Análise do Sistema de Gerenciamento de Contatos

Estruturas de Dados e Complexidade

Relatório Técnico

10 de março de 2025

1 Introdução

Este relatório apresenta uma análise detalhada do sistema de gerenciamento de contatos (Contatinho), focando nas estruturas de dados utilizadas, complexidade computacional, análise de memória e comparações com sistemas de banco de dados relacionais.

2 Estruturas de Dados

2.1 Visão Geral

O sistema utiliza quatro estruturas principais:

- Red-black BST para indexação por nome
- Red-black BST para indexação por Instagram
- Red-black BST com listas encadeadas para indexação por prioridade
- Fila para manter ordem de inserção

2.2 Justificativa das Escolhas

2.2.1 Red-black BST para Nome e Instagram

Vantagens:

- Busca, inserção e remoção em $O(\log n)$
- Manutenção automática da ordem dos elementos
- Garantia de unicidade das chaves

Alternativas Consideradas:

- Hash Tables: Descartada por não manter ordenação
- Lista: Descartada por busca $O(n)$
- Lista Ligada: Descartada por busca $O(n)$
- Árvore AVL: Descartada por ter maior overhead de implementação e manutenção, com benefícios marginais de performance para o caso de uso

2.2.2 Red-black BST com Listas para Prioridade

Vantagens:

- Ordenação automática das prioridades
- Suporte a múltiplos contatos por prioridade através de listas encadeadas
- Acesso eficiente $O(\log n)$

Alternativas Consideradas:

- Heap: Complexo para gerenciar múltiplos elementos com mesma prioridade
- Lista Ordenada: Inserção ineficiente $O(n)$
- Árvore AVL: Complexidade desnecessária para dados não únicos

2.2.3 Fila para Ordem de Chegada

Vantagens:

- Implementação natural do princípio FIFO
- Remoção do primeiro elemento em $O(1)$
- Overhead mínimo de memória
- Simplicidade de implementação

Alternativas Consideradas:

- Lista Encadeada: Funcionalidade excessiva para o requisito
- Pilha: Ordem LIFO não atenderia o requisito
- Lista Circular: Complexidade desnecessária

3 Análise de Complexidade

3.1 Complexidade Temporal

Adicionar Contato: $O(\log n)$

A operação de adição envolve:

- Inserção na árvore de nomes: $O(\log n)$
- Inserção na árvore de Instagram: $O(\log n)$
- Inserção na árvore de prioridades: $O(\log n)$
- Inserção na fila: $O(1)$

A complexidade final é $O(\log n)$. Todas as operações são independentes e executadas sequencialmente.

Buscar Contato: $O(\log n)$

A busca é realizada em duas etapas:

- Busca na árvore de nomes: $O(\log n)$
- Se não encontrado, busca na árvore de Instagram: $O(\log n)$

Mesmo no pior caso (quando o contato não existe), a complexidade mantém-se em $O(\log n)$ pois as buscas são sequenciais.

Atualizar Contato: $O(\log n)$

A atualização consiste em:

- Busca do contato na árvore de nomes: $O(\log n)$
- Remoção da antiga referência na árvore de Instagram: $O(\log n)$
- Inserção da nova referência na árvore de Instagram: $O(\log n)$
- Atualização na árvore de prioridades: $O(\log n)$ para remoção e inserção

Como todas as operações são executadas sequencialmente, mantendo a complexidade final em $O(\log n)$.

3.1.1 Operações de Listagem

Listar por Prioridade: $O(n \log n)$

- Percorrer a árvore de prioridades: $O(\log n)$ por nível
- Para cada prioridade, acessar a lista de contatos: $O(k)$ onde k é o número de contatos naquela prioridade
- Como $\sum k = n$, e precisamos percorrer $O(\log n)$ níveis da árvore
- A complexidade total resulta em $O(n \log n)$

Listar em Ordem Alfabética: $O(n)$

Operação simples:

- Percorrer a árvore em ordem (in-order traversal): $O(n)$
- Cada nó é visitado exatamente uma vez
- Não requer ordenação adicional pois a árvore já se mantêm ordenada

Remover Primeiro Contato: $O(1)$

A remoção envolve:

- Acesso ao primeiro elemento da fila: $O(1)$
- Remoção nas árvores: $O(\log n)$ para cada árvore
- Porém, como temos acesso direto ao elemento a ser removido, a complexidade amortizada é $O(1)$

3.1.2 Análise do Pior Caso

Em cenários específicos, algumas operações podem ter desempenho diferente:

- Inserções ordenadas poderiam degradar uma árvore não balanceada para $O(n)$
- Múltiplos contatos com mesma prioridade podem aumentar o tempo de listagem
- Remoções consecutivas podem causar rebalanceamento em cascata

3.1.3 Otimizações Possíveis

Algumas melhorias poderiam ser implementadas:

- Cache de resultados frequentes
- Lazy deletion para remoções
- Bulk operations para operações em lote
- Estruturas auxiliares para casos específicos de uso

3.2 Análise de Memória

3.2.1 Estruturas Principais

Consumo de memória por contato:

- Objeto Contato: $O(1)$ - campos fixos
- Referências em TreeMaps: $O(1)$ por índice
- Lista na prioridade: $O(k)$ onde k é número de contatos com mesma prioridade

3.2.2 Overhead Total

- Espaço total: $O(n)$ para n contatos
- Overhead de indexação: $O(n)$ devido às estruturas auxiliares