República Bolivariana de Venezuela.

Ministerio del Poder Popular para la Educación Universitaria, Ciencias y Tecnología.

Colegio Universitario "Francisco de Miranda".

Carrera (Informática)

Sección I08-302. (Nocturno)

Manejo de Archivos, Pilas, Colas y Árboles.

Profesor: **Estudiantes:**

Marcos Andrade **Javier Ramírez C.I.: V- 26.152.070**

Caracas, Octubre de 2016.

Índice

Pag.

Introducción

La siguiente información organizada de manera explícita y detallada añadiendo ejemplos para complementar los datos en cuanto a las interrogantes expuestas como referencia a un lenguaje de programación especifico, con el qué se trabajará durante el desarrollo de este trabajo de investigación, en el cual se manifiestan diferentes aspectos, proporcionando equidad y variedad de información útil para el desarrollo óptimo de programas que pueden ser de gran ayuda para la preparación y ejecución de Listas con la finalidad de obtener datos como por ejemplo el de una guía telefónica, listas de asistencia a un curso, índice de un libro en especial, listado de compras domesticas o laborales, listado de ingredientes de una receta, entre otros.

Los elementos que marcan pauta encontrados en el presente trabajo son: El manejo de archivos y sus aspectos más importantes, Funciones que también realizan una tarea específica. En general toman ciertos valores de entrada, llamados parámetros y proporcionan un valor de salida o valor de retorno, también detallado en este medio de trabajo.

A continuación se presentarán los complementos restantes en la inserción de datos en el Manejo de archivos como son las pilas, colas, listas y árboles, siendo de suma importancia y de gran enfoque en el mismo.

Manejo de Archivos

A) Inserción

En el archivo de cabecera **fstream.h** define las clases **ifstream** para operaciones de lectura, **ostream** y **fstream** escritura y lectura/escritura en archivos respectivamente. Para trabajar con archivos debemos crear objetos de éstas clases de acuerdo a las operaciones que deseamos efectuar. Empezamos con las operaciones de escritura, para lo cual básicamente declaramos un objeto de la clase ofstream, después utilizamos la función miembro **open** para abrir el archivo, escribimos en el archivo los datos que sean necesarios utilizando el operador de inserción y por último cerramos el archivo por medio de la función miembro **close**, éste proceso está ilustrado en nuestro primer programa, con el nombre que se le quiera asignar, como por ejemplo; **archiv01.cpp**.

```
int main()
{
  ofstream archivo;  // objeto de la clase ofstream

  archivo.open("datos.txt");

  archivo << "Primera línea de texto" << endl;
  archivo << "Segunda línea de texto" << endl;
  archivo << "Última línea de texto" << endl;

  archivo.close();
  return 0;
}
```

Por ejemplo si se crea un programa y se añade un objeto de la clase ofstream llamado archivo, posteriormente se utiliza la función miembro open para abrir el archivo especificado en la cadena de texto que se encuentra dentro del paréntesis de la función. Podemos invocar a la función constructora de clase de tal manera que el archivo también se puede abrir utilizando la siguiente instrucción:

```
ofstream archivo("datos.txt");  // constructora de ofstream
```

Al utilizar la función constructora ya no es necesario utilizar la función miembro open, ésta forma de abrir un archivo es preferida porque el código es más compacto y fácil de leer. De la misma forma que se utilizan manipuladores de salida para modificar la presentación en pantalla de los datos del programa, igual es posible utilizar estos manipuladores al escribir datos en un archivo como lo demuestra el programa **archiv02.cpp**, observe que se utiliza un constructor para crear y abrir el archivo llamado **Datos.txt**.

B) Consulta

Usar streams facilita mucho el acceso a ficheros en disco, sé verá que una vez que se cree un stream para un fichero, se puede trabajar con él igual que se hace con cin o cout.

Mediante las clases ofstream, ifstream y fstream se tiene acceso a todas las funciones de las clases base de las que se derivan estas: ios, istream, ostream, fstreambase, y como también contienen un objeto filebuf, se puede acceder a las funciones de filebuf y streambuf.

Evidentemente, muchas de estas funciones puede que no sean de utilidad, pero algunas de ellas se usan con frecuencia, y facilitan mucho el trabajo con ficheros.

El programa sobrescribe cualquier archivo existente llamado Datos.txt en el directorio de trabajo del programa. Dependiendo del propósito del programa es posible que sea necesario agregar datos a los ya existentes en el archivo, o quizá sea necesario que las operaciones del programa no se lleven a cabo en caso de que el archivo especificado exista en el disco, para éstos casos podemos especificar el modo de apertura del archivo incluyendo un parámetro adicional en el constructor, cualquiera de los siguientes:

ios::app Operaciones de añadidura.

ios::ate Coloca el apuntador del archivo al final del mismo.

ios::in Operaciones de lectura. Esta es la opción por defecto para objetos de la clase ifstream.

ios::out Operaciones de escritura. Esta es la opción por defecto para objetos de la clase ofstream.

ios::nocreate Si el archivo no existe se suspende la operación.

ios::noreplace Crea un archivo, si existe uno con el mismo nombre la operación se suspende.

ios::trunc Crea un archivo, si existe uno con el mismo nombre lo borra.

ios::binary Operaciones binarias.

De esta manera, podemos modificar el modo de apertura del programa **archiv02.cpp** para que los datos del programa se concatenen en el archivo **Datos.txt** simplemente escribiendo el constructor así: **ofstream archivo("Datos.txt", ios::app);** Si deseamos que el programa no sobrescriba un archivo existente especificamos el constructor de ésta manera: **ofstream archivo("Datos.txt", ios::noreplace);** Utilizando los especificadores de modo de apertura se puede conseguir un mayor control en las operaciones de E/S en archivos.

Se creara un fichero mediante un objeto de la clase ofstream, y posteriormente se leeram mediante un objeto de la clase ifstream:

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  char cadena[128];
  // Crea un fichero de salida
  ofstream fs("nombre.txt");

  // Enviamos una cadena al fichero de salida:
  fs << "Hola, mundo" << endl;
  // Cerrar el fichero,
  // para luego poder abrirlo para lectura:
  fs.close();

  // Abre un fichero de entrada
  ifstream fe("nombre.txt");

  // Leeremos mediante getline, si lo hiciéramos
  // mediante el operador << sólo leeríamos
  // parte de la cadena:
  fe.getline(cadena, 128);

  cout << cadena << endl;

  return 0;
```

```
}
```

Se mostrará otro ejemplo, para ilustrar algunas limitaciones del operador >> para hacer lecturas, cuando no queremos perder caracteres.

Se supondrá que el programa será llamado "streams.cpp", y que se autoimprima en pantalla:

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  char cadena[128];
  ifstream fe("streams.cpp");

  while(!fe.eof()) {
    fe >> cadena;
    cout << cadena << endl;
  }
  fe.close();

  return 0;
}
```

El resultado quizá no sea el esperado. El motivo es que el operador >> interpreta los espacios, tabuladores y retornos de línea como separadores, y los elimina de la cadena de entrada.

C) Eliminación

Código C++ – Agregar, eliminar, insertar y buscar elementos

```
1  //codificado por sAfOrAs
2  //Agregar , eliminar e insertar y buscar elementos
3  //el tamaño maximo del arreglo es de 100 pero el número de elementos debe elegirlo.
4  #include<iostream>
5  #include "leearray.h"
6  using namespace std;
7  #define MAX 100
8  int leeCantidadElem()
9  {
10    int n;
11    do{
12        cout<<"Cantidad de elementos a ingresar: ";cin>>n;
13            if(n<=0)
14                cout<<"...No seas payaso(a), ingresa una cantidad correcta: "<<endl;
15            if(n>MAX)
16                cout<<"...La cantidad maxima permitida es "<<MAX<<" : "<<endl;
17    }while(n<=0 || n>MAX);
18    return n;
19 }
20 int elegirEvento(int cant,int A[])
21 {
22 Opciones:
23    int i,k,elem,opt;
```

```
24      cout<<"1. Insertar elemento: "<<endl;
25      cout<<"2. Eliminar elemento: "<<endl;
26      cout<<"3. Agregar elemento: "<<endl;
27      cout<<"4. Buscar elemento: "<<endl;
28      cout<<"Elija una opcion 1 , 2 , 3 o 4: ";cin>>opt;
29      switch(opt)
30      {
31          case 1:
32              {
33                  cout<<"\t>>Que elemento desea insertar: ";cin>>elem;
34                  do{
35                  cout<<"\t>>En que posicion desea insertar...de [0] hasta ["<<cant-1<<"]: ";cin>>k;
36                      if(k>(cant-1)||k<0)
37                          cout<<">>Ingrese una posicion valida!!!"<<endl;
38                  }while(k>(cant-1)||k<0);
39                  cant++;
40                  for(i=cant-1;i>=k;i--)
41                  {
42                      A[i+1]=A[i];
43                      if(k==i)
44                      A[k]=elem;
45                  }
46              }break;
```

```
47      case 2:
48          {
49             do{
50             cout<<"\t>>Que posicion desea eliminar...de [0] hasta ["<<cant-1<<"]: ";cin>>k;
51             if(k>(cant-1)||k<0)
52                   cout<<">>Ingrese una posicion valida!!!"<<endl;
53             }while(k>(cant-1)||k<0);
54             for(i=k;i<cant;i++)
55             {
56                A[i]=A[i+1];
57             }
58                cant--;
59          }break;
60      case 3:
61          {
62             for(i=0;i<1;i++)
63             {
64                cout<<"\t>>Que elemento desea agregar : ";cin>>elem;
65             Agregar:
66                cant++;
67                A[cant-1]=elem;
68             }
69          }break;
```

```
70      case 4:
71          {
72              cout<<"\t>>Que elemento desea buscar: ";cin>>elem;
73              for(i=0;i<cant;i++)
74              {
75                  if(A[i]==elem)
76                  {
77                      cout<<"\t>>El elemento buscado se encuentra en: A["<<i<<"]"<<endl;
78                      //Añandir el elemento al final de arreglo
79                      cout<<"\t>>El elemento se agregara al final"<<endl;
80                  goto Agregar;
81                  }
82                  else
83                  {
84                      if(i==cant-1)
85                      {
86                      cout<<"\t>>No se encuetra el elemento que busca!!!"<<endl;
87                      cout<<"\t>>Puede confirmarlo viendolo Ud. mismo!!!"<<endl;
88                      }
89                  }
90              }
91          }break;
92      default:system("cls");cout<<"No existe esa opcion, vuelva a intentar: "<<endl;goto Opciones;break;
```

```
93      }
94  return cant;
95  }
96  void main()
97  {
98      int c;
99      char opt;
100     int n[MAX];
101     cout<<"\t\t\tAGREGAR 2 ELEMENTOS AL FINAL"<<endl;
102     c=leeCantidadElem();
103     leeCadena(c,n);
104     do{
105     c=elegirEvento(c,n);
106     muestraCadena(c,n);
107     cout<<"Desea realizar otra operacion!!!... S/s, caso contrario pulse otra tecla: ";cin>>opt;
108     }while(opt=='s'||opt=='S');
109 }
```

Funciones para el manejo de datos en archivos C++

Las funciones son un conjunto de instrucciones que realizan una tarea específica. En general toman ciertos valores de entrada, llamados parámetros y proporcionan un valor de salida o valor de retorno.

Las funciones son una herramienta muy valiosa, y como se usan en todos los programas C++, se debe tener, al menos, una primera noción de su uso. A fin de cuentas, todos los programas C++ contienen, como mínimo, una función.

B) Prototipos de funciones

En C++ es obligatorio usar prototipos. Un prototipo es una declaración de una función. Consiste en una presentación de la función, exactamente con la misma estructura que la definición, pero sin cuerpo y terminada con un ";". La estructura de un prototipo es:

```
[extern|static] <tipo_valor_retorno> [<modificadores>]
<identificador>(<lista_parámetros>);
```

El prototipo de una función se compone de las siguientes secciones:

- Opcionalmente, una palabra que especifique el tipo de almacenamiento, puede ser **extern** o **static**. Si no se especifica ninguna, por defecto será **extern**.
- El tipo del valor de retorno, que puede ser **void**, si no necesitamos valor de retorno. En C++ es obligatorio indicar el tipo del valor de retorno.
- Modificadores opcionales. Tienen un uso muy específico, de momento no entraremos en este particular.
- El identificador de la función. Es muy útil y también recomendable, poner nombres que indiquen, lo más claramente posible, qué es lo que hace la función, y que permitan interpretar qué hace el programa con sólo leerlos. Cuando se precisen varias palabras para conseguir este efecto se puede usar alguna de las reglas más usuales. Una consiste en separar cada palabra con un "_". Otra, consiste en escribir la primera letra de cada palabra en mayúscula y el resto en minúsculas. Por ejemplo; si hacemos una función que busque el número de teléfono de una persona en una base de datos, podríamos llamarla **"busca_telefono"** o **"BuscaTelefono".**

- Una lista de declaraciones de parámetros entre paréntesis. Los parámetros de una función son los valores de entrada (y en ocasiones también de salida). Para la función se comportan exactamente igual que variables, y de hecho cada parámetro se declara igual que una variable.
- Una lista de parámetros es un conjunto de declaraciones de parámetros separados con comas. Puede tratarse de una lista vacía. En C es preferible usar la forma "func(void)" para listas de parámetros vacías. En C++ este procedimiento se considera obsoleto, se usa simplemente "func()".

Por ejemplo:

```
int Mayor(int a, int b);
```

Un prototipo sirve para indicar al compilador los tipos de retorno y los de los parámetros de una función, de modo que compruebe si son del tipo correcto cada vez que se use esta función dentro del programa, o para hacer las conversiones de tipo cuando sea necesario.

En el prototipo, los nombres de los parámetros son opcionales, y si se incluyen suele ser como documentación y ayuda en la interpretación y comprensión del programa.

El ejemplo de prototipo anterior sería igualmente válido si se escribiera como:

```
int Mayor(int, int);
```

Pilas

Una de las aplicaciones más interesantes y potentes de la memoria dinámica y de los punteros son, sin duda, las estructuras dinámicas de datos. Siendo las estructuras básicas disponibles en lenguaje C++, (structs y *arrays*) tienen una importante limitación: no pueden cambiar de tamaño durante la ejecución. Los *arrays* están compuestos por un determinado número de elementos, número que se decide en la fase de diseño, antes de que el programa ejecutable sea creado.

En muchas ocasiones se necesitan estructuras que puedan cambiar de tamaño durante la ejecución del programa. Por supuesto, podemos crear arrays dinámicos, pero una vez creados, tu tamaño también será fijo, y para hacer que crezcan o disminuyan de tamaño, deberemos reconstruirlos desde el principio.

Las estructuras dinámicas nos permiten crear estructuras de datos que se adapten a las necesidades reales a las que suelen enfrentarse nuestros programas, también nos permitirán tener relación entre los elementos que las componen.

Una estructura básica de un nodo para crear listas o pilas dinámicas de datos seria:

```
struct nodo {
  int dato;
  struct nodo *otronodo;
};
```

El campo "otronodo" puede apuntar a un objeto del tipo nodo. De este modo, cada nodo puede usarse como un ladrillo para construir listas de datos, y cada uno mantendrá ciertas relaciones con otros nodos: para acceder a un nodo de la estructura sólo necesitaremos un puntero a un nodo.

Dependiendo del número de punteros y de las relaciones entre nodos, podemos distinguir varios tipos de estructuras dinámicas. Enumeraremos ahora sólo de los tipos básicos:

- **Pilas:** Son un tipo especial de lista, conocidas como listas LIFO (Last In, First Out: el último en entrar es el primero en salir). Los elementos se "amontonan"

o apilan, de modo que sólo el elemento que está encima de la pila puede ser leído, y sólo pueden añadirse elementos encima de la pila.

Es una estructura ideada con el propósito de albergar datos agrupados bajo un mismo nombre. Al respecto, podemos pensar que las listas son como arreglos de datos, es decir, para hacer una introducción al manejo y programación de listas encadenadas podemos tomar como punto de partida a los arreglos estáticos. Es así como en esta sección se descubrirá la forma de operación de tres tipos comunes de listas conocidas como: PILAS, COLAS Y DOBLE COLA (STACK, QUEUE, DQUEUE). En programación, el uso de listas es una práctica tan extendida que lenguajes tales como (por ejemplo) Java, Python y C++ soportan los mecanismos necesarios para trabajar con estructuras de: Vectores, Pilas, Colas, Listas, etc. En C++.

D) Pilas o Stacks

Es una estructura en donde cada elemento es insertado y retirado del tope de la misma, y debido a esto el comportamiento de un una pila se conoce como LIFO (último en entrar, primero en salir).

Un ejemplo de pila o stack se puede observar en el mismo procesador, es decir, cada vez que en los programas aparece una llamada a una función el microprocesador guarda el estado de ciertos registros en un segmento de memoria conocido como Stack Segment, mismos que serán recuperados al regreso de la función.

E) Pila en arreglo estático

En el programa que se verá en seguida, se simula el comportamiento de una estructura de pila. Aunque en el mismo se usa un arreglo estático de tamaño fijo se debe mencionar que normalmente las implementaciones hechas por fabricantes y/o terceras personas se basan en listas dinámicas o enlazadas.

Para la implementación de la clase Stack se han elegido los métodos:

```
put(),   poner un elemento en la pila
get(),   retirar un elemento de la pila
empty(), regresa 1 (TRUE) si la pila esta vacia
size(),  número de elementos en la pila
```

F) Operaciones básicas de la PILA

- *Crear* Inicializar una lista vacía.
- *Lista vacía* Determinar si una lista está vacía.
- *Lista llena* Determina si la lista se ha llenado.
- *Insertar* Inserta un elemento en la lista de forma que siga ordenada.
- *Buscar* Busca un determinado elemento dentro de la lista.
- *Borrar* Busca y elimina un elemento en la lista, manteniendo el orden.

Colas

Es una estructura de datos, caracterizada por ser una secuencia de elementos en la que la operación de inserción push se realiza por un extremo y la operación de extracción pop por el otro. También se le llama estructura FIFO (del inglés First In First Out), debido a que el primer elemento en entrar será también el primero en salir.

Las colas se utilizan en dónde los objetos, personas o eventos son tomados como datos que se almacenan y se guardan mediante colas para su posterior procesamiento. Este tipo de estructura de datos abstracta se implementa en lenguajes orientados a objetos mediante clases, en forma de listas enlazadas.

Ejemplo de colas en c++:

```
  #ifndef COLA
#define COLA // Define la cola
using namespace std;
template <class T>
class Cola{
   private:
     struct Nodo{
        T elemento;
        struct Nodo* siguiente;  // coloca el nodo en la segunda posición
     }* primero;
     struct Nodo* ultimo;
     unsigned int elementos;
   public:
     Cola(){
        elementos = 0;
     }
       cout<<" Hola Mundo! " <<endl;
       cout<<" Hello, World! " <<endl;
     ~Cola(){
        while (elementos != 0) pop();
     }
     void push(const T& elem){
        Nodo* aux = new Nodo;
        aux->elemento = elem;
```

```
        if (elementos == 0) primero = aux;
        else ultimo->siguiente = aux;
        ultimo = aux;
        ++elementos;
    }
    void pop(){
        Nodo* aux = primero;
        primero = primero->siguiente;
        delete aux;
        --elementos;
    }
    T consultar() const{
        return primero->elemento;
    }
    bool vacia() const{
        return elementos == 0;
    }
    unsigned int size() const{
        return elementos;
    }
};
#endif
```

Listas

Una lista lineal es un conjunto de elementos de un tipo dado que se encuentran **ordenados** y pueden variar en número.

Una lista enlazada o encadenada es un conjunto de elementos más un campo especial que contiene el pun-tero al elemento siguiente de la lista.

- Cada elemento de la lista debe tener al menos dos campos:
- Elemento o dato.
- Enlace, link, conexión al siguiente elemento.
- Los elementos de una lista son enlazados por medio de los campos enlaces.

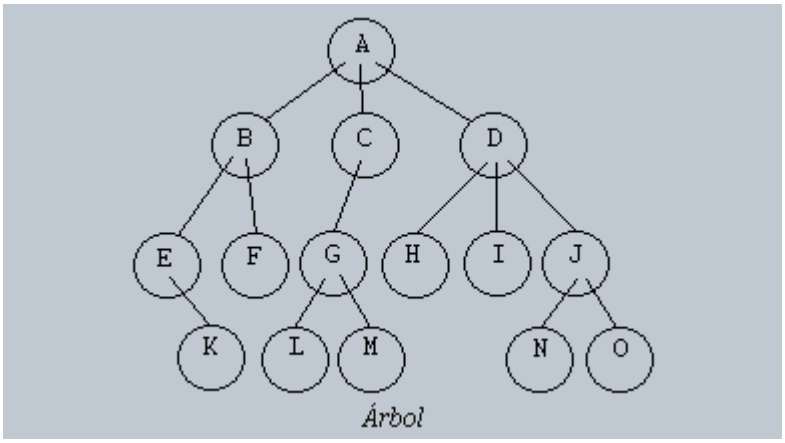
A) Utilidad de las listas

- Permite el recorrido de todos y cada uno de sus elementos, sin saltear ninguno y en forma **ordenada**.
- Guía telefónica
- Lista de asistencia a un curso
- Índice de un libro
- Listado de compras
- Listado de ingredientes de una receta

Árboles

Son estructuras de datos ampliamente usadas que imitan la forma de un **árbol** (un conjunto de nodos conectados). Un **nodo** es la unidad sobre la que se construye el árbol y puede tener cero o más nodos hijos conectados a él. Se dice que un nodo {**a**} es **padre** de un nodo {**b**} si existe un enlace desde {**a**} a hasta {**b**} (en ese caso, también decimos que {**b**} es hijo de {**a**}). Sólo puede haber un único nodo sin padres, que llamaremos **raíz**.

Un nodo que no tiene hijos se conoce como hoja. Los demás nodos (tienen padre y uno o varios hijos) se les conoce como **rama**. Un árbol es una estructura en compuesta por un dato y varios árboles.



Árbol

En relación con otros nodos:

- **Nodo hijo**: Cualquiera de los nodos apuntados por uno de los nodos del árbol.

En el ejemplo, 'L' y 'M' son hijos de 'G'.

- **Nodo padre**: Nodo que contiene un puntero al nodo actual.

En el ejemplo, el nodo 'A' es padre de 'B', 'C' y 'D'.

Se puede trabajar con árboles que tengan una característica adicional muy importante: cada nodo sólo puede ser apuntado por otro nodo, es decir, cada nodo sólo tendrá un padre. Esto hace que estos árboles estén fuertemente jerarquizados, y es lo que en realidad les da la apariencia de árboles.

A) Tipos de Árboles

- **Árbol binario**: Es una estructura de datos en la cual cada nodo puede tener un hijo izquierdo y un hijo derecho. No pueden tener más de dos hijos (de ahí el nombre "binario"). Si algún hijo tiene como referencia a **null**, es decir que no almacena ningún dato, entonces este es llamado un nodo externo. En el caso contrario el hijo es llamado un nodo interno.
- **Árbol binario de búsqueda auto-balanceable o equilibrado**: Es un árbol binario de búsqueda que intenta mantener su altura, o el número de niveles de nodos bajo la raíz, tan pequeños como sea posible en todo momento, automáticamente. Esto es importante, ya que muchas operaciones en un árbol de búsqueda binaria tardan un tiempo proporcional a la altura del árbol, y los árboles binarios de búsqueda ordinarios pueden tomar alturas muy grandes en situaciones normales, como cuando las claves son insertadas en orden.
- **Árbol AVL:** Es un tipo especial de árbol binario ideado por los matemáticos rusos **A**delson-**V**elskii y **L**andis. Fue el primer árbol de búsqueda binario auto-balanceable que se ideó.
- **Árbol rojo-negro**: Es un tipo abstracto de datos. Concretamente, es un árbol binario de búsqueda equilibrado.
- **Árbol AA**: Es un tipo de árbol binario de búsqueda auto-balanceable utilizado para almacenar y recuperar información ordenada de manera eficiente.
- **Árbol de segmento:** Es una estructura de datos en forma de árbol para guardar intervalos o segmentos. Permite consultar cuál de los segmentos guardados contiene un punto.
- **Árboles multicamino o árboles multirrama**: Son estructuras de datos de tipo árbol usadas en computación.
- **Árbol-B:** La idea tras los árboles-B es que los nodos internos deben tener un número variable de nodos hijo dentro de un rango predefinido. Cuando se inserta o se elimina un dato de la estructura, la cantidad de nodos hijo varía dentro de un nodo. Para que siga manteniéndose el número de nodos dentro del rango predefinido, los nodos internos se juntan o se parten. Dado que se permite un rango variable de nodos hijo, los árboles-B no necesitan rebalancearse tan frecuentemente como los árboles binarios de búsqueda auto-balanceables.
- **Árbol B+:** Es un tipo de estructura de datos de árbol, representa una colección de datos ordenados de manera que se permite una inserción y borrado eficientes de elementos. Es un índice, multinivel, dinámico, con

un límite máximo y mínimo en el número de claves por nodo. Un árbol B+ es una variación de un árbol B.

- **Árbol-B*:** Es una estructura de datos de árbol, una variante de Árbol-B utilizado en los sistemas de ficheros HFS y Reiser4, que requiere que los nodos no raíz estén por lo menos a 2/3 de ocupación en lugar de 1/2.

B) Uso de Los Árboles

- Representación de datos jerárquicos.
- Como ayuda para realizar búsquedas en conjuntos de datos (ver también: algoritmos de búsqueda en Árboles).

Conclusión

A partir de la información percibida en el anterior trabajo se aclaran puntos que a margen del lenguaje de programación C++ son de suma importancia, de esta forma se puede ejecutar, resolver y continuar utilizando las herramientas existentes para desarrolla un programa de gran utilidad

En estos casos se puede usar terminología de relaciones familiares para descubrir las relaciones entre los nodos de un árbol; y que un árbol puede ser implementado fácilmente en una computadora.

Es bueno hacer énfasis en esto ya que se puede saber mucho sobre lo que tiene que ver con los árboles; entre las cosas que podemos mencionar se encuentra la raíz, los nodos de un árbol y la diferencia entre nodos sucesores y nodos terminales, como se muestran en el contenido del trabajo.

Bibliografías.

Manejo de archivos en C++:

http://www.programacionenc.net/index.php?option=com_content&view=article&id=69:manejo-de-archivos-en-c&catid=37:programacion-cc&Itemid=55

B) Consulta: http://c.conclase.net/curso/?cap=039
C) Eliminación: https://saforas.wordpress.com/2008/01/03/codigo-c-agregar-eliminar-insertar-y-buscar-elementos/

Funciones para el manejo de datos en archivos C++:

http://c.conclase.net/curso/?cap=003#inicio

Pilas:

http://c.conclase.net/edd/?cap=000#inicio

https://es.wikibooks.org/wiki/Programaci%C3%B3n_en_C%2B%2B/Estructuras_II

Colas:

https://es.wikipedia.org/wiki/Cola_(inform%C3%A1tica)

Listas:

https://es.wikibooks.org/wiki/Programaci%C3%B3n_en_C%2B%2B/Librer%C3%ADa_Est%C3%A1ndar_de_Plantillas/Listas#C.2B.2B_List_est.C3.A1ndar

Árboles, tipos y usos:

https://es.wikipedia.org/wiki/%C3%81rbol_(inform%C3%A1tica)#Tipos_de_.C3.A1rboles