Estructuras de datos 2023-1

Proyecto final

Informe de complejidades

Estudiante: Juan Diego Valencia Alomia

Docentes: Carlos Ramírez & Gonzalo Noreña

Ingeniería de Sistemas y Computación

Santiago de Cali mayo 29 2023

TABLA DE CONTENIDO

INTRODUCCIÓN

En los lenguajes de programación los tipos de datos nativos, como int o long, tienen un rango limitado de valores que pueden representar. Por ejemplo, en C++, un int típicamente tiene un rango de -2,147,483,648 a 2,147,483,647. Esto representa un limitante a la hora de trabajar con números que sobrepasan este espacio, por lo tanto, surge la necesidad de implementar un TAD (Tipo Abstracto de Datos) como el BigInteger con la necesidad de manipular números enteros de gran magnitud que exceden la capacidad de los tipos de datos nativos

Un BigInteger permite almacenar y manipular números enteros de cualquier tamaño, sin restricciones de rango. Puedes realizar operaciones aritméticas como suma, resta, multiplicación y división con números de gran magnitud de manera precisa y sin perder información. Esta implementación permite a los programadores trabajar con números enteros, por ejemplo, esto es necesario en criptografía, las matemáticas o algoritmos numéricos.

En este informe se pretende exponer y analizar las complejidades de las operaciones de la implementación, explicando una por una la lógica utilizada y la eficiencia obtenida.

IMPLEMENTACIÓN

Para la implementación de este TAD se escogió la estructura de datos vector, este contenedor permite almacenar un conjunto de elementos de tamaño variable en una secuencia ordenada. Es una estructura de datos dinámica que proporciona muchas funcionalidades útiles y simplifica la manipulación de elementos, por ejemplo, se puede acceder a los elementos almacenados a partir de el índice. En el caso del BigInteger resulta útil ya que en todas las operaciones hay que hacer recorridos del número, y acceder por medio del índice resulta eficiente ya que al tratarse de una secuencia ordenada este proceso es constate. Además, se usó un entero que almacena el signo del número, en la implementación el valor de este entero varía entre 1 y -1, porque resulta útil para calcular cual es el signo del resultado de una operación o al momento de comparar.

COMPLEJIDADES

Constructores

1. **BigInteger();**

Este es el constructor por defecto, tiene la función de inicializar un BigInteger sin parámetros, solo le asigna el vector y el entero que guarda el signo para una posterior asignación. Por lo Tanto su complejidad es O(1), ya que no hace ningún tipo de operación adicional

2. **BigInteger(const string &n);**

Este constructor recibe una cadena de texto que contiene los dígitos del numero que se quiere operar, lo que realiza esta operación es el recorrido de la cadena para almacenarla en un vector de forma invertida ya que esto facilita el posterior manejo de datos, además revisa si el signo es positivo o negativo. Por lo tanto al tratarse de un recorrido la complejidad de esta función es O(n), siendo n el tamaño de la cadena, cabe resaltar, que al hacer push_back al vector este tiene complejidad constante amortizada.

3. **BigInteger(const vector<int> &n, int s);**

Este constructor tiene como función la copia de un BigInteger conociendo su vector interno y el signo. Se creó con el objetivo de crear un objeto a partir de un vector, esto en el caso de la suma y la resta resulta de gran utilidad. Al tratarse de una copia, tan solo copia los parámetros entregados en el nuevo objeto. Por esta razón tiene complejidad constante O(1)

4. **BigInteger(const BigInteger &est);**

Este es el segundo constructor de copia, que toma como parámetro otro BigInteger, este constructor copia los atributos privados del numero que se pasa por parámetro al objeto nuevo. Por esta razón tiene una complejidad O(1) ya que no realiza ninguna operación adicional.

Operaciones analizadoras

Las siguientes funciones se crearon con el objetivo de mantener el principio de los TAD de no revelar como está implementado internamente la interfaz, por lo que tienen como función mostrar alguna característica de los objetos BigInteger para facilitar algunas operaciones que son necesarias para el posterior manejo operacional

1. **string toString();**

Esta función tiene como objetivo convertir el contenido de un BigInteger en una cadena de texto, para esto se recorre el vector interno con el fin de concatenar los dígitos en forma de cadena de texto, además tiene en cuenta el signo para asignarlo al comienzo de la cadena en el caso de que se trate de un numero negativo. Por esta razón al recorrer el vector interno, esta operación tiene complejidad O(n), siendo n el tamaño del vector.

2. **int sizeInt();**

Esta operación retorna el tamaño del vector interno, en otras palabras, retorna la cantidad de dígitos que tiene el número, esto resulta útil a la hora de querer recorrer un numero. De esta manera, la complejidad de esta operación es O(1) ya que obtener el tamaño de un vector por medio de size() es constante.

3. **int infoNum(int v);**

Esta función tiene el objetivo de retornar el digito del vector interno con el índice v, esta operación resulta útil para el recorrido de los números que se crean dentro de las operaciones. Al tratarse del índice de un vector entonces esta operación tiene complejidad O(1).

4. **int getSign();**

Esta función tiene el objetivo de obtener el signo del objeto, esto resulta útil para el calculo de los signos de las operaciones. Al tratarse de retornar un entero ya predefinido, entonces la complejidad es O(1), ya que no se realizan operaciones adicionales

Sobrecargas de los operadores de comparación

1. **bool operator==(BigInteger &est);**

Esta operación compara dos BigInteger, lo hace por medio de la comparación de sus vectores internos y por el signo, si estos dos factores resultan ser iguales retorna true, de lo contrario false. La complejidad de hacer la operación == entre dos vectores es O(n) siendo n el tamaño de los vectores, ya que revisa que todos los elementos sean iguales. Por lo tanto, la complejidad de esta sobrecarga también será O(n), que hace referencia al tamaño de los vectores.

2. **bool operator<(BigInteger &est);**

Esta operación tiene la función de comparar dos BigInteger mediante el operador <, para esto tiene 4 condicionales: El primer caso es si los dos son iguales, si esto resulta ser verdadero retorna falso, como mencioné anteriormente la operación == entre dos BigIntger es de O(n). En el segundo caso se compara los signos, si el actual es negativo y el otro positivo entonces retorna true, de lo contrario false, por lo tanto este caso tiene complejidad O(1). El tercer caso es que tienen igual tamaño e igual signo, por lo cual se recorre los vectores hasta que alguno sea distinto, en el peor caso son iguales hasta el ultimo digito, por lo que tendría complejidad O(n). Finalmente, el ultimo caso es que son de diferente tamaño e igual signo, por lo cual el retorno se decide por cual tiene mayor o menor tamaño, por lo tanto este cuarto caso tiene complejidad O(1). Teniendo en cuenta todo lo mencionado, se puede concluir que la complejidad de esta función es O(n), ya que en el peor de los casos recorre los vectores hasta el final para conocer la respuesta.

3. **bool operator<=(BigInteger &est)**

Esta función tiene la función de comparar dos BigInteger mediante el operador <=, utiliza la lógica del == y del <, por lo tanto se puede concluir que también tendrá la misma complejidad O(n), n siendo el tamaño de los vectores. La única diferencia con el < es que cuando se compara los dos objetos con el == retorna true si se cumple.

Operaciones auxiliares

Para el desarrollo más eficiente del programa se creó un método privado con el fin de poder utilizarlo en el desarrollo de la interfaz

1. **BigInteger abso();**

Esta función retorna el valor absoluto del objeto actual, esto lo hace por medio del constructor de copia del vector y el signo. Por lo tanto al tratarse de una copia, la complejidad de esta función es O(1), ya que simplemente retorna el mismo numero pero con su signo igual a 1

Sobrecarga de los operadores

En esta sección se desarrollo la mayoría de la lógica operacional entre dos objetos BigInteger, estas operaciones retornan un nuevo objeto con el resultado de la operación.

1. **BigInteger operator+(BigInteger &est);**

Esta función utiliza el operador + para sumar dos enteros independientemente de su signo o tamaño. El proceso empieza por determinar cuál de los dos números que se quiere sumar es el mayor y cual es el menor, esto por medio del <, por lo tanto al usar el < ya tendríamos una complejidad de O(n).

Posteriormente, hay tres posibles casos, el primero de ellos es trata de que los dos números tienen igual signo, en este caso se sumarán y el signo de resultado será cualquiera de los dos números, el segundo caso es que si el mayor es positivo y el menor negativo, en este escenario hay que hacer otra comparación, la cual es la comparación de los valores absolutos, si en esta comparación resulta que el menor es el mayor, entonces se intercambian los signos, y se le resta el mayor al menor, y al resultado se le asigna el signo del menor. Por último, si no se cumplieron las dos condiciones antes mencionadas entonces se resta el menor al mayor. Todas estas operaciones se realizan por el recorrido de los vectores de los números, por lo tanto, esta parte tiene complejidad O(n) siendo n el vector de mayor tamaño.

De esta manera se puede concluir que la complejidad de sumar es O(n) siendo n el numero de dígitos del BigInteger de mayor tamaño.

2. **BigInteger operator-(BigInteger &est);**

Esta función utiliza el operador – para restar dos enteros independientemente de su signo o tamaño. Esta función en realidad por debajo es la misma suma, que sigue el mismo algoritmo, lo único que cambia es el hecho de que al BigInteger que se recibe como parámetro de multiplica por -1 su signo, con ese numero se realiza la suma. Por lo tanto al tratarse del mismo algoritmo entonces se puede concluir que esta operación tendrá complejidad O(n), siendo n el tamaño del Biginteger con más dígitos.

3. **BigInteger operator*(BigInteger &est);**

Esta función utiliza el operador * para realizar la multiplicación, el algoritmo de esta función de basa por el recorrido individual de cada digito del objeto actual por el numero que se recibe como parámetro, por lo cual se realiza en un ciclo anidado, es decir que si llámanos n al tamaño del numero del objeto actual y m al tamaño del objeto que se pasa por parámetro, entonces hace n * m recorridos, por lo cual si acotamos al caso de que los números tengan igual tamaño entonces la complejidad de esta operación sería O(n^2)

4. **BigInteger operator/(BigInteger &est);**

Esta función utiliza el operador / para realizar la división, el algoritmo empieza preguntando si el actual es menor que el que se recibe por parámetro, como es una división entera entonces si le cumple el resultado será 0, como hace uso del < entonces ese if tendría una complejidad de O(n). Posteriormente si resulta siendo falso, entonces comienza extrae un numero a partir del dividendo que sea de igual tamaño y mayor que el divisor, una vez hecho eso empieza la búsqueda del múltiplo del divisor por el cual se puede restar, es decir que hace uso del <= y de la suma, en el peor caso el divisor tiene solo 1 digito, por lo que haría esa comparación m veces, siendo m el tamaño del dividendo, por lo tanto al considerar que la suma tiene complejidad O(n) y este proceso se realiza por la longitud del dividendo entonces la complejidad total del algoritmo es O(n * m).

5. **BigInteger operator%(BigInteger &est);**

Esta función utiliza el operador % para realizar el modulo de la división, es decir, el residuo, esta función realiza el mismo proceso de la división pero retorna la ultima resta que se hizo, por lo cual al hacer el mismo proceso, se puede concluir que la complejidad del modulo es de O(n*m)

Operaciones Modificadoras del actual

Las siguientes operaciones modifican el objeto actual asignándole el resultado de la operación. Además, todas funcionan con los operadores antes mencionados, por lo que comparten la misma complejidad, por lo que solo se ejemplificará la pontencia

1. **void pow(int est);**

Esta operación eleva el numero actual al exponente est, el algoritmo de esta función de basa en hacer multiplicaciones hasta que est sea igual a 0, de este modo la complejidad de esta operación será O(est * n^2) siendo n el tamaño del número actual

Operaciones de manejo de listas

Las siguiente dos operaciones son estáticas, es decir que no se haces sobre un objeto sino sobre la clase, por lo que en este caso no hay objeto actual.

1. **static BigInteger sumarListaValores(list<BigInteger> &n);**

Esta operación suma todos los valores de la lista mediante un contador, si tenemos que la suma tiene una complejidad de O(n) entonces esta función tiene una complejidad de O(m * n) siendo m el tamaño de la lista y n el tamaño del vector del acumulador.

2. **static BigInteger multiplicarListaValores(list<BigInteger> &n);**

Esta operación multiplica todos los valores de la lista mediante un contador, si se tiene que la complejidad de la multiplicación es O(n^2) entonces esta función tiene una complejidad de O(m * n^2) siendo m el tamaño de la lista.