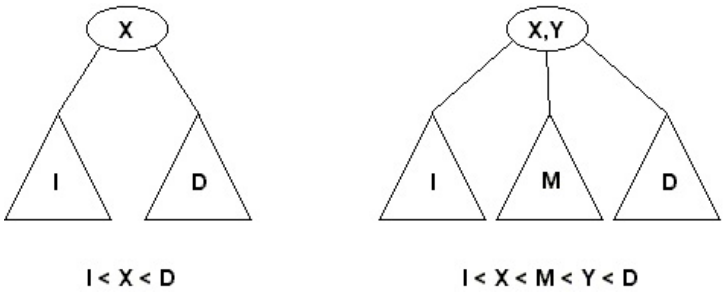
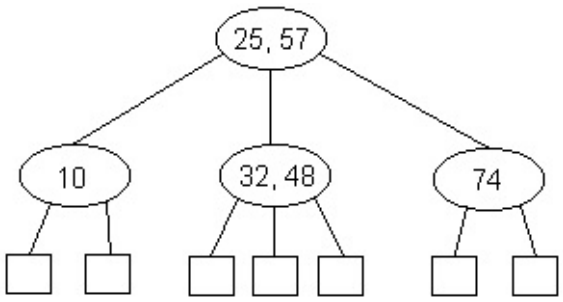
Nombre y Apellido:

# Examen Parcial 2

**1.** Los árboles $2-3$ son árboles cuyos nodos internos pueden contener hasta 2 elementos (todos los árboles vistos con anterioridad pueden contener sólo un elemento por nodo) y, por lo tanto, un nodo interno puede tener 2 o 3 hijos, dependiendo de cuántos elementos posea el nodo. De este modo, un nodo de un árbol $2-3$ puede tener una de las siguientes formas:

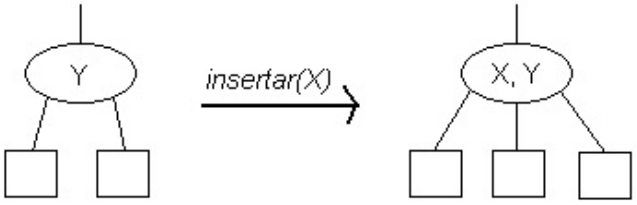



Una propiedad de los árboles $2-3$ es que todas las hojas están a la misma profundidad, es decir, los árboles $2-3$ son árboles perfectamente balanceados. La siguiente figura muestra un ejemplo de un árbol $2-3$ donde los cuadrados son hojas:

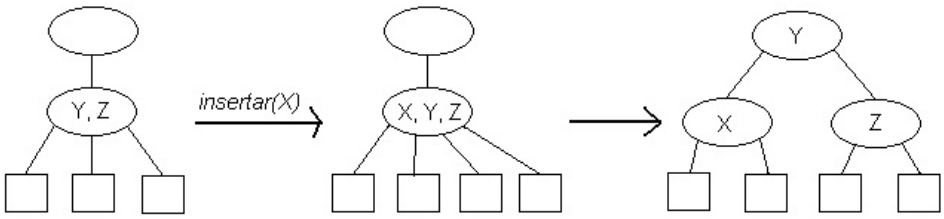



Para insertar un elemento $X$ en un árbol $2-3$ se realiza una búsqueda infructuosa y se inserta dicho elemento en el último nodo visitado durante la búsqueda, lo cual implica manejar dos casos distintos:

- Si el nodo donde se inserta $X$ tenía una sola llave (dos hijos), ahora queda con dos llaves (tres hijos).

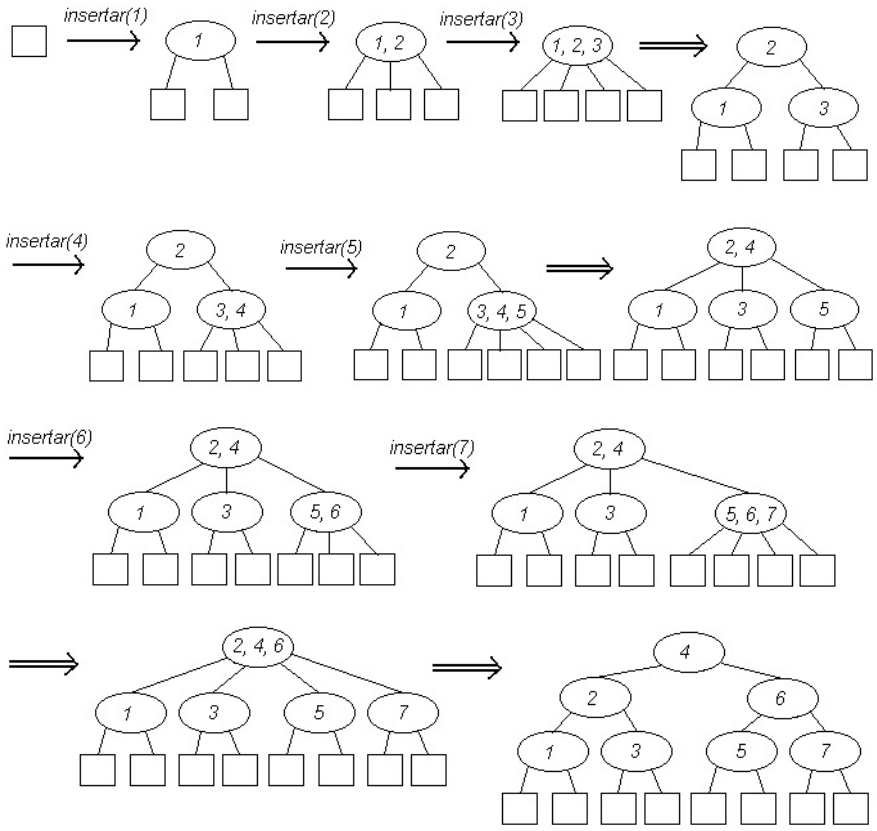



- Si el nodo donde se inserta $X$ tenía dos llaves (tres hijos), queda transitoriamente con tres llaves (cuatro hijos) y se dice que está saturado (overflow).




El problema se resuelve a nivel de $X$ y $Z$, pero es posible que el nodo que contiene a $Y$ ahora este saturado. En este caso, se repite el mismo prodecimiento anterior un nivel más arriba. Finalmente, si la raíz es el nodo saturado, éste se divide y se crea una nueva raíz un nivel más arriba. Esto implica que los árboles $2-3$ crecen "hacia arriba".

Ejemplos de inserción en árboles $2-3$:




Considere la siguiente definición de la estructura `Arbol23`:

```
typedef struct nodo_arbol23* Arbol23;

// Representa un arbol especial utilizado para marcar que un arbol es binario
   poniendolo como tercer hijo
#define CENTINELA ((Arbol23)1)
```

```
// Nodo de arbol 2-3
// Los nodos con un solo valor (solo dos hijos) tienen right=CENTINELA
struct nodo_arbol23 {
    int k1, k2;
    Arbol23 left, middle, right;
};
```

Donde:

- $k1 < k2$;
- los datos de *left* se asumen menores a $k1$;
- los datos de *middle* se asumen mayores a $k1$ y menores a $k2$;
- los datos de *right* se asumen mayores a $k2$;

Se pide que implemente la inserción en árboles $2-3$ siguiendo los siguientes pasos:

**a)** Implemente la función

```
Arbol23 arbol23_crear();
```

que crea un árbol $2-3$ vacío.

**b)** Implemente la función

```
int cantidad_de_valores(Arbol23 t);
```

que recibe un árbol $2-3$ y devuelve la cantidad de valores que guarda el nodo raíz.

**c)** Implemente la función

```
Arbol23 crear_nodo_binario(int x, Arbol23 l, Arbol23 r);
```

que dado un valor y dos árboles $2-3$ crea un nodo binario con el valor como dato y los dos árboles como hijos.

**d)** Implemente la función

```
int insertar_impl(Arbol23 t, int x, int* out_x, Arbol23* out_l, Arbol23*
    out_r);
```

que inserta el valor `x` en el árbol `t` siguiendo las reglas de inserción del árbol $2-3$ y conservando sus invariantes.

- En caso de saturarse `t`, la función devuelve `1` (*true*) y guarda en `out_l` el subárbol izquierdo, en `out_r` el subárbol derecho y en `out_x` el dato.
- En caso de no saturarse `t`, la función devuelve `0` (*false*) y no guarda nada en los punteros pasados por argumento.

**2.** Considere las siguientes funciones, las cuales son parte de la implementación de *max-heap* con arreglos.

```
// Intercambia los valores entre dos punetros a entero.
void swap_int(int* x, int* y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

```
// Devuelve la posicion del padre del elemento en la posicion i
int padre(int i) {
    return (i-1)/2;
}

// Devuelve la posicion del hijo izquierdo del elemento en la posicion i
int izq(int i) {
    return i*2+1;
}

// Devuelve la posicion del hijo derecho del elemento en la posicion i
int der(int i) {
    return i*2+2;
}

// Hunde el elemento en la posicion pos del arreglo hasta que este cumpla la
   condicion de max-heap. n es la cantidad de elementos del arreglo
void hundir(int pos, int* arr, int n) {
   int max_pos = pos;
   if (izq(pos) < n && arr[izq(pos)] > arr[max_pos]) max_pos = izq(pos);
   if (der(pos) < n && arr[der(pos)] > arr[max_pos]) max_pos = der(pos);
   if (max_pos != pos) {
      swap_int(&arr[pos], &arr[max_pos]);
      hundir(max_pos, arr, n);
   }
}

// Flota el elemento en la posicion pos hasta que el arreglo cumpla con la
   condicion de max-heap. n es la cantidad de elementos del arreglo.
void flotar(int pos, int* arr, int n) {
   if (pos == 0) return;
   int pos_padre = padre(pos);
   if (arr[pos_padre] < arr[pos]) {
      swap_int(&arr[pos], &arr[pos_padre]);
      flotar(pos_padre, arr, n);
   }
}
```

Defina una función:

```
    void ordenar(int* arr, int n);
```

que implemente un algoritmo de ordenamiento *in-place*, es decir, que ordene el arreglo `arr` de tamaño `n` dentro del espacio de memoria dado en lugar de devolver un nuevo arreglo ordenado, utilizando las funciones de manipulación de *max-heap* definidas previamente. Esta función debe definirse de la manera más eficiente posible ya que se evaluará la eficiencia del algoritmo. Además, no deben utilizarse estructuras adicionales.