Segundo Parcial de Programación 2

5 de julio de 2019

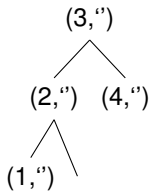
Pregunta sobre el Laboratorio

En la Tarea 4 se presentó la función `avl_min` que tiene como parámetro un entero no negativo. Describa la función: ¿qué representa el parámetro de entrada y qué devuelve la función? Dibuje el resultado cuando el parámetro de entrada es 3.

Solución:

El parámetro representa la altura de un `AVL`. La función devuelve un `AVL` de elementos de tipo `info_t` con la mínima cantidad de elementos que puede tener un `AVL` de la altura dada en el parámetro. Los datos numéricos de los elementos van desde 1 hasta la cantidad de nodos del árbol. El dato de texto de cada elemento es la cadena vacía. En ningún nodo puede ocurrir que el subárbol derecho tenga más nodos que el subárbol izquierdo.

Si la altura es 3 el árbol devuelto es el siguiente:



Problema 1 (19 puntos)

Considere el *TAD Cola de Prioridad* **acotada** de enteros, con operaciones *crear*, *esVacia*, *insertar*, *prioritario*, *eliminarPrioritario* y *estaLlena*. La prioridad de cada elemento está dada por su valor y el elemento prioritario es el entero de menor valor. Puede haber elementos (prioridades) repetidos.

(a) Escriba una matríz comparativa con los órdenes de tiempos de ejecución en el peor caso de las operaciones del *TAD Cola de Prioridad* **acotada** para implementaciones basadas en: `Lista ordenada enlazada`, `AVL` y `heap`. En las filas de la matriz debe tener los nombres de las tres implementaciones y en las columnas las operaciones del *TAD*.

Justifique brevemente el orden de las operaciones en el peor caso para *insertar* y *prioritario* para cada una de las tres implementaciones.

(b) Considere la siguiente implementación de `heap`, donde `max` es la cantidad máxima de elementos que puede almacenar, los elementos se almacenan en las posiciones $1..$`tope` y `tope` $\leq$ `max`:

```
struct rep_heap {
int max;
int tope;
int * elems ;
};

typedef struct rep_heap * heap ;
```

Implemente la función `filtradoAscendente` de `heap`:

```
//Filtra de forma ascendente el elemento en la posicion i del heap h
//Precondición: 1 <= i <= h->tope
void filtradoAscendente(int i, heap & h) ;
```

(c) Dada la siguiente representación de Cola Prioridad mediante `heap`, implemente la operación *insertar*:

```
typedef heap ColaPrioridad;

//Inserta el elemento p en la cola de prioridad cp
//Precondición: cp no debe estar llena
void insertar(int p, ColaPrioridad & cp) ;
```

Solución:

(a)

	crear	esVacia	insertar	prioritario	eliminarPrioritario	estaLlena
Lista ordenada	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$
AVL	$O(1)$	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$
heap	$O(1)$	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$

`Lista ordenada` Se implementa con un registro cabecera compuesto por un puntero al primer nodo y un entero con la cantidad de elementos que aún se pueden insertar. Este último campo permite resolver `estaLlena` en $O(1)$. El tiempo de `insertar` es $O(n)$ porque, en el peor caso, hay que recorrer toda la lista hasta encontrar el punto de inserción. El tiempo de `prioritario` es $O(1)$ porque es el primer elemento de la lista y se accede a él mediante el puntero en la cabecera.

`AVL` Se implementa con un registro cabecera compuesto por un puntero a la raíz, un entero con la cantidad de nodos que aún se pueden insertar y el valor del elemento prioritario (el mínimo). Es un árbol en el que todas las hojas están en un nivel $O(\log n)$ y la inserción siempre es un una hoja por lo que la insercion es $O(\log n)$. El prioritario se obtiene en $O(1)$ mediante el valor en la cabecera. Si no hay un puntero al mínimo el costo es $O(\log n)$ porque el mínimo siempre es una hoja.

`heap` Es un árbol completo por lo que su altura es $\lceil \log(n+1) \rceil$. El tiempo de `insertar` es $O(\log n)$ porque, en el peor caso, el nuevo elemento debe filtrarse desde el último nivel hasta la raíz. El tiempo de `prioritario` es $O(1)$ porque está en la raíz del árbol.

(b)
```
// Versión iterativa
// Precondición: 1 <= i <= h->tope
void filtradoAscendente(int i, heap &h) {
  int dato = h->elems[i];
  while ((i > 1) && (h->elems[i/2] > dato)) {
    h->elems[i] = h->elems[i/2];
    i /= 2;
  }
  h->elems[i] = dato;
}

// Versión recursiva
// Precondición: 1<=i <= h->tope
void filtradoAscendente ( int i, heap &h ) {
  if (i > 1 && h->elems [ i/2 ] > h->elems [ i ] ) {
    int swap = h->elems [ i ] ;
    h-> elems [ i ] = h->elems [ i/2 ];
    h-> elems [ i/2 ] = swap ;
    filtradoAscendente ( i/2, h );
  }
}
```

(c)
```
//Precondición: cp no debe estar llena
void insertar(int p, ColaPrioridad &cp) {
  cp->tope++;
  cp->elems[cp->tope] = p;
  filtradoAscendente(cp->tope, cp);
}
```

Problema 2 (14 puntos)

(a) Un árbol `AVL` es un árbol binario de búsqueda (ABB) que cumple una propiedad estructural característica. Defina esta propiedad.

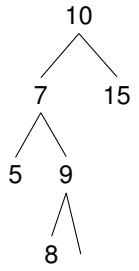
(b) Se pretende insertar en un ABB de enteros (inicialmente vacío) los elementos 10, 7, 15, 5, 9, 8, 20 (en ese orden). ¿Al insertar qué elemento se dejaría de cumplir la propiedad de estructura de los `AVL`? Dibuje el árbol luego de la inserción de dicho elemento, indicando qué nodo, o nodos, del árbol violan la propiedad. Justifique.

(c) Dado el árbol dibujado en la parte anterior: dibuje la secuencia de árboles resultantes de ejecutar la o las rotaciones **simples** necesarias para restablecer la propiedad estructural de `AVL`. Para cada rotación indique en qué nodo se aplica y si es una rotación `izquierda` o `derecha`.

Solución:

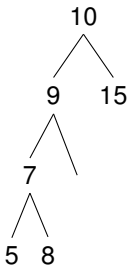
(a) La propiedad de estructural de los `AVL` es que en cada nodo el valor absoluto de la diferencia de las alturas de sus dos subarboles es 0 o 1.

(b) La propiedad estructural se deja de cumplir al insertar el valor 8. El árbol es:

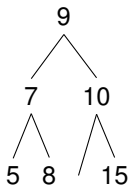


El único nodo en el que no se cumple la propiedad estructural es la raíz. Su subárbol izquierdo pasa a tener altura 3 mientras que su subárbol derecho mantiene altura 1.

(c)



Árbol desequilibrado después de rotación a la izquierda en 7.



Árbol equilibrado después de rotación a la derecha en 10.

Problema 3 (21 puntos)

(a) Especifique el TAD `Tabla(Mapping)` de *Strings* a enteros con las siguientes restricciones:

- Los enteros están en el rango $1..$`M`, siendo `M` un entero positivo que recibe la operación de creación como parámetro de entrada.
- No hay enteros repetidos.
- La única precondición de la operación de inserción es que el entero debe estar en el rango $1..$`M`. Si el *String* ya tiene un entero asociado o el entero ya corresponde a un *String* la operación no tiene efecto.

El `TAD` debe incluir las siguientes operaciones adicionales:

- Dado un entero en el rango $1..$`M` devuelve `true` si al entero le corresponde un *String* y `false` en caso contrario.
- Recibe un entero en el rango $1..$`M` y devuelve el *String* correspondiente si lo tiene; en caso contrario la operación no está definida.
- La operación *intercambio*, que recibe dos enteros en el rango $1..$`M`, que tienen asociados un *String* cada uno y hace el intercambio de las correspondencias. Por ejemplo, si los enteros `x` y `z` tienen asociados los *String* `sx` y `sz` en una `Tabla`, luego de un *intercambio* de `x` y `z` , a `x` le pasa a corresponder `sz` y a `z` le pasa a corresponder `sx`.

No considere como parte de la especificación del `TAD` una operación para liberar la memoria de la representación.

(b) Defina una representación para `Tabla` que permita que el orden del tiempo de ejecución de las operaciones estándar (excepto la de creación) sea $O(1)$ caso promedio y el de las operaciones adicionales sea $O(1)$ peor caso. Cada *String* se debe almacenar en un solo lugar de la memoria. Justifique por qué la representación elegida cumple con las restricciones de tiempo.

(c) Implemente solamente las operaciones de inserción e *intercambio* del `TAD`. Se asume disponible:

- El tipo *String*, con las operaciones típicas de asignación y comparación de los tipos elementales (=, ==, !=, <, >).
- La función `int str_a_num(String s, int N)` que dados `s` y `N` devuelve un entero en el rango $0..$`N-1`, y tal que dados dos strings diferentes `s` y `t` la probabilidad de `str_a_num(s, N)` $=$ `str _a _num(t, N)` es $1/N$. Esto quiere decir que `s` y `t` son igualmente probables.

Solución:

(a)

```
/* Devuelve una tabla sin asociaciones.
   M es un entero / 1..M es el rango de los enteros con los que se forman las ↩
       asociaciones. */
Tabla crear(int M);

/* Asocia 'clave' con 'valor' en 't'.
   Si 'clave' ya tiene asociado un entero o 'valor' ya corresponde a algún String la ↩
       operación no tiene efecto.
   Precondición: 'valor' pertenece al rango 1..M. */
void asociar(String clave, int valor, Tabla &t);

/* Elimina de 't' la asociación de 'clave'.
   Precondición: esClave(clave, t) */
void desasociar(String clave, Tabla &t);

/* Devuelve true si y solo si 'clave' tiene entero asociad en 't'. */
bool esClave(String clave, Tabla t);

/* Devuelve el valor entero que tiene asociado 'clave'.
   Precondición: esClave(clave, t) */
int valor(String clave, Tabla t);
```

```
/*********************************/
/**** Operaciones adicionales ****/
/*********************************/

/* Devuelve true si y solo si a 'valor' le corresponde una clave en 't'. */
bool esValor(int valor, Tabla t);

/* Devuelve la clave que le corresponde a 'valor'.
   Precondición: esValor(valor, t) */
String clave(int valor, Tabla t);

/* Intercambia las claves que le correspondes a los enteros x y z.
   Precondición: esValor(x, t) && esValor(z, t)  */
void intercambio(int x, int z, Tabla &t);
```

(b)
```
struct Par {
    int valor;
    String clave;
}

struct nodo {
    Par asoc;
    nodo * sig;
}

typedef nodo * Lista_Pares;

struct repTabla {
    int max_rango; // define el límite superior del rango de valores válidos.
    Lista_Pares * tabla_hash; // tabla indizada por h(clave).
    nodo ** valores; // tabla indizada por valor.
}

typedef repTabla * Tabla;
```

El tipo `Tabla` es un puntero a la estructura `repTabla`. La estructura `repTabla` consta de un entero max_rango que representa el límite superior del rango de enteros válidos $[1..max\_rango]$ a los que se les puede asignar una clave. Además, `repTabla` contiene una tabla de Hash abierta indizada por claves `tabla_hash` y un arreglo al que se accede por valor `valores`, ambos son arreglos de punteros a elementos de tipo `nodo *`.

`tabla_hash` tiene tamaño max_rango y la posición en el arreglo está dada por el entero que se obtiene al aplicar la función *str_to_num* al string `clave`. En cada posición hay una lista de asociaciones que permite resolver las posibles colisiones. La estructura `Lista_Pares`, permite armar una lista encadenada simple donde cada elemento contiene una asociación (clave de tipo string y valor de tipo entero).

El arreglo `valores` tiene tamaño M+1 y cada posición en el arreglo se corresponde con el valor del par asociado.

La Figura 1 muestra la estructura definida anteriormente.

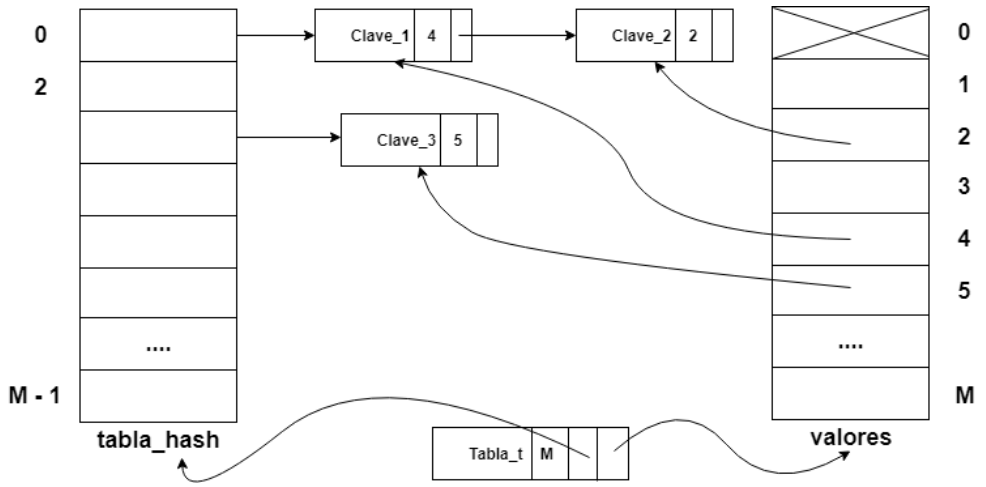


Figura 1: Multiestructura definida para la solución.

- La operación `crear` es $O(M)$ porque se deben inicializar los arreglos, asignando `null` en cada posición.
- La operación `asociar` es $O(1)$ en el caso promedio porque debe realizarse una búsqueda del String 'clave' en alguna de las listas que se encuentran en el arreglo `tabla_hash`. Se espera que, en promedio, dichas listas tengan largo 1.
- La operación `desasociar` es $O(1$ en el caso promedio por idénticos motivos que `asociar`.
- La operación esClave es $O(1)$ en el caso promedio por idénticos motivos que `asociar`.
- La operación `esValor` es $O(1)$ en el peor caso porque implica verificar que el contenido de una posición conocida del arreglo `valores` es distinto de `NULL`.
- La operación `valor` es $O(1)$ en el caso promedio por idénticos motivos que `asociar`.
- La operación `clave` es $O(1)$ en el peor caso porque implica obtener un dato que se encuentra en una posición conocida del arreglo `valores`.
- La operación `intercambio` es $O(1)$ en el peor caso porque implica el cambio de valores en posiciones conocidas del arreglo `valores`.

(c)

```
void intercambio(int x, int z, Tabla &t){
  t->arrValores[x]->pos = z;
  t->arrValores[z]->pos = x;
  nodo_pares *elem_intercambio = t->arrValores[x];
  t->arrValores[x] = t->arrValores[z];
  t->arrValores[z] = elem_intercambio;
}

bool esClave (String s, Tabla t) {
  bool clave_libre = true;
  lista_pares pares = t->tabla_hash [ str_to_num ( s, t->max_range ) ] ;
  while ( pares != NULL && clave_libre ) {
    clave_libre = pares->clave != s;
    pares = pares->sig;
  }
  return !clave_libre;
}

bool esValor (int valor, Tabla t) {
  return t->claves [ valor ] != NULL;
}

void asociar ( string s, int valor, Tabla &t) {
  if (!esValor(valor,t) &&  // valor no está asociado a ningún String
```

```
        !esClave(s,t)) {      // s no está asociado a ningún int
      Lista_Pares nuevo = new nodo;
      nuevo->asoc = {s,valor};
      int clave = str_to_num ( s, t->max_range );
      nuevo->sig = t->tabla_hash [ clave ];
      t-> tabla_hash [ clave ] = nuevo;
      t->valores [ valor ] = nuevo;
    }
  }
```