Deuxième partie

Structures de données, arbres

Chapitre 3

Structures de données

Dans le chapitre précédant, nous avons utilisé des tableaux pour organiser des données ensemble. Il existe beaucoup d'autres structures de données que les tableaux qui répondent chacune à des besoins particuliers.

Une structure de donnée peut être caractérisée par les opérations fondamentales que l'ont peut faire dessus. L'ajout et le retrait d'un élément sont deux opérations fondamentales que l'on retrouve dans toutes les structures données. Sur les tableaux, une autre opération fondamentale est l'accès à un élément à partir de son rang (son indice) pour lire ou modifier sa valeur.

Si on ne spécifie pas la manière dont les données sont organisées en mémoire et la manière dont les opérations fondamentales sont implantés, on parle de structure abstraite de donnée.

On s'intéresse particulièrement à la complexité en temps et/ou en espace des opérations fondamentales. Cette complexité est généralement faible (temps ou espace constant, sub-linéaire ou linéaire). Autrement l'opération ne mérite pas l'appelation fondamentale. Ainsi accèder à un élément dans un tableau à partir de son rang se fait en temps et en espace constant. Mais l'insertion et la suppression d'un élément à un rang quelconque demande en pire cas et en moyenne un temps linéaire en la taille du tableau.

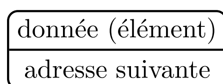
Nous voyons dans ce chapitre trois structures de données élémentaires : la pile, la file et la file de priorité encore appelée tas (ou maximier, ou minimier). Pour les besoins de la présentation de cette dernière structure de donnée (les tas), nous présentons rapidement les arbres (ou arborescences) et plus particulièrement les arbres binaires quasi-complets.

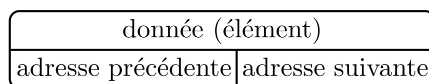
Si les tableaux sont directement implantés dans les langages de programmation, ce n'est pas nécessairement le cas des autres structures de données. Pour implanter les piles et les files, il est courant d'utiliser des listes chaînées. Ce chapitre commence donc par un rappel sur les listes chaînées et leur(s) implantation(s) en C.

3.1 Listes chaînées en C

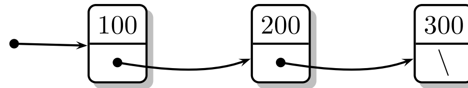
En anglais : *linked lists*

Liste chaînée. Une liste chaînée est une structure de donnée séquentielle qui permet de stocker une suite d'éléments en mémoire. Chaque élément est stocké dans une cellule et les cellules sont *chaînées* : chaque cellule contient, en plus d'un élément, l'adresse de la cellule suivante, et éventuellement l'adresse de la cellule précédente dans le cas des listes doublement chaînées. On peut réprésenter graphiquement une cellule de liste simplement chaînée par une boîte contenant deux champs et une cellule de liste doublement chaînée par une boîte contenant trois champs :





Au lieu d'écrire l'adresse mémoire d'une cellule, on représente cette adresse par une flèche vers cette cellule. La liste simplement chaînée des entiers 100, 200, 300 se représente ainsi :



On utilise une valeur spéciale, \ dans la notation graphique et `NULL` en C, pour signaler une adresse qui ne mène nulle part. Dans le cas des listes simplement chaînées cette valeur signale la fin de la liste.

La première cellule de la liste s'appelle la *tête* de la liste et le reste de la liste la *queue*. D'un point de vue mathématique, les listes chaînées sont définies par induction en disant qu'une liste est soit la liste vide, soit un élément (la tête) suivi d'une liste (la queue).

Le type des listes simplement chaînées peut être défini en C par les instructions :

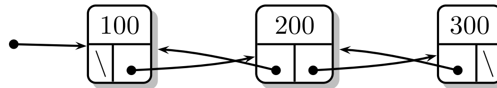
```
struct cellsimple_s {
    element_t element;
    struct cellsimple_s * suivant;
};

typedef struct cellsimple_s * listesimple_t;
```

Ces instructions déclarent :

1. qu'une cellule est une structure contenant un élément (`element`) et un pointeur sur une cellule (`suivant`) ;
2. qu'une liste est un pointeur sur une cellule (une variable contenant l'adresse d'une cellule).

La liste vide sera simplement égale au pointeur `NULL`.

La variante doublement chaînée de la liste précédente se représente ainsi :



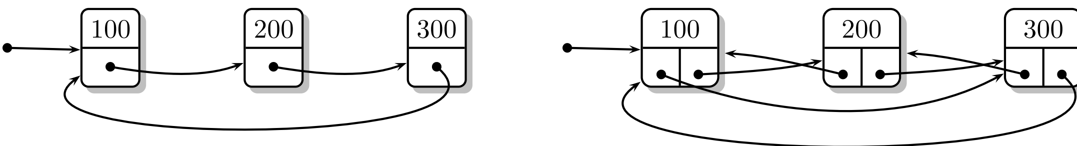
Le type des listes doublement chaînées peut être défini en C par :

```
struct celldouble_s {
    element_t element;
    struct celldouble_s * precedant;
    struct celldouble_s * suivant;
};

typedef struct celldouble_s * listedouble_t;
```

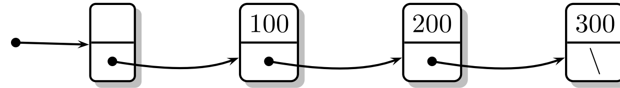
qui ne diffère du type des listes simplement chaînées que par l'ajout du champ `precedant` dans les cellules.

Dans les deux cas, listes simplement chaînées et listes doublement chaînées, on peut introduire des variantes dans la représentation des listes. En voici deux qui ne changent pas la manière de définir les listes en C, mais changent la manière de les manipuler.

Listes circulaires. On peut utiliser des listes circulaires, où après le dernier élément de la liste on revient au premier. Graphiquement on fait comme ceci :



Utilisation d'une sentinelle. Il est parfois plus facile de travailler avec des listes chaînées dont le premier élément ne change jamais. Pour cela on introduit une première cellule *sentinelle* dont l'élément ne compte pas. La liste vide contient alors cette unique cellule. L'avantage est que la modification (insertion, suppression) du premier élément de la liste (qui est alors celui dans la seconde cellule) ne nécessite pas de mettre à jour le pointeur qui indique le début de la liste (voir l'exemple de code C pour la suppression avec et sans sentinelle un peu plus loin). La liste de notre exemple, se représente alors comme ceci :



On ne tient en général jamais compte de la valeur de l'élément stocké dans la cellule sentinelle.

3.1.1 Opérations fondamentales

Les opérations fondamentales sur les listes chaînées sont : le test à vide ; l'insertion d'un élément en tête de liste ; la lecture de l'élément en tête de liste ; la suppression de l'élément en tête de liste. Voici une implantation de ces fonctions en C pour les listes simplement chaînées (non circulaires et sans sentinelle). Lorsque ces fonctions fonctionnent sans modification pour les listes doublement chaînées, on utilise un type générique `liste_t`.

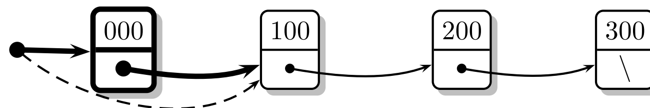
Test à vide. Le test à vide prend une liste en paramètre, renvoie vrai (ou 1) si la liste est vide et faux (0) sinon.

```
int listeVide(liste_t x){
  if (x == NULL) return 1;
  else return 0;
}
```

Lecture.

```
element_t lectureTete(liste_t x){
  return x->element;
}
```

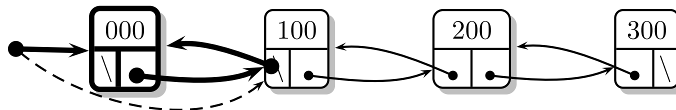
Insertion. L'insertion d'un élément en tête d'une liste prend en paramètre une liste et un élément et ajoute une cellule en tête de la liste contenant cet élément.



Ceci modifie la liste et il est nécessaire de récupérer sa nouvelle valeur. On peut renvoyer la nouvelle liste comme valeur de retour de la fonction (`insererListe1`), où bien utiliser la liste comme un argument-résultat en passant son adresse à la fonction (`insererListe2`).

```
listesimple_t insererListe1(listesimple_t x, element_t e){
    listesimple_t y;
    y = malloc(sizeof(*y)); /* Création d'une nouvelle cellule. */
    if (y == NULL) perror("échec malloc");
    y->element = e;
    y->suivant = x;
 // y->precedant = NULL;  <--  Pour les listes doublement chaînées
 // x->precedant = y;     <--
    return y;
```

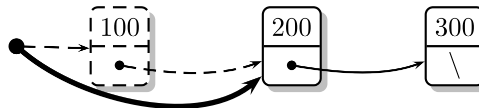
```
}

void insererListe2(listesimple_t * px, element_t e){
    listesimple_t y;
    y = malloc(sizeof(*y)); /* Création d'une nouvelle cellule. */
    if (y == NULL) perror("échec malloc");
    y->element = e;
    y->suivant = *px;
 // y->precedant = NULL;  <--  Pour les listes doublement chaînées
 // (*px)->precedant = y; <--
    *px = y;
}
```



La lecture de l'élément en tête de liste, prend une liste et renvoie la valeur de son élément de tête.

```
element_t listeTete(liste_t x){
  assert(x != NULL); /* Pour le débogage, on vérifie que la liste n'est pas vide. */
   return x->element;
}
```

Suppression.



La suppression de l'élément en tête de liste prend une liste et supprime sa première cellule. Comme pour l'insertion, il faut mettre à jour la valeur de la liste. Soit on renvoie cette valeur en sortie de fonction (`suppressionListe1`) soit on passe la liste par adresse (`suppressionListe2`). À titre d'exemple, la fonction `suppressionListeSentinelle` montre le code de la fonction de suppression lorsque les listes ont une sentinelle (c'est alors la deuxième cellule qu'il faut supprimer).

```
listesimple_t supprimerListe1(listesimple_t x){
    listesimple_t y;
    y = x->suivant;
    free(x);  /* Suppression de la première cellule */
    return y; /* On renvoie la cellule suivante */
}

void supprimerListe2(listesimple_t * px){
    listesimple_t y;
    y = *px;
    *px = y->suivant; /* Fait pointer px sur la liste commençant à
     la deuxième cellule */
    free(y); /* Suppression de la première cellule */
}

void supprimerListeSentinelle(listesimple_t x){
    listesimple_t y;
```

```
    y = x->suivant;
    x->suivant = y->suivant; /* On fait commencer la liste à l'élément
suivant */
    free(y); /* Suppression du premier élément (seconde cellule) */
}
```

3.2 Piles

En anglais : *stacks*

Une pile est une structure de donnée que l'on rencontre très souvent dans la vie de tous les jours : pile de papiers, pile d'assiettes (ou si on est programmeur, pile d'exécution). Sa principale caractéristique est que l'élément que l'on retire est toujours le dernier élément ajouté et qui n'a pas encore été retiré, que nous appelerons *haut* de la pile. On parle de structure LIFO (*last in first out*). Les opérations fondamentales sont le test à vide, l'ajout d'un élément (appelé empilement), le retrait (dépilement) qui rend le dernier élément retiré et la lecture de l'élément en haut de la pile qui rend la valeur de cet élément sans le dépiler.

L'implantation d'une pile peut se faire à l'aide d'un tableau (voir TD) ou d'une liste simplement chaînée, le haut de la pile correspondant à la tête de la liste. Dans ce second cas, le test à vide correspond au test à vide des listes chaînées, l'empilement correspond à l'insertion en tête, la lecture du haut de la pile correspond à la lecture de l'élément de tête et le dépilement à une combinaison de lecture de l'élément de tête et de suppression. La fonction de dépilement modifie la pile et rend un élément. Elle peut s'écrire comme ceci, en passant la pile par adresse (argument-résultat) :

```
typedef listesimple_t pile_t;

element_t depiler(pile_t * px){
    pile_t y;
    element_t e;
    y = *px; /* y pointe sur le haut de la pile */
    e = y->element;
    *px = y->suivant; /* la pile commence désormais à l'élément du dessous */
    free(y);  /* libère la mémoire devenue inutilée */
    return e; /* rend l'ancien haut de pile */
}
```

3.3 Files

En anglais : *queues*

Une file est une structure de donnée que l'on rencontre aussi très souvent dans la vie de tous les jours : la file d'attente. L'élément que l'on retire est toujours le premier élément ajouté et qui n'a pas encore été retiré. On l'appelle tête de la file. Le dernier élément ajouté est l'élément de queue. On parle alors de structure FIFO (*first in first out*). Les opérations fondamentales sont les mêmes que celles des piles : test à vide, ajout, retrait et lecture de la valeur du prochain élément pouvant être retiré, sans faire le retrait.

L'implantation d'une file se fait à l'aide d'un tableau circulaire (voir TD) ou, le plus souvent, à l'aide d'une liste simplement chaînée. Dans le cas d'une implantation par liste chaînée, il faut enrichir un peu la structure : l'ajout se faisant à un bout de la liste et la suppression à l'autre bout, une file doit fournir l'adresse du premier et du dernier élément de la liste chaînée de ses

éléments. La tête de la liste sera la tête de file (où se fait le retrait) et le dernier élément de la liste son élément de queue (où se fait l'ajout). Voici le code d'une implantation en C des files basée sur les listes simplement chaînées.

```
typedef struct {
  liste_t tete;
  liste_t fin;
} * file_t;


int fileVide(file_t x){
  if (x->tete == NULL) return 1;
  else return 0;
}

element_t teteFile(file_t x){
  return (x->tete)->element; /* valeur de l'élément en tête de file */
}

element_t retirerFile(file_t x){
  element_t e;
  liste_t y;
  y = x->tete;    /* le début de la file */
  e = y->element; /* l'élément de tête */
  x->tete = y->suivant; /* la file débute à l'élément suivant */
  free(y);
  return e;
}

void ajouterFile(file_t x, element_t e){
  liste_t y;
  /* création de la cellule y, qui contiendra e */
  y = malloc(sizeof(liste_t));
  if (y == NULL) perror("panne mémoire");
  y->element = e;
  /* On place y à la fin de la liste */
  y->suivant = NULL;
  if (estVide(x)) x->tete = y; /* si x était vide, y devient la tete de liste */
  else (x->fin)->suivant = y; /* sinon, chaînage des deux derniers éléments */
  x->fin = y; /* Le pointeur de fin de file est désormais sur la cellule y */
}
```

3.4 Arborescences

Définition 3.1. Une *arborescence binaire* est une structure de donnée contenant un nombre fini d'éléments rangés dans des *nœuds*, telle que :

- lorsque la stucture n'est pas vide, un nœud particulier, unique, appelé *racine* sert de point de départ ;
- tout nœud x autre que la racine fait référence de manière unique à un autre nœud, appelé son *parent* et la racine n'a pas de parent ;
- chaque nœud peut faire référence à zéro, un ou deux nœuds fils, un fils gauche et un fils droit, dont le parent est alors nécessairement x ;

– Tous les nœuds ont la racine pour ancêtre commun (parent, ou parent de parent, ou parent de parent de parent, *etc.*).

Il y a au moins quatre opérations de lecture (qui ne modifient pas la structure) communes à toutes les structures de données basées sur les arborescences binaires : le test à vide qui prend un arbre en argument et rend vrai s'il ne contient pas de nœud et les trois opérations de lecture des références entre les nœuds. Ces trois opérations prennent en entrée un nœud. L'une rend son nœud parent (lorsqu'il existe), et les deux autres rendent respectivement le fils gauche et le fils droit (lorsqu'ils existent).

Définition 3.2. Plus généralement, une *arborescence* est comme une arborescence binaire mais où les fils d'un nœud peuvent aussi être en nombre supérieur à deux et sont donnés par une liste ordonnée.

Attention : dans une arborescence binaire on fait la distinction entre un nœud ayant seulement un fils gauche et un nœud ayant seulement un fils droit mais dans une arborescence (non binaire) on ne la fait pas (il s'agit deux fois d'un nœud ayant un seul fils).

En mathématiques, on peut donner une autre définition des notions d'arborescence et d'arborescence binaire équivalentes à celles-ci. Nous ne rentrerons pas dans ces détails. Une chose importante à retenir est qu'en mathématiques on parle plus volontier d'arbre et que dans ce cas, en générale, on désigne une structure dans laquelle on a pas encore choisi de nœud particulier pour jouer le rôle de la racine. Ainsi il y a une différence subtile entre la définition mathématique d'arbre et celle d'arborescence : une arborescence est un arbre dans lequel on a choisi une racine. Quelques mathématiciens parlent plutôt d'algue pour les arbres sans racine. Dans ce cours nous prenons le parti d'appeler arbre une structure arborescente et donc de donner une racine aux arbres.

La distance d'un nœud x à la racine est le nombre de fois où il faut remonter à un parent pour passer de x à la racine : si ce nœud est la racine cette distance est 0, si le parent de x est la racine c'est 1, *etc.*

la *profondeur* d'un nœud x dans un arbre est sa distance à la racine. La profondeur de la racine est donc 0, de ses fils éventuels 1, *etc.* Nous définissons la *hauteur* d'un arbre comme le maximum des profondeurs de ses nœuds, *plus un* (ce « plus un » est affaire de convention, et cette convention n'est pas la même partout). La hauteur d'un nœud est la différence entre la hauteur de l'arbre et la profondeur du nœud.

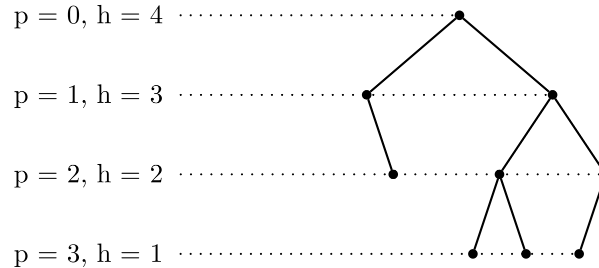


Fig. 3.1 – Exemple d'arbre binaire avec hauteur et profondeur des nœuds

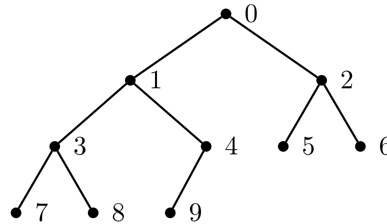
Un nœud qui n'as pas de descendant est une *feuille*. Un nœud qui n'est pas une feuille est parfois dénommé nœud interne (ou encore nœud *strict*).

3.4.1 Arbres binaires complets et quasi-complets

Définition 3.3. Un arbre binaire est complet lorsque son nombre de nœuds est $2^h - 1$ où h est sa hauteur.

Dans un arbre binaire complet tout nœud de hauteur strictement inférieure à h a ses deux fils (et les nœuds de hauteur h sont des feuilles). À profondeur $p \leq h-1$ on a 2^p nœuds et si $p = h-1$ ces nœuds sont les feuilles de l'arbre.

Arbre binaire quasi-complet. Ordonnons les nœuds d'un arbre binaire complet selon leur profondeur puis de gauche à droite. En premier on a la racine, puis ensuite ses deux fils (de profondeur 1), le gauche, puis le droit, ensuite les nœuds de profondeur 2, d'abord les fils du fils gauche de la racine, le gauche puis le droit, puis les fils du fils droit de la racine *etc.* Si on supprime un nombre quelconque de nœuds en partant des derniers pour cet ordre on obtient ce qu'on appelle un arbre binaire *quasi-complet.* Voici un exemple d'arbre quasi-complet où les nœuds sont numérotés dans l'ordre, en commençant à zéro. On ne représente pas les éléments contenus dans les nœuds.



On peut stocker un arbre binaire quasi-complet dans un tableau en plaçant ses éléments dans l'ordre (la racine à l'indice 0, *etc.*). On fait alors référence aux nœuds par leurs indices dans le tableau. Partant d'un nœud d'indice i on trouve : son nœud parent à l'indice parent(i) son nœud fils gauche à l'indice gauche(i) et son nœud fils droit à l'indice droite(i), avec :

$$\begin{aligned} \text{parent}(i) &= \lfloor \frac{i-1}{2} \rfloor \\ \text{gauche}(i) &= 2i+1 \\ \text{droite}(i) &= 2i+2. \end{aligned}$$

En C, on peut ainsi définir un arbre binaire quasi-complet comme un tableau. Voici un exemple d'implantation en C (voir aussi le TD pour des variantes dans le choix de l'indice de la racine, et de l'ordre dans lequel sont écrits les nœuds – ordre direct ou ordre inverse). On adjoint à la donnée du tableau quelques données auxilliaires : le nombre d'éléments de l'arbre et l'espace mémoire disponible dans le tableau. Ces données permettent de gérer plus efficacement la mémoire lors de l'ajout et de la suppression d'éléments (voir fichier C inclus en fin de chapitre).

```
typedef struct {
  element_t *tab; // le tableau
  int mem;        // Nombre maximum d'éléments dans le tableau
  int taille;     // nombre d'éléments stockés (taille =< mem)
} *arbrebqc_t;

int parent(int i){// indice du parent de t[i]
  return (i - 1)/2;
}

int gauche(int i){// indice du descendant gauche de t[i]
  return 2*i + 1;
}

int droite(int i){// indice du descendant droit de t[i]
  return 2*i + 2;
}

int racine(arbrebqc_t x){// la racine de l'arbre
```

```
  return 0;
}

int dernier(arbrebqc_t x){// indice du dernier élément
  return x->taille - 1;
}

int taillearbrebqc(arbrebqc_t x){// taille de l'arbre
  return x->taille;
}
```

3.5 Files de priorité (tas)

Une file de priorité est une structure de donnée qui permet de stocker des éléments ayant une *priorité* – comme pour les tris chaque élément possède une clé, sa priorité, ainsi que des données satellites – et telle que l'opération de retrait d'un élément rend toujours un élément de priorité maximum. Il est aussi possible d'ajouter un élément de priorité quelconque ou de modifier la clé d'un élément existant quelconque.

Un tas est une structure de donnée basée sur les arbres binaires quasi-complets, qui réalise efficacement les files de priorité, c'est à dire avec de bons résultats de complexité algorithmique sur les opérations fondamentale (ajout, retrait, modification).

Définition 3.4. Un *tas max* (respectivement *tas min*), ou *maximier* (resp. *minimier*, est un arbre binaire quasi-complet tel que tout nœud différent de la racine possède une clé (ou priorité) plus petite (resp. plus grande) que celle de son parent.

Les deux notions, tas max et tas min, sont symétriques, il suffit d'inverser l'ordre des clés pour passer de l'une à l'autre. On travaille ici avec des tas max.

Première conséquence de la définition de tas : dans un tas max non vide, l'élément de clé maximum est toujours à la racine.

Remarque : un tableau trié en ordre décroissant est un tas max.

Pour la suite, concernant l'implantation en C, on se donne une macro pour le pré-processeur pour simplifier l'adressage des éléments d'un arbre quasi-complet cette macro ne marche que si l'arbre est noté `x`) et une fonction d'échange similaire à celle des tableaux.

```
/* Macro pour simplifier l'accès aux tableau. */
#define x(indice) (x->tab[indice])
/* Exemple : x(i/2) sera remplacé par (x->tab[i/2]) */

void echangetas(arbrebqc_t x, int i, int j){
  element_t e;
  e = x(i);
  x(i) = x(j);
  x(j) = e;
}
```

3.5.1 Changement de priorité d'un élément dans un tas

Changer la priorité (clé) d'un élément dans un tas peut rendre un arbre qui ne satisfait plus la propriété de tas. On doit alors modifier l'arrangement des éléments dans l'arbre pour rétablir cette propriété. On dit qu'on maintient que la propriété de tas. La forme même de l'arbre reste inchangée, puisque le nombre d'éléments ne change pas.

Augmentation : maintien vers le haut. Si la priorité d'un élément e dans un nœud i augmente, il est possible qu'elle dépasse celle de son parent (lorsqu'il existe). Dans ce cas, on fait remonter l'élément en l'échangeant avec l'élément e' son parent $j = \text{parent}(i)$. Les descendants du nœud i ont bien des priorités inférieures à celle de e' puisque c'était le cas avant le changement de la priorité de l'élément e. Par contre il est possible que l'élément e qui est maintenant dans le nœud j n'ait pas une priorité inférieure à celle de son parent. Il faut donc recommencer la procédure avec le nœud j. Ceci a pour effet de faire remonter l'élément e vers la racine par échanges, jusqu'à ce qu'il se trouve dans un nœud dont le parent a une priorité supérieure, ou à la racine. La fonction `maintienTasHaut` est une implantation en C de cette opération de maintien.

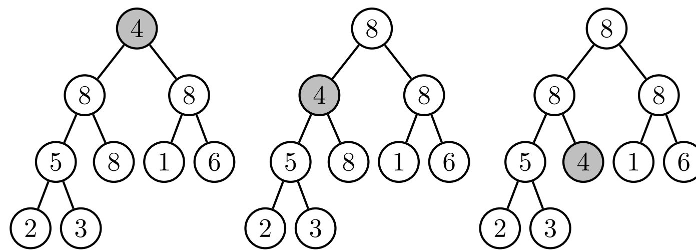
```
void maintienTasHaut(arbrebqc_t x, int i){
  if ( (i > 0) && (x(parent(i)).cle < x(i).cle) ){
    echangetas(x, parent(i), i);
    maintienTasHaut(x, parent(i));
  }
}
```

Diminution : maintien vers le bas. Si la priorité d'un élément e dans un nœud i augmente, il est possible qu'elle devienne inférieure à la priorité de l'un de ses deux descendants, ou des deux. On cherche alors parmi les nœuds i, gauche(i) et droite(i) lequel renferme l'élément de priorité maximale. Appelons j ce nœud. Il y a deux cas. Si $j = i$, il n'y a rien à faire la propriété de tas n'a pas été violée par la diminution de la priorité de e. Si j est l'un des fils de i, on échange le contenu des nœuds i et j. Cela a pour effet de rétablir localement la propriété de tas entre les nœuds i, gauche(i) et droite(i). Mais cette propriété peut maintenant être violée par le nœud j (l'un des fils), parce que sa priorité a diminuée. On recommence donc la procédure sur j. Ceci a pour effet de faire descendre e dans l'arbre par échanges successifs, jusqu'à ce qu'il se trouve dans un nœud dont chacun des fils a une priorité inférieure à celle de e.

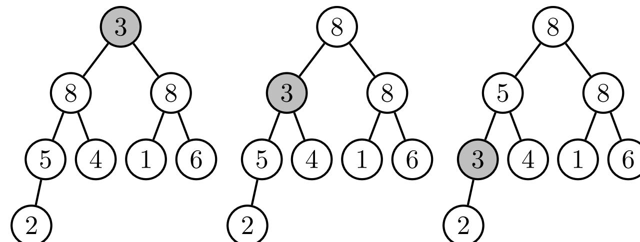
La fonction `maintienTasBas` est une implantation en C de cette opération de maintien. Deux exemples d'exécution de cette fonction sont donné ci-dessous : le nœud grisé est celui d'indice i (dans les deux exemples, au départ c'est la racine).

```
void maintienTasBas(arbrebqc_t x, int i){
  int imax;
  imax = i;                  /* option affichage */ AFFICHAGE_TAS_X_i;
  /* On cherche l'indice de l'élément maximum parmi i, gauche(i) et
     droite(i), lorsqu'ils existent */
  if ( (gauche(i) <= dernier(x)) && (x(gauche(i)).cle > x(imax).cle) )
    imax = gauche(i);
  if ( (droite(i) <= dernier(x)) && (x(droite(i)).cle > x(imax).cle) )
    imax = droite(i);
  /* Si ce maximum n'est pas en i on procède à un échange et on
     relance sur le noeud qui contenait le maximum */
  if ( imax != i ) {
    echangetas(x, i, imax);
    maintienTasBas(x, imax);
  }
}
```

Premier exemple :



Deuxième exemple :



3.5.2 Ajout et retrait des éléments

Ajout d'un élément. Pour ajouter un élément e dans un tas, on ajoute en fin de tableau un nœud n le contenant (sans modifier le contenu des autres nœuds). Le nœud n devient ainsi la dernière feuille de l'arbre. Le nœud n n'ayant pas de descendant la seule violation possible de la propriété de tas qu'ait pu introduire cette insertion est entre le nœud n et son parent. Il est en effet, possible que la priorité de e soit supérieure à celle du parent de n. On appelle donc la procédure de maintien vers le haut sur le nœud n.

```
void insererTas(arbrebqc_t x, element_t e){
  augmenterArbreBQC(x); // ajout d'un noeud final et gestion mémoire
  x(dernier(x)) = e;
  maintienTasHaut(x, dernier(x));
}
```

Retrait du maximum. Le retrait d'un élément dans un tas se fait toujours par retrait de l'élément de priorité maximum, c'est dire l'élément à la racine de l'arbre. Pour que la structure d'arbre quasi-complet reste correcte, après avoir retiré l'élément à la racine, on le remplace par l'élément de la dernière feuille de l'arbre (le dernier élément du tableau) et on supprime cette feuille. Comme la nouvelle priorité de la racine est inférieure à l'ancienne, on appelle sur la racine, la procédure de maintien vers le bas de la propriété de tas.

```
element_t retirerMax(arbrebqc_t x){
  element_t e;
  /* On prend l'élément maximum à la racine */
  e = x(racine(x));
  /* On remplace la racine par le dernier élément */
  x(racine(x)) = x(dernier(x));
  /* On diminue le nombre d'élément de l'arbre de un */
  diminuerArbreBQC(x);  /* Fait : x->taille-- et gestion mémoire */
  /* On reforme le tas */
  maintienTasBas(x, racine(x));
  /* On sort en rendant l'élément maximum */
  return e;
}
```

3.5.3 Formation d'un tas

Par une série d'insertions. Pour former un tas à partir d'une liste de n éléments, on peut commencer par un tas vide et lui ajouter les éléments un à un. Cet algorithme à une complexité en pire cas en $O(n \log n)$ échanges.

En effet, l'ajout d'un élément dans un tas de $k-1$ éléments prend au pire des cas $h-1$ échanges où h est la hauteur de l'arbre obtenu après l'ajout. Mais $2^{h-1}-1 < k$ donc $2^{h-1} \leq k$ ce qui donne : $h-1 \leq \log k$, par passage au log. Le nombre d'échanges en pire cas d'un ajout est donc inférieur à $\log k$.

On effectue l'ajout n fois, en commençant avec un tas à zéro élément puis en augmentant de un son nombre d'éléments à chaque fois. Le nombre total d'échanges est borné supérieurement par :

$$\sum_{k=1}^{n} \log k = \log \Pi_{k=1}^{n} k = \log(n!) = O(n \log n).$$

Algorithme en temps linéaire en pire cas. Il est toutefois possible de faire mieux que $O(n \log n)$, en changeant d'algorithme. On part de l'arbre quasi-complet A contenant tous les éléments et on applique à ses nœuds internes, en allant du dernier au premier, la procédure de maintien vers le bas de la structure de tas.

À la fin de ce nouvel algorithme, on obtient un tas. En effet, un arbre réduit à un seul nœud est toujours un tas. Donc dans A chaque feuille est déjà un tas. Et si on applique l'algorithme de maintien vers le bas à un nœud dont les fils sont déjà des racines de tas, alors l'arbre qui a pour racine ce nœud devient à son tour un tas. Ainsi, en procédant du dernier au premier nœud interne, à chaque étape, le nœud qui vient d'être traité est la racine d'un tas. Comme le dernier élément traité est la racine, A est bien transformé en tas. De plus, les seuls modifications apportées à A le sont par échange d'éléments entre des nœuds. L'ensemble des éléments à la fin est donc bien le même qu'au début.

```
void formerTas(arbrebqc_t x){
  int i;
  if ( taillearbrebqc(x) > 1 ){
    for (i = parent(dernier(x)); i >= racine(x); i--){
      maintienTasBas(x, i);
    }
  }
}
```

Complexité. On peut facilement montrer que cet algorithme est en $O(n \log n)$ en pire cas. En fait, on peut serrer plus la borne. On va montrer que le nombre d'échanges requis par ce nouvel algorithme pour former un tas de n éléments est linéaire en n en pire cas.

Pour simplifier les calculs, on se place dans le cas où l'arbre est complet : dans ce cas $n = 2^h - 1$ où h est la hauteur de l'arbre. Il est ensuite facile ensuite de généraliser comme on l'avait fait pour la recherche dichotomique (voir section **??** page 25).

En pire cas, le nombre d'échanges requis par l'application de l'algorithme de maintien vers le bas à un nœud m est $h'-1$ où h' est la hauteur du sous-arbre de racine m.

Dans l'arbre, il y a 2^{h-1} feuilles toutes de hauteur 1. Pour chaque i tel que $1 < i \leq h$, il y a 2^{h-i} nœuds internes de hauteur i. Si m est un nœud de hauteur i, la hauteur d'un sous-arbre de racine m est i.

Le nombre total d'échanges est donc majoré en pire cas par la somme :

$$E(h) = \sum_{i=1}^{h} i 2^{h-i} = 2^{h-1} \left(\sum_{i=1}^{h} i \left(\frac{1}{2}\right)^{i+1} \right)$$

Pour évaluer cette somme on pose :

$$f(z) = \sum_{i=1}^{h} z^i$$

On a alors $E(h) = 2^{h-1} f'(1/2)$, il reste à évaluer $f'(1/2)$.
On a :

$$f(z) = \frac{z^{h+1} - 1}{z - 1}$$

D'où :

$$f'(z) = \frac{(h+1)z^h(z-1) - (z^{h+1} - 1)}{(z-1)^2}$$

$$f'\left(\frac{1}{2}\right) = -\frac{(h+1)(1/2)^{h+1} + ((1/2)^{h+1} - 1)}{(1/2)^2}$$

Comme $\sum_{i=1}^{h+k} i\left(\frac{1}{2}\right)^{i+1} \geq \sum_{i=1}^{h} i\left(\frac{1}{2}\right)^{i+1}$ pour $k \geq 0$ on obtient :

$$f'\left(\frac{1}{2}\right) \leq \lim_{h\to\infty} \frac{-(h+1)(1/2)^{h+1} - (1/2)^{h+1} + 1}{(1/2)^2} = 4$$

D'où $E(h) \leq 4 \times 2^{h-1} = 4 \times (n+1)/2$ de quoi l'on déduit facilement que $E(h) = O(n)$, ce qui conclut.

3.5.4 Le tri par tas

En anglais : *heap sort, Williams, 1961*

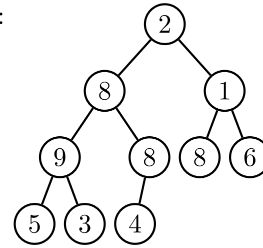
La structure de tas, permet d'écrire un algorithme de tri en place optimal en temps, c'est à dire en $\Theta(n \log n)$. Le principe est de prendre un tableau en entrée, d'y former un tas ($O(n)$) et de retirer un a un les maximums successifs en les rangeant à la fin du tableau (n fois $O(\log n)$). En C, cela donne le code suivant :

```
void triTas(tableau_t *t){
  /* 1) On tranforme le tableau en un arbre bqc */
  arbrebqc_t x;
  x = newArbreBQC(); /* allocation mémoire */
  x->tab = t->tab; /* On fait juste référence au tableau sans le
      recopier. On travaille donc "en place".*/
  x->mem = t->taille;
  x->taille = t->taille;
  /* 2) On forme le tas */
  formerTas(x);                 /* option affichage */ AFFICHAGE_TAS_X;
  /* 3) On extrait les maximums successifs en les plaçant en fin de
        tableau */
  while (x->taille > 1) {
    /* a) On place le maximum à la fin du tas */
    echangetas(x, racine(x), dernier(x));
    /* b) On diminue de un le nombre d'éléments du tas, en préservant le
       reste du tableau */
    x->taille--;
    /* c) On reforme le tas */
    maintienTasBas(x, racine(x));
```

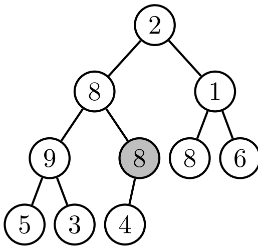
```
  }
  /* Fin) le tableau t->tab est trié, t->taille a la bonne valeur, on
     peut libérer l'espace mémoire pris par x. */
  free(x);
}
```

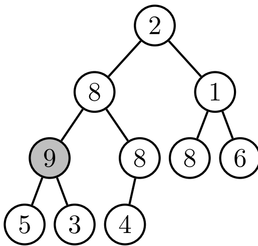
Voici un exemple d'exécution du tri, à partir du tableau : [2 8 1 9 8 8 6 5 3 4].

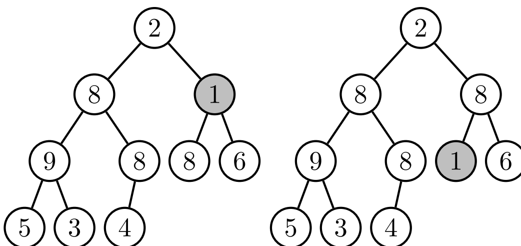
1. On regarde le tableau comme un arbre quasi-complet :

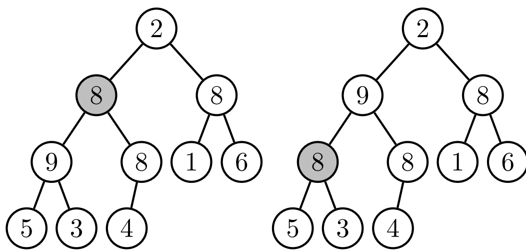


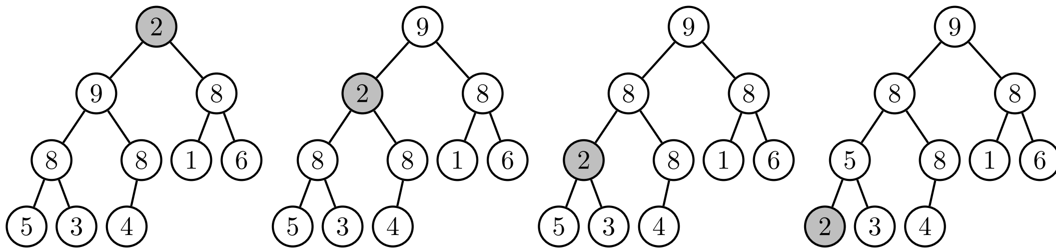
2. On forme le tas en faisant appel à la fonction `maintienTasBas` sur chaque nœud interne en allant du dernier au premier. Certains de ces appels donnent lieu à de nouveaux appels de la fonction `maintienTasBas` (appels récursifs).



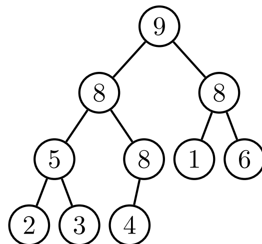






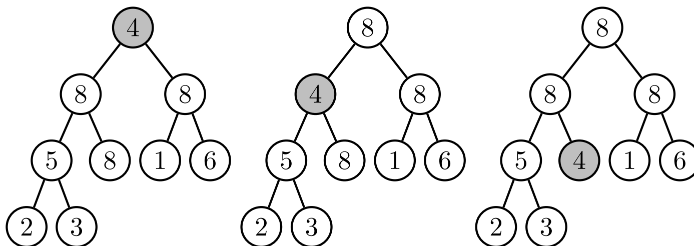


On obtient le tas :



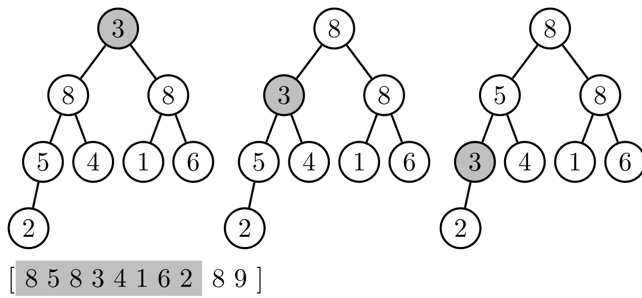
Le tableau est donc maintenant : [9 8 8 5 8 1 6 2 3 4].

3. On extrait l'élément maximum (à la racine) en l'échangeant avec le dernier élément du tableau (la dernière feuille) ; on sort ce dernier élément de l'arbre ; et on reforme le tas.

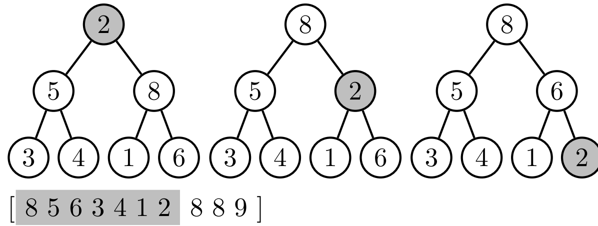


Le tableau est maintenant : [8 8 8 5 4 1 6 2 3 9] (en grisé, les éléments encore dans l'arbre).

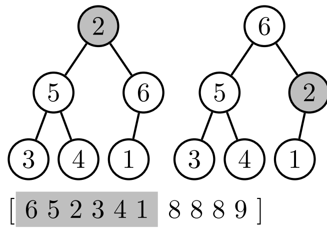
On recommence avec le nouveau maximum :



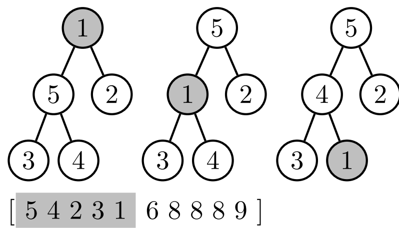
Encore. . .



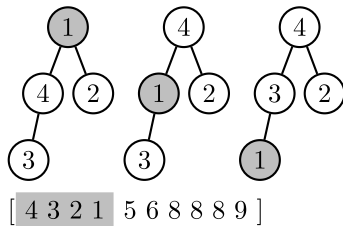
. . . et encore. . .



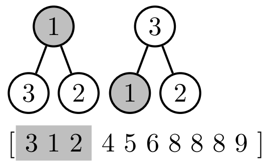
. . . et encore. . .



. . . et encore. . .

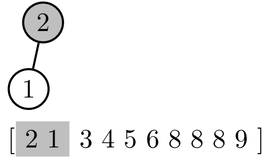


. . . et encore. . .



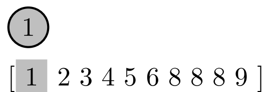
[3 1 2 4 5 6 8 8 8 9]

. . . et encore. . .



[2 1 3 4 5 6 8 8 8 9]

. . . jusqu'au dernier élément.



[1 2 3 4 5 6 8 8 8 9]

4. On obtient le tableau [1 2 3 4 5 6 8 8 8 9].

3.5.5 Complément : gestion mémoire

À titre de complément, voici une implantation minimaliste de la gestion mémoire pour les tas.

```
/* Taille des (re)allocations mémoire */
# define ARBREBQC_MEM_SEG 256

arbrebqc_t newArbreBQC(void){
  arbrebqc_t x;
  x = malloc(sizeof(arbrebqc_t));
  if (x == NULL) perror("Echec malloc");
  x->tab = NULL;
  x->taille = 0;
  x->mem = 0;
  return x;
}

void detruireArbreBQC(arbrebqc_t x){
  free(x->tab);
  free(x);
}

void augmenterArbreBQC(arbrebqc_t x){
  x->taille++;
  if (x->mem < x->taille) {
    /* augmentation de l'espace mémoire du tableau */
    element_t *tmp;
    tmp = realloc(x->tab, x->mem + ARBREBQC_MEM_SEG);
    if (tmp == NULL) perror("Ã©chec realloc");
    x->tab = tmp;
    x->mem = x->mem + ARBREBQC_MEM_SEG;
  }
}
```

```
void diminuerArbreBQC(arbrebqc_t x){
  x->taille--;
  if (x->mem >= x->taille + ARBREBQC_MEM_SEG) {
    /* diminution de l'espace mémoire du tableau */
    element_t *tmp;
    tmp = realloc(x->tab, x->mem - ARBREBQC_MEM_SEG);
    if (tmp == NULL) perror("Echec realloc");
    x->tab = tmp;
    x->mem = x->mem - ARBREBQC_MEM_SEG;
  }
}
```