Guía de Ejercicios 7: Diseño de estructuras de datos

[Version: 4 de junio de 2024]

Objetivos:

- Introducir estructuras de datos avanzadas (pila, cola, heap, arbol binario, tabla de hash) que permiten resolver requisitos de complejidad temporal no triviales.

Ejercicio 1. Se quiere implementar un tipo de datos `Pila` utilizando `std::list` como estructura de representación.

```
template <typename T>
class Pila{
  public:
    Pila();
    // POST: La pila está vacía.
    bool es_vacia() const;
    const T& tope() const;
    // PRE: es_vacia() da falso.
    void apilar(T elem);
    // PRE: Verdadero
    // POST: Se agrega a la pila el valor elem, por lo tanto tope() es igual a elem.
    void desapilar();
    // PRE: es_vacia() da falso.
    // POST: elimina de la pila el elemento devuelto por tope()

  private:
    list<T> elementos;
};
```

(a) Escribir el invariante de representación en español.

(b) Implementar las operaciones e indicar el orden de complejidad de cada una.

Ejercicio 2. Se quiere implementar un tipo de datos `FilaDeSuper` utilizando `std::queue` como parte de su estructura de representación.

```
class FilaDeSuper{
  public:
    FilaDeSuper();
    // POST: La FilaDeSuper es vacía y sus cajas están libres.
    int cantidad_esperando() const;
    string siguiente() const;
    bool esta_ocupada(int caja) const;
    // PRE: caja es igual a 1 o 2.
    void llega_persona(string dni);
    // PRE: dni es un dni válido
```

```
    // POST: El dni se agrega al final de la fila
    void atender_siguiente(int caja);
    // PRE: caja es igual a 1 o 2, cantidad_esperando() no es 0 y esta_ocupada(caja) es false.
    // POST: Se quita de la fila a la siguiente() persona que estaba esperando.
    //       Y esta_ocupada(caja) se vuelve true.
    void liberar_caja(int caja);
    // PRE: caja es igual a 1 o 2, esta_ocupada(caja) es true.
    // POST: esta_ocupada(caja) se vuelve false.

  private:
    queue<string> fila_de_espera;
    bool caja_uno_ocupada;
    bool caja_dos_ocupada;
};
```

(a) Escribir el invariante de representación en español.

(b) Implementar las operaciones e indicar el orden de complejidad de cada una.

(c) Extender la implementación para aceptar mayor cantidad de cajas. Se indicará la cantidad de cajas en el constructor `FilaDeSuper(int cantidad_cajas)`. Puede modificar la estructura de representación.

(d) Extender la implementación agregando un método `int siguiente_caja_disponible() const` cuya precondición es que haya algúna caja disponible y debe devolver en $O(1)$ la caja disponible con número más bajo. Puede modificar la estructura de representación.

Ejercicio 3. Se quiere diseñar un tipo de datos `Diccionario` para asociar claves de tipo `T1` con valores de tipo `T2`. El `Diccionario` proveerá operaciones para definir el valor asociado a una clave, consultar si una clave está definida en el `Diccionario` y, en ese caso, obtener el valor asociado a una clave.

```
template <typename T1, typename T2>
// Asumir que el tipo T1 tiene operador de igualdad ==
class Diccionario{
  public:
    Diccionario();
    // POST: El diccionario está vacío.
    bool esta_definida(T1 clave) const;
    // PRE: Verdadero
    const T2 & obtener_valor(T1 clave) const;
    // PRE: esta_definida(clave)
    void definir(T1 clave, T2 valor);
    // PRE: Verdadero
    // POST: esta_definida(clave) es verdadero y obtener_valor(clave)
    //       es igual a valor. El resto del diccionario se mantiene igual.
    void borrar(T1 clave);
    // PRE: esta_definida(clave)
    // POST: esta_definida(clave) es falso. El resto del diccionario
    //       se mantiene igual.

  private:
    list< pair<T1,T2> > claves_valores;
};
```

Los observadores de `std::pair` son son `p.first` y `p.second`. Su documentación está en `https://en.cppreference.com/w/cpp/utility/pair`.

(a) Escribir el invariante de representación en español,

(b) Implementar las operaciones e indicar el orden de complejidad de cada una.

(c) Se quiere agregar una operación `void actualizar_ultimo(T2 valor)` que en $O(1)$ cambie el valor asociado a la última clave que se definió. Para eso se agregará la siguiente variable a la estructura de representación: `list< pair<T1,T2> >::iterator iterador_al_ultimo`. Implementar la nueva operación y modificar la implementación de otros métodos si es necesario.

Ejercicio 4. Sea el conjunto de elementos {1, 4, 5, 10, 16, 17, 21}.

(a) Dibujar distintos árboles binarios de búsqueda para el conjunto, con alturas 2, 3, 4, 5, y 6.

(b) ¿Alguno de los árboles está balanceado? Si la respuesta es no, dibujar uno que esté balanceado. (Un árbol binario está balanceado, según el criterio de AVL, si para todo nodo vale que la altura de su subarbol izquierdo y la altura de su subarbol derecho difieren en a lo sumo 1).

(c) Tomar el árbol balanceado y escribir el resultado de recorrerlo de forma inorder y preorder.

Ejercicio 5. Sea el conjunto de elementos {1, 4, 5, 10, 16, 17, 21}.

(a) Dibujar un árbol binario min-heap para el conjunto.

(b) Dibujar un árbol binario max-heap para el conjunto.

Ejercicio 6. Sea la siguiente definición de la estructura de datos arbol binario:

```
struct arbol_binario {
  int valor;
  arbol_binario* izq;
  arbol_binario* der;
};
```

(a) Escribir la función

`bool verificar_max_heap(const arbol_binario* raiz)`

que devuelve `true` si y solo si en todos los nodos del árbol se cumple la propiedad de MaxHeap. Un nodo cumple la propiedad de MaxHeap si el valor almacenado en ese nodo es mayor o igual a ambos de sus hijos directos, si los tuviera. Un arbol vacío (`nullptr`) cumple trivialmente la propiedad de MaxHeap.

Ejercicio 7. Se quiere diseñar un tipo de datos `DiccionarioDeInt` para asociar claves de tipo `int` con valores de tipo `T`. La interfaz es similar a `Diccionario` pero agregamos explícitamente el destructor `~DiccionarioDeInt()`. Se eligió como estructura de representación un Arbol Binario de Búsqueda.

```
template <typename T>
class DiccionarioDeInt{
public:
```

```
    DiccionarioDeInt();
    bool esta_definida(int clave) const;
    const T& obtener_valor(int clave) const;
    void definir(int clave, T valor);
    void borrar(int clave);
    ~DiccionarioDeInt();
private:
    struct nodo {
        nodo(int c, T v) : clave(c), valor(v), hijo_izq(nullptr), hijo_der(nullptr) {}
        int clave;
        T valor;
        nodo* hijo_izq;
        nodo* hijo_der;
    };
    nodo* raiz;
};
```

(a) Escribir el invariante de representación en español,

(b) Implementar las operaciones `esta_definida` y `definir` e indicar el orden de complejidad de cada una en peor caso.

(c) El mejor caso para la complejidad de los algoritmos es que el árbol esté perfectamente balanceado a la hora de llamar a una función. ¿Cuál sería la complejidad de las operaciones en ese caso?

Ejercicio 8. Se quiere extender el `DiccionarioDeInt` con una operacion `list<int> aplanar_ordenado() const` que devuelva una lista de enteros con todas las claves del diccionario en orden ascendente. La operación debe recorrer el árbol de forma `inorder`.

(a) Implementar `aplanar_ordenado` sin hacer recursión, usando una variable auxiliar de tipo `std::stack`.

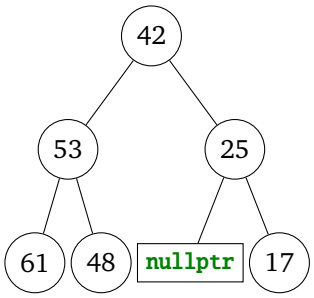
(b) Implementar `aplanar_ordenado` con un algoritmo recursivo siguiendo la estrategia Divide & Conquer.

Ejercicio 9. Sea la siguiente definición de la estructura de datos arbol binario:

```
1 struct arbol_binario {
2   int valor;
3   arbol_binario* izq;
4   arbol_binario* der;
5   int cantidad_total_hijos;
6 };
```

Asuma que este árbol binario cumple el invariante de Árbol Binario de Búsqueda **reverso** (es decir, ordenado de mayor a menor) y, además, para todo nodo `n` del arbol, `n.cantidad_total_hijos` indica la cantidad total de nodos presentes en ambos subárboles `n.izq` y `n.der`.

Por ejemplo, en el siguiente Árbol Binario de Búsqueda **reverso**, la cantidad total de hijos del nodo (42) es 5, la del nodo (25) es 1, la del nodo (53) es 2, la del nodo (61) es 0.



(a) Implementar la operación

`arbol_binario* insertar(arbol_binario* raiz, int elem)`

que realiza una inserción ordenada respetando el invariante requerido por la estructura. El algoritmo debe tener complejidad en peor caso $O(h)$ donde h es la altura del árbol.

(b) Implementar la operación

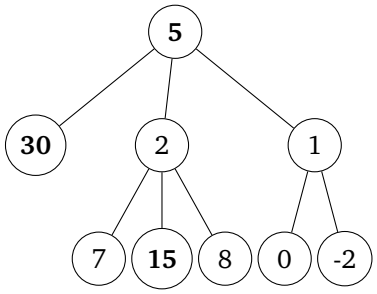
`int contar_menores_a(const arbol_binario* raiz, int valor)`

que devuelve la cantidad de elementos del árbol menores a `valor`. Asumiendo que el árbol binario está balanceado, el algoritmo debe tener complejidad $O(\log\ n)$ en peor caso, donde n es la cantidad total de elementos.

Por ejemplo, en el árbol anterior, la cantidad de elementos menores a 52 es 4.

Ejercicio 10. Sea la siguiente definición de la estructura de datos arbol ternario:

```
1  struct arbol_ternario {
2      int valor;
3      arbol_ternario* hijo_izquierdo;
4      arbol_ternario* hijo_medio;
5      arbol_ternario* hijo_derecho;
6  };
```



(a) Implementar la función `void multiplicar_por(arbol_ternario* raiz, int k)`, que modifica el valor de cada nodo en el árbol multiplicándolo por k.

Ejercicio 11. Asuma que cuenta con un tipo de datos `BolsaMinHeap` que permite almacenar enteros y provee la siguiente interfaz:

```
class BolsaMinHeap{
public:
    BolsaMinHeap();

    bool es_vacio() const;
    // Indica si no hay más elementos.
    void encolar(int c);
    // Almacena el elemento c.
    int desencolar_minimo();
    // Quita el elemento más chico y lo devuelve.
private:
    /* ... */
};
```

`BolsaMinHeap` está implementado sobre un arbol binario que mantiene el invariante de min-heap. Por lo tanto, las operaciones `encolar` y `desencolar_minimo` tienen orden de complejidad $O(\log n)$ y la operación `es_vacio` tiene orden de complejidad $O(1)$, donde n es la cantidad total de elementos almacenados.

(a) Implementar un algoritmo para ordenar un `vector<int>` en orden $O(n \log n)$ haciendo uso del tipo de datos `BolsaMinHeap`. (No es necesario implementar el tipo `BolsaMinHeap`)

(b) La biblioteca estándar de C++ provee la clase `std::priority_queue<int>` que cuya estructura de representación es un max-heap. Provee las siguientes operaciones:

- `void push(int valor)` inserta un nuevo valor en $O(\log n)$,
- `int top()` devuelve el elemento máximo en $O(1)$,
- `void pop()` quita el elemento máximo en $O(\log n)$,

Utilizar `std::priority_queue<int>` para ordenar un `vector<int>` en orden $O(n \log n)$ de menor a mayor.

Ejercicio 12. Se quiere diseñar un tipo de datos `ConjuntoDeStr` para almacenar elementos de tipo `string`. Se eligió como estructura de representación una Tabla de Hash.

```
class ConjuntoDeStr{
public:
    ConjuntoDeStr();
    // POST: El conjunto está vacío.
    bool esta(string elem) const;
    // PRE: Verdadero
    int cardinal() const;
    // PRE: Verdadero
    void agregar(string elem);
    // PRE: Verdadero
    // POST: esta(elem) es verdadero y el resto del conjunto se mantiene igual.
    void borrar(string elem);
    // PRE: esta(elem) es verdadero
    // POST: esta(elem) es falso y el resto del conjunto se mantiene igual.
private:
    int funcion_hash(string clave) const {
        return clave.length() % 100;
    }
    vector<list<string>> tabla;
    int cardinal;
};
```

La tabla contará con 100 posiciones. `funcion_hash` implementa una función de hash muy simple (y no muy efectiva) usando el método de la división. Se decidió resolver colisiones de la función de hash usando encadenamiento, por eso cada lugar de la tabla almacena una lista de elementos.

(a) Escribir el invariante de representación en español,

(b) Implementar las operaciones `agregar` y `esta` e indicar el orden de complejidad de cada una en peor caso.

(c) Si no hubiese colisiones en la función de hash, ¿cuál sería la complejidad temporal de la operación `esta(...)`?

Ejercicio 13. Se quiere diseñar un tipo de datos `DiccionarioDeStr` para asociar claves de tipo `string` con valores de tipo `T`. La interfaz es similar a `DiccionarioDeInt`. La estructura de representación sigue la misma lógica que `ConjuntoDeStr` pero ahora se almacenan `claves` y `valores` en un `std::pair`.

```
template <typename T>
class DiccionarioDeStr{
public:
    DiccionarioDeStr();
    bool esta_definida(string clave) const;
    int cantidad() const;
    const T& obtener_valor(string clave) const;
    void definir(string clave, T valor);
    void borrar(string clave);
private:
    int funcion_hash(string clave) const {
        return clave.length() % 100;
    }
    vector<list<pair<string, T>>> tabla;
    int cantidad;
};
```

(a) Escribir el invariante de representación en español,

(b) Implementar la operacion `definir` y `obtener_valor`.
Tip: la implementación debería ser similar a las operaciones `agregar` y `esta` del punto anterior.

(c) ¿Cambia el análisis de complejidad con respecto a `ConjuntoDeStr`?

Ejercicio 14. Se quiere implementar un sistema para registrar los alumnos que se presentan a una evaluación, junto con las notas que obtuvieron.

```
class Evaluaciones{
  public:
    Evaluacion(set<int> inscriptos);

    const set<int> & inscriptos() const;

    // PRE: el alumno está inscripto en el sistema, y nota está entre 0 y 100.
    void calificar(int alumno, int nota);

    // PRE: el alumno está inscripto en el sistema
    bool se_presentó(int alumno) const;

    // PRE: el alumno está inscripto y se presentó al examen.
    bool aprobó(int alumno) const;

    // PRE: el alumno está inscripto y se presentó al examen.
    int nota(int alumno) const;

  private:
    set<int> _inscriptos;
    set<int> _presentados;
    unordered_map<int, int> _nota_por_alumno;
```

```
23      vector<int> _aprobados;
24  };
```

Como estructura de representación se decidió almacenar todos los alumnos inscriptos en `_inscriptos`, aquello que se presentaron y fueron calificados en `_presentados`, la nota con la que cada uno fue calificado en `_nota_por_alumno` y aquellos que aprobaron (obtuvieron al menos 60) en `_aprobados`.

Se tienen las siguientes restricciones de complejidad temporal:

- `inscriptos()` - $O(1)$ en peor caso,
- `se_presentó(int alumno)` - $O(log\ n)$ en peor caso,
- `aprobó(int alumno)` - $O(log\ n)$ en peor caso,

donde n es la cantidad total de alumnos inscriptos.

(a) Explicar, para cada operación con restricción de complejidad, de qué manera se podría utilizar la estructura de representación dada para garantizar la complejidad pedida. No puede agregar más campos a la estructura.

(b) Elegir un invariante de representación adecuado para la estructura dada que permita resolver el problema planteado en las complejidades requeridas. Escribirlo en castellano.

(c) Dar un ejemplo de valores para las variables de la estructura interna que cumpla el invariante de representación elegido y otro ejemplo de valores que no lo cumpla.

(d) Dar la implementación de los métodos `calificar(int alumno, int nota)` y `aprobó(int alumno)` respetando el invariante propuesto.