Projeto de EDA - Contador de frequências usando Estruturas de Dados Avançadas

Nome: Willgner Silva Ferreira
Matrícula: 567152

1 Implementação

Para a construção do contador de frequências, foram construídas inicialmente 4 estruturas de dados génericas que serão descorridas ao longo dessa sessão com mais detalhamento, sendo elas: tabela de dispersão de endecadeamento exterior ($ChainedHashTable$), tabela de dispersão de endereçamento aberto ($OpenAdressingHashTable$), árvore AVL e uma árvore rubro-negra (RBT).

1.1 Hash Table

A tabela por dispersão ou *Hash Table*, é uma estrutura de dados que busca armazenar valores em outra estrutura de forma semi aleatória, a referência utilizada usava um $vector$ como estrutura base, essa semi aleatóridade é dada pela função $hashing()$, que de maneira estável retorna um índice i da tabela, para uma chave k, pode ocorrer de chaves k diferentes receberem o mesmo índice, quando isso ocorre, é chamado de colisão, para evitar um número de grande de colisões é selecionado um número primo como tamanho da tabela, pois ele tem menos divisores em comum, em uma possível operação de módulo realizada no $hashing()$, tem menos chance de causar uma colisão.

Além disso, quando é chegado em um certo fator de carga específico, razão entre tamanho da tabela e número de elementos, é realizada a operação de $rehash()$, ou seja, a tabela de dispersão tem seu número de ”slots”, espaços reservados para um valor futuro, aumentado.

Em uma Hash Table perfeita, ela teria a função $hashing\_perfeita$ que evitaria a ocorrência de colisões, como é uma função inalcansável, foram criadas alguns tipos de Hash Tables que utilizam diferentes métodos para o tratamento de colisões, entre elas, foram selecionadas a *Chained Hash Table* e *Open Adressing Hash Table*, para serem utilizadas como referência.

Por motivos de análises futuros, foram selecionados 2 contadores específicos para as tabelas de dispersão, sendo eles, um contador de comparação entre chaves para a construção de uma Hash Table, e outro para contar as colisões que ocorrem na Hash Table.

1.1.1 Chained Hash Table

Em sua definição, ela é uma tabela de dispersão que trata a ocorrências de colisões, com encadeamento exterior, ou seja, cada ”espaço”em sua estrutura, é formado por uma estrutura de lista, que armazena os valores associoados aquele espaço pela função $hash()$.

As suas principais funções são:

1. **bool add(const Key &k, const Value &v);**

Resumo: Promete inserir uma chave k e valor v na tabela de dispersão, se o fator de carga for maior que o fator de carga máximo, faz uma chamada para função rehash(), passando como parâmetro o dobro do tamanho atual da tabela. O elemento é inserido somente se a chave já não estiver presente na tabela(cada comparação realizada é contabilizada no contador), caso a lista do "slot", referente a essa chave, não estiver vazia, adiciona o número de elementos presenta na lista ao contador de colisões, e insere ao fim da lista. caso esteja vazia, apenas insere o elemento, e incrementa o número de elementos.

Parâmetros: recebe uma referência de chave constante, e uma referência de valor constante.

Retorno: retorna **TRUE** se a inserção der certo, se não (caso que a chave já esta na tabela) retorna **FALSE**.

2. **bool remove(const Key &k);**

   Resumo: Promete remover um elemento com chave k da tabela de dispersão, caso a chave exista, apaga ela e decrementa o número de elementos.

   Parâmentros: recebe uma refência constante da chave k.

   Retorno: retorna **TRUE** se a remoção der certo, se não (caso que a chave não exista na tabela) retorna **FALSE**.

3. **Value &at(const Key &k);**

   Resumo: Promete retorna uma referência para o valor associado a chave k. Se k não estiver na tabela, a função lança uma invalid_argument exception.

   Parâmetros: recebe uma referência constante da chave k.

   Retorno: retorna o valor associado a chave k, retorna uma exception caso não exista tal valor.

4. **Value &operator[](const Key &k);**

   Resumo: Sobrecarga do operador de indexação, caso não exista valor associado a chave k, ele insere o valor, e retorna uma referência do valor inserido.

   Parâmetros: recebe uma referência constante da chave k.

   Retorno: retorna o valor associado a chave k.

5. **void rehash(size_t m);**

   Resumo: Recebe um inteiro nao negativo m e faz com que o tamanho da tabela seja um numero primo maior ou igual a m. Se m for maior que o tamanho atual da tabela, um rehashing eh realizado. Se m for menor que o tamanho atual da tabela, a funcao nao tem nenhum efeito. Um rehashing eh uma operacao de reconstrucao da tabela: Todos os elementos no container sao rearranjados de acordo com o seu valor de hashing dentro na nova tabela. Isto pode alterar a ordem de iteracao dos elementos dentro do container. Operacoes de rehashing sao realizadas automaticamente pelo container sempre que load_factor() ultrapassa o m_max_load_factor.

Parâmetros: recebe um valor não negativo m.

Retorno: vazio.

6. **size_t hash_code(const Key &k) const;**

   Resumo: Retorna um inteiro no intervalo [0 ... m_table_size-1].Esta funcao recebe uma chave k e faz o seguinte:

   (1) computa o codigo hash h(k) usando a funcao no atributo privado m_hashing

   (2) computa um indice no intervalo [0 ... m_table_size-1] aplicando o metodo da divisao: $h(k)\%m\_table\_size$

   Parâmetros: Recebe uma referência constante da chave k.

   Retorno: O index, que a chave k está associada.

7. **void show_ordened() const;**

   Resumo: Guarda todos os valores da tabela hash em um vector, ordena ele, e depois imprime no terminal.

   Parâmetros: Nenhum.

   Retorno: Vazio.

1.1.2 Open Adressing Hash Table

Em sua definição, ela é uma tabela de dispersão que trata ocorrências de colisões por endereçamento aberto, ou seja, todos os elementos são armazenados diretamente no vetor principal e, quando a função $hash()$ aponta para uma posição já ocupada, novas posições são buscadas por sondagens sucessivas (linear, quadrática ou dupla), até encontrar um slot livre. As suas principais funções são:

1. **bool add(const Key &k, const Value &v);**

   Resumo: Insere ou atualiza o par (k,v) na tabela usando sondagens sucessivas. Se a chave já estiver presente, atualiza o valor. Caso contrário, executa probes até encontrar um slot com status diferente de ACTIVE, conta colisões em cada tentativa falha e insere o novo elemento.

   Parâmetros:

   - `k`: referência constante à chave a inserir.
   - `v`: referência constante ao valor associado.

   Retorno:

   - **true** se a inserção ou atualização for bem-sucedida.
   - **false** se não houver slot livre após `m_table_size` sondagens (tabela cheia).

2. **bool remove(const Key &k);**

   Resumo: Marca o slot onde a chave k está como DELETED e decrementa o contador de elementos, abortando se a chave não existir.

   Parâmetros:

   - `k`: referência constante à chave a remover.

   Retorno:

   - **true** se a chave for encontrada e removida.
   - **false** se a chave não estiver presente na tabela.

3. **Value& operator[](const Key &k);**

   Resumo: Retorna referência ao valor associado a k. Se não existir, insere (k, Value{}) com valor padrão — possivelmente fazendo `rehash()` — e então retorna a nova referência.

   Parâmetros:

   - `k`: referência constante à chave a acessar ou criar.

   Retorno:

   - Referência ao `Value` associado a k.
   - Pode lançar `std::runtime_error` se a inserção falhar mesmo após `rehash()`.

4. **Value& at(const Key &k);**

   Resumo: Retorna referência ao valor associado a k. Não insere novos elementos; se k não existir, lança `std::invalid_argument`.

   Parâmetros:

   - `k`: referência constante à chave a consultar.

   Retorno:

   - Referência ao `Value` armazenado para k.
   - Lança `std::invalid_argument` caso k não seja encontrada.

5. **void rehash(size_t m);**

   Resumo: Ajusta o tamanho da tabela para o menor número primo m. Se m ultrapassar o tamanho atual, aloca nova tabela, reinsere apenas elementos ACTIVE e zera o contador de elementos.

   Parâmetros:

   - `m`: tamanho mínimo desejado da tabela (não negativo).

Retorno: vazio

6. **void show_ordened() const;**

   Resumo: Extrai todos os pares (Key,Value) com status ACTIVE, armazena-os num vetor, ordena por chave em ordem crescente e imprime no terminal.

   Parâmetros: nenhum

   Retorno: vazio

7. **size_t aux_search(const Key &k);**

   Resumo: Executa sondagem (probing) para localizar a chave k. Chama `hash_code(k,i)` para i = 0... até encontrar:

   - slot ACTIVE com chave igual → retorna índice;
   - slot EMPTY → interrompe e lança `std::invalid_argument`.

   Parâmetros:

   - k: referência constante à chave a procurar.

   Retorno:

   - Índice (size_t) onde a chave está armazenada.
   - Lança `std::invalid_argument` se não encontrar k após `m_table_size` tentativas.

8. **size_t hash_code(const Key &k, size_t i) const**

   Resumo: Retorna um inteiro no intervalo [0 ... m_table_size-1]. Esta funcao recebe uma chave k e um indice i faz o seguinte: Realiza dois modelos de hashing:

   $$h1 = m\_hashing(k)\%m\_table\_size$$

   $$h2 = 1 + (m\_hashing(k)\%(m\_table\_size - 1))$$

   Juntando as duas num modelo de hashing duplo, tal que: hash_duplo:

   $$(h1 + i * h2)\%m\_table\_size$$

   Parâmetros: Recebe uma referência constante à chave k e o índice i de um "slot"da tabela.

   Retorno: Retorna o "slot"da tabela associada a chave k e índice i.

1.2 Árvores

A Árvore Binária de Busca (ou BST) é uma estrutura de dados que organiza pares (chave, valor) em nós interligados por ponteiros, de modo que, para cada nó, todas as chaves na subárvore esquerda sejam menores e todas na subárvore direita sejam maiores. Essa ordenação estável é garantida pela função de comparação, que ao inserir ou buscar uma chave k percorre o caminho apropriado (esquerda ou direita) até encontrá-la ou atingir um nó nulo.

Em vez de "colisões" como na hash table, na BST o principal risco é a degeneração em lista (quando as chaves chegam em ordem ascendente ou descendente), elevando a complexidade de busca de $O(logn)$ a $O(n)$. Para mitigar esse problema, existem variantes autocontroladas—como AVL e Red–Black—que, ao detectar desbalanceamento (diferença de alturas maior que 1, ou violações de cor), executam rotações simples ou duplas para restaurar $O(logn)$ de profundidade.

Por motivos de análises futuras, foram selecionados 2 contadores específicos para as árvores de busca binária, sendo eles, um contador de comparação entre as chaves para a construção da árvore, e outro para contar as rotações que ocorrem na árvore.

1.2.1 AVL

Em sua definição, a AVL é uma árvore binária de busca auto-balanceada que mantém em cada nó uma altura e garante que, para todo nó, a diferença entre as alturas de suas subárvores esquerda e direita seja no máximo 1. Quando uma operação de inserção ou remoção provoca um desequilíbrio, rotações simples ou duplas (LL, LR, RL, RR) são aplicadas localmente para restaurar esse balanceamento, assegurando profundidade sempre $\theta(logn)$.

As suas principais funções são:

1. **void insert(Key key, Value value);**

   Resumo: Insere um novo nó com chave `key` e valor `value`. A inserção é feita recursivamente em `insert(node, key, value)`, e após posicionar o nó, chama `fixup_node(...)` para restaurar o balanceamento, aplicando rotações simples ou duplas conforme necessário.

   Parâmetros:

   - `key`: cópia da chave a ser inserida.
   - `value`: cópia do valor associado à chave.

   Retorno:

   - vazio (o nó raiz `m_root` é atualizado internamente).

2. **void erase(Key key);**

   Resumo: Remove o nó cuja chave é igual a `key`. A remoção é feita recursivamente em `erase(node, key)`, que substitui o nó por seu sucessor quando necessário e depois chama `fixup_deletion(...)` para manter o balanceamento.

   Parâmetros:

- `key`: cópia da chave a ser removida.

Retorno:

- vazio (a raiz `m_root` é ajustada internamente).

3. **void update(Key key, Value value);**

   Resumo: Procura iterativamente, a partir da raiz, o nó com chave `key`. Se encontrado, atribui `value` ao seu campo de valor. Caso a chave não exista, lança `std::invalid_argument`.

   Parâmetros:

   - `key`: cópia da chave cujo valor será atualizado.
   - `value`: nova cópia do valor a associar à chave.

   Retorno:

   - vazio (lança exceção se a chave não for encontrada).

4. **Node<Key,Value>* fixup_node(Node<Key,Value>* node, Key key);**

   Resumo: Calcula o fator de balanceamento de `node`. Se o nó estiver desbalanceado após inserção de `key`, aplica rotações simples ou duplas (LL, LR, RL, RR) para restaurar o equilíbrio. Atualiza a altura do nó antes de retornar.

   Parâmetros:

   - `node`: ponteiro para o nó raiz do subárvore a ajustar.
   - `key`: chave inserida que pode ter causado o desbalanceamento.

   Retorno:

   - novo ponteiro raiz desse subárvore, após as rotações e ajuste de altura.

5. **Node<Key,Value>* fixup_deletion(Node<Key,Value>* node);**

   Resumo: Verifica o fator de balanceamento de `node` após uma remoção. Se desbalanceado, aplica rotações simples ou duplas para restaurar o equilíbrio. Atualiza a altura do nó antes de retornar.

   Parâmetros:

   - `node`: ponteiro para o nó raiz do subárvore a ajustar.

   Retorno:

   - novo ponteiro raiz desse subárvore, após as rotações e ajuste de altura.

6. **Node<Key,Value>* remove_successor(Node<Key,Value>* p, Node <Key,Value>* node);**

   Resumo: Encontra e remove o menor nó da subárvore de `node`, substituindo o valor de `p` pelo valor do sucessor. Usada durante a remoção para tratar nós com dois filhos.

   Parâmetros:

   - `p`: ponteiro para o nó que receberá o valor do sucessor.
   - `node`: ponteiro raiz da subárvore onde se procura o sucessor.

   Retorno:

   - ponteiro para a subárvore resultante após remover o sucessor.

7. **void show();**

   Resumo: Percorre a árvore em ordem (inorder) usando pilha para não recursão, imprimindo cada par (chave, valor) separado por vírgulas.

   Parâmetros:

   - nenhum.

   Retorno:

   - vazio (imprime a saída diretamente no `std::cout`).

1.2.2 RBT

Em sua definição, a Red–Black Tree é uma árvore binária de busca balanceada na qual cada nó recebe uma cor (vermelho ou preto) e se obedecem quatro invariantes principais: a raiz e os nós-nil sentinela são pretos; nós vermelhos têm Ambos os filhos pretos; e todos os caminhos da raiz até folhas contêm o mesmo número de nós pretos. Inserções e remoções que violam essas regras são corrigidas por rotações e recolorações locais, o que assegura que sua altura permaneça sempre$O(logn)$.

As suas principais funções são:

1. **void insert(Key key, Value value);**

   Resumo: Insere um novo nó com chave `key` e valor `value`. Percorre a partir da raiz até um sentinela `nil`, cria o nó colorido RED, conecta-o ao pai apropriado e invoca `fixup_node(aux);` para restaurar as propriedades da RBT (balanceamento de cores e rotações).

   Parâmetros:

   - `key`: cópia da chave a inserir.
   - `value`: cópia do valor associado.

   Retorno: vazio (ajusta internamente `m_root`).

2. **void remove(Key key);**

   Resumo: Localiza iterativamente o nó com chave `key`. Se encontrado (diferente de `nil`), chama `deletion(atual);` para retirar o nó, possivelmente transplantar seu sucessor e, se necessário, executa `fixup_deletion(...);` para manter as invariantes de cor.

   Parâmetros:

   - `key`: cópia da chave a remover.

   Retorno: vazio (ajusta internamente `m_root`).

3. **void update(Key key, Value value);**

   Resumo: Percorre a árvore em busca do nó cuja chave é `key`. Se encontrado, atualiza seu campo de valor para `value`. Caso não exista, lança `std::invalid_argument("invalid key on update")`.

   Parâmetros:

   - `key`: cópia da chave a atualizar.
   - `value`: nova cópia do valor.

   Retorno: vazio (lança exceção se não encontrar).

4. **void fixup_node(Node_RBTK<Key,Value>* node);**

   Resumo: Após uma inserção, ajusta cores e realiza rotações (LL, LR, RL, RR) enquanto o pai de `node` for RED, garantindo que caminhos tenham o mesmo número de nós negros e que não haja dois vermelhos consecutivos.

   Parâmetros:

   - `node`: ponteiro para o nó recém-inserido a equilibrar.

   Retorno: vazio (atualiza `m_root-¿color` para BLACK ao final).

5. **void fixup_deletion(Node_RBT<Key,Value>* node);**

   Resumo: Após remoção de um nó preto, corrige "nó duplo-negro" usando rotações e recolorings até restaurar as propriedades da RBT.

   Parâmetros:

   - `node`: ponteiro para o filho que substituiu o nó removido.

   Retorno: vazio (finaliza fazendo `node-¿color = BLACK`).

6. **void deletion(Node_RBT<Key,Value*>node);**

   Resumo: Realiza o transplante do nó `node`. Se `node` tiver dois filhos, encontra seu sucessor mínimo, copia valor e remove o sucessor. Em seguida, se `aux-¿color` for BLACK, chama `fixup_deletion(`$_a ux); e finalmente deleta o nó.$

Parâmetros:

- `node`: ponteiro para o nó a ser removido.

Retorno: vazio (libera memória e mantêm invariantes).

7. **void show();**

   Resumo: Percorre a árvore em ordem (inorder) usando uma pilha não recursiva, imprimindo cada par `(chave, valor)`. Nós vermelhos são exibidos em cor vermelha e nós pretos em cor preta no terminal, separados por vírgulas.

   Parâmetros: nenhum.

   Retorno: vazio (saída via `std::cout`).