IUT De Villetaneuse
Dép[t] informatique

Cours d'algorithmique
2[éme] Année

Cours 6

Une utilisation des arbres

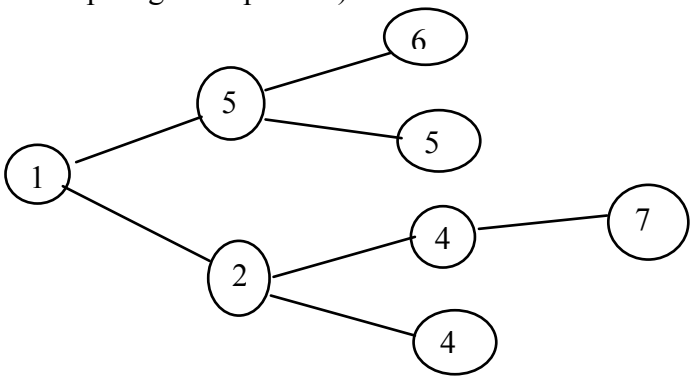
---

1. Une nouvelle structure de donnée : la file de priorité

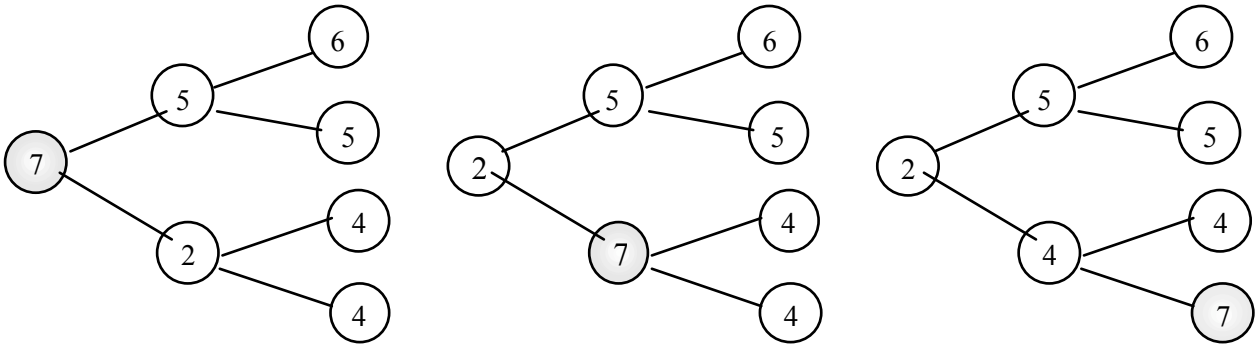
Le problème à résoudre : un certain nombre d'éléments (processus, clients, requêtes dans une BD, …) se présentent à différent moments pour utiliser une ressource. Quand celle-ci n'est pas libre, ils doivent attendre, et quand elle se libère on doit choisir l'élément le plus prioritaire. Comment tenir à jour la liste des éléments en attente ?

Structure de donnée : file de priorité =
un arbre binaire chaque nœud contient un nombre (une *priorité*) et tel que tout nœud a une priorité supérieure ou égale à celle de chacun de ses fils.

exemple : (le plus petit nombre a la plus grande priorité)



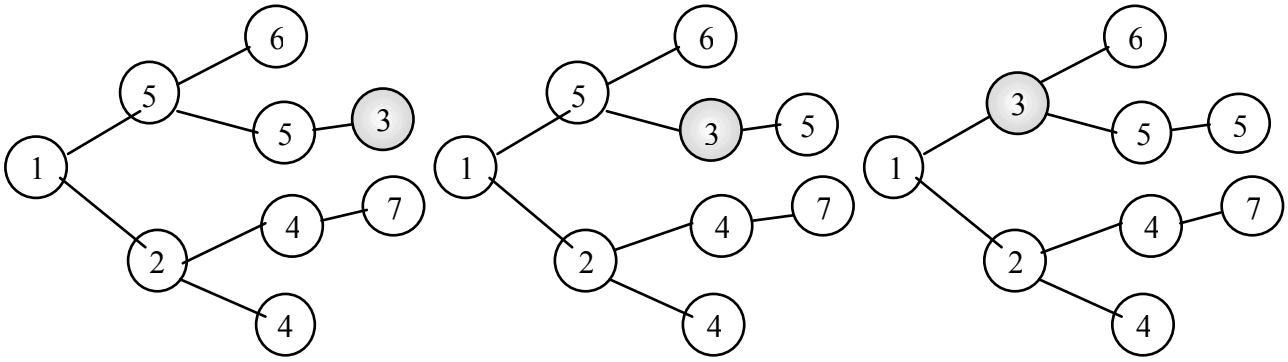
Opérations :
a. prélever le nœud le plus prioritaire (la racine) ⇒ il faut reconstruire l'arbre.



1. ……

2. tant que

b. insérer un nouveau nœud.



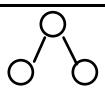
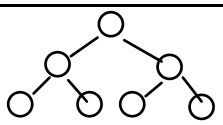
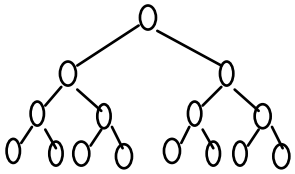
1. …… 2. tant que

1. Files de priorité, arbres quasi-complets et tas

Difficulté : ces opérations sont répétées assez souvent. Si l'on choisit mal la feuille que l'on met à la racine ou celle ou l'on ajoute le nouveau nœud, .

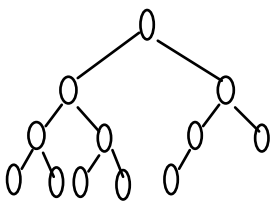
1.1. Arbre binaire quasi complet

⇒ **arbre binaire complet** : toutes les feuilles sont de même profondeur, et tous les nœuds non-feuille ont deux fils exactement.

arbre				
profondeur	0	1	2	3
nombre de noeuds	1	3	7	15

Arbre binaire de profondeur n . … feuilles, (……) nœuds

→ on ne peut pas toujours avoir un arbre complet



⇒ **Arbre binaire quasi complet de profondeur n** : toutes les feuilles sont de profondeur n ou n-1 ; à la profondeur n-1, les nœuds non feuille sont à gauche (les premiers dans l'ordre largeur d'abord) et les feuilles à droite ; tous les nœuds non feuille ont exactement deux fils, sauf peut-être le dernier nœud non feuille de profondeur n – 1 .

Arbre quasi complet de profondeur n : entre 2^n et $2^{n+1} - 1$ nœuds.

donc quel que soit le nombre de nœuds, on peut les ranger dans un arbre quasi complet.

1.1. Tas

Définition : c'est une file de priorité qui forme un arbre quasi complet.

Opérations sur un tas :

a. **prélever** : comme pour une file de priorité, mais on choisit le dernier nœud feuille pour le mettre à la racine

⇒ l'arbre reste quasi complet pendant toute l'opération. Après réorganisation, le nouvel arbre est encore un tas.

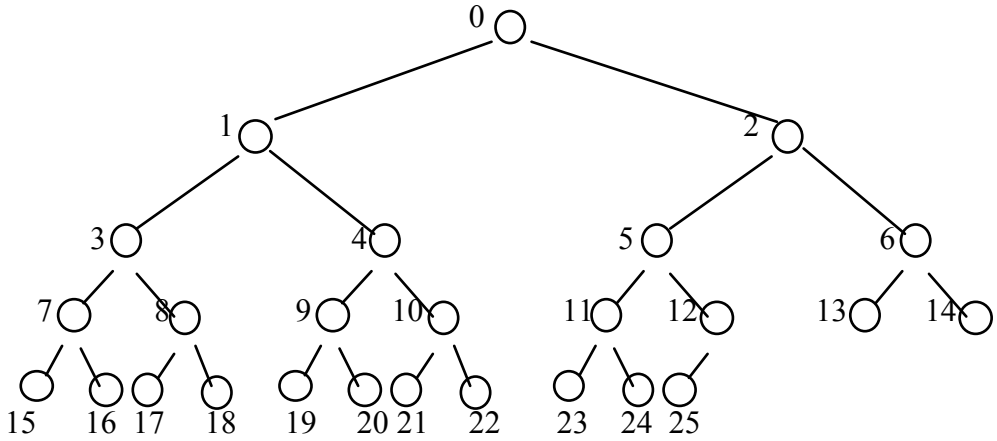
b. **insérer** : comme pour une file de priorité, mais on ajoute le nouveau nœud juste après le dernier nœud feuille (et on réorganise).

⇒ l'arbre reste quasi complet pendant toute l'opération. Après réorganisation, le nouvel arbre est encore un tas.

Conclusion : si l'on utilise un tas pour réaliser une file de priorité, l'arbre reste toujours équilibré.

1.1. Implémentation des arbres quasi complets (et des tas)

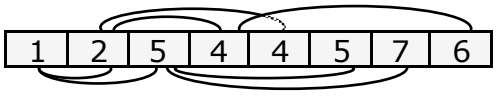
Il existe une implémentation particulièrement simple. Pour comprendre comment elle fonctionne, il faut numéroter les nœuds en largeur d'abord.



Les deux fils du nœud numéro n ont les numéros …… et ……

/* Implémentation d'un tas de taille au maximum t dans un tableau */
variables : Elem tabElem[t] ; int nbElem ;
/* nbElem est le nombre d'éléments dans le tableau */
procédures : / le fils gauche de tabElem[n] est tabElem[……], son fils droit est tabElem[……], son père est tabElem[……] */

Utiliser 1 et 2.2 pour écrire les procédures



1. Mesures de qualité d'un algorithme

1.1. Principe

Quand on veut comparer deux algorithmes qui exécutent (différemment) la même tâche, on va comparer principalement :

- le temps mis pour faire le calcul
- l'espace mémoire occupé par ce calcul (le plus grand espace utilisé pendant l'exécution)

Comme la réponse dépend des données sur lesquelles on exécute l'algorithme, on calcule un **ordre de grandeur** :

à quelle vitesse le temps ou l'espace consommés augmentent-ils quand le nombre de données augmente ?

Dans ce cours, on s'intéresse surtout au temps. On a deux mesure du temps : le **temps moyen** mis par un algorithme pour traiter un ensemble de données de taille n, et le **temps maximum** (le pire des cas, sur l'ensemble de données de taille n le plus défavorable). On dit qu'un algorithme s'exécute en temps O(f(n)) (en moyenne, dans le pire des cas) si il existe deux constantes k_1, k_2 telles que le temps considéré soit entre $k_1 f(n)$ et $k_2 f(n)$. Exemple : n désigne la taille de l'ensemble de données.
Un algorithme est **en temps moyen O(n)** si ……
Approximativement, sii on a le double de données à traiter, il faudra (en moyenne) 2 fois plus de temps.

Exemple :
Lire le $i^{ème}$ élément de la liste l de taille n (accès à un élément par son rang). Unité de temps : temps pour lire le suivant (lire une adresse + lire à cette adresse, donc temps constant). Il faut lire i éléments, donc temps i, avec : i peut valoir 1 ou 2 ou … ou n, avec la même probabilité
→ temps moyen $(1 + 2 + \ldots + n) / n$ soit $n (n + 1) / 2$ $n = (n + 1) / 2 \approx O(n)$
Si la taille de la liste est doublée, le temps moyen d'accès à un élément par son rang est doublé (ça n'est pas tout à fait exact, mais la différence est négligeable).
Exemple :
prélever l'élément le plus prioritaire d'un tas de taille n : ……
……
insérer un élément dans un tas de taille n : idem

1.2. Exemple : tri par recherche du maximum

On a un ensemble de données que l'on veut trier. Les données sont dans un tableau de taille n. Le principe de l'algorithme est le suivant : on cherche dans le tableau le plus grand élément ; on le permute avec le dernier ; puis on recommence avec le tableau sauf le dernier. Au bout de n itérations, le tableau est trié.

Le temps consommé peut être estimé :
- temps pour rechercher le maximum dans un tableau de taille t :
- temps pour permuter 2 éléments :
- temps pour les n itérations :
Donc on a une complexité en $O(n^2)$: si on trie 10 fois plus d'éléments, le temps de calcul est multiplié par environ …

Importance pratique de la complexité : exemple d'un algorithme qui traite 100 données en 1 seconde → temps de calcul pour 1000 données

Complexité	O(n)	$O(n^2)$	$O(n^3)$	$O(n^4)$	$O(2^n)$
temps approximatif	10 sec	1 mn 40 sec	16 mn 40 s	2 h 46 mn 40 sec	10^{270} sec $> 10^{260}$ années

1.3. Exemple : tri par tas

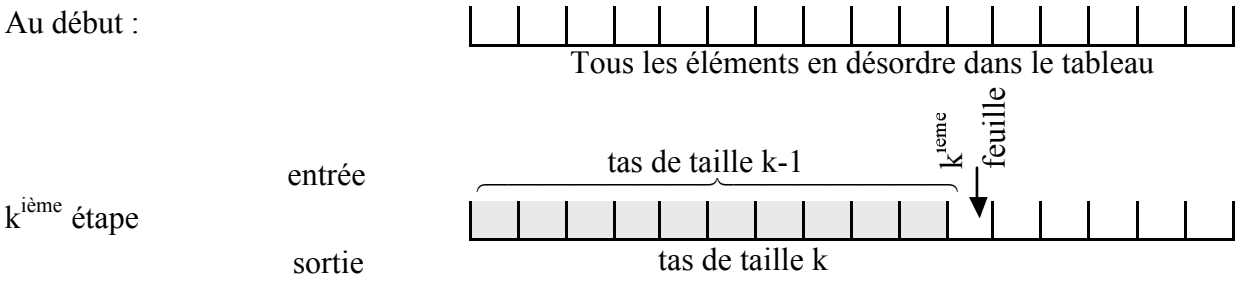
C'est la même idée que le tri par recherche du maximum, mais on organise les données dans le tableau pour qu'elles forment un tas. Si l'on veut trier en ordre croissant, on organisera le tas pour que le père contienne toujours une valeur au moins égale à celle de ses fils.

Au départ les données sont en désordre dans le tableau, comme dans le tri par recherche du maximum. On a donc 2 phases :

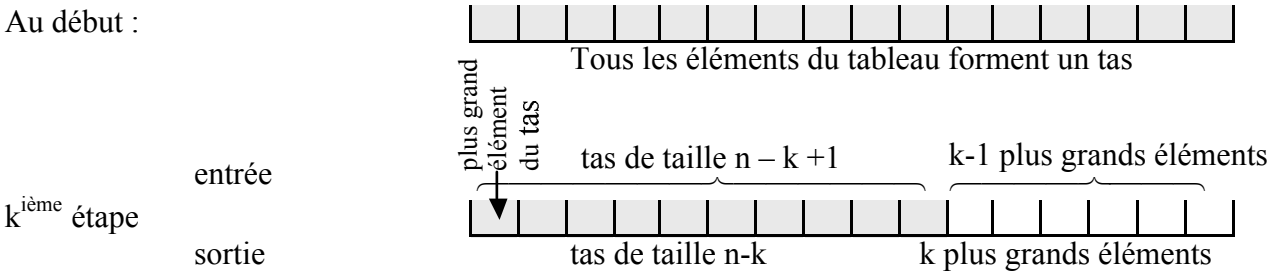
i) construire le tas en insérant les nœuds un par un

ii) tri par recherche du maximum dans un tas : prélever le plus grand élément et le mettre à la fin, à la place de la feuille qui sert à reconstruire le tas.

Phase i :



Phase ii :



Complexité de la phase i :
$\log_2(1) + \log_2(2) + \ldots + \log_2(n) = \log_2(n!)$

Complexité de la phase ii :
$\log_2(n) + \log_2(n-1) + \ldots + \log_2(1) = \log_2(n!)$

Complexité totale :
$O(\log_2(n!)) \leq O(\log_2(n^n)) = \mathbf{O(n.\log_2(n))}$

Si l'on multiplie le nombre de données par 10, on multiplie le temps par moins de $10.\log_2(10) \cong 30$ (au lieu de 100 pour le tri par recherche du maximum).