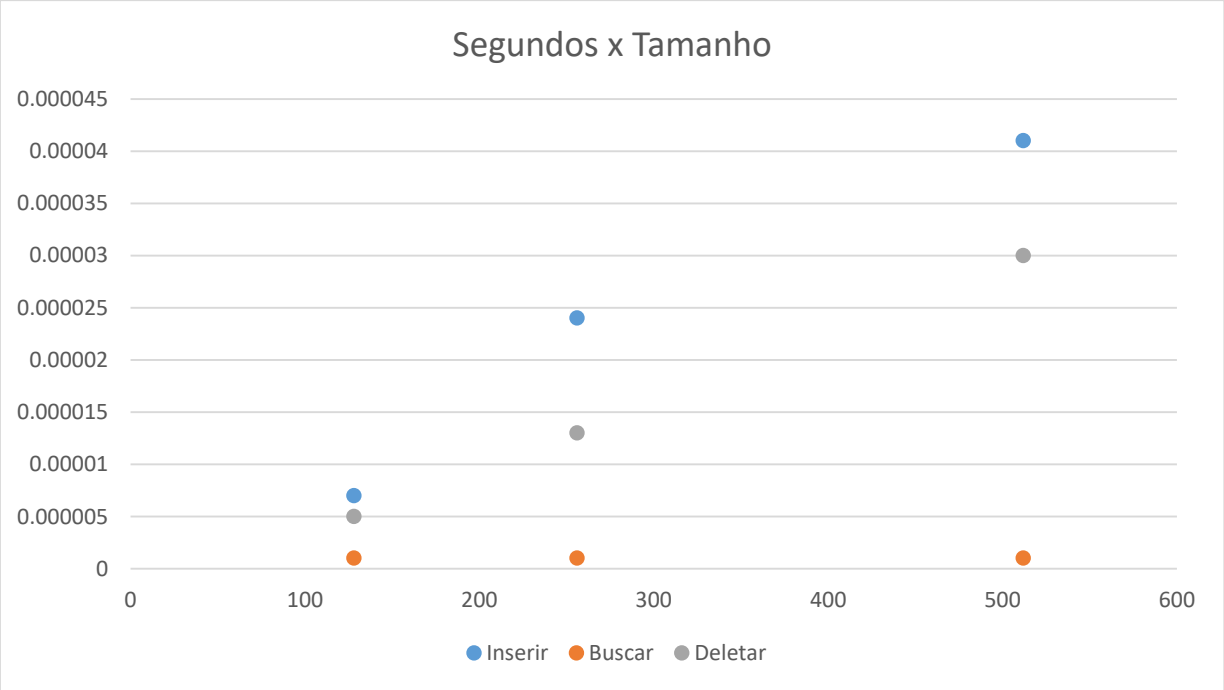
//Solução

- **Inicialização do HashTable**
  - Feito uma estrutura do tipo (Hashtable) que continha a quantidade máxima de espaços, a quantidade atual de espaços sendo usados e uma estrutura do tipo Placa* que continha a "string" para placas de carro e um "cheque" para ver se tinha uma placa de licença lá antes, com 0 sendo nunca e 1 sendo uma placa de licença, mas foi excluída. Dentro da função initHashTable, eu alocaria dinamicamente ht->HashArray e definiria os valores do inicializador para currSize e maxSize, ele também teria initPlate que inicializa o cheque e a placa do carro.
- **Extraindo o informação do arquivo**
  - Tem um funcao chamado loadLicensePlates que abria o “licensePlates.txt” arquivo. Dai eu aloquei espaço para ter um vetor de strings de tamanho “numPlacas.” Mais sobre numPlacas no “Determinando o quantidade de placas terão.”
- **Inserindo placas e medindo quantidade de colisões**
  - Para inserir existe um loop para que possamos inserir todas as placas de veículos na HashTable. Para inserir a função faria uma variável chamada hashNum que seria igual ao número retornado da função hash. Em seguida, a função verificaria se o hash gerado já possui uma placa no lugar, caso afirmativo, pula para o próximo hash usando `hashedNum = (hashedNum + 1) % new_ht->maxSize;` (Faria que se era no final hash ele ia procurar na primeira hash do HashTable). Caso contrário, ele faz um strdup para alocar e copiar uma nova string no lugar do hash. Ao longo da função, haveria uma variável de contagem que seria atualizada sempre que houvesse uma placa já dentro de um hash. Essa contagem seria então retornada e usada para imprimir a quantidade de colisões dentro do teste.
- **Buscando placas**
  - Para procurar a placa, usaríamos um loop na função principal após a conclusão da inserção que garantiria que todas as placas estivessem sendo usadas na função de pesquisa, uma a uma. Dentro da função "Search", faria o hash da placa e verificaria se estava no hash. Em seguida, usaria `hashedNum = (hashedNum + 1) % ht->maxSize;` para ir para o próximo hash se o string no hash tem o valor de NULL. Isso se repete até que o hash tenha uma placa que não seja NULL. Quando isso acontecer, ele verificará se é a mesma placa. Se sim, ele retorna o número do hash, caso contrário, ele vai para o próximo hash usando o mesmo método de antes.
- **Deletando placas**
  - Para excluir as placas, um loop foi novamente usado para garantir que todas as placas fossem removidas da HashTable. “deletePlate” é bastante semelhante a "Search" em como encontra o hash que possui a placa. Mas, em vez de retornar o número do hash, ele liberaria a placa, definiria a placa como nula e definiria a verificação como 1 para avisar no futuro "Search" e "deletePlate" que costumava haver uma placa naquele hash.
- **Achando os tempos**
  - Incialmente na linha 47 eu criei dois variáveis start e end de forma clock_t. clock_t veio do #include <time.h>. Linha 28 foi criado um cpuTimeUsed para pegava um tempo do computador usando a seguinte formula `cpuTimeUsed = ((double)(end - start)) / CLOCKS_PER_SEC;`. Esse pegava a difference em tempos do start e end e dividia pelo tempo do cpu por segundo para pegar tempo puro. Dai com os três testes

eu pegava o valor do clock no start rodava o código do insertHash e depois do loop de insertHash peguei o valor do clock no end. E usando cpuTimeUsed dito antes nesse paragrapho, printava o cpuTimeUsed dentro de um printf. A mesma processo foi feito por a Search(busca) e deletePlate(deletar). (Aviso por quem testaria o código, os tempos so iriao ser gerados quando usando o site replit).

- **Free no HashTable**
  - Para liberar o HashTable, "freeHashTable" foi usado para obter o ponteiro para Hashtable e primeiro faz uma verificação NULL. Ele então fará um "forloop" usando o "maxSize" (tamanho do HashTable) para liberar cada string de placas (neste caso de atribuições tudo já deveria ser NULL mas isso estaria lá como um padrão profissional se alguém fosse inserir novos placas e não excluí-las). Depois disso, liberaria toda a HashTable.

//Grafico Dos Tempos



// Conclusão

- Foi muito difícil interpretar o que fazer se tivéssemos uma colisão no último hash da HashTable. Depois de ficar preso nisso por algum tempo, um método foi desenvolvido para onde, se tivéssemos uma colisão no último Hash, o próximo Hash a ser observado seria o início da HashTable (como um círculo, em vez de uma linha reta com e fim). Devido à rapidez com que o programa seria executado, acabaria obtendo 0,00000 como meu relógio para quando executasse os testes. Então tive que usar o replit que ai dava os tempos dados no gráfico. Os pontos laranja no gráfico representando buscar podem parecer estranhos, mas está correto,

pois há testes e verificações usando prints nas linhas 171 e 165 (ambas com // para não atrapalhar quem estiver fazendo o teste com o código) . Se quem estiver testando tiver dúvidas pode retirar o // e ver que quando houver uma colisão dentro da busca, ele imprimirá de fato o que está dentro do printf. Desenvolver a função Hash também provou ser um desafio, mas depois de estudar as funções Hash e por meio de tentativa e erro bruto, uma função Hash suficiente foi desenvolvida para que as colisões não se tornassem um número grande. No geral, esta tarefa foi uma ótima maneira de desenvolver minhas habilidades pessoais de hash.