Examen Final:

A. (1.5 pts) ¿En qué **casos conviene** los siguientes algoritmos de búsqueda en grafos?

A*:_ _

Dijsktra: _

Floyd Warshall: _

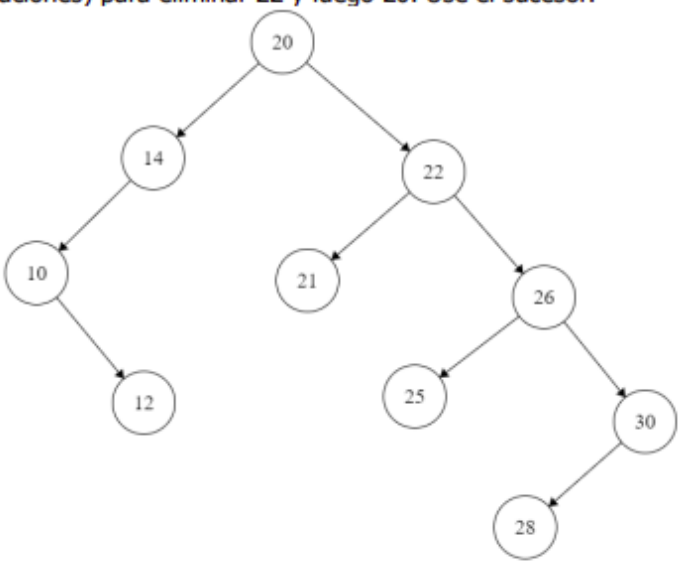
A*Conviene cuando se quiere realizar una búsqueda rápida en base a la heurística y el peso real de las aristas, para hallar el camino más corto desde un vértice hacia otro.
Dijkstra: Conviene cuando se quiere buscar el camino más corto desde un vértice hacia los demás, de manera exacta.
Floyd Warshal: Conviene cuando se quiere saber todos los caminos cortos desde un vértice hacia los demás, lo cual cumple para cada vértice.

B. (1.5 pts) Analice las ventajas y desventajas del Splay Tree frente al Self Organizing List (Move-to-Front method).

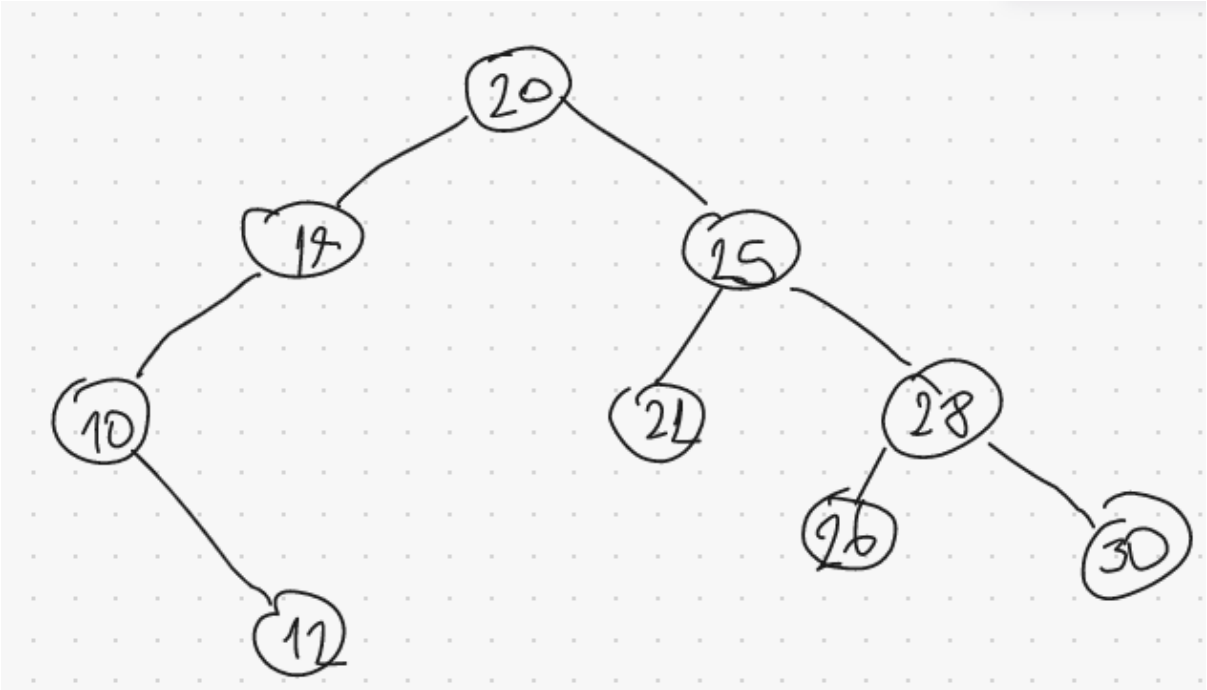
El splay tree permite tener una estructura de un árbol, lo cual provoca que sus operaciones básicas sean de complejidad O(log(n)), además de que siempre se mantendrá así, ya que este árbol es auto-balanceable.
Por otro lado, Self Organizing List consume menos memoria al momento de crearse, y mantenerse. Aparte de que mover el elemento accedido al principio es más sencillo de hacer. Sin embargo, hay casos en los cuales mover el elemento buscado al principio resulta ineficiente. En un self organizing list tomarán operaciones, mientras que para el Splay tree log(n)

Pregunta 2 (9 puntos): Ejercicios

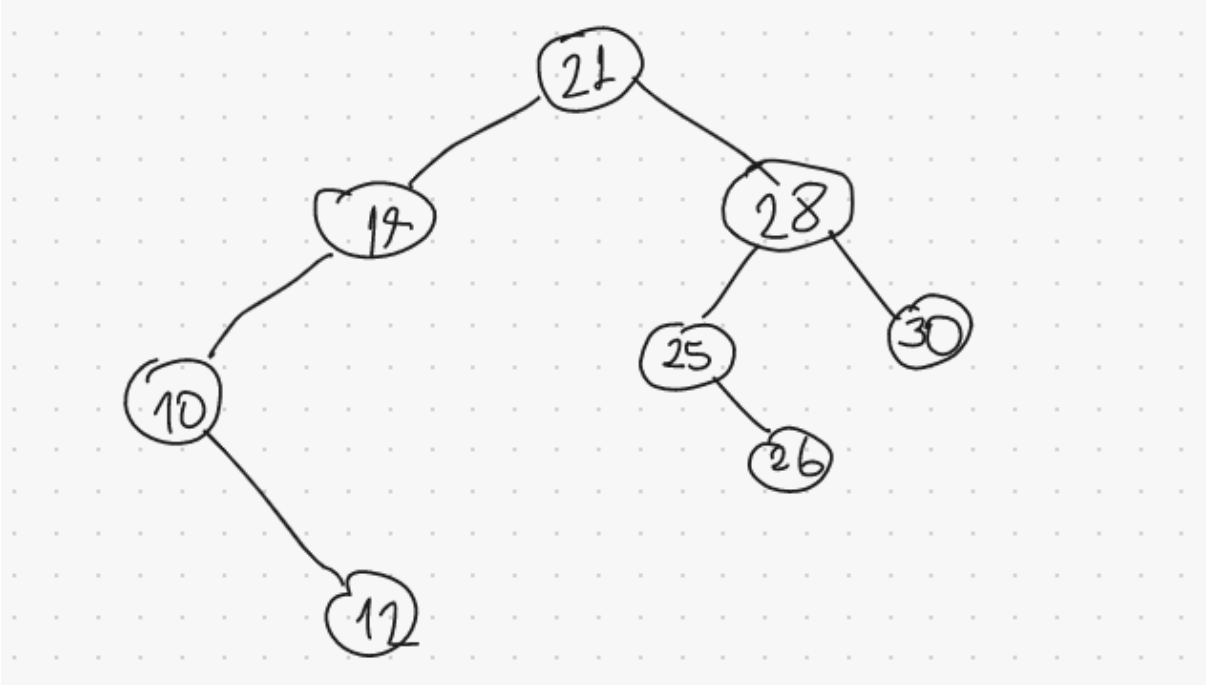
A. (2 pts) AVL: dado el siguiente BST no balanceado, se le pide aplicar el algoritmo de **eliminación del AVL** (con rotaciones) para eliminar 22 y luego 20. Use el sucesor.



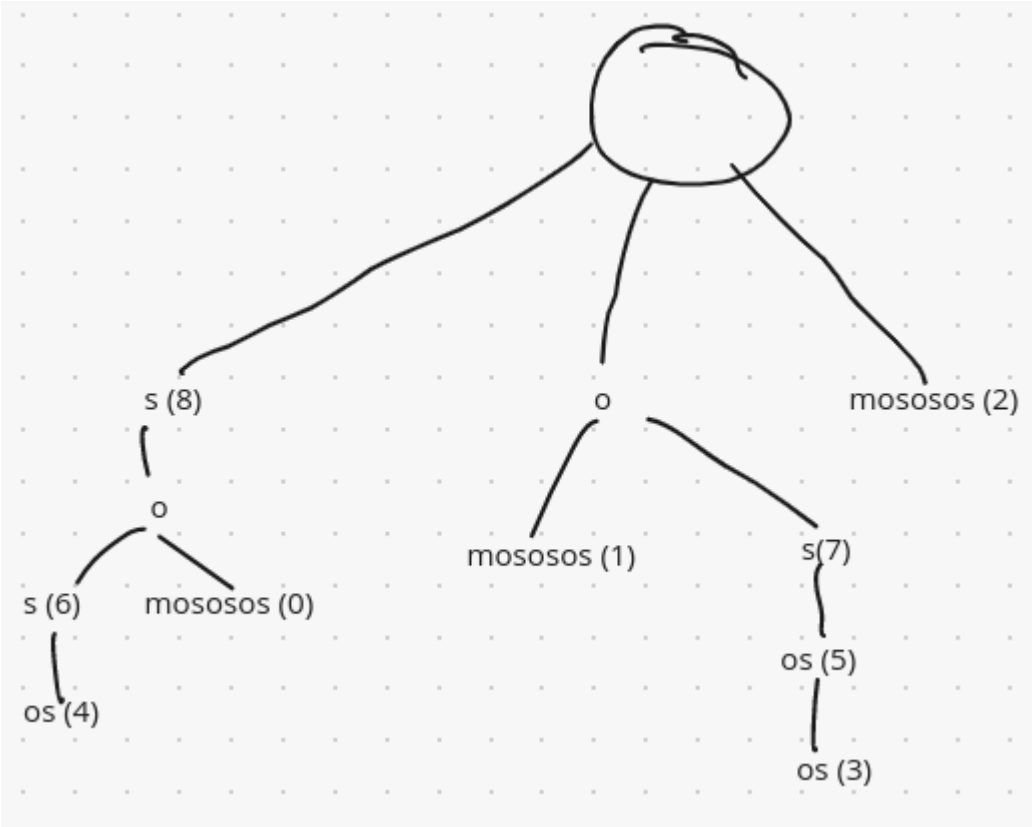
Eliminando el 22:



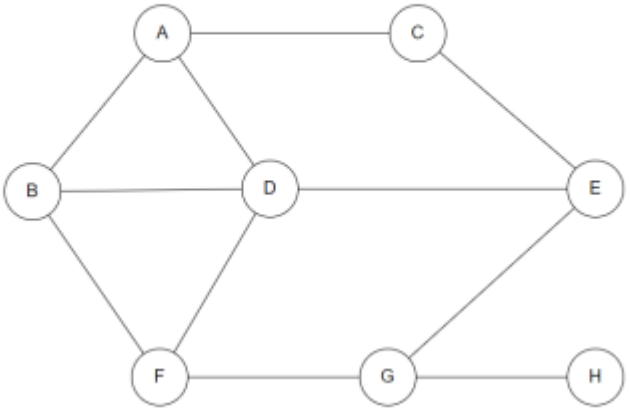
Eliminando el 20:

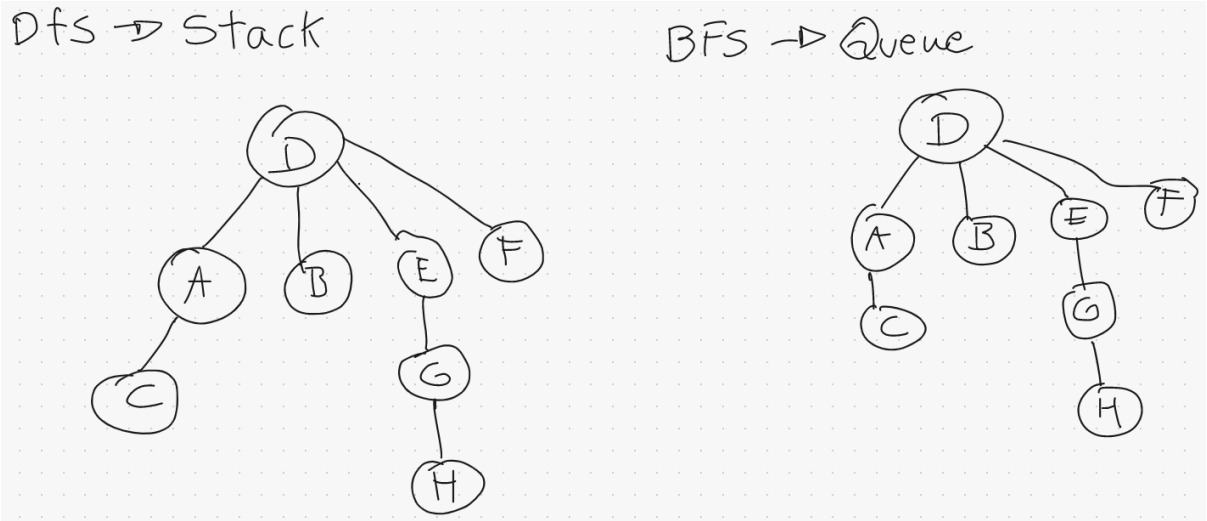


C. (2 pts) Suffix Tree: dado el texto "somososos", construir el árbol de sufijos y luego señalar el camino del string matching para "oso".



D. (2 pts) Grafo: mostrar el árbol DFS y BFS siguiendo el **orden lexicográfico** de los nodos.





Implementación:

A. (3 puntos) Patricia Tree: se le pide diseñar el seudocódigo lo más detallado posible para eliminar una palabra del Árbol Patricia. Indique la complejidad algorítmica en el código.

```
def remove(root, key,&itr):
    string temp = ""
    for(int i = 0; i < key.size();i+=1){ #Iteración de las palabras para la búsqueda
            temp+=key[i] #los posibles valores de key
            for(int j = 0; j < nodo->children.size();j+=1){
                if(nodo != nullptr and nodo->children[j] == temp){ #Si es que existe una llave, sigo
por esa ruta
                    key-=temp #eliminar el subfijo de temp a la palabra key
                    remove(root->children[j],key,itr+1)
                };
            }
    }
    #Complejidad: O(k*h*y) siendo k el tamaño de la palabra, y la altura del árbol, h la
cantidad de hijos, cabe resaltar que nodo->children es un mapa, para así ahorrarnos los
problemas al momento de verificar si la ruta es nullptr
    #Se podría afirmar que k podria tomar un valor pequeño, pues en cada iteración va
disminuyendo su tamaño considerablemente
    if(nodo!= nullptr and !nodo->endWord and itr != -1){ #Una vez terminado verificó si antes
no se ha eliminado
            itr = -1;
            return;
    }
    if(itr !=-1){
        int  cnt = 0;
        for(int j = 0; j < nodo->children.size();j+=1){
            if(nodo->children[j] != nullptr) cnt+=1;
        }
        if(!cnt){ #Si solo hay una rama, la elimino
            delete nodo;
```

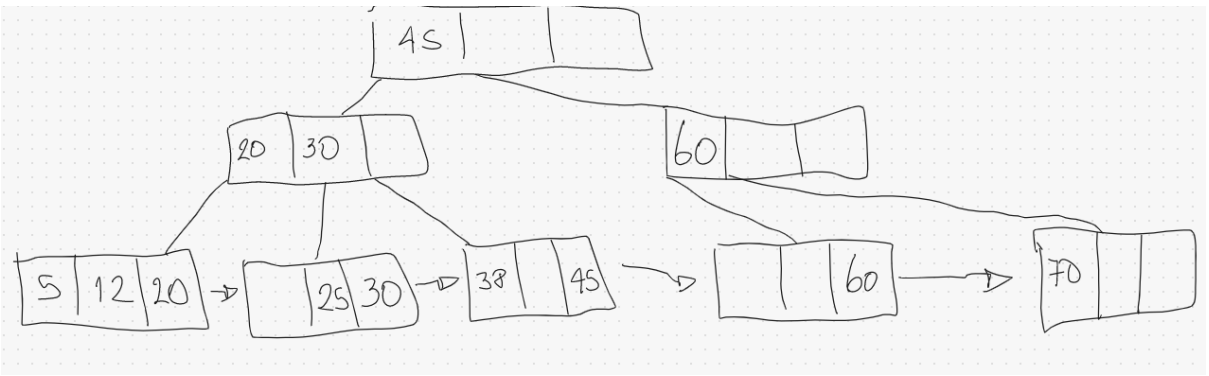
```
            nodo = nullptr;
        }else if(i!=-1){ #Si hay varias ramas, termino el proceso de eliminación
            nodo->endWord = false;
            i = -1;
        }
    }

if root == nullptr:
    print("No existe un arbol")
elif key == '':
    print("No existe la llave")
else:
            itr = 0
    remove(root,key,itr)
```

B. (5 puntos) Sparse Matrix: se le pide diseñar el algoritmo en C++ para devolver la transpuesta de una matriz lo más eficiente posible.

```
struct SparseMatrix {
    struct Node {
        T data;
        int pos_row;
        int pos_col;
        Node<T>* next_row;
        Node<T>* next_col;
    };
    vector<Node<T>*> rows;
    vector<Node<T>*> cols;
    int n_cols;
    int n_rows;
};
//devolver una nueva matriz como la transpuesta de la original
SparseMatrix transpose(SparseMatrix original);
```

```
#include <iostream>
#include <vector>

using namespace std;

template <class T>
class SparseMatrix {
private:
    struct Node {
    public:
        T data;
        int pos_row;
        int pos_col;
        Node* next_row;
        Node* next_col;
```

```
        Node(){
            pos_col=pos_row=-1;
            next_col = next_row = nullptr;
        }
        Node(int i, int j, T value){
            pos_row = i;
            pos_col = j;
            data = value;
            next_col = next_row = nullptr;
        }
    };

    vector<Node*> rows;
    vector<Node*> cols;
    int n_cols;
    int n_rows;
public:
    SparseMatrix(int nr, int nc) {
        this->n_cols = nc;
        this->n_rows = nr;
        for(int i=0;i<nr;i+=1)
            rows.push_back(nullptr);
        for(int i=0;i<nc;i+=1)
            cols.push_back(nullptr);
    }


    SparseMatrix transpose(SparseMatrix<T> & otro){
        SparseMatrix<T> ans(otro.n_cols,otro.n_rows);
        vector<Node*> temporales(otro.n_rows);
        for(int i = 0; i < otro.n_rows;i++){
            temporales[i] = ans.rows[i];
        }
        for(int i = 0; i < otro.n_cols;i++){
            ans.rows[i].push_back(otro.cols[i]); //Por simplicidad el metodo push_back de la lista.
            temporales[i].push_back(otro.rows[i]);
        }
        for(int i = 0; i < otro.n_rows;i++){
            ans.cols[i] = temporales[i];
        }
        return ans;
    }
};
B+tree
```


45
20
30
60
5
12
20
25
30
38
45
60
70