Documentación del proyecto BigInteger

Análisis de operaciones y algoritmos

Marco Antonio Riascos Salguero

Código: 8952273

Pontificia Universidad Javeriana Cali

Estructura de Datos

Carlos Ramírez, Gonzalo Noreña

25 de mayo de 2023

Santiago de Cali

Introducción

En el presente documento se hará una descripción detalla de como se realizo el proyecto BigInteger, con explicaciones de las diferentes funciones que se aplicaron para la ejecución de dicho programa, se tendrá en cuenta los tipos de datos con los que se trabajó, explicaciones de las diferentes estructuras de datos implementadas, así como la complejidad de dichos algoritmos usados. En dicho documento se expone lo que es un BigInteger y porque la necesidad de trabajar con ello.

¿Qué es un BigInteger?

BigInteger es una clase utilizada en programación para representar números enteros arbitrariamente grandes. A diferencia de los tipos de datos estándar en muchos lenguajes de programación; que tienen un tamaño fijo (como int en C), BigInteger permite almacenar y acceder a números enteros que exceden el rango de los tipos de datos de números enteros estándar.

BigInteger se implementa utilizando estructuras de datos internas que almacenan secuencialmente números únicos enteros. Permite representar números enteros de cualquier tamaño, desde pequeños hasta muy grandes, sin los límites de tamaño de los tipos de datos estándar.

La clase BigInteger proporciona operaciones aritméticas básicas como suma, resta, multiplicación y división que se pueden aplicar a números enteros arbitrariamente grandes.

Almacenamiento de Datos

La forma en la que se reciben los datos es a través de una lectura de datos estándar que recibe una cadena de caracteres de tipo string, donde se hace uso de un constructor que convierte carácter a carácter, a un numero entero, que luego se almacena un atributo de la clase.

constructores
BigInteger (string cad);
BigInteger ();
BigInteger (BigInteger& copia);

La forma que se utilizó para almacenar un BigInteger es utilizando un vector (o arreglo dinámico) de enteros, donde cada elemento del vector representa un dígito del entero. Los dígitos se almacenan en orden, de manera que el dígito de menor peso (menos significativo) se encuentra en la posición 0 del vector, y los dígitos siguientes se almacenan en posiciones sucesivas.

Por ejemplo, si queremos representar el número 1234567890 como un BigInteger, podríamos almacenarlo en un vector de esta manera:

vector<int> vec_big = {0, 9, 8, 7, 6, 5, 4, 3, 2, 1};

La clase BigInteger también cuenta con una variable buleana (bool sign) que permite identificar si el BigInteger es negativo o positivo, para realizar las operaciones designadas.

La manera como se almacena cada dato en el vector es buscando obtener la menor complejidad posible, ya que al momento de agregar un nuevo numero al vector se utiliza push_back, lo que se convierte en una complejidad de lineal amortizado por el hecho de usar la función de c.

Otros tipos de constructores que contine la clase son un constructor que crea un vector de tamaño con 0; lo que lo convierte de complejidad constante, ya que siempre es igual, y un constructor que recibe otro bigInteger y lo copia, como es el caso de la función abs.

BigInteger abs ();

que recibe un vector y lo copia y lo convierte en una versión idéntica con signo positivo. Si consideramos la copia del vector y las operaciones adicionales dentro de la función, se espera que la complejidad sea lineal en relación con el tamaño del vector de dígitos (vec_big), lo cual se denota como O(N), donde N es el tamaño del vector. Finalmente, si el BigInteger es negativo, se modifica el bit de signo en la copia. Esta operación es constante, ya que solo implica modificar un valor booleano.

Funciones utilizadas en la clase BigInteger

Como parte de la clase BigInteger tenemos diferentes operaciones y funciones que permiten modificar y crear nuevos objetos de la clase; como vendrían siendo algunas sobrecargas de operadores tales como:

SOBRECARGA DE OPERADORES	DEFINICION Y COMLEJIDAD
BigInteger operator+ (BigInteger& new_BigInteger)	Suma dos objetos de la clase con una complejidad lineal O(n)
BigInteger operator- (BigInteger& new_BigInteger)	Resta dos objetos de la clase con una complejidad lineal O(n)
BigInteger operator*(BigInteger& new_BigInteger)	Multiplica dos objetos de la clase y su complejidad tiene que ver con la suma de la complejidad del operador +
bool operator< (BigInteger& new_BigInteger)	Verifica si el primer objeto es menor que el segundo, en el mejor de los casos es O (1)
bool operator== (BigInteger& new_BigInteger)	Verifica si el primer objeto es igual que el segundo, en el mejor de los casos es O (1), en el pero que sean iguales es O(n)
bool operator<= (BigInteger& new_BigInteger)	Verifica si el primero es menor o igual que el segundo, se le suman las complejidades de los operadores pasados
BigInteger operator/ (BigInteger& new_BigInteger)	Divide dos objetos de la clase, y su complejidad es mucho mas baja que lineal; podría ser logarítmica, pero se le suma la complejidad del operador -
BigInteger operator% (BigInteger& new_BigInteger)	Permite encontrar el módulo de la división, en este caso el residuo, por lo que su complejidad debe ser sumada al del operador /

Los cuales tienen como propósito crear nuevos objetos a partir de recibir parámetros de otros. Cada sobrecarga tiene su función y complejidad. Un ejemplo es la sobrecarga de operador = =, en el cual

verifica si dos objetos de la clase son iguales, y su complejidad pude llegar a constante en caso de que no lo sean, solo por el simple hecho de tener signos distintos, en un atributo de la clase; y en el peor de los casos, cuando son iguales, su complejidad es lineal O(n).

Cabe resaltar que la sobrecarga de estos operadores nos permite simplificar operaciones en otras funciones, ya que el programa al recibir dicho carácter sabe como comparar dos objetos de la clase.

En caso de trabajar con otras funciones, en las cuales no se retorna nada, sino que se modifica un objeto de la clase; como lo son:

Función	Especificación y complejidad
bool isNegative();	verifica si un objeto es negativo en tiempo constante a través de un atributo de clase, O (1)
BigInteger abs () const;	Copia un objeto de la clase y modifica la copia para que tenga signo positivo, teniendo una complejidad O(n); se le agrega (const) para que no modifique el objeto que se pasa por referencia
void add (BigInteger new_BigInteger);	suma dos objetos usando la sobrecarga del operador +, su complejidad es O(n)
void substract (BigInteger new_BigInteger);	resta dos objetos de la clase usando la sobrecarga del operador -
void product (BigInteger new_BigInteger);	Multiplica dos objetos de la clase con la sobrecarga del operador *, su complejidad varia ya que dicho operador invoca la función add
string toString ();	Convierte el objeto de valores enteros a caracteres singulares que luego se agregan a un string, su complejidad es lineal, ya que debe recorrer todo el objeto
void quotient (BigInteger new_BigInteger);	Permite modificar el objeto al dividirlo por su otro, con la sobrecarga del operador /
void remainder(BigInteger new_BigInteger);	Permite obtener el residuo de una división, a lo que llamamos modulo, y su complejidad varía dependiendo de la ejecución del operador /
void pow(int e);	Función que permite elevar objetos a una potencia determinada, gracias al algoritmo con el que se codifico, su complejidad es O (log n), convirtiéndola en una de las funciones más eficaces del programa y eficientes

Dichas operaciones también tienen su complejidad y función, algunas de ellas solo usan la sobrecarga del operador antes mencionados, como en el caso de la operación product. Esta función usa el * sobrecargado para modificar el objeto de la clase con el cual fue invocado a través de la notación num. product(num2), siendo num1 el BigInteger que se va a modificar y num2, el que se recibe como parámetro para cumplir con dicha función

Conclusión

Para poder trabajar con tipos de datos enteros, que son demasiado grandes, se recomienda usar una clase, ya sea vectores, listas o cualquier tipo de datos abstracto que permita organizar los datos de manera ordenada a fin tener la información completa. Así como se respeta el principio de abstracción al tener funciones especificas que accedan a los datos privados de la clase.