Estudante: Vitor Lemos Silva
Professor: Luis Antonio Kowada
Disciplina: Estrutura de Dados e seus Algoritmos
Turma: TCC 00348

Documentação do trabalho

1. Objetivo do trabalho:

Tinha como objetivo implementar as bibliotecas de Heap, Hah e Árvore B + em memória secundária. Este método de implementação é perfeito quando trabalhamos com uma grande quantidade de registros que é o nosso caso, visto que trabalhamos com 10 mil estudantes.

2. Organização do meu projeto:

2.1 Pasta "*data*":

É aqui onde gero/armazeno todos os meus arquivos binários

2.2 Pasta "*library*":

Subdividida em 2 outras pastas.

2.2.1 Pasta "*headers*":

É aqui onde armazeno todos os *.h* do meu projeto, assim como sugere o nome da pasta

2.2.2 Pasta "*src*":

É aqui onde armazeno todos os *.c* relacionados a biblioteca do meu projeto, podemos encontrar explorando os arquivos desta pasta a lógica da implementação de todas as estruturas, bem como a lógica da geração dos meus estudantes que uso como exemplo para o código e configuração dos meus menus.

2.3 Arquivo "*mainFile.c*":

Esse arquivo atua como a main, responsável por centralizar todos os menus.

2.4 Pasta "*output*":

Gero nessa pasta o *.exe* do *mainFile.c*

3. Arquivos:

3.1 "*student.bin*": Armazena todos os meus estudantes.

3.2 "*heap.bin*": Armazena a estrutura da minha Heap.

3.3 "*hash.bin*": Armazena a estrutura da minha Hash.

3.4 "*index_btree.bin*": Armazena os índices da minha Árvore B +.

3.5 “*leaf_btree.bin*”: Armazena as folhas da minha Árvore B +.

4. Structs:

4.1 “TS”:

```
typedef struct student{
    long long int cpf;
    int score;
    char name[50];
}TS;
```

4.2 “btree_node”:

```
typedef struct bplus_tree{
    int is_leaf;//bool
    int num_keys;//numero de chaves
    long long int parent_pointer;//posição do pai
    long long int keys[2 * d];//cpfs
    long long int children[2 * d + 1];//endereço de cada filho
    long long int position;//endereço atual
    long long int records[2 * d];//endereço de todos os elementos presentes no meu nó
    long long int next_node;//posição do proximo
} btree_node;
```

5. Análise da implementação das bibliotecas:

5.1 Heap:

5.1.1 Inserção:

A inserção em Heap pode ser separada em 2 passos:

- Escrever nossa nova informação no fim do arquivo da heap.
- Chamar a função *rise*. Essa função é responsável por fazer a comparação entre a nota do seu pai e a sua. Se a nota do elemento inserido for maior ele assume a posição do pai.

5.1.2 Busca:

A busca em Heap não é muito otimizada. Aqui fizemos um *for* que irá ler todos os elementos em ordem na minha Heap até achar o elemento que procuro, transformando minha busca em O(n).

5.1.3 Remoção:

A remoção em Heap pode ser separada em 3 passos:

- Trocar o primeiro elemento, aquele com a maior prioridade, com o último.
- Diminuir o tamanho da Heap, fazendo com que o último elemento, que anteriormente era o que tinha maior prioridade, seja excluído de fato.
- Chamar a função *descend*. Essa função é responsável por fazer a comparação entre a nota do seu do seus filhos e a sua. Se a nota de um dos filhos for maior esse filho assume a posição do pai.

5.2 Hash:

Implementei uma Hash de Endereçamento Aberto. Para calcular a posição usamos o resto da divisão do *rand* com o tamanho atual da minha Hash.

5.2.1 Inserção:

A inserção da Hash pode ser separada em 2 passos:
- Chamar a função *h* para calcular a posição no qual vamos inserir.
- Verificar se a posição está disponível, caso não incrementamos a colisão em 1 e chamamos a função *h* novamente.

Repetimos esse processo até achar um espaço vazio, caso isso não ocorra teremos um overflow.

5.2.2 Busca:

A busca funciona de forma similar a inserção. Calculamos a posição de onde o elemento deveria estar, se não acharmos o elemento aumentamos a tentativa em 1 e calculamos a posição novamente, mesmo usando o *rand* nossa inserção não é completamente aleatória, devia ao *srand*.

5.2.3 Remoção:

Com a função *hash_search* em nossas mãos a remoção se torna um pouco mais simples. Basta chamar a busca e reescrever o elemento na posição que a busca retornou por um elemento vazio.

5.3 Árvore B +:

5.3.1 Inserção:

Na inserção da Árvore B + utilizamos a estratégia de divisão. Descemos na árvore, verificando se o próximo nó a ser visitado já está cheio. Caso o nó esteja cheio, dividimos ele antes de descer. Dessa forma garantimos que o nó pai sempre terá espaço para um novo elemento.

5.3.2 Busca:

Começamos na raiz e vamos descer de nível por nível. As chaves nos nós internos nos direcionam até chegar em uma folha, onde reside a informação. Na folha, fazemos uma busca simples para encontrar o elemento

5.3.3 Remoção:

A remoção é a operação mais complexa de se implementar no trabalho, e pode ser dividido em 4 partes:
- Realizar uma busca para encontrar o nó em uma das folhas.
- Fazer uma simples remoção para remover o nó da folha.
- Verificar se obedecemos o número mínimo de elementos em uma chave, d - 1.
- Se não tivermos obedecendo esta regra realizamos uma concatenação.