

Complexidade de Algoritmos

Prof. Diego Buchinger
diego.buchinger@outlook.com
diego.buchinger@udesc.br

Prof. Cristiano Damiani Vasconcellos
cristiano.vasconcellos@udesc.br

Análise de complexidade de Estruturas de Dados Fundamentais

Estruturas de Dados Básicas

Pilha, Fila, Lista Encadeada Simples, Lista Duplamente Encadeada, Árvore Binária, Árvore Rubro Negra

Analisar os seguintes métodos:

* OBS: Fila tem um ponteiro para o início e o fim!

Fila::enfileirar(elemento);

Fila::remover();

Fila::busca(elemento);

Fila::tamanho();

Estruturas de Dados Básicas

Pilha, Fila, Lista Encadeada Simples, Lista Duplamente Encadeada, Árvore Binária, Árvore Rubro Negra

Analisar os seguintes métodos:

* OBS: Lista Encadeada Simples => ponteiro 1º elemento da lista

Elemento => elemento + ponteiro p/ próximo elemento

ListaSimples::adicionarNoInicio(elemento);

ListaSimples::adicionarNoFim(elemento);

ListaSimples::busca(elemento);

ListaSimples::anterior(*elemento);

ListaSimples::proximo(*elemento);

Estruturas de Dados Básicas

Pilha, Fila, Lista Encadeada Simples, Lista Duplamente Encadeada, Árvore Binária, Árvore Rubro Negra

Analisar os seguintes métodos:

* OBS: Lista Dup. Enc. => ponteiro 1º elemento da lista

Element => elemento + ponteiro p/ anterior e próximo elemento

ListaDup::adicionarNoInicio(elemento);

ListaDup::adicionarNoFim(elemento);

ListaDup::busca(elemento);

ListaDup::anterior(*elemento);

ListaDup::proximo(*elemento);

Estruturas de Dados Básicas

Pilha, Fila, Lista Encadeada Simples, Lista Duplamente Encadeada, Árvore Binária, Árvore Rubro Negra

Analisar os seguintes métodos:

* OBS: Árvore => elemento + ponteiro para filhos

Árvore::inserir(elemento);

Árvore::buscar(elemento);

Árvore::pai(*no);

Árvore::maximo();

** E se cada nó tiver um ponteiro para o pai?

Atividade

Analise o código fonte do arquivo estruturas-dados.cpp e informe as complexidades de tempo e espaço dos seguintes métodos considerando o **pior** e **melhor** caso. Ilustre um exemplo de situação onde o **pior** caso e o **melhor** caso ocorreria para cada uma das funções.

[Faça em uma folha para entregar] [no máximo **duplas**]

Pilha	ListaEncadeada	ÁrvoreVP
Empilha	Adicionar	rotateLeft
Desempilha	Anexar	Inserir
Limpar	Remover	Busca
Print	Encontrar	PreOrder
Tamanho	RemoverDuplicatas	Maximo

Hash

Hash - Introdução

Uso de vetores para armazenar elementos é bom.

- Qual a complexidade para acessar um elemento?
- Qual a complexidade para adicionar um elemento?

Limitações dos vetores: os índices só podem ser inteiros

P: Alguma sugestão para podermos usar outros valores (de qualquer tipo) como índices?

Hash - Introdução

Podemos criar uma operação de conversão de um tipo qualquer para um valor inteiro positivo = **FUNÇÃO HASHING**

(chave) \Rightarrow índice

Como poderíamos criar uma função para converter uma string em um inteiro positivo?

- 1 – Somar o código ascii de cada letra
- 2 – Somar o código ascii de cada letra * posição
- 3 – Somatório(código ascii de cada letra * $255^{\text{posição}}$)

Testes: “Joao”, “Maria”, “ola”, “alo”, “oi”, “io”

Hash - Introdução

Na conversão precisamos nos preocupar com mais um detalhe:

Qual é o tamanho do nosso vetor?

Se tivermos um vetor de **10 posições**, qual operação podemos usar para garantir que o nosso inteiro positivo esteja entre 0 e 9?

Fator de carga do Hash: $\lambda = \frac{n^{\circ} \text{ elementos}}{\text{tamanho vetor}}$

Hash - Introdução

Quando usamos uma função hashing que realiza um mapeamento único para cada índice do vetor temos um **hash perfeito**

Se os nomes só podem ter no máximo 5 letras, qual tamanho de vetor podemos utilizar para garantir um hash perfeito?

(note que um hash perfeito depende da função de hashing)

Hash - Introdução

Quando usamos uma função hashing que não realiza um mapeamento único para cada índice do vetor precisamos considerar também que a função de hashing pode gerar **números iguais** para **chaves diferentes**:

Hashing (2) + vetor de 10 posições: “Maria”, “Ana”

A esse evento chamamos de **COLISÃO!**

Nem sempre é viável utilizar um hash perfeito. Logo, uma função hashing deve sempre buscar **minimizar** o número de colisões

Hash - Colisões

Ok, Mas o que fazer quando ocorre uma colisão?

Abordagem 1: Não adicionamos o registro

Abordagem 2 – endereçamento aberto:

Procurar pela próxima posição vaga

❖ Sondagem Linear: avança de um em um

Método deve ser usado para inserção e busca

exemplo: busca pelo 55 (hash: 0); busca pelo 66 (hash: 1)

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17	None	None	31	54

Hash - Colisões

Ok, Mas o que fazer quando ocorre uma colisão?

Abordagem 2 – endereçamento aberto:

Sondagem Linear pode gerar agrupamentos (*clusters*) de elementos (não interessante)

❖ Sondagem Quadrática: avança usando os quadrados

$x+1, x+4, x+9, x+16, x+25, (...)$

Método deve ser usado para inserção e busca

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17	None	None	31	54

Hash - Colisões

Ok, Mas o que fazer quando ocorre uma colisão?

Abordagem 2 – endereçamento aberto:

❖ Hash duplo (rehash): aplicação de uma nova função hash

novo hash = rehash(hash antigo)

rehash(x) = (x + 1) % tamanho vetor

rehash deve garantir que, eventualmente, todas as posições serão avaliadas

Método deve ser usado para inserção e busca

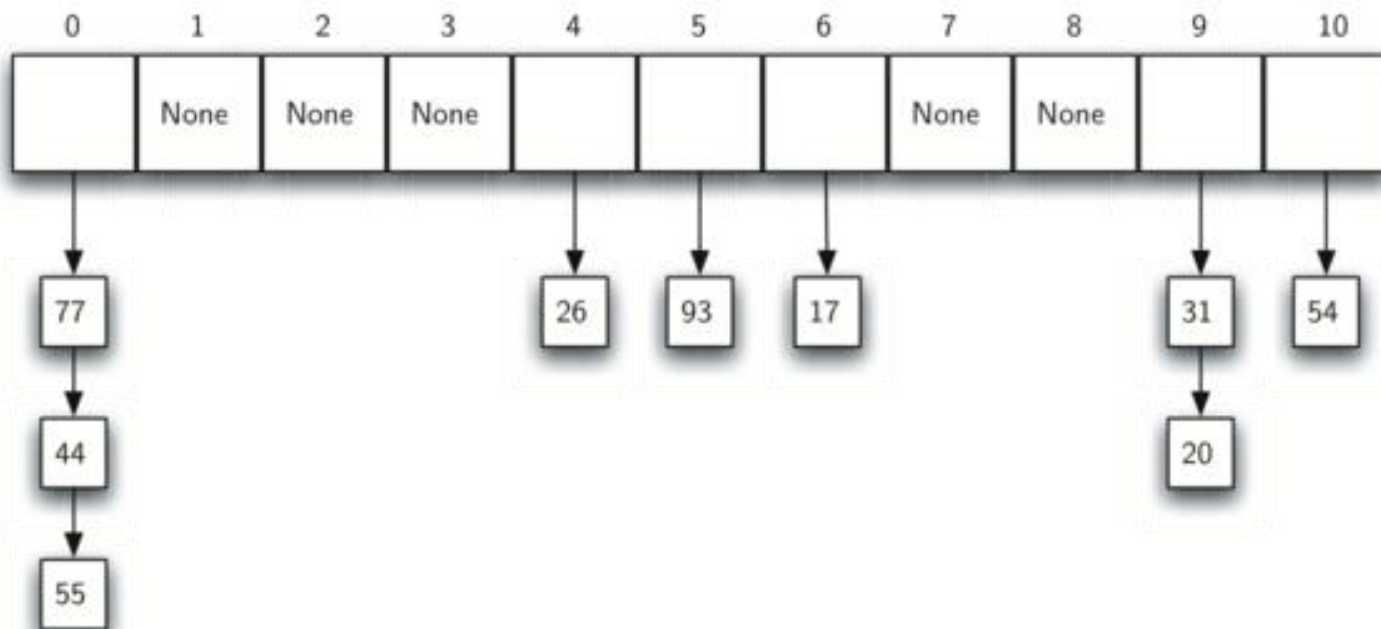
0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17	None	None	31	54

Hash - Colisões

Ok, Mas o que fazer quando ocorre uma colisão?

Abordagem 3 – encadeamento:

Ao invés de utilizarmos um vetor simples utilizamos um vetor de alguma estrutura de dados (lista, árvore)



Hash - Introdução

Hash - método da divisão:

Escolhemos um valor m como divisor de modo a evitar colisões (um número que **não** seja potência de 2 $\Rightarrow \neq 2^p$)

Um bom valor pode ser um número primo ou então $2^p - 1$

Podemos realizar duas divisões: $(valor \% m) \% tam$

Hash – método da multiplicação:

Pesquisar =)

Hash – Principais funções

Hash(tamanho)

put(chave, valor) => boolean

get(chave) => valor

listar() => (chaves, valores)

Hash – Analise Complexidade

Considerando um hash de strings com capacidade n , e tamanho de strings m :

- Qual a complexidade para calcular a função hash (1), (2) e (3)?
- Qual a complexidade (melhor e pior caso) para inserir e buscar um elemento usando sondagem linear? E para listar todos os elementos do hash?

Hash – Analise Complexidade

Considerando um hash de strings com capacidade n , e tamanho de strings m :

- Qual a complexidade (melhor e pior caso) para inserir e buscar um elemento usando hash encadeado com lista? E para listar todos os elementos do hash?
- Qual a complexidade (melhor e pior caso) para inserir e buscar um elemento usando hash encadeado com árvore balanceada (árvore rubro-negra)?

Comparação entre o uso de busca linear, busca binária e Hash

Escreva um programa que leia um número inteiro **n** e em seguida leia **n** pares **p s**, onde **s** indica o nome de um personagem fictício (nome e sobrenome) e **p** indica um número de ordem associado a esse personagem. Depois, leia um número inteiro **m** e mais **m** nomes (nome e sobrenome). Para cada um desses nomes, mostre o número de ordem associado ao personagem.

Escreva três versões para este programa e as compare:

- 1) Use um vetor ou lista e realize buscas lineares
- 2) Use um vetor ou árvore e realize buscas binárias
- 3) Use uma estrutura de Hash encadeado com lista ou árvores

Comparação entre o uso de busca linear, busca binária e Hash

Os casos de teste e as respostas se encontram na página da disciplina: exercício-hash.zip. Trata-se de cinco casos de teste:

- 1 – 5.000 nomes / 1.000 consultas
- 2 – 25.000 nomes / 10.000 consultas
- 3 – 50.000 nomes / 10.000 consultas
- 4 – 50.000 nomes / 25.000 consultas
- 5 – 100.000 nomes / 75.000 consultas

Compare as suas respostas com os gabaritos fornecidos (saida-n.txt)

Trabalho

Comparação entre o uso de busca linear, busca binária e Hash

O que deve ser entregue?

Preparar um relatório de no máximo cinco páginas (seguindo o modelo de artigos da SBC) descrevendo os detalhes relevantes da implementação dos algoritmos (ex: como foi implementada a função de hash) e mostrando a comparação dos tempos de execução.

Além do relatório impresso, trazer no dia da entrega os algoritmos implementados para checagem de autoria.

Atividade Extra

Resolva os seguintes problemas do URI online Judge e determine a complexidade para o pior caso:

- 1861 – O Hall dos Assassinos

(http://www.urionlinejudge.com.br/repository/UOJ_1861.html)

OBS: Listagem de resultados:

Accepted – sua solução foi aceita;

Wrong Answer – o algoritmo produziu uma resposta incorreta para alguma situação;

Time Limit Exceded – seu algoritmo entrou em loop infinito ou demorou muito (rever algoritmo e reduzir a sua complexidade de tempo);

Runtime Error – seu programa fez alguma operação indevida (divisão por zero ou acesso a memória inválida);

Referências

Algoritmos. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Campus.

Algorithms. Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani. McGraw Hill.

Concrete Mathematics: A Foundation for Computer Science (2nd Edition). Ronald L. Graham, Donald E. Knuth, Oren Patashnik. Addison Wesley.

M. R. Garey and D. S. Johnson. 1978. “*Strong*” *NP-Completeness Results: Motivation, Examples, and Implications*. J. ACM 25, 3 (July 1978)