TD 1

# Exo 1

recherche.liste(l,k) O(n)

x = l.tete

TANTQUE x != NULL ET x.clé != k

x = x.succ

Retourner x

inserer.liste(l,k) O(1)

x.secc = l.tete

SI l.tete != NULL

l.tete.pred = x

l.tet = x

x.pred = NULL

supprimer.liste(l,x) O(1)

SI x.pred != NULL

x.pred.succ = x.succ

SINON

l.tete = x.succ

SI x.succ != NULL

x.succ.pred = x.pred

# Exo 2

3.a) O(S+A)

3.b) O(S²)

4.a) O(degré(u))

4.b) O(S)

5.a) O(degré(u))

5.b) O(1)

# Exo 3

2)

cellule.h :

typedef struct cellule\_type{

    int id\_sommet;

    struct cellule\_type \*pred;

    struct cellule\_type \*succ;

}cellule\_t;

int initialiser\_cellule(cellule\_t\* c, int id\_sommet);

cellule.c :

int initialiser\_cellule(cellule\_t\* c, int id\_sommet){

    ...

    return 0;

}

liste.h :

typedef struct{

    cellule\_t\* tete;

}liste\_t;

int initialiser\_liste(liste\_t\* l);

int inserer(liste\_t\* l, cellule\_t\* c);

...

liste.c :

progListe.c :

int main(...){

    cellule\_t cl;

    initialiser\_cellule(&cl,4);

    cellule\_t\* iz;

    iz=(cellule\_t\*) malloc(sizeof(cellule\_t));

    initialiser\_cellule(iz,4);

}

graphe.h :

typedef struct{

    int n\_sommets;

    int oriente;

    int value;

    liste\_t\* l\_adj;

    int \*\* m\_adj;

}graphe\_t;

int initialiser\_graphe(graphe\_t\* g, char \* nom\_fichier);

int initialiser\_graphe(graphe\_t \* g, char \* nom\_fichier){

    FILE \* fichier;

    char ch\_temp[30];

    fichier = fopen(nom\_fichier, "r"); //lecture

    fscanf(fichier, "%s", ch\_temp);

    fscanf(fichier, "%d", g->n\_sommets)

    g->m\_stockage = (int\*) malloc(sizeof(int)\*g->n\_sommets\*g->n\_sommets);

    g->m\_adj = (int\*\*)malloc(sizeof(int\*)\*g->n\_sommets);

    f(i...){

        g->m\_adj[i]=&g->ù\_stockage[i\*g->n\_sommet];

    }

}

TD 2

**Attention,** certains NULL peuvent être des NIL jusqu'à exo 3.3, exclu

# Exercice 1

1. Schéma

pile\_vide(p)    O(1)

  si p.sommet == 0

    retourner vrai

  sinon

    retourner faux

empiler(p,x)    O(1)

  p.sommet = p.sommet +1

  p[p.sommet] = x

depiler(p)     O(1)

  si pile\_vide(p)

    erreur("débordement négatif")

  sinon

    p.sommet = p.sommet-1

    retourner p[p.sommet+1]

# Exercice 2

1. Schéma

enfiler(f,x) O(1)

  f[f.queue] = x

  si f.queue == f.longueur

    f.queue = 1

  sinon

    f.queue = f.queue +1

defiler(f) O(1)

  x = f[f.tete]

  si f.tete == f.longueur

    f.tete = 1

  sinon

    f.tete = f.tete +1

  retourner x

# Exercice 3

1. Schéma

parcours\_largeur(G,s) O(S+A)

  Pour chaque sommet u appartenant à G.s

    u.couleur = BLANC

    u.d = infinie (max int)

    u.pere = NULL

  s.couleur = GRIS

  s.d = 0

  s.pere = NULL

  f = NULL

  enfiler(f,s)

  Tant que f != NULL

    u = defiler(f)

    Pour chaque sommet v appartenant à G.adj[u]

      si v.couleur == BLANC

        v.couleur = GRIS

        v.d = u.d +1

        v.pere = u

        enfiler(f.v)

    u.couleur = NOIR

intialisation : O(S)

enfiler/defiler : O(S)

balayage des listes d'adjacences : O(A)

* O(S+A)



afficher\_chemin(G,s,v)  O(S) //Graphe G, Sommet départ S, Sommet D'arrivé V

  si v == s

    afficher s

  sinon

    si u.pere == NIL

      afficher "aucun chemin de s à v"

    sinon

      afficher\_chemin(G,s,v.pere)

      afficher v

# Exercice 4

Schéma

parcours\_profondeur(G)  O(S+A)

  pour chaque sommet u appartenant à G.s

    u.couleur = BLANC

    u.pere = NIL

  date = 0

  pour chaque sommet u appartenant à G.s

    s u.couleur == BLANC

    visiter.pp(G,u)

visiter\_pp(G,u)

  date = date +1

  u.d = date

  u.couleur == GRIS

  pour chaque sommet v appartenant à G.adj[u]

    si v.couleur == BLANC

      v.pere = u

      visiter\_pp(G,v)

  u.couleur = NOIR

  date = date+1

  u.f = date

parcours-profondeur-ieratif (G)

  pour chaque sommet u appartenant à G.S

    u.couleur = BLANC

    u.pere = NIL

  p = vide

  pour chaque sommet u appartenant à G.S

    si u.couleur = BLANC

      u.couleur = GRIS

      empiler(p,u)

      tant que p != vide

        v = depiler(p)

        afficher v

        pour chaque sommet w appartenant à G.adj[v]

          si w.couleur = BLANC

            w.couleur = GRIS

            w.pere = v

            empiler(p,w)

        v.couleur = NOIR

# exercice 5

tri\_topologique(G)    O(S+A)

  appeler parcours\_profondeur(G)

  calculer les dates de fin de traitement vf pour chaque sommet v

  chaque fois que le traitement d un sommet se termine l insérer au début d une liste chainée

  retourner la liste chainée des sommets

TD 3

# Exercice 1

construire\_tas\_max(t)   O(n) //t = tableau

  t.taille = t.longueur

  pour i = (division entière)t.longueur/2 decr jusqu à 1

    entasser.max(t,i)

parent(i) //i = indice

  retourner (division entière)i/2

gauche

  retourner i\*2

droite

  retourner i\*2+1

entasser\_max(t,i)

  g=gauche(i)

  d=droite(i)

  si g<=t.taille && t[g]>t[i]

    max=g

  sinon

    max = i

  si d<= t.taille && t[d]>t[max]

    max=d

  si max != i

    echanger t[i] && t[max]

    entasser\_max[t,max]

tri\_par\_tas(t)    O(n)

  construire\_tas\_max(t)

  pour i =t.longueur decr jusqu à 2

    echanger t[1] et t[i]

    t.taille = t.taille-1

    entasser\_max(t,1)

maximum\_tas(t)    O(1)

  retourne t[1]

extraire\_max\_tas(t)   O(log n)

  si t.taille <1

    erreur "limite inférieure dépassée"

  max = t[1]

  t[1] = t[t.taille]

  t.taille = t.taille -1

  entasser\_max(t, 1)

  retourner max

augmenter\_cle\_tas(t, i, clé)

  si clé < t[i]

    erreur "nouvelle clé plus petite que l'ancienne"

  t[i] = clé

  tant que i>1 && t[parant(i)]<t[i]

    echange t[i] et t[parent(i)]

    i = parent(i)

inserer\_tas\_max(t, clé)

  t.taille = t.taille +1

  t[t.taille] = - infiny

  augementer\_clé\_tas (t, t.taille, clé)

# exercice 4

composantes\_connexes  (G)

  pour chaque sommet v appartenant à G.S

    créer\_ensemble(v)

  pour chaque arrete (u,v) appartenant à G.A

    si trouver\_ensemble(u)!= trouver\_ensemble(v)

      union (u,v)

meme\_composante(u,v)

  si trouver\_ensemble(u)==trouver\_ensemble(v)

    retourner vrai

  sinon

    retourner faux

creer\_ensemble(x)   O(1)

  creer une nouvelle liste chainée dont l objet est x

trouver\_ensemble(x) O(1)

  retourner x.ensemble.tete

union(u,v)    O(n)

  x.ensemble.queue.succ = y.ensemble.tete

  x.ensemble.queue = y.esemble

  pour chque objet z appartenant à y.ensemble

    z.ensemble = x.ensemble

# Exercice 5

Tri des arrêtes par ordre croissant :

(h;g) = 1

(g;f) = 2

…

(b;h) = 11

(d;f) = 14

E={(h,g);(g,f);(i,c);(a,b);(c,f);(c,d);(b,c);(d,e)}

acpm\_kruskal\_tableau(G)     tri des arrêtes : O(A log(A)) ; init CC : O(S) ; boucle pour chaque arrête : O(A) \* mise à jour cc O(S)

                            => O(A\*S)

    trier les arrêtes de G.A par ordre croissant de poids

    pour i = 1 à |S|

        cc [i] = i

    E=vide

    Pour chaque arrête (u,v) appartenant à G.A prise par ordre croissant de poids

        si cc[u]! cc[v]

            E=EU{(u,v)}

            icc=cc[v]

            pour i =1 allant à |S|

                si cc[i] == icc

                    cc[i] = cc[u]

    retourner E

acpm\_kruskal\_ensemble(G)    pour chaque sommet : O(S) + tri des aretes O(A log A) + pour chaque arete O(A) + opérations ensembles O(A) O(log A)

                            => O(A log S)

    E = vide

    pour chaque sommet v appartenant à G.S

        créer\_ensemble(v)

    trier les arêtes de G.A par ordre croissant de poids

    pour chaque arete (u,v) appartenant à G.A prise par ordre croissant de poids

        si trouver\_ensemble(u) != trouver\_ensemble(v)

            E=EU{(u,v)}

            union (u,v)

    retourner E

Exercice 6

acpm\_prim\_tableau(G,p,r) //graphe, poids de l'arrette(u,v)

    //O(s)  + O(s²) + O(a)

    //=> O(s²)

    pour i =1 à |s|

        d[i] = infiny

        pere[i] = NIL

        couvert[i] = false

    d[r]=0

    pour i=1 à |s|-1

        u=min(d,couvert)

        couvert[u] = true

        pour chaque sommet v appartenant à G.adj[u]

            si non couvert et p(u,v)<d[v]

                pere[u] = u

                d[v] = p(u,v)

acpm\_prim\_file\_priorites(G,p,r) //O(A lg S)

pour chaque sommet u appartenant à G.S

    u.clé = infini

    u.pere = NIL

r.clé = 0

f = G.S

tant que f\*=vide

    u = extraire\_min(f)

    pour chaque sommet v apaprtenant à G.adj[v] //pas sûr du G.adj[v]

        so v appartient à f et p(u,v)<u.clé

            v.pere =u

            v.cle = p(u,v)

TD4

Exo 1

Exo 2

transpose\_liste(G) //O(s) + O(s) + O(A) = O(S+A)

Gt.s = G.s

    pour chaque sommet x appartenant à G.s

        creer la liste Gt.adj[x]

    pour chaque sommet x appartenant à G.s

        pour chaque spùùet y appartenant à G.adj[x]

            inserer(Gt.adj[Y],X)

    retourner Gt

transpose\_matrice(G)

    Gt.s = G.s

    pour i =1 à |S|

        pour j = 1 à |S|

            Gt.madj[j][i] = G.madj[i][j]

    retourner Gt

exo 3

exo 4

exo 5

TD 5

Exo1

rechercher\_adressage\_direct(t,k) //t=table ; k=valeur

    //O(1)

    retourner t[k]

inserer\_adressage\_direct(t, x) //t=table; x=pointeur sur la valeur

    //O(1)

    t[x.cle]=x

supprimier\_adressage\_direct(t, x)

    //O(1)

    t[x.cle]=NIL

exo 2

inserer\_hachage\_chainee(t, x)

    //O(1)

    inserer x en tete de la liste t[h(x.cle)]

supprimer\_hachage\_chaine(t, x)

    //O(1)

    supprimer x de la liste t[h(x.cle)]

rechercher\_hachage\_chainee(t, k)

    //O(n)

    rechercher un élément de clé k dans la liste t[h(k)]

TD 6

Exo1

afficher\_arbre\_recursif(x)

    //O(n)

    si x != NIL

        afficher x.cle

        afficher\_arbre\_recursif(x.gauche)

        afficher\_arbre\_recursif(x.droite)

afficher\_arbre\_iteratif(x)

    si x != NIL

        empiler(p,x)

        tant que non pile\_vide(p)

            x = depiler(p)

            afficher(x.cle)

            si x.droite != NIL

                empiler(p,x.droite)

            si x.gauche != NIL

                empiler(p,x.gauche)

exo 2

afficher\_arbre\_trié(a)

    parcours\_infixe\_recursif(a.racine)

rechercher\_arbre\_recurssif(x,k) //noeud, clé

    si x == NIL OU K == x.clé

        retourner x

    si k < x.clé

        retourner rechercher\_arbre\_recurssif(x.gauche)

    sinon

        retourner rechercher\_arbre\_recurssif(x.droite)

rechercher\_arbre\_iteratif(x, k)

    tant que x != NIL ET k != x.clé

    si k<x.cle

        x=x.gauche

    sinon

        x=x.droite

minimum\_arbre(x)

    //O(h)

    tant que x.gauche != NIL

        x= x.gauche

    retourner x

//maximum x.droite O(h)

successeur\_arbre(x) //O(h)

    si x.droite != NIL

        retourner minimum\_arbre(x.droite)

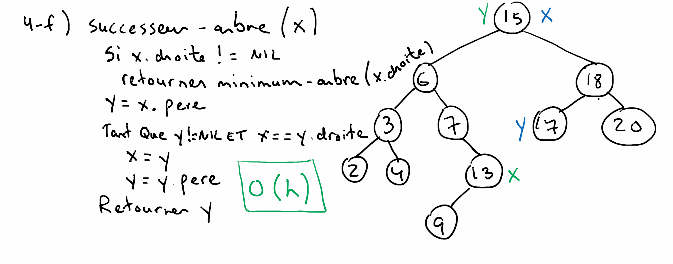
    y = x.pere

    tant que y != NIL ET x == y.droite

        x=y

        y=y.pere

    retourner y



insérer\_arbre(a,z)  //o(h)

    y = NIL

    x = a.racine

    tant que x!=NIL

        y=x

        si z.clé < x.clé

            x= x.gauche

        sinon

            x=x.droite

    z.pere =y

    si y==NIL

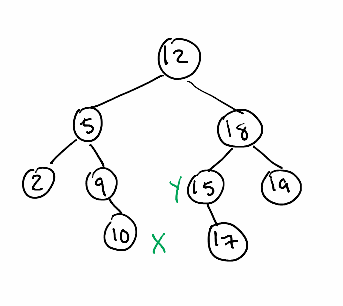
        a.racine = z

    sinon si z.cle < y.cle

        y.gauche = z

    sinon

        y.droite = z



transplanter(a, u, v) //O(1)

    si u.pere == nil

        a.racine = v

    sinon si u == u.pere.gauche

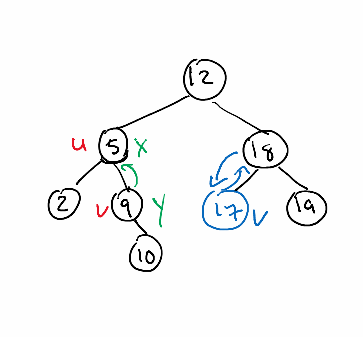
        u.pere.gauche = v

    sinon

        u.pere.droite = v

    si v!=NIL

        v.pere = u.pere



supprimer\_arbre(a,z) // O(h)

    si z.gauche == NIL

        transplanter(a,z,z.droite)

    sinon si z.droite == NIL

        transplanter(a,z,z.gauche)

    sinon

        y=minimum\_arbre(z.droite)

        si y.pere != z

            transplanter(a,y,y.droite)

            y.droite = z.droite

            y.droite.pere = y

        transplanter(a,z,y)

        y.gauche= z.gauche

        y.gauche.pere = y

