Segundo Trabalho Prático de Algoritmos e Estruturas de Dados II

Arthur Antunes Santos Silva, Dilvonei Lacerda e Lucas Gonçalves Nojiri
Professor: Rafael Sachetto Oliveira
Departamento de Ciências da Computação – Universidade Federal de São João del-Rei - UFSJ – Campus Tancredo Neves

1- Introdução

Este trabalho teve como propósito a fixação de conteúdos aprendidos durante o curso de AEDS 2, sendo eles tipo abstrato de dados, estruturas de dados, árvores binárias e hash. O tema do trabalho foi a implementação de um tipo de dados Dicionário usando estruturas diferentes, sendo elas: árvore binária não balanceada, hash com solução de colisão por listas lineares e hash com solução de colisão por endereçamento aberto. Também foi pedido que 4 operações fossem implementadas para cada estrutura, a inicializar, inserir, remover e pesquisar. Em seguida foi feita uma análise do desempenho das estruturas aplicadas.

2- Árvore binária não balanceada

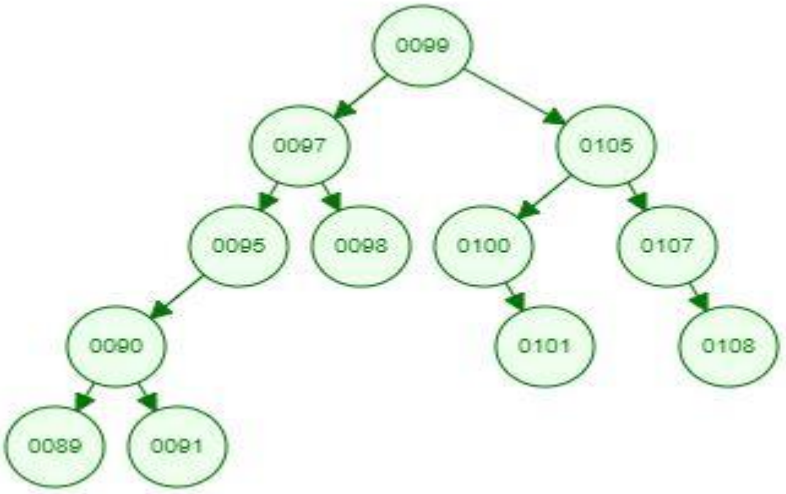

Figure 1 - Exemplo de BST sem balanceamento com elementos sendo maiores que a raiz sendo inseridos a direita e menores a esquerda

2.1-void pesquisar(Registro *x, Apontador p)

Essa função recebe um registro e um apontador como parâmetros, tem como objetivo percorrer a árvore, em busca da chave. A função verifica se a chave existe, se está nos ramos da direita ou na esquerda da árvore.

2.2-void inserir(Registro *x, Apontador p)

Possui como parâmetros um registro e um apontador. Nessa função é feita a inserção de uma chave na árvore verificando se existe espaço, o nível e a posição(direita ou esquerda) que essa chave ficará de um nó. Os nós a esquerda da raiz são elementos menores, e os a direita são elementos maiores.

2.3-void inicializar(Apontador *dicionario)

Essa função faz a inicialização de um dicionário.

2.4-void retirar(Registro *x, Apontador p)

Recebe como parâmetros um registro e um apontador. Essa função faz a remoção de uma chave de um nó da árvore,o processo funciona da seguinte forma: se um nó possuir somente um filho, o elemento que for removido é substituído pelo seu nó filho, porém se o nó possuir dois filhos o registro a ser substituído deve ser, na subárvore à esquerda, o registro mais a direita, no caso da subárvore à direita será o registro mais a esquerda.

3-Hash com solução de colisão por listas lineares

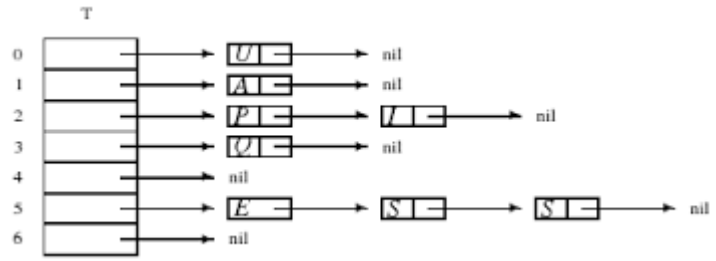

Figure 2 - Exemplo de um Hash por listas lineares

3.1-void inicializar(Dicionario t)

Inicializa cada posição do dicionário, sendo cada posição do dicionário uma lista encadeada que é criada através da função void criar_lista.

3.2-void criar_lista(Lista *lista)

Essa função recebe um parâmetro do tipo lista, e aloca as posições da lista.

3.3-int vazia(Lista lista)

Retorna se a lista está vazia, comparando se o último é igual ao primeiro elemento.

3.4-Apontador pesquisar(Chave ch, peso p, Dicionario t)

Essa função recebe uma chave, peso e o dicionário parâmetros, faz o uso da função h para encontrar o índice do dicionário que será correspondente a determinada lista encadeada. Com a posição da lista encadeada é verificado se a posição é válida, ou seja, se já existe algum elemento já armazenado na mesma, caso seja válido a posição, é comparado os elementos da lista. Se for encontrado retorna o próprio elemento, se não é retornado NULL.

3.5-void ins(Item x, Lista *lista)

Essa função insere um elemento na última posição da lista e aloca memória para o próximo elemento ser inserido, sem necessidade de percorrer toda a lista, desta forma sendo de complexidade O(1), pois já usou o endereço da última posição.

3.6-void inserir(Item x, peso p, Dicionario t)

Essa função verifica se a palavra a ser inserida já não existe no dicionário, se não existir, faz o uso da função h, para encontrar um índice para aquela palavra, com o índice é feita a chamada da função ins que insere o item na última posição.

3.7-void ret(Apontador p, Lista *lista, Item *item)

Recebe como parâmetros um apontador, uma lista e um item. A função verifica se a lista está vazia ou se o item é existente na mesma, se o item for existente ele é removido da lista.

3.8-void retirar(Item x, peso p, Dicionario t)

Essa função faz a remoção de um item. Primeiramente é pesquisado através da chave se o item é existente, caso não seja uma mensagem de erro aparecerá. Se houver um item, a função ret é chamada para remover o item.

3.9-int h(Chave chave, peso p)

Nesta função é feita a transformação de uma palavra para um valor entre 0 e m-1. Na transformação é realizada a soma de cada letra da palavra e multiplicando por um peso, evitando que haja colisões com outras palavras que possuem as mesmas letras.

3.10-void gerar_pesos(peso p)

Gera pesos de forma aleatória no início do programa, usa a struct timeval, e a função gettimeofday, que dá que horas são no momento da execução, desta forma gerando pesos diferentes a cada vez que rodar o programa. Também utiliza a função srand que cria uma semente para o gerador de números aleatórios, dando um valor aleatório em 0 e RAND_MAX, se não houvesse essa função a função sempre geraria a mesma sequência de números aleatórios.

4-Hash com solução de colisão por endereçamento aberto

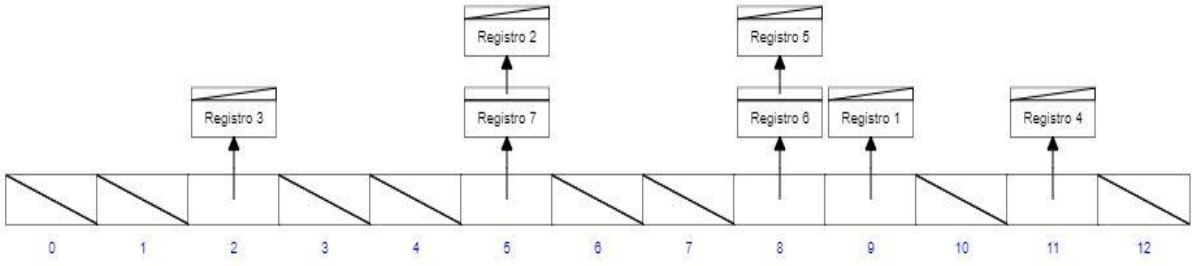

Figure 3 - Exemplo de um Hash por endereçamento aberto

4.1-void inicializar(Dicionario t)

Essa função possui apenas um parâmetro dicionário, tem como objetivo a iniciação de uma tabela vazia.

4.2-Apontador pesquisar(Chave chave, Peso p, Dicionario t)

Faz a pesquisa do elemento na lista, enquanto a posição estiver marcada como VAZIA o programa continua a procura do elemento, caso não seja encontrado, a pesquisa foi sem sucesso, se for encontrado é retornado o elemento.

4.3-void inserir(Item x, Peso p, Dicionario t)

Primeiramente é feito uma pesquisa no hash para verificar se o elemento que está sendo inserido já existe na lista. Se o elemento não for existente, é feito uma busca para encontrar uma posição vazia para que o elemento seja inserido.

4.4-void retirar(Chave chave, Peso p, Dicionario t)

É feito a pesquisa do elemento no hash, caso ele exista é definido na posição do elemento como RETIRADO, se o elemento não estiver na lista é retornado uma mensagem de erro.

5-Análise de desempenho

Foram feitos diversos testes em relação ao tempo com valores diferentes.

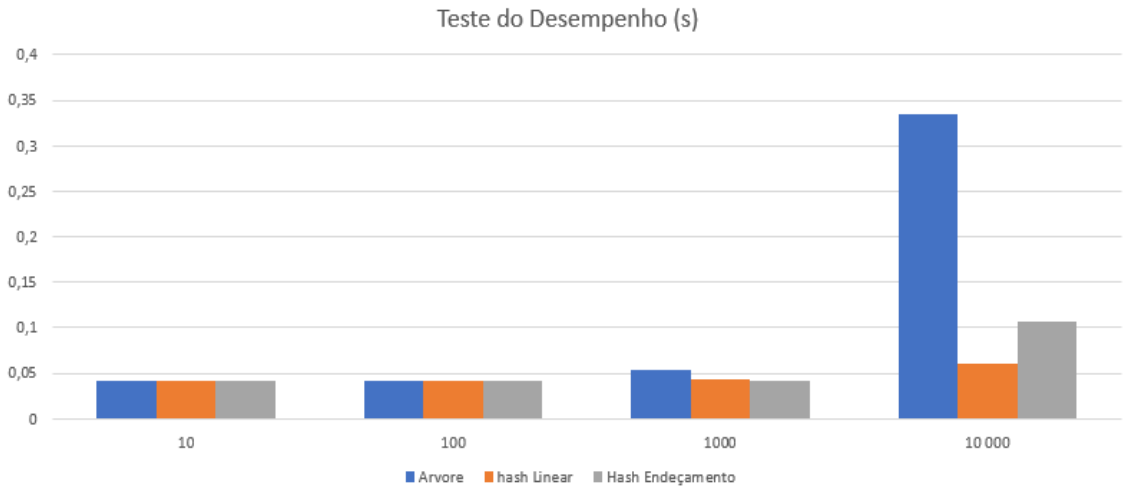

Figure 4 - Gráfico de desempenho

Para valores muito pequenos as três estruturas de dados tiveram o mesmo desempenho basicamente, com valores muito próximos, e com a medida que o tamanho dos dados cresciam as hash 's apresentavam um melhor desempenho, e a BST sem balanceamento. A hashing com solução de colisão por endereçamento aberto e Hash com solução de colisão por listas lineares, possui tempos parecidos, por exemplo quando os valores começam a ficar maiores, em 1000 endereçamento aberto apresenta um tempo menor, mas já em 10.000 lista lineares ganha nesse quesito, no entanto não foi preciso essa análise sobre essa quantidade de dados no endereçamento aberto, pois, houve falha de segmentação, porque faltava memória para ler a quantidade de dados.

6-Conclusão

No decorrer do trabalho foram encontradas algumas dificuldades, que foram no entendimento do endereçamento aberto, a criação do makefile e durante o processo de testagem para a análise de desempenho. De acordo com nossa análise, podemos concluir que para menor quantidade de dados todas as estruturas apresentaram desempenho semelhante, enquanto para uma quantidade maior, o hashing foi o melhor e para uma abundância de dados muito grande devido a falta de memória não foi possível a testagem ideal.

De forma geral, o grupo alcançou êxito no entendimento do código e pôde visualizar a diferença de desempenho e tempo das diferentes estruturas.