45EST – Algoritmos e Estruturas de Dados

Mapas

Prof. Marcelo de Souza

UDESC Ibirama
Bacharelado em Engenharia de Software

marcelo.desouza@udesc.br
Versão compilada em 13 de agosto de 2020

Leitura obrigatória:

- Capítulo 10 de Goodrich et al. [2014] – Mapas, tabelas hash e skip lists.

Leitura complementar:

- Capítulo 3 (3.1) de Sedgewick and Wayne [2011] – Tabelas de símbolos.
- Capítulo 8 de Preiss [2001] – Dispersão, tabelas de dispersão e tabelas de espalhamento.

Mapas

Ideia geral:

- Armazena entradas chave/valor, onde as chaves são únicas.
- Também chamados de **dicionários**.
- Chave funciona como índice da estrutura.
- Aplicações:
  - Armazenar alunos pela sua matrícula.
  - Armazenar automóveis pela sua placa.
- Operações: `get(k)`, `put(k, v)`, `remove(k)`.
- Funcionamento (e implementação) baseado em tabela.

Exemplo (mapa/dicionário para o índice de um livro):

Chave (termo)	Valor (páginas)
herança	12, 23, 78, 81
polimorfismo	66, 80, 93
encapsulamento	10, 24, 26, 33
associação	41, 43, 54
agregação	48, 60, 62
composição	48, 61, 62

Interface `Map`:

```
1  public interface Map<K, V> {
2     int size();
3     boolean isEmpty();
4     V get(K key);
5     V put(K key, V value);
6     V remove(K k);
7  }
```

Implementação não ordenada – `UnsortedTableMap`:

```
1  public class UnsortedTableMap<K,V> implements Map<K,V> {
2
3     protected static class MapEntry<K,V> implements Entry<K,V> {
4        private K k;
5        private V v;
6
7        public MapEntry(K key, V value) {
8           k = key;
9           v = value;
```

```
10          }
11
12          public K getKey() { return k; }
13          public V getValue() { return v; }
14
15          protected void setKey(K key) { k = key; }
16          protected V setValue(V value) {
17             V old = v;
18             v = value;
19             return old;
20          }
21
22          public String toString() { return "<" + k + ", " + v + ">"; }
23       }
24
25       private ArrayList<MapEntry<K,V>> table = new ArrayList<>();
26
27       public int size() { return table.size(); }
28       public boolean isEmpty() { return size() == 0; }
29
30       private int findIndex(K key) {
31          int n = table.size();
32          for (int j=0; j < n; j++)
33          if (table.get(j).getKey().equals(key))
34             return j;
35          return -1;
36       }
37
38       public V get(K key) {
39          int j = findIndex(key);
40          if (j == -1) return null;
41          return table.get(j).getValue();
42       }
43
44       public V put(K key, V value) {
45          int j = findIndex(key);
46          if (j == -1) {
47             table.add(new MapEntry<>(key, value));
```

```
48            return null;
49         } else
50            return table.get(j).setValue(value);
51      }
52
53      public V remove(K key) {
54         int j = findIndex(key);
55         int n = size();
56         if (j == -1) return null;
57         V answer = table.get(j).getValue();
58         if (j != n - 1)
59            table.set(j, table.get(n-1));
60         table.remove(n-1);
61         return answer;
62      }
63   }
```

Comentários:

- O mapa implementa sua estrutura interna para a entrada e utiliza o `ArrayList` para armazenar a lista de entradas.
- O método `findIndex` devolve o índice da entrada com a respectiva chave, ou -1, caso não exista entrada com a chave buscada.
- O método `put` substitui o valor da chave, caso ela exista, ou insere uma nova entrada, caso contrário.
- O método `remove` substitui o elemento a ser removido pela última entrada, e então remove o último elemento. Isso faz com que não seja necessária realocação dos elementos no vetor.

Análise de complexidade:

- Todas as operações tem complexidade linear $O(n)$, dada a necessidade de buscar a entrada.

Mapa ordenado

- Permite aplicar uma busca de entrada mais eficiente – **busca binária**.
- Permite implementar outras operações de maneira eficiente:
  - `firstEntry()`: retorna a entrada com menor chave.
  - `lastEntry()`: retorna a entrada com maior chave.
  - `ceilingEntry(k)`: retorna a entrada com a menor chave $\geq k$.
  - `floorEntry(k)`: retorna a entrada com a maior chave $\leq k$.
  - `higherEntry(k)`: retorna a entrada com a menor chave $> k$.
  - `lowerEntry(k)`: retorna a entrada com a maior chave $< k$.

Interface `SortedMap`:

```
1  public interface SortedMap<K,V> extends Map<K,V> {
2     Entry<K,V> firstEntry();
3     Entry<K,V> lastEntry();
4     Entry<K,V> ceilingEntry(K key);
5     Entry<K,V> floorEntry(K key);
6     Entry<K,V> lowerEntry(K key);
7     Entry<K,V> higherEntry(K key);
8  }
```

Implementação da `SortedTableMap`:

```
1  public class SortedTableMap<K,V> implements SortedMap<K,V> {
2
3     protected static class MapEntry<K,V> implements Entry<K,V> {
4        private K k;
5        private V v;
6
7        public MapEntry(K key, V value) {
8           k = key;
```

```
9           v = value;
10       }
11
12       public K getKey() { return k; }
13       public V getValue() { return v; }
14
15       protected void setKey(K key) { k = key; }
16       protected V setValue(V value) {
17          V old = v;
18          v = value;
19          return old;
20       }
21
22       public String toString() { return "<" + k + ", " + v + ">"; }
23    }
24
25
26    private Comparator<K> comp;
27    private ArrayList<MapEntry<K,V>> table = new ArrayList<>();
28
29    protected SortedTableMap(Comparator<K> c) {
30       comp = c;
31    }
32
33    protected SortedTableMap() {
34       this(new DefaultComparator<K>());
35    }
36
37    public int size() { return table.size(); }
38    public boolean isEmpty() { return size() == 0; }
39
40    protected boolean checkKey(K key) {
41       try {
42          return (comp.compare(key,key)==0);
43       } catch (ClassCastException e) {
44          throw new IllegalArgumentException("Incompatible key");
45       }
46    }
```

```
47
48      private int findIndex(K key) { return findIndex(key, 0,
        ↪  table.size() - 1); }
49
50      private int findIndex(K key, int low, int high) {
51         if (high < low) return high + 1;
52         int mid = (low + high) / 2;
53         int result = comp.compare(key, table.get(mid).getKey());
54         if (result == 0)
55            return mid;
56         else if (result < 0)
57            return findIndex(key, low, mid - 1);
58         else
59            return findIndex(key, mid + 1, high);
60      }
61
62      public V get(K key) {
63         checkKey(key);
64         int j = findIndex(key);
65         if (j==size() || comp.compare(key, table.get(j).getKey())!=0)
66            return null;
67         return table.get(j).getValue();
68      }
69
70      public V put(K key, V value) {
71         checkKey(key);
72         int j = findIndex(key);
73         if (j<size() && comp.compare(key, table.get(j).getKey())==0)
74            return table.get(j).setValue(value);
75         table.add(j, new MapEntry<K,V>(key,value));
76         return null;
77      }
78
79      public V remove(K key) {
80         checkKey(key);
81         int j = findIndex(key);
82         if (j==size() || comp.compare(key, table.get(j).getKey())!=0)
83            return null;
```

```
84          return table.remove(j).getValue();
85      }
86
87      private Entry<K,V> safeEntry(int j) {
88          if (j < 0 || j >= table.size()) return null;
89          return table.get(j);
90      }
91
92      public Entry<K,V> firstEntry() {
93          return safeEntry(0);
94      }
95
96      public Entry<K,V> lastEntry() {
97          return safeEntry(table.size()-1);
98      }
99
100     public Entry<K,V> ceilingEntry(K key) {
101         return safeEntry(findIndex(key));
102     }
103
104     public Entry<K,V> floorEntry(K key) {
105         int j = findIndex(key);
106         if (j == size() || ! key.equals(table.get(j).getKey()))
107             j--;
108         return safeEntry(j);
109     }
110
111     public Entry<K,V> lowerEntry(K key) {
112         return safeEntry(findIndex(key) - 1);
113     }
114
115     public Entry<K,V> higherEntry(K key) {
116         int j = findIndex(key);
117         if (j < size() && key.equals(table.get(j).getKey()))
118             j++;
119         return safeEntry(j);
120     }
121 }
```

Comentários:

- A classe exige um comparador ou um tipo comparável.
- O método `findIndex` implementa uma **busca binária**.
  - Encontra entrada em $O(\log n)$, caso exista.
  - Caso contrário, retorna posição onde entrada deve ser armazenada.
    - Observe que a chamada recursiva ocorre com `mid` $\pm$ `1`.
- Método `safeEntry` verifica se o acesso ao vetor é válido.
- O `ceiling` é obtido pela busca binária.
- O `floor` é obtido pela busca binária, caso existente, ou a posição anterior, caso contrário.
- O `lower` é obtido pela posição anterior ao retorno da busca binária.
- O `higher` é obtido pela busca binária, caso não existente, ou a posição posterior, caso contrário.
- Importante: a chave precisa implementar o método `equals`.

Análise de complexidade:

- Método `get` executa em $O(\log n)$.
- Método `put` executa em $O(\log n)$ na substituição e $O(n)$ na inserção de um novo elemento.
- Método `remove` executa em $O(n)$.
- Os métodos `firstEntry` e `lastEntry` executam em $O(1)$.
- Os demais métodos executam em $O(\log n)$.

Atividades

1. Leia a respeito das estratégias de *hashing* e da estrutura de dados chamada *tabela hash*. Essa estrutura armazena mapas e fornece operações em tempo constante. Como isso é possível? Veja as formas de implementação detalhadas em Goodrich et al. [2014].

2. Desenvolva um sistema para gerenciamento de restaurantes para um sistema de recomendação. O usuário escolherá uma categoria (japonês, massas, churrasco ou lanches) e informará o valor que deseja pagar pela refeição. O sistema então recomendará o melhor restaurante para as opções do cliente. Para isso, restaurantes de diferentes categorias devem ser armazenados em mapas distintos, cuja chave consiste na tupla `<nota, preço médio>`, enquanto o valor armazena as demais informações do restaurante, como nome, horário de funcionamento e endereço. Crie rotinas para a criação de novos restaurantes, remoção de restaurantes da base de dados e consulta. Implemente o sistema utilizando mapas ordenados e não-ordenados. Compare o desempenho das operações.

3. Faça os seguintes exercícios de Goodrich et al. [2014]:

R-10.1: Qual é o tempo de execução no pior caso para inserir n pares chave-valor em um mapa inicialmente vazio, implementado pela classe `UnsortedTableMap`?

R-10.2: Reimplemente a classe `UnsortedTableMap` usando uma lista posicional (`PositionalList`) ao invés de um `ArrayList`.

R-10.3: O uso de valores `null` em um mapa é problemático, uma vez que não é possível diferenciar se um retorno `null` do método `get(k)` representa um valor legítimo de uma entrada `(k, null)`, ou representa que a chave `k` não foi encontrada. A interface `java.util.Map` inclui um método booleano `containsKey(k)` que resolve essa ambiguidade. Implemente este método na classe `UnsortedTableMap`.

R-10.18: Qual é o tempo de execução assintótico do pior caso para realizar n remoções de uma instância de `SortedTableMap` que contém inicialmente $2n$ entradas?

R-10.19: Implemente o método `containKey(k)` para a classe `SortedTableMap`.

R-10.20: Descreva como uma lista ordenada implementada como uma lista duplamente encadeada poderia ser usada para implementar um mapa ordenado.

R-10.21: Considere a variante abaixo do método `findIndex` para a classe `SortedTableMap`.

```
1  private int findIndex(K key, int low, int high) {
2     if(high < low) return high + 1;
3     int mid = (low + high) / 2;
4     if(compare(key, table.get(mid).getKey()) < 0)
5        return findIndex(key, low, mid - 1);
6     else
7        return findIndex(key, mid + 1, high);
8  }
```

Este método sempre produz o mesmo resultado que a versão original? Justifique sua resposta.

C-10.33: Considere o objetivo de adicionar uma entrada `(k, v)` em um mapa somente se não existir outra entrada com a mesma chave `k`. Para um mapa M sem valores `null`, isso pode ser feito da seguinte forma:

```
1  if(M.get(k) == null)
2     M.put(k, v);
```

Apesar de atingir o objetivo, esta estratégia é ineficiente, uma vez que gasta tempo para verificar que não existe entrada com a chave `k`, e novamente para buscar a posição de inserção da nova entrada. Para evitar isso, algumas implementações de mapas suportam um

método `pullAbsent(k, v)`, que realiza a inserção assim que identifica a não existência de entrada com a chave `k`. Forneça a implementação deste método para a classe `UnsortedTableMap`.

C-10.45: Desenvolva uma versão de `UnsortedTableMap` baseada em (com conhecimento de) localização, de tal forma que a operação `remove(e)` para uma entrada `e` existente possa ser implementada em tempo $O(1)$.

Referências

Goodrich, M. T., Tamassia, R., and Goldwasser, M. H. (2014). *Data structures and algorithms in Java*. John Wiley & Sons, 6th edition.

Preiss, B. R. (2001). *Estruturas de dados e algoritmos: padrões de projetos orientados a objetos com Java*. Campus.

Sedgewick, R. and Wayne, K. (2011). *Algorithms*. Addison-Wesley Professional.