ACH2024 - Algoritmos e Estruturas de Dados II

Docente: Karina Valdivia Delgado

Beatriz Abreu - 11795846

Beatriz Thompson - 8586728

Guilherme Umemura - 9353592

Felipe Antonio R Bordin - 10716421

SÃO PAULO

2023

1. Qual a diferença entre pos de bTreeNode e nextPos de tree?

Na estrutura *bTreeNode*, o campo *pos* representa a posição do nó dentro do arquivo da árvore B. Ele indica o deslocamento ou posição no arquivo onde os dados do nó são armazenados. Essa posição é usada para ler e gravar o nó no arquivo.

O campo *nextPos* é uma propriedade da estrutura *tree*. Ele representa a posição onde o próximo nó será inserido no arquivo da árvore B. Quando um novo nó é criado e precisa ser inserido no arquivo, o *nextPos* é usado para determinar a posição para o novo nó. Após a inserção, o *nextPos* é atualizado para apontar para a próxima posição disponível para futuras inserções.

Resumindo: *pos* é a posição de um nó no arquivo da árvore B, e *nextPos* é a posição onde o próximo nó será inserido no arquivo da árvore B.

2. Quais as funções relacionadas com a inserção e o que cada uma delas faz? A inserção dessa implementação funciona corretamente? Caso não, que mudanças foram feitas e em que funções para obter uma versão correta da inserção?

Por exemplo, teste com:

make all t=3

./run -b

How many records do you want to build from dataset? 28

Faça um desenho da árvore B que deveria ser criada mostrando as chaves dos registros. A lista de links de todos os nós criados pelo algoritmo está correta?

insert: Essa função é responsável por inserir um novo registro na árvore B. Ela chama a função auxiliar *insertNonFull* para realizar a inserção em um nó que não está cheio. Se o nó raiz estiver cheio, a função cria um novo nó raiz e divide o nó cheio.

splitChild é responsável por dividir um nó filho quando ele está cheio durante a operação de inserção de uma nova chave na árvore B. A divisão ocorre para acomodar a nova chave e

manter a propriedade da árvore B, que estabelece um limite superior no número de chaves em um nó.

insertNonFull: Essa função é chamada pela função insert e é responsável por inserir um novo registro em um nó que não está cheio. Ela realiza a inserção de forma recursiva, movendo-se para o nó filho apropriado até encontrar um nó folha adequado para a inserção.

A implementação da inserção fornecida no script não está inteiramente correta. Para corrigir esses problemas e obter uma versão correta da inserção, foram feitas as seguintes alterações:

Na função *insertNonFull*, foi adicionada uma verificação para evitar a inserção de registros duplicados na árvore. Foi corrigido o loop de movimentação dos registros durante a divisão de um nó cheio na função *splitChild*. Foi adicionada uma verificação na função insert para lidar com a inserção em uma árvore vazia. Essas alterações devem corrigir os principais problemas na função de inserção e garantem que ela funcione corretamente.

3. Quais as funções relacionadas com a remoção e o que cada uma delas faz? A remoção dessa implementação funciona corretamente? Caso não, que mudanças foram feitas e em que funções para obter uma versão correta da remoção?
Faça as remoções dos seguintes registros da árvore criada no item 2: 774597, 996522, 782891 e 676106.
Faça um desenho do estado da árvore após essas remoções.

A função *removeNode* é responsável por remover um nó da árvore B, dado um nó específico e uma chave a ser removida. Vou explicar o funcionamento do removeNode passo a passo:

A função recebe como parâmetros a árvore B *(bTree* tree)*, o nó a ser removido (*bTreeNode* node*) e a chave a ser removida *(int k)*.

A função inicia procurando a posição da chave dentro do nó. A função *findKey* é chamada, passando o nó e a chave como parâmetros. Essa função realiza uma busca binária no nó para encontrar a posição da chave ou a posição onde a chave deveria estar.

Se o nó for uma folha, ou seja, não tiver filhos, a função chama *removeFromLeaf*, passando o nó e a posição da chave como parâmetros. Essa função remove a chave do nó folha.

Caso contrário, se o nó tiver filhos, a função chama *removeFromNonLeaf*, passando o nó e a posição da chave como parâmetros. Essa função remove a chave de um nó não folha.

Após a remoção da chave, a função faz uma verificação para garantir que o nó ainda esteja seguindo as propriedades da árvore B. Se o número de chaves no nó for menor que *t-1*, a função realiza uma série de operações para reequilibrar a árvore.

A função verifica se o nó tem um filho à esquerda e, se tiver, verifica se esse filho tem pelo menos *t* chaves. Se tiver, a função chama *borrowFromPrev*, passando o nó e a posição da chave como parâmetros. Essa função transfere uma chave do filho esquerdo para o nó atual.

Se o nó não tiver um filho à esquerda com pelo menos t chaves, a função verifica se ele tem um filho à direita e, se tiver, verifica se esse filho tem pelo menos t chaves. Se tiver, a função chama *borrowFromNext*, passando o nó e a posição da chave como parâmetros. Essa função transfere uma chave do filho direito para o nó atual.

Se o nó não tiver filhos com pelo menos *t* chaves, a função verifica se pode realizar uma fusão. A função chama *merge*, passando o nó, a posição da chave e o filho direito ou esquerdo correspondente como parâmetros. Essa função funde o nó com o filho correspondente, removendo a chave do nó e reposicionando os ponteiros.

Após todas as operações de reequilíbrio, a função chama recursivamente *removeNode* novamente para remover a chave do filho correspondente.

A função retorna quando a chave é removida ou quando a recursão chega a um nó folha.
removeFromTree: Essa função é responsável por remover um registro com base na chave especificada. Ela chama outras funções auxiliares para lidar com a remoção em diferentes casos, dependendo se o nó a ser removido é uma folha ou um nó interno.

Abaixo uma explicação detalhada de cada um dos métodos chamados por *removeNode:*

removeFromNonLeaf: quando o nó a ser removido não é uma folha. Ela encontra o sucessor do registro a ser removido, substitui o registro pelo sucessor e, em seguida, remove o sucessor recursivamente.

removeFromLeaf: Essa função é chamada pela função *removeFromTree* quando o nó a ser removido é uma folha. Ela remove o registro diretamente do nó e rearranja os registros restantes, se necessário.

merge: Essa função é chamada quando um nó interno tem um número menor de registros do que o mínimo permitido. Ela mescla o nó com seu irmão adjacente e a chave correspondente do pai para manter a propriedade da árvore B.

borrowFromNext e *borrowFromPrev*: Essas funções são chamadas quando um nó interno tem um número menor de registros do que o mínimo permitido, mas pode obter um registro extra do irmão adjacente. Elas realizam a transferência de registros entre o nó e seu irmão.

fill: Essa função é chamada quando um nó interno tem um número menor de registros do que o mínimo permitido, mas não pode obter um registro extra do irmão adjacente. Ela preenche o nó com registros do irmão adjacente e a chave correspondente do pai.

getSucc e *getPred*: Essas funções são usadas para obter o sucessor e o predecessor de um registro, respectivamente, quando a remoção ocorre em um nó interno.

A função de remoção fornecida no script tem alguns problemas e não funciona corretamente em certos casos. Um problema é que a função *getSucc* não lida corretamente com casos em que o sucessor está em um nó filho. Além disso, a função removeFromLeaf não remove o registro corretamente e não rearranja os registros restantes adequadamente.

Para corrigir esses problemas e fazer com que a função de remoção funcione corretamente, as seguintes alterações podem ser feitas: na função *getSucc*, é necessário adicionar um caso de tratamento quando o sucessor está em um nó filho. Na função *removeFromLeaf*, é necessário atualizar o loop que move os registros após a remoção para garantir que eles sejam movidos corretamente.

4. Quais as funções relacionadas com a busca e o que cada uma delas faz? A operação de busca está corretamente implementada? Buscar 132486 e 990171.

A função *search* é a função principal de busca. Ela recebe como parâmetros a árvore B (*bTree* tree*) e a chave a ser buscada (*int key*). Essa função chama a função *searchRecursive*, passando a raiz da árvore (*tree->root*) como parâmetro para iniciar a busca recursiva.

A função *search*Recursive é uma função auxiliar que realiza a busca recursiva pela chave na árvore B. Ela recebe como parâmetros a árvore B (*bTree* tree*), a chave a ser buscada (*int key*) e o nó atual em que a busca está sendo realizada *(bTreeNode* roo*t). Essa função implementa a lógica de busca na árvore B, verificando se a chave está presente no nó atual e, caso contrário, descendo para o filho apropriado até encontrar a chave desejada ou chegar a um nó folha.

A função *findKey* é uma função auxiliar que busca a posição de uma chave em um nó. Ela recebe como parâmetros o nó em que a busca será realizada (*bTreeNode* node*) e a chave a ser encontrada (*int k*). Essa função realiza uma busca binária no nó, comparando a chave desejada com as chaves presentes no nó. Se a chave for encontrada, a função retorna a posição da chave no nó. Caso contrário, a função retorna a posição onde a chave deveria ser inserida.

No que se refere à implementação da busca, com base no código fornecido, parece que a operação de busca está corretamente implementada. As funções search e *searchRecursive* seguem a lógica básica de busca em uma árvore B, percorrendo os nós e comparando as chaves para encontrar a chave desejada.

5. O que faz a função traverse?

A função *traverse* le da memória secundária as informações contidas no nó *bTree* tree* e as passas para a variável *toPrint*(tipo bTree*,contida em memória principal) ,que é utilizada pela função *dispNode* que printa as informações contidas em to *print.* Esse processo

é feito de forma recursiva pela função *traverse* de forma a ser executado no primeiro nó passado e seus descendentes.

6.Os registros criados do tipo recordNode estão sendo armazenados na memória principal ou em um arquivo? Depois de fechar o programa é possível recuperar esses registros criados?

Os registros do tipo *recordNode* são criados na memória principal, logo não é possível de ser recuperado após o fechamento do programa.

Extra:
Modificações na entrada do usuário que podia gerar vários segfault