***Apellidos, Nombre:*** ***DNI:***

# ***Examen PED julio 2021***
# ***Modalidad 0***

**Normas:**

- Tiempo para efectuar el test: **25 minutos**.
- Una pregunta mal contestada elimina una correcta.
- Las soluciones al examen se dejarán en el campus virtual. Este test vale 4 puntos (sobre un total de 10 de la nota de Teoría).
- **Una vez empezado el examen no se puede salir del aula hasta finalizarlo.**
- En la **hoja de contestaciones** el verdadero se corresponderá con la **A**, y el falso con la **B**.

| | **V** | **F** | | |
|---|---|---|---|---|
| En C++, la forma canónica de una clase debe contener al operador "= =". | ❑ | ❑ | 1 | F |
| La complejidad temporal (en su caso peor) del siguiente fragmento de código es $O(n^2)$<br><br>int i, j, n, sum;<br>for (i = 4; i < n; i++) {<br>  for (j = i–3, sum = a[i-4]; j <= i; j++) sum += a[j];<br>  cout << "La suma del subarray " << i-4 << " es " << sum << endl; } | ❑ | ❑ | 2 | F |
| crear_pila(), cima(pila), apilar(pila,item) y desapilar(pila) son operaciones constructoras del tipo pila. | ❑ | ❑ | 3 | F |
| Se puede reconstruir un único árbol binario de búsqueda teniendo su recorrido por niveles. | ❑ | ❑ | 4 | V |
| La siguiente operación borrar definida para listas borra el elemento que está a la derecha del elemento apuntado por la posicion p:<br><br>borrar( lista, posicion) -> lista<br>VAR L1, L2: lista; x: item; p: posicion;<br>borrar( crear( ), p ) = crear( )<br>si p == primera( inscabeza( L1, x ) ) entonces<br>  borrar( inscabeza( L1, x ), p ) = inscabeza( L1, x )<br>  si no borrar( inscabeza( L1, x ), p ) = inscabeza( borrar( L1, p ), x) | ❑ | ❑ | 5 | F |
| En C++, cuando se sobrecarga un operador que modifica al operando izquierdo (por ejemplo, el operador asignación), no se debe crear un objeto temporal que luego el método devuelva por valor. | ❑ | ❑ | 6 | V |
| El ítem medio (según la relación de orden de los elementos) almacenado en un árbol binario de búsqueda lleno siempre se encuentra en la raíz. | ❑ | ❑ | 7 | V |
| En un árbol AVL cuya raíz tiene un factor de equilibrio de +1, al insertar un elemento en el subárbol izquierdo de la raíz, el árbol siempre vuelve al estado de equilibrio (FE == 0). | ❑ | ❑ | 8 | F |
| Un árbol AVL completo es un árbol completamente equilibrado. | ❑ | ❑ | 9 | F |
| En el algoritmo del borrado de un elemento en un árbol 2-3-4 siempre que "p" sea 2-nodo hay que hacer una reestructuración. | ❑ | ❑ | 10 | F |
| La complejidad temporal en el peor caso de la operación inserción en un árbol 2-3-4 es $\log_2(n+1)$. | ❑ | ❑ | 11 | V |
| En un árbol 2-3, la altura siempre disminuye si la raíz es de tipo 2-nodo y al efectuar el borrado de un elemento es necesario realizar una combinación con el nodo raíz. | ❑ | ❑ | 12 | V |
| El TAD Cola de Prioridad en el que no se permiten elementos repetidos, representado por una lista desordenada, tendrá coste O(n) para la inserción, con n el número de elementos del TAD. | ❑ | ❑ | 13 | V |
| Sea una tabla de dispersión cerrada con estrategia de redispersión $h_i(x)=(H(x) + C*i)$ MOD B, con B=1000 y C=74. Para cualquier clave "x" que se desee insertar, se recorrerán todas las posiciones de la tabla buscando una posición libre. | ❑ | ❑ | 14 | F |
| La representación de un grafo mediante una lista de adyacencia siempre va a ser mejor tanto espacial como temporalmente que la representación mediante una matriz de adyacencia. | ❑ | ❑ | 15 | F |
| Un árbol binario de búsqueda completo con 4 elementos también es un montículo o HEAP mínimo. | ❑ | ❑ | 16 | F |
| Un árbol extendido en anchura de un grafo dirigido tiene el mismo número de vértices que el grafo original. | ❑ | ❑ | 17 | V |
| Un bosque extendido en profundidad de un grafo no dirigido es un grafo acíclico. | ❑ | ❑ | 18 | V |

# *Examen PED julio 2021*

**Normas:**
- Tiempo para efectuar el examen: **2 horas**
- En la cabecera de cada hoja **Y EN ESTE ORDEN** hay que poner: ***APELLIDOS, NOMBRE***.
- Cada pregunta se escribirá en hojas diferentes.
- Se puede escribir el examen con lápiz, siempre que sea legible

**1.** Utilizando las operaciones definidas en clase para el tipo árbol binario, define la sintaxis y la semántica de la operación Es_AVL que indica si un árbol binario (cuyas etiquetas son números naturales) es un árbol AVL. **(1,5 puntos)**
**Nota**: se asume que está definido el TAD de los números enteros con todas sus operaciones.

**2. a)** Dada la siguiente función, calcula su complejidad temporal. Para ello, deberás identificar la talla del problema, identificar los casos mejor y peor (si los hay), y calcular las cotas de complejidad, así como las clases de equivalencia de notación asintótica. **(0,5 puntos)**

```
void examen1(int vector1[], int n){
 for(int i=0; i< 2; i++){
  for(int j=0; j<=i; j++){
   int prod=1;
   int k=0;
   while(prod != 0 && k< n){
    prod=prod*vector1[k];
    k++;
   }
   cout << prod << endl;
   j=i+1;
  }
 }
}
```

**b)** Realiza el mismo proceso para esta otra función, que realiza la unión de dos conjuntos del mismo tamaño n representados como arrays de C++, y la almacena en un tercer array. Puedes asumir que el tercer array tiene tamaño suficiente para albergar la unión. **(0,5 puntos)**

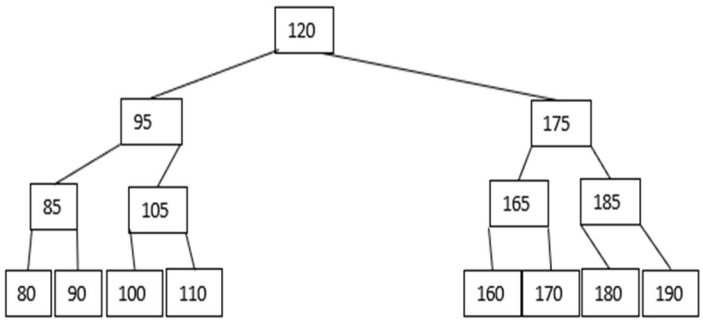
```
void une_conjuntos(int conjunto1[], int
conjunto2[], int n, int conjuntounion[], int &
tam_union){
 for( int i=0; i<n; i++){
  conjuntounion[i]=conjunto1[i];
 }
 tam_union=n;
 for (int i=0; i<n; i++){
  bool encontrado=false;
  int j=0;
  while(!encontrado && j< n){
   if(conjunto1[j] == conjunto2[i])
    encontrado=true;
   j++;
  }
  if(! encontrado){
   conjuntounion[tam_union]=conjunto2[i];
   tam_union++;
  }
 }
}
```

**c)** Suponiendo que los arrays están ordenados, escribe una función en C++ que realice el cálculo de la unión entre conjuntos. El algoritmo debe tener la complejidad temporal en el peor caso más baja posible. Indica justificadamente cuál es la complejidad temporal asintótica en el peor caso de ese algoritmo. La función debe tener los siguientes parámetros:

```
void une_conjuntos_ordenado(int conjunto1[], int
conjunto2[], int n, int conjuntounion[], int & tam_union){
...
}
```

**(0,5 puntos)**

**3.** Realizad el borrado de la clave 120:
**a)** Considerando que el siguiente árbol es 2-3
**b)** Considerando que el siguiente árbol es 2-3-4
**c)** Considerando que el siguiente árbol es AVL
NOTA: se sustituirá por el mayor del subárbol izquierdo en caso de estar en un nodo interior, y para las reestructuraciones se consultará el hermano de la izquierda.
**(Cada apartado vale 0,5 puntos).**




**4.** Dada esta secuencia de elementos [23, 14, 10, 15, 3, 5, 7, 8, 36, 47, 4]:
**1)** Insertar los elementos en una tabla de dispersión HASH CERRADA de tamaño B=11, sin reestructurar la misma, con función de dispersión **H(x) = x MOD B** y con estrategia de REDISPERSIÓN "**2ª FUNCIÓN HASH**". **(0,5 puntos)**
**2)** Indica el número de intentos necesarios para almacenar cada elemento. Indica el número de intentos necesarios para almacenar el total de elementos. **(0,1 puntos)**
**3)** De acuerdo al significado del FACTOR DE CARGA (**α**), ¿Habría que efectuar una reestructuración de la tabla HASH? Caso de responder SÍ, ¿a partir de la inserción de qué elemento se recomienda reestructurar la tabla? **(0,2 puntos)**
**4)** Caso de haber respondido SÍ en **3)**, ¿qué tamaño debería tener esa tabla tras la reestructuración? Explícalo. **(0,1 puntos)**
**5)** Insertar los mismos elementos en una tabla de dispersión HASH ABIERTA de tamaño B=11. **(0,5 puntos)**
**6)** Indica el número de intentos necesarios para almacenar cada elemento. Indica el número de intentos necesarios para almacenar el total de elementos. **(0,1 puntos)**

# *Examen PED julio 2021. Soluciones*

**1.**

*sintaxis:*

*Es_AVL: arbin -> bool*

*semántica:*

*var i,d: arbin; x,y:entero;*

*Es_AVL(crear_arbin()) = VERDADERO*

*Es_AVL(enraizar (i,x,d) =*
*si ( [x>raiz(i)] Y [x<raiz(d)] ) Y*
*( [ (altura(d) – altura(i)) == 0 ] O*
*[ (altura(d) – altura(i)) == 1 ] O*
*[ (altura(d) – altura(i)) == -1 ] )*
*entonces*
*Es_AVL(i) Y Es_AVL(d)*
*sino*
*FALSO*

**2.**

a)

La talla del problema viene determinada por el valor del parámetro n. A mayor valor de n, mayor número de iteraciones del bucle while.

El mejor caso se produce cuando el primer elemento del array tiene el valor 0. En ese caso, se ejecutará una única iteración del bucle while.

El peor caso se produce cuando no hay ningún 0 en el array. En ese caso, el bucle while se ejecuta n veces.

En el mejor caso, podemos considerar toda la función como un único paso de programa, ya que el primer bucle for se ejecuta siempre 2 veces. En cada una de estas iteraciones, el segundo bucle for se ejecuta una sola vez, y el bucle while también se ejecuta una sola vez. Por tanto, el coste temporal en el mejor caso o cota inferior $c_i(n)$, sería $c_i(n)=1$. Y $c(n) \in \Omega(1)$

En el peor caso, el bucle while siempre se ejecutará n veces. Por tanto, podemos calcular el número de pasos de programa que ejecutan mediante la siguiente expresión:

$c_s(n) = \Sigma_{i=0..1}( \ 1 \ + \ \Sigma_{k=0..n-1} 1 \ ) = \Sigma_{i=0..1}( \ 1 \ + \ n \ ) = 2n + 2.$

Por tanto, $c(n) \in O(n)$.

b) La talla del problema viene determinada por el valor del parámetro n, es decir, el tamaño de los conjuntos.

El mejor caso se produce cuando los dos conjuntos son iguales. Encontraremos cada elemento del conjunto 2 en el conjunto 1 y no habrá que recorrer todo el conjunto 1 en el bucle while.

El peor caso se produce cuando los conjuntos son disjuntos: no encontramos ningún elemento del conjunto 2 en el conjunto 1 y el bucle while siempre recorre todo el conjunto 1.

Para simplificar cálculos en el mejor caso, vamos a asumir que los dos conjuntos contienen los elementos en el mismo orden. El coste temporal en el mejor caso o cota inferior vendrá determinado por la expresión:

$c_i(n) = \ \Sigma_{i=0..n-1} \ 1 + 1 + \ \Sigma_{i=0..n-1} ( \ 1 + \Sigma_{j=0..i+1} \ 1 \ ) = \Sigma_{i=0..n-1} \ 1 + 1 + \ \Sigma_{i=0..n-1} ( \ 1 + i + 1 \ ) =$

$= \Sigma_{i=0..n-1} \ 1 + 1 + \ n*(2 + n+1)/2 = n + 1 + n + n^2/2 + n/2$

$c(n) \in \Omega(n^2)$

El coste temporal en el peor caso o cota superior vendrá determinado por la expresión:

$c_i(n) = \ \Sigma_{i=0..n-1} \ 1 + 1 + \ \Sigma_{i=0..n-1} ( \ 1 + \Sigma_{j=0..n-1} \ 1 \ ) = \Sigma_{i=0..n-1} \ 1 + 1 + \ \Sigma_{i=0..n-1} ( \ 1 + n \ ) =$

$= n + 1 + \ n( \ 1 + n \ )$

$c(n) \in O(n^2)$

$c(n) \in \Theta(n^2)$

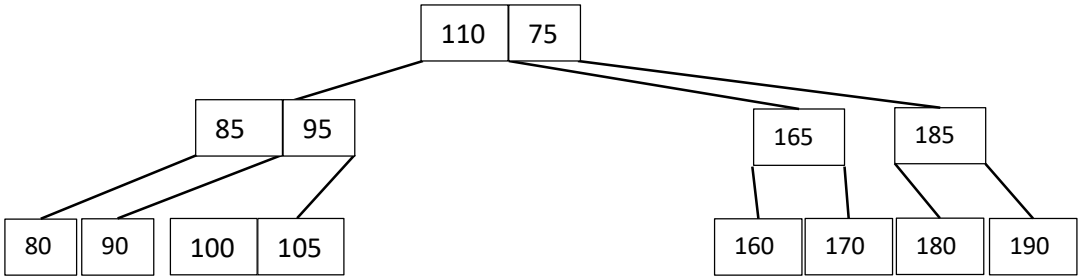
c)

```
void une_conjuntos_ordenados(int conjunto1[], int conjunto2[], int n, int conjuntounion[], int & tam_union){
  tam_union=0;
  int pos1=0;
  int pos2=0;

  while(pos1< n && pos2 < n){
    if(conjunto1[pos1] == conjunto2[pos2]){
      conjuntounion[tam_union]=conjunto1[pos1];
      tam_union++;
      pos1++;
      pos2++;
    }else if(conjunto1[pos1] < conjunto2[pos2]){
      conjuntounion[tam_union]=conjunto1[pos1];
      tam_union++;
      pos1++;
    } else{
      conjuntounion[tam_union]=conjunto2[pos2];
      tam_union++;
      pos2++;
    }
  }
  for(int i=pos1; i<n; i++){
    conjuntounion[tam_union]=conjunto1[i];
    tam_union++;
  }
  for(int i=pos2; i<n; i++){
    conjuntounion[tam_union]=conjunto2[i];
    tam_union++;
  }

}
```

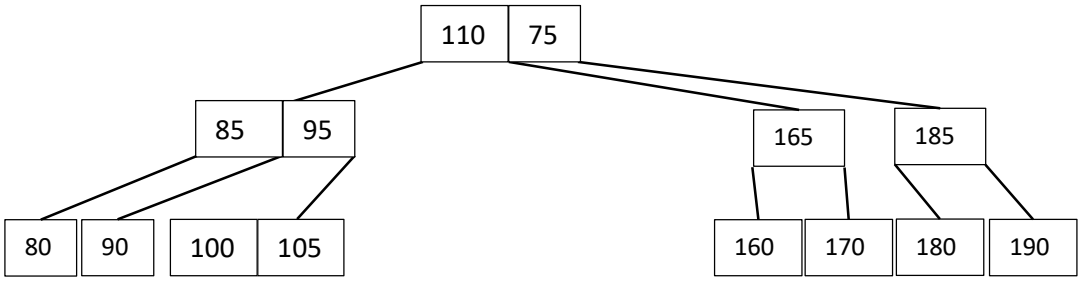
El coste temporal en el peor caso es proporcional a O(n). Cada uno de los bucles tiene un único paso de programa en su interior. Y el número total de iteraciones, en el peor caso, es 2n, siendo n el tamaño de los conjuntos.

**3.**

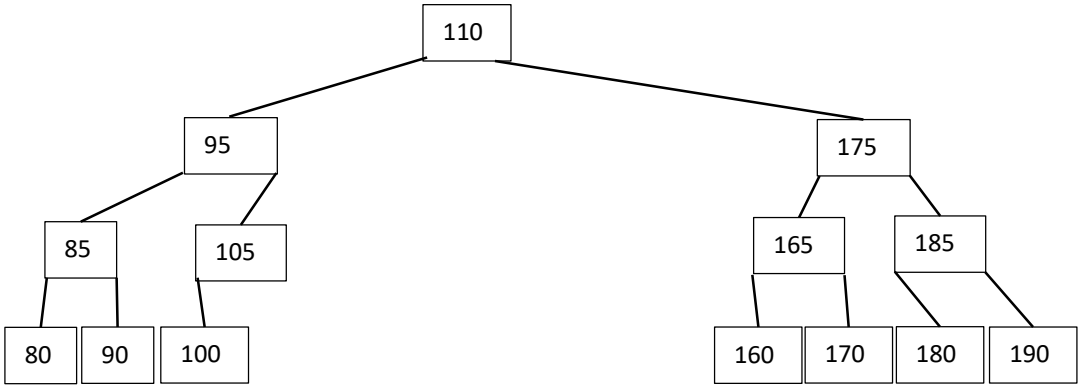
a) 3 combinaciones




b) 3 combinaciones




c) Ninguna rotación



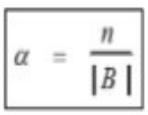
## 4.a.1) HASH CERRADO
## 4.a.2) NÚM. INTENTOS (PARCIALES / TOTAL)

**[23, 14, 10, 15, 3, 5, 7, 8, 36, 47, 4]** ***B=11***
***H(x)= x MOD 11***
**k (x) = (x MOD (10)) + 1**

| | | | | | |
|---|---|---|---|---|---|
| 0 | **47** | **H(47) = 3** | (k(47) = 8) | **h1(47)= (3+8) MOD 11= 0** | **(2º INTENTO)** |
| 1 | **23** | **H(23) = 1** | | | **(1º INTENTO)** |
| 2 | **4** | **H(4) = 4 ;** | (k(7) = 5) | **h1(4)= (4+5) MOD 11= 9**<br>**h2(4)= (9+5) MOD 11= 3**<br>**h3(4)= (3+5) MOD 11= 8**<br>**h3(4)= (8+5) MOD 11= 2** | **(5º INTENTO)** |
| 3 | **14** | **H(14) = 3 ;** | | | **(1º INTENTO)** |
| 4 | **15** | **H(15) = 4** | | | **(1º INTENTO)** |
| 5 | **5** | **H(5) = 5** | | | **(1º INTENTO)** |
| 6 | **36** | **H(36) = 3 ;** | (k(36) = 7) | **h1(36)= (3+7) MOD 11= 10**<br>**h2(36)= (10+7) MOD 11= 6** | **(3º INTENTO)** |
| 7 | **3** | **H(3) = 3** | (k(3) = 4) | **h1(3)= (3+4) MOD 11= 7** | **(2º INTENTO)** |
| 8 | **8** | **H(8) = 8** | | | **(1º INTENTO)** |
| 9 | **7** | **H(7) = 7 ;** | (k(7) = 8) | **h1(7)= (7+8) MOD 11= 4**<br>**h2(7)= (4+8) MOD 11= 1**<br>**h3(7)= (1+8) MOD 11= 9** | **(4º INTENTO)** |
| 10 | **10** | **H(10) = 10** | | | **(1º INTENTO)** |

**22 INTENTOS**

## 4.a.3) REESTRUCTURACIÓN (SEGÚN "FACTOR DE CARGA")




**α** = "Factor de carga" → nivel de ocupación de la tabla **HASH .** En HASH Cerrado se cumple: **[0 <= α <= 1]**
**( n** = num. de elementos insertados)
**( B** = tamaño de la tabla )

**En HASH CERRADO se reestructura tabla cuando α ≥ 0,9**
→ Por tanto, cuando **n/B ≥ 0,9**
→ Por tanto, cuando **n ≥ 0,9 * B**
→ Por tanto, en este caso cuando **n ≥ 0,9 * 11** → Por tanto, en este caso cuando **n ≥ 9,9**
→ Por tanto, SÍ , hay que reestructurar después de insertar el **elemento 9º** (que es el **36**) y antes de insertar el **elemento 10º** (que es el **47**)

## 4.a.4) REESTRUCTURACIÓN (II)

Reestructurar a tamaño de tabla TEORICO **2*B= 22**
Sin embargo, se debe seguir cumpliendo restricción HASH CERRADO : "B debe ser nº PRIMO" → por tanto, se puede reestructurar al nº PRIMO más cercano a 22 (valen las 2 opciones):

a) **B = 19**
b) **B = 23**

**4.a.5) HASH ABIERTO**
**4.a.6) NÚM. INTENTOS (PARCIALES / TOTAL)**

**[23, 14, 10, 15, 3, 5, 7, 8, 36, 47, 4]** ***B=11***
***H(x)= x MOD 11***

| | | | | |
|---|---|---|---|---|
| 0 | | | | |
| 1 | **23**(1ºINTENTO) | | | |
| 2 | | | | |
| 3 | **14**(1ºINTENTO) | **3** (2º INTENTO) | **36**(3º INTENTO) | **47**(4º INTENTO) |
| 4 | **15**(1ºINTENTO) | **4** (2º INTENTO) | | |
| 5 | **5** (1º INTENTO) | | | |
| 6 | | | | |
| 7 | **7** (1º INTENTO) | | | |
| 8 | **8** (1º INTENTO) | | | |
| 9 | | | | |
| 10 | **10**(1ºINTENTO) | | | |

**18 INTENTOS**