UNIVERSIDAD DE GRANADA

/ UGR / decsai

Departamento de Ciencias de la Computación e Inteligencia Artificial

Estructuras de datos grupo B

Ejercicios sobre contenedores set, map, list, vector

Esta entrega corresponde a la relación de ejercicios sobre la varios contenedores de la STL.

- Debajo de cada ejercicio puedes encontrar la función que debes implementar. Además, debe haber una función main con las distintas pruebas que has realizado para verificar el funcionamiento de la función.
- las cabeceras de las funciones deben ser exactamente tal y como aparecen.
- Si un ejercicio no funciona correctamente o se tiene claro que no resuelve todos los casos posibles, se debe explicar adecuadamente.
- La solución debe ser la más eficiente posible. incluida la constante, tanto en tiempo como en espacio.
- Los ficheros no deben dar errores de compilación.

Cualquier entrega que no cumpla con estas normas no será tenida en cuenta.

Ejercicios:

1. Un método de encriptación basado en un código consiste en transformar cada letra del alfabeto por otra. Implementa una función encripta a la que se le pasa una cadena de caracteres y un map <char, char > con el código y que devuelva la cadena encriptada.

   string encripta(string cad, const map<char, char> &codigo)

2. Implementa una función desencripta que haga el proceso inverso al del ejercicio anterior.

   string desencripta(string cad, const map<char, char> &codigo)

3. Construye una función a la que se le pase un texto (que contiene signos de puntuación, etc) y devuelva un map que contenga las palabras que aparecen y el número de veces que aparece cada una.

   map<string,int> contar(string texto)

4. Construye una función que dado el map del ejercicio anterior devuelva un vector que nos permita consultar las palabras que aparecen un número determinado de veces.

   vector<string> veces(const map<string,int> &palabras, int num)

5. Construye una función a la que se le pasen los parámetros con los que se ha ejecutado un programa y determine qué opciones se han seleccionado y, en su caso, el valor que se le ha asignado. Por ejemplo:

   tar -z -x -f fichero debería indicar que se han activado z,x y también f que tiene un parámetro fichero.

   map<string, string> params(string cad)

6. Implementa una función

void elimina(list<T> &l, const vector<T> &v)

que, dada una lista l de tipo T y un vector v de tipo T, elimine cualquier elemento de v de todas las posiciones en las que aparezca en l. Por ejemplo: si v = (1,6) y l = (2, 1, 1, 1, 1, 5, 3, 6) el resultado debería ser (2, 5, 3)

7. Implementa una función void elimina_duplicados(list<T> &l) que elimine los elementos duplicados (sin ordenar previamente la lista). Deben aparecer en el mismo orden en que estaban inicialmente en l. Y debe tener una eficiencia O(n x log n)
8. Implementa una función que dada una lista l devuelva una lista que tenga los elementos de l pero en orden inverso.

   list<T> inversa(const list<T> &l)
9. Resuelve el problema anterior pero sobre la lista pasada por referencia y sin usar ningún contenedor adicional.

   void inversa(list<T> &l)
10. Implementa una función list<T> mezclar(const list<T> &l1, const list<T> &l2) que dadas dos listas ordenadas l1 y l2 devuelva una lista ordenada mezclando las dos listas.
11. Implementa una función a la que se le pase una lista de enteros y un entero x de manera que cada vez que aparezca en una posición ese valor, se inserte x-1 en la posición siguiente.

    void inserta_despues(list<int> &l, int x)
12. Implementa una función

    typename list<T>::const_iterator contenida(const list<T> &l1, const list<T> &l2)

    a la que se le pasen dos listas y compruebe si la lista l1 está contenida en l2 (si los elementos aparecen en la otra y en el mismo orden). Devuelve la posición de l2 a partir de la cual se encuentra l1 o end() en el caso de no estar contenida.
13. Dadas dos listas: l, que contiene n elementos y otra li que contiene una serie de posiciones de la lista anterior (valores de la clase iterador). Construye una función a la que se le pasen esas dos listas y devuelva otra que contenga los elementos de l indicados por las posiciones de la lista li.

    list<T> lista_posiciones(const list<T> &l, const list< typename list<T>::iterator> &li)
14. Un vector disperso es un vector en el que la inmensa mayoría de sus elementos son nulos. Para representar ese tipo de vectores se suele utilizar como representación una lista:

    ```
    template <typename T>
    class vdisperso {
    public:
      vdisperso();
      vdisperso(const vector<T> &v);
      void asignar_coeficiente(int i, const T &x);
      T operator[] (int i) const;
      vector<T> convertir() const;
      int size() const;
    ```

```
  int size_not_nulls() const;  // número de coeficientes no nulos
  ...
private:
  list< pair<int, T> > coefs;
  int n; // número de elementos del vdisperso (no el de coefs)
}
```

De esa forma se ahorra una gran cantidad de espacio. Implementa los métodos indicados en la parte pública.

15. Una variante del vector disperso es aquella en la que se puede definir cual es el valor nulo. Implementa una variante de la clase vdisperso de forma que se pueda definir cual es el valor nulo en los constructores:

    vdisperso(const T &valor_nulo);
    vdisperso(const vector<T> &v, const T &valor_nulo);

    Implementa también la función

    void cambiar_nulo(const T &valor_nulo)

16. Una forma eficiente de guardar secuencias de valores iguales consiste en guardar cada uno de los valores seguido del número de veces que aparece en la secuencia. Por ejemplo,

    <1,1,1,2,2,2,2,2,2,7,7,7,7,7,12,1,1,5,5>

    se representaría como:

    < (1,3), (2,6), (7,5), (12,1), (1,2), (5,2) >

    Implementa las funciones:

    - list<pair<T, int> > comprimir(const list<T> &l)
    - list<T> descomprimir(const list<pair<T, int> > &lc)

    que permitan convertir entre ambas representaciones.

17. Implementa la clase vector_dinamico<T>, usando como representación el tipo list. ¿Qué orden de eficiencia tiene cada operación?

```
class vector_dinamico {
  public:
    iterator insert(iterator it, const T &x);
    iterator erase(iterator it);
    iterator erase(int i);
    void push_back(const T &x);
    T& operator[](int i);
    int size() const;
    iterator begin();
    iterator end();
    class iterator {
```

…

}

}

la clase iterator debe tener las operaciones *, ++, ==, !=

18. Implementa la función intercalar(vector<T> &v, vector<pair<int, T> > pos) que inserta el segundo elemento de cada par de pos en v en los lugares indicados por el primer elemento de cada par de pos. Por ejemplo, v = (1,3,7,3), pos = ((1,5), (0, 7), (4, 12)) daría como resultado v =(7, 1, 5, 3, 12, 7, 3). Las posiciones deben ser >=0 y <=v.size()
19. Define una clase contenedor **pilotos** que permita guardar los datos de los pilotos de F1. Usa un struct **piloto** con los datos: nombre, apellidos, escuderia (nombre de la escudería), puntos. Además debe permitir que se pueda buscar a un piloto por sus apellidos (suponiendo que no hay repetidos). ¿Qué podríamos añadir a esa clase para que se pueda buscar también por cualquiera de los restantes campos?.

    Implementa las funciones:

    - piloto buscar_apellidos(string apellidos) const;// Si no hay nadie devolvería un “piloto vacío”
    - piloto buscar_nombre(string nombre) const; // Si no hay nadie devolvería un “piloto vacío”
    - list<piloto> buscar_escuderia(string ecuderia) const ; // Si no hay nadie devolvería una lista vacía
    - piloto posicion(int num) const; // Si no hay nadie en esa posición devolvería un “piloto vacío”
    - void insertar_piloto(const piloto &p);
    - void cambiar_puntuacion(string apellidos, string nombre, int puntos);
    - Define un iterador de la clase que itere por los pilotos de menor a mayor posición.
20. Escribe una función

    vector<pair<pair<float, float>, int> > contar(const vector< pair<float, float> > &v)

    que cuente el número de veces que aparecen una serie de coordenadas (como un par de float). En el vector los datos deben estar ordenados por la primera componente y en caso de ser iguales por la segunda. La eficiencia debe ser O(n log n)
21. En un servicio de urgencias de un hospital quieren tener la posibilidad de poder gestionar el orden en que los pacientes se irán atendiendo. Para ello, de cada paciente se guarda un struct **paciente** con 3 strings para nombre, apellidos, dni y un entero para la gravedad. Al mismo tiempo se quiere poder acceder por dni. Construye una clase **urgencias** adecuada e implementa las operaciones de inserción y borrado.

    class urgencias {

    urgencias();

    insertar_paciente(const paciente &p);

    // Devuelve la persona más grave (mayor valor de gravedad) y la saca del contenedor

    paciente mas_grave();

```
  paciente buscar_paciente(string nombre, string apellidos) const;
  int size() const;
  iterator begin() const;
  iterator end() const;
  class iterator; // que permita iterar por los pacientes por orden de apellidos
  // con las operaciones ,++(int), ==, !=
}
```