05 Implémentations et applications des piles, files et files de priorité

niveau : MP2I prérequis : Connaissance de ces types abstraits

Les types LIFO/FIFO et files de priorité sont systématiquement manipulés dans une variété de contexte producteurs/consommateurs (ordonnancement, planification), si bien que l'efficacité de leurs implémentations est critique pour les performances desdites applications.

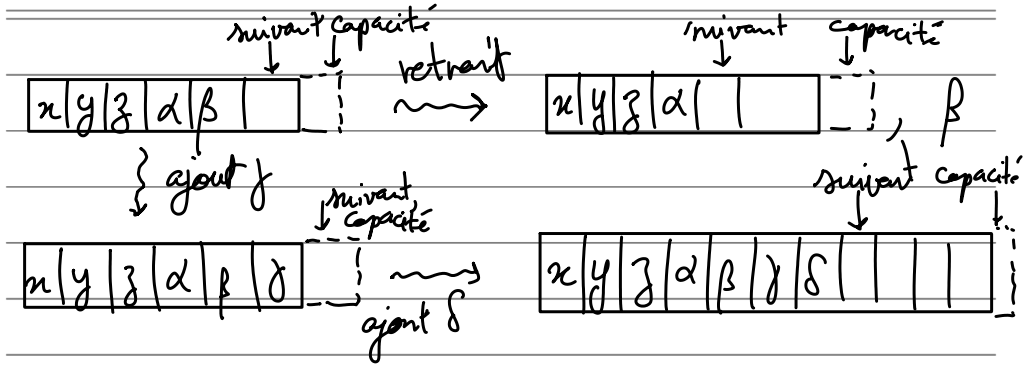
(I) Implémentation

Les spécifications de ces types peuvent légèrement varier. On met ici l'accent sur le caractère borné de la taille.

> on pourrait aussi redonner une définition succinte des types

1. Piles

Il s'agit d'utiliser un tableau augmenté d'un pointeur fournissant la prochaine case.
La taille du tableau détermine sa capacité.
Si bornée, on renvoie une erreur, sinon, on agrandit la capacité du tableau.



en C, on propose :

```
void ajout(pile_t p, int elt) {
  if (p.suivant == p.capacite) {
    p.capacite *= 2;
    p.tampon = reallocarray(p.tampon, p.capacite,
                            sizeof(*p.tampon));
  }
  p.tampon[p.suivant++] = elt;
}

int retrait(pile_t p) {
  p.suivant--;
  assert(p.suivant >= 0);
  return p.tampon[p.suivant];
}
```

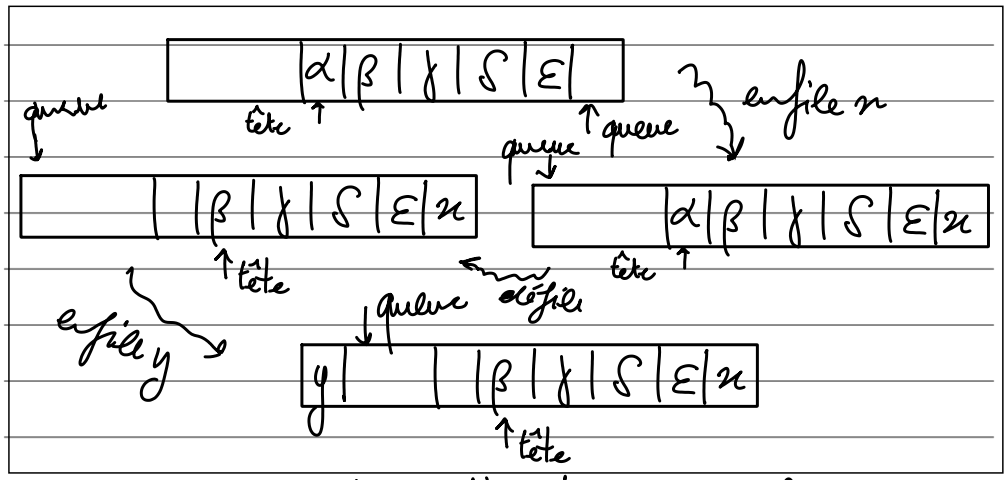
sans préciser les types exacts. La capacité est ici dynamique. On double la capacité de la pile à chaque dépassement.

Une implémentation avec des listes chaînées est également possible.

!OO En raison de l'utilisation du cache et de prédicteurs d'accès, les structures contiguës sont à privilégier dans ces situations.
→ Cela pourrait faire l'objet d'une illustration.

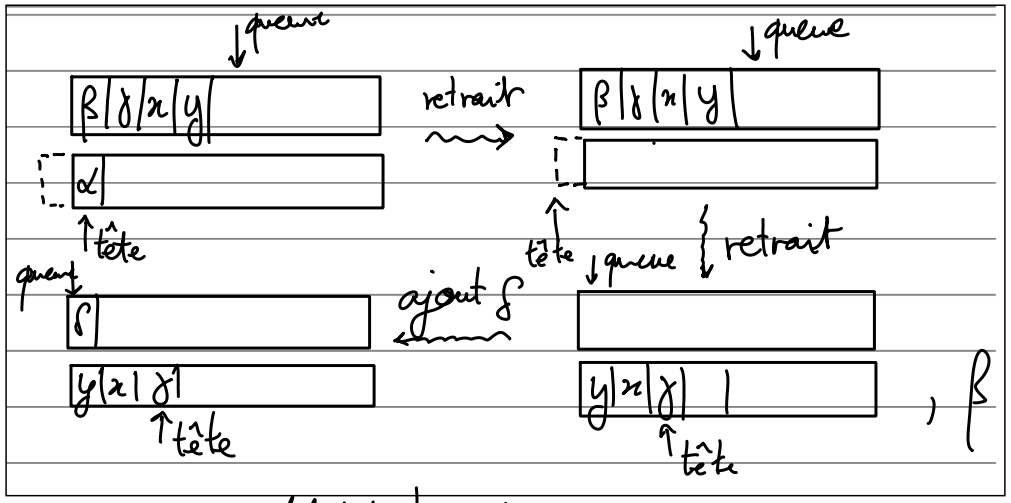
2. Files

Fondamentalement, il s'agit de retenir les positions des éléments extrêmaux. Plusieurs implémentations possibles :



Utilisation d'un tampon circulaire

!OO Pour ajuster la taille d'un tampon circulaire, il faut introduire une opération supplémentaire qui effectue replace les éléments contiguëment, avec la tête au début du tableau.



Utilisation de 2 piles

→ Toutes les opérations sont en $\mathcal{O}(1)$ (éventuellement amorti)

3. Files de priorité

Une première implémentation est le tas binaire : on utilise un tableau vu comme un arbre
- à l'ajout d'un élément, celui-ci est d'abord placé en feuille, puis "remonte" le long de l'arbre.
- au retrait du sommet (de priorité maximale), on le substitue avec une feuille, puis on fait "redescendre" le sommet substitué.

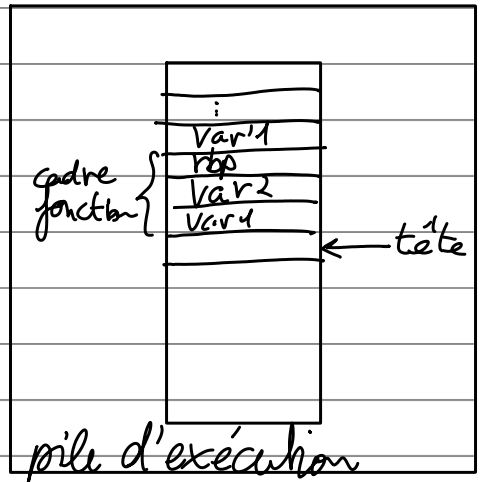
Cette implémentation effectue toutes les opérations voulues en $\mathcal{O}(\log n)$. Une autre approche est celle des tas binomial DEV1, qui propose également un ajout d'élément en

$\mathcal{O}(1)$ amorti, ainsi qu'une opération de fusion.

II. Applications

Une application directe et complète de la structure de tas est celle du tri par tas. DEV2, un tri optimal et en place.

Une autre application plus implicite mais omniprésente est celle de la pile d'exécution : les programmes maintiennent une partie de leur état sur une pile (variables locales, retour d'appels).



→ Ainsi, de nombreux algorithmes reposent sur cette structure indirectement.

Par exemple : le parcours en profondeur récursif utilise une telle structure pour garantir l'ordre de visite des sommets. D'autres structures permettent d'autres ordres : une file permet de réaliser un parcours en largeur.

Ces différents ordres permettent de mettre en place une variété d'algorithmes :
- recherche de composantes fortement connexes (pile, Kosaraju) ;
- recherche du plus court chemin (BFS non pondéré, files de priorité pour Dijkstra) : dans ce cas, le tas permet de maintenir un ensemble de sommets à traiter, dans l'ordre croissant des distances au sous-graphe exploré.

```
def dfs(u, graph, f, visite):
    visite[u] = True
    f(u)
    for v in graph.voisins[u]:
        if not visite[v]:
            dfs(v, graph, f, visite)
```

```
ENTRÉE : G=(S,A,p) pondéré, s_deb ∈ S
DONNÉE : T un tas, d tableau distances
SORTIE : d
    ajout(T, s_deb, 0)
d := [+∞ ; S], d[s_deb] := 0
Tant que T ≠ ∅ :
    a := extraire_min(T)
    Pour tout b, (a,b) ∈ A :
        dist = d[a] + p(a,b)
        si d[b] > dist :
            ajout(T, b, dist)
            d[b] := dist
```