Implantación de un *B Tree+*

Amilcar Meneses Viveros
ameneses@computacion.cs.cinvestav.mx

Original: Mayo – 1995
Correcciones: Agosto – 2003

Resumen

Los árboles **B** o multivias ofrecen ventajas en la búsqueda de datos. Sin embargo su deficiencia se presenta cuando se desean hacer recorridos secuenciales de sus registros. Un *B-Tree+* es una estructura de datos que se utiliza como alternativa eficaz de implantación de los árboles **B**. Corrigiendo el problema de búsqueda secuencial al ligar los nodos finales del árbol como si fuera una lista. En este trabajo se presenta la implantación de un *B-Tree +*.

Introducción

Un *B-Tree+* —algunos autores lo llaman simplemente *B+*— es una estructura de datos que se utiliza como mecanismo de organización para un número considerable de datos dispuestos en registros, estos datos son accesados por un campo principal denominado *campo llave*. Cuando se maneja un número considerable de registros organizados de manera secuencial es difícil obtener un buen rendimiento en la organización de estos datos —al mencionar organización nos referimos a la búsqueda o recuperación de datos y a la eliminación e inseción de datos—. Por otro lado existen otras estructuras de datos que se especializan en resolver los problemas de eficiencia que presenta el modelo secuencial; sin embargo, su implantación ocasiona algunos problemas de tiempo y, en algunos aspectos, de organización. Entre estas estructuras podemos mencionar a las listas, pilas, árboles binarios simples, árboles binarios balanceados y árboles multivias (árboles B), entre otros.

1 Árboles

Un árbol es un conjunto de nodos donde cada nodo puede tener 0 o más hijos y a lo más un padre. Existe un único nodo que no tiene padre y es denominado *nodo raíz*[1]. Los nodos que tienen hijos se llaman ramas. Y los nodos que no tienen hijos se llaman hojas.

Tomaremos el concepto general de árbol como el definido por algunos autores como *árbol de búsqueda de acceso múltiple*:

> "Un **árbol de búsqueda de acceso múltiple de orden n** es un árbol general en el cual cada nodo tiene n o menos subárboles y contiene una llave menos que la cantidad de subárboles... Además si $s_0, s_1, \ldots, s_{m-1}$ son los m subárboles de un nodo que contiene llaves $k_0, k_1, \ldots, k_{m-2}$ en orden ascendente, todas las llaves en el subárbol s_0 son menores o iguales que k_0, todas las llaves en el subárbol s_j (donde j está entre

[1] Si exiten más nodos que no tiene padre, entonces no referimos a una estructura llamada *bosque*

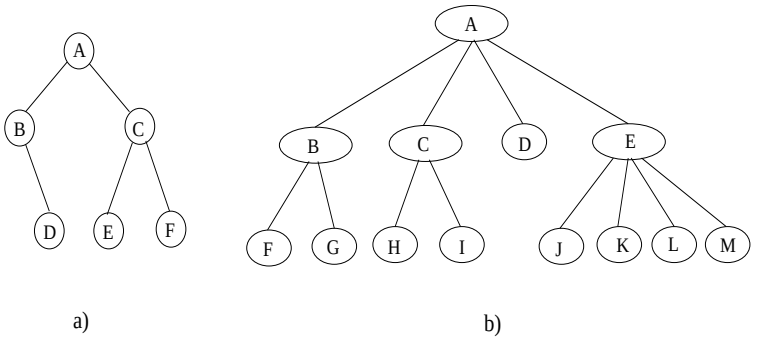

Figura 1: Árboles de orden 2 (a) y de orden 4 (b).

1 y $m-2$) son mayores que k_{j-1} y menores o iguales que k_j, y todas las llaves en el subárbol s_{m-1} son mayores que k_{m-2}. El subárbol s_j se llama el *subárbol izquierdo* de llave k_j y su raíz el *hijo izquierdo* de llave k_j. De manera similar s_j se llama el *subárbol derecho* y su raíz el *hijo derecho*, de llave k_{j-1}"[1].

Así, de manera particular un árbol binario es un árbol de orden 2 donde cada nodo tiene una llave y dos subárboles hijos. Como se puede apreciar en la figura 1.

Los árboles presentan problemas de eficiencia si no se construyen en forma balanceada, lo que puede resultar en la creación de un árbol degenerado —árboles que se construyen en base a una entrada ordenada y al terminar de construirlos se asemenjan a una lista ligada —. Los otros problemas a considerar son las operaciones que se realizan sobre los datos del árbol como son: el tiempo de recuperación, borrar datos, insertar datos y recorrer en forma ordenada todos sus elementos.

2 El B+

Un $B+$ es un árbol balanceado multivías de orden mayor a 2 que tiene las siguientes carácterísticas:

- Los nodos "rama" únicamente tienen las llaves y apuntadores a sus subárboles hijos.
- Los nodos "hoja" son los únicos que tienen apuntadores a los registros correspondientes a las llaves. Esto es, si un nodo que no es hoja tiene la llave que se busca, entonces este nodo indica que tiene un subárbol-hijo que tiene la hoja con el elemento deseado.
- La altura de cada hoja siempre es la misma o igual a la profundidad[2] del árbol. Es decir, para accesar a un elemento del árbol siempre hay que recorrer el mismo número de niveles desde la raíz.
- Las hojas tiene ligas entre si, semejantes a las listas ligadas, lo que facilita el recorrido secuencial de los datos.

como se muestra en la figura 2.

[2]La profundidad de un árbol es el nivel máximo de sus hojas.

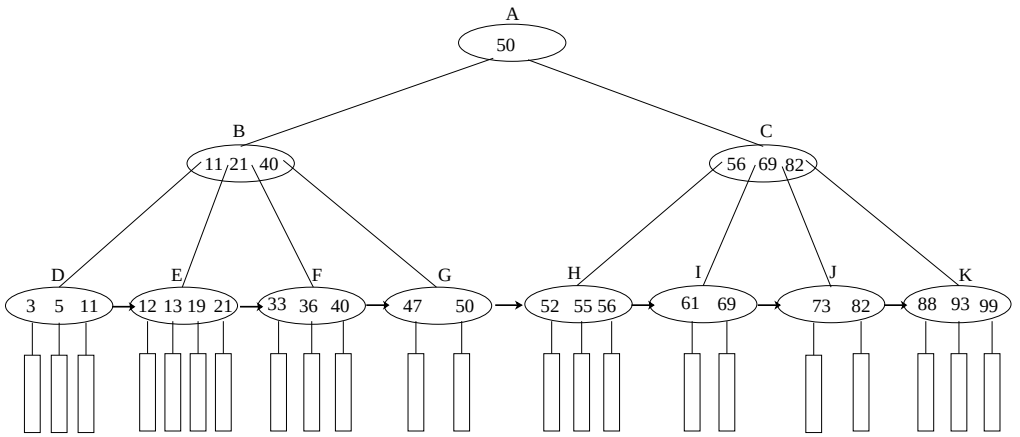

Figura 2: Estructura de un **B+**

3 Los Nodos

Los nodos del *B+* deben almacenar hasta $n-1$ llaves[3], el tipo de datos de la llave puede variar dependiendo de que campo se desea utilizar como "campo llave". Esto es, el campo llave puede ser de tipo entero, flotante, carácter o cadena.

En la implantación que presentamos se incluye un archivo de cabecera auxiliar *tipos.h*, donde se definen la estructura del registro que manejará el *B+* y el tipo de campo llave que utilizará, además de otras definiciones que resulten convenientes para trabajar la estructura de los registros en el árbol[4].

```
/* tipos.h                              */
/* Archivo donde se define la estructura */
/* y llave que maneja el arbol B+       */
/* [ABRIL-97]                           */
/* AMILCAR MENESES VIVEROS              */

typedef int Bkey;                       /* Tipo de llave */

/* Estructura del registro que se maneja en el arbol B+ */
typedef struct s_register {             /* Variable tipo registro */
   int edad;
   char nombre[30];
} Bregister;

/* Llave que se maneja en el arbol B+ */
#define Fkey       %d                  /* Formato de la llave           */
```

[3]Donde n es el orden del árbol.

[4]Evidentemente podría ser más significativo tener un apuntador general para manejar cualquier tipo de registro, pero por fines ilustrativos se presenta en esta forma.

```
#define FFkey      "%d"                /* Formato de la llave             */
#define BKeyNull   0                   /* Valor nulo de la llave          */
#define KeyNode(p) (p.edad)            /* Campo que se utiliza como llave */
#define PKeyNode(p) ((p)->edad)        /* Campo llave desde un apuntador  */
```

Debido al tipo de organización de los datos que se debe realizar con el *B+*, la estructura de un nodo de estos árboles deben tener un arreglo de n apuntadores a los subárboles "hijo" (donde n es el orden del árbol); un arreglo de $n-1$ llaves; un arreglo de $n-1$ apuntadores a los registros del nodo "hoja"; un contador para el número de llaves que tenga el nodo; un apuntador al nodo padre; y un apuntador al nodo "hoja" subsecuente. De esta forma, la implantación de la estructura *node* queda de la siguiente forma:

```
typedef struct _node {
   int  nKeys;                 /* Numero de hojas                        */
   Bkey ak[ORDER-1];           /* Arreglo de llaves                      */
   Bregister *ap[ORDER-1];     /* Arreglo de apuntadores a los registros */
   struct _node *an[ORDER];    /* Arreglo de apuntadores a los hijos     */
   struct _node *nxt;          /* Apuntador a la hoja mas proxima        */
   struct _node *up;           /* Apuntador al nodo padre                */
} node;
```

4 Búsqueda

La búsqueda en un *B+* es similar a la busqueda de un *BTree*[5], la única diferencia con éste último es que el nodo que contiene el apuntador al registro, en el *B+*, debe ser un nodo hoja.

Suponiendo que deseamos encontrar la llave **k**, entonces el algoritmo inicia la búsqueda en el nodo raíz y compara todos los elementos llaves, en orden ascendente, con **k** hasta que encontremos una llave mayor o terminemos de comparar todas las llaves sin encontrar la llave mayor. Para esta tarea implantamos una funcón llamada *node_search*, la cual tiene 2 entradas: el nodo y **k**, y regresa el índice donde podría estar la llave, en realidad este número es el índice de la máxima llave menor o igual a **k**. Por ejemplo, si el nodo tuviese las llaves (5, 10, 12, 17, 20) y $\mathbf{k} = 15$, entonces *node_search* nos indicaría que el tercer elemento (12) es la máxima llave menor o igual a 15, lo que establece que si el nodo es una rama es que posiblemente 15 se encuentre en el tercer subárbol hijo de nodo. En C esta función queda implantada como:

```
/* Regresa el menor entero j    */
/* tal que k <= key(p,j)        */
/* [ABRIL-97]                   */
/* AMILCAR MENESES VIVEROS      */
int nodesearch(node *p, Bkey k)
{
   int i;

   for (i=0; i < (p->nKeys); i++)
```

[5]En los *Btrees* el nodo que tiene la llave buscada también tiene el apuntador al resto de los datos correspondientes a la llave, no importando si el nodo es rama u hoja.

```
     if ( cmpkey(k,p,i) <= 0 )
         return i;

   return i;
}
```

Esta función incluye, a su vez, a la función *cmpkey()*, que compara a la llave **k**, con el i-ésimo elemento del nodo p, si **k** es menor regresa -1, 0 si es igual y 1 si es mayor.

Retomando la filosofía de *node_seach* escribimos una función que se encarga de buscar, en el $B+$, la hoja y posición donde posiblemente se encuentre **k**, esto és, la posición correspondiente con la máxima llave menor o igual que **k**. Esto tiene sus ventajas para otras operaciones que se realizan en el árbol. Así, la función *search_leaf* tiene como valores de entrada el nodo donde se inicia la búsqueda, la llave **k**, y un apuntador a la posición en el nodo; y regresa el apuntador al nodo hoja donde debe estar **k**. Esta función opera de la siguiente forma: busca el índice i correspondiente a **k** en el nodo, una vez obtenido este valor, se pregunta si el nodo es una hoja, si lo és, entonces se coloca a i como valor de la posición y se regresa el apuntador de este nodo; en caso de ser una rama, entonces se regresa el valor de la misma función, pero ahora se inyecta el i-ésimo árbol-hijo como valor de entrada:

```
/* Busca la hoja y posicion donde se debe estar la llave k, */
/* regresa NULL si el arbol esta vacio                      */
/* [ABRIL-97]                                               */
/* AMILCAR MENESES VIVEROS                                  */
node *search_leaf(node *p, Bkey k, int *position)
{
   int i;

   if (p==NULL) {
      *position = -1;
      return NULL;
   }
   i = nodesearch(p, k);
   if (isleaf(p)==BTRUE) {
      *position = i;
      return p;
   }
   return (search_leaf(p->an[i], k, position));
}
```

De esta forma cuando deseamos buscar un elemento en el árbol llamamos a la función *search_leaf* y comparamos que el valor del índice sea menor que el orden del árbol —esto se debe a que el arreglo de llaves tiene un elemento menos que el orden del árbol— y comparamos a **k** con la i-ésima llave de la hoja encontrada.

```
void search(node *root, Bkey k)
{
   node *ns;
   Bregister *r;
   int i;
```

```
   if ((ns=search_leaf(root,k,&i))==NULL) {
        printf("\n El elemento"); printf(FFkey,k); printf("no se encontro");
   }
   else if ( i < (ORDER-1) && k == (ns->ak[i]) ) {
        r = ns->ap[i];
        printf("\n Llave  : "); printf(FFkey, PKeyNode(r));
        printf("\n Nombre : %s", r->nombre);
        printf("\n Edad   : %d", r->edad);
   }
   else printf("\n El elemento FKey no se encontro", k);

   printf("\n Presione cualquier tecla para continuar ...");
   getchar();
}
```

5 Recorrido Secuencial

Esta es la mayor ventaja que ofrece el *B+* sobre el árbol **B**, ya que además de ser un árbol, sus hojas se pueden manejar como una lista ligada. La forma de recorrer secuencialmente los registros en el *B+* es simple, primero buscamos la hoja más a la izquierda y después recorremos todos los elementos de cada hoja. Para esta tarea primero implantamos la rutina *first_leaf*, que recorre las ramas más a la izquierda del árbol hasta llegar a una hoja.

```
/* Encuentra la hoja mas a la izquierda */
/* [MAYO-97] */
/* AMILCAR MENESES VIVEROS */
node *first_leaft(node *n)
{
    if ( isleaf(n) == BTRUE ) return n;
    else return first_leaf(n->an[0]);
}
```

Una vez hallada la primera hoja se listan todos sus elementos en orden ascendente, y después saltar al siguiente nodo hoja al que hace referencia el apuntador *nxt*, y así sucesivamente hasta que algún nodo hoja tenga el valor de NULL en *nxt* (la hoja más a la derecha).

```
/* Lista todos los registros en forma secuencial */
/* [MAYO-97]  */
/* AMILCAR MENESES VIVEROS  */
void list_all(node *root)
{
   int  i, l, c;
   node *h;

   c=0;
   h = first_leaf(root);
   while( h != NULL ) {
```

```
      l = h->nKeys;                    /* Numero de elementos   */
      for(i=0; i<l; i++) {
         c++;                          /* Contador de elementos */
         list_element(h,i,c);          /* Despliega el registro */
      }
      h = h->nxt;                      /* Proximo nodo hoja     */
   }
}
```

6 Eliminación

La eliminación en un *B+* se realiza únicamente a nivel de nodo hoja, si la llave que se desea eliminar está en una rama, se conserva y se sigue hasta encontrar la llave y su respectivo registro en una hoja, esta operación se realiza con la función *search_leaf* explicada anteriormente. Una vez localizado el elemento (nodo hoja y posición) se llama a la rutina *erase_register* que libera la parte de memoria direccionada por el apuntador del arreglo de registros y recorre (o compacta) los elementos que se encuentran a la derecha del registro eliminado y decrementa el número de llaves en la hoja.

```
/* Borra el elemento "pos" de la hoja n */
/* [MAYO-97] */
/* AMILCAR MENESES VIVEROS */
void erase_register(node *n, int pos)
{
   int i;

   free(n->ap[pos]);
   for (i=pos; i< n->nKeys; i++) {
       n->ap[i] = n->ap[i+1];
       n->ak[i] = n->ak[i+1];
   }

   n->ap[i] = NULL;
   n->ak[i] = BKeyNull;
   (n->nKeys)--;
}
```

7 Inserción

Ya se ha mencionado que el *B+* es un árbol balanceado, cuando se le agregan elementos se debe mantener este balance, esto es, cuando un nodo se llena se divide en 2 partes, las llaves se distribuyen en mitades y el elemento medio se inserta como una llave más en el nodo padre. Si esto nodo no existe, entonces se crea un nuevo nodo, se inserta el elemento y se actualiza la dirección del nodo raíz. En caso de que el nodo si exista y esté lleno, se le aplica el mismo método de división.

El primer paso en la inserción de un registro es encontrar la hoja y posición donde debe quedar el nuevo registro —este proceso se realiza con la fución ya dicutida *search_leaf*— y verificamos el número de elementos en el nodo. Si el número de elementos es menor que el orden del árbol, entonces insertamos el registro con la función *node_register* que explicaremos más adelante. En

caso contrario, si el número de elementos es igual al orden del nodo, entonces dividimos el nodo e insertamos el nuevo registro en el nodo que le corresponda (en la división), conectamos las nuevas hojas, y actualizamos la raíz del árbol. La función queda de la siguiente forma:

```
/* Inserta en el arbol un nuevo registro */
/* [ABRIL-97]                            */
/* AMILCAR MENESES VIVEROS               */
node *insert_register(node *root, Bkey k, Bregister *r)
{
   int i, f;
   Bkey mkey;
   node *ph1, *ph2, *rf;

   rf = root;
   if ((ph1=search_leaf(root,k,&i))==NULL) {
      printf("\n Error!, no existe arbol\n"); exit(1);
   }

   if (ph1->nKeys < (ORDER-1)) insnoderegister(ph1,i,r,k);
   else {
      ph2 = divide(ph1, k, i, &f, &mkey);
      i = nodesearch(ph2, k);
      insnoderegister(ph2, i, r, k);
      conect(ph1, ph2);
      rf = search_root(ph1);
   }

   return rf;
}
```

Se aprecia que la función *insert_register* tiene 3 parámetros de entrada: el nodo raíz *root*, la llave **k** y el apuntador al registro correspondiente *r*. Además regresa el apuntador a la raíz del árbol, esto es porque al conservar el balance del árbol, la raíz puede puede cambiar si se altera la profundidad del *B+*. La función *insert_register* se auxilia de 5 rutinas: *insnoderegister, divide, nodesearch, concect* y *search_root* que explicamos a continuación, exepto *nodesearch* que ya se ha discutido previamente.

7.1 La función *insnoderegister*

Esta función recorre una posición a la derecha los elementos del nodo desde el índice *position*, agrega los nuevos valores e incrementa el número de elementos de este nodo.

```
/* Inserta un registro en una hoja */
/* [ABRIL-97]                      */
/* AMILCAR MENESES VIVEROS         */
void insnoderegister(node *p, int position, Bregister *r, Bkey k)
{
    int i;

    /* for(i=ORDER-1; i>position; i--) */
```

```
    for (i = p->nKeys; i > position; i--) {
        p->ap[i] = p->ap[i-1];
        p->ak[i] = p->ak[i-1];
    }
    p->ak[position] = k;
    p->ap[position] = r;
   (p->nKeys)++;
}
```

7.2 La función *divide*

Es la rutina principal en la construcción de la forma balanceada del *B+*. Esta rutina regresa el apuntador al nodo donde debe insertarse el nuevo elemento. Tiene 5 parámetros de entrada: el nodo a dividir n, la llave **k** que se desea insertar, la posición i de la llave en n, un apuntador a una bandera auxiliar fl, y la llave que se colocará en el nodo padre. La primera parte de esta rutina se encarga de dividir el nodo y decidir en cual de los dos nuevos nodos quedará **k**. La segunda parte se encarga de insertar la llave media en el nodo padre, considerando si existe nodo padre, si está lleno este nodo , o si quedan lugares en dicho nodo.

```
/* Genera el arbol dividiendo un nodo en 2 partes   */
/* [ABRIL-97]                                       */
/* AMILCAR MENESES VIVEROS                          */
node *divide(node *n, Bkey k, int i, int *fl, Bkey *mk)
{
    int aux, j;
    node *mid, *nr, *father, *mid2;
    Bkey midkey;

    /* Divide el nodo en "mitades" y decide  */
    /* que mitad debe contener  la llave 'k' */
    mid = makenode();
    if (i< (mdOr-1)) {
       copy(n, mdOr, ORDER, mid);
       nr = n;  *fl = 1; midkey = n->ak[mdOr-1];
    }
    else if ( i == (mdOr-1)) {
       copy(n, mdOr-1, ORDER, mid);
       nr = n; *fl = 1; midkey = k;
    }
    else {
       copy(n, mdOr, ORDER, mid);
       nr = mid; *fl = -1; midkey = n->ak[mdOr-1];
    }

    /* Revisa la insercion de los nuevos nodos hijos en el nodo padre */
    /* con su llave correspondiente                                   */
    father = n->up;
    if (father == NULL) {
```

```
        father = maketree(midkey);
        addnode(father, n, 0);
        addnode(father, mid, 1);
    }
    else if ( (father->nKeys) < (ORDER-1) ) {
         j = nodesearch(father, midkey);
         ins_key(father, midkey, j);
         addnode(father, mid, j+1);
    }
    else {
         j = nodesearch(father, midkey);
         mid2 = divide(father, midkey, j, &aux, mk);
         j = nodesearch(mid2, midkey);
         ins_key(father, midkey, j);
         addnode(mid2, mid, j+1);
    }
    *mk = midkey;
    return nr;
}
```

7.3 La función *concect*

Esta función liga las dos nuevas hojas para el recorrido secuencial. Esta función, a su vez, se hace un llamado dos funciones:*index_node* y *conectnode*.

La función *index_node* tiene 2 parámetros de entrada: p (nodo padre) y h (nodo hijo). Y tiene la tarea de regresar el índice de h en el arreglo correspondiente del arreglo nodos hijos de p. Esto es, determina si h es hijo de p y cual es el valor del índice. Si este valor es negativo, entonces h no es hijo directo de p.

```
/* Regresa el indice de un nodo relativo a su padre */
/* [SEPTIEMBRE-1997]                                */
/* AMILCAR MENESES VIVEROS                          */
int index_node(node *p, node *h)
{
  int i;
  for( i = p->nKeys; i>=0; i--)
     if ( (p->an[i])==h ) return i;
  return -1;
}
```

La función *conectnode* se encarga de establecer las ligas entre los dos nodos que tiene como parámetros de entrada.

```
/* Conecta secuencialmente 2 nodos  */
/* [ABRIL-97]                       */
/* AMILCAR MENESES VIVEROS          */
void conectnode(node *n1, node *n2)
{
    node *pnext;
```

```
    pnext = n1->nxt;
    n1->nxt = n2;
    n2->nxt = pnext;
}
```

La rutina conecta se encarga de establecer el orden secuencial de los nodos, dependiendo del valor que regrese *index_node*. Esto es, determina que nodo se conectará al final del primer nodo.

```
/* Conecta la hoja nd1 y nd2 en forma secuencial */
/* [ABRIL-97]  */
/* AMILCAR MENESES VIVEROS        */
void concect(node *nd1, node *nd2)
{
    node *ndup;
    int i1, i2;

    ndup = nd1->up;
    if ( (i1 = index_node(ndup, nd1)) < 0 ) {
        puts("Error en coneccion!!"); exit(1);
    }
    if ( (i2 = index_node(ndup, nd2)) < 0 ) {
        puts("Error en coneccion!!"); exit(1);
    }

    if (i1 < i2)      conectnode(nd1, nd2);
    else if (i1 > i2) conectnode(nd2, nd1);
    else { puts("Traslape de nodos!!!"); exit(1); }
}
```

7.4 La función *search_root*

La función se encarga de buscar el nodo raíz. Esta funcón sube un nivel en cada paso de recusrsión y se detiene cuando el apuntador *up* es nulo, es decir, cuando se encuentra en el nodo raíz.

```
/* Encuentra la raiz del nodo nd */
/* [MAYO-97]                     */
/* Amilcar Meneses Viveros       */
node *search_root(node *nd)
{
   if(nd->up == NULL) return nd;
   else return search_root(nd->up);
}
```

8 La Interfaz de Usuario

El programa se puede ejecutar desde el *shell* de una terminal UNIX. Al ejecutar el programa aparece una leyenda del programa y pide al usuario el nombre de el archivo que tiene la informacón que se va a manejar. Si el archivo especificado no existe, se tiene la opción de crear uno nuevo para iniciar una sesión de trabajo.

El programa se realiza las siguientes tareas:

Buscar Busca un registro en el árbol a traves de su campo llave.

Insertar Inserta un nuevo elemento en el árbol.

Eliminar Borra un elemento del árbol.

Listar Despliega todos los elementos del árbol en orden ascendente.

Guardar Guarda el contenido del árbol en un archivo.

Las tres primeras operaciones se manejan relativas a una llave en particular, es decir, una vez determinado el contenido del campo llave se procede a realizar la operación deseada. Las dos últimas opciones operan con todos los elemento del árbol[6].

Las operaciones pueden seleccionarse en el menú que aparece después de cargar al archivo de trabajo o realizar alguna operación.

```
        [B]     Busca
        [I]     Inserta
        [E]     Elimina
        [T]     Lista
        [G]     Guarda
        [F]     Fin
 Elige una opcion:
```

8.1 La opción *Busca*

Como se mencionó anteriormente, esta opción muestra el registro asociado con una llave. Primero se pregunta por el valor de la llave, cuando se presiona la tecla *enter* se procede a realizar la busqueda, si se encuentra esta llave muestra el valor del registro asociado. Esto es, cuando se selecciona esta opción, aparece el siguiente mensaje:

```
Llave (EDAD):
```

al dar el valor de la llave (edad) y si encuentra el valor muestra los valores correspondientes[7]:

```
Llave  : <llave>
Nombre : <nombre>
Edad   : <edad>
```

En caso de no encontrarse se muestra el siguiente mensaje

```
El elemento FKey no se encontro
```

8.2 La opción *Inserta*

Agrega un nuevo elemento al árbol, cuando se selcciona esta opción únicamente se piden los valores de los elementos del registro:

```
Nombre: <Nombre>

Edad: <Edad>
```

[6]Los elementos se consideran como las llaves que tienen asociado un apuntador a la estructura que compone el registro

[7]Las cadenas entre <> indican el valor del campo correspondiente

8.3 La opción *Elimina*

Esta opción borra un elemento del árbol[8]. Al entrar esta opción se pregunta por la llave que hace referencia al registro que se desea borrar del árbol:

```
Llave (EDAD): <llave>
```

Después se verifica si existe el registro, si es así, se procede a desplegarlo y verifica con el usuario la operación de borrado, es decir, confirma que este registro es el que se va ha borrar.

```
Llave  : <llave>
Nombre : <Nombre>
Edad   : <Edad>
Esta usted seguro (s/n)?
```

Si se confirma la operación aparece el siguiente letrero:

```
Registro Borrado
```

En caso contrario se despliega el menu principal.

8.4 La opción *Lista*

Esta opción despliega la lista de todos los registros almacenados en el árbol en orden ascendente.

```
1]      <Nombre>   <edad>      <Llave>
2]      <Nombre>   <edad>      <Llave>
           ...
           ...
           ...
n]      <Nombre>   <edad>      <Llave>
```

Y posteriormente despliega el menu principal.

8.5 La opción *Guarda*

Esta opción guarda en un archivo los registros almacenados en el $B+$ en orden asendente. Es muy similar a la opción lista, salvo que la salida la manda a un archivo.

```
Nombre del archivo de salida:<Nombre_archivo>
```

8.6 La opción *Fin*

Termina la ejecución del programa, libera la memoria ocupada por el $B+$. Las modificaciones al árbol —inserción y borrado de elementos— se perderán si no fueron salvadas anteriormente (opción *Guarda*).

[8]Recuerde que el elemento únicamente se da de baja en los nodos hoja, y se mantiene el valor de la llave en los nodos rama.

9 Observaciones

El funcionamiento de este *B+* puede mejorarse en varios aspectos, en esta sección se mencionan algunos problemas detectados, después de haber entregado el programa; y algunas consideraciones que pueden agregarse para un mejor uso del *B+*.

- Existe el arreglo de apuntadores, a la estructura de los registros, en los nodos "rama". En este tipo de nodo no tiene ningún sentido tener este arreglo, pues en un árbol de orden grande se desperdicia una gran cantidad de memoria. De hecho se podría considerar de colocar una estructura de datos que maneje memoria dinámica en lugar de tener un arreglo, evidentemente esto tendría un costo en el tiempo de ejecución.
- La definición de la llave puede realizarse en un *union* de varios tipos para evitar, al máximo posible, volver a compilar el programa cada vez que se desee cambiar la estructura.
- Intentar establecer una "meta estructura" para poder manejar distintas estructuras en los registro, similar a las que utilizan los manejadores de bases de datos.
- La interfaz de usuario puede ser mejorada en varios aspectos como el uso de herramientas para manejo en ambientes de ventanas e incorporación de nuevas opciones, entre otras cosas.

El código fuente puede falicitarse escribiendo al autor.

Referencias

[1] Aaron M. Tenenbaum, Yedidyah Langsam, Moshe A. Augesnstein; "Estrucuturas de Datos en C"; Ed. Prentice Hall Hispanoamericana; 1993.

[2] Henry F. Korth, Abraham Silberschatz; "Fundamentos de Bases de Datos"; Ed. MacGraw Hill Inc.; 1993.

[3] Robert Sedgewick; "Algorithms in C++"; Addison Wesley Publishing Company; 1992.

[4] Tomas H. Corben, Charles E. Leiserson & Ronald L. Rivest; "Introduction to Algorithms"; The MIT Press; Cambridge, Massachuset, 1992.

[5] Nicklaus Wirth; "Algorithms + Data Structures = Programs"; Prentice-Hall, Inc.; Englewood Cliffs, New Jersey, 1976.