Estructuras de Datos y Algoritmos

Doble Grado, Ingeniería Informática, de Computadores y del Software

Examen final, 9 de Septiembre de 2016.

1. **(2,5 puntos)** Se tiene almacenado en un vector de enteros el recorrido en preorden de un árbol completo de $n \geq 1$ niveles. Diseñar una función booleana recursiva que determine si el árbol que corresponde a dicho recorrido es un árbol binario de búsqueda.

   Nota: en el recorrido en preorden de un árbol binario completo de n niveles, si la raiz está en la posición r, el recorrido del subárbol izquierdo abarca $2^{n-1} - 1$ elementos a partir de la posición $r + 1$ incluida, y el recorrido del subárbol derecho otros $2^{n-1} - 1$ elementos a partir de la posición $r + 2^{n-1}$ incluida.

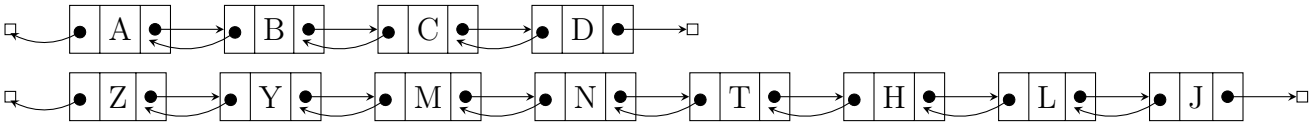
2. **(2,5 puntos)** Un comerciante del desierto dispone de un camello para transportar sus mercancías. El camello tiene dos alforjas cuya capacidad en peso, p_1 y p_2, son conocidas. El comerciante quiere cargar el camello de tal forma que se maximice el valor de los objetos transportados, teniendo en cuenta que los objetos no se pueden fraccionar. Sean v_i y w_i (ambos positivos) el valor y el peso del objeto i respectivamente ($0 \leq i < n$). Para poder viajar el camello debe ir equilibrado, por lo que se exigirá que la diferencia de pesos entre las alforjas no supere el 5% del peso total. Implementar un algoritmo que le indique al comerciante cómo cargar su camello.

3. **(2 puntos)** Sea la clase `ListaEnlazadaDoble` que implementa listas doblemente enlazadas con punteros al primer y último nodo, no circulares y sin nodo cabecera:

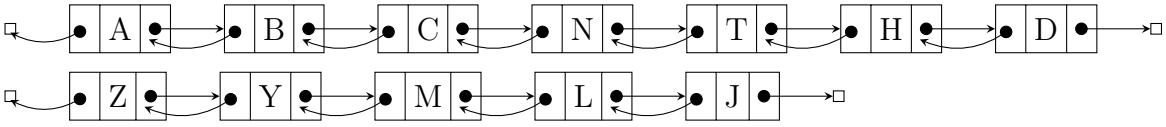
```
template <class T>
class ListaEnlazadaDoble{
private:
    class Nodo {
    public:
        Nodo() : _ant(NULL), _sig(NULL) {}
        Nodo(const T &elem) :  _elem(elem), _sig(NULL), _ant(NULL) {}
        Nodo(Nodo* ant, const T &elem, Nodo *sig) :  _elem(elem),
            _sig(sig), _ant(ant) {}
        T _elem;
        Nodo *_sig;    Nodo *_ant;
    };
   Nodo *_prim;   Nodo *_ult;
   ...
}
```

   Se pide implementar el método `splice`, que recibe como parámetro una lista `l` y tres valores de tipo `T, v1, v2 y v3`. El método debe modificar la lista original añadiendo después del valor `v1` los nodos de la lista `l` que se encuentran entre los valores `v2` y `v3` (ambos inclusive), y debe eliminarlos de la lista `l`. Si el valor `v1` no existe en la lista original, o si alguno de los valores `v2, v3` no existe en la lista `l`, o si `v3` no se encuentra en una posición posterior a `v2`, las listas no se modifican. Se supone que todos los valores de las listas son diferentes.
   La implementación debe ser lo mas eficiente posible. Para ello, se debe: evitar libera/reservar memoria y hacer copias de valores cuando no sea necesario, y recorrer las listas el menor número posible de veces. No se puede utilizar ninguna operación de la clase `ListaEnlazadaDoble` en la implementación.

Por ejemplo, dadas las siguientes listas de caracteres `l1` y `l2`:



el resultado de la operación `l1.splice(l2,'C','N','H')` será:



4. **(2 puntos)** A nuestro comerciante del ejercicio 2. le van tan bien los negocios desde que optimiza la carga de su camello, que ha decidido instalar en su almacén un sistema de gestión de los pedidos por internet. El sistema cuenta con una tabla en la que se almacenan todos los objetos junto con información sobre el precio, peso y número de unidades disponibles de cada uno de ellos. Ya se tienen implementadas operaciones para dar de alta nuevos objetos, y para modificar las características de un objeto existente.

```
typedef string Objeto;
typedef struct {
    float precio;
    float peso;
    int numUnidades;
} InfoObjeto;
class Pedidos {
    public:   ... operaciones para dar de alta objetos y modificarlos ...
    private:
      HashMap<Objeto,InfoObjeto> objetos;
}
```

El sistema debe almacenar información sobre el pedido (lista de objetos) que debe entregarse cada día y la dirección de envío. Solo da tiempo a entregar un pedido al día. La clase `Pedidos` debe contar con las siguientes operaciones:

- `nuevoPedido`: Recibe una lista con los objetos que un cliente solicita, la fecha en la que desea recibir la mercancía y la dirección de envío. La operación debe construir una lista con los objetos existentes en el almacén que maximicen las ganancias del comerciante teniendo en cuenta que por ahora no cuenta nada mas que con su camello (función del ejercicio 2.), y encontrar la fecha en la que serán servidos. Ésta debe ser la fecha más próxima posterior a la solicitada que no tenga todavía asignada ningún pedido.
- `preparaPedidos`: Dadas dos fechas, devuelve una lista, ordenada por fecha, con información sobre los envíos que se realizarán entre las dos fechas dadas, ambas incluidas. Cada elemento de la lista contendrá la fecha de entrega, la lista de objetos a enviar y la dirección de envío.

Se puede hacer uso de un tipo `Fecha` que dispone de una operación para obtener la siguiente fecha a una dada. Se pide:

- **(0.5 puntos)** Diseñar una representación que permita implementar las operaciones pedidas de forma que el acceso a la información almacenada sea eficiente. Indicar qué operaciones adicionales ha de tener disponibles el tipo `Fecha` para poder usarlo en la representación propuesta.
- **(1.5 puntos)** Implementar la función `nuevoPedido` e indicar su coste.

5. **(1 punto)** Responde a las siguientes cuestiones (0.2 puntos cada una):

5.1) Dadas dos listas `List<int> l,t;` se quieren copiar los k primeros elementos de la lista `l` insertándolos en la lista `t`, de manera que queden en el mismo orden en que se encuentran en la lista `l`. Dado un iterador `ConstIterator it = l.cbegin()`, la operación correcta es:

a) `for (;it != l.cend(); ++it) t.push_back(*it);`

b) `for (;it != l.cend(); ++it) t.push_front(*it);`

c) No se puede realizar utilizando este iterador

d) Ninguna de las anteriores

5.2) Dada una operación de concatenación de listas con coste lineal en el número de nodos de la lista argumento, la siguiente implementación del recorrido en preorden de un árbol

```
List<T> Arbin<T>::preordenAux(Nodo *p) {
  if (p == nullptr) return List<T>(); // Lista vacia
  List<T> ret;  ret.push_front(p->_elem);
  ret.concat(preordenAux(p->_iz));
  ret.concat(preordenAux(p->_dr));
  return ret;
}
```

tiene un coste en el caso peor (en función del número n de nodos del árbol) que está en:

a) $O(1)$

b) $O(\log n)$

c) $O(n)$

d) $O(n\ \log n)$

e) $O(n^2)$

5.3) En la pregunta anterior, si el árbol está balanceado, el coste del recorrido en preorden con la implementación dada será

a) $O(1)$

b) $O(\log n)$

c) $O(n)$

d) $O(n\ \log n)$

e) $O(n^2)$

5.4) La operación de añadir un nodo en un árbol de búsqueda tiene coste en el caso peor

1. logarítmico en el número de nodos del árbol
2. lineal en el número de nodos del árbol
3. logarítmico en la altura del árbol
4. lineal en la altura del árbol

Son correctas

a) 1 y 3

b) 2 y 3

c) 1 y 4

d) 2 y 4

5.5) Dada la secuencia de números $\mathbf{6}, \mathbf{13}, \mathbf{5}, \mathbf{7}, \mathbf{1}, \mathbf{20}, \mathbf{15}$. Dibujar el árbol binario de búsqueda que se obtiene al insertar los elementos de la secuencia en ese orden. Partiendo de ese mismo árbol, dibujar también el árbol resultante tras borrar los elementos 6 y 7 en ese orden.