Estructuras de Datos

Relación de ejercicios 1: implementación de estructuras de datos

1. Vectores, listas, pilas, colas

1. Implementar una función recursiva que devuelva el máximo de un array de enteros.
2. Construir una clase VAjustado que herede de la clase Vdinamico estudiada en la teoría e incluya una operación ajustar() que sustituya el array utilizado internamente como soporte por un array con el tamaño justo para contener los elementos almacenados en el vector.
3. Implementar una función que devuelva los 10 valores mayores de un array y calcular su tiempo de ejecución.
4. Implementar una clase TresEnRaya que utilice un vector bidimensional de 10x10 para implementar el juego tres en raya. Las operaciones a incluir serán:

   - *void fichaEnColumna(int columna, char jugador)*: deja caer la ficha del jugador en la columna indicada. La ficha caerá hasta la primera posición vacía de la columna.
   - *bool ganaJugador(int jugador)*: Indica si el jugador ha ganado.

   Donde el jugador viene identificado por los caracteres 'X' u 'O'
5. Implementar la clase Bicola<T> que permite realizar inserciones y borrados tanto por el comienzo como por el final de una lista en tiempo constante. Utilizad para ello la representación interna de las listas doblemente enlazadas.
6. Una expresión matemática se puede expresar mediante un árbol binario, pero también como una secuencia de operandos y operadores resultado de recorrer dicho árbol en postorden de modo que la secuencia quedara: *[operando1][operando2][operador]*. Por ejemplo: la expresión "(a+b)*(c-d)" quedaría como "ab+cd-*".
   Implementad una función evalua(const string&) que resuelva dicha expresión matemática utilizando una pila.
   Para simplificar el ejercicio se asume que los operandos sólo poseen una cifra.
7. Implementar la clase Cola<T> heredando de la clase Pila<T> implementada mediante listas enlazadas utilizando convenientemente las reglas de privacidad.
8. Implementar la función ListaEnlazada<T>::elimina(T &ele); que elimine de la lista todas las instancias del dato igual a *ele*.
9. Implementar la función ListaEnlazada<T>::operator+(ListaEnlazada<T> &l); que concatene los elementos de la lista con los de la lista dada por parámetro.
10. Implementar una operación void invertir() en la clase ListaEnlazada<T> (lista simplemente enlazada) para invertir el orden de los nodos de la misma.
11. Implementar una operación que calcule el número de nodos de una lista circular simplemente enlazada como la estudiada en prácticas.
12. Implementar una operación *vector<int> primosMenores(int n)* que devuelva la lista de los números primos menores que n usando la criba de Euler. Su funcionamiento es el siguiente:

    a) Generar una lista con los números desde 2 a n
    b) El primer número de la lista es primo (añadir al resultado)
    c) Multiplicar el primer número por todos los elementos de la lista y marcar los elementos resultantes
    d) Eliminar el primer elemento y todos los elementos marcados

e) Volver al paso (b) si la lista no está vacía

13. Describir como implementar una pila utilizando únicamente dos colas. Indicar el tiempo de ejecución de las operaciones push y pop.

2. Árboles y heaps

1. Como vimos en la lección 10 (transparencia 7). Un árbol puede representar una expresión matemática. Implementar la operación *evalua()* de la clase *ArbolExpresion* con la siguiente definición:

```
class NodoExpresion {
  char operador; // Valores *, -, +, / o " "
  float operando; // Sólo válido cuando operador = " "
  NodoExpresion *op1, *op2;
};

class ArbolExpresion {
  NodoExpresion *raiz;
public:
  ...
  float evalua();
  ...
};
```

2. Implementar una operación *int factorEquilibrio(Nodo *nodo)* que devuelva la diferencia de altura entre el subárbol izquierdo y derecho del nodo en un árbol binario.
3. Implementar una operación Nodo *minimoComunAncestro(Nodo *a, Nodo *b) que devuelva el nodo más abajo en el árbol que tenga a los nodos a y b como descendientes.
4. Implementar el constructor de un árbol ABB a partir de un vector ordenado
5. Implementar una función que recorra un árbol (avl o abb) por niveles
6. Implementar la versión no recursiva del recorrido en Preorden un árbol (avl o abb)
7. Implementar la función k-prioritario en la clase Heap<T> para que devuelva (pero no elimine) el elemento k prioritario. Si k=1 será el más prioritario, si k=2 el segundo y así sucesivamente.
8. Implementar una función privada: list<Nodo<T> > ABB<T>::listarHojas() que devuelva un listado ordenado de los nodos hojas de izquierda a derecha.
9. Implementar una función: bool ABB<T>::completo() de devuelva true si el arbol es completo. Para ello se recorren todas las hojas de izquierda a derecha, se determina la altura de todos los nodos hoja. Un árbol es completo si todas las hojas tienen la misma altura h o altura h-1; en el caso de existir nodos hoja con altura h-1 deben estar agrupados al final de dicha lista.

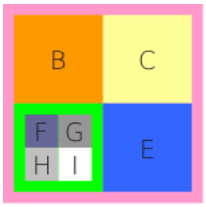
3. Tablas hash

1. Implementad la función DispCerrada::redispersar() que redisperse los datos de una tabla hash cerrada.
2. Mostrar el estado de una tabla de 11 entradas que usa la función hash h(k)=(3k + 5) % 11 tras insertar las claves 12, 44, 13, 88, 23, 94, 11, 39, 20, 16 y 5 asumiendo que las colisiones son resueltas mediante exploración lineal.
3. Mostrar el resultado de la tabl anterior usando dispersión doble y función hash secundaria h'(k)= 7 - (k % 7)

4. Estructuras de datos espaciales

1. Implementar la función de la clase Nodo de un Quadtree que devuelva el nivel donde se encuentra dicho nodo.
2. Implementar las siguientes funciones de la clase Nodo asociada a un Quadtree, de modo que devuelvan los nodos situados a la izquierda, derecha, arriba o abajo del nodo dado, de acuerdo a la siguiente figura:

- Nodo* Nodo::izquierda();
- Nodo* Nodo::derecha();
- Nodo* Nodo::arriba();
- Nodo* Nodo::abajo();



De modo que:

- B es el nodo que está arriba de G
- E es el nodo que está abajo de C
- Los cuatro nodos F,G,H,I son los nodos que están a la izquierda de E (se devuelve el nodo padre)
- Se devuelve 0 como el nodo que está abajo de E

5. Grafos

1. Dado un grafo G con 8 vértices y los siguientes vértices adyacentes: 1->(2, 3, 4), 2->(1, 3, 4), 3->(1, 2, 4), 4->(1, 2, 3, 6), 5->(6, 7, 8), 6->(4, 5, 7), 7->(5, 6, 8), 8->(5, 7). Mostrar:
   a) El grafo gráficamente
   b) Su matriz de adyacenciaç
   c) La secuencia de vértices visitados en un recorrido primero en anchura comenzando en el vértice 1
   d) La secuencia de vértices visitados en un recorrido primero en profundidad comenzando en el vértice 1.
2. Implementar un constructor de un grafo implementado mediante una matriz de adyacencia a partir de un grafo implementado con lista de adyacencia
3. Implementar un constructor de un grafo implementado mediante una lista de adyacencia a partir de un grafo implementado con matriz de adyacencia
4. Dado un grafo ponderado implementado mediante lista de adyacencia, implementar dos funciones, una para encontrar la arista con menor longitud, y otra para encontrar la de mayor longitud.
5. Implementar una función que determine si un grafo es o no dirigido, tanto para una implementación con matrices de adyacencia y con listas de adyacencia

6. Ficheros

1. Implementar la función FicheroIndexadoLibros::eliminar(string &isbn) que elimina el libro con el isbn dado en un fichero indexado que maneja un índice primario por isbn y un índice secundario por autor.
2. Modificar las funciones FicheroLibro::insertar() y FicheroLibro::borrar() para que permitan aprovechar las posiciones de registros borrados mediante una pila de borrados (lección 19).

3. Implementar la función FicheroIndexadoLibros::ListadoOrdenado(char *fichListado) para que liste en un fichero de texto cuyo nombre se da como parámetro, todos los libros ordenados por autor considerando que dicha la clase FicheroIndexadoLibros mantiene un índice primario por ISBN y un índice secundario por autores. Cada libro debe aparecer en una línea distinta del fichero de texto.