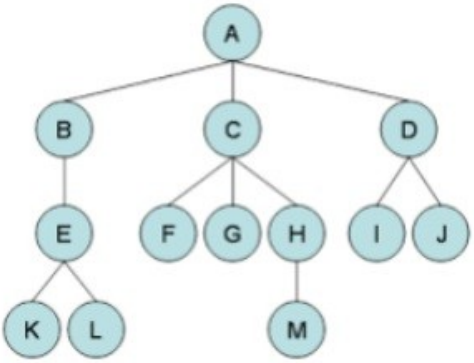
TP3 - Organización indexada – Arboles

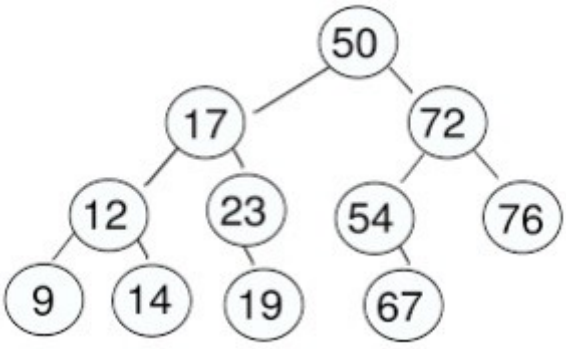
Alumno: Juan Cruz Mateos
Mat. 15134

1) Completar:



Raiz	A
Hermanos	(A), (B,C,D), (E), (F, G, H), (I, j), (K, L), (M)
Terminales	K, L, F, G, M, I, J
Interiores	B, E, C, H, D
Grado (*nro descendientes directos nodo)*	A:3, B:1, C:3, D:2, E:2, F:0, G:0, H:1,I:0, J:0, K:0, L:0, M:0
Grado del Arbol	3
Nivel (*nro nodos que deben ser recorridos desde la raiz para llegar a un determinado nodo)*	A:0, B:1, C:1, D:1, E:2, F:2, G:2, H:2,I:2, J:2, K:3, L:3, M:3
Altura (*maximo numero de niveles de entre todas las ramas del arbol +1)*	4

2) Completar y realizar recorridos : Preorden, Inorden, y Postorden



*Preorden: **50**, 17, 12, **9**, **14**, 23, **19**, 72, 54, **67**, **76***
*Inorden: **9,** 12, **14**, 17, 23, **19**, **50**, 54, **67**, 72, **76***
*Postorden: **9**, **14**, 12, **19**, 23, 17, **67**, 54, **76**, 72, **50***

¿Es estricto? ¿Es lleno? ¿Es completo?

- *Estricto: No. Para ser estricto si un subarbol esta vacio, entonces el otro tambien debe estarlo. Es decir, un nodo puede tener 0 o 2 hijos unicamente. Pero los nodos 23 y 54 poseen solo un hijo a derecha, por lo tanto no es estricto.*
- *Lleno: No. Para el nodo 72 la altura del subarbol derecho e izquierdo no coinciden.*
- *Completo: No. Si bien es un arbol lleno hasta el penultimo nivel, en el ultimo nivel sus nodos no estan agrupados a la izquierda.*

3) Diseñe la estructura para implementar un árbol binario que contenga como datos números enteros. Implemente funciones para agregar nodos, recorrer inorden y buscar un elemento en particular (en forma recursiva e iterativa).

```
typedef struct nodo {
    int clave;
    struct nodo *izq;
    struct nodo *der;
} nodo_t;

typedef nodo_t *arbol_t;

void preorden(arbol_t root) {
    if (root != NULL) {
        print("%d ", root->clave);
        preorden(root->izq);
        preorden(root->der);
    }
}

void inorden(arbol_t root) {
    if (root != NULL) {
        inorden(root->izq);
        printf("%d ", root->clave);
        inorden(root->der);
    }
}

void postorden(arbol_t root) {
    if (root != NULL) {
        postorden(root->izq);
        postorden(root->der);
        printf("%d ", root->clave);
    }
}

Arbol Binario:
void addNodo(arbol_t *root, int x) {
    *root = (arbol_t)malloc(sizeof(nodo_t));
    (*root)->clave = x;
    (*root)->izq = NULL;
    (*root)->der = NULL;
}

int esta(arbol_t root, int x) {
    if (root == NULL)
        return 0;
    else if (root->clave == x)
        return 1;
    else
        return estaroot->izq, x) || esta(root->der, x);
}
```

ABB:

```
void add_nodo_ABB(arbol_t *root, int x) {
    if (*root == NULL) {
        *root = (arbol_t)malloc(sizeof(nodo_t));
        (*root)->clave = x;
        (*root)->izq = NULL;
        (*root)->der = NULL;
    } else {
        if (x < (*root)->clave)
            add_nodo_ABB(&(*root)->izq, x);
        else
            add_nodo_ABB(&(*root)->der, x);
    }
}

int esta_ABB_rec(arbol_t root, int x) {
    if (root == NULL)
        return 0;
    else if (root->clave == x)
        return 1;
    else if (x < root->clave)
        return esta_ABB_rec(root->izq, x);
    else
        return esta_ABB_rec(root->der, x);
}

int esta_ABB_it(arbol_t root, int x) {
    while (root != NULL && root->clave != x) {
        root = x < root->clave ? root->izq : root->der;
    }
    return root != NULL;
}
```

4) Realice las rutinas necesarias para agregar 100.000 elementos al azar en el árbol.

```
int main(int argc, char const *argv[]) {
    arbol_t root = NULL;

    srand(time(NULL));
    for (int i = 0; i < 100000; i++) {
        add_nodo_ABB(&root, rand() % 100000);
    }
    inorden(root);
    return 0;
}
```

5) Busque un elemento cualquiera (conceptualmente) ¿Cuántas comparaciones fueron necesarias en promedio?

Cantidad de comparaciones = log(N)

6) Plantee (implemente) una solución de persistencia para su árbol. ¿Con qué problemas se encontró?

```
void persistir_txt(arbol_t root, int nivel, FILE *file) {
    if (root != NULL) {
        if (root->izq != NULL && root->der != NULL)
               fprintf(file, "clave=%d\tnivel=%d\tizq=%d\tder=%d\n", root->clave, nivel,
root->izq->clave, root->der->clave);
        else if (root->izq != NULL && root->der == NULL)
              fprintf(file, "clave=%d\tnivel=%d\tizq=%d\tder=null\n", root->clave, nivel,
root->izq->clave);
        else if (root->izq == NULL && root->der != NULL)
              fprintf(file, "clave=%d\tnivel=%d\tizq=null\tder=%d\n", root->clave, nivel,
root->der->clave);
        else
                  fprintf(file,  "clave=%d\tnivel=%d\tizq=null\tder=null\n",  root->clave,
nivel);
        persistir_txt(root->izq, nivel + 1, file);
        persistir_txt(root->der, nivel + 1, file);
    }
}

void persistir_bin(arbol_t root, int nivel, FILE *file) {
    nodo_persist_t persist;

    if (root != NULL) {
        persist.clave = root->clave;
        persist.nivel = nivel;
        if (root->izq != NULL && root->der != NULL) {
            persist.izq = root->izq->clave;
            persist.der = root->der->clave;
        } else if (root->izq != NULL && root->der == NULL) {
            persist.izq = root->izq->clave;
            persist.der = -1;
        } else if (root->izq == NULL && root->der != NULL) {
            persist.izq = -1;
            persist.der = root->der->clave;
        } else {
            persist.izq = -1;
            persist.der = -1;
        }
        fwrite(&persist, sizeof(nodo_persist_t), 1, file);
        persistir_bin(root->izq, nivel + 1, file);
        persistir_bin(root->der, nivel + 1, file);
    }
}
```

```
typedef struct nodo_persist {
   int clave;
   int nivel;
   int izq;
   int der;
} nodo_persist_t;
```

Problema: Al despersistir el arbol, es dificil reconstruirlo con la misma estructura, obteniendose un arbol con las mismas claves pero diferente estructura general que el persistido.

7) Realice los puntos 3, 4, 5 y 6 pero utilizando montículos.

7.3)
```
#define MAX 50

typedef struct heap {
    int *datos;
    int ult;
} heap;

typedef heap *heap_t;

heap_t new_heap() {
    heap_t h;

    h = (heap_t)malloc(sizeof(heap));
    if (h == NULL)
        return NULL;
    h->datos = (int *)(malloc(sizeof(int) * MAX));
    if (h->datos == NULL)
        return NULL;
    h->ult = -1;
    return h;
}

int padre(int i) {
    return (i - 1) / 2;
}

int izq(int i) {
    return 2 * i + 1;
}

int der(int i) {
    return 2 * i + 2;
}

void push(heap_t heap, int x) {
    int i, temp;

    if (heap->ult != MAX - 1) {
        heap->datos[++heap->ult] = x;
        i = heap->ult;
        // percolate_up(heap, heap->ult);
        while (heap->datos[padre(i)] > heap->datos[i]) {
            temp = heap->datos[padre(i)];
            heap->datos[padre(i)] = heap->datos[i];
            heap->datos[i] = temp;
            i = padre(i);
        }
    }
}

void percolate_up(heap_t heap, int i) {
    int temp;

    // padre(0) == 0
    if (i != 0 && heap->datos[padre(i)] > heap->datos[i]) {
        temp = heap->datos[padre(i)];
        heap->datos[padre(i)] = heap->datos[i];
        heap->datos[i] = temp;
```

```
        percolate_up(heap, padre(i));
    }
}

int peek(heap_t heap) {
    if (heap->ult != -1) {
        return heap->datos[heap->ult];
    } else {
        return __INT_MAX__;
    }
}

int pop(heap_t heap) {
    int min = __INT_MAX__;

    if (heap->ult != -1) {
        min = heap->datos[0];
        heap->datos[0] = heap->datos[heap->ult];
        heap->ult -= 1;
        percolate_down(heap, 0);
    }
    return min;
}

void percolate_down(heap_t heap, int i) {
    int min, temp;

    if (i < heap->ult) {
        min = i;
        if (izq(i) <= heap->ult && heap->datos[izq(i)] < heap->datos[min])
            min = izq(i);
        if (der(i) <= heap->ult && heap->datos[der(i)] < heap->datos[min])
            min = der(i);
        if (min != i) {
            temp = heap->datos[min];
            heap->datos[min] = heap->datos[i];
            heap->datos[i] = temp;
            percolate_down(heap, min);
        }
    }
}

int empty(heap_t heap) {
    return heap->ult == -1;
}

void preorden(heap_t heap, int pos) {
    if (pos <= heap->ult) {
        printf("%d ", heap->datos[pos]);
        preorden(heap, izq(pos));
        preorden(heap, der(pos));
    }
}

void inorden(heap_t heap, int pos) {
    if (pos <= heap->ult) {
        inorden(heap, izq(pos));
        printf("%d ", heap->datos[pos]);
        inorden(heap, der(pos));
    }
}
```

```
void postorden(heap_t heap, int pos) {
    if (pos <= heap->ult) {
        postorden(heap, izq(pos));
        postorden(heap, der(pos));
        printf("%d ", heap->datos[pos]);
    }
}

void print_heap(heap_t heap) {
    int i;

    printf("| ");
    for (i = 0; i <= heap->ult; i++) {
        printf("%d ", heap->datos[i]);
    }
    printf("|\n");
}

void free_heap(heap_t heap) {
    free(heap->datos);
    free(heap);
}

int esta_it(heap_t heap, int x) {
    int i;

    if (heap->ult == -1 || x < heap->datos[0])
        return 0;
    else {
        i = 0;
        while (i <= heap->ult && heap->datos[i] != x) {
            i += 1;
        }
        return i <= heap->ult;
    }
}

int esta_rec(heap_t heap, int pos, int x) {
    if (pos > heap->ult)
        return 0;
    else if (x < heap->datos[pos])
        return 0;
    else if (x == heap->datos[pos])
        return 1;
    else
        return esta_rec(heap, izq(pos), x) || esta_rec(heap, der(pos), x);
}

7.4)
int main(int argc, char const *argv[]) {
    heap_t heap;

    heap = new_heap();
    srand((unsigned int)time(NULL));
    for (int i = 0; i < 100000; i++) {
        push(heap, rand() % 100000);
    }
    inorden(heap, 0);
    free_heap(heap);
    return 0;
}
```

7.5) Dada la implementacion en forma de arreglo, y cumpliendo la propiedad de heap, la operacion de busqueda no es muy eficiente ya que se comporta como un vector desordenado. Luego, la compledidad del algorimo resulta O(n).

7.6)

```
void persistir(heap_t heap, char *filename) {
    FILE *file = fopen(filename, "wb");
    int i = 0;

    while (i <= heap->ult) {
        fwrite(&(heap->datos[i]), sizeof(int), 1, file);
        i++;
    }
    fclose(file);
}

void despersistir(heap_t heap, char *filename) {
    FILE *file = fopen(filename, "rb");
    int i, clave;

    if (file != NULL) {
        i = -1;
        while (fread(&clave, sizeof(int), 1, file) == 1) {
            heap->datos[++i] = clave;
        }
        heap->ult = i;
        fclose(file);
    }
}
```

Dado que el heap se implementa como un arreglo, es posible persistir en un arhicvo de registros cada uno de los elementos del heap en el orden en que se encurntran en el arreglo y luego despersistirlo sin problemas obteniendose el mismo orden.

ÁRBOLES B, B* y B+

8) Dada la secuencia de claves enteras: 190, 57, 89, 90, 121, 170, 35, 48, 91, 22, 126, 132 y 80;dibuje el árbol B de orden 2 (2d+1 punteros) cuya raíz es R, que se corresponde con dichas claves.

9) En el árbol R del problema anterior,elimine la clave 91 y dibuje el árbol resultante. Elimine ahora la clave 48. Dibuje el árbol resultante, ¿se redujo el número de nodos?

10) Dada la siguiente secuencia de claves: 7, 25, 27, 15, 23, 19, 14, 29, 10, 50, 18, 22, 46, 17, 70, 33 y 58 dibuje el árbol B+ cuya raíz es R, que se corresponde con dichas claves.

11) Construir cada uno de los arboles B (B, B+ y B*) que se van generando conforme se van insertando los números 1, 9, 32, 3, 53, 43, 44, 57, 67, 7, 45, 34, 12, 23, 56, 73, 65, 49, 85, 89, 64, 54, 75, 77 en un árbol B de orden 5.