Projet de synthèse

Giorgio Lucarelli

giorgio.lucarelli@univ-lorraine.fr

Janvier MMXXII

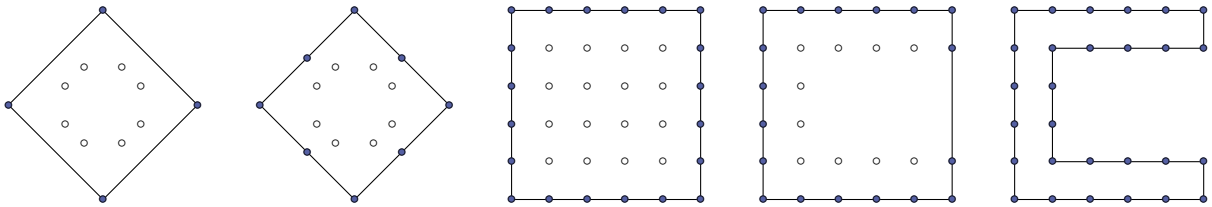
UNIVERSITÉ DE LORRAINE | UFR MATHÉMATIQUES INFORMATIQUE MÉCANIQUE ET AUTOMATIQUE

Algorithmes pour calculer l'enveloppe convexe en 2D

1 Introduction

L'objectif de ce projet est d'utiliser et d'appliquer les connaissances acquises au cours de l'année telles que Maths Discrètes, Programmation en C, Algorithmique et Structures de Données, Programmation en Java et Interfaces Graphiques. Le projet va être réalisé en binômes du même groupe TP. Pour des raisons d'organisation, **aucune exception de cette règle ne sera toléré.**

Dans ce projet, vous allez implémenter et simuler de différents algorithmes pour calculer l'enveloppe convexe (*convex hull* en anglais) d'un ensemble des points dans le plan. Un polygone est appelé convexe si "tout segment joignant deux sommets du polygone est inclus dans la composante fermée bornée délimitée par le polygone" (source : widipedia). Par exemple les polygones (a)–(d) de la figure suivante sont convexes, tandis que le polygone (e) n'est pas convexe car le segment qui lie le point en haut à gauche avec le point en bas à droite n'est pas complètement à l'intérieur du polygone.



(a) (b) (c) (d) (e)

L'entrée du problème consiste d'un ensemble de N points $\mathcal{P} = \{P_1, P_2, \ldots, P_N\}$. Chaque point $P_i \in \mathcal{P}$ est caractérisée par son abscisse X_i et son ordonnée Y_i. Notre objectif est de trouver le plus petit polygone convexe qui contient tous les points de $\mathcal{P}$. Cet enveloppe convexe (la sortie de notre problème) est décrit par un ensemble de points $\mathcal{H} \subseteq \mathcal{P}$ ordonnés dans le sens horaire.

Plusieurs algorithmes sont proposés dans la littérature scientifique pour ce problème. Dans le cadre de notre projet, nous implémenterons 3 algorithmes, chacun d'une complexité théorique différente, avec l'objectif final de comparer expérimentalement la performance de ces trois algorithmes.

Un de ces algorithmes a besoin de trier les points de l'entrée. Pour cette raison, notre projet consiste d'implémenter aussi trois algorithmes de tri avec de différents caractéristiques concernant leur complexité et la taille de la mémoire utilisée. Plus précisément, on utilisera un tri par sélection, un tri par tas représenté par un arbre binaire complet, et un tri par tas représenté par un tableau. Les structures de données nécessaires vont aussi être implémentées.

1.1 Comment commencer ?

Pour vous aider à l'organisation de votre projet, le squelette de la partie en C est fourni sur Arche. Il contient 7 fichiers `.h` avec les prototypes des fonctions et les structures que vous devrez utiliser

(répertoire `include\`) ainsi que 8 fichiers `.c` qui sont à modifier afin de réaliser le projet (répertoire `src\`).

> **Attention !** Vous n'avez pas le droit de modifier les prototypes des fonctions et des structures fournis. Autrement dit, vous ne devez pas modifier le nom des fonctions, le type et le nom des paramètres des fonctions et des structures, ainsi que leur nombre (vous ne pouvez pas ajouter ou supprimer des paramètres). En revanche, vous pouvez ajouter des fonctions auxiliaires.

Avant de commencer à coder, n'oubliez pas de créer un projet GIT par équipe (par exemple sur https://about.gitlab.com/ ou sur https://gitlab.univ-lorraine.fr/ en utilisant votre adresse email et votre mot de passe universitaire). Ajoutez y le squelette fourni. Ensuite, chaque membre de l'équipe peut avoir une copie locale du projet et commencer à travailler. N'oubliez pas de mettre à jour le serveur du GIT régulièrement (par exemple, une fois que le code d'une fonction sera finie et testée ou si un bogue sera corrigé), afin de communiquer à vos coéquipiers l'avancement que vous avez fait. *L'historique du GIT va faire partie de l'évaluation.*

Le document actuel donne une description des fichiers fournis et vous propose l'ordre d'implémentation du projet.

1.2 Structures de données

Dans ce projet on aura besoin d'implémenter les structures de données suivantes :

— Listes Doublement Chaînées : pour la représentation de la sortie (enveloppe convexe) mais aussi utilisée pendant l'exécution de certains algorithmes.
— Arbres Binaires Complets : pour la représentation d'un tas.
— Tas avec pointeurs en utilisant la structure de l'Arbre Binaire Complet ci-dessus : pour l'implémentation de notre première version de l'algorithme tri par tas.
— Tas avec tableau de taille fixe : pour l'implémentation de la deuxième version de l'algorithme tri par tas.

Les deux structures de tas seront mis en compétition par rapport à leur performance.

2 Fonctions auxiliaires fournies

Les fichiers `include/util.h` et `src/util.c` contiennent la définition et l'implémentation de certains éléments auxiliaires. D'abord, l'énumération suivante définit le type de la structure de données à utiliser.

Les sous-programmes suivants sont fournies et vous aiderons dans la réalisation du projet.

```
/* Renvoyer la valeur maximale parmi a et b. */
#define max(a,b) \
    ({ __typeof__ (a) _a = (a); \
        __typeof__ (b) _b = (b); \
        _a > _b ? _a : _b; })

/* Renvoyer la valeur minimale parmi a et b. */
#define min(a,b) \
    ({ __typeof__ (a) _a = (a); \
        __typeof__ (b) _b = (b); \
        _a < _b ? _a : _b; })

/* Affiche le message d'erreur msg et interrompt l'exécution
 * du programme si le paramètre interrupt vaut 1.*/
void ShowMessage(char * msg, int interrupt);
```

```c
/* Afficher l'entier indiqué par le pointeur i. */
void viewInt(const void *i);

/* Libérer la mémoire de l'entier indiqué par le pointeur i. */
void freeInt(void * i);

/* Compare les entiers a et b et renvoie 1 si a<b, sinon 0. */
int intSmallerThan(const void* a, const void* b);

/* Compare les entiers a et b et renvoie 1 si a>b, sinon 0. */
int intGreaterThan(const void* a, const void* b);
```

3 Listes doublement chaînées

Vous avez déjà implémenté une liste doublement chaînée au cours "Programmation Avancée" du semestre précédent. Nous allons donc refaire cette partie et ajouter quelques fonctions supplémentaires pour commencer notre projet. En suite, vous trouverez une description des prototypes fournis ainsi que les fonctions que vous devez coder en forme d'exercices.

La structure suivante correspond à un nœud d'une liste (voir `include/list.h`) :

```c
typedef struct ListNode {
    void * data; // donnée
    struct ListNode * succ, * pred; // successeur - prédécesseur
} LNode;
```

Notez que le type de données stockées dans chaque nœud de la liste est `void*`. Il s'agit d'un pointeur vers un élément dont le type n'est pas encore défini. L'objectif d'avoir des données de type `void*` est d'éviter d'implémenter plusieurs listes chaînées, une pour chaque utilisation différente. Par exemple, si nous n'utilisions pas le type `void*`, nous devrions implémenter une liste pour stocker de valeurs entières, les points, etc.

Exercice 1 Implémentez dans le fichier `src/list.c` les primitives de la structure correspondante au nœud d'une liste (les prototypes se trouvent dans `include/list.h`).

```c
// Construction (n'oubliez pas d'allouer de la mémoire)
LNode * newLNode(void* data);

// Consultation
void* getLNodeData(const LNode* node);
LNode* Successor(const LNode* node);
LNode* Predecessor(const LNode* node);

// Modification
void setLNodeData(LNode* node, void* newData);
void setSuccessor(LNode* node, LNode* newSucc);
void setPredecessor(LNode* node, LNode* NewPred);
```

La structure suivante correspond à une liste (voir `include/list.h`) :

```c
typedef struct List {
    LNode *head;  // pointeur vers le premier nœud
    LNode *tail;  // pointeur vers le dernier nœud
    int numelm;   // nombre d'éléments
```

```
    void (*viewData)(const void* data); // affiche les données
    void (*freeData)(void* data);       // supprime les données
} List;
```

Exercice 2 Implémentez dans le fichier `src/list.c` les primitives de la structure d'une liste (les prototypes se trouvent dans `include/list.h`).

```
// Construction (n'oubliez pas d'allouer de la mémoire)
List* newList(void (*viewData)(const void*), void (*freeData)(void*));

// Consultation
int listIsEmpty(List* L);
int getListSize(const List* L);
LNode* Head(const List* L);
LNode* Tail(const List* L);

// Modification
void increaseListSize(List* L);
void decreaseListSize(List* L);
void setListSize(List* L, int newSize);
void resetListSize(List* L);
void setHead(List* L, LNode* newHead);
void setTail(List* L, LNode* newTail);

// Affichage
void viewList(const List * L);

// Libération de la mémoire
void freeList(List * L, int deleteData);
```

Exercice 3 Implémentez dans le fichier `src/list.c` les fonctions suivantes qui insèrent un élément dans la liste `L` (les prototypes se trouvent dans `include/list.h`). Faites attention de mettre à jour la taille de la liste lors de l'insertion d'un élément.

```
/* Insère en tête de la liste L un nouveau nœud de donnée data. */
void listInsertFirst(List * L, void * data);

/* Insère à la fin de la liste L un nouveau nœud de donnée data. */
void listInsertLast(List * L, void * data);

/* Insère un nouveau nœud de donnée data dans la liste L
 * après le nœud indiqué par le pointeur ptrelm. */
void listInsertAfter(List * L, void * data, LNode * ptrelm);
```

Exercice 4 Implémentez dans le fichier `src/list.c` les fonctions suivantes qui suppriment un élément de la liste `L` (les prototypes se trouvent dans `include/list.h`). Faites attention de mettre à jour la taille de la liste lors de la suppression d'un élément.

```
/* Supprime le premier nœud de la liste L et restitue sa donnée. */
void* listRemoveFirst(List * L);
```

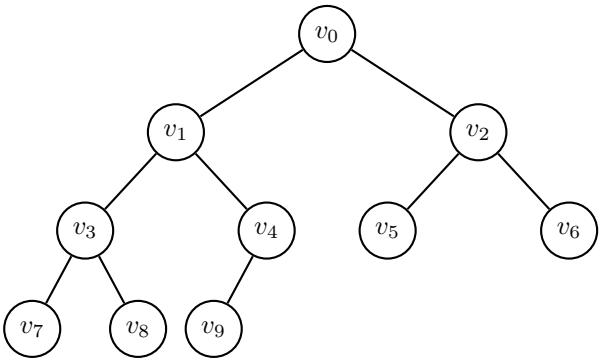
```c
/* Supprime le dernier nœud de la liste L et restitue sa donnée. */
void* listRemoveLast(List * L);

/* Supprime le nœud de la liste L indiqué par le pointeur node
 * et restitue sa donnée. */
void* listRemoveNode(List * L, LNode * node);
```

Exercice 5 Implémentez dans le fichier `src/list.c` la fonction suivantes qui concatène deux listes (le prototype se trouve dans `include/list.h`).

```c
/* Concatène les listes L1 et L2 */
List* listConcatenate(List* L1, List* L2);
```

4 Arbres binaires complets

Un arbre binaire complet (*complete binary tree* en anglais) est un arbre binaire dans lequel tous les niveaux sont remplis à l'exception éventuelle du dernier. En plus, au dernier niveau les feuilles sont alignées à gauche comme vous voyez sur la figure ci-dessous. Il faut toujours insérer un nouveau élément au dernier niveau, le plus à gauche possible. Si le dernier niveau est déjà rempli, on passe au niveau suivant (toujours le plus à gauche possible).



La structure suivante correspond à un nœud d'un arbre binaire (voir `include/tree.h`).

```c
typedef struct TreeNode {
    void * data; // la donnée
    struct TreeNode * left;  // le fils gauche
    struct TreeNode * right; // le fils droit
} TNode;
```

Exercice 6 Implémentez dans le fichier `src/tree.c` les primitives de la structure correspondante au nœud d'un arbre binaire (les prototypes se trouvent dans `include/tree.h`).

```c
// Construction (n'oubliez pas d'allouer de la mémoire)
TNode * newTNode(void *data);

// Consultation
void* getTNodeData(const TNode* node);
TNode* Left(const TNode* node);
TNode* Right(const TNode* node);
```

```
// Modification
void setTNodeData(TNode* node, void* newData);
void setLeft(TNode* node, TNode* newLeft);
void setRight(TNode* node, TNode* newRight);
```

La structure suivante correspond à un Arbre Binaire Complet (voir `include/tree.h`).

```
typedef struct CompleteBinaryTree {
    TNode * root; // pointeur vers la racine
    int numelm;   // nombre d'éléments
    void (*viewData)(const void*); // affiche les données
    void (*freeData)(void*);       // supprime les données
} CBTree;
```

Exercice 7 Implémentez dans le fichier `src/tree.c` les primitives de la structure d'un arbre binaire complet (les prototypes se trouvent dans `include/tree.h`).

```
// Construction (n'oubliez pas d'allouer de la mémoire)
CBTree* newCBTree(void (*viewData)(const void*), void (*freeData)(void
   *));

// Consultation
int treeIsEmpty(CBTree* T);
int getCBTreeSize(const CBTree* T);
TNode* Root(const CBTree* T);

// Modification
void increaseCBTreeSize(CBTree* T);
void decreaseCBTreeSize(CBTree* T);
void resetCBTreeSize(CBTree* T);
void setRoot(CBTree* T, TNode* newRoot);

// Affichage
static void preorder(TNode *node, void (*viewData)(const void*));
static void inorder(TNode *node, void (*viewData)(const void*));
static void postorder(TNode *node, void (*viewData)(const void*));
void viewCBTree(const CBTree* T, int order);

// Libération de la mémoire
static void freeTNode(TNode* node, void (*freeData)(void*));
void freeCBTree(CBTree * T, int deleteData);
```

Notez que les fonctions `static` sont privées dans le fichier sur lequel elles sont déclarées (ici `src/tree.c`), c'est-à-dire nous ne pouvons pas les appeler dehors de ce fichier. Pour cette raison, on n'a pas besoin de définir leurs prototypes dans le fichier .h (ici `include/tree.h`).

Exercice 8 Implémentez dans le fichier `src/tree.c` les sous-programmes suivants qui réalisent l'insertion d'un nouveau élément dans un arbre binaire complet (les prototypes se trouvent dans `include/tree.h`). La fonction `insertAfterLastTNode` doit être implémentée de façon récursive, tandis que la procédure `CBTreeInsert` doit lancer la récursion. Le paramètre `position` indique la première position disponible de l'arbre sur laquelle le nouveau élément va être inséré. Les positions de l'arbre sont numérotées par niveau et de gauche à droite.

```
/* Numérotation des positions d'un arbre binaire complet :
 *           0
 *         /   \
 *       1       2
 *      / \     / \
 *     3   4   5   6
 *    / \
 *   7  ...
 *
 * Insertion au dernier niveau et le plus à gauche possible
 * dans l'arbre T un nouveau nœud de donnée data */
static TNode* insertAfterLastTNode(TNode* node, int position, void*
   data);
void CBTreeInsert(CBTree* T, void* data)
```

Exercice 9 Implémentez dans le fichier `src/tree.c` les fonctions suivantes qui suppriment un élément de l'arbre `T` (les prototypes se trouvent dans `include/tree.h`). La fonction `removeLastTNode` doit être implémentée de façon récursive, tandis que la fonction `CBTreeRemove` doit lancer la récursion. Le paramètre `position` indique la position du dernier élément de l'arbre, c'est-à-dire de l'élément qui va être supprimé.

```
/* Supprime le dernier nœud de l'arbre T et restitue sa donnée.
 * La mémoire du nœud supprimé est libérée mais pas la mémoire
 * de la donnée. */
static TNode* removeLastTNode(TNode* node, int position, void** data);
void* CBTreeRemove(CBTree* T);
```

Exercice 10 Implémentez dans le fichier `src/tree.c` les fonctions suivantes qui trouvent le dernier élément de l'arbre `T` (les prototypes se trouvent dans `include/tree.h`). La fonction `getLastTNode` doit être implémentée de façon récursive, tandis que la fonction `CBTreeGetLast` doit lancer la récursion. Le paramètre `position` indique la position du dernier élément de l'arbre, c'est-à-dire de l'élément qu'on cherche.

```
/* Restitue le dernier nœud de l'arbre T. */
static TNode* getLastTNode(TNode* node, int position);
TNode* CBTreeGetLast(CBTree* T);
```

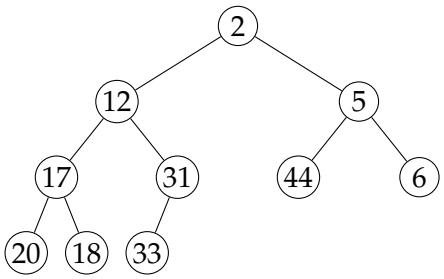
Exercice 11 Implémentez dans le fichier `src/tree.c` la procédure suivante qui échange les données de deux nœud d'un arbre donnés en paramètre (le prototype se trouve dans `include/tree.h`).

```
/* Permute les données des nœuds node1 et node2. */
void CBTreeSwapData(TNode* node1, TNode* node2);
```

5 Tas

Un arbre binaire complet peut être représenté par un tableau, sans avoir besoin d'une structure avec des nœuds et des pointeurs (fils gauche/droit). Une telle représentation est donnée dans la figure ci-dessous où l'arbre binaire complet est transformé à tableau en le parcourant niveau par niveau et

de gauche à droite. Pour un nœud représenté par la position i du tableau, son fils gauche se trouve dans la position $2i + 1$, son fils droit dans la position $2i + 2$ et son père dans la position $(i - 1)/2$.



0	1	2	3	4	5	6	7	8	9
2	12	5	17	31	44	6	20	18	33

Dans ce projet, on va utiliser les deux représentations d'un arbre binaire complet afin d'implémenter deux différentes structures de tas.

L'idée principale de l'implémentation d'un tas basé sur un arbre binaire complet est d'appliquer deux opérateurs de correction récursifs (vers le bas et vers le haut) après chaque modification du tas (ajout/suppression d'un élément). Pour plus d'information sur ces deux opérateurs, veuillez regarder la présentation du projet et les notes du cours de AP3.

5.1 Tas représenté par un tableau

La structure suivante correspond à un tas basé sur un arbre binaire complet représenté par un tableau de taille fixe (voir `include/heap.h`).

```
typedef struct {
    void** A;  // le tableau
    int MAX;   // taille du tableau
    int N;     // nb d'éléments actuellement dans le tableau
    int (*preceed)(const void*, const void*); // compare deux éléments
    void (*viewHeapData)(const void*);        // affiche les données
    void (*freeHeapData)(void*);              // supprime les données
} ArrayHeap;
```

Exercice 12 Implémentez dans le fichier `src/heap.c` les primitives de la structure d'un tas basé sur un arbre binaire complet représenté par un tableau de taille fixe (les prototypes se trouvent dans `include/heap.h`).

```
// Consultation
int getAHMaxSize(const ArrayHeap* AH);
int getAHActualSize(const ArrayHeap* AH);
void* getAHNodeAt(const ArrayHeap* AH, int pos);

// Modification
void decreaseAHActualSize(ArrayHeap* AH);
void setAHNodeAt(ArrayHeap* AH, int position, void* newData);

// Affichage
void viewArrayHeap(const ArrayHeap* AH);

// Libération de la mémoire
void freeArrayHeap(ArrayHeap* AH, int daletedata);
```

Exercice 13 Implémentez dans le fichier `src/heap.c` l'opérateur "correction vers le bas" de la structure d'un tas basé sur un arbre binaire complet représenté par un tableau de taille fixe.

```
/* Corrige la position de l'élément de la position pos du tas AH
 * en le comparant avec ses fils et en l'échangeant avec
 * le fils de la plus grande priorité si nécessaire */
static void updateArrayHeapDownwards(ArrayHeap* AH, int pos);
```

Exercice 14 Implémentez dans le fichier `src/heap.c` la fonction suivante qui transforme un tableau quelconque à un tas basé sur un arbre binaire complet représenté par un tableau de taille fixe (le prototype se trouve dans `include/heap.h`). Il s'agit du constructeur de ce type de tas.

```
/* Transforme le tableau A à un tas en réorganisant ses éléments
 * selon le pointeur de fonction preceed. */
ArrayHeap* ArrayToArrayHeap(void** A, int N,
                    int (*preceed)(const void*, const void*),
                    void (*viewHeapData)(const void*),
                    void (*freeHeapData)(void*));
```

Exercice 15 Implémentez dans le fichier `src/heap.c` la fonction suivante qui extrait l'élément avec la plus grande priorité d'un tas basé sur un arbre binaire complet représenté par un tableau de taille fixe (le prototype se trouve dans `include/heap.h`).

```
/* Supprime du tas AH l'élément avec la plus grande priorité.
 * La mémoire de l'élément n'est pas libérée.
 * Au contraire, l'élément est restitué à la fin de la fonction. */
void* ArrayHeapExtractMin(ArrayHeap* AH);
```

5.2 Tas représenté par des pointeurs

La structure suivante correspond à un tas basé sur un arbre binaire complet représenté par une structure avec des pointeurs (voir `include/heap.h`).

```
typedef struct {
    CBTree* T;
    int (*preceed)(const void*, const void*);
    void (*viewHeapData)(const void*);
    void (*freeHeapData)(void*);
}  CBTHeap;
```

Exercice 16 Implémentez dans le fichier `src/heap.c` les primitives de la structure d'un tas basé sur un arbre binaire complet représenté par une structure avec des pointeurs (les prototypes se trouvent dans `include/heap.h`).

```
// Construction
CBTHeap* newCBTHeap(int (*preceed)(const void*, const void*),
                    void (*viewHeapData)(const void*),
                    void (*freeHeapData)(void*));

// Consultation
CBTree* getCBTree(const CBTHeap* H);
```

```c
// Affichage
void viewCBTHeap(const CBTHeap* H);

// Libération de la mémoire
void freeCBTHeap(CBTHeap* H, int deletenode);
```

Exercice 17 Implémentez dans le fichier `src/heap.c` les opérateurs de correction (vers le bas et vers le haut) de la structure d'un tas basé sur un arbre binaire complet représenté par une structure avec des pointeurs.

```c
/* Corrige la position du nœud à la position pos de l'arbre raciné à
 * node en le comparant avec son père et en l'échangeant avec lui
 * si nécessaire. */
static void updateCBTHeapUpwards(TNode* node, int pos, int (*preceed)(
   const void*, const void*));

/* Corrige la position du nœud node en le comparant avec ses fils
 * et en l'échangeant avec le fils de la plus grande priorité
 * si nécessaire. */
static void updateCBTHeapDownwards(TNode* node, int (*preceed)(const
   void*, const void*));
```

Exercice 18 Implémentez dans le fichier `src/heap.c` l'opération de l'insertion d'un nouveau élément dans un tas basé sur un arbre binaire complet représenté par une structure avec des pointeurs (les prototypes se trouvent dans `include/heap.h`).

```c
/* Insère dans le tas H un nouveau nœud de donnée data. */
void CBTHeapInsert(CBTHeap *H, void* data);
```

Exercice 19 Implémentez dans le fichier `src/heap.c` l'opération de l'extraction de l'élément avec la plus grande priorité d'un tas basé sur un arbre binaire complet représenté par une structure avec des pointeurs (les prototypes se trouvent dans `include/heap.h`).

```c
/* Supprime du tas H l'élément avec la plus grande priorité.
 * La mémoire de l'élément n'est pas libérée.
 * Au contraire, l'élément est restitué à la fin de la fonction. */
void* CBTHeapExtractMin(CBTHeap *H);
```

6 Tri

Exercice 20 Implémentez dans le fichier `src/sort.c` les trois sous-programmes qui réalisent les trois algorithmes de tri de ce projet (les prototypes se trouvent dans `include/sort.h`).

```c
/* Tri par tas (implémenté par un tableau). */
void ArrayHeapSort(void** A, int N,
                   int (*preceed)(const void*, const void*),
                   void (*viewHeapData)(const void*),
                   void (*freeHeapData)(void*));
```

```c
/* Tri par tas (implémenté par un arbre binaire complet). */
void CBTHeapSort(void** A, int N,
                 int (*preceed)(const void*, const void*),
                 void (*viewHeapData)(const void*),
                 void (*freeHeapData)(void*));

/* Tri par sélection (tri par échange). */
void SelectionSort(void** A, int N, int (*preceed)(const void*, const
   void*));
```

7 Géométrie

La structure suivante correspond à un point (voir `include/geometry.h`).

```c
typedef struct {
    long long int x;  // abscisse
    long long int y;  // ordonnée
} Point;
```

Exercice 21 Implémentez dans le fichier `src/geometry.c` les primitives de la structure d'un point (les prototypes se trouvent dans `include/geometry.h`).

```c
// Construction
Point* newPoint(long long int x, long long int y);

// Consultation
long long int X(const Point* P);
long long int Y(const Point* P);

// Affichage
void viewPoint(const void* P);

// Libération de la mémoire
void freePoint(void* P);
```

Exercice 22 Implémentez dans le fichier `src/geometry.c` les quatre fonctions suivantes qui décident si un point se trouve à droite ou à gauche d'une droite orientée / est colinéaire avec deux autres points / est inclus dans un segment, respectivement (les prototypes se trouvent dans `include/geometry.h`). Afin de décider le trois premières question, il faut calculer le signe de l'expression suivante :

$$(X_B - X_A) * (Y_P - Y_A) - (Y_B - Y_A) * (X_P - X_A)$$

```c
int onRight(const Point* origin, const Point* destination, const Point
   * P);
int onLeft(const Point* origin, const Point* destination, const Point*
    P);
int collinear(const Point* origin, const Point* destination, const
   Point* P);
int isIncluded(const Point* origin, const Point* destination, const
   Point* P);
```

La structure suivante correspond à une arête orientée - arc (voir `include/geometry.h`).

```
typedef struct DirectedEdge{
    Point* origin;  // début de l'arc
    Point* destination;  // fin de l'arc
} DEdge;
```

Exercice 23 Implémentez dans le fichier `src/geometry.c` les primitives de la structure d'un arc (les prototypes se trouvent dans `include/geometry.h`).

```
// Construction
DEdge* newDEdge(Point* origin, Point* destination);

// Consultation
Point* getOrigin(const DEdge* DE);
Point* getDestination(const DEdge* DE);

// Affichage
void viewDEdge(const void* DE);

// Libération de la mémoire
void freeDEdge(void* DE);
```

8 Enveloppe convexe

La dernière phase de la partie en C consiste à écrire le code pour les trois algorithmes qui calculent l'enveloppe convexe présentés au cours. Avant faire cela, on a besoin de implémenter deux fonctions auxiliaires qui vont lire/écrire un ensemble de points de/à un fichier. Notez bien que la communication entre notre code en C et l'interface graphique qui sera implémenté en Java se fait en utilisant ces fichiers avec le format suivant :

— la première ligne contient le nombre de points du fichier, et
— chaque ligne différente de la première contient l'abscisse et l'ordonnée d'un seul point.

Un exemple avec 12 points est donné ci-dessous.

```
12
0 5
4 7
5 10
3 6
7 6
4 3
10 5
6 7
7 4
6 3
3 4
5 0
```

Exercice 24 Implémentez dans le fichier `src/algo.c` les deux procédures suivantes qui respectivement écrit/lit un ensemble de points de/à fichier.

```
/* Réalise la lecture d'un ensemble des points à partir du fichier
 * filename et crée un tableau avec ces points. */
static Point** readInstance(const char* filename, int* N) {

/* Réalise l'écriture de l'ensemble des points de la liste L
 * à un fichier avec le nom filename. */
static void writeSolution(const char* filename, List* L) {
```

Exercice 25 Implémentez dans le fichier `src/algo.c` le premier algorithme (glouton) pour calculer l'enveloppe convexe (les prototypes se trouvent dans `include/algo.h`). La fonction auxiliaire `DedgesToClockwisePoints` transforme une liste d'arcs à une liste de points et sera utilisée à la fin de cet algorithme glouton.

```
/* Transforme une liste d'arcs décrivant les arêtes du polygone
 * de l'enveloppe convexe à une liste des points ordonnés
 * dans le sens horaire. */
static List * DedgesToClockwisePoints(List* dedges);

/* Calcule l'enveloppe convexe d'un ensemble des points en 2D
 * en utilisant un algorithme glouton qui examine tous les couples
 * de points. */
void SlowConvexHull(const char* infilename, const char* outfilename);
```

Exercice 26 Implémentez dans le fichier `src/algo.c` le deuxième algorithme (itératif, basé sur le tri) pour calculer l'enveloppe convexe (les prototypes se trouvent dans `include/algo.h`). Les fonctions auxiliaires `smallerPoint` et `biggerPoint` comparent deux points par rapport à leur position. Un point a précède un point b, si l'abscisse de a est plus petite que l'abscisse de b. Dans le cas d'égalité des abscisses, le point a précède le point b si l'ordonnée de a est plus petite que l'ordonnée de b.

```
/* Compare le points a et b et renvoie 1 si a précède b
 * ou 0 si b précède a. */.
static int smallerPoint(const void* a, const void* b);

/* Compare le points a et b et renvoie 1 si b précède a
 * ou 0 si a précède b. */
static int biggerPoint(const void* a, const void* b

/* Calcule l'enveloppe convexe d'un ensemble des points en 2D
 * en utilisant un algorithme itératif basé sur un tri. */
void ConvexHull(const char* infilename, const char* outfilename, int
   sortby);
```

Exercice 27 Implémentez dans le fichier `src/algo.c` le troisième algorithme (itérant sur les points de l'enveloppe convexe) pour calculer l'enveloppe convexe (le prototype se trouve dans `include/algo.h`).

```
/* Calcule l'enveloppe convexe d'un ensemble des points en 2D
 * en utilisant l'algorithme rapide qui travaille sur les points
 * de l'enveloppe convexe. */
void RapidConvexHull(const char* infilename, const char* outfilename);
```

9 Évaluation de performance d'algorithmes et de structures de données

Exercice 28 Réfléchissez sur la complexité asymptotique des trois algorithmes de tri que vous avez implémentez dans ce projet et remplissez le tableau suivant. La réponse à cette question doit apparaître à votre rapport avec une petite justification d'une ou deux phrases (sans calculer le nombre d'instructions exact).

Algorithme	Complexité
tri par sélection	
tri par tas (tableau)	
tri par tas (arbre)	

Exercice 29 Comparez le temps d'exécution des trois algorithmes de tri en utilisant les données fournies. Désignez des diagrammes pour visualiser cette comparaison. Quelle implémentation auriez vous choisi et pourquoi ? La réponse à cette question doit apparaître à votre rapport ainsi que la présentation des résultats de vos expériences.

Exercice 30 Réfléchissez sur la complexité asymptotique des trois algorithmes pour calculer l'enveloppe convexe que vous avez implémentez dans ce projet et remplissez le tableau suivant. La réponse à cette question doit apparaître à votre rapport avec une petite justification d'une ou deux phrases (sans calculer le nombre d'instructions exact). Dans l'algorithme `ConvexHull` utilisez le meilleur algorithme de tri selon votre réponse à l'Exercice 28.

Algorithme	Complexité
`SlowConvexHull`	
`ConvexHull`	
`RapidConvexHull`	

Exercice 31 Comparez le temps d'exécution des trois algorithmes pour calculer l'enveloppe convexe en utilisant les données fournies. Désignez des diagrammes pour visualiser cette comparaison. Quelle implémentation auriez vous choisi et pourquoi ? La réponse à cette question doit apparaître à votre rapport ainsi que la présentation des résultats de vos expériences. Dans l'algorithme `ConvexHull` utilisez le meilleur algorithme de tri selon votre réponse à l'Exercice 29.

10 Interface graphique

La deuxième partie du projet consiste à implémenter une interface graphique simple pour visualiser les résultats des algorithmes en utilisant JavaFX. L'interface graphique donnera à l'utilisateur les options suivantes :

- Donner l'entrée du problème de l'enveloppe convexe (un ensemble des points) avec deux différentes façons :
  - choisir un fichier contenant les points
  - créer un ensemble des points en cliquant avec la souris sur une fenêtre de l'interface et enregistrer ces points dans un fichier
- Choisir l'algorithme qui calcule l'enveloppe convexe (1–SlowCOnvexHull, 2–ConvexHull, 3–RapidConvexHull)
- Choisir l'algorithme de tri dans le cas où le choix 2–ConvexHull est sélectionné avant (1–tri par tas avec arbres, 2–tri par tas avec tableau, 3–tri par sélection)

— Appeler le bon sous-programme qui calcule l'enveloppe convexe implémenté en C avec les paramètres choisis (fichier d'entrée, algorithme de tri si besoin) ainsi que le nom du fichier où la solution va être enregistrée
— afficher les points d'entrée ainsi que la solution (par exemple, en désignant le polygone qui correspond à l'enveloppe convexe)

11 Dépôt et Évaluation

11.1 Le dépôt

Le projet sera rendu en deux parties. La première partie (date limite : vendredi, 29 avril, 23h59) devra contenir :

1. le répertoire `.git/`
2. le totalité du code en C (fichiers .h et .c, Makefile) jusqu'à l'exercice 27 (incluse)
3. un README (facultatif) : comment lancer le programme C, explications divers, etc

La deuxième partie (date limite : vendredi, 20 mai, 23h59) devra contenir :

1. le répertoire `.git/`
2. le code en C (fichiers .h et .c, Makefile)
3. le code en Java (fichiers .java, Makefile)
4. un rapport de maximum 5 pages (format pdf)
5. un README : comment compiler/lancer l'interface Java

Pour chaque rendu vous devrez soumettre un fichier compressé par binôme contenant les points ci-dessus. Aucune prolongation ne sera accordée. Dans le cas de dépassement de la date limite, vous aurez une pénalité d'un point par jour de retard.

11.2 Soutenance

Chaque équipe doit aussi présenter son projet pendant environ 30 minutes (questions incluses). Tous les membres de l'équipe participeront dans cette présentation. La note de cette présentation peut être différente pour les membres de la même équipe. Les présentations auront lieux la semaine du 23 mai pendant les heures marquées comme TD sur l'emploi du temps. Le programme des soutenances sera annoncé ultérieurement au cours du semestre.

11.3 Évaluation

Il n'y a pas d'examen terminal écrit. Veuillez trouver ci-dessous un barème indicatif :
— Utilisation du GIT : 5%
— Utilisation des primitives : 5%
— Listes doublement chaînées : 10%
— Arbres binaires complets : 10%
— Tas : 10%
— Tri : 10%
— Géométrie : 5%
— Enveloppe convexe : 10%
— Interface graphique + JNI : 15%
— Rapport : 12.5%
— Soutenance : 12.5%

Il n'y aura pas d'épreuve de Seconde Chance.

11.4 Consignes pour le rapport

Le rapport final ne doit pas dépasser 5 pages en total. Le rapport n'est pas une documentation du code et doit impérativement contenir une introduction et une section des conclusions. Par exemple, vous pouvez expliquer :

— Quels outils vous avez utilisé ?
— Quelles fonctions supplémentaires vous avez introduit et pourquoi ?
— Quelle structure de données fonctionne le mieux en termes de temps d'exécution pour de petits instances ? Pour des instances moyennes ? Pour des grandes instances ? Pourquoi ?
— Quelles étaient les difficultés que vous avez rencontrées ?
— Quelles améliorations envisageriez vous ?