Examen juin 2010 - 1ère session

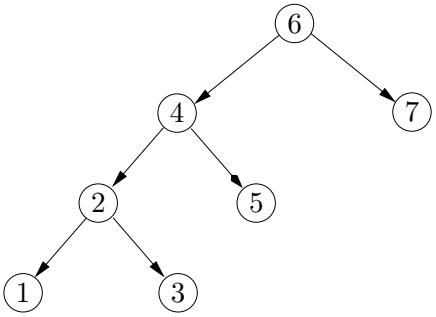
Durée : 2H

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

Note importante : Les questions peuvent être traitées indépendamment les unes des autres, au besoin en considérant écrites les fonctions des questions précédentes.

Exercice 1 (4 points) – Questions de cours

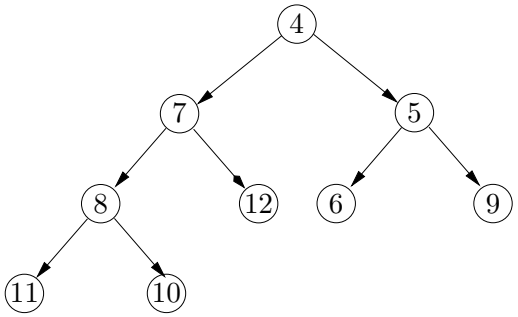
Q 1.1 Voici un arbre :



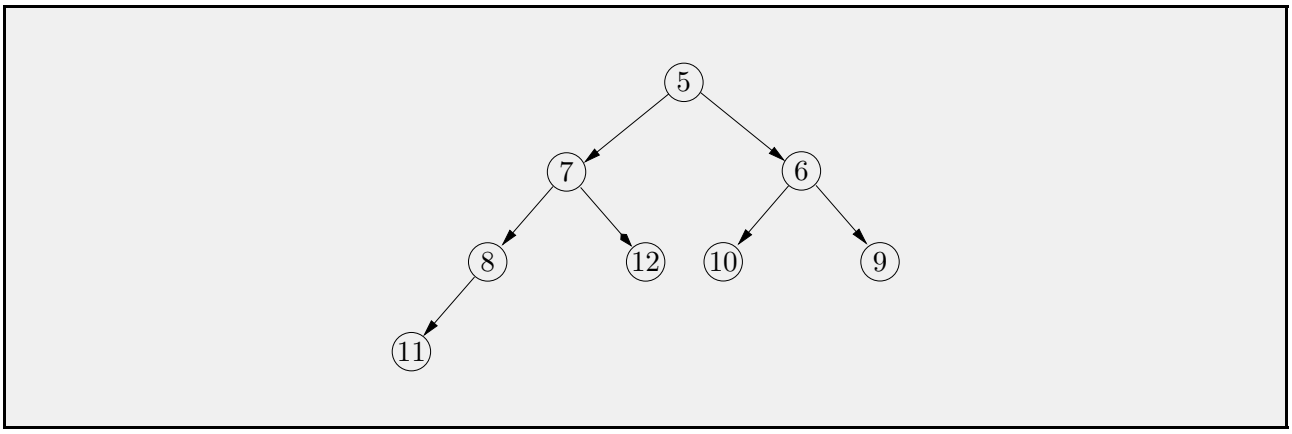
Est-ce a) un tas ? b) un ABR ? c) un ABR équilibré ? Justifier la réponse.

La réponse est b).
Ce n'est pas un tas car le sommet-racine n'est pas de plus faible ou plus haute priorité. C'est un ABR car chaque sommet est plus grand que tous ses fils gauches et plus petit que tous ses fils droits. Il n'est pas équilibré car le sous-arbre gauche de 6 est plus profond de 2 que son sous-arbre droit.

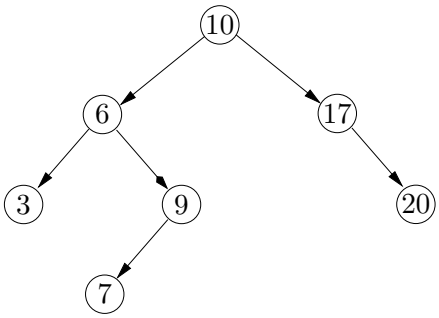
Q 1.2 Voici un tas représenté sous la forme d'un arbre :



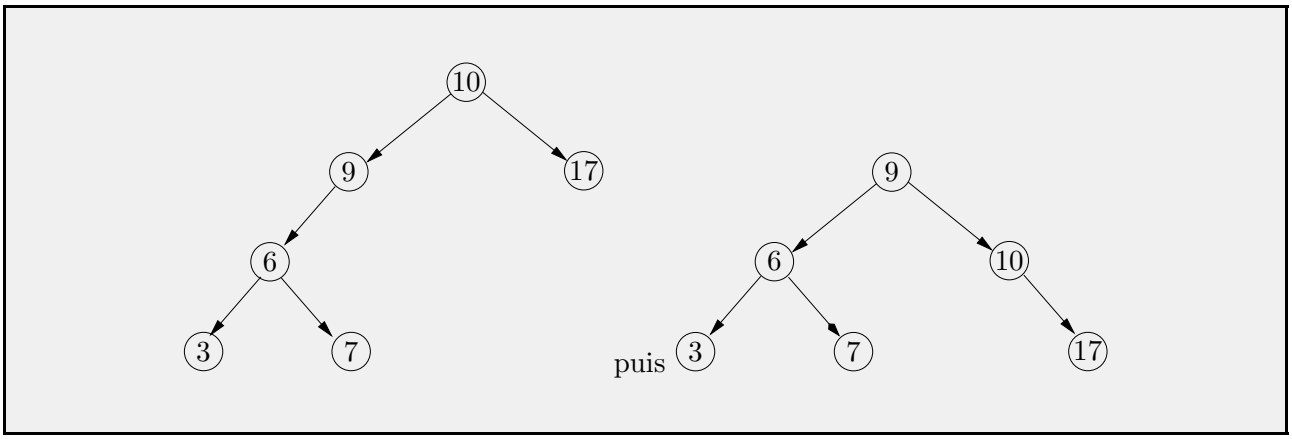
Après suppression de l'élément de priorité 4, quel tas obtient-on ?



Q 1.3 Voici un ABR équilibré :



Après suppression de l'élément 20, quel ABR équilibré obtient-on ?



Exercice 2 (2 points) – Utilisation de structures

On considère ici un ensemble de données provenant d'un ouvrage de philosophie. On a listé tous les mots utiles de l'ouvrage, c'est-à-dire des mots différents des verbes, des pronoms etc et qui correspondent à des "idées". Soit $m1$ et $m2$ deux mots utiles différents, on recherche dans le texte s'il existe une phrase les contenant tous les deux et on évalue cette phrase de manière à décider si elle constitue un lien logique direct entre $m1$ et $m2$.

On veut déterminer s'il existe un lien logique entre deux mots utiles quelconque pris dans l'ouvrage. Proposer une structure permettant de stocker ces données ainsi qu'un algorithme permettant d'effectuer cette recherche.

On considère un graphe non-orienté $G(V, E)$ où chaque sommet est associé à un mot utile et où chaque arête entre deux mots utiles correspond à un lien logique direct (i.e. une phrase). L'ensemble des données utiles est ainsi stocké dans cette structure de graphe. Soit u et v deux mots utiles. Rechercher s'il existe un lien logique entre ces deux mots revient à déterminer s'il existe une chaîne entre ces deux mots dans G. Ceci peut être fait par un parcours (par exemple en largeur) du graphe à partir de u et de voir si v apparaît dans ce parcours. Si c'est le cas, c'est qu'il existe un chemin entre u et v.

Exercice 3 (6 points) – Dictionnaire LZ78

Un dictionnaire LZ78 sert à stocker de manière compacte des chaînes de caractères. Il est composé d'un tableau de couples de type `(int,char)`. Chaque case d'indice `i` du tableau contenant le couple `(ind,c)` correspond à une chaîne obtenue en concaténant la chaîne correspondant à la case d'indice *ind* suivie du caractère `c`.
Certaines cases du tableau ne suivent pas cette règle : la case d'indice 0 qui contient `(-1,' ')` et qui représente la chaîne de caractères vide et les dernières cases du tableau (de valeur `(-2,'_')`) ne représentant aucune chaîne lorsque le dictionnaire n'est pas entièrement rempli.

0	`(-1, ' ')`	
1	`(0, 'l')`	`"l"`
2	`(0, 'b')`	`"b"`
3	`(2, 'a')`	`"ba"`
4	`(3, 'c')`	`"bac"`
5	`(1, 'e')`	`"le"`
6	`(3, 'a')`	`"baa"`
7	`(-2, '_')`	case vide
8	`(-2, '_')`	case vide

Fig. 1 – un dictionnaire LZ78

La figure 1 présente un exemple de dictionnaire pouvant contenir jusqu'à 8 mots mais qui n'en contient que 6. La case 6 contient le mot se terminant par `'a'`, précédé du mot de la case 3 ; celle-ci contient le mot terminé par `'a'`, précédé du mot de la case 2 ; celle-ci contient le mot `'b'`. La case 6 représente donc le mot `"baa"`.

Q 3.1 Proposer les types `couple` et `dictLZ78` nécessaires à la représentation d'un dictionnaire LZ78 qui contient au plus `MAX` éléments où `MAX` est une constante. Proposer les lignes de code permettant de remplir une variable `dico1` avec le dictionnaire de l'exemple donné par la figure 1.

```c
#define MAX 9

typedef struct {
  int ind;
  char c;
} couple;

typedef couple dictLZ78[MAX];
```

```c
dictLZ78 dico1={
  {-1,'␣'},
  { 0,'a'},
  { 0,'b'},
  { 3,'a'},
  { 3,'c'},
  { 1,'e'},
  { 3,'a'},
  {-2,'_'},
  {-2,'_'}
};
```

Q 3.2 Proposer une fonction `int find_char(int ind, char c, dictL278 dico, int debut)` qui recherche le couple `(index,c)` dans le dictionnaire `dico` à partir de la case d'indice `debut`. Si elle trouve ce couple, alors elle renvoie l'indice de la case correspondante, sinon elle renvoie -1.
Par exemple, pour le dictionnaire `dico1` de la figure 1 , `find_char(2,'a',dico1,0)` renvoie 3 car `(2,'a')` se trouve à l'indice 3 du tableau. En revanche, `(find_char (2,'a',dico, 4)` renvoie -1 car le tableau ne contient pas `(2,'a')` dans les cases 4, 5, 6, etc.
Attention, n'oubliez pas que le tableau `dico` n'est pas forcément entièrement rempli ...

```c
int find_char(int ind,char c,dictLZ78 dico,int debut) {
  int i;
  for(i=debut;i<MAX;i++) {
    if (dico[i].ind==-2) return -1;
    if ((dico[i].ind==ind) && (dico[i].c==c)) return i;
  }
  return -1;
}
```

Q 3.3 Proposer une fonction `int find_str(char* mot, dictLZ78 dico)` qui recherche la chaîne de caractères `mot` dans le dictionnaire `dico`. Si celle-ci existe, la fonction renvoie l'indice de la case correspondant à la chaîne de caractères[1] , sinon elle renvoie -1.
Par exemple, dans le dictionnaire `dico1` de la figure 1, `find_str("bac",d)` renverrait 4 car l'indice 4 du tableau correspond à la chaîne `"bac"`

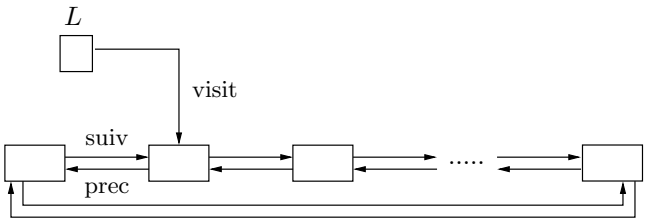
[1]Rappels : une chaîne de caractère en C est un tableau de caractères. On accède donc au i-ème caractère de la chaîne `s` par `s[i]`. Le module standard `string.h` permet d'utiliser, entre autre, la fonction `strlen(s)` qui renvoie la longueur de la chaîne `s`.

```
int find_str(char* mot,dictLZ78 dico) {
  int i;
  int indice=0;

  for(i=0;i<strlen(mot);i++) {
    int ind2=find_char(indice,mot[i],dico,0);
    if (ind2==-1) return -1;
    indice=ind2;
  }
  return indice;
}
```

Exercice 4 (8 points) – Liste circulaire doublement chaînée

On désire gérer un fichier d'individus triés alphabétiquement par une liste chaînée. On remarque que chaque recherche débute toujours par le premier nom du classement alphabétique, ce qui n'est pas très efficace si l'on recherche plusieurs fois de suite des noms commençant par z par exemple. Une façon d'éviter ce défaut est d'utiliser une liste doublement chaînée circulaire.
Une liste chaînée est dite *circulaire* si son dernier élément pointe sur son premier élément. Une liste chaînée est dite *doublement chaînée* si chaque élément pointe sur l'élément précédent. Une *liste doublement chaînée circulaire* LDCC est donc à la fois doublement chaînée et circulaire. Dans une telle liste, il n'y a pas de premier élément mais il existe un pointeur sur un élément quelconque que l'on dit être *l'élément visité*.
Note Importante : Pour simplifier l'exercice, nous considérons ici que la liste contient toujours au moins un élément.



La figure ci-dessus donne le schéma de cette structure de données. Chaque cellule contient un nom codé par le type `char[20]` et un numéro de téléphone codé par le type `int`. Chaque liste contient un champ `int numero` permettant de nommer la liste et un pointeur `visit` sur l'élément visité.

Q 4.1 Donner le code définissant le type d'un élément `Cellule` de cette structure. Donner le code définissant le type `LDCC` d'une liste doublement chaînée circulaire.

```
  typedef struct cellule{
  char nom[20];
  int num;

  struct cellule *suiv;
  struct cellule *prec;

```

```
8
9  } Cellule;
10
11
12 typedef struct ldcc{
13   int numero;
14   Cellule * visit;
15 } LDCC;
```

Non demandé dans l'examen, mais nécessaire, l'initiatisation de la liste avec un premier élément

```
1  void init(LDCC *L, char nom[20], int num){
2   Cellule *nouv=(Cellule *) malloc(sizeof(Cellule));
3
4    strcpy(nouv->nom,nom);
5    nouv->num=num;
6    nouv->suiv=nouv;
7    nouv->prec=nouv;
8
9    L->visit=nouv;
10
11 }
```

Q 4.2 Donner la fonction `void affiche(LDCC L, int sens)` qui affiche une liste `LDCC` à partir de l'élément visité dans le sens de lecture par la gauche (`sens==0`) ou par la droite (`sens==1`).

```
1     void affiche(LDCC L, int sens){
2    Cellule *cour;
3    printf("Liste numero: %d\n",L.numero);
4
5    cour=L.visit;
6
7    do{
8      printf("  %s %d\n",cour->nom, cour->num);
9      if (sens==1) cour=cour->suiv;
10     else cour=cour->prec;
11
12   }while (cour!=L.visit);
13   printf("\n");
14 }
```

Q 4.3 Donner une fonction `inserer_apres_visit` qui insère un élément (repéré par son nom et son numéro) dans la liste à la droite de l'élément pointé par `visit`.
On rappelle que la fonction `strcpy(char *ch1, char* ch2)` permet de copier le contenu de la chaîne de caractère `ch2` dans la chaîne de caractères `ch1`.

```
1     void insere_apres_visit(LDCC *L, char nom[20], int num){
2    Cellule *nouv=(Cellule *) malloc(sizeof(Cellule));
3
4    strcpy(nouv->nom,nom);
5    nouv->num=num;
```

```
6
7     nouv->prec=L->visit;
8     nouv->suiv=L->visit->suiv;
9     L->visit->suiv->prec=nouv;
10    L->visit->suiv=nouv;
11
12  }
```

Q 4.4 Donner une fonction `recherche` qui recherche si un élément de nom `char nom[20]` est dans la liste en effectuant un parcours de la liste démarrant à gauche ou à droite de l'élément visité. La fonction renvoie vraie (valeur 1) si et seulement si l'élément est trouvé et place dans ce cas le pointeur `visit` sur cet élément.
On rappelle que la fonction `strcmp(char *ch1, char *ch2)` renvoie 0 si et seulement si les deux chaînes `ch1` et `ch2` sont égales.

```
1  int recherche(LDCC *L, char nom[20], int sens){
2
3    Cellule *cour;
4
5    cour=L->visit;
6
7    if (strcmp(nom,cour->nom)==0){
8        return 1;
9      }
10   else{
11       if (sens==1) cour=cour->suiv;
12    else cour=cour->prec;
13       while ( (cour!=L->visit) && (strcmp(nom,cour->nom)!=0) ){
14    if (sens==1) cour=cour->suiv;
15    else cour=cour->prec;
16       }
17       if (strcmp(nom,cour->nom)==0){
18         L->visit=cour;
19         return 1;
20       }
21
22       return 0;
23   }
24
25 }
```

Et un main (non demandé) pour tester tout cela.

```
1  int main(){
2    LDCC L;
3
4    L.numero=77;
5    init(&L,"aa",5);
6    insere_apres_visit(&L,"cc",6);
7    insere_apres_visit(&L,"ee",56);
8    affiche(L,0);
9    affiche(L,1);
10
11   if (recherche(&L,"ee",0)){
12     printf("Trouve\n");
```

```
13      affiche(L,0);
14    }
15
16
17    return 0;
18  }
```

Q 4.5 Est-ce que cette recherche possède une meilleure complexité que celle de la liste chaînée simple ?

Non, le pire des cas est le même, il faut parcourir toute la liste. La recherche reste donc en $O(n)$ où n est le nombre d'éléments.

Q 4.6 Proposer brièvement un protocole expérimental permettant de comparer la vitesse de la fonction `rechercher` pour une liste LDCC et pour une liste simplement chaînée. Que pensez-vous obtenir comme conclusion à cette étude expérimentale ?
Lors de la recherche, il peut être plus rapide de parcourir la liste vers la gauche plutôt que vers la droite. Décriver (sans donner le code) une façon d'utiliser cette idée en modifiant légèrement la fonction `inserer`.

Un protocole expérimental peut être décrit de la façon suivante. On génère aléatoirement un annuaire de très grande taille. On remplit une liste simple et une liste LDCC avec cet annuaire. On lance des recherche aléatoire des noms de l'annuaire sur les deux listes. On mesure le temps total de recherche pour les deux listes et on calcule le temps moyen de recherche d'un nom.
Suivant comment la recherche aléatoire a été faite, il est possible que la recherche dans une liste LDCC soit plus intéressante en moyenne car démarrer du pointeur visit constitue un départ aléatoire de recherche qui peut être corrélé avec la recherche en cours. Par contre, si la génération aléatoire est totalement uniforme, les deux listes restent encore équivalente.
Une amélioration nette de ce temps moyen de recherche peut être obtenue en pilotant la recherche par la connaissance de l'annuaire. Par exemple, si visit pointe sur un nom commençant par M et que l'on recherche un L, on partira par la gauche etc. Cette astuce améliorera en moyenne la vitesse de recherche (même si la complexité pire-cas reste $O(n)$.)