**CONTENIDO**

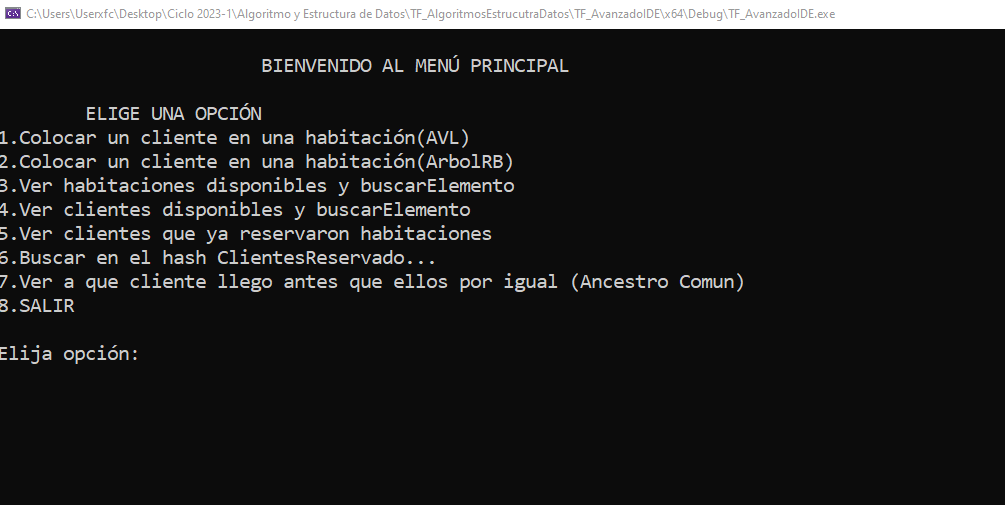
**Resumen**

A lo largo del curso se vieron diversas estructuras datos, desde listas enlazadas hasta hash tables, comprobando su eficiencia en diversas situaciones. Para este trabajo final se nos presenta la problemática de gestionar hoteles, el cual se trató de resolver con las estructuras de datos aprendidas a lo largo del curso, se espera de este trabajo haber obtenido los Student Outcomes respectivos, los cuales son:

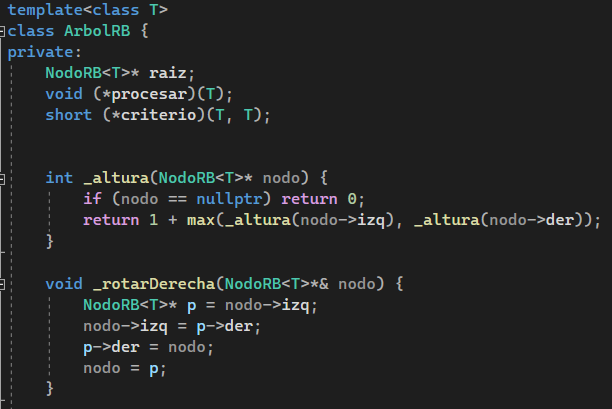
* En Ingeniería de Software, el logro contribuye a alcanzar el: ABET – EAC - Student Outcome 2: La capacidad de aplicar el diseño de ingeniería para producir soluciones que satisfagan necesidades específicas con consideración de salud pública, seguridad y bienestar, así como factores globales, culturales, sociales, ambientales y económicos.
* En Ciencias de la Computación, el logro contribuye a alcanzar el: ABET – CAC - Student Outcome 2: Diseñar, implementar y evaluar una solución basada en la computación para cumplir con un conjunto de requisitos computacionales en el contexto de la disciplina del programa.

**Diseño del producto**

* El diseño del producto inicia desde la problemática que se nos plantea, la problemática a resolver en cuestión es sobre como gestionar hoteles, a primera se pensó en justamente un software de ordenamiento de hoteles, es decir por precios o estrellas, etc. Sin embargo, se optó por un software que ordené a un hotel en cuestión. Adjunto imagen de la vista del prototipo:



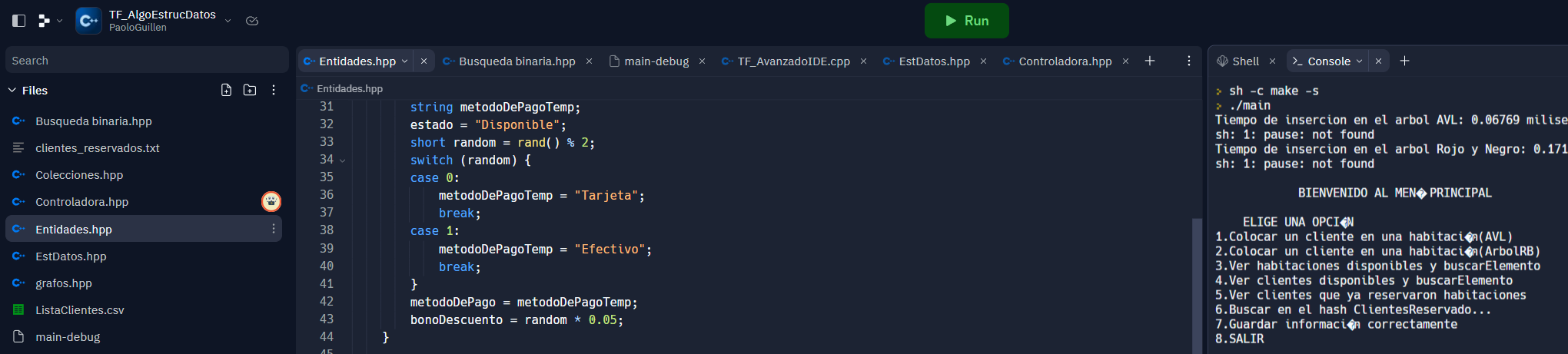
como se puede apreciar, es una gestión la cual manejaría los clientes de un hotel, resolviendo la problemática general que se nos brindó un software para gestión de hoteles, entre las buenas prácticas que se usaron fue la de aplicar estructuras de datos reutilizables, es decir que se implementaron con “templates” lo que permite que distintas entidades de objetos puedan usarlas debidamente con las funciones que estas tienen, además de utilizar una Programación Orientada a Objetos. Se adjunta código donde se aprecia “templates” y que la estructura puede ser usada con distintas variables justo por eso:



* El diseño de nuestro producto a pesar de ser un prototipo breve, consideramos que puede haber abordado esta problemática planteada y brindar una posible solución, sobre todo porque debido a la eficacia de estas estructuras de datos utilizadas, estructuras como arboles balanceados AVL y Rojo Y Negro o hashTables, se cree que puede ser tomado modelo temprano en caso fuese un proyecto de software de pequeña escala. Además, que el prototipo planteado soluciona la presente problemática de no poder llegar a encontrar ciertos clientes por lentitud del sistema o no poder asignarles sus habitaciones respectivas y tener un registro sobre esto.

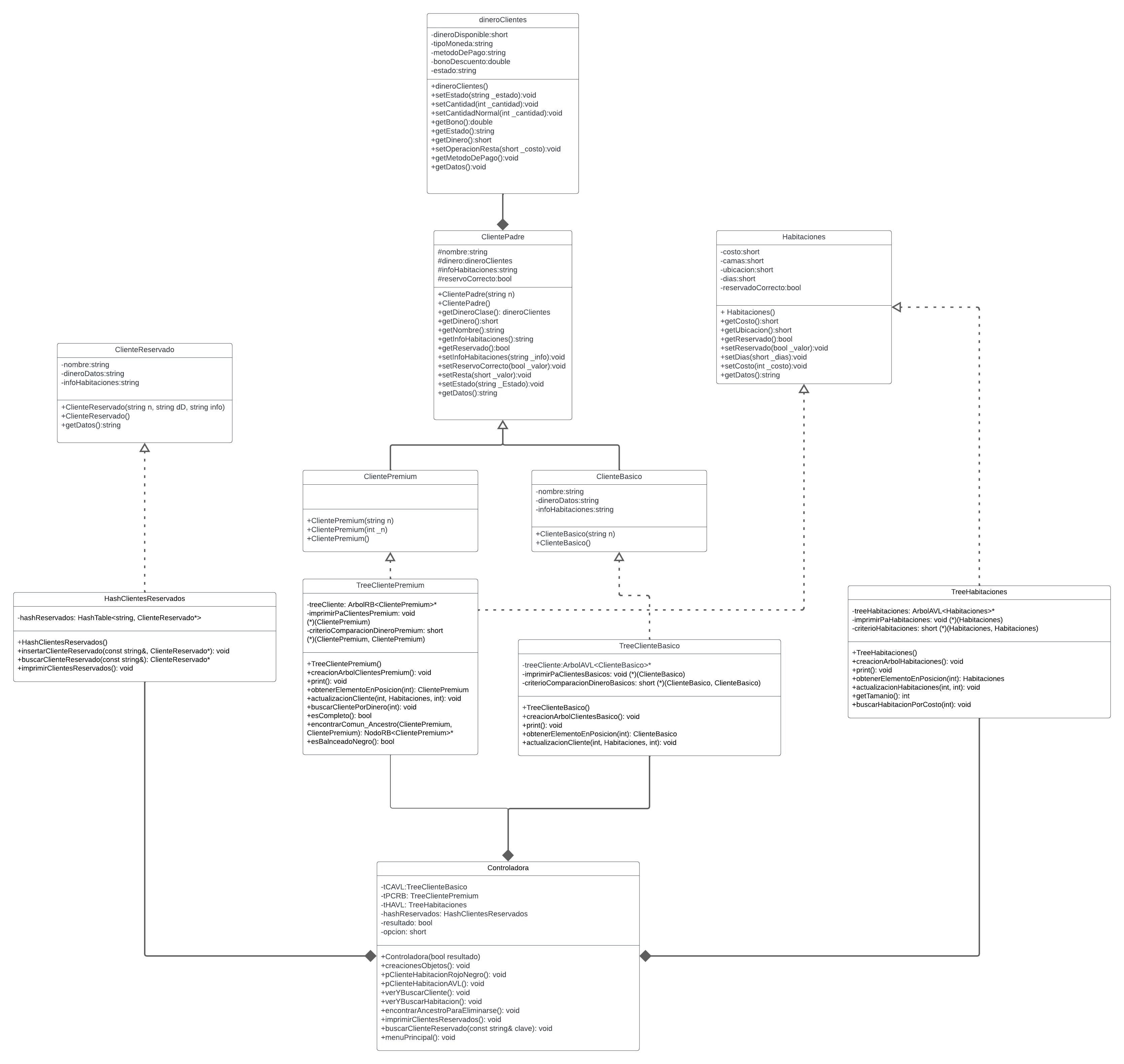
**Diseño del proyecto**

* El proyecto que se desarrolló para solucionar la problemática planteada se realizó en código C++, utilizando estructuras de datos relativamente complejas como árboles AVL y hashTable. La metodología que se usó en el diseño fue la de realizar las estructuras de datos con templates, esto para que en caso se quiera hacer cambios a futuros o añadir otras entidades que utilicen las funciones de estas mismas estructuras de datos, no se tenga que cambiar nada más de que tipo de variable es, como por ejemplo de un objeto “Cliente” a uno “Habitaciones”. El proyecto se planeó a pasos, desde hacer una breve tormenta de ideas, hasta encontrar con un prototipo que nos agradé y llamé la atención para poder realizarlo de forma adecuada. Para esto se realizó 2 proyectos pequeños los cuales uno era sobre manejar hoteles, es decir una cadena de hoteles y otro sobre un software que maneje individualmente a un hotel, el cual optamos por justamente el software que maneja individualmente a este. Al tener un tiempo limitado se tuvo que desarrollar esta solución sin tener que perder mucho tiempo, es decir sin realizar muchas pruebas para ver si el código está libre de errores, el código puede tener fallas imprevistas las cuales nosotros no nos percatamos por el tiempo limitado para la entrega de la solución.
* Entre los imprevistos que impactaron nuestro desarrollo fue la de implementar justamente funciones que trabajen de una forma correcta con la estructura de datos en cuestión, donde tuvimos más problemas fue trabajar con los árboles, esto por la similitud que pueden llegar a tener con listas dobles, que es lo que se quería evitar desde un inicio, puesto que, a pesar de tener una estructura similar, sus funciones de inserción y búsqueda tienen tiempos diferentes. Esta problemática la realizamos con el uso de fuentes externas para guiarnos de como adecuar un árbol a nuestro proyecto. Otro imprevisto se podría considerar la de fallas con el IDE de visual studio, el cual a veces corrompía nuestro trabajo, para esto se decidió optar por la plataforma de Replit que funciona como compilador de C++. Imagen usando Replit para el código:



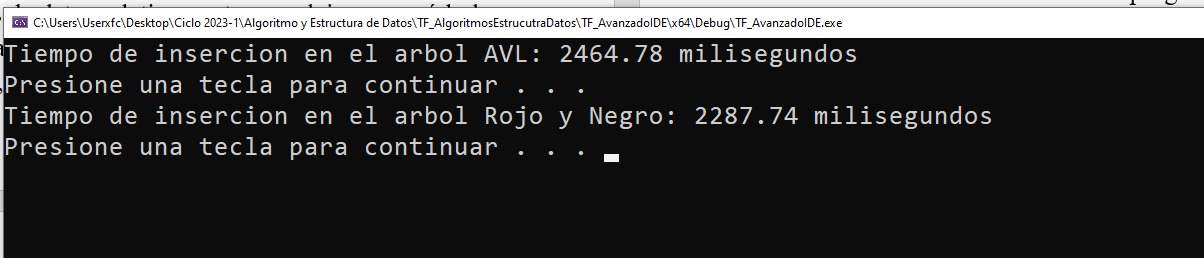
**Diseño del proceso de desarrollo de la solución**

* El caso de estudio se enfoca en resolver el problema de gestión de hoteles, para resolver esto se necesita el uso de programación orientada a objetos y estructuras de datos avanzadas, como árboles y hashTables. Este caso de estudio se abordó de la siguiente forma. Primeramente, se tiene a la parte “Gestión de Clientes", para esto es necesario el uso de una estructura de datos fundamental como un árbol AVL y RojoYnegro, luego se tiene a la "Reserva y disponibilidad de habitaciones" la cual de igual forma se usó un árbol AVL, finalmente se tiene la parte de "clientesReservados", estos son clientes los cuales a partir de unas ciertas condiciones se insertan a la HashTable para tener un registro ordenado y fácil de acceder para hallar información de estos.
* Ahora se adjunta el diagrama de clases respectivo, junto a un pdf para visualizar mejor:

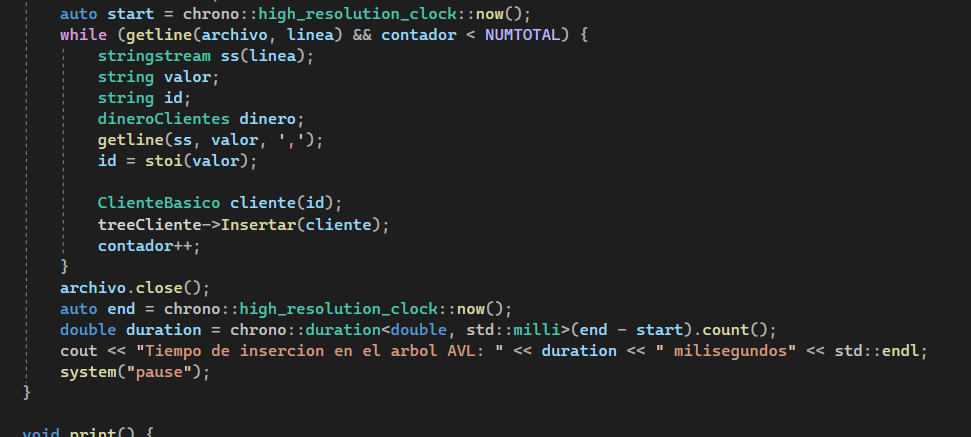




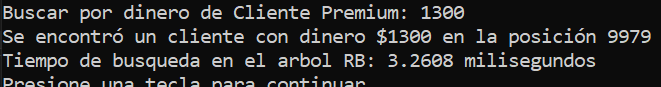
* **ARBOL AVL:** Este tipo de árbol balanceado se implementó para hacer la prueba de velocidad de inserción y búsqueda en comparación al árbol rojo y negro, este tipo de estructura se caracteriza por su eficaz en estos dos criterios, sin embargo, tras pruebas realizadas se vio que el árbol rojo y negro tuvo una mayor velocidad en el mayor de los casos para la inserción de 10^4 elementos. Adjunto captura.

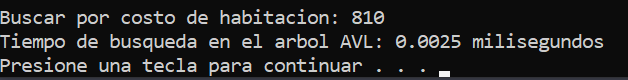


Esta comparación se hizo a partir de la ejecución de un código que mide el tiempo que demora en realizarse una función, adjunto captura de igual forma:



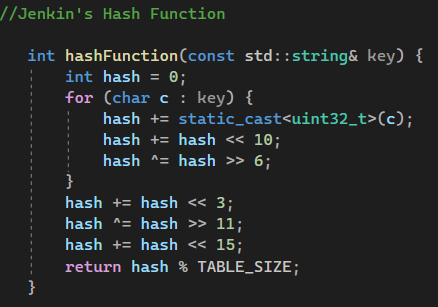
* **ARBOL Rojo y Negro:** Este tipo de árbol es de igual forma balanceado sin embargo bajo diferentes criterios, como su nombre lo dice separa sus nodos por “colores” balanceándose de esta forma, en las pruebas de inserción con 10^4 elementos mostró una mayor eficacia que el árbol AVL, mientras que en las búsquedas alcanzaron un tiempo muy similar o sino el árbol AVL era el que destacaba por milisegundos importantes.





Esta diferencia de tiempos en la búsqueda se puede deber lo más probable a que son árboles distintos con criterios distintos, sin embargo, es sorprendente que en el árbol AVL el tiempo para la búsqueda de un elemento sea de menos de un milisegundo, es decir casi al instante.

* **HashTable:** La estructura de datos de hashtable es sumamente eficaz para la búsqueda de elementos, puesto que procura en buscar el elemento en base de una key, la cual es brindada por el usuario y a partir de la función de hash, puede hacer que varíe el tiempo de búsqueda, respecto a la inserción es lo mismo, sin embargo para la inserción se procura normalmente almacenar la key del elemento junto a su value en una posición vacía; el hecho de que ya se carga previamente la cantidad de elementos que pueden haber en una tabla puede facilitar la inserción como búsqueda. Las funciones hash pueden afectar al rendimiento de igual forma, a continuación, adjunto captura de la Jenkin’s Hash Function la cual es comúnmente implementada por su simplicidad:



1. **Jenkin’s Hash Function (HashTable)**

int hashFunction(const std::string& key) { //1

int hash = 0; //1

for (char c : key) { //1+2+1+n(1+\_\_)+

hash += static\_cast<uint32\_t>(c); //2

hash += hash << 10; //2

hash ^= hash >> 6; //2

} //4+n(1+7)+\_\_

hash += hash << 3; //2

hash ^= hash >> 11; //2

hash += hash << 15; //2

return hash % TABLE\_SIZE; //2

}

Complejidad Algorítmica detallada en Big O: 12+8n - O(n)

1. **Bernstein’s Hash Function (HashTable)**

int hashFunction(const string& key) { //1

int hash = 5381; //1

for (char c : key) { //1+2+n(1+\_\_\_)

hash = ((hash << 5) + hash) + static\_cast<uint32\_t>(c); //4

}

return hash % TABLE\_SIZE; //2

}

Complejidad Algorítmica detallada en Big O: 5+4n - O(n)

1. **Ancestro común (ArbolRB)**

NodoRB<T>\* \_encontrarAncestroComun(NodoRB<T>\* nodo, T elemento1, T elemento2) { //1

if (nodo == nullptr) { //1

return nullptr; //1

}

if (criterio(elemento1, nodo->elemento) < 0 && criterio(elemento2, nodo->elemento) < 0) { //4

return \_encontrarAncestroComun(nodo->izq, elemento1, elemento2);

} //1

else if (criterio(elemento1, nodo->elemento) > 0 && criterio(elemento2, nodo->elemento) > 0) { //4

return \_encontrarAncestroComun(nodo->der, elemento1, elemento2); //1

}

else {

return nodo; //1

}

}

Complejidad Algorítmica detallada en Big O: 14 (En sí esta función depende mucho de la altura del árbol es decir puede ser O(n), donde n es la altura, o también puede ser O log(n).

1. **Balanceo (ArbolRB)**

void \_balanceo(NodoRB<T>\*& nodo) { //1

if (nodo == nullptr) return; //1

if (nodo->der != nullptr && nodo->der->esRojo && (nodo->izq == nullptr || !nodo->izq->esRojo)) { //5

\_rotarIzquierda(nodo); //1

}

else if (nodo->izq != nullptr && nodo->izq->esRojo && nodo->izq->izq != nullptr && nodo->izq->izq->esRojo) { //7

\_rotarDerecha(nodo); //1

}

else if (nodo->izq != nullptr && nodo->izq->esRojo && nodo->der != nullptr && nodo->der->esRojo) { //7

nodo->esRojo = true; //2

nodo->izq->esRojo = false; //3

nodo->der->esRojo = false; //3

}

}

Complejidad Algorítmica detallada en Big O: 31 (En sí esta función depende mucho de la altura del árbol es decir puede ser O(n), donde n es la altura, o también puede ser O log(n).

1. **\_inOrden(ArbolAVL)**

void \_inOrden(NodoArbol<T>\* nodo) { //1

if (nodo == nullptr) return; //1

\_inOrden(nodo->izq); //1

procesar(nodo->elemento); //1

\_inOrden(nodo->der); //1

}

Complejidad Algorítmica detallada en Big O: 5, prácticamente realiza un O(n), esto porque recorre cada elemento del árbol inOrden.

1. **\_actualizarElementoEnPosicion(ArbolAVL)**

if (nodo == nullptr) return nullptr; //1

nodo->izq = \_actualizarElementoEnPosicion(nodo->izq, contador, posicionObjetivo, elemento); //2

contador++; //1

if (contador == posicionObjetivo) { //1

nodo->elemento = elemento; //2

}

nodo->der = \_actualizarElementoEnPosicion(nodo->der, contador, posicionObjetivo, elemento); //2

actualizaAltura(nodo); //1

\_balanceo(nodo); //1

return nodo; //1

Complejidad Algorítmica detallada en Big O: 10, O(n) peor de los casos, O log(n); su tiempo de ejecución depende de la estructura del árbol y que tan desordenado se encuentre.

* La solución se implementó a partir de analizar el diseño del prototipo que se realizó al inicio de saber la problemática, posteriormente se aplicaría un diseño más detallado a este para elegir las herramientas adecuadas, como estructuras de datos que se emplearían al momento de empezar a codificar el código. Una vez realizada la solución se hicieron las pruebas que se consideraron convenientes para una correcta ejecución del programa, al menos en los puntos iniciales. Finalmente se estaría una documentación de este tipo redactando todos los pasos que se siguió para llegar a la solución presentada de la problemática de “Gestión de Hotels”
* Con la solución propuesta garantizamos que para nuestro caso se podrá gestionar de una mejor forma los clientes de un hotel, además que garantizamos la seguridad del programa, puesto que se siguió los protocolos para el manejo de objetos, es decir, no permitir que los atributos se encuentren en públicos, así evitando el manejo libre de estos. De igual forma con las estructuras de datos, se usaron con templates genéricos, algunos punteros a función y lambdas, esto para hacer un código lo más limpio posible y reutililizable.

**Conclusiones**

Las conclusiones finales a las que se llega tras la finalización de este proyecto es que el uso de estructuras de datos más avanzadas como árboles, permiten un uso más efectivo de la información que se guarda en estas. Además, que con el conocimiento del ciclo previo sobre programación orientada a objetos hizo este trabajo más fácil de comprender, puesto que en algunos casos se presentaban fallos con las clases, pero con este conocimiento ya identificábamos el error de una manera pronto y se resolvía lo antes posible. El uso de génericos, como templates ayudaron a reutilizar partes del código las cuales en caso distintos tendría que haber creado toda una función nueva justo para esa necesidad. A lo largo del curso se presentaron herramientas y diversas fuentes de información, las cuales se utilizaron para realizar un correcto trabajo. Páginas como GeeksForGeeks, stackOverflow y el “supersite“del profesor Walter Cueva, fueron útiles para la correcta implementación de nuestra solución en código.

**Referencias bibliográficas**

* Arbol Rojo y Negro: <https://www.geeksforgeeks.org/introduction-to-red-black-tree/>
* Inserción para AVL Tree: <https://www.geeksforgeeks.org/introduction-to-red-black-tree/>
* Supersite de WalterCueva: <https://waltercueva.super.site/algorithms-data-structures>
* RedBlackTree vs AVL Tree: <https://www.geeksforgeeks.org/red-black-tree-vs-avl-tree/>
* RedBlackTree vs AVL Tree: [**https://www.javatpoint.com/red-black-tree-vs-avl-tree**](https://www.javatpoint.com/red-black-tree-vs-avl-tree)
* Jenkin’s Hash Function: <https://www.opentechguides.com/how-to/article/vb/173/jenkins-hash.html>
* Bernstein’s Hash Function: <https://theartincode.stanis.me/008-djb2/>