Justificación de recorrido_con_max_beneficio: (PRIVATE)

CÓDIGO:

ESPECIFICACIÓN:

```
/**
 * @brief Obtiene la ruta más beneficiosa para el barco.
 *
 * @param nodo Árbol binario que representa la ciudad donde estamos localizados.
 *
 * @return Lista de nombres de ciudades.
 *
 * @pre Se proporciona el árbol binario de la cuenca.
 * @post Se obtiene la ruta más beneficiosa para el barco.
 */
list<pair<char,int>> recorrido_con_max_beneficio(const BinTree<pair<int,int>>& nodo);
```

IMPLEMENTACIÓN:

```cpp
list<pair<char,int>> Cuenca::recorrido_con_max_beneficio(const BinTree<pair<int,int>>& nodo){
    // Caso Base ##########################################################################################

    // Si el nodo es vacío
    if (nodo.empty()) return list<pair<char,int>>();

    // Paso Inductivo #####################################################################################

    // Construimos las listas postorden
    list<pair<char,int>> recorido_izquierda = recorrido_con_max_beneficio(nodo.left());
    list<pair<char,int>> recorido_derecha = recorrido_con_max_beneficio(nodo.right());
    //--------------------------------

    // Inicializamos los beneficios por ruta
    int beneficio_izquierda = 0, beneficio_derecha = 0;

    // Beneficios por ruta
    if (!recorido_izquierda.empty()) beneficio_izquierda = recorido_izquierda.front().second;
    if (!recorido_derecha.empty()) beneficio_derecha = recorido_derecha.front().second;
    //--------------------

    // Calculamos el beneficio en nuestra posición
    int beneficio_nodo_actual = nodo.value().first + nodo.value().second;

    // Distinguimos por diferentes casos:

    // En caso de que el beneficio de la parte izquierda y la parte derecha sea 0, entonces finalizamos la función
    if (beneficio_izquierda + beneficio_derecha == 0) {
        // Devolvemos la lista formada únicamente por un elemento
        list<pair<char,int>> lista_unica;
        // Char u -> Representa una única ciudad en la lista
        lista_unica.push_front(make_pair('u',beneficio_nodo_actual));
        return lista_unica;
    }

    // Si el beneficio de la derecha es mayor que el de la izquierda
    if (beneficio_derecha > beneficio_izquierda){
```

```cpp
        // Si beneficio_derecha és mayor que beneficio_izquierda quiere decir que es mayor que 0 en el peor de los casos cosa que quiere decir que existe.

        // Incrementamos el beneficio
        beneficio_nodo_actual += recorido_derecha.front().second;
        // Añadimos el nuevo nodo y la dirección por la que debemos ir
        recorido_derecha.push_front(make_pair('d',beneficio_nodo_actual));
        // Devolvemos el recorrido hecho
        return recorido_derecha;

    } // Si el beneficio de la izquierda es mayor que el de la derecha
    else if (beneficio_derecha < beneficio_izquierda){
        // Si beneficio_izquierda és mayor que beneficio_izquierda quiere decir que es mayor que 0 en el peor de los casos cosa que quiere decir que existe.

        // Incrementamos el beneficio
        beneficio_nodo_actual += recorido_izquierda.front().second;
        // Añadimos el nuevo nodo y la dirección por la que debemos ir
        recorido_izquierda.push_front(make_pair('i',beneficio_nodo_actual));
        // Devolvemos el recorrido hecho
        return recorido_izquierda;
    }
    else { // Si el beneficio de la izquierda es igual que el de la derecha
        // Miramos el recorrido más corto de los dos, si son iguales escogeremos el de la izquierda
        // Si el recorrido de la derecha es más pequeño que el de la izquierda
        if (recorido_derecha.size() < recorido_izquierda.size()){
            // Entonces, como existe el recorrido por la derecha: (Explicado arriba)
            // Incrementamos el beneficio
            beneficio_nodo_actual += recorido_derecha.front().second; //Incrementamos el beneficio
            // Añadimos el nuevo nodo y la dirección por la que debemos ir
            recorido_derecha.push_front(make_pair('d',beneficio_nodo_actual));
            // Devolvemos el recorrido hecho
            return recorido_derecha;
        }
        else { // Si el recorrido de la izquierda es más pequeño o igual que el de la derecha
            // Entonces, si existe el recorrido por la izquierda
            // Incrementamos el beneficio
            if (!recorido_izquierda.empty()) beneficio_nodo_actual += recorido_izquierda.front().second; //Incrementamos el beneficio
            // Añadimos el nuevo nodo y la dirección por la que debemos ir
            recorido_izquierda.push_front(make_pair('i',beneficio_nodo_actual));
            // Devolvemos el recorrido hecho
```

```
            return recorido_izquierda;
        }
    }
}
```

Paso 1: Caso base:

> Si el **nodo es vacío**, retornamos una lista vacía porque no hay ningún recorrido posible.

Paso 2: H.I.:

> Suponemos que '**recorrido_con_max_beneficio**' produce resultados correctos para subárboles más pequeños que el árbol actual por la hipótesis de inducción. Entonces, después de las llamadas recursivas '**recorrido_con_max_beneficio(nodo.left())**' y '**recorrido_con_max_beneficio(nodo.right())**', los recorridos 'recorrido_izquierda' y 'recorrido_derecha' contienen las listas de pares <dirección, beneficio> con el máximo beneficio acumulado desde los respectivos subárboles **izquierdo** y **derecho**.

Paso 3: Paso Inductivo:

> La función '**recorrido_con_max_beneficio**' determina el beneficio acumulado de cada subárbol:
>
> - Si el ***subárbol izquierdo*** está vacío, el beneficio acumulado de 'recorrido_izquierda' es 0.

- Si el ***subárbol derecho*** está vacío, el beneficio acumulado de 'recorrido_derecha' es 0.

Se calcula el beneficio del nodo actual como la suma del valor del nodo y el beneficio acumulado del camino seleccionado (izquierdo o derecho) con mayor beneficio.

La función luego compara los beneficios acumulados de ambos subárboles:

- Si el beneficio del **subárbol derecho es mayor,** se selecciona el camino derecho, se incrementa el beneficio del nodo actual, se añade el nodo actual al comienzo de la lista de '***recorrido_derecha***' y se retorna esta lista.
- Si el beneficio del **subárbol izquierdo es mayor,** se selecciona el camino izquierdo, se incrementa el beneficio del nodo actual, se añade el nodo actual al comienzo de la lista de '***recorrido_izquierda***' y se retorna esta lista.
- Si los **beneficios son iguales**, se selecciona el camino más corto; en caso de igualdad de longitud, se elige el camino izquierdo, se incrementa el beneficio del nodo actual, se añade el nodo actual al comienzo de la lista de 'recorrido_izquierda' y se retorna esta lista.

Como las llamadas recursivas operan sobre subárboles más pequeños y se combinan correctamente los resultados, por la hipótesis de inducción y el correcto manejo de los beneficios, la función '**recorrido_con_max_beneficio**' retorna una lista que representa el camino con el máximo beneficio acumulado.

Paso 4: Terminación:

> Una función de cota para '**recorrido_con_max_beneficio**' viene definida por la profundidad del nodo actual en el árbol, que disminuye con cada llamada recursiva:
>
> - Profundidad del nodo:
>   - Si el nodo no es vacío, la **profundidad disminuye en 1** con cada llamada recursiva a 'nodo.left()' y 'nodo.right()'.
>   - Si el **nodo es vacío**, la ***recursión termina*** inmediatamente con el caso base retornando una lista vacía.
>
> Por lo tanto, la recursión eventualmente llega al caso base, asegurando que la función '**recorrido_con_max_beneficio**' siempre termina.

Conclusión:

> El método **Cuenca**::**recorrido_con_max_beneficio** es correcto.