Examen juin 2010 - 2nde session

Durée : 2H

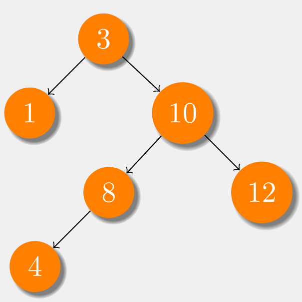
Documents autorisés: supports et notes manuscrites de cours et de TD/TME
– Barême indicatif –

Note importante : Les questions peuvent toutes être traitées indépendamment les unes des autres, au besoin en s'appuyant sur l'existence des fonctions des questions précédentes.

Exercice 1 (4 points) – **Question de cours**

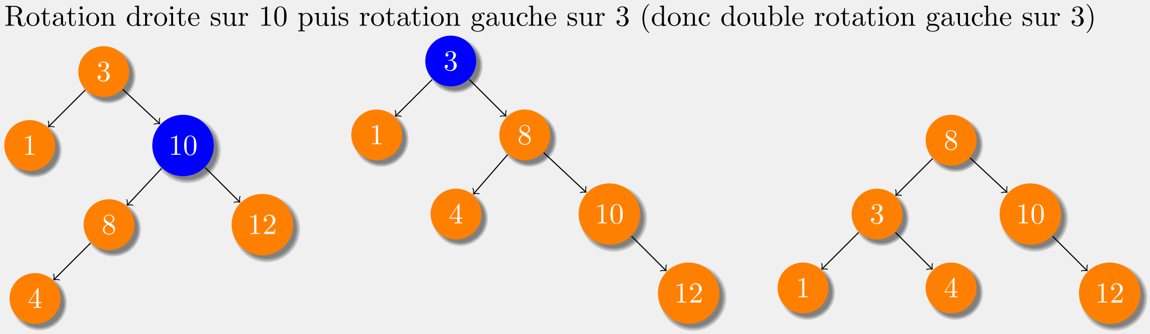
Soit la séquence d'entiers suivante : $\mathbf{3, 10, 12, 8, 4, 1}$.

Q 1.1 Arbre Binaire de Recherche
• En supposant que les entiers sont insérés dans l'ordre de la séquence, dessiner l'arbre binaire de recherche issu de cette séquence (il n'est pas demandé d'équilibrer l'arbre).

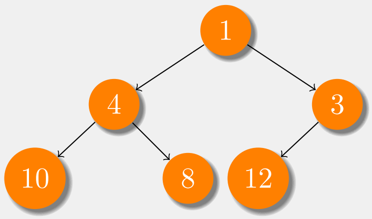


• Equilibrer l'arbre obtenu à la question précédente en proposant une rotation ou une double rotation permettant d'atteindre un ABR équilibré. Dessinez le ou les arbres correspondants.

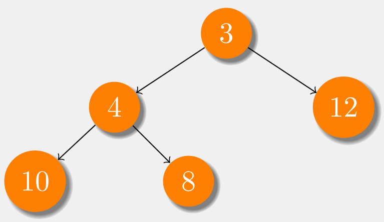
Rotation droite sur 10 puis rotation gauche sur 3 (donc double rotation gauche sur 3)



Q 1.2 Tas
• En supposant que les entiers sont insérés dans l'ordre de la séquence dans un tas manipulant un minimum, dessiner ce tas.



• Dessiner le tas obtenu après la suppression du minimum dans le tas de la question précédente.



Exercice 2 (2 points) – Utilisation de structures

On désire implémenter une structure de données abstraite nommée `table` (ou `map`) qui est définie comme un ensemble de valeurs indexées par une chaîne de caractères que l'on nomme alors `clé`. On suppose ici que la clé permet d'identifier un élément de manière unique (on dit que la clé est discriminante). Ainsi, connaissant la clé, il est possible de retrouver l'élément. Une `table` peut ainsi facilement représenter un annuaire. Dans ce cas, le tableau suivant donne toutes les informations sur la table :

Clé	Infos	
Dupont	4, place Jussieu	0685674539
Wuillemin	18, rue Sébastien	0456379413
Segura	98, place Monge	0145683498

Q 2.1 Indiquer quelle structure de données permet d'implémenter efficacement cette structure de données abstraites `table`. Justifier votre réponse.

Comme la clé est une chaîne de caractères plutôt longue, une table de haschage est tout à fait indiquée.

Q 2.2 Est-ce que le fait que la clé soit discriminante permet une gestion particulière de cette structure ? Dans le cas où la clé serait non-discriminante, votre proposition reste-t-elle valable ? efficace ?

Le fait que la clé soit discriminante permet de dire qu'un haschage à adressage ouvert peut être très efficace. Par contre, s'il y a de nombreuses clés proches voire identiques, il vaut mieux utiliser un adressage par chaînage.

Q 2.3 L'annuaire devient très volumineux avec beaucoup de personnes de même nom. Est-ce que votre réponse aux questions précédentes reste pertinente ?

> Après une certaine taille, la table de haschage n'est plus efficace. On lui préfèrerait un haschage indirect sur les premières lettres du nom puis sur le nom complet. Si l'annuaire est vraiment trop grand, il faut prévoir une gestion par logiciel de base de données qui permet de laisser une partie des données sur disque.
> *Si l'étudiant répond non en proposant une autre structure (exemple arbre), la question a tous les points*

Exercice 3 (6 points) – **Multiplication et Matrices Creuses**

Dans ce qui suit, nous supposerons la valeur N définie comme suit :

```c
#define  N 50
```

Soit $\mathcal{M}_N$ l'ensemble des matrices carrées (N, N).
$\forall A \in \mathcal{M}_N$ on indique par $A[i, j]$ l'élément (i, j) (avec $i, j \in \{0, \cdots, N-1\}$).

Q 3.1 Beaucoup de représentations d'une matrice de $\mathcal{M}_N$ sont possibles en C. En voici deux parmi les plus classiques :

```c
/* implementation A*/
typedef float Matrice[N][N];
```

```c
/* implementation B*/
typedef float Matrice[N*N];
```

Quelle est la différence en terme d'implémentation ? En supposant qu'un `float` prend 8 octets en mémoire et un pointeur 4 octets, quelle est la taille d'un objet de ces 2 types ? Qu'en concluez-vous ?

> Tableau de pointeurs de tableau de float versus tableau de float
> A : N*N*8+N*4
> B : N*N*8
> B est moins gourmand sans perte en terme de temps d'accès (remplacer un adressage indirect par un calcul de multiplication). Par contre, B demande une place disponible contiguë plus importante que A, cette place peut ne pas être disponible.

Q 3.2 Afin de ne plus avoir à se poser de questions sur l'implémentation, on propose de mettre en place une couche d'abstraction sous la forme de 2 fonctions :

```c
float get(Matrice m,int i,int j); /* retourne la valeur de la matrice m en [i,j] */
void set(Matrice m,int i,int j,float f); /* affecte f dans la matrice m en [i,j] */
```

• Pourquoi la matrice n'est-elle pas passée par pointeur dans les deux fonctions ?

> Pour les 2 implémentations, Matrice est un alias pour un pointeur, donc accès rapide (sans copie des éléments pointés) et modification du contenu possible.

• Proposer un code pour ces 2 fonctions et pour chacun des deux types d'implémentations A et B (il y a donc au total 4 petites fonctions à écrire).

```c
typedef float Matrice[N][N];

void set(Matrice m,int i,int j,float f) {
  m[i][j]=f;
}
float get(Matrice m,int i,int j) {
  return m[i][j];
}
```

```c

typedef float Matrice2[N*N];

void set(Matrice m,int i,int j,float f) {
  m[i*N+j]=f;
}
float get(Matrice2* m,int i,int j) {
  return m[i*N+j];
}
```

Q 3.3 On rappelle que le produit matriciel entre $A \in \mathcal{M}_N$ et $B \in \mathcal{M}_N$ s'écrit :

$$(A \times B)[i,j] = \sum_{k=0}^{N-1} A[i,k] \cdot B[k,j]$$

Écrire la fonction `mult(Matrice A,Matrice B,Matrice C)` qui calcule $C = A \times B$. Quelle est sa complexité (en fonction de N) ?

```c
void mult(Matrice A,Matrice B,Matrice C) {
  int i,j,k;
  for(i=0;i<N;i++)
    for(j=0;j<N;j++) {
      float res=0.0;
      for(j=0;j<N;j++) {
        res+=get(A,i,k)*get(B,k,j);
      }
      set(C,i,j,res);
    }
}
```

$O(N^3)$

Q 3.4 On a souvent besoin de représenter des matrices dites 'creuses', c'est-à-dire des matrices dont les valeurs non nulles sont très rares. Ce qui implique une représentation classique en mémoire remplie d'un très grand nombre de zéros. Afin d'éviter une telle consommation de mémoire, on représente plutôt les matrices creuses sous la forme de dictionnaire de valeurs. Ainsi une matrice telle que :

$$\begin{vmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 3 \end{vmatrix}$$

serait représentée plus efficacement par la liste :

$$\{(0,4) \Rightarrow 1; (2,0) \Rightarrow 1; (2,2) \Rightarrow 1; (3,3) \Rightarrow 2; (4,4) \Rightarrow 3\}$$

• Quelle structure de données proposeriez-vous pour représenter une telle matrice creuse (le temps d'accès à un élément est TRÈS important) ? Quelle est la complexité en mémoire de cette représentation ?

> Une table de hashage dont la clé serait basée sur l'index : $i*N+j$. Complexité en mémoire linéaire en fonction du nombre d'éléments non nuls dans le tableau.
> PS- on a bien une complexité en $O(1)$ pour l'accès aux éléments.

• Pour ce qui est de programmer la multiplication, une information importante (encore plus que le temps d'accès à un élément) serait de pouvoir connaître les colonnes ou les lignes complètement vides. Comment pourriez-vous modifier votre structure de données pour prendre en compte cette demande ?

> Deux tableaux de listes linéaires chaînées : un tableau correspondant aux lignes et un aux colonnes. Chaque case correspondant à un coefficient non nul est ainsi utilisé à la fois comme élement de la liste correspondant à sa ligne et comme élément de la liste correspondant à sa colonne. Cela double le besoin en mémoire mais (du coup) ça reste linéaire en fonction du nombre d'éléments non nuls dans la matrice.
> ou alors
> une table de hashage (comme la question précédente) + 2 vecteurs de booléen (ligne et colonnes) indiquant les lignes vides ou non.

Exercice 4 (8 points) – Détection de circuit

Le but de cet exercice est d'implanter un algorithme de détection de circuits dans un graphe orienté différent de celui vu en TD.

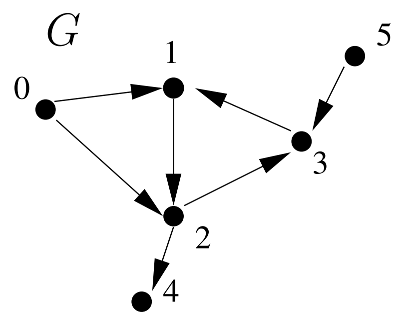
Soit $G = (S, A)$ un graphe orienté où V est un ensemble de n sommets et A un ensemble de m arcs. S'il existe un arc (i, j) dans A, on dit que i est *prédécesseur* de j et que j est *successeur* de i. On représente en mémoire un tel graphe par un tableau de listes d'adjacence qui est implémenté par les déclarations suivantes :

```
typedef struct arc {
  int i ;     // sommet i de l'arc (i,j)
  int j ;     // sommet j de l'arc (i,j)
} Arc ;

typedef struct elementListeA {
    Arc a ;
    struct elementListeA * suiv ;
} ElementListeA ;

typedef struct sommet {
    int num;
  ElementListeA *LA; // liste des aretes incidentes a ce sommet
} Sommet ;

typedef struct graphe {
    int n ; // nb de sommets
```

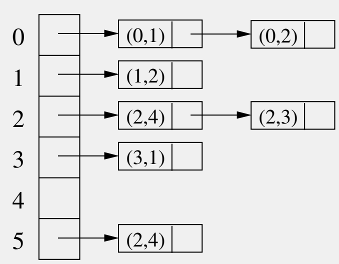
```
    Sommet* tabS ; // tableau de sommets
} Graphe ;
```

On considère qu'un graphe a déjà été alloué et affecté à des valeurs. Dans ce graphe les n sommets sont numérotés de 0 à $n-1$ et sont stockés respectivement dans les cases de 0 à $n-1$ du tableau `tabS`.

Q 4.1 Considérons le graphe G suivant :



Donner unr représentation de la structure de données correspondant à ce graphe.



Q 4.2 On veut connaître le nombre de prédécesseurs de chaque sommet. Pour cela, on définit un tableau de n entiers `t_nbpred` initialisé avec les valeurs 0. Ecrire une fonction `void compte_nbpred(Graphe G, int* t_nbpred) ;` qui affecte à chaque case i le nombre de prédécesseurs du sommet i.

```
void compte_nbpred(Graphe G, int* t_nbpred){
  int i;
  ElementListeA *cour;

  for (i=0;i<G.n;i++){
    cour=G.tabS[i].LA;
    while (cour!=NULL){
      t_nbpred[cour->a.j]++;
      cour=cour->suiv;
    }
  }
}
```

Q 4.3 Soit $T \subset S$ un sous-ensemble de sommets (cela peut-être, par exemple, tout S ou un seul sommet). Un *sous-graphe induit* par T est le graphe obtenu à partir de G en ne gardant que les sommets

de T et uniquement les arcs entre deux sommets de T. Ainsi, pour un sommet i de T, on compte uniquement les prédécesseurs de i qui sont dans T.
On code T par un tableau de n sommets tel que $T[i]$ vaut 1 si le sommet est dans T et vaut 0 si le sommet n'est pas contenu dans T.
On suppose dans cette question que, pour tout sommet i de T, $t\_nbpred[i]$ contient le nombre des prédécesseurs de i qui sont dans T et, pour tous les autres sommets, $t\_nbpred[i]$ contient une valeur quelconque. Donner une fonction `int rech_sanspred(Graphe G, int* T, int* t_nbpred) ;` qui retourne le numéro d'un sommet de T qui n'a pas de prédecesseur dans T, s'il en existe un, et -1 s'il n'existe pas un tel prédécesseur.

```c
int rech_sanspred(Graphe G, int* T, int* t_nbpred){
  int i=0;
  while (i!=G.n) {
    if ((T[i]==1)&&(t_nbpred[i]==0))
      return i;
    i++;
  }
  return -1;
}
```

Q 4.4 Soit $T \subset S$ et le tableau `t_nbpred` du nombre de prédécesseurs des sommets de T. On veut supprimer un sommet u de T, c'est-à-dire effectuer l'instruction $T[u] = 0$. Donner la fonction `void maj_nbpred(Graphe G, int* T, int* t_nbpred, int u) ;` qui effectue la mise à jour du tableau `t_nbpred` pour l'ensemble $T \setminus \{u\}$.

```c
void maj_nbpred(Graphe G, int* T, int* t_nbpred, int u){
  ElementListeA *cour;
  cour=G.tabS[u].LA;
  while (cour!=NULL){
    if (T[cour->a.j]==1)  // On peut se passer de ce test
      t_nbpred[cour->a.j]--;
    cour=cour->suiv;
  }
}
```

Q 4.5 L'idée de l'algorithme de détection d'un circuit dans un graphe repose sur les deux propriétés suivantes :
Proposition 1 : Si un graphe est sans circuit, alors il contient un sommet sans prédécesseur.
Proposition 2 : Soit $G = (S, A)$ un graphe (avec ou sans circuit) et u un sommet sans prédécesseur de G. Alors G contient un circuit si et seulement si le sous-graphe induit par $S\{u\}$ contient un circuit.
En utilisant ces deux propositions et les 3 fonctions vues dans les questions précédentes, proposer une fonction `int detect_circuit(Graphe G) ;` qui retourne vrai si et seulement si G contient un circuit.

```c
int detect_circuit(Graphe G){
  int* T;
  int* t_nbpred;
```

```
  int cpt;
  int i,u;

  T=(int *) malloc(G.n*sizeof(int));
  for (i=0;i<G.n;i++)
    T[i]=1;

  t_nbpred=(int *)  malloc(G.n*sizeof(int));
  for (i=0;i<G.n;i++)
    t_nbpred[i]=0;
  compte_nbpred(G,t_nbpred);

  cpt=G.n;
  while (cpt!=0){
    u=rech_sanspred(G,T,t_nbpred);
    if (u==-1)
      return 1==1;
    T[u]=0;
    maj_nbpred(G,T,t_nbpred,u);

    cpt--;
  }

  return 1==2;
}
```

Q 4.6 (Bonus) Donner la complexité de la fonction `compte_nbpred` en fonction du nombre d'arcs dans le graphe.
Au total des appels de la fonction `maj_nbpred`, combien de fois (en fonction du nombre d'arcs) sont appelées les lignes à l'intérieur de la boucle contenue dans la fonction `maj_nbpred` ?
Quelle est la complexité de la fonction `rech_sanspred` et combien de fois est-elle utilisée ?
En déduire la complexité de la fonction `detect_circuit`.
Proposer une structure de données pour améliorer la complexité de `detect_circuit` ? Quelle complexité obtiendrait-on alors ?

`compte_nbpred` parcourt une fois chaque arc du graphe : $O(m)$.
Au total des appels, `maj_nbpred` passe une fois par arcs, ainsi les lignes de la fonction sont appelées au plus m fois.
Par contre, la fonction `rech_sanspred` a une complexité de $O(m)$ dans le pire des cas car elle parcourt tout le tableau. Or cette fonction est appelée n fois.
Au total, on a donc une détection de circuit en $O(n^2)$.
En utilisant une structure de tas à la place d'un tableau pour coder les nombres de prédécesseurs, on pourrait accéder en $O(1)$ au sommet ayant le moins de prédécesseur (donc celui sans prédécesseur s'il existe). En revanche, la mise à jour du tableau serait alors en $O(log(n))$ pour chaque sommet rencontré, soit $O(mlog(n)$ au total des mises à jour. La fonction `detect_circuit` serait donc alors en $O(mlog(n))$.