Graphes, Arbres, Arbres Binaires et Arbres Binaires de Recherche

1. Notions et éléments de vocabulaire liés aux graphes (donnés en cours, ou voir le wiki relatif à la *Théorie des graphes*) :
   (a) Structure simple ou complexe de stockage, de représentation de données et des relations entre celles-ci (une relation entre deux données est elle-même une donnée) ;
   (b) Un graphe peut être composé de **nœuds** (ou **sommets**) et d'**arcs** (ou **arêtes**) permettant de relier les nœuds entre-eux ;
   (c) Graphe **dirigé** (aussi **orienté**) ou **non-dirigé** ;
   (d) Graphe **pondéré** ou **non-pondéré** : les arcs (ou arêtes) peuvent être pondérés (un ou plusieurs **poids** chacun représentant une notion de coût) ;
   (e) Graphe **cyclique** ou **acyclique** ;
   (f) Les **composantes connexes** d'un graphe : graphe à plusieurs composantes connexes qui sont disjointes entre-elles ;
   (g) Un nœud **intermédiaire**, un nœud **terminal** (ou **feuille**) ;
   (h) L'arbre : cas particulier du graphe.
2. Notions et éléments de vocabulaire liés aux arbres :
   (a) La **racine**, les **nœuds intermédiaires** ou les **feuilles** ;
   (b) La **profondeur** d'un arbre ;
   (c) La relation **parent** $\rightarrow$ **enfant** ;
   (d) L'arbre **quelconque** ;
   (e) L'arbre **n-aire** ;
   (f) L'arbre **binaire**.
3. **Partitionner l'espace** pour **rechercher** un (des) élément(s) :
   (a) Faire, sur une donnée trié, un comparatif entre une recherche naïve et une recherche par **dichotomie** ;
   (b) L'**arbre binaire de recherche** partitionne l'espace $1D$;
   (c) Le **quatree** partitionne l'espace $2D$;
   (d) L'**octree** partitionne l'espace $3D$ ...
4. **Implémentation** (faite en cours) d'un arbre binaire de recherche appliqué à des entiers. Pour une structure `node_t` telle que :

```
typedef struct node_t node_t;
struct node_t {
  int v;
  /* deux pointeurs, un left-child et un right-child */
  struct node_t * lc, * rc;
};
```

   Voici les règles suivies pour insérer un élément (si on a un pointeur de pointeur de nœud - `node_t ** pbt`) :
   (a) si le pointeur (`pbt`) pointe sur un pointeur indiquant un arbre vide, alors l'emplacement d'insertion est atteint, c'est ici qu'il faut créer et insérer le nouveau nœud ;
   (b) si la valeur que je souhaite insérer est inférieure à la valeur contenu dans le nœud doublement pointé, alors modifier votre pointeur (`pbt`) pour aller à gauche ;

(c) sinon (c'est que la valeur que je souhaite insérer est supérieure ou égale à la valeur contenu dans le nœud doublement pointé), alors modifier votre pointeur (`pbt`) pour aller à droite.

Et pour un `main` (donné en cours) tel que :

```
int main(void) {
  int i, t[] = { 8, 12, 10, 4, 9, 11, 2, 14,
                 15, 13, 6, 5, 3, 1, 0, 7 };
  node_t * ab = NULL;
  for(i = 0; i < (sizeof t / sizeof *t); ++i)
    btInsert(t[i], &ab);
  btPrint(ab);
  btFree(&ab);
  return 0;
}
```

Écrire :

(a) La fonction récursive `void btPrint(node_t * bt);` qui imprime (printf) les éléments de l'arbre dans l'ordre croissant. **Version proposée en cours :**

```
void btPrint(node_t * bt) {
  if(!bt)
    return;
  btPrint(bt->lc);
  printf("%d ", bt->v);
  btPrint(bt->rc);
}
```

(b) La fonction `void btInsert(int v, node_t ** pbt);` qui créé le nœud contenant `v` et l'ajoute à l'arbre binaire (trié croissant) pointé par `pbt`. **Version proposée en cours :**

```
void btInsert(int v, node_t ** pbt) {
  while(*pbt) {
    if(v < (*pbt)->v)
      pbt = &((*pbt)->lc);
    else
      pbt = &((*pbt)->rc);
  }
  *pbt = malloc(1 * sizeof **pbt);
  assert(*pbt);
  (*pbt)->v = v;
  (*pbt)->lc = (*pbt)->rc = NULL;
}
```

(c) La fonction récursive `void btFree(node_t ** bt);` qui libère l'arbre binaire. **A vos claviers !**

(d) Ajouter l'arbre binaire dans le comparatif d'insertion (liste chaînée VS tableau) donné sur la page de cours `comparatif-1.0.tgz`. **A vos claviers !**

(e) La fonction itérative `void bt_iprint(node_t * bt);` qui imprime (printf) les éléments de l'arbre binaire dans l'ordre croissant ; vous pouvez vous aider d'une pile. **A vos claviers !**

(f) Réécrire l'ensemble de manière à stocker des éléments de type générique. **C'est un peu de choses près le DM 02 ci-après.**

DM 02 (lien et dates sur la page de cours) : Pour une structure `node_t` telle que :

```
typedef struct node_t node_t;
struct node_t {
    void * data;
    struct node_t * lc, * rc;
};
```

1. La fonction `void bt_add_value(node_t ** bt, void * data, size_t s, int (* compar)(const void *, const void *));` qui créé le nœud contenant une copie de `data`, une donnée de taille `s`, et l'ajoute à l'arbre binaire `bt` trié en ordre croissant en fonction de la valeur de retour de la fonction de comparaison `compar`.
2. La fonction `void bt_print(node_t * bt, void (*print)(const void *));` qui imprime (utilise le pointeur de fonction `print` passé en argument) les éléments de l'arbre dans l'ordre croissant.
3. La fonction `void bt_free(node_t ** bt);` qui libère l'arbre binaire.

Préparer l'ensemble dans un couple `bintree.c / bintree.h` le tout dans un dossier archivé en tar.gz et transmettez-le via l'interface de soumission disponible en ligne. Rajoutez, dans le fichier `bintree.c` l'entête :

```
/*
 * NOM : <votre nom et prénoms>
 * NUMERO : <votre numéro d'étudiant>
 * EMAIL : <votre email>
 */
```