Resolución de Tarea 6 - Árboles (Fecha: 10 de Noviembre de 2025)

Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI5651 - Diseño de Algoritmos I
Septiembre - Diciembre 2025
Estudiante: Junior Miguel Lara Torres (17-10303)

Tarea 6 (9 puntos)

Indice

Pregunta 1

Las pistas sugieren usar una estructura que permita **dividir o reunir subarreglos eficientemente con una alta probabilidad** y **evitar mover los elementos uno por uno**, por lo que la solución propuesta a continuación usará **treaps**.

Un Treap (combinación de *tree* y *heap*) es un árbol binario donde cada nodo tiene un par (clave, prioridad).

- El Treap es un árbol binario de búsqueda (BST) respecto a las claves.
- Es un *heap* respecto a las prioridades.
- Al asignar prioridades aleatorias a los nodos, **el Treap logra mantener un promedio de** $O(\log n)$ **niveles**, lo que lo hace útil para implementar BSTs eficientes.
- Llamaremos **Treap Implícito** a una variación de **treaps** donde las claves corresponden a los índices de una secuencia, sin ser almacenados explícitamente. Esta estructura es ideal para manejar operaciones sobre rangos en **tiempo** $O(\log n)$ **en promedio**.

La operación multiswap(a, b) implica modificar grandes sub-rangos del arreglo mediante intercambios secuenciales y recursivos. Dado que un Treap Implícito está diseñado para manipular eficientemente los subarreglos (rangos) de una secuencia en tiempo $O(\log N)$ en promedio, modelar la permutación $A[1..N]$ como un Treap Implícito permite implementar la operación eficientemente, evitando mover los elementos uno a uno.

Si cada una de las N operaciones multiswap se implementa utilizando las manipulaciones de Treap (que toman $O(\log N)$ en promedio), **la complejidad total promedio resultante es** $O(N \log N)$, cumpliendo con la restricción de tiempo propuesta.

Implementación en C++

El archivo funcional se encuentra en treaps.cpp

```cpp
struct Node {
  int value;
  int priority;
  int size;
  Node *left, *right;

  // Asignamos una prioridad aleatoria (crucial para el balanceo O(log n) promedio)
  Node(int val) : value(val), size(1), left(nullptr), right(nullptr) {
    priority = rand();
  }
};

int get_size(Node *t) {
  return t ? t->size : 0;
}

void update_size(Node *t) {
  if (t)
    t->size = get_size(t->left) + get_size(t->right) + 1;
}

/**
 * Operación SPLIT (División)
 * Divide el Treap 't' en dos ('l' con los primeros 'k' elementos y 'r' con el resto).
 * Tiempo promedio O(log n).
 */
void split(Node *t, Node *&l, Node *&r, int k) {
  if (!t) {
    l = r = nullptr;
    return;
  }

  int current_rank = get_size(t->left) + 1;

  if (k < current_rank) {
    split(t->left, l, t->left, k);
    r = t;
  } else {
    split(t->right, t->right, r, k - current_rank);
    l = t;
  }
  update_size(t);
}

/**
 * Operación MERGE (Unión)
 * Combina dos Treaps 'l' y 'r' en 't', manteniendo la propiedad de heap.
 * Tiempo promedio O(log n).
 */
void merge(Node *&t, Node *l, Node *r) {
  if (!l) {
    t = r;
    return;
  }
  if (!r) {
    t = l;
    return;
  }

```

```
  // Se elige la raíz basada en la prioridad (la de menor prioridad va arriba)
  if (l->priority < r->priority) {
    merge(l->right, l->right, r);
    t = l;
  } else {
    merge(r->left, l, r->left);
    t = r;
  }
  update_size(t);
}

/**
 * Implementa la reordenación de subarreglos simulando multiswap(a, b).
 * Intercambia el segmento T1 = A[a..b-1] por T2 = A[b..2b-a-1].
 */
void multiswap_treap(Node *&t, int a, int b, int N) {
  int L = b - a;
  int end_T2 = b + L - 1;

  // a y b están basados en 1. Las operaciones split esperan un número de elementos.
  if (end_T2 > N || L <= 0)
    return;

  Node *T_prefix = nullptr, *T_rest = nullptr;
  Node *T1 = nullptr, *T_T2_suffix = nullptr;
  Node *T2 = nullptr, *T_suffix = nullptr;

  // 1. Separar T_prefix (A[1..a-1])
  split(t, T_prefix, T_rest, a - 1);

  // 2. Separar T1 (A[a..b-1]). Longitud L.
  split(T_rest, T1, T_T2_suffix, L);

  // 3. Separar T2 (A[b..b+L-1]). Longitud L.
  split(T_T2_suffix, T2, T_suffix, L);

  // 4. Re-unir en el orden: T_prefix, T2, T1, T_suffix
  Node *T_new_1 = nullptr, *T_new_2 = nullptr;

  merge(T_new_1, T_prefix, T2);
  merge(T_new_2, T_new_1, T1);
  merge(t, T_new_2, T_suffix);
}

// Recorrido In-order para obtener la secuencia resultante
vector<int> inorder_traversal(Node *t) {
  vector<int> result;
  function<void(Node *)> traverse = [&](Node *node) {
    if (!node)
      return;
    traverse(node->left);
    result.push_back(node->value);
    traverse(node->right);
  };
  traverse(t);
  return result;
}
```

Pregunta 2

Implementación en C++

Pregunta 3

Implementación en C++