Relatório descritivo

A primeira vista se tratava de um problema complexo, com várias ‘frentes’ a serem abordadas, mas após desmembrar a tarefa em algumas partes se tornou muito mais simples a compreensão.

A primeira parte do problema era realizar a leitura dos 100.000 nomes contidos no arquivo .txt, por conta de estarem contidos cada um deles em uma linha apenas o método de leitura não podia ser outro, a meu ver, a não ser ler a linha por linha, até encontrar um ‘\n’, e realizar a inserção na lista, como mostra a imagem abaixo.

```
    FILE *in_file = fopen(filename, "r");

    struct stat sb;
    stat(filename, &sb);
    int aux;
    do
    {
        char *file_contents = malloc(sizeof(char) * 25);
        fscanf(in_file, "%[^\n] ", file_contents);
        funcaoHash(hash, file_contents);
        aux = file_contents[0];
    } while (aux != 0);
```

Após solucionar como seria feito o primeiro passo, a leitura dos nomes, chegou a vez de criar a estrutura que armazenaria os mesmos. Foi criada uma struct ListaHash, que possui o comportamento de uma lista encadeada dupla:

```
typedef struct sListaHash
{
    ElementoHash *head, *tail;
    int size;
} ListaHash;
```

contendo dentro dela a struct ElementoHash, que foi a parte mais dificil de idealizar deste passo, pois se comporta ao mesmo tempo como um elemento e uma lista encadeada dupla:

```
typedef struct sElementoHash
{
    struct sElementoHash *next;
    struct sElementoHash *prev;
    int chave;
    Elemento *head, *tail;
    int size;
} ElementoHash;
```

E, por sua vez, contém elementos, estes que possuem um ponteiro para o nome que foi lido e alocado durante o primeiro passo:

```
typedef struct sElemento
{
    struct sElemento *next;
    struct sElemento *prev;
    char *data;
} Elemento;
```

Todas as funções, para inserir elementos, alocar memória, liberar memória, excluir elementos, buscar valores, e listar os elementos já tinham sido implementados em trabalhos passados e foram apenas reutilizados e adaptados dentro do projeto, concluindo assim o segundo passo, que era uma forma de armazenar estes nomes numa estrutura não vetorial.

O terceiro passo era a indexação Hash para pulverizar os elementos em diferentes ElementosHash, passo esse que passou por um processo de tentativa e erro para tentar deixar as listas o mais balanceadas possível. utilizando a função:

```
int calcularHash(char *nome){
    int valor = 0;
    for (int i = 0; i < strlen(nome); i++)
    {

        valor += nome[i] * 91;

    }

    return valor % TAM;
}
```

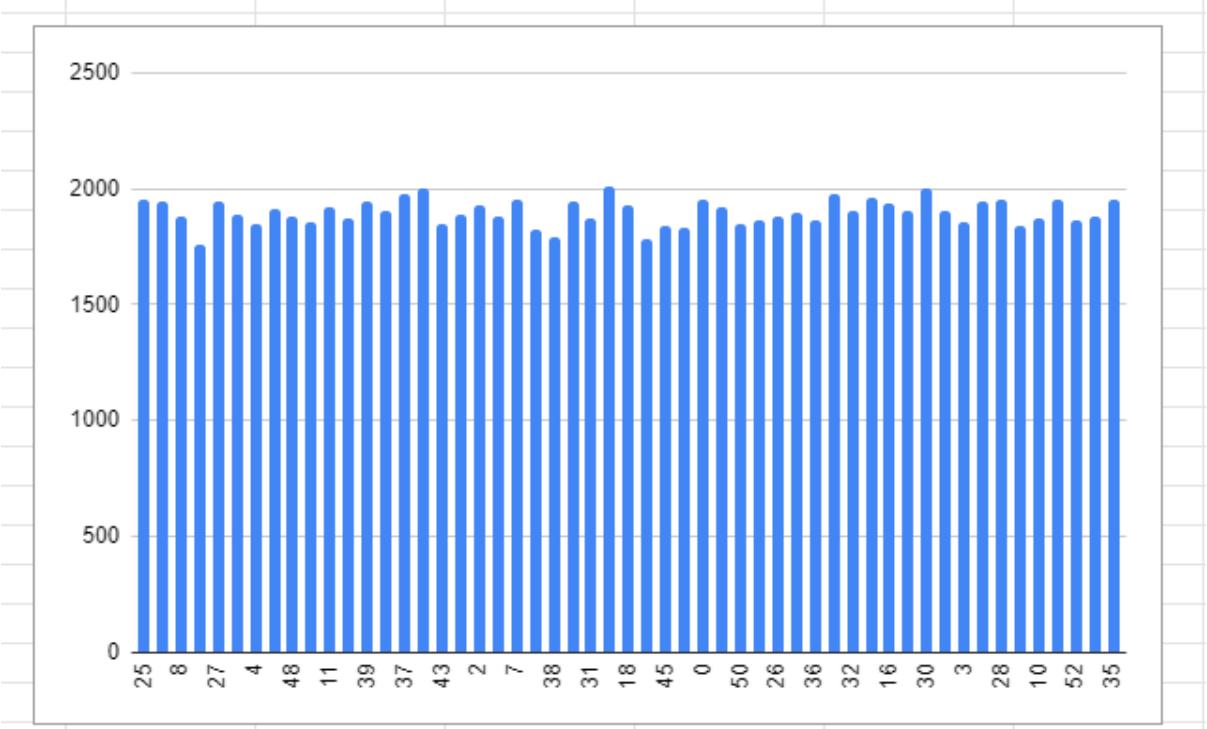
para criar uma chave para cada nome, e a função:

```
void funcaoHash(ListaHash *hash, char *nome)
{

    int chave = calcularHash(nome);
    ElementoHash *elementoHash;
    elementoHash = buscarElementoHash(hash, chave);
    if (elementoHash == NULL)
    {
        insereElementoHash(hash, chave, hash->tail);
        elementoHash = hash->tail;
    }

    insereElemento(elementoHash, nome, elementoHash->tail);
}
```

para criar e/ou inserir o nome na chave adequada o terceiro passo foi finalizado, criando o seguinte cenário:



Media	1902
Maior	2008
Menor	1761
Desvio Padrão	54,972

onde o desvio padrão equivale a aproximadamente 2,9% da média de elementos contidos dentro de cada chave, não sendo o cenário ideal, onde todas possuem o mesmo número de elementos, mas chegando bem próximo disso.

E chegamos no quarto passo, ordenar os elementos por ordem alfabética dentro de suas chaves. para isso foi criado tres funções, a primeira:

```
void ordenar(ListaHash *hash)
{
    ElementoHash *ordenar;
    for(int i = 0; i<53; i++){
        ordenar = buscarElementoHash(hash, i);
        if (ordenar != NULL)
        {
            // printf("\n %i \n", i);
            quickSort(ordenar, ordenar->head, ordenar->tail);
            // percorreElementoHash(ordenar);
        }

    }
}
```

que tinha um papel de ‘função pai’, percorrendo a ListaHash e chamando o quickSort para cada ElementoHash.

```
void quickSort(ElementoHash *li, Elemento *inicio, Elemento *fim)
{
  if (inicio != fim && inicio != NULL && fim != NULL && inicio != fim->next)
  {

    // find the pivot element such that
    // elements smaller than pivot are on left of pivot
    // elements greater than pivot are on right of pivot
    Elemento *pi = partition(inicio, fim);

    // recursive call on the left of pivot
    quickSort(li, inicio, pi->prev);

    // recursive call on the right of pivot
    quickSort(li, pi->next, fim);
  }
}
```

A quickSort por sua vez chamava a partition e ela mesma para as duas ‘sublistas’ que eram criadas pelo algoritmo.

```c
// function to find the partition position
Elemento *partition(Elemento *inicio, Elemento *final)
{

  // select the rightmost element as pivot
  Elemento *pivot = final;

  // pointer for greater element
  Elemento *i = inicio;
  Elemento *j = inicio;

  // traverse each element of the array
  // compare them with the pivot
  while (j != pivot)
  {
    if(strcmp(j->data, pivot->data) < 0){
      swap(i, j);
      i = i->next;
    }
    j = j->next;
  }

  // swap the pivot element with the greater element at i
  swap(i, pivot);


  // return the partition point
  return (i);
}
```

E a partition, que realizava a ordenação.

A parte de interface e outras miscelâneas estão contidas no código também, mas acho que fugiriam da intenção deste relatório, caso surja alguma dúvida estou à disposição para ser consultado.