TP6 : Tas binaire

Le but de ce TP est la mise en place d'une structure de données de tas binaire.

1 Rappels de cours

Un tas binaire est une structure de données permettant de modéliser une file à priorités. Elle permet de réaliser les opérations suivantes :

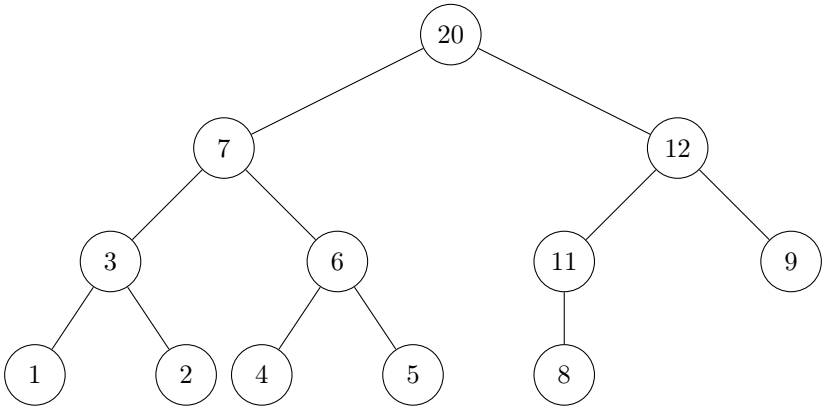
- insérer un nouvel élément associé à une priorité en $O(\log n)$;
- accéder à l'élément de priorité maximale en $O(1)$;
- supprimer l'élément de priorité maximale en $O(\log n)$;
- modifier la priorité d'un élément (si vous pouvez le retrouver dans le tableau) en $O(\log n)$.

Pour simplifier la suite, nous ne noterons que la priorité des éléments, et omettrons leur valeur.

Cette structure de données maintient un ordre partiel sur les données via un arbre binaire. Cet arbre respecte les propriétés suivantes :

- chaque nœud a une priorité supérieure à celles de ses enfants ;
- un nouveau niveau n'est créé que si le niveau précédent est plein ;
- les niveaux de l'arbre sont remplis de gauche à droite.

Par exemple :



Cette structure de données peut être implémentée efficacement dans un tableau : la racine est la première case du tableau, les deux cases suivantes contiennent le premier niveau, les quatre suivantes le second niveau, et ainsi de suite. L'arbre précédent est ainsi encodé dans le tableau :

20	7	12	3	6	11	9	1	2	4	5	8

Pour circuler dans l'arbre, étant donné un indice i, nous avons :

- $parent(i) = \lfloor \frac{i-1}{2} \rfloor$ où $\lfloor . \rfloor$ représente l'arrondi entier inférieur ;
- $gauche(i) = 2i + 1$ pour obtenir l'enfant gauche ;
- $droite(i) = 2i + 2$ pour obtenir l'enfant droit.

La racine de l'arbre a par construction la plus forte priorité, et est stockée dans la première case du tableau. Il est donc possible d'y accéder en temps constant. Pour les autres opérations, l'arbre étant équilibré, toute action qui nécessite de le parcourir de haut en bas ou de base en haut aura donc au pire une complexité en $O(\log n)$, la hauteur de l'arbre en fonction du nombre d'éléments.

Insertion : pour préserver l'agencement de l'arbre, l'élément est ajouté à la fin du tableau, et remonté ensuite vers la racine selon sa priorité. Pour déterminer si un nœud doit remonter, sa priorité est comparée avec celle de son parent.

Procédure InsertionTas(t, e)
- **données :** t : un tableau contenant un tas, e un élément
- **Algorithme**
  - $i \leftarrow$ taille de t
  - **tant que** $i > 0$ et $e > t\lfloor \frac{i-1}{2} \rfloor$ **faire**
    - $t[i] \leftarrow t\lfloor \frac{i-1}{2} \rfloor$
    - $i \leftarrow \lfloor \frac{i-1}{2} \rfloor$
  - $t[i] \leftarrow e$

Suppression : pour préserver l'agencement de l'arbre, l'élément en fin de tableau est déplacé à la racine. Il est ensuite descendu dans l'arbre jusqu'à retrouver une place qui convienne à sa priorité.

Procédure SuppressionTas(t)
- **données :** t : un tableau contenant un tas
- **Algorithme**
  - $s \leftarrow$ taille de $t - 1$
  - $e \leftarrow t[s]$
  - réduire t d'un élément
  - $i \leftarrow 0$
  - **tant que** $(2i + 1 < s$ et $e < t[2i + 1])$ ou $(2i + 2 < s$ et $e < t[2i + 2])$ **faire**
    - **si** $2i + 2 < s$ et $t[2i + 1] < t[2i + 2]$ **alors**
      - $t[i] \leftarrow t[2i + 2]$
      - $i \leftarrow 2i + 2$
    - **sinon**
      - $t[i] \leftarrow t[2i + 1]$
      - $i \leftarrow 2i + 1$
  - $t[i] \leftarrow e$

2 Fonctionnalités de base

Pour simplifier le problème, votre tas binaire contiendra des entiers, qui correspondent aux priorités des éléments. Le sommet du tas (la première case du tableau) est donc l'entier le plus grand. Vous coderez les fonctionnalités de base du tas binaire :

Avec le style `C` :

```
void init_tas(tas* t) ;
void detruire_tas(tas* t) ;
void inserer_tas(tas* t, int valeur) ;
int consulter_tas(const tas* t) ;
void supprimer_tas(tas* t) ;
```

Avec le style `C++` :

```
void init_tas(tas& t) ;
void detruire_tas(tas& t) ;
void inserer_tas(tas& t, int valeur) ;
int consulter_tas(const tas& t) ;
void supprimer_tas(tas& t) ;
```

3 Tri par tas

Le tri par tas est un tri qui a une complexité optimale $\Theta(n \log n)$. Il consiste à transformer le tableau initial en tas, puis à supprimer un à un les éléments du tas pour les replacer à la fin du tableau.

```
void tri_tas(int* tableau, int taille) ;
```

4 Généricité

Modifiez vos structures de données pour rendre votre tas générique. Vous devriez ainsi avoir une interface du type :

```
void init_tas(tas& t, int octets, bool (*compare)(void*, void*)) ;
void detruire_tas(tas& t) ;
void inserer_tas(tas& t, const void* valeur) ;
void consulter_tas(const tas& t, void* destination) ;
void supprimer_tas(tas& t) ;
```

Dans cette interface, `bool (*compare)(void*, void*)` est une fonction de comparaison de deus éléments génériques. Un exemple d'implémentation pour des entiers est :

```
bool compare_int(void* i1, void* i2){
  int* ii1 = (int*) i1 ;
  int* ii2 = (int*) i2 ;
  return *ii1 < *ii2 ;
}
```

Un tas d'entiers serait donc initialisé par :

```
init_tas(t, sizeof(int), compare_int) ;
```

5 Utilisation de la librairie standard

La librairie standard vous propose deux façons de gérer des tas via l'entête `<queue>`. Vous pouvez tout d'abord utiliser le type `std::priority_queue`. Ce type prend trois paramètres *templates* (Vous verrez cette notion l'an prochain). Le premier est le type des éléments stockés, le second est le type de conteneur utilisé pour stocker le tas (par défaut un vector du type fourni comme premier argument) et le troisième est la fonction de comparaison (par défaut l'opérateur $<$). La seconde façon de gérer un tas est via les fonctions `make_heap`, `push_heap` et `pop_heap`. Elles s'appliquent sur un tableau que vous fournissez en paramètre via la notion d'*itérateur*.

Lisez la documentation en ligne et réfléchisez à l'intégration de ces outils dans votre projet pour l'algorithme de Dijkstra. Notez bien qu'aucun de ces outils ne propose la fonctionalité de changement de priorité qui est compliquée à mettre en œuvre pour un tas binaire.

6 Pour aller plus loin

Si les files à priorité vous intéressent, il en existe de nombreuses variantes. En terme de complexité théorique, le tas binaire n'est pas optimal. Vous pouvez vous documenter sur le *tas fibonacci* (fibonacci heap) et le *pairing heap*. Si ces variantes sont théoriquement meilleures du point de vue de la complexité, il reste nécessaire de s'assurer qu'en prtique pour les cas d'utilisation courants, ils sont réellement efficaces. Dans la pratique les tas binaires restent très compétitifs.