Búsqueda

Searching

Problemas de búsqueda

- La búsqueda es un problema fundamental que aparece en innumerables aplicaciones de procesamiento de información.
- Ejemplos:
  - Buscar en una tabla de contenido
  - Buscar en un diccionario
  - Buscar la web
  - Buscar en una base de datos

Abstracción del problema de búsqueda

- Se identifica la existencia de unas *llaves* con respecto a las cuales se hace la búsqueda y uno *valor* asociado a cada llave, que es el resultado de la búsqueda.
- Se define el ADT *tabla de símbolos* para modelar esta estructura. Las llaves son los símbolos y a cada llave se asocia un valor.

API de la tabla de símbolos

class ST<Key, Value>		
void	put(key, value)	Asigna un valor a una llave
Value	get(key)	Busca una llave y retorna el valor asociado
void	delete(key)	Borrar una llave y su valor
boolean	contains(key)	Contiene una llave?
boolean	isEmpty()	Está vacía?
int	size()	Número de llaves
Iterable<Key>	keys()	Iterador sobre las llaves

Aclaraciones sobre la API

- No se permiten llaves duplicadas. `put(key,value)` en una llave repetida sobre-escribe el valor.
- No se permiten llaves `null`.
- Un valor `null` significa que la llave fue borrada.
- El borrado puede utilizar estrategias perezosa o proactiva (*lazy* / *eager*).

Tablas de símbolos ordenadas

- Cuando las llaves implementan la interfaz Comparable, se pueden hacer consultas que dependen del orden de las llaves: Por rangos, menores, mayores.
- Ejemplos
  - Libreta de contactos
  - Diccionario/Enciclopedia
  - Sesiones (cookies)

API tabla de símbolos ordenada

class ST<Key extends Comparable<Key>, Value>		
Key	min()	Retorna el valor de la llave menor
Key	max()	Retorna el valor de la mayor llave
Key	floor(key)	El mayor por debajo de la llave
Key	ceil(key)	El menor por encima de la llave
int	rank(key)	Número de llaves menores a key
Key	select(int k)	Valor en la posición k
int	size(Key lo, Key hi)	Número de llaves en un rango
Iterable<Key>	keys(Key lo, Key hi)	Iterador de llaves en un rango

Modelo de costo

- Las operaciones de búsqueda dependen en su ciclo más interno de operaciones de comparación (`.compareTo, .equals`).
- Como regla general se contabilizan las comparaciones.
- En algunos casos no se usan comparaciones, se contabilizan los accesos al arreglo.

Cliente ejemplo de la tabla de símbolos

```
public static void main(String[] args) {
        int distinct = 0, words = 0;
        int minlen = Integer.parseInt(args[0]);
        ST<String, Integer> st = new ST<String, Integer>();
        // compute frequency counts
        while (!StdIn.isEmpty()) {
            String key = StdIn.readString();
            if (key.length() < minlen) continue;
            words++;
            if (st.contains(key)) {
                st.put(key, st.get(key) + 1);
            }
            else {
                st.put(key, 1);
                distinct++;
            }
        }
```

Código fuente

Implementación de ST

Características del cliente

- Operaciones `get`/`put` son frecuentes y mezcladas.
- El número de llaves a manejar es elevado.
- Es común tener un mayor número de búsquedas que inserciones.

Implementaciones elementales

- Mediante listas no ordernadas
- Con arreglos ordenados

Implementación con lista no ordenada

```
public class SequentialSearchST<Key, Value> {
    private int N;
    private Node first;
    private class Node {
        private Key key;
        private Value val;
        private Node next;
        public Node(Key key, Value val, Node next)  {
            this.key  = key;
            this.val  = val;
            this.next = next;
        }
    }
    public Value get(Key key) {
        if (key == null) throw new NullPointerException("argument to get() is null");
        for (Node x = first; x != null; x = x.next) {
            if (key.equals(x.key))
                return x.val;
        }
        return null;
    }
    public void put(Key key, Value val) {
        if (key == null) throw new NullPointerException("first argument to put() is null");
        if (val == null) {
            delete(key);
            return;
        }
        for (Node x = first; x != null; x = x.next) {
            if (key.equals(x.key)) {
                x.val = val;
                return;
            }
        }
        first = new Node(key, val, first);
        N++;
    }
}
```

Código fuente

Eficiencia de la lista no ordenada

Proposiciones

- Inserciones requieren N comparaciones.
- Búsquedas fallidas (*search misses*) requieren N comparaciones.
- Insertar N llaves distintas en una lista inicialmente vacía requiere ~$N^2/2$ comparaciones.
- Búsquedas éxitosas aleatorias (*random search hits*) requieren ~N/2 comparaciones.

Implementación en arreglo ordenado: Búsqueda binaria

- Se usan dos arreglos, uno para las llaves y otro para los valores.
- Operación `put` debe mover items del arreglo para insertar el item en la posición que le corresponde según el orden de las llaves.
- Operación `get` retorna el item del vector de valores de la posición en la que se encuentra el item.
- Ambas operaciones dependen de la operación `rank(key)`.

Operación rank(key)

- Retorna el número de elementos menores a la llave.
- Corresponde a la posición del la llave

```
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;
    }
```

Operaciones `get/put`

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

public void put(Key key, Value val)  {
    if (val == null) { delete(key); return; }
    int i = rank(key);
    // key is already in table
    if (i < N && keys[i].compareTo(key) == 0) {
        vals[i] = val;
        return;
    }
    // insert new key-value pair
    if (N == keys.length) resize(2*keys.length);
    for (int j = N; j > i; j--)  {
        keys[j] = keys[j-1];
        vals[j] = vals[j-1];
    }
    keys[i] = key;
    vals[i] = val;
    N++;
}
```

Código fuente

Otras operaciones

- Gracias a estar ordenado el arreglo de llaves, resulta fácil implementar los métodos de ST ordenadas: `max, min, ceil, floor, select, delete`.

Eficiencia de la búsqueda binaria

Proposición

- La búsqueda binaria requiere como máximo lg(N)+1 comparaciones.

Dem:

- Sea C(N) el número de comparaciones.
- Se parte de los casos C(0)=0 y C(1)=1.
- El número de comparaciones esta descrito por la recurrencia:

$$C(N) = C\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + 1$$

Eficiencia de otras operaciones

Operación	Orden de crecimiento
`put(), delete(), deleteMin()`	~N
`get(), contains(), rank()`	~log(N)
`size(), min(), max(), select()`	~1
`floor(), ceiling()`	~log(N)

En particular las operaciones `put`,`delete` pueden requerirse con mucha frecuencia, lo que hace muy ineficiente esta estructura para operar con volúmenes grandes de datos.