Autres structures

20 novembre 2023

Plan

Administratif

Retour sur la semaine dernière

Tableau dynamique

Administratif

Devoirs

- Je récupère les DM ;
- vous avez un DS d'informatique la semaine prochaine.

Devoir Surveillé 2

Au programme :

- **Fonctions d'ordre supérieurs ;**
- Types construits en OCaml ;
- Structures linéaires.

Le sujet sera plus difficile que le premier, et noté plus durement, en particulier sur votre code. En l'absence de justification pour les questions théoriques, vous n'aurez peu ou pas de points.

Information sur le programme

À la fin du semestre vous aurez deux options :

- Continuer en informatique, et vous diriger vers la MPI l'année prochaine ;
- S'orienter vers la SI et repartir en MP option SI ou en PSI l'année prochaine.

Dans les deux cas, il est probable que vous restiez dans la classe, mais vous aurez des différences d'emplois du temps au second semestre pour rattraper l'informatique. (Pas de certitude encore sur l'organisation)
Vous ne pouvez pas vous diriger vers une MP option Informatique.

Différences de programme en informatique

- En MPI : OCaml, C (et SQL), de la théorie (logique, étude théorique de la complexité, de la calculabilité), du système, de l'algorithmique (6h par semaine) ;
- En MP Info Tronc Commun : Python et SQL, un peu de théorie, de l'algorithmique (2h par semaine).

Différences de programme dans les autres matières

Vous devrez rattraper la chimie principalement pour ce qui est de la physique. Vous devrez aussi rattraper le Python de l'informatique tronc commun (les élèves de MPSI et PCSI ont fait du python à leurs deux semestres).

Quelques indications pour l'orientation

- Vous ferez de l'informatique quoi qu'il arrive quand vous faites des sciences ;
- Cependant, si vous voulez spécifiquement faire de l'informatique, la formation proposée en tronc commun n'est qu'une base.

Semaines à venir

- Pas de nouvelles de la classe mobile avec les ordinateurs sous Linux ;
- on commence le C le 4 décembre.

Retour sur la semaine dernière

Listes chaînées

```
type 'a listmut = Vide | Cell of {valeur: 'a ;
    mutable suiv: 'a listmut}
type 'a liste_chainee = {mutable premier : 'a
    listmut}
```

Attention : le type `'a listmut` n'est pas un enregistrement, on ne peut pas le manipuler pour accéder à des champs comme si c'était le cas.

```
lc.premier.suiv
```

Listes chaînées

```
type 'a listmut = Vide | Cell of {valeur: 'a ;
    mutable suiv: 'a listmut}
type 'a liste_chainee = {mutable premier : 'a
    listmut}
```

Attention : le type `'a listmut` n'est pas un enregistrement, on ne peut pas le manipuler pour accéder à des champs comme si c'était le cas.

```
lc.premier.suiv
```

```
match lc.premier with
| Vide -> ...
| Cell c -> c.suiv
```

Listes chaînées

```
1 type 'a listmut = Vide | Cell of {valeur: 'a ;
    mutable suiv: 'a listmut}
2 type 'a liste_chainee = {mutable premier : 'a
    listmut}
```

Listes chaînées

```
type 'a listmut = Vide | Cell of {valeur: 'a ;
    mutable suiv: 'a listmut}
type 'a liste_chainee = {mutable premier : 'a
    listmut}
```

```
let ajouter_dernier lc x =
    let rec aux lm = match lm with
    | Vide -> Cell {valeur = x ; suiv = Vide}
    | Cell c -> Cell {valeur = c.valeur ; suiv = aux
     c.suiv}
    in lc.premier <- aux lc.premier
```

Listes chaînées

```
type 'a listmut = Vide | Cell of {valeur: 'a ;
    mutable suiv: 'a listmut}
type 'a liste_chainee = {mutable premier : 'a
    listmut}
```

```
let ajouter_dernier lc x =
    let rec aux lm = match lm with
    | Vide -> Cell {valeur = x ; suiv = Vide}
    | Cell c -> Cell {valeur = c.valeur ; suiv = aux
     c.suiv}
    in lc.premier <- aux lc.premier
```

Attention : Ce code crée intégralement une nouvelle liste.

Listes chaînées

```
let ajouter_dernier lc x =
    let nouvelle = Cell {valeur = x ; suiv = Vide}
    in
    let rec aux l1 l2 = match l1, l2 with
    | Vide, _ -> failwitth "Cas impossible"
    | _, Cell c -> aux l2 c.suiv
    | Cell c, Vide -> c.suiv <- nouvelle
    in match lc.premier with
    | Vide -> lc.premier <- nouvelle
    | Cell c -> aux lc.premier c.suiv
```

Listes chaînées

```
1 let retirer_dernier lc =
2     let rec aux l1 l2 = match l2 with
3     | Vide -> failwith "Cas impossible"
4     | Cell c when c.suiv = Vide -> l1.suiv <- Vide
5     | Cell c -> aux l2 c.suiv
6     in match lc.premier with
7     | Vide -> failwith "Rien a retirer"
8     | Cell c when c.suiv = Vide -> lc.premier <-
      Vide
9     | Cell c -> aux lc.premier c.suiv
```

Listes chaînées

Nous reverrons les listes chaînées en C dans la seconde moitié du semestre.

Tableau dynamique

Tableau dynamique (1)

On a deux structures linéaires principales en OCaml : les listes pour lesquelles l'accès peut être long, et les tableaux dont on ne peut pas modifier la taille.
On aimerait avoir une structure qui donne les avantage de ces deux structures. On peut utiliser une structure spécifique : le tableau dynamique.

Tableau dynamique (2)

Un **tableau dynamique** est une structure linéaire de taille variable dans laquelle on peut accéder en temps constant aux éléments quelqu'en soit la position.
L'idée est d'avoir une mémoire plus grande que les éléments réellement utilisés :

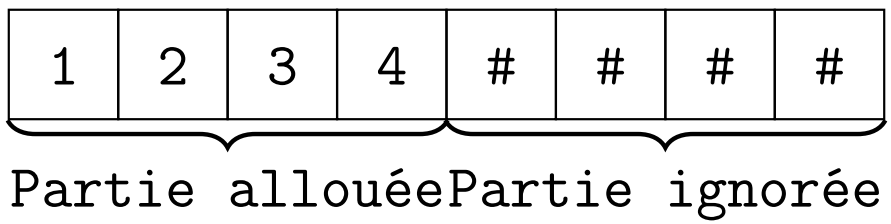
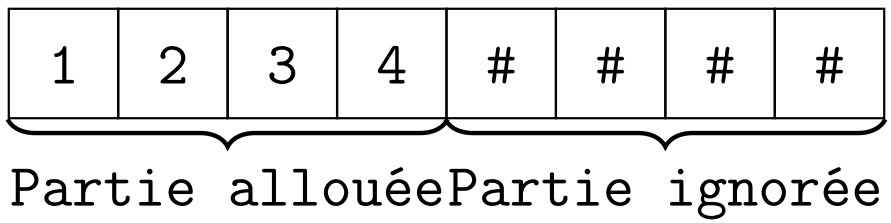


Tableau dynamique (2)

Un **tableau dynamique** est une structure linéaire de taille variable dans laquelle on peut accéder en temps constant aux éléments quelqu'en soit la position.
L'idée est d'avoir une mémoire plus grande que les éléments réellement utilisés :



On peut ainsi ajouter des valeurs à la fin du tableau dynamique sans modifier la place occupée dans la mémoire

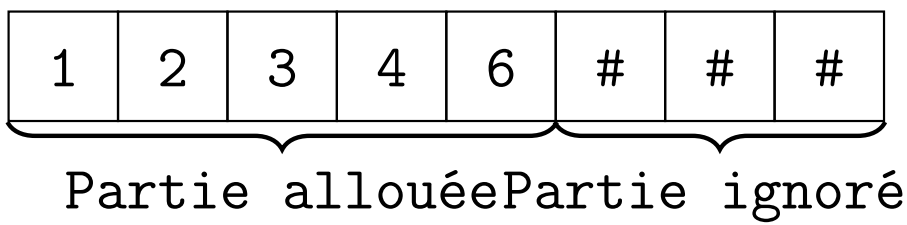


Tableau dynamique (3)

Lorsqu'on veut ajouter un élément alors que la mémoire est pleine, on a besoin de procéder à un redimensionnement :

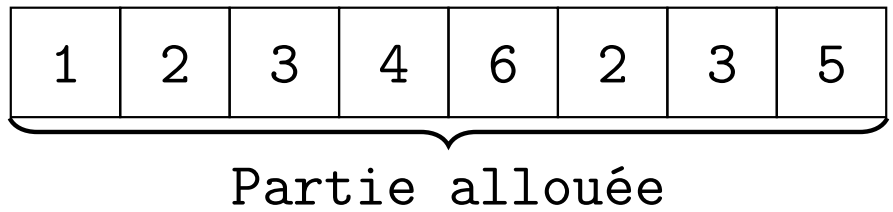
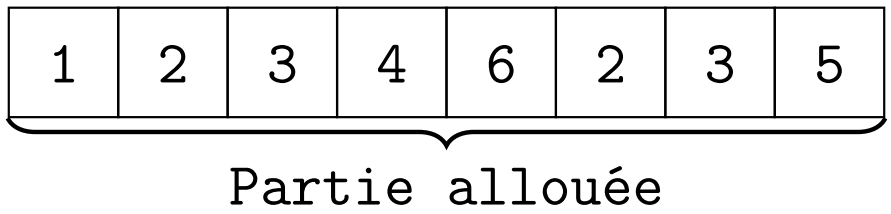


Tableau dynamique (3)

Lorsqu'on veut ajouter un élément alors que la mémoire est pleine, on a besoin de procéder à un redimensionnement :



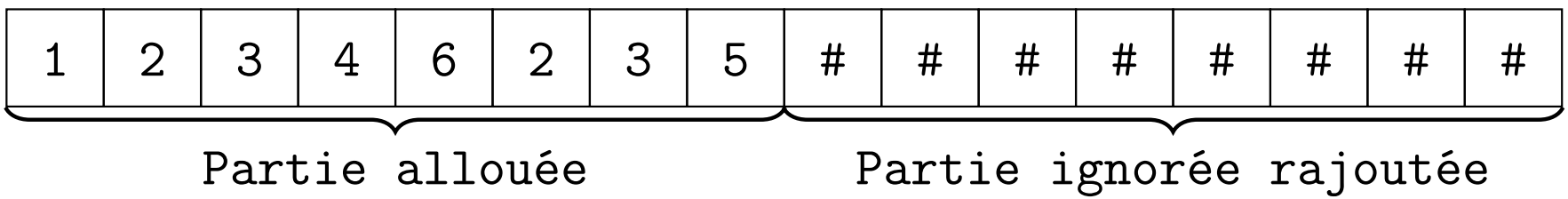
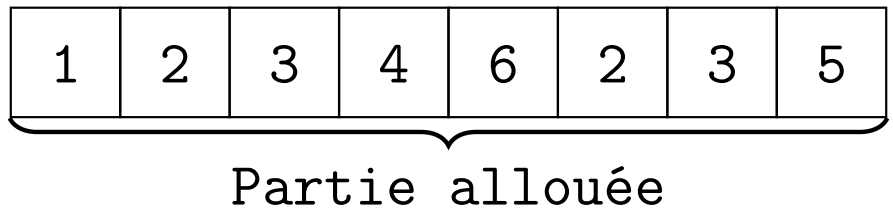
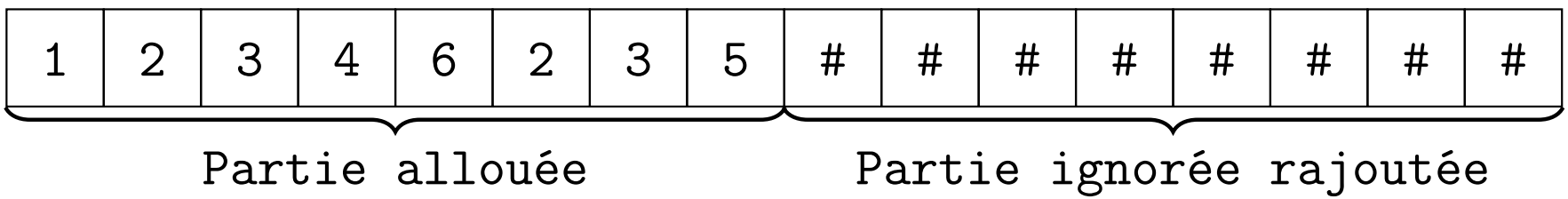


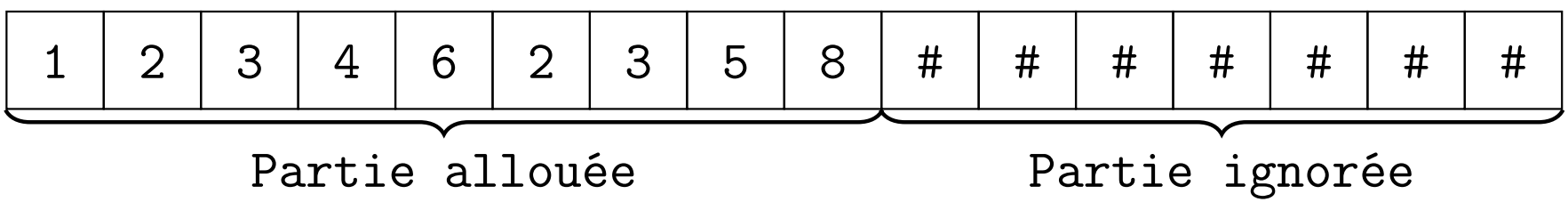
Tableau dynamique (3)

Lorsqu'on veut ajouter un élément alors que la mémoire est pleine, on a besoin de procéder à un redimensionnement :





On peut ensuite rajouter l'élément, 8 par exemple et on obtient :



Interface d'un tableau dynamique

Il nous faut un type qui corresponde au tableau dynamique. Ce type est paramétrique

```
1 type 'a dynamique
```

Nous avons aussi besoin des fonctions suivantes :

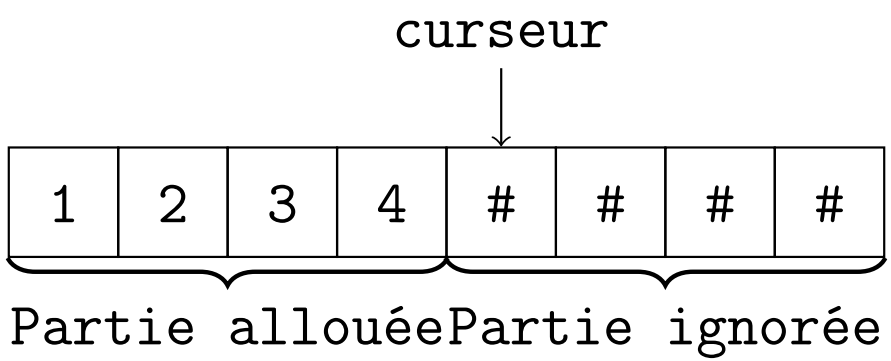
- creer de signature unit -> 'a dynamique qui crée un tableau dynamique vide ;
- ajouter de signature 'a dynamique -> 'a -> unit qui ajoute un élément à la fin d'un tableau dynamique ;
- retirer de signature 'a dynamique -> 'a qui retire et renvoie le dernier élément du tableau ;
- acceder de signature 'a dynamique -> int -> 'a qui renvoie la valeur d'un tableau dynamique à un indice donné ;
- modifier de signature 'a dynamique -> int -> 'a -> unit qui change la valeur d'un tableau dynamique à un indice donné.

Implémentation d'un tableau dynamique

Nous proposons le type suivant :

```
type 'a dynamique = {mutable memoire : 'a array ;
    mutable curseur : int}
```

L'idée est d'avoir un curseur qui indique l'indice du prochain élément que l'on peut modifier (ou la taille de la mémoire si la mémoire est pleine).



Ces deux champs sont mutables de sorte à pouvoir modifier le tableau dynamique directement.

Creer

Comment implémenter la fonction de création de signature
unit -> 'a dynamique ?

Creer

Comment implémenter la fonction de création de signature unit -> 'a dynamique ?

```
let creer () = {memoire = [||] ; curseur = 0}
```

On prend soin de créer un tableau totalement vide de sorte à pouvoir avoir la bonne signature.

Accès et modification

Comment implémenter la fonction d'accès de signature
'a dynamique -> int -> 'a ?

Accès et modification

Comment implémenter la fonction d'accès de signature 'a dynamique -> int -> 'a ?

```
let acceder td k =
    if k>= td.curseur
    then failwith "Indice non alloue."
    else td.memoire.(k)
```

Accès et modification

Comment implémenter la fonction d'accès de signature
'a dynamique -> int -> 'a ?

```
let acceder td k =
    if k>= td.curseur
    then failwith "Indice non alloue."
    else td.memoire.(k)
```

Comment implémenter la fonction de modification de signature
'a dynamique -> int -> 'a -> unit ?

Accès et modification

Comment implémenter la fonction d'accès de signature 'a dynamique -> int -> 'a ?

```
let acceder td k =
    if k>= td.curseur
    then failwith "Indice non alloue."
    else td.memoire.(k)
```

Comment implémenter la fonction de modification de signature 'a dynamique -> int -> 'a -> unit ?

```
let modifier td k a =
    if k>= td.curseur then failwith "Indice non
    alloue."
    else td.memoire.(k) <- a
```

Retrait

Comment implément le retrait de signature `'a dynamique -> 'a` ?

Retrait

Comment implément le retrait de signature 'a dynamique -> 'a ?

```
let retirer td =
    if td.curseur = 0 then failwith "Tableau vide"
    else
        begin
            td.curseur <- td.curseur - 1 ;
            td.memoire.(td.curseur + 1)
        end
```

Ajout

Comment implémenter l'ajout de signature

'a dynamique -> 'a -> unit ?

Ajout

Comment implémenter l'ajout de signature 'a dynamique -> 'a -> unit ?

```
let ajouter td a =
    td.memoire(td.curseur) <- a ;
    td.curseur <- td.curseur + 1
```

Ajout

Comment implémenter l'ajout de signature

'a dynamique -> 'a -> unit ?

```
let ajouter td a =
    td.memoire(td.curseur) <- a ;
    td.curseur <- td.curseur + 1
```

```
let ajouter td a =
    let n = Array.length td.memoire in
    if td.curseur = n then
    begin
        let taille = max (2 * n) 1 in
        let nouvelle_memoire = Array.make taille a
    in
        for i = 0 to n-1 do
            nouvelle_memoire.(i) <- td.memoire.(i)
        done ;
        td.memoire <- nouvelle_memoire
    end ;
    td.memoire(td.curseur) <- a ;
    td.curseur <- td.curseur + 1
```

Complexité de l'ajout

Quelle est la complexité de l'ajout dans un tableau dans le pire des cas en fonction du nombre n d'éléments dans le tableau dynamique ?

Complexité de l'ajout

Quelle est la complexité de l'ajout dans un tableau dans le pire des cas en fonction du nombre n d'éléments dans le tableau dynamique ?
Le pire des cas correspond au cas où on recopie : on a besoin de faire une allocation d'un nouveau tableau (potentiellement en $O(n)$), et on doit recopier le tableau (en $O(n)$).
Ainsi, la complexité est en $O(n)$.

Complexité de l'ajout

Quelle est la complexité de l'ajout dans un tableau dans le pire des cas en fonction du nombre n d'éléments dans le tableau dynamique ?
Le pire des cas correspond au cas où on recopie : on a besoin de faire une allocation d'un nouveau tableau (potentiellement en $O(n)$), et on doit recopier le tableau (en $O(n)$).
Ainsi, la complexité est en $O(n)$.
Cependant, les redimensionnement sont rares. Comment pourrait-on quantifier cela dans le calcul de la complexité ?

Analyse amortie (1)

Définition 1 : Analyse Amortie

L' **analyse de complexité amortie** est une méthode d'analyse de la complexité d'une suite d'opération qui associe à chaque opération la moyenne de ces opérations.

Attention : l'analyse amortie n'est pas un calcul probabiliste, mais une moyenne arithmétique dans le pire des cas d'une suite d'opération.

Analyse amortie (2)

Il existe plusieurs manière de procéder, mais on choisit ici la méthode *par agrégation*. On calcule le coût d'un certains nombre d'opération dont on réalise la moyenne.
Lors de 2^k ajouts à partir d'un tableau dynamique vide avec une case mémoire, on doit réaliser $k+1$ reaffectations qui consomment $\sum_{i=0}^{k+1} 2^i = 2^{k+2} - 1$ en complexité.
Ainsi, le coût total pour 2^k ajouts est le coût des 2^k modifications du tableau dynamique, et le coût des k réaffectation, ce qui nous donne une complexité en $O(2^k)$.

Analyse amortie (3)

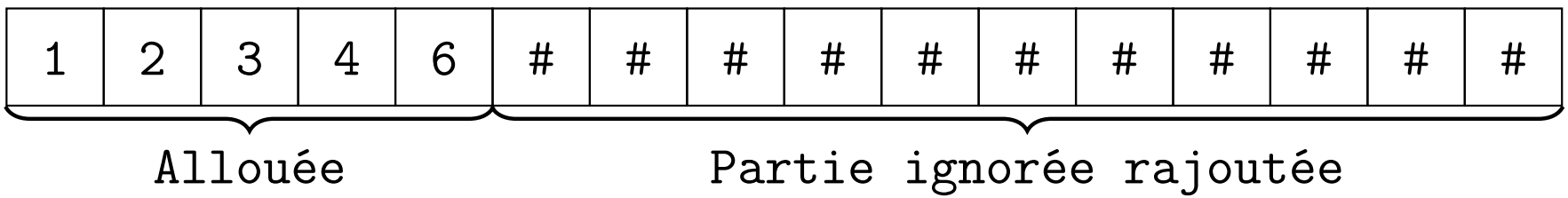
Ainsi, lors de l'ajout de n élément, on doit réaliser au plus $\lceil \log_2 n \rceil$ réaffectation, ce qui nous donne une complexité en $O(2^{\lceil \log_2 n \rceil})$, et donc en $O(n)$.
Le coût par opération est donc bien en $O(1)$.

Redimensionnement d'un tableau presque vide

On peut aussi décider de redimensionner le tableau dynamique quand il est presque vide : lorsque moins du quart des éléments de la mémoire sont occupés, on divise par deux la taille de la mémoire occupée par le tableau.

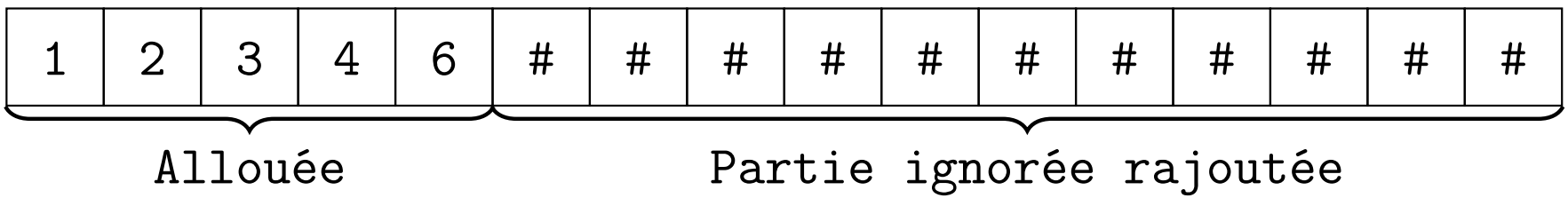
Redimensionnement d'un tableau presque vide

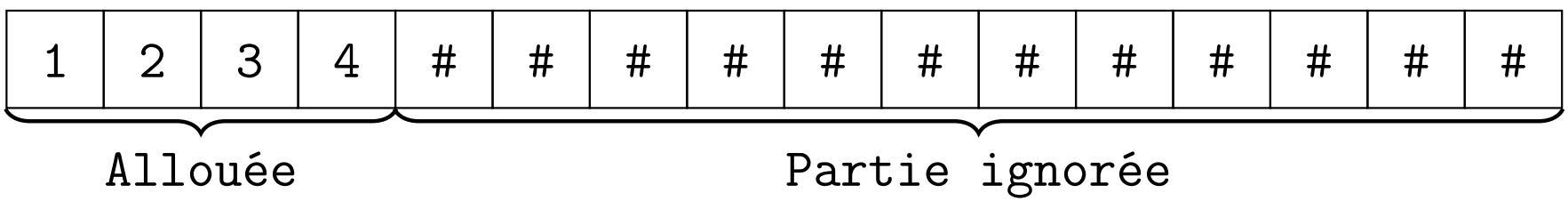
On peut aussi décider de redimensionner le tableau dynamique quand il est presque vide : lorsque moins du quart des éléments de la mémoire sont occupés, on divise par deux la taille de la mémoire occupée par le tableau.



Redimensionnement d'un tableau presque vide

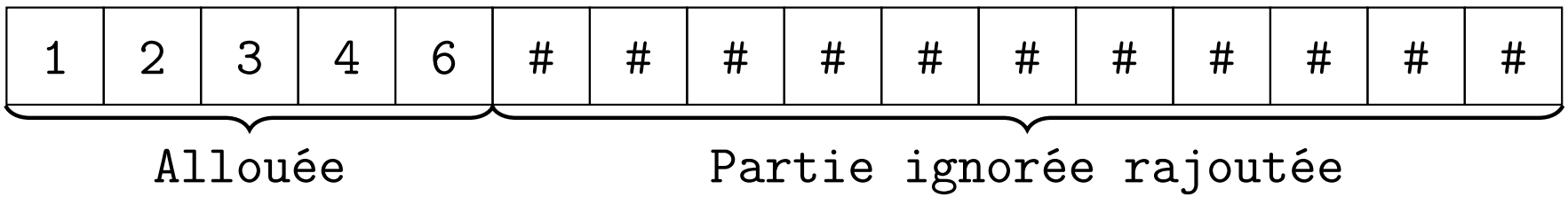
On peut aussi décider de redimensionner le tableau dynamique quand il est presque vide : lorsque moins du quart des éléments de la mémoire sont occupés, on divise par deux la taille de la mémoire occupée par le tableau.

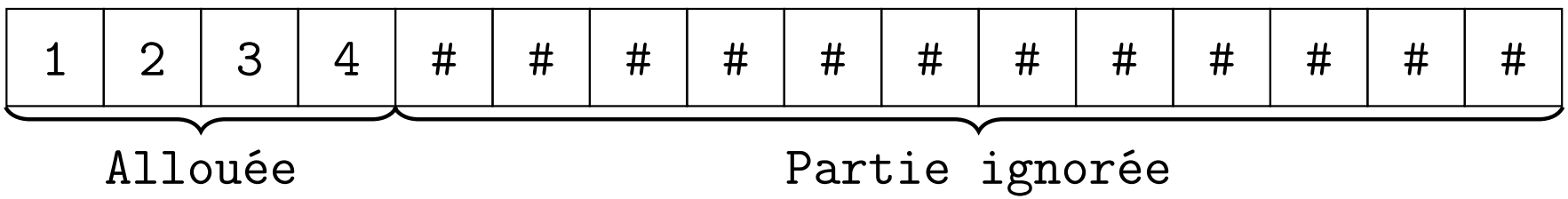


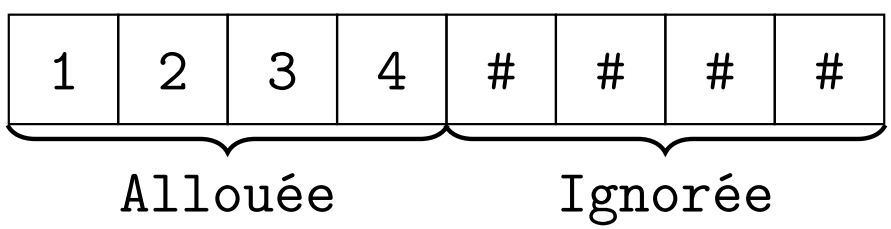


Redimensionnement d'un tableau presque vide

On peut aussi décider de redimensionner le tableau dynamique quand il est presque vide : lorsque moins du quart des éléments de la mémoire sont occupés, on divise par deux la taille de la mémoire occupée par le tableau.







Redimensionnement d'un tableau presque vide

Pourquoi ne pas faire le redimensionnement quand on occupe moins de la moitié du tableau ?

Redimensionnement d'un tableau presque vide

Pourquoi ne pas faire le redimensionnement quand on occupe moins de la moitié du tableau ?

On peut montrer que la complexité temporelle amortie pour une série d'ajouts et de retraits dans le tableau est toujours en $O(1)$ pour chaque opération.