Nota de Aula 3

Bacharelado em Sistemas de Informação | Algoritmos e Estruturas de Dados 2 | Prof. Raimundo Osvaldo

Conteúdo

Mapas e Tabelas de Espalhamento (*Hashing*)

Referências

GOODRICH, Michael T.; TAMASSIA, Roberto. **Estruturas de Dados e Algoritmos em Java**. Porto Alegre: Bookman, 2013

CORMEN, Thomas H. **Algoritmos: Teoria e Prática**. Rio de Janeiro: Elsevier, 2012

Resumo Teórico: Mapas

Um **mapa** permite armazenar elementos que podem ser localizados rapidamente usado chaves. Um mapa armazena um par *chave-valor* (k, v), chamado de **entradas**, onde k é a chave e v é o valor correspondente. O TAD mapa requer que toda chave seja única, e associação da chave com o valor define o mapeamento.

Um mapa é mais apropriado em situações em que cada chave deve ser vista como um **índice** único para seu valor. A chave associada com um objeto pode ser vista como um "endereço" para um objeto. Os mapas podem ser referidos como **armazenamento associativo**, porque a chave associada com um determinado objeto determina sua "localização" na estrutura de dados.

O TAD Mapa

Como um TAD, um **mapa M** suporta os seguintes métodos:

Método	Descrição
`size()`	Retorna o número de entradas de **M**
`isEmpty()`	Teste se **M** está vazio
`get(`k`)`	Se **M** contém uma entrada e com chave igual a k, então retorna o valor de e, senão retorna **`null`**
`put(`k, v`)`	Se **M** não tem uma entrada com chave igual a k, então adiciona a entrada (k, v) em **M** e retorna **`null`**; senão, substitui com o valor de v o valor existente na entrada com chave k e retorna o valor antigo
`remove(`k`)`	Remove a entrada de **M** com chave igual a k, e retorna seu valor; se **M** não possui entrada com chave igual a k, então retorna **`null`**
`keySet()`	Retorna uma coleção iterável contendo todas as chaves armazenadas em **M**
`values()`	Retorna uma coleção contendo todos os valores associados com as chaves armazenadas em **M**
`entrySet()`	Retorna uma coleção contendo todas as entradas (chave-valor) de **M**

Um Exemplo Ilustrativo

Vamos analisar a execução de uma série de operações em um mapa inicialmente vazio que armazena chaves inteiras e valores de um único caractere:

Operação	Saída	Mapa
`isEmpty()`	`true`	$\emptyset$
`put(5, A)`	`null`	$\{(5,A)\}$
`put(7, B)`	`null`	$\{(5,A),(7,B)\}$
`put(2, C)`	`null`	$\{(5,A),(7,B),(2,C)\}$
`put(8, D)`	`null`	$\{(5,A),(7,B),(2,C),(8,D)\}$
`put(2, E)`	`C`	$\{(5,A),(7,B),(2,E),(8,D)\}$
`get(7)`	`B`	$\{(5,A),(7,B),(2,E),(8,D)\}$
`get(4)`	`null`	$\{(5,A),(7,B),(2,E),(8,D)\}$
`get(2)`	`E`	$\{(5,A),(7,B),(2,E),(8,D)\}$
`size()`	`4`	$\{(5,A),(7,B),(2,E),(8,D)\}$
`remove(5)`	`A`	$\{(7,B),(2,E),(8,D)\}$
`remove(2)`	`E`	$\{(7,B),(8,D)\}$
`get(2)`	`null`	$\{(7,B),(8,D)\}$
`isEmpty()`	`true`	$\{(7,B),(8,D)\}$
`entreySet()`	$\{(7,B),(8,D)\}$	$\{(7,B),(8,D)\}$
`keySet()`	$\{7,8\}$	$\{(7,B),(8,D)\}$
`values()`	$\{B,D\}$	$\{(7,B),(8,D)\}$

Uma Implementação

Uma forma simples de implementar um mapa é armazenar suas n entradas em uma sequência S, implementada como uma lista duplamente encadeada. A execução dos métodos fundamentais, `get(`k`)`, `put(`k,v`)` e remove(k), envolve buscas simples sobre S procurando por uma entrada com chave k.

Algoritmo **`get(k)`**

Entrada: uma chave k

Saída: o valor para a chave k em M, ou `null` se não existir chave k em M

```
inicio
   para cada posição p em S.positions() faça
      se p.element().getKey() = k então
         retorna p.element().getValue()
      fimse
   fimpara
   retorna null
fimalgoritmo
```

Algoritmo **put(k, v)**

Entrada: um par chave-valor (k, v)
Saída: o antigo valor associado para a chave k em M ou `null` se é uma nova chave

```
inicio
   para cada posição p em S.positions() faça
      se p.element().getKey() = k então
         t ← p.element().getValue()
         S.set(p, (k,v))
         retorna t
      fimse
   fimpara
   S.addLast((k,v))
   n ← n + 1
   retorna null
fimalgoritmo
```

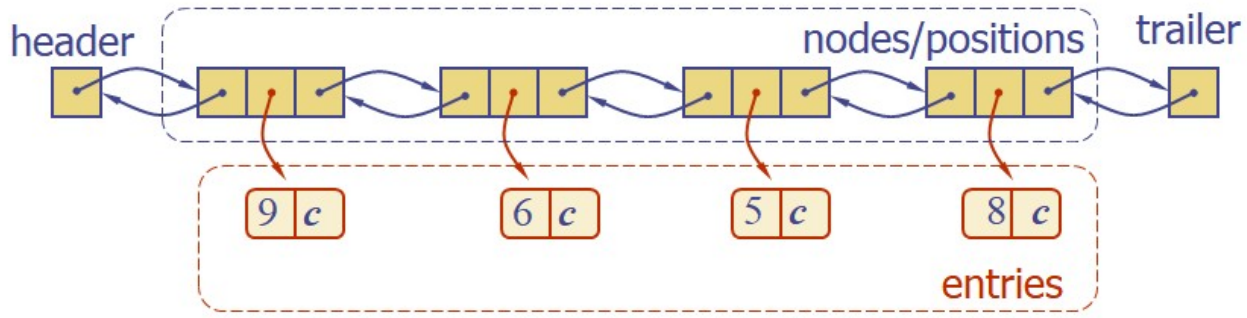
Algoritmo **remove(k)**

Entrada: uma chave k
Saída: o valor removido para a chave k em M, ou `null` se não existir chave k em M

```
inicio
   para cada posição p em S.positions() faça
      se p.element().getKey() = k então
         t ← p.element().getValue()
         S.remove(p)
         n ← n - 1
         retorna t
      fimse
   fimpara
   retorna null
fimalgoritmo
```

Atividades Propostas

1. Implemente um TAD Mapa conforme interface apresentada. Use a Linguagem Java. Você pode usar uma lista duplamente encadeada para representação interna do TAD, conforme conjunto de códigos fornecido.



O TAD Lista: tem a seguinte lista de métodos para uma lista S

`first()`	Retorna a posição do primeiro elemento de S; ocorre um erro se S está vazio
`last()`	Retorna a posição do primeiro elemento de S; ocorre um erro se S está vazio

`prev(p)`	Retorna a posição do elemento de S que precede o que se encontra na posição p; ocorre um erro se p for a primeira posição
`next(p)`	Retorna a posição do elemento de S que segue o que se encontra na posição p; ocorre um erro se p for a última posição
`set(p, e)`	Substitui o elemento que se encontra na posição p por e, retornando o elemento que se encontrava antes na posição p
`addFirst(e)`	Insere o novo elemento e como primeiro elemento
`addLast(e)`	Insere o novo elemento e como último elemento
`addBefore(p, e)`	Insere o novo elemento e antes da posição p
`addAfter(p, e)`	Insere o novo elemento e após a posição p
`remove(e)`	Remove e retorna o elemento na posição p, invalidando esta posição p

Os métodos permitem fazer referência a posições relativas de uma lista, começando no início ou no fim e deslocando-se por incremento para cima ou para baixo. As posições podem ser intuitivamente entendidas como sendo os nodos da lista, porém, observa-se que não existem referências específicas a objetos nodo.

2. Qual é o pior caso de tempo de execução para inserção de $\boldsymbol{n}$ elementos chave-valor em um mapa M inicialmente vazio que é implementado como uma sequência?

Resumo Teórico: Tabelas *Hash*

Uma tabela de hashing ou de espalhamento é uma estrutura de dados eficaz para implementar dicionários. Um dicionário funciona com chaves que são associadas a elemento e tais chaves são consideradas "endereços" dos elementos. Uma tabela de hash generaliza a noção mais simples de um arranjo comum.

Em geral, uma tabela de hash consiste em dois componentes principais: um **arranjo de buckets** e uma **função de hash**.

Arranjos de Buckets

Um **arranjo de buckets** para uma tabela de hash é um arranjo A de tamanho N, em que cada célula de A é considerada como um *bucket* (um conteiner para pares chave-elemento), e o inteiro N determina a capacidade do arranjo. Se as chaves forem inteiros bem distribuídos no intervalo $[0, N-1]$, esse arranjo é tudo o que precisamos. Um elemento e com chave k é simplesmente inserido no *bucket* $A[k]$.

Se as chaves são inteiros únicos no intervalo $[0, N-1]$, então cada bucket armazenará no máximo uma entrada. Deste modo, pesquisas, inserções e remoçoes levarão tempo $O(1)$. **Desvantagens**: espaço utlizado proporcional a N; exigência de as chaves serem inteiros no intervalo $[0, N-1]$.

Funções de Hash

Mapeia cada chave k para uma posição na tabela de hash, onde o tamanho N da tabela normalmente é muito menor que o número de elementos do conjunto universo U de chaves para as posições. A função de hash reduz o número de índices do arranjo e, consequentemente, o tamanho do arranjo.

$$h: U \rightarrow \{0, 1, \ldots, N-1\}$$

Dizemos que um elemento com chave k se espalha (hashes) até a posição $h(k)$; dizemos também que $h(k)$ é o ***valor hash*** da chave k.

Com a função de hash, duas chaves podem ser mapeadas para a mesma posição: **colisão**. Devemos escolher uma função de hash mais adequada para minimizar a ocorrência de colisões. Quando estas

ocorrem, há técnicas para resolvê-las. Por razões práticas, é desejável que uma função de hash seja rápida e fácil de computar.

Seguindo as convenções do Java, visualiza-se a evolução de uma função de hash, $h(k)$, consistindo em duas ações: 1) mapeamento da chave k para um número inteiro (**código de hash**). Este inteiro não precisa estar no intervalo $[0, N-1]$ e pode inclusive ser negativo; e 2) mapeamento do código hash para um inteiro em um intervalo de índices $[0, N-1]$ de um arranjo de buckets (**função de compressão**).

Código de Hash em Java

A classe `Object` definida em Java é equipada com um método padrão `hashCode()` para mapear as instâncias de um objeto em um inteiro que é a "representação" do objeto. Essa função retorna um **int** de 32 bits. A não ser que seja especificamente sobrescrito, esse método é herdado por todo objeto em um programa Java. Cuidado ao usar a versão padrão, pois ela pode ser uma interpretação inteira da posição do objeto na memória. Isso não funciona bem com Strings, por exemplo, pois Strings com os mesmos caracteres podem ocupar posições diferentes na memória. Para vários tipos de dados comuns há métodos para associar códigos hash a esses tipos de dados.

O Método da divisão

Uma função simples que mapeia um inteiro i para $i \ mod \ N$, onde N é o tamanho de um arranjo de buckets, sendo um inteiro positivo fixo. Se escolhemos N um número primo, esssa função de compressão ajuda a espalhar a distribuição dos valores.

O Método MAD

Uma função de compressão mais sofisticada que ajuda a eliminar padrões mais repetitivos em um conjunto de chaves inteiras. MAD = método de Multiplicação, Adição e Divisão. Mapeia um inteiro i para

$$[(ai + b) \ mod \ p] \ mod \ N$$

onde N é o tamanho do arranjo de buckets, i é um número primo maior que N e a e b são inteiros escolhidos aleatoriamente em um intervalo $[0, p-1]$, com $a > 0$.

Atividades Propostas

3. Pesquise sobre códigos hash polinomiais e códigos hash com shift. Qual seria um bom código hash para um número de identifcação de veículo que é uma cadeia de caracteres representando números e letras no formato "9X9XX99X9XX999999" onde um 9 representa um dígito e um X representa uma letra?
4. Desenhe a tabela hash com 11 elementos, que resulta a partir do uso da função de hash, $\boldsymbol{h(i) = (3i + 5) \ mod \ 11}$, para colocar as chaves 12, 44, 13, 88, 23, 94, 11, 39, 20, 16 e 5. Indique quando ocorre colisão, porém não precisa tratar ainda.