Relatório do Hashing e Quicksort

Aluno: William Dutra Ribeiro

O trabalho proposto foi elaborar um trabalho de separação e ordenação para aumento na eficiência de busca e pesquisa dos elementos, para isso foi utilizado o método hashing para separação e o quicksort para ordenação. Para armazenar os valores foi utilizado um vetor de 53 casas de lista encadeada dupla, com o intuito de facilitar a referência pelo hashing e o motivo de ser lista encadeada dupla foi para não haver uma limitação na quantidade de elementos, removendo assim possíveis problemas de colisões futuras.

A função de Hashing é responsável por pegar um nome/palavra, processá-la e definir em qual index ele deve entrar, no meu caso a melhor função de hashing que encontrei foi somar o valor decimal de cada caractere na tabela ASCII e ir multiplicando pela casa desse caractere, no final após a soma realizar de todos os caracteres realizar o módulo por 53, esse valor retornado será o index.

```
int indexHashing(char * palavra){
    int soma = 0;

    for(int i = 0; i < strlen(palavra); i++){
        soma += palavra[i] * (i+1);
    }

    soma %= TAM;

    return soma;
}
```

Uma função auxiliar ao hashing interessante de se pontuar é o Histograma, ele tem a função de verificar se a distribuição dos elementos entre os array foi bem disposta, ele

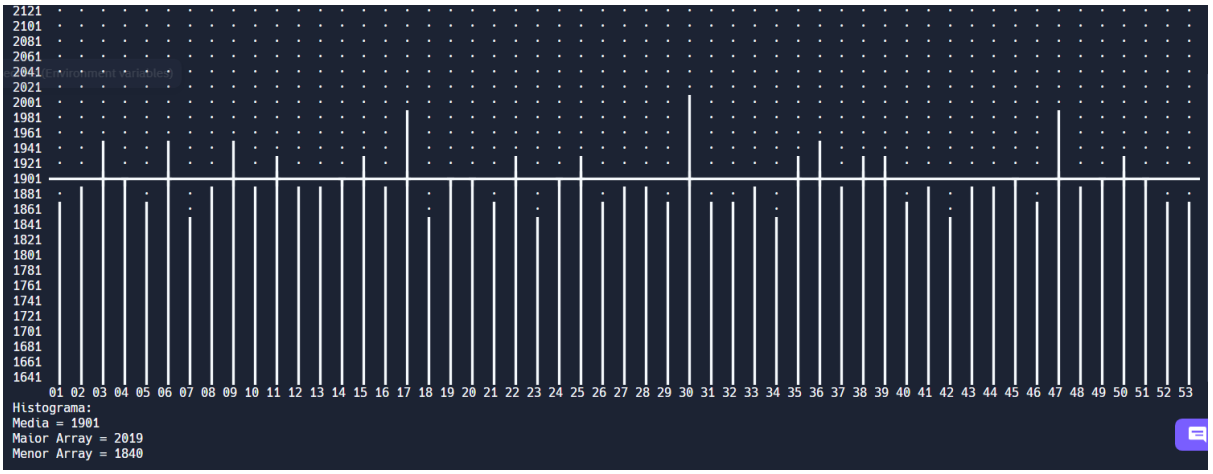
apenas verifica qual foi a média das distribuições, o maior e o menor valor.

```
int * somaMedia(Lista ** lista, int size, int * valores){

    int media = 0;
    for(int i = 0; i < size; i++){
        media += lista[i] -> size;
    }
    media /= size;

    int menorValor = media;
    int maiorValor = 0;

    for(int i = 0; i < size; i++){
        if(lista[i] -> size < menorValor){
            menorValor = lista[i] -> size;
        }

        if(lista[i] -> size > maiorValor){
            maiorValor = lista[i] -> size;
        }
    }

    valores[0] = media;
    valores[1] = maiorValor;
    valores[2] = menorValor;

    return valores;
}
```

E a função Histograma que faz um gráfico no próprio console, sobre a quantidade dos elementos em cada casa do vetor.

```c
void histograma(Lista ** lista, int size){
    int temp[3];
    int * valores = somaMedia(lista, size, temp);

    int media = valores[0];
    int maiorValor = valores[1];
    int menorValor = valores[2];


    for(int i = maiorValor/20 + 6; i >= menorValor/20 - 10; i--){

        if(i == media / 20){
            printf("%4i ", media);
        } else {
            printf("%4i ", media + (i - (media / 20)) * 20);
        }


        for(int j = 0; j < size; j++){

            if(i == media / 20 && i == lista[j] -> size / 20){
                printf("┬┴");
            }else if(i == media / 20 && i <= lista[j] -> size / 20){
                printf("┼┴");
            }else if(i == media / 20){
                printf("──");
            }else if(i <= lista[j] -> size / 20){
                printf(" │ ");
            }else{
                printf(" · ");
            }

        }

        printf("\n");
    }

    printf("     ");

    for(int i = 1; i <= size; i ++){
        if(i <= 9){
            printf("0%i ",i);
        }else{
            printf("%i ",i);
        }
    }

    printf("\nHistograma:\nMedia = %i\nMaior Array = %i\nMenor Array = %i\n", media, maiorValor, menorValor);
}
```

Resulta:



O QuickSort teve a finalidade de pegar cada uma das 53 casas do vetor de lista encadeada e as ordená-las por ordem alfabética. O QuickSort é dividido em 3 principais partes, sendo elas:
Swap(Trocar) - função responsável por trocar dois elementos dentro da lista encadeada dupla.

```
void troca(Lista * lista, Node * A, Node * B){

    char bkp[30];
    strcpy(bkp, A -> dado);
    strcpy(A -> dado, B -> dado);
    strcpy(B -> dado, bkp);

}
```

Partition - função responsável por separar os valores por um Pivô (No meu caso o Tail) e colocar os valores menores a ele à sua esquerda e os maiores à sua direita.

```
Node* partition(Lista * lista, Node * left, Node * pivo){
    Node* aux = left -> back;

    for (Node *i = left; i != pivo; i = i -> next){
        if (strcmp(i -> dado, pivo -> dado) < 0){

            if(aux == NULL){
                aux = left;
            } else {
                aux = aux -> next;
            }

            troca(lista, aux, i);
        }
    }

    if(aux == NULL){
        aux = left;
    } else {
        aux = aux -> next;
    }
    troca(lista, aux, pivo);
    return aux;
}
```

E o QuickSort - função que primeiro verifica os elementos left e right, caso não seja igual ou existam então chama função partição para pegar um pivô e ordenar os valores abaixo e acima desse número, após isso se chama duas vezes uma para os valores maiores que o pivô e outra para os menores assim gerando um ciclo.

```
void _quickSort(Lista * lista, Node* left, Node * right){
    if (right != NULL && left != NULL && left != right && left != right -> next){

        Node *pivo = partition(lista, left, right);
        _quickSort(lista, left , pivo -> back);
        _quickSort(lista, pivo -> next, right);
    }
}
```

Extra:

Teste de eficiência da função de Hashing - Para testar se a função de hashing faz uma diferença utilizar uma função onde realiza um cálculo baseado no tempo atual, onde

salva o tempo exato antes da operação e após a operação pega o tempo atual pós execução e subtrai o valor de antes da execução, isso resulta no tempo de execução.

```
t = clock();
node = pesquisaNaLista(listaArr[indexHashing(nome)], nome);
t = clock() - t;

printf("\nTempo para execução: %lf ms\n", ((double)t)/((CLOCKS_PER_SEC/1000)));
```

Testes de tempo com a palavra ZANIA e as listas ordenadas.

Primeiro teste foi com a função Hashing em funcionamento normal, os tempos foram:

1 - 0.289 ms;
2 - 0.257 ms;
3 - 0.348 ms;
4 - 0.281 ms;
5 - 0.316 ms.
Média = 0.2982 ms.

No segundo teste foi forçado um hashing, ou seja toda inserção, pesquisa ou exclusão foi no hashing 1, os tempos foram:

1 - 1.316 ms;
2 - 1.334 ms;
3 - 1.314 ms;
4 - 1.341 ms;
5 - 1.315 ms.
Média = 1.324 ms.

Como podemos notar apenas pela utilização da função hashing tivemos uma alteração grandiosa entre os valores, o segundo teste foi aproximadamente 443% maior que o primeiro, oque nos prova a eficiência absurda que tem na utilização da função de hashing.