

Aula 07: 05/04/2017

Tópico

Tabelas de símbolos

Leitura

[Tabelas de símbolos \(PF\)](#), [Elementary Symbol Tables \(S&W\)](#), [slides \(S&W\)](#)

Vídeo

[Elementary Symbol Table, S&W](#)

Tabela de Símbolos

Uma **tabela de símbolos** (= *symbol table*) é um ADT que consiste em um conjunto de itens, sendo cada item um par (*chave, valor* ou *key-value*), munido de duas operações fundamentais:

- `put()`, que insere um novo item no conjunto, e
- `get()`, que busca o valor associado a uma dada chave.

Convenções sobre TSs:

- não há chaves repetidas (as chaves são duas a duas distintas),
- `null` nunca é usado como chave,
- `null` nunca é usado como valor associado a uma chave.

API

```
public class ST<Key,Value>
-----
    ST()                cria uma tabela de símbolos vazia
    void put(Key key,   Value val)  insere o item (key,val) nesta tabela
    Value get(Key key)  busca o valor associado a key
    boolean isEmpty()   esta tabela está vazia?
    boolean contains(Key key) a chave key está nesta tabela?
    Iterable<Key> keys() lista todas as chaves desta tabela
```

Como estimar o desempenho de uma implementação de TS?

Durante a execução de `get(k)` ou `put(k,v)`, uma chave da TS é tocada quando comparada com `k`. O consumo de tempo é proporcional ao *número de chaves tocadas*

SequentialSearchST.java

Esta TS é não orientada. O único método exigido de Key é equals().

```
public class SequentialSearchST<Key,Value> {

    private Node first;

    // nó da lista ligada
    private class Node {
        private Key key;
        private Value val;
        private Node next;
        public Node(Key key, Value val, Node next) { // construtor
            this.key = key;
            this.val = val;
            this.next = next;
        }
    }

    public Value get(Key key) {
        for (Node x = first; x != null; x = x.next)
            if (key.equals(x.key))
                return x.val; // acerto
        return null; // falha
    }

    public void put(Key key, Value val) {
        // tabelas de símbolo não podem ter chaves repetidas
        for (Node x = first; x != null; x = x.next)
            if (key.equals(x.key)) {
                x.val = val;
                return;
            } // acerto: atualiza val
        first = new Node(key, val, first); // falha: acrescenta novo nó
    }

    public void delete(Key key) {
        // quando invocar uma exceção, indica o método que "gritou"
        if (key == null) throw new IllegalArgumentException("argument to delete() is null");

        // chama o método que faz de fato a remoção;
        first = delete(first, key);
    }

    // delete key in linked list beginning at Node x
    // warning: function call stack too large if table is large
    private Node delete(Node x, Key key) {
        if (x == null) return null;
    }
}
```

```

    if (key.equals(x.key)) {
        n--;
        return x.next;
    }
    x.next = delete(x.next, key);
    return x;
}

public Iterable<Key> keys() {
    // cria e retorna uma coleção iterável
    Queue<Key> queue = new Queue<Key>();
    for (Node x = first; x != null; x = x.next)
        queue.enqueue(x.key);
    return queue;
}
}

```

Desempenho do SequentialSearchST

- `get()`: n chaves tocadas
- `put()`: n chaves tocadas

A busca e inserção são muito lentas. A inserção é lenta em virtude da não inserir chaves repetidas na TS.

Inserir n chaves distintas numa TS inicialmente vazia implementada em lista ligada consome $1+2+\dots+n \sim n^2/2$ unidades de tempo no pior caso.

Espaço gasto por SequentialSearchST

Cada nó tem 3 referências

```

public Node(Key key, Value val, Node next) { // construtor
    this.key = key;
    this.val = val;
    this.next = next;
}

```

Cada referência gasta 8 bytes. Assim, um nó gasta 32 bytes com referências. Gastamos ainda com cada objeto 16 bytes de *overhead*: referência para a classe do objeto, informação para o coletor de lixo.

Assim cada nó gasta 48 bytes. Uma ST com n itens gasta, portanto, $48n$ mais o espaço necessário para armazenarmos os itens (=pares key-val)

Espaço gasto por LinkedListST

O mesmo que SequentialSearchST: $48n$ mais o espaço necessário para os pares key-val

ArrayST.java

```
public class ArrayST<Key, Value> {
    private static final int INIT_SIZE = 8;

    private Value[] vals;    // symbol table values
    private Key[] keys;     // symbol table keys
    private int n = 0;      // number of elements in symbol table

    public ArrayST() {
        keys = (Key[]) new Object[INIT_SIZE];
        vals = (Value[]) new Object[INIT_SIZE];
    }

    // return the number of key-value pairs in the symbol table
    public int size() {
        return n;
    }

    // is the symbol table empty?
    public boolean isEmpty() {
        return size() == 0;
    }

    // resize the parallel arrays to the given capacity
    private void resize(int capacity) {
        Key[] tempk = (Key[]) new Object[capacity];
        Value[] tempv = (Value[]) new Object[capacity];
        for (int i = 0; i < n; i++)
            tempk[i] = keys[i];
        for (int i = 0; i < n; i++)
            tempv[i] = vals[i];
        keys = tempk;
        vals = tempv;
    }

    // insert the key-value pair into the symbol table
    public void put(Key key, Value val) {

        // to deal with duplicates
        delete(key);

        // double size of arrays if necessary
        if (n >= vals.length) resize(2*n);

        // add new key and value at the end of array
        vals[n] = val;
        keys[n] = key;
        n++;
    }
}
```

```

public Value get(Key key) {
    for (int i = 0; i < n; i++)
        if (keys[i].equals(key)) return vals[i];
    return null;
}

public Iterable<Key> keys() {
    Queue<Key> queue = new Queue<Key>();
    for (int i = 0; i < n; i++)
        queue.enqueue(keys[i]);
    return queue;
}

// remove given key (and associated value)
public void delete(Key key) {
    for (int i = 0; i < n; i++) {
        if (key.equals(keys[i])) {
            keys[i] = keys[n-1];
            vals[i] = vals[n-1];
            keys[n-1] = null;
            vals[n-1] = null;
            n--;
            if (n > 0 && n == keys.length/4) resize(keys.length/2);
            return;
        }
    }
}

```

Desempenho ArrayST

- `get()`: n chaves tocadas
- `put()`: n chaves tocadas

A busca e inserção são muito lentas. A inserção é lenta em virtude da não inserir chaves repetidas na TS.

Inserir n chaves distintas numa TS inicialmente vazia implementada em lista ligada consome $1+2+\dots+n \sim n^2/2$ unidades de tempo no pior caso.

Espaço gasto por ArrayST

Supondo que temos uma tabela de símbolo com n itens. Gastaremos $16 + 4 + 8n$ bytes com o vetor `keys` e $16 + 4 + 8n$ bytes com o vetor `vals`.

Portanto, o total gasto é $40 + 16n$ bytes mais o espaço gasto com os itens (= pares `key-value`). Bem, devido ao redimensionamento, é possível que o espaço gasto seja até $40 + 16 \times 2^k$, onde k é o menor inteiro tal que $2^k \geq n$. Isso pode chegar a ser $40 + 32n$, no pior caso.

BinarySearchST.java

Esta TS é ordenada: utilizamos o método `compareTo()`. No que é exigido que `Key` extends `Comparable<Key>`.

```
public class BinarySearchST <Key extends Comparable<Key>, Value> {

    private Key[] keys;
    private Value[] vals;
    private int n;

    public BinarySearchST(int capacity) { // construtor
        keys = (Key[]) new Comparable[capacity];
        vals = (Value[]) new Object[capacity];
    }

    public Value get(Key key) {
        int i = rank(key);
        if (i < n && keys[i].compareTo(key) == 0)
            return vals[i];
        else return null;
    }

    public void put(Key key, Value val) {
        int i = rank(key);
        if (i < n && keys[i].compareTo(key) == 0) {
            // acerto de busca
            vals[i] = val;
            return;
        }
        // falha de busca
        for (int j = n; j > i; j--) {
            keys[j] = keys[j-1];
            vals[j] = vals[j-1];
        }
        keys[i] = key;
        vals[i] = val;
        n++;
    }

    // devolve o posto de key
    public int rank(Key key) {
        int lo = 0, hi = N-1;
        while (lo <= hi) {
            int mid = lo + (hi - lo) / 2;
            int cmp = key.compareTo(keys[mid]);
            if (cmp < 0) hi = mid - 1;
            else if (cmp > 0) lo = mid + 1;
            else return mid;
        }
        return lo;
    }
}
```

```
}  
}
```

Desempenho BinarySearchST

- `get()`: no máximo $\lg n$ comparações
- `put()`: no máximo $\lg n$ comparações e no máximo n deslocamentos

Espaço gasto por BinarySearchST

O mesmo que o espaço gasto com `ArrayST`.

Resumo

Resumo de custos das implementações básicas `SequentialSearchST.java` e `BinarySearchST.java`

	pior caso		caso medio		operações ordenação
	get()	put()	get()	put()	eficientes

busca sequencial (lista ligada ou vetor)	n	n	n/2	n	não
busca binária (vetor ordenado)	$\lg(n)$	2n	$\lg n$	n	sim

Resultados experimentais

```
linked-list> java Driver ../testes/les-miserables.txt  
Criando a ST com as palavras do arquivo '../testes/les-miserables.txt' ...  
ST criada em 92.785 segundos  
ST contém 26764 itens  
Início da consulta interativa. Tecler ctrl+D encerrar
```

```
sorted-linked-list> java Driver ../testes/les-miserables.txt  
Criando a ST com as palavras do arquivo '../testes/les-miserables.txt' ...  
ST criada em 84.192 segundos  
ST contém 26764 itens  
Início da consulta interativa. Tecler ctrl+D encerrar  
>>>
```

```
array> java Driver ../testes/les-miserables.txt  
Criando a ST com as palavras do arquivo '../testes/les-miserables.txt' ...  
ST criada em 60.084 segundos  
ST contém 26764 itens  
Início da consulta interativa. Tecler ctrl+D encerrar  
>>>
```

```
binary-search> java Driver ../testes/les-miserables.txt
Criando a ST com as palavras do arquivo '../testes/les-miserables.txt' ...
ST criada em 1.283 segundos
ST contém 26764 itens
Início da consulta interativa. Tecla ctrl+D encerrar
>>>
```

```
binary-search-tree> java Driver ../testes/les-miserables.txt
Criando a ST com as palavras do arquivo '../testes/les-miserables.txt' ...
ST criada em 0.687 segundos
ST contém 26764 itens
Início da consulta interativa. Tecla ctrl+D encerrar
>>>
```