TP2 : Tas

L'objectif de ce TP est d'implanter les opérations de base sur les tas dans une première partie, et d'en déduire une implantation des files de priorités dans une seconde partie, et enfin (en bonus) de s'en servir pour implanter l'algorithme de fusion de listes vu en TD.

Représentations des tas
Les tas seront représentés par des tableaux. Pour simplifier les implantations, nous travaillerons avec des tableaux de taille fixée ($N = 1000$), le tas représenté n'occupant qu'une partie du tableau. Ainsi, le fichier `TP2.cc` déclare une macro (`#define N 1000`) puis tous les tableaux sont déclarés par exemple avec `int T[N];`. Pour définir un tas, on utilise donc un tableau `T` de taille N et un entier `n`, dit *taille utile*, indiquant le nombre d'éléments du tas. Les fonctions qui manipulent un tas doivent donc prendre en paramètre non seulement le tableau mais aussi la taille du tas (`int n`).

Utilisation de `template`
Pour autoriser différents types de données dans les tableau, toutes les fonctions (ou presque) utilisent un `template` : le tas est donc représenté dans les fonctions par un entier (`int n`) et un tableau (`E T[]`) d'éléments de type `E`. *En pratique, vous pouvez tout coder comme si les tableaux étaient des tableaux de `int` ou `float` !*

Tests et affichage graphique
La fonction `main` contient des tests pour chaque question du sujet. Décommentez les tests au fur et à mesure de votre avancée. **Il est impératif de tester toutes vos fonctions et de bien vérifier que le résultat attendu est le bon !**
Pour aider la visualisation, vous pouvez obtenir une représentation graphique d'un tas `T` de taille `n` avec la fonction `dessinTas(n, T, "nom")`, qui produira un fichier `"nom.pdf"`. Des exemples d'utilisation (commentés) sont fournis dans les tests.

Exercice 1. *Tas et tri par tas*
L'objectif du premier exercice est d'implanter les fonctions sur les tas nécessaires au tri par tas, puis d'implanter celui-ci.

1. Compléter la fonction `void AfficherTableau(int n, E T[])` qui affiche le tableau T de taille utile n. *Voir les tests pour le type d'affichage souhaité.*
2. Compléter la fonction `int RemplitTableau(E T[])` qui remplit le tableau T en demandant à l'utilisateur sa taille n, puis chacune des n valeurs du tableau. La fonction renvoie l'entier n rentré par l'utilisateur.
3. Compléter la fonction `void TableauAleatoire(int n, int T[], int m, int M)` qui remplit le tableau T avec n entiers aléatoires compris entre m et M (inclus). *On pourra utiliser `rand()%k` qui produit un entier aléatoire entre 0 et $k-1$. L'initialisation de la graine avec `srand (time(NULL))` est effectuée dans la fonction `main`.*
4. Compléter fonction `void Entasser(int n, E T[], int i)` qui entasse le nœud i dans le tableau T de taille utile n.
5. Utiliser la fonction `Entasser` pour compléter la `void Tas(int n, E T[])` qui transforme le tableau T de taille utile n en un tas.
6. Compléter la fonction `void Trier(int n, E T[], E Ttrie[])` qui implémente l'algorithme TriTas du cours où `Ttrie` contiendra les valeurs de `T` triées en ordre croissant.

7. *(bonus)* Réaliser une implémentation `TrierSurPlace` de TriTas qui trie le tableau sur place, c'est-à-dire sans l'utilisation du tableau `Ttrie` annexe.

Exercice 2. *File de priorité*

Dans cet exercice, vous devez continuer l'implantation des tas, avec comme objectif de réaliser une file de priorité. On rappelle que dans une file de priorité, on dispose d'un ensemble X d'éléments (des entiers ici) et chaque élément $x \in X$ possède une priorité p_x (flottant ici). On représentera une file de priorité par un tas donc chaque nœud contient un couple (x, p_x).
Pour cela, on fournit dans `Couples.h` et `Couples.cc` un `struct couple`, contenant deux champs : `val` est un champ de type `int` contenant la valeur (x) et `priorite` est un champ de type `float` contenant la priorité (p_x). De plus, les opérateurs de comparaison sur les `couples` sont fournis (ils comparent les priorités) et on peut afficher un `couple c` avec `cout << c`. **Toutes vos fonctions sur les tas devraient encore marcher si `T` est un tableau de `couples`.**

1. Compléter la fonction `void Remonter(int n, E T[], int i)` qui fait remonter le nœud i jusqu'à sa bonne place dans le tas T de taille utile n.
2. En déduire la fonction `E Supprimer(int n, E T[], int i)` qui supprime le nœud i de T et retourne cette valeur, en conservant la structure de tas de T.
3. Utiliser les fonctions sur les tas pour compléter les fonctions `void AjoutElem(int n, couple F[], int v, float p)` qui ajoute un élément x de priorité p à la file de priorité F, et `couple ExtraitMax(int n, couple F[])` qui extrait l'élément de priorité maximale de la file F et renvoie sa valeur (le `couple` (x, p_x)). *Ces deux fonctions doivent bien sûr conserver la structure de tas de F.*
4. Compléter les fonctions `AugmentePriorite(int n, couple F[], int i, float gain)` et `DiminuePriorite(int n, couple F[], int i, float perte)` qui changent la priorité du nœud i de F (en ajoutant `gain` et retirant `perte`, respectivementi, à la priorité), en conservant la structure de tas de F.

Exercice bonus

Dans cet exercice, vous devez implanter l'algorithme de fusion de tableaux triés basé sur les files de priorités, vu en TD. L'idée est de conserver à chaque instant un tas contenant le plus grand élément de chaque tableau à trier. À chaque étape, on extrait le plus grand élément de ce tas pour l'insérer dans le tableau de sortie, et on ajoute au tas un nouvel élément du tableau auquel appartenait l'élément qui vient d'être inséré.
On utilisera une file de priorité, donc les priorités seront les éléments des tableaux à fusionner, et dont les valeurs seront le numéro de leur tableau d'origine. Par exemple, si le tas courant est `[(2, 6.5), (0, 1.3), (1, 3.5)]`, cela signifie que le prochain élément à insérer est 6.5, et qu'il provient du tableau numéro 2.
Remarque. Puisqu'on utilise le `struct couple` avec des priorités qui sont des flottants, les tableaux à fusionner seront nécessairement des tableaux de flottants.

✎ Compléter la fonction `void Fusion(int k, int n, float T[N][N], float Sortie[])` qui complète le tableau `Sortie` avec la fusion des tableaux `T[0]`, `T[1]`, ..., `T[k]` qui sont tous de la même taille utile `n`.