**David C. Cortés, Kevin J. Carranza, Ricardo A. Marino**

**No. de Equipo Trabajo: {4}**

Headfuck, IDE para Brainfuck++

# [[1]](#footnote-1) INTRODUCCIÓN

Este documento consiste en una descripción detallada del desarrollo de un IDE (Entorno de desarrollo integrado) para el lenguaje de programación Brainfuck++.

# DESCRIPCIÓN DEL PROBLEMA A RESOLVER

Brainfuck es un lenguaje de programación esotérico creado en 1993 por Urban Müller, el cual contiene solo 8 comandos, de los cuales 2 son para la salida y entrada de datos. Al igual que muchos otros lenguajes de programación, este lenguaje tiene diferentes extensiones, como lo es el caso de Brainfuck++, una extensión hecha por la página web *CodeAbbey*.

Hoy día, la mayoría de los lenguajes que se usan, como Python, C++ o JAVA, poseen, entre muchas cosas, una sintaxis más legible y estructurada para hacer más cosas con menos líneas de código. Sin embargo, al Brainfuck++ ser un lenguaje esotérico, desafía y reta al programador a ser más ingenioso y eficiente, desarrollando un mejor pensamiento lógico-matemático, que en un futuro facilita el aprendizaje de lenguajes con más *“comodidades”*.

Es por esta razón que como grupo hemos decidido desarrollar un IDE para Brainfuck++, que básicamente permita escribir y compilar código.

# usuarios DEL PRODUCTO DE SOFTWARE

Este software está diseñado para dos tipos de usuarios.

Primero: los estudiantes que están cursando cursos como *“Programación de computadores”* y no tienen conocimientos previos de programación, debido a que el funcionamiento de este lenguaje de programación es muy similar al de una máquina de Turing.

Segundo: Los usuarios con conocimientos previos de programación que quieran retarse a sí mismos con este lenguaje y de paso mejorar sus habilidades de razonamiento lógico-matemático.

# REQUERIMIENTOS FUNCIONALES DEL SOFTWARE

## Abrir archivo

## Descripción: Esta funcionalidad permite abrir un archivo de extensión .bfck.

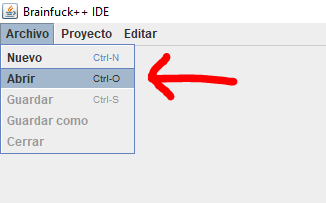
* *Acciones iniciadoras y comportamiento esperado*:

1. Abrir el IDE de Brainfuck++.
2. A continuación, se abrirá la interfaz gráfica del IDE de Brainfuck++. La siguiente acción que debe hacer el usuario es hacer clic en el botón llamado *Archivo* de la esquina superior izquierda.

Captura de pantalla de un celular

Descripción generada automáticamente

1. Con esto se desplegará un panel en el cual el usuario debe seleccionar la opción *Abrir.*



1. Esto hará que se despliegue una ventana en la cual el usuario podrá seleccionar el archivo de extensión .bfck que desea abrir.

Captura de pantalla de un celular

Descripción generada automáticamente

1. Después, el usuario debe dar clic en el botón *Abrir.*
2. Finalmente, el IDE de Brainfuck abrirá el archivo y lo desplegará en el panel de programación.

Captura de pantalla de un celular

Descripción generada automáticamente

## Guardar archivo

## Descripción: Esta funcionalidad permite que el usuario pueda guardar un archivo o actualizar un archivo ya guardado .bfck.

* *Acciones iniciadoras y comportamiento esperado*:

1. Crear o Abrir un archivo .bfck en el IDE de Brainfuck++. Para crear un archivo nuevo, solo es necesario ir a la pestaña superior izquierda que dice *Archivo* y seleccionar la opción *Nuevo*.
2. Ir a la pestaña superior izquierda llamada *Archivo*, cuando se dé clic sobre esta pestaña se desplegará un menú y el usuario deberá seleccionar la opción guardar.

Captura de pantalla de un celular

Descripción generada automáticamente

1. En caso de que el usuario no haya guardado ese archivo antes, se desplegará una ventana similar a la del paso 4 de *Abrir un archivo* en la cual el usuario podrá seleccionar en dónde quiere guardar su archivo. De lo contrario, los cambios hechos al archivo se actualizarán dentro del archivo sobre el cual está trabajando.

## Abrir una ventana nueva de programación

## Descripción Esta funcionalidad permite al usuario abrir diferentes ventanas de programación para que este pueda trabajar más archivos a la vez. La implementación de esta funcionalidad se hizo con una árbol AVL de nodos llamados Archivo, ya que no se conocen de antemano la cantidad de ventanas que el usuario abrirá y se desea que la inserción sea rápida .

* *Acciones iniciadoras y comportamiento esperado*:

1. Para crear una nueva ventana de programación existen dos opciones. Primero, que el usuario cree un nuevo archivo o abra otro archivo, con lo cual se desplegaran sobre el panel de programación dos pestañas con los respectivos nombres de los archivos que se abrieron.

Imagen que contiene reloj

Descripción generada automáticamente

Y la otra manera para abrir una nueva ventana de programación es cuando el usuario da clic sobre el símbolo de *+* al lado de las pestañas de programación.

Captura de pantalla de un celular

Descripción generada automáticamente

## Seleccionar una ventana de programación

## Descripción: Esta funcionalidad permite seleccionar una ventana de programación específica y cargar en el panel de programación el respectivo código. Básicamente, lo que hace a nivel de código es buscar un nodo en el árbol AVL que contiene todas las ventanas.

* *Acciones iniciadoras y comportamiento esperado*:

1. Para acceder a una ventana específica el usuario deberá dar clic sobre la pestaña de la ventana que desea abrir. Seguidamente, se desplegará en el panel de programación el código que estaba en dicha pestaña y esta pestaña se pondrá de un color gris oscuro.

## Eliminar una ventana de programación

## Descripción: Esta funcionalidad permite que el usuario pueda cerrar una ventana de programación, lo cual a nivel de código es eliminar un nodo específico en un árbol AVL.

* *Acciones iniciadoras y comportamiento esperado*:

1. Seleccionar la ventana que desea cerrar.
2. Dar clic en el botón *Archivo* que está en la esquina superior izquierda. Esto hará que se despliegue un menú, y el usuario deberá dar clic en *Cerrar,* lo cual eliminará la ventana seleccionada y se seleccionará la ventana que este atrás de esta.

Captura de pantalla de un celular

Descripción generada automáticamente

## Deshacer en el panel de programación.

## Descripción: Esta funcionalidad permite que el usuario puede deshacer la última acción que hizo. Esta funcionalidad se implementó con una pila y con patrones de comandos, para recuperar la anterior acción.

* *Acciones iniciadoras y comportamiento esperado*:

1. Para deshacer la última acción hecha el usuario debe seleccionar la ventana en la cuál quiere deshacer la última acción y luego oprimir las teclas *Ctrl-z* y el IDE actualizará el panel de programación deshaciendo la última acción hecha. También en lugar de dar *Ctrl-z* el usuario puede ir a la opción de *Editar* en el menú superior, y seleccionar la opción *Deshacer.*

Captura de pantalla de un celular

Descripción generada automáticamente

## Rehacer en el panel de programación

## Descripción: Esta funcionalidad permite al usuario rehacer la última acción deshecha. Para hacer esto cada acción que se deshace se apila en una pila, de modo que cuando el usuario desee rehacer la última acción, la pila retorna y elimina el elemento del tope.

* *Acciones iniciadoras y comportamiento esperado*:

1. Para rehacer la última acción, el usuario debe seleccionar la ventana de programación en la cual quiere rehacer la última acción deshecha. Una vez hecho esto, el usuario o bien puede oprimir *Ctrl-y* para rehacer la última acción hecha, con la cual el IDE actualizará el panel de programación rehaciendo los cambios hechos, o también puede ir al menú superior en la opción *Editar* y seleccionar la opción *Rehacer.*

Captura de pantalla de un celular

Descripción generada automáticamente

## Ingresar las entradas para el código de Brainfuck++

## Descripción: Esta funcionalidad ingresa los datos del panel de entradas y los ingresa en una cola, para que luego el compilador vaya desencolando cada vez que el código lo pida. Se utilizó una cola para esta funcionalidad ya que sólo se necesita acceder a la primera entrada ingresada y para ingresar una entrada esta se debe entrar detrás de las anteriores.

* *Acciones iniciadoras y comportamiento esperado*:

1. El usuario deberá seleccionar la ventana de programación la cual desea compilar. Luego ingresar en el panel de entradas, las entradas que su código necesita. Luego deberá ir al menú de *Proyecto* y dar la opción *Correr*. Una vez esto hecho el compilador leerá el código y las entradas las encolará, para luego desplegar en el panel de *Salida* las salidas del código.

Captura de pantalla de un celular

Descripción generada automáticamente

# DESCRIPCIÓN DE LA INTERFAZ DE USUARIO PRELIMINAR

La interfaz de usuario se enfatiza en el uso de tres paneles dentro de una ventana, el primer panel se encuentra dedicado a la programación en sintaxis de Brainfuck++, el segundo a las entradas y el tercero a la salida del programa.

La ventana también contiene una cinta de opciones, en la cual se podrán ejecutar distintas acciones como abrir, guardar, cerrar, deshacer, rehacer y correr. Está interfaz podrá cambiar de acuerdo a las nuevas implementaciones del software.

Primer Mockup:

Captura de pantalla de un celular

Descripción generada automáticamente

Segundo Mockup:

Captura de pantalla de un celular

Descripción generada automáticamente

Versión Actual:

Captura de pantalla de un celular

Descripción generada automáticamente

# Entornos de desarrollo y de operación

El software se está desarrollando en el SO windows 10 64 bits, en un procesador intel i5 6200U @ 2.3 GHz y una memoria RAM de 4 GB, SO windows10, 64 bits, en un procesador AMD Phenom(tm) II X6 1090T 3.20GHz y una memoria RAM de 8 GB.

Las aplicaciones creadas en java pueden ejecutarse en cualquier máquina virtual Java (JVM) sin importar la arquitectura de la computadora, por lo que nuestra aplicación podrá ejecutarse sin problemas en los sistemas operativos más comunes (Windows, MacOS, Linux).

# PROTOTIPO DE SOFTWARE INICIAL

El prototipo de este proyecto se puede encontrar en el siguiente repositorio en la carpeta *Dist*:

<https://github.com/Marinovsky/Brainfuck_IDE/tree/Entrega-2>

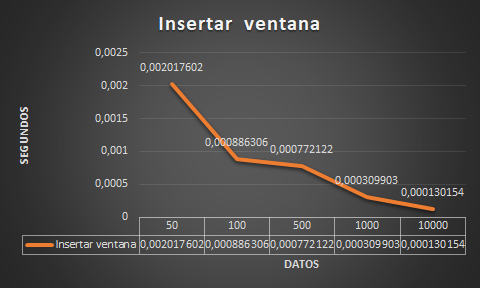
# PRUEBAS DEL PROTOTIPO

A continuación, se presentan las gráficas de las pruebas que se le hicieron a las diferentes funcionalidades del IDE.

***Ventanas de programación:*** Para las ventanas de programación se hicieron pruebas dependiendo de la cantidad de ventanas. A continuación, se muestran las tablas de comparación del tiempo de ejecución respecto a la cantidad de ventanas y sus respectivas gráficas.

*Agregar ventana al árbol AVL:*

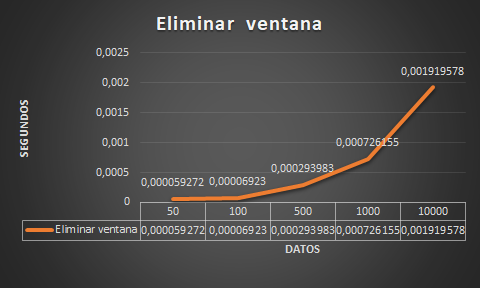
|  |  |  |  |
| --- | --- | --- | --- |
| Cantidad de ventanas | Tiempo en nanosegundos | Tiempo en milisegundos | Tiempo en segundos |
| 50 | 2017602ns | 2,017602ms | 0,00017602s |
| 100 | 886306ns | 0,886306ms | 0,000886306s |
| 500 | 772122ns | 0,772122ms | 0,000772122s |
| 1000 | 309903ns | 0,309903ms | 0,000309903s |
| 10000 | 130154ns | 0,130154ms | 0,000130154s |



*Complejidad: La complejidad es O(log(n)) ya que al ser un árbol AVL la cantidad de nodos recorridos será por mucho la altura del albor, y las rotaciones tienen complejidad O(1). La grafica se muestra distinta porque se están reutilizando espacios en memoria.*

*Remover ventana aleatoria en el árbol AVL*

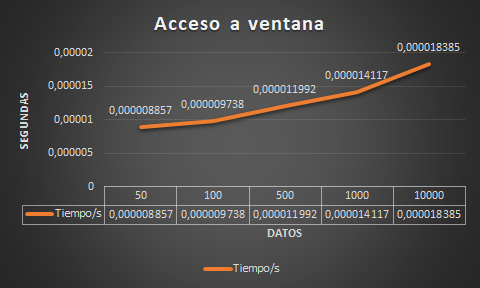
|  |  |  |  |
| --- | --- | --- | --- |
| Cantidad de ventanas | Tiempo en nanosegundos | Tiempo en milisegundos | Tiempo en segundos |
| 50 | 59272ns | 0,059272  ms | 0,000059272s |
| 100 | 69230ns | 0,06923ms | 0,00006923s |
| 500 | 293983ns | 0,29398ms | 0,000293983s |
| 1000 | 726155ns | 0,72615ms | 0,000726155s |
| 10000 | 1919578ns | 1,919578ms | 0,001919578s |



*Complejidad: La complejidad es O(log(n)) ya que al estar implementado con un árbol AVL, en el peor de los casos para eliminar un nodo se va a recorrer una cantidad de nodos igual a la altura del árbol (log(n)). La grafica se muestra de esa forma ya que se están reutilizando espacios en memoria.*

*Acceder a una ventana específica en el árbol AVL*

|  |  |  |  |
| --- | --- | --- | --- |
| Cantidad de ventanas | Tiempo en segundos | Tiempo en milisegundos | Tiempo en segundos |
| 50 | 8857ns | 0,00885ms | 0,00000885s |
| 100 | 9738ns | 0,00973ms | 0,00000973s |
| 500 | 11992ns | 0,01199ms | 0,00001199s |
| 1000 | 14117ns | 0,01411ms | 0,00001411s |
| 10000 | 18385ns | 0,01838ms | 0,00001838s |

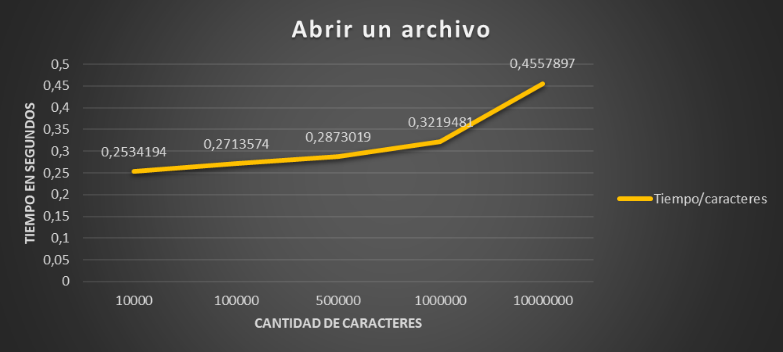


*Complejidad: La complejidad es O(log(n)) ya que al estar implementado con un árbol AVL, en el peor caso para acceder a un nodo, se va a recorrer una cantidad de nodos igual a la altura del árbol (log(n)). La grafica se muestra de esa forma ya que se están reutilizando espacios en memoria.*

***Abrir y guardar archivos***: Para abrir y guardar archivos se hicieron pruebas dependiendo de la cantidad de caracteres en el archivo. A continuación, se muestran las tablas de comparación del tiempo de ejecución respecto a la cantidad de caracteres y sus respectivas gráficas.

*Abrir un archivo*

|  |  |  |  |
| --- | --- | --- | --- |
| Cantidad de caracteres | Tiempo en nano segundos | Tiempo en milisegundos | Tiempo en segundos |
| 10000 | 253419400ns | 253,4194ms | 0,2534194s |
| 100000 | 271357400ns | 271,3574ms | 0,2713574s |
| 500000 | 287301900ns | 287,3019ms | 0,2873019s |
| 1000000 | 321948100ns | 321,9481ms | 0,3219481s |
| 10000000 | 455789700ns | 455,7897ms | 0,4557897s |



*Complejidad: La complejidad es O(n) ya que se tiene que escribir cada carácter uno por uno.*

*Guardar archivo*

|  |  |  |  |
| --- | --- | --- | --- |
| Cantidad de caracteres | Tiempo en nanosegundos | Tiempo en milisegundos | Tiempo en segundos |
| 10000 | 212341800 | 212,3418 | 0,2123418 |
| 100000 | 206196300 | 206,1963 | 0,2061963 |
| 500000 | 218279400 | 218,2794 | 0,2182794 |
| 1000000 | 214049200 | 214,0492 | 0,2140492 |
| 10000000 | 227424900 | 227,4249 | 0,2274249 |

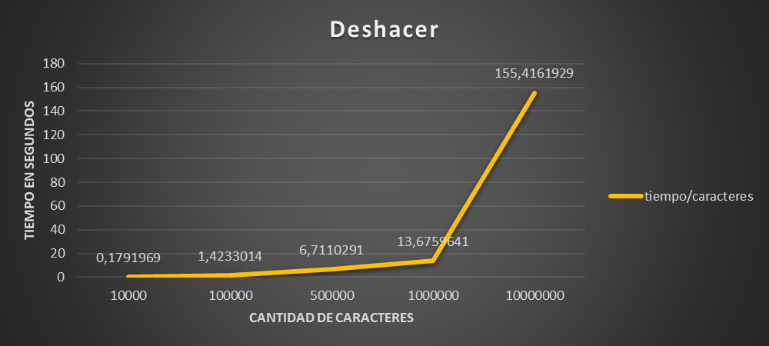


*Complejidad: La complejidad es O(n) ya que se tiene que guardar cada carácter código uno por uno.*

***Deshacer y rehacer:*** Para hacer las pruebas de los botones *deshacer* y *rehacer* se midió el tiempo de ejecución al momento de deshacer y rehacer cierta cantidad de caracteres. A continuación, se muestran las tablas de comparación del tiempo de ejecución respecto a la cantidad de caracteres y sus respectivas gráficas.

*Deshacer*

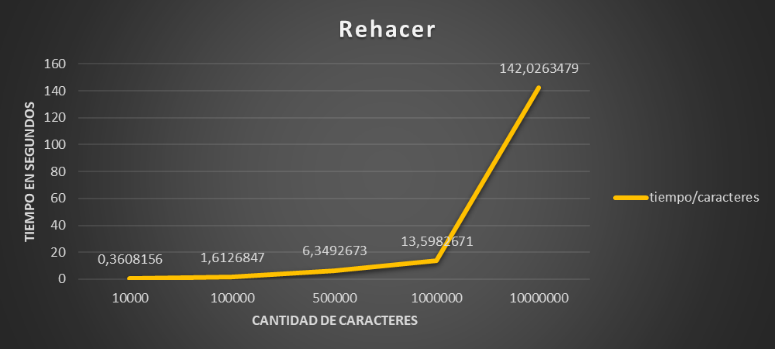
|  |  |  |  |
| --- | --- | --- | --- |
| Cantidad de caracteres | Tiempo en nanosegundos | Tiempo en milisegundos | Tiempo en segundos |
| 10000 | 179196900ns | 179,1969ms | 0,1791969s |
| 100000 | 1423301400ns | 1423,3014ms | 1,4233014s |
| 500000 | 6711029100ns | 6711,0291ms | 6,7110291s |
| 1000000 | 1,3676E+10ns | 13675,9641ms | 13,6759641s |
| 10000000 | 1,5542E+11ns | 155416,193ms | 155,416193s |



*Complejidad: Complejidad: La complejidad es O(1) ya que está devolviendo y removiendo el último elemento ingresado en una pila.*

*Rehacer*

|  |  |  |  |
| --- | --- | --- | --- |
| Cantidad de caracteres | Tiempo en nanosegundos | Tiempo en milisegundos | Tiempo en segundos |
| 10000 | 360815600ns | 360,8156ms | 0,3608156s |
| 100000 | 1612684700ns | 1612,6847ms | 1,6126847s |
| 500000 | 6349267300ns | 6349,2673ms | 6,3492673s |
| 1000000 | 1,3598E+10ns | 13598,2671ms | 13,5982671s |
| 10000000 | 1,4203E+11ns | 142026,348ms | 142,026348s |



*Complejidad: La complejidad es O(1) ya que está devolviendo y removiendo el último elemento ingresado en una pila.*

***Entradas del código:*** Las pruebas de las entradas del código se hicieron dependiendo de la cantidad de entradas. A continuación, se muestran las tablas de comparación del tiempo de ejecución respecto a la cantidad de entradas y sus respectivas gráficas.

*Ingresar entradas*

|  |  |  |  |
| --- | --- | --- | --- |
| Cantidad de entradas | Tiempo en nanosegundos | Tiempo en milisegundos | Tiempo en segundos |
| 10000 | 2461600ns | 2,4616ms | 0,0024616s |
| 100000 | 6989900ns | 6,9899ms | 0,0069899s |
| 500000 | 17990200ns | 17,9902ms | 0,0179902s |
| 1000000 | 425298800ns | 425,2988ms | 0,4252988s |
| 10000000 | 1,1125E+10ns | 11125,1426ms | 11,1251426s |



*Complejidad: La complejidad es O(1) ya que cada entrada se está ingresando al final de una cola.*

*Devolver entradas y eliminarlas*

|  |  |  |  |
| --- | --- | --- | --- |
| Cantidad de entradas | Tiempo en nanosegundos | Tiempo en milisegundos | Tiempo en segundos |
| 10000 | 1232300ns | 1,2323ms | 0,0012323s |
| 100000 | 1672700ns | 1,6727ms | 0,0016727s |
| 500000 | 8944900ns | 8,9449ms | 0,0089449s |
| 1000000 | 8246100ns | 8,2461ms | 0,0082461s |
| 10000000 | 70042100ns | 70,0421ms | 0,0700421s |



*Complejidad: La complejidad es O(1) ya que se está devolviendo y removiendo el primer elemento de una cola.*

# roles y actividades

|  |  |  |  |
| --- | --- | --- | --- |
| Nombre | Grupo | Rol | Actividad |
| Kevin J. Carranza | 5 | Experto | Revisión y corrección de problemas en la implementación del árbol AVL. |
| Ricardo A. Marino | 5 | Líder | Revisión de problemas en la implementación del árbol AVL y preparación del documento a entregar. |
| David C. Cortés | 2 | Investigador | Implementación de árbol AVL para guardar las ventanas de programación. |

# DIFICULTADES Y LECCIONES APRENDIDAS

Una de las mayores de las mayores dificultades fue la implementación de un árbol AVL para guardar los nodos, debido a que cuando se necesitaba acceder a un nodo se debía recorrer el árbol original y no una copia, cómo se haría cuando se requiere solo saber si dicho nodo está en el árbol o no.

Por otro lado, cuando se hizo la comparación de tiempos con la anterior implementación para guardar las ventanas de programación, se descubrió que, aunque un árbol AVL es mucho mejor que una lista enlazada en el acceso de un elemento arbitrario, la implementación de lista enlazada que estábamos utilizando era la que tiene por defecto JAVA y está tiene muchas mejoras de modo que la hace incluso mas eficiente en ciertos casos que un árbol AVL implementado comúnmente. Cuando hicimos las pruebas de tiempo entre un árbol AVL y una lista enlazada implementados comúnmente, los tiempos empezaron a ser los esperados.

1. [↑](#footnote-ref-1)