Com o objetivo de explorar o comportamento de uma grande quantidade de dados através de uma estrutura de dados, foi implementado nesse projeto uma tabela hash capaz de ler um arquivo txt com 100.000 nomes cujo objetivo final é testar a capacidade dessa estrutura de dados, desde performance de busca até nível de espalhamento dentro dos buckets da tabela.

O grande volume de dados fez com que a possibilidade de colisão aumentasse consideravelmente, afim de contornar isso foi utilizado a técnica de chaining, dentro do código isso foi implementado utilizando uma Struct Hash onde cada posição dessa lista encadeada levava a outra Struct Bucket, essa por sua vez contendo os nodos que levam as informações e que serão adicionados gradualmente conforme demanda do sistema. Isso justifica a utilização de listas encadeadas duplas para desenvolver a tabela hash.

Para que o tratamento de colisão estivesse completo foi necessário implementar uma funçao hashing para identificar em qual bucket cada nome deveria ser armazenado, portanto como se tratavam de strings o cálculo utilizou como base o fundamento de ASCII que trata a conversão de strings para inteiros, a função hashing também tem um papel muito importante pois é por meio dela que vamos espalhar de forma uniforme os dados entre os buckets:

```c
int hashFunction(char *name){
  int key = 0;
  for (int i=0; i<strlen(name); i++){
    key+= 5 * name[i];
  }
  return (key % 53);
}
```

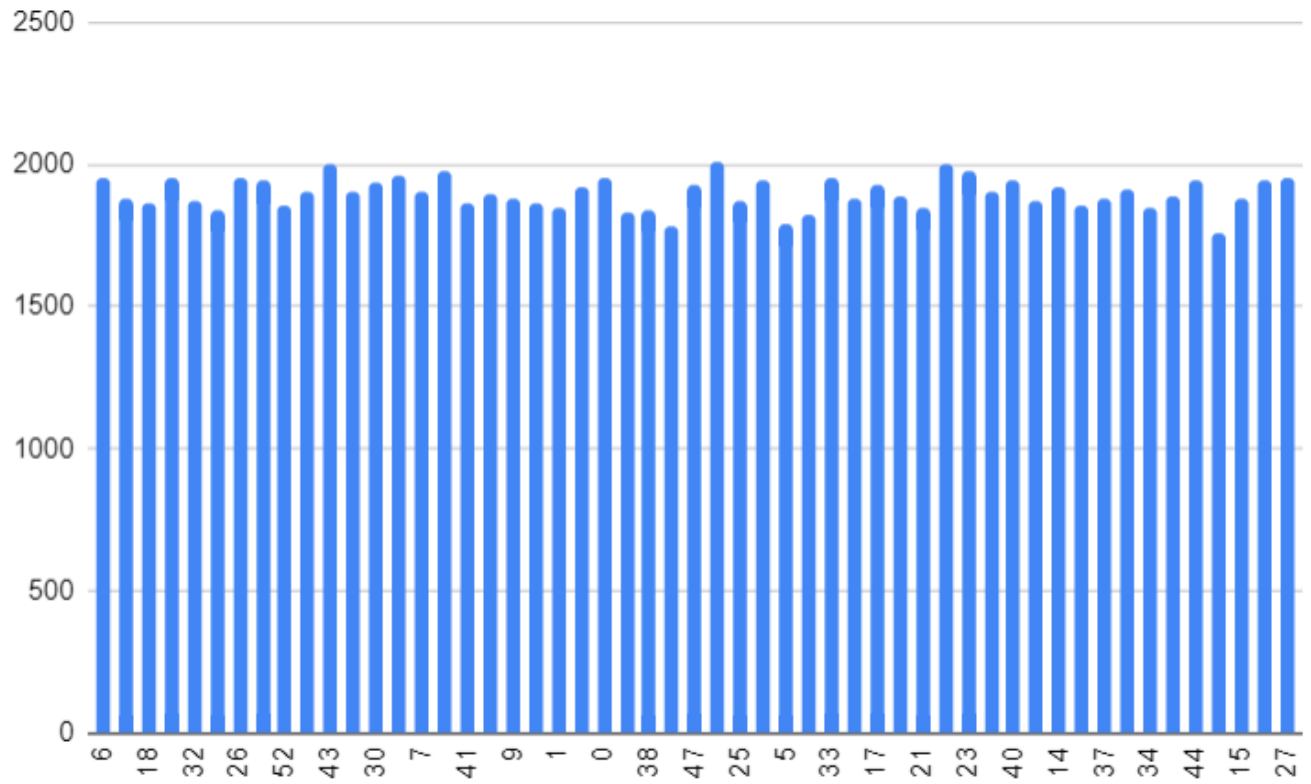
Mais detalhadamente, na função hashing o nome recebido foi iterado utilizando a técnica de hashing modular, isso permite realizar um cálculo baseado nos caracteres do nome, o multiplicador 5 foi utilizado por conta de ser um número baixo e um número primo evitando que exista algum overflow, e para finalizar o retorno do valor da chave, pegamos o resto da divisão entre a iteração e o número total de buckets, nesse caso 53.

Ao vencer os desafios de resolução de colisão e de espalhamento, foi hora de adicionar as funções que preenchem a tabela hash e que permitissem realizar consultas, remoções, etc.

Após realizar a inserção dos mais de 100.000 nomes na tabela hash, conseguimos observar a seguinte distribuição através dos buckets:

Chave	Quantidade
6	1956
31	1883
18	1866
3	1951
32	1874
4	1841
26	1952
39	1946
52	1856
11	1904
43	2006
28	1902
30	1937
13	1965
7	1903
50	1974
41	1866
22	1898
9	1880
12	1864
1	1852
42	1923
0	1955
19	1832
38	1844
45	1786
47	1930
49	2008
25	1875
24	1946
5	1792
51	1826
33	1953
48	1878
17	1929
8	1890
21	1851
16	2001
23	1976
29	1903
40	1942
35	1873
14	1923
20	1855
37	1884
36	1912
34	1849
46	1885
44	1949
2	1761
15	1882
10	1948
27	1952

Adicionando esses valores em um histograma é possível visualizar o seguinte resultado:



Podemos notar a constância na quantidade de nomes por buckets onde a média foi de 1.901,68, o máximo de nomes em um bucket foi de 2 008 e o mínimo de 1 761.
Mesmo tendo um bom resultado no espalhamento dos dados, não foi alcançado um hashing uniforme, mas sim aproximadamente uniforme.

Para finalizar o projeto com o último desafio, foi implementado um método de ordenação por bucket onde informado o bucket, seus elementos seriam ordenados pelo método quicksort, que possui um ótimo desempenho para essas situações. A ordenação foi feita utilizando pivôs que inicialmente foram definidos através dos valores inicias e finais do bucket, onde percorreram a lista realizando as

devidas trocas quando submetida uma comparação sobre os valores dos mesmos utilizando recursividade até que todos os elementos do bucket estivessem ordenados de forma alfabética.

O projeto ainda conta com outras funcionalidades além dessas revisadas aqui, que são elas:

Inserir nome;
Remover nome;
Imprimir todas as primeiras letras dos nomes dos buckets;
Pesquisar nome;
Imprimir todos os nomes.

Conclusão

A utilização da tabela hash proporcionou agilidade e facilidade ao armazenar, acessar e modificar os dados nela inseridos, isso por conta da implementação de listas encadeadas duplas para a construção da mesma. Por conta de um grande número de valores, a técnica de chaining foi muito eficaz para evitar colisões, separando os valores em buckets utilizando a função hash que foi implementada visando a técnica modular. A nível de ordenação foi utilizado o algortimo de quicksort com a variante de Lomuto, um algoritmo não muito eficaz por conta da variante utilizada mas de fácil aplicação no código.
Unindo todos esses pontos foi possível visualizar a eficiência em utilizar a estrutura de dados Tabela Hash para lidar com uma grande quantidade de dados.