Projet de synthèse

Giorgio Lucarelli
giorgio.lucarelli@univ-lorraine.fr

Stéfane Paris
stephane.paris@univ-lorraine.fr

Janvier MMXX

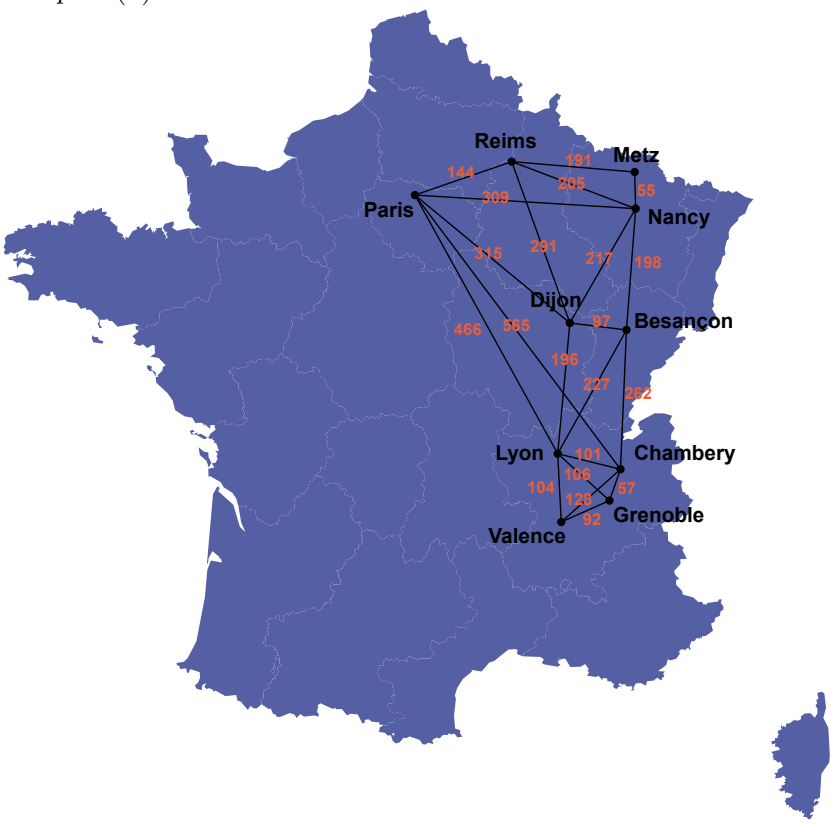
UNIVERSITÉ DE LORRAINE | UFR MATHÉMATIQUES INFORMATIQUE MÉCANIQUE ET AUTOMATIQUE

Le plus court chemin – Algorithme de Dijkstra

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 équipes de 2 ou 3 personnes qui appartiennent au même groupe TP. Pour des raisons d'organisation, **aucune exception de cette règle ne sera toléré.**

Dans ce projet, vous allez implémenter l'algorithme proposé par Edsger Dijkstra en 1959 pour trouver *les plus court chemins* dans un graphe. Dans le problème du plus court chemin, l'entrée est un graphe simple non-orienté $G = (V, E)$, une fonction de pondération $w : E \to \mathbb{N}$ sur les arrêtes du graphe et un sommet $s \in V$ (appelé *départ*). Le poids d'un chemin entre deux sommets $u, v \in V$ est égal à la somme de poids des arrêtes qui forment ce chemin. L'objectif est de trouver le chemin de poids minimum (appelé *le plus court chemin*) entre le sommet de départ s et tout sommet $v \in V$, $v \neq s$. Pour chaque sommet $v \in V$, la sortie est le poids du plus court chemin $dist(v)$ (distance) de s à v, ainsi que son prédécesseur $pred(v)$ dans ce chemin.



Par exemple, ils existent plusieurs chemins pour aller de Metz à Lyon, passant par Dijon, par Besançon ou même par Paris. Par contre, le plus court chemin est Metz–Nancy–Dijon–Lyon de coût total égal à 468.

1.1 Comment commencer ?

Pour vous aider à organisation 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 ainsi que 8 fichiers `.c` qui sont à modifier afin de réaliser le projet.

> **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 commencer à coder, n'oubliez pas à créer un projet GIT par équipe sur `https://about.gitlab.com/`. 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 est finie et testée ou si un bogue est 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 pour l'algorithme de Dijkstra

L'algorithme de Dijkstra a besoin d'une *file de priorité* pour conserver les sommets qui ne sont pas encore étudiés avec l'ordre qu'il va les visiter. La priorité de chaque sommet correspond à sa distance actuelle depuis le sommet de départ : plus petite distance signifie plus grande priorité. Les primitives de cette file de priorité sont les suivantes :

insertion **:** ajouter un nouveau sommet à la file dans la bonne position (servi pour l'initialisation de la file de priorité).

extraction du sommet avec la plus grande priorité **:** retirer de la file le sommet avec la plus petite distance actuelle depuis le sommet de départ afin de traiter ses voisins.

incrémentation de la priorité d'un nœud donné **:** diminuer la distance actuelle depuis le sommet de départ (augmenter la priorité) d'un sommet donné.

Afin d'implémenter cette file de priorité, on va utiliser 3 différentes structures de données et comparer leur performance au niveau théorique et pratique. Les structures de données qui nous intéressent sont les suivantes :

1. liste doublement chaînée (*double linked list*)
2. arbre binaire complet (*complete binary tree*) : un arbre binaire dans lequel tous les niveaux sont remplis à l'exception éventuelle du dernier, dans lequel les feuilles sont alignées à gauche (il doit être rempli de gauche à droit).
3. tableau dynamique (*dynamic table*) : un tableau dont la taille change en fonction du nombre d'éléments qui contient.

L'objectif initial de ce projet et d'implémenter en langage C les trois structures de données ci-dessus. En suite, en utilisant chacune de ces structures de données, on va implémenter un tas.

2 Listes doublement chaînées

Vous avez déjà implémenté une liste doublement chaînée au cours "Programmation Avancée" du semestre précédent. Nous allons donc utiliser cette implémentation et ajouter quelques fonctions nécessaires pour notre projet. En suite, on vous donne une description du code fourni ainsi que les fonctions que vous devez coder en forme des exercices.

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

```
typedef struct ListNode {
    void * data;
    struct ListNode * suc, * pred;
} LNode;
```

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

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

Noter 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 les éléments d'un tas, les sommets d'un graphe, les voisins d'un sommet du graphe, etc.

Les fonctions suivantes sont fournies (voir `include/list.h` pour les prototypes et `src/list.c` pour leur code).

```
/* Construire une liste vide */
List * newList ();

/* Détruire la liste L
 * en utilisant ptrF pour détruire chaque élément */
void deleteList(List * L, void (*ptrF)());

/* Afficher les éléments de la liste L
 * en utilisant ptrF pour afficher chaque élément */
void viewList(const List * L, void (*ptrF) ());

/* Ajouter en tête de la liste L un élément de donnée data */
void listInsertFirst(List * L, void * data);

/* Ajouter à la fin de la liste L un élément de donnée data */
void listInsertLast(List * L, void * data);

/* Insérer un élément de liste de donnée data dans la liste L
 * après le pointeur d'élément ptrelm */
void listInsertAfter(List * L, void * data, LNode * ptrelm);
```

Exercice 1 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.

```
/* Supprimer le premier élément de la liste L.
```

```
 * Assurez vous que la liste n'est pas vide. */
LNode* listRemoveFirst(List * L);

/* Supprimer le dernier élément de la liste L.
 * Assurez vous que la liste n'est pas vide. */
LNode* listRemoveLast(List * L);

/* Supprimer l'élément pointé par node de la liste L.
 * L'élément est supposé appartenir effectivement à la liste. */
LNode* listRemoveNode(List * L, LNode * node);
```

Exercice 2 Implémentez dans le fichier `src/list.c` la fonction suivante qui échange l'ordre des nœuds `left` et `right` dans la liste `L` (le prototype se trouve dans `include/list.h`). Assurez vous que `right` est bien le successeur du `left`. Attention! vous devez échanger les nœuds `LNode*` et pas seulement leurs données (modifier les pointeurs).

```
/* Permuter les positions des nœuds left et right dans la liste L. */
void listSwap(List * L, LNode * left, LNode *right);
```

3 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). Afin d'éviter chaque fois à chercher la bonne position pour la prochaine insertion et payer beaucoup en terme de complexité, il nous faut un pointeur vers l'élément ajouté en dernier (par exemple, avoir un pointeur vers v_9 sur la figure).

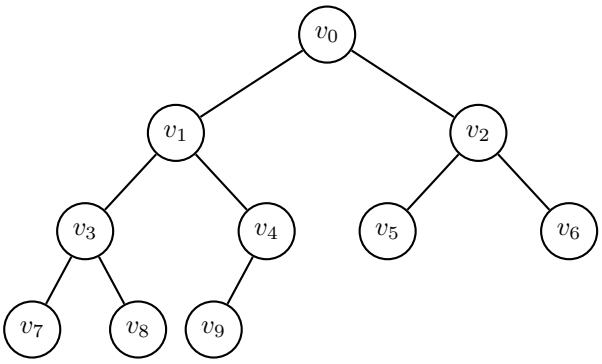


FIGURE 1 – Arbre binaire complet

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

```
typedef struct TreeNode {
    void * data;
    struct TreeNode * left;   // pointeur vers le fils gauche
    struct TreeNode * right;  // pointeur vers le fils droit
    struct TreeNode * parent; // pointeur vers le père
} TNode;
```

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

```c
typedef struct CompleteBinaryTree {
    TNode * root;  // pointeur vers la racine
    TNode * last;  // pointeur vers le dernier élément
    int numelm;    // nombre d'éléments
} CBTree;
```

Exercice 3 Implémentez dans le fichier `src/tree.c` la fonction suivante qui crée un nouveau nœud de l'arbre avec la donnée `data` (le prototype se trouve dans `include/tree.h`). N'oubliez pas de faire l'allocation mémoire pour ce nouveau nœud en utilisant la fonction `malloc` et d'initialiser tous les pointeurs de la structure. La fonction renvoie le nouveau nœud créé.

```c
/* Construire et initialiser un nouveau noeud d'un arbre binaire.
 * Renvoie le nouveau noeud créé. */
TNode * newTNode(void* data);
```

Exercice 4 Implémentez dans le fichier `src/tree.c` la fonction suivante qui crée un nouveau arbre binaire complet vide (le prototype se trouve dans `include/tree.h`).

```c
/* Construire un arbre vide */
CBTree * newCBTree();
```

Pour afficher les éléments d'un arbre binaire, il faut d'abord choisir l'ordre selon lequel on va les visiter. Nous avons trois options :

ordre préfixe : père – fils gauche – fils droit (Figure 1 : $v_0, v_1, v_3, v_7, v_8, v_4, v_9, v_2, v_5, v_6$)

ordre postfixe : fils gauche – fils droit – père (Figure 1 : $v_7, v_8, v_3, v_9, v_4, v_1, v_5, v_6, v_2, v_0$)

ordre infixe : fils gauche – père – fils droit (Figure 1 : $v_7, v_3, v_8, v_1, v_9, v_4, v_0, v_5, v_2, v_6$)

Les fonctions suivantes visitent les nœuds de l'arbre binaire complet et affichent son contenu en utilisant ptrF. Notez que les trois fonctions sont déclarée `static` et elles sont privées dans le fichier `src/tree.c`, c'est-à-dire nous ne pouvons pas les appeler dehors du fichier `src/tree.c`. Pour cette raison, on n'a pas besoin de définir leurs prototypes dans le fichier `include/tree.h`.

```c
static void preorder(TNode *node, void (*ptrF)(const void*))
static void inorder(TNode *node, void (*ptrF)(const void*))
static void postorder(TNode *node, void (*ptrF)(const void*))
```

La fonction suivante affiche les éléments de l'arbre binaire `tree` selon l'ordre `order` en utilisant la fonction `ptrF` (le prototype se trouve dans `include/tree.h`).

```c
/* Afficher les éléments de l'arbre.
 * Chaque élément est affiché grace au pointeur de fonction ptrF.
 * L'attribut order spécifie l'ordre d'affichage :
 * (+) 0 = ordre préfixe
 * (+) 1 = ordre postfixe
 * (+) 2 = ordre infixe */
void viewCBTree(const CBTree* tree, void (*ptrF)(const void*), int
   order);
```

Exercice 5 Implémentez dans le fichier `src/tree.c` la fonction suivante qui réalise l'insertion d'un nouveau nœud avec la donnée `data` dans l'arbre `tree` (le prototype se trouve dans `include/tree.h`). Utilisez la fonction `TNode * newTNode(void* data)` pour la création du nouveau nœud. Le nouveau nœud doit être ajouter au dernier niveau de l'arbre le plus à gauche possible, afin que l'arbre reste complet. N'oubliez pas à mettre à jour le pointeur `last`.

```c
/* Ajouter dans l'arbre binaire complet tree
 * un élément de donnée data */
void CBTreeInsert(CBTree* tree, void* data);
```

Exercice 6 Implémentez dans le fichier `src/tree.c` la fonction suivante qui réalise la suppression du nœud indiqué par le pointeur `last` de l'arbre `tree` (le prototype se trouve dans `include/tree.h`). La fonction renvoie la donnée du nœud supprimé. N'oubliez pas à mettre à jour le pointeur `last`.

```c
/* Supprimer l'élément indiqué par le pointeur last de l'arbre tree.
 * Renvoie la donnée du nœud supprimé. */
void * CBTreeRemove(CBTree* tree);
```

Exercice 7 Implémentez dans le fichier `src/tree.c` la fonction suivante qui échange l'ordre des nœuds `parent` et `child` dans l'arbre `tree` (le prototype se trouve dans `include/tree.h`). Assurez vous que `child` est bien un fils de `parent`. Attention ! vous devez échanger les nœuds `TNode*` et pas seulement leurs données (il faut modifier les pointeurs `pred` et `suc`).

```c
/* Permuter les positions des nœuds parent et child dans l'arbre tree.
 */
void CBTreeSwap(CBTree* tree, TNode* parent, TNode* child);
```

Exercice 8 Implémentez dans le fichier `src/tree.c` la fonction suivante qui échange l'ordre des nœuds indiqués par les pointeurs `root` et `last` de l'arbre `tree` (le prototype se trouve dans `include/tree.h`). Attention ! vous devez échanger les nœuds `TNode*` et pas seulement leurs données (il faut modifier les pointeurs `pred` et `suc`).

```c
/* Permuter les nœuds root et last de l'arbre tree. */
void CBTreeSwapRootLast(CBTree* tree);
```

4 Tableaux dynamiques

La taille d'un tableau classique en C est définie dans sa définition et n'est pas modifiable. Par contre, dans certaines applications nous avons besoin de tableaux de taille qui évolue pendant l'exécution de l'algorithme. Dans ce contexte, nous allons implémenter une structure de données pour représenter un tableau de taille dynamique. La taille va être toujours de forme 2^k, pour $k \in \mathbb{N}$. Pour de raisons de complexité, on impose que le tableau doit avoir au moins taille$/4$ cases remplies.

La structure suivante correspond à un Tableau Dynamique (voir `include/dyntable.h`).

```c
/* Le tableau dynamique est une structure contenant
 * (+) Un tableau (T) de pointeurs vers de données de type void,
 * (+) La taille (size) du tableau,
 * (+) Le nombre de cases actuellement utilisées */
typedef struct DynamicTable {
    void **T;
    int size;
    int used;
} DTable;
```

Exercice 9 Implémenter dans les fichier `src/dyntable.c` la fonction suivante qui crée un nouveau tableau dynamique vide de taille 1 (le prototype se trouve dans `include/dyntable.h`). N'oubliez pas de faire l'allocation memoire en utilisant la fonction `malloc` et d'initialiser toutes les variables de la structure.

```
/* Construire un tableau dynamique */
DTable * newDTable();
```

Exercice 10 Implémenter dans les fichier `src/dyntable.c` les fonctions suivantes qui redimensionnent un tableau dynamique. Vous pouvez utiliser la fonction `realloc`. Notez que les deux fonctions sont définies comme `static` : on ne peut pas les appeler en dehors du fichier `src/dyntable.c`.

```
/* Dédoubler la taille du tableau dtab */
static void scaleUp(DTable *dtab)

/* Diviser par 2 la taille du tableau dtab */
static void scaleDown(DTable *dtab)
```

Pour afficher les données d'un tableau dynamique, nous vous donnons la fonction suivante qui utilise le pointeur de fonctions `ptrF` (le prototype se trouve dans `include/dyntable.h`).

```
/* Afficher les éléments du tableau dynamique dtab.
 * Chaque élément est affiché grace au pointeur de fonction ptrF. */
void viewDTable(const DTable * dtab, void (*ptrF)(const void*));
```

Exercice 11 Implémentez dans le fichier `src/dyntable.c` la fonction suivante qui réalise l'insertion d'un nouveau élément avec la donnée `data` dans la première case vide du tableau dynamique `dtab` (le prototype se trouve dans `include/dyntable.h`). Si le tableau est déjà plein, il faut utiliser la fonction `scaleUp` avant l'insertion.

```
/* Ajouter dans le tableau dynamique dtab un élément de donnée data.
 */
void DTableInsert(DTable * dtab, void * data);
```

Exercice 12 Implémentez dans le fichier `src/dyntable.c` la fonction suivante qui réalise la suppression de l'élément qui se trouve dans la dernière case utilisée du tableau dynamique `dtab` (le prototype se trouve dans `include/dyntable.h`). La fonction renvoie la donnée de l'élément supprimé. Dans le cas où le nombre des cases remplies après la suppression est inférieur ou égale à `size`/4, n'oubliez pas de redimensionner le tableau en utilisant la fonction `scaleDown`.

```
/* Supprimer l'élément dans la position used-1 (dernier élément)
 * du tableau dynamique dtab. */
void * DTableRemove(DTable * dtab);
```

Exercice 13 Implémentez dans le fichier `src/dyntable.c` la fonction suivante qui échange le contenu des positions `pos1` et `pos2` du tableau dynamique `dtab` (le prototype se trouve dans `include/dyntable.h`).

```
/* Permuter la donnée des positions pos1 et pos2 dans le tableau
 * dynamique dtab. */
void DTableSwap(DTable * dtab, int pos1, int pos2);
```

5 Tas minimier

En suite, nous allons implémenter trois différents *tas minimier* basé sur les structures de données présentées ci-dessus. Un tas est appelé *minimier* si l'élément avec la plus grande priorité corresponde à l'élément avec la plus petite valeur (au contraire, dans un tas maximier l'élément avec la plus grande priorité corresponde à l'élément avec la plus grande valeur). Dans ce projet, la priorité sera définie par rapport à la distance actuelle entre le sommet de départ et le sommet en considération.

La structure suivante correspond à un nœud d'un tas (voir `include/heap.h`).

```
/* Un élément du tas contient
 * (+) Une donnée (data)
 * (+) Sa valeur (priorité)
 * (+) Sa position au dictionnaire (à utiliser seulement avec DTHeap)
 */
typedef struct HeapNode {
    int value;
    void *data;
    void *ptr;
} HNode;
```

La structure suivante correspond à un Tas (voir `include/heap.h`). Comme expliqué ci-dessus, un tas est une structure de données pour implémenter une file de priorité et offre trois opérations (primitives) :

— *insertion* d'un élément dans le tas
— *extraction de l'élément avec la plus grande priorité* de tas
— *incrémentation de la priorité* d'un élément du tas

On va utiliser de pointeurs de fonctions pour définir de façon générique ces trois primitives, ainsi que la fonction d'affichage d'un tas.

```
/* Le tas est une structure contenant
 * (+) La structure de données heap qui implémente le tas,
 * (+) Le dictionnaire dict qui pointe vers la position actuelle
 *     de chaque ville à la structure de données DTHeap
 *     (à utiliser seulement avec DTHeap),
 * (+) Les pointeurs des 3 fonctions génériques implémentant
 *     les primitives du tas:
 *     (-) HeapInsert
 *     (-) HeapExtractMin
 *     (-) HeapIncreasePriority
 * (+) Le pointeur de la fonction générique qui affiche
 *     le tas (viewHeap).
 */
typedef struct Heap {
    void *heap;
    DTable *dict;
    void* (*HeapInsert)(struct Heap * H, int value, void * data);
    HNode* (*HeapExtractMin)(struct Heap * H);
    void (*HeapIncreasePriority)(struct Heap * H, void * ptr, int
       value);
    void (*viewHeap)(const struct Heap * H, void (*ptrf)());
} Heap;
```

La fonction suivante crée et initialise un tas générique (son code est fourni dans `src/heap.c`).

```
/* Construire un tas vide en choisissant son implémentation (type) :
```

```
 * (0) DynTableHeap
 * (1) CompleteBinaryTreeHeap
 * (2) ListHeap */
Heap * newHeap(int type) {
    assert(type == 0 || type == 1 || type == 2);
    Heap* H = calloc(1, sizeof(Heap));
    H->dict = newDTable();
    switch (type) {
        case 0:
            H->heap = newDTable();
            H->HeapInsert = DTHeapInsert;
            H->HeapExtractMin = DTHeapExtractMin;
            H->HeapIncreasePriority = DTHeapIncreasePriority;
            H->viewHeap = viewDTHeap;
            break;
        case 1:
            H->heap = newCBTree();
            H->HeapInsert = CBTHeapInsert;
            H->HeapExtractMin = CBTHeapExtractMin;
            H->HeapIncreasePriority = CBTHeapIncreasePriority;
            H->viewHeap = viewCBTHeap;
            break;
        case 2:
            H->heap = newList();
            H->HeapInsert = OLHeapInsert;
            H->HeapExtractMin = OLHeapExtractMin;
            H->HeapIncreasePriority = OLHeapIncreasePriority;
            H->viewHeap = viewOLHeap;
            break;
    }
    return H;
}
```

Exercice 14 Implémenter dans le fichier `src/heap.c` la fonction suivante qui crée un nouveau nœud du tas avec la donnée `data` et la priorité `value` (le prototype se trouve dans `include/heap.h`). N'oubliez pas de faire l'allocation mémoire pour ce nouveau nœud en utilisant `malloc`. La fonction renvoie le nouveau nœud crée.

```
/* Construire et initialiser un nouveau nœud d'un tas.
 * Renvoie le nouveau nœud créé. */
HNode * newHNode(int value, void * data);
```

En suite, on va implémenter les trois primitives d'un tas pour chacun de trois types différents de tas introduits ci-dessus.

5.1 Implémentation par listes ordonnées

Une liste ordonnée (*ordered list*) est une liste doublement chaînée, dont les éléments sont triés par ordre de priorité décroissant. L'élément avec la plus grande priorité (c'est-à-dire, le sommet avec la plus petite distance actuelle depuis le sommet de départ) se trouve dans la première position de la liste.

Exercice 15 Implémenter dans le fichier `src/heap.c` la fonction suivante qui réalise l'insertion d'un nouveau élément de donnée `data` et priorité `value` dans le tas `H` représenté par une liste ordonnée (le prototype se trouve dans `include/heap.h`). Le nouveau élément doit être placé dans la bonne position. Utiliser les fonctions `listInsertAfter` et `newHNode`.

```
/* Ajouter dans le tas H un élément de donnée data et priorité value.
 */
void* OLHeapInsert(Heap *H, int value, void* data);
```

Exercice 16 Implémenter dans le fichier `src/heap.c` la fonction suivante qui extrait l'élément avec la plus grande priorité du tas `H` représenté par une liste ordonnée (le prototype se trouve dans `include/heap.h`). La fonction renvoie l'élément extrait.

```
/* Extraire du tas H le nœud avec la plus grande priorité. */
HNode * OLHeapExtractMin(Heap * H);
```

Exercice 17 Implémenter dans le fichier `src/heap.c` la fonction suivante qui augmente à `value` la priorité du nœud `LNode` indiqué par le pointeur `lnode` dans le tas `H` représenté par une liste ordonnée (le prototype se trouve dans `include/heap.h`). Utiliser la fonction `listSwap`.

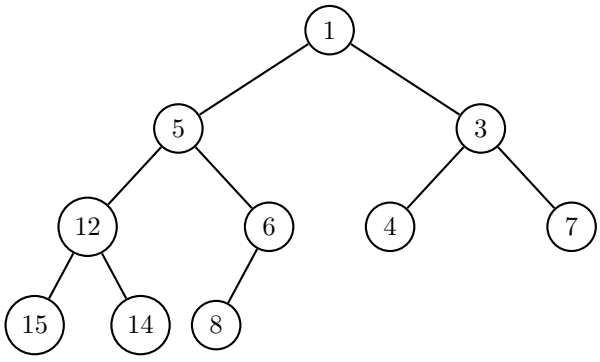
```
/* Dans le tas H, incrémenter la priorité du nœud
 * indiqué par le pointeur position à value. */
void OLHeapIncreasePriority(Heap * H, void * lnode, int value);
```

Exercice 18 Implémenter dans le fichier `src/heap.c` la fonction suivante qui affiche les éléments du tas `H` représenté par une liste ordonnée en utilisant le pointeur de fonction ptrF (le prototype se trouve dans `include/heap.h`).

```
/* Afficher les éléments du tas H.
 * Chaque élément est affiché grace au pointeur de fonction ptrF. */
void viewOLHeap(const Heap * H, void (*ptrf)());
```

5.2 Implémentation par arbres binaires complets

Dans un tas minimier implémenté par un arbre binaire complet la propriété suivante est toujours vraie : pour tout nœud du tas sa valeur (priorité) est plus petite ou égale que les valeurs situées dans ses fils gauche et droit. Autrement dit, l'élément avec la plus grande priorité se trouve toujours dans la racine, et chaque chemin de la racine vers une feuille quelconque est ordonné par ordre croissant.



Le fonctionnement d'un tas implémenté par un arbre binaire complet est expliqué dans la présentation du projet qui se trouve sur Arche.

Exercice 19 Implémenter dans le fichier `src/heap.c` la fonction suivante qui réalise l'insertion d'un nouveau élément de donnée `data` et priorité `value` dans le tas `H` représenté par un arbre binaire complet (le prototype se trouve dans `include/heap.h`). Le nouveau élément doit être placé dans la bonne position. Utiliser les fonctions `CBTreeInsert`, `CBTreeSwap` et `newHNode`.

```
/* Ajouter dans le tas H un élément de donnée data et priorité value.
 */
void * CBTHeapInsert(Heap * H, int value, void * data);
```

Exercice 20 Implémenter dans le fichier `src/heap.c` la fonction suivante qui extrait l'élément avec la plus grande priorité du tas `H` représenté par un arbre binaire complet (le prototype se trouve dans `include/heap.h`). La fonction renvoie l'élément extrait. Utiliser les fonctions `CBTreeRemove`, `CBTreeSwapRootLast` et `CBTreeSwap`.

```
/* Extraire du tas H le nœud avec la plus grande priorité. */
HNode * CBTHeapExtractMin(Heap *H);
```

Exercice 21 Implémenter dans le fichier `src/heap.c` la fonction suivante qui augmente à `value` la priorité du nœud `TNode` indiqué par le pointeur `tnode` dans le tas `H` représenté par un arbre binaire complet (le prototype se trouve dans `include/heap.h`). Utiliser la fonction `CBTreeSwap`.

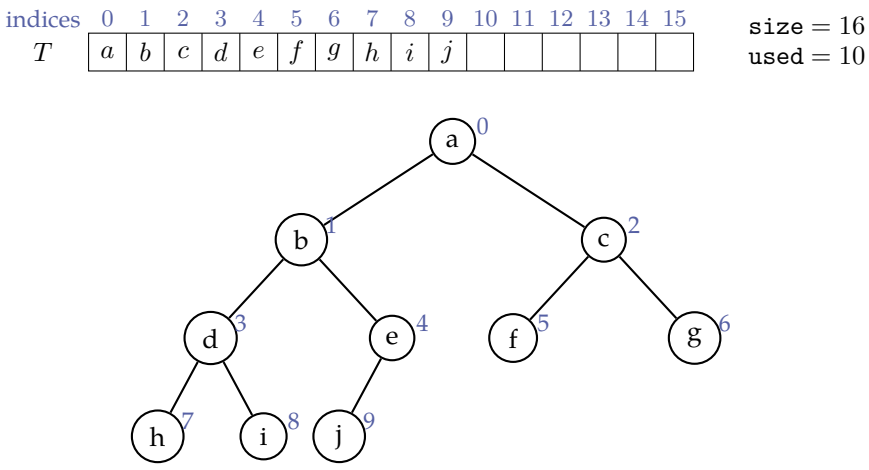
```
/* Dans le tas H, incrémenter la priorité du nœud
 * indiqué par le pointeur position à value. */
void CBTHeapIncreasePriority(Heap *H, void *tnode, int value);
```

Exercice 22 Implémenter dans le fichier `src/heap.c` la fonction suivante qui affiche les éléments du tas `H` représenté par un arbre binaire complet en utilisant le pointeur de fonction ptrF (le prototype se trouve dans `include/heap.h`). Utiliser la fonction `viewCBTree`.

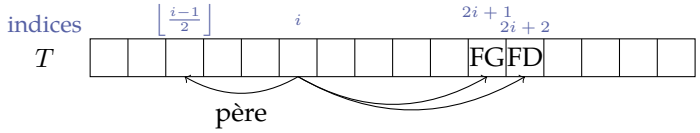
```
/* Afficher les éléments du tas H.
 * Chaque élément est affiché grace au pointeur de fonction ptrF. */
void viewCBTHeap(const Heap *H, void (*ptrf)(const void*));
```

5.3 Implémentation par tableaux dynamiques

Un tableau peut simuler le fonctionnement d'un arbre binaire complet de façon beaucoup plus compact (en évitant les pointeurs et les remplaçant par les indices).



En général, pour le nœud à la position i du tableau, son père se trouve dans la position $\left\lfloor \frac{i-1}{2} \right\rfloor$, son fils gauche dans la position $2i+1$ et son fils droit dans la position $2i+2$. Pour plus de détails sur le fonctionnement d'un tas implémenté par un tableau, voir la présentation du projet sur Arche.



Exercice 23 Implémenter dans le fichier `src/heap.c` la fonction suivante qui réalise l'insertion d'un nouveau élément de donnée `data` et priorité `value` dans le tas `H` représenté par un tableau dynamique (le prototype se trouve dans `include/heap.h`). Le nouveau élément doit être placé dans la bonne position. Utiliser les fonctions `DTableInsert`, `DTableSwap` et `newHNode`.

```
/* Ajouter dans le tas H un élément de donnée data et priorité value.
 */
void* DTHeapInsert(Heap *H, int value, void *data);
```

Exercice 24 Implémenter dans le fichier `src/heap.c` la fonction suivante qui extrait l'élément avec la plus grande priorité du tas `H` représenté par un tableau dynamique (le prototype se trouve dans `include/heap.h`). La fonction renvoie l'élément extrait. Utiliser les fonctions `DTableRemove` et `DTableSwap`.

```
/* Extraire du tas H le nœud avec la plus grande priorité. */
HNode * DTHeapExtractMin(Heap* H);
```

Exercice 25 Implémenter dans le fichier `src/heap.c` la fonction suivante qui augmente à `value` la priorité de l'élément dans la position `position` dans le tas `H` représenté par un tableau dynamique (le prototype se trouve dans `include/heap.h`). Utiliser la fonction `DTableTreeSwap`.

```
/* Dans le tas H, incrémenter la priorité du nœud
 * indiqué par le pointeur position à value. */
void DTHeapIncreasePriority(Heap* H, void *position, int value);
```

Exercice 26 Implémenter dans le fichier `src/heap.c` la fonction suivante qui affiche les éléments du tas `H` représenté par un tableau dynamique en utilisant le pointeur de fonction ptrF (le prototype se trouve dans `include/heap.h`). Utiliser la fonction `viewDTable`.

```
/* Afficher les éléments du tas H.
 * Chaque élément est affiché grace au pointeur de fonction ptrF. */
void viewDTHeap(const Heap* H, void (*ptrf)(const void*));
```

6 Graphe

Le code pour la représentation d'un graphe est entièrement donné et correspond au code que vous avez vu au cour "Programmation Avancée". En suite, nous vous présentons seulement les structures utilisées afin de vous signaler les différences par rapport au semestre précédent. Pour plus de détails, voir les fichiers `include/graph.h`, `include/town.h`, `include/road.h`, `src/graph.c`, `src/town.c` et `src/road.c`.

Ils existent trois nouveaux attributs dans la structure `town` :

— `dist` : la distance finale ou actuelle (pendant l'exécution de l'algorithme) du sommet nommé `name` à partir du sommet de départ,

— `pred` : le prédécesseur finale ou actuelle dans le plus court chemin entre le sommet nommé `name` et le sommet de départ,
— `ptr` : un pointeur vers le nœud du tas qui contient le sommet nommé `name`.

```
/** TYPE ABSTRAIT
 * Cette structure contient les champs suivants :
 *  (+) name : un nom de ville
 *  (+) alist : la liste d'adjance;
 *      liste des routes reliant cette ville à une autre ville.
 */
struct town {
    char * name; // e.g. L.A., N.Y., S.F.
    List * alist; // adjacency list which data are roads
    int dist; // distance actuelle ou finale
    struct town * pred; // prédécesseur

    /**
     * Pointeur vers le nœud du tas qui contient la ville:
     * (+) un entier (position au dictionnaire) pour un Tableau
        Dynamique
     * (+) un LNode pour une Liste Ordonnée
     * (+) un TNode pour un Arbre Binaire Complet
     * Utiliser comme deuxième argument (position)
     * dans la fonction HeapIncreasePriority
     */
    void * ptr;
};
```

Dans la structure `road`, nous avons ajouté le nouveau attribut `km` qui correspond à la distance en kilomètres de l'arête (U, V).

```
/** TYPE ABSTRAIT
 * Cette structure décrit une route
 * par les deux villes U et V qu'elle relie.
 */
struct road {
    struct town * U, * V; // indices of towns in graph->towns
    int km; // distance en kilomètres
};
```

7 L'algorithme de Dijkstra

7.1 Fonctions auxiliaires fournies

Les fonctions suivantes vous aiderons à visualiser la solution ainsi que les résultats intermédiaires de l'exécution de l'algorithme de Dijkstra (`src/main.c`).

```
/* Exemple d'une fonction qui affiche le contenu d'un HNode.
 * A modifier si besoin. */
void viewHNode(const HNode *node)
```

```
/* Affichage de la solution de l'algorithme de Dijkstra.
 * Pour chaque ville du graphe G on doit déjà avoir défini
 * les valeurs dist et pred en exécutant l'algorithme de Dijkstra. */
void viewSolution(graph G, char * sourceName)
```

7.2 L'algorithme

Vous avez maintenant tous les éléments nécessaires pour coder l'algorithme proposé par Dijkstra. Une description de l'algorithme est donnée en cours. Vous pouvez trouver la présentation sur Arche.

Exercice 27 Implémentez dans le fichier `src/main.c` l'algorithme de Dijkstra.

```
/**
 * Algorithme de Dijkstra
 * inFileName : nom du fichier d'entrée
 * outFileName : nom du fichier de sortie
 * sourceName : nom de la ville de départ
 * heaptype : type du tas {0--tableaux dynamiques,
 * 1--arbres binaires complets, 2--listes ordonnées}
 */
void Dijkstra(char * inFileName, char * outFileName, char * sourceName
   , int heaptype)
```

8 Évaluation de performance de l'algorithme

Exercice 28 Réfléchir sur la complexité des trois primitives pour les trois structures de données ci-dessus et remplir le tableau suivant. (la réponse à cette question doit apparaître à votre rapport)

	insertion	extraction min	incrémentation de la priorité
liste ordonnée			
arbre binaire complet			
tableau dynamique			

Exercice 29 Comparer le temps d'exécution de l'algorithme de Dijkstra en utilisant les trois implémentations du tas. Quelle implémentation auriez vous choisi et pourquoi ? (la réponse à cette question doit apparaître à votre rapport)

9 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 :

- Choisir un fichier contenant l'entrée du problème (villes et routes).
- Afficher les villes et les routes dans une fenêtre.
- Choisir le type du tas (0–tableaux dynamiques, 1–arbres binaires complets, 2–listes ordonnées).
- Choisir la ville de départ.
- Appliquer l'algorithme avec la structure de données choisie en appelant la fonction `Dijkstra` déjà implémenter en C (interface JNI).
- Afficher en gras les arrêtes qui participent aux plus courts chemins issus de la ville de départ ainsi que la distance optimale à coté de chaque ville.

10 Dépôt et Évaluation

10.1 Le dépôt

Le projet devra contenir :

1. le répertoire .git

2. le code en C
3. le code en Java
4. un rapport de maximum 5 pages

Chaque équipe doit soumettre un fichier `.zip` contenant les 4 points ci-dessus. La date limite de dépôt est fixée au mercredi **6 mai, 23h59**. Aucune prolongation ne sera accordée. Dans le cas de dépassement de cette date, vous aurez une pénalité d'un point par jour de retard.

10.2 Soutenance

Chaque équipe doit aussi présenter son projet pendant environ 20 minutes. Tous les membres de l'équipe participeront dans cette présentation. La note de cette présentation peut être différent pour les membres de la même équipe. Les présentations auront lieux la semaine du 11 mai pendant les heures marquées comme TD sur l'emploi du temps. Le programme des soutenances sera annoncé ultérieurement au cours du semestre.

10.3 Évaluation

Il n'y a pas de 2eme session ni pour le CC ni pour le CT.

La note du contrôle continue se compose de :

— Rapport : 50%
— Soutenance : 50%

Il n'y a pas d'examen terminal écrit. La note du code sera utilisée à la place de l'épreuve écrite. Veuillez trouver ci-dessous un barème indicatif :

— Utilisation du GIT : 5%
— Listes Doublement Chaînées : 5%
— Arbres Binaires Complets : 15%
— Tableaux Dynamiques : 10%
— Tas - Listes Ordonnées : 10%
— Tas - Arbres Binaire Complets : 10%
— Tas - Tableaux Dynamiques : 10%
— Algorithme de Dijkstra : 15%
— Interface graphique : 20%

La note finale sera : 0,4 * CC + 0,6 * Code.

A la fin, on mettra en **compétition** vos implémentations de l'algorithme de Dijkstra utilisant le tas basé sur les tableaux dynamiques. L'équipe avec l'implémentation la plus rapide va gagner +1 sur la note finale.

10.4 Consignes pour le rapport

Le rapport final ne doit pas dépasser 5 pages en total. Le rapport n'est pas une documentation du code. Par exemple, vous pouvez expliquer :

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