

1. Exemple introductif

Nombre d'occurrence des mots d'un texte

A partir du texte de l'oeuvre de J. Verne, « *20 000 lieux sous les mers* », on a construit la liste des mots (sans accent, ni majuscules) qui apparaissent dans cet oeuvre. A titre d'exemple voici un extrait du fichier obtenu

```
1      cetace
2      extraordinaire
3      pouvait
4      se
5      transporter
6      un
7      endroit
8      un
9      autre
```

C10 Tableaux associatifs, hachage

1. Exemple introductif

Nombre d'occurrence des mots d'un texte

A partir du texte de l'oeuvre de J. Verne, « *20 000 lieux sous les mers* », on a construit la liste des mots (sans accent, ni majuscules) qui apparaissent dans cet oeuvre. A titre d'exemple voici un extrait du fichier obtenu

```
1      cetace
2      extraordinaire
3      pouvait
4      se
5      transporter
6      un
7      endroit
8      un
9      autre
```

Le but du problème est de trouver une méthode **efficace** afin d'obtenir le nombre d'occurrence de chaque mot

Définition

Un **tableau associatif** (ou **dictionnaire**) est une structure de données constituée d'un ensemble de **clés** C et d'un ensemble de **valeurs** V . Chaque clé n'apparaît qu'une fois dans la structure et est associée à un élément de V .

Les dictionnaires étendent en quelque sorte la notion de tableau, les indices des éléments d'un tableau associatif de taille n n'étant plus nécessairement les entiers $\llbracket 0; n - 1 \rrbracket$ comme pour les tableaux classiques.

C10 Tableaux associatifs, hachage

2. Définition

Définition

Un **tableau associatif** (ou **dictionnaire**) est une structure de données constituée d'un ensemble de **clés** C et d'un ensemble de **valeurs** V . Chaque clé n'apparaît qu'une fois dans la structure et est associée à un élément de V .

Les dictionnaires étendent en quelque sorte la notion de tableau, les indices des éléments d'un tableau associatif de taille n n'étant plus nécessairement les entiers $\llbracket 0; n - 1 \rrbracket$ comme pour les tableaux classiques.

Exemple

On peut associer à chaque mot d'un texte, son nombre d'occurrence dans ce texte :

C10 Tableaux associatifs, hachage

2. Définition

Définition

Un **tableau associatif** (ou **dictionnaire**) est une structure de données constituée d'un ensemble de **clés** C et d'un ensemble de **valeurs** V . Chaque clé n'apparaît qu'une fois dans la structure et est associée à un élément de V .

Les dictionnaires étendent en quelque sorte la notion de tableau, les indices des éléments d'un tableau associatif de taille n n'étant plus nécessairement les entiers $\llbracket 0; n - 1 \rrbracket$ comme pour les tableaux classiques.

Exemple

On peut associer à chaque mot d'un texte, son nombre d'occurrence dans ce texte :

- L'ensemble des clés est l'ensemble des mots du texte.

Définition

Un **tableau associatif** (ou **dictionnaire**) est une structure de données constituée d'un ensemble de **clés** C et d'un ensemble de **valeurs** V . Chaque clé n'apparaît qu'une fois dans la structure et est associée à un élément de V .

Les dictionnaires étendent en quelque sorte la notion de tableau, les indices des éléments d'un tableau associatif de taille n n'étant plus nécessairement les entiers $\llbracket 0; n - 1 \rrbracket$ comme pour les tableaux classiques.

Exemple

On peut associer à chaque mot d'un texte, son nombre d'occurrence dans ce texte :

- L'ensemble des clés est l'ensemble des mots du texte.
- L'ensemble des valeurs est \mathbb{N} .

C10 Tableaux associatifs, hachage

2. Définition

Définition

Un **tableau associatif** (ou **dictionnaire**) est une structure de données constituée d'un ensemble de **clés** C et d'un ensemble de **valeurs** V . Chaque clé n'apparaît qu'une fois dans la structure et est associée à un élément de V .

Les dictionnaires étendent en quelque sorte la notion de tableau, les indices des éléments d'un tableau associatif de taille n n'étant plus nécessairement les entiers $\llbracket 0; n - 1 \rrbracket$ comme pour les tableaux classiques.

Exemple

On peut associer à chaque mot d'un texte, son nombre d'occurrence dans ce texte :

- L'ensemble des clés est l'ensemble des mots du texte.
- L'ensemble des valeurs est \mathbb{N} .

Ce tableau associatif peut se noter :

$\{(\text{"un"}, 10), (\text{"cours"}, 3), (\text{"exercice"}, 7) \dots\}$

C10 Tableaux associatifs, hachage

2. Définition

Interface d'un tableau associatif

En notant T un tableau associatif d'ensemble de clé C et de valeur V , $c \in C$ et $v \in V$:

Interface d'un tableau associatif

En notant T un tableau associatif d'ensemble de clé C et de valeur V , $c \in C$ et $v \in V$:

- Tester si une clé c appartient à T

Interface d'un tableau associatif

En notant T un tableau associatif d'ensemble de clé C et de valeur V , $c \in C$ et $v \in V$:

- Tester si une clé c appartient à T
- Ajouter à T une association (c, v)

Interface d'un tableau associatif

En notant T un tableau associatif d'ensemble de clé C et de valeur V , $c \in C$ et $v \in V$:

- Tester si une clé c appartient à T
- Ajouter à T une association (c, v)
- Supprimer une association (c, v) de T

Interface d'un tableau associatif

En notant T un tableau associatif d'ensemble de clé C et de valeur V , $c \in C$ et $v \in V$:

- Tester si une clé c appartient à T
- Ajouter à T une association (c, v)
- Supprimer une association (c, v) de T
- Obtenir la valeur v associée à une clé c .

Interface d'un tableau associatif

En notant T un tableau associatif d'ensemble de clé C et de valeur V , $c \in C$ et $v \in V$:

- Tester si une clé c appartient à T
- Ajouter à T une association (c, v)
- Supprimer une association (c, v) de T
- Obtenir la valeur v associée à une clé c .
- Modifier la valeur associée à une clé.

Possibilités d'implémentation

- Une implémentation naïve à l'aide d'une liste chaînée de maillons contenant les couples (clé, valeur) est possible mais clairement inefficace

Possibilités d'implémentation

- Une implémentation naïve à l'aide d'une liste chaînée de maillons contenant les couples (clé, valeur) est possible mais clairement inefficace (le test d'appartenance est alors en $O(n)$)

Possibilités d'implémentation

- Une implémentation naïve à l'aide d'une liste chaînée de maillons contenant les couples (clé, valeur) est possible mais clairement inefficace (le test d'appartenance est alors en $O(n)$)
- Une implémentation utilisant les arbres sera vue ultérieurement.

Possibilités d'implémentation

- Une implémentation naïve à l'aide d'une liste chaînée de maillons contenant les couples (clé, valeur) est possible mais clairement inefficace (le test d'appartenance est alors en $O(n)$)
- Une implémentation utilisant les arbres sera vue ultérieurement.
- Lorsque l'ensemble des clés est inclus dans $\llbracket 0; N - 1 \rrbracket$, on peut utiliser un tableau de taille N , pour un couple (c, v) , on stocke alors la valeur v dans la case d'indice c du tableau.

Possibilités d'implémentation

- Une implémentation naïve à l'aide d'une liste chaînée de maillons contenant les couples (clé, valeur) est possible mais clairement inefficace (le test d'appartenance est alors en $O(n)$)
- Une implémentation utilisant les arbres sera vue ultérieurement.
- Lorsque l'ensemble des clés est inclus dans $\llbracket 0; N - 1 \rrbracket$, on peut utiliser un tableau de taille N , pour un couple (c, v) , on stocke alors la valeur v dans la case d'indice c du tableau.
- Pour un ensemble de clés quelconque C , on se ramène au cas précédent en deux étapes :

Possibilités d'implémentation

- Une implémentation naïve à l'aide d'une liste chaînée de maillons contenant les couples (clé, valeur) est possible mais clairement inefficace (le test d'appartenance est alors en $O(n)$)
- Une implémentation utilisant les arbres sera vue ultérieurement.
- Lorsque l'ensemble des clés est inclus dans $\llbracket 0; N - 1 \rrbracket$, on peut utiliser un tableau de taille N , pour un couple (c, v) , on stocke alors la valeur v dans la case d'indice c du tableau.
- Pour un ensemble de clés quelconque C , on se ramène au cas précédent en deux étapes :
 - On utilise une fonction $h : C \rightarrow \mathbb{N}$, dite **fonction de hachage** (*hash function*).

Possibilités d'implémentation

- Une implémentation naïve à l'aide d'une liste chaînée de maillons contenant les couples (clé, valeur) est possible mais clairement inefficace (le test d'appartenance est alors en $O(n)$)
- Une implémentation utilisant les arbres sera vue ultérieurement.
- Lorsque l'ensemble des clés est inclus dans $\llbracket 0; N - 1 \rrbracket$, on peut utiliser un tableau de taille N , pour un couple (c, v) , on stocke alors la valeur v dans la case d'indice c du tableau.
- Pour un ensemble de clés quelconque C , on se ramène au cas précédent en deux étapes :
 - On utilise une fonction $h : C \rightarrow \mathbb{N}$, dite **fonction de hachage** (*hash function*).
 - L'indice de la clé c dans le tableau s'obtient alors en prenant $h(c) \bmod N$.

Possibilités d'implémentation

- Une implémentation naïve à l'aide d'une liste chaînée de maillons contenant les couples (clé, valeur) est possible mais clairement inefficace (le test d'appartenance est alors en $O(n)$)
 - Une implémentation utilisant les arbres sera vue ultérieurement.
 - Lorsque l'ensemble des clés est inclus dans $\llbracket 0; N - 1 \rrbracket$, on peut utiliser un tableau de taille N , pour un couple (c, v) , on stocke alors la valeur v dans la case d'indice c du tableau.
 - Pour un ensemble de clés quelconque C , on se ramène au cas précédent en deux étapes :
 - On utilise une fonction $h : C \rightarrow \mathbb{N}$, dite **fonction de hachage** (*hash function*).
 - L'indice de la clé c dans le tableau s'obtient alors en prenant $h(c) \bmod N$.
- ⚠ Deux clés différentes peuvent alors donner le *même* indice dans le tableau, on parle alors de **collision**.

C10 Tableaux associatifs, hachage

3. Techniques d'implémentation

Visualisation

```
{ "dans":12 , "le" : 8, "un" : 16, "bol" : 10}
```

Clés

Indice

Contenu

"dans"

0

1

2

⋮

42

⋮

$N-1$

"le"

"un"

C10 Tableaux associatifs, hachage

3. Techniques d'implémentation

Visualisation

{ "dans":12 , "le" : 8, "un" : 16, "bol" : 10}

Clés

"dans"

"le"

"un"

hash + mod

Indice

0
1
2
⋮
42
⋮
N-1

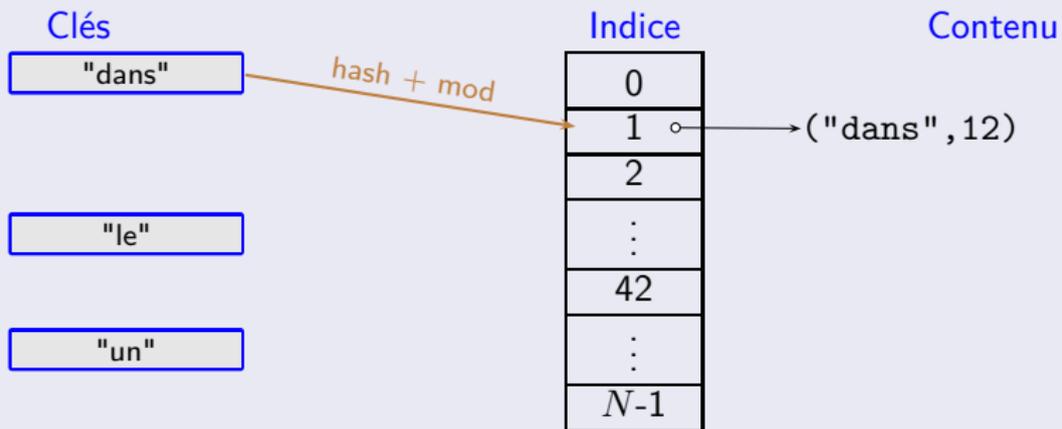
Contenu

C10 Tableaux associatifs, hachage

3. Techniques d'implémentation

Visualisation

{ "dans":12 , "le" : 8, "un" : 16, "bol" : 10}

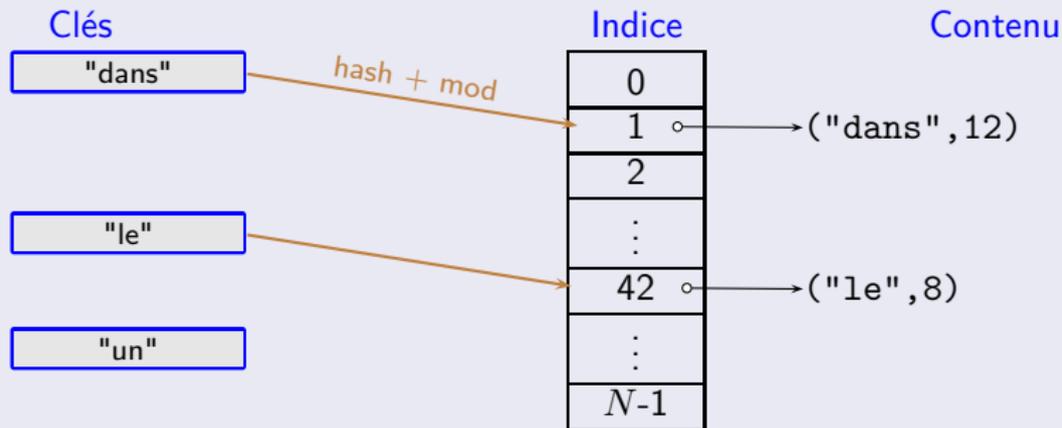


C10 Tableaux associatifs, hachage

3. Techniques d'implémentation

Visualisation

{ "dans":12 , "le" : 8, "un" : 16, "bol" : 10}

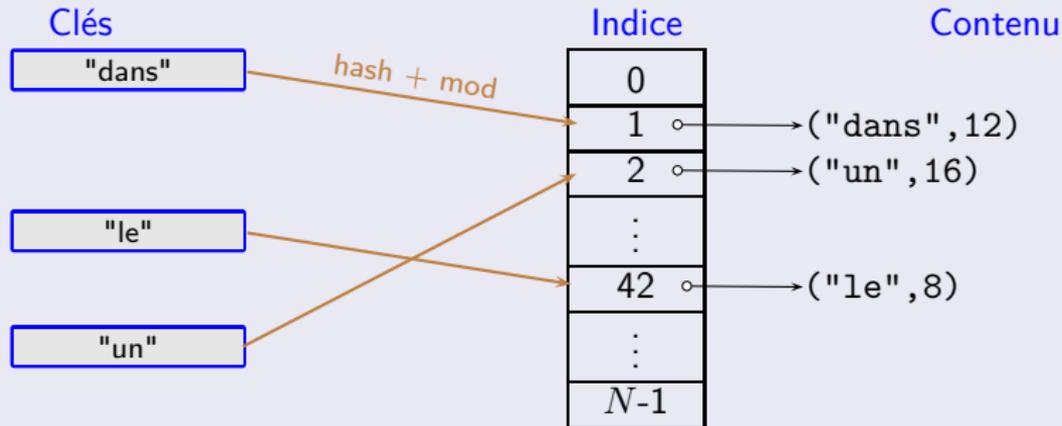


C10 Tableaux associatifs, hachage

3. Techniques d'implémentation

Visualisation

{ "dans":12 , "le" : 8, "un" : 16, "bol" : 10}

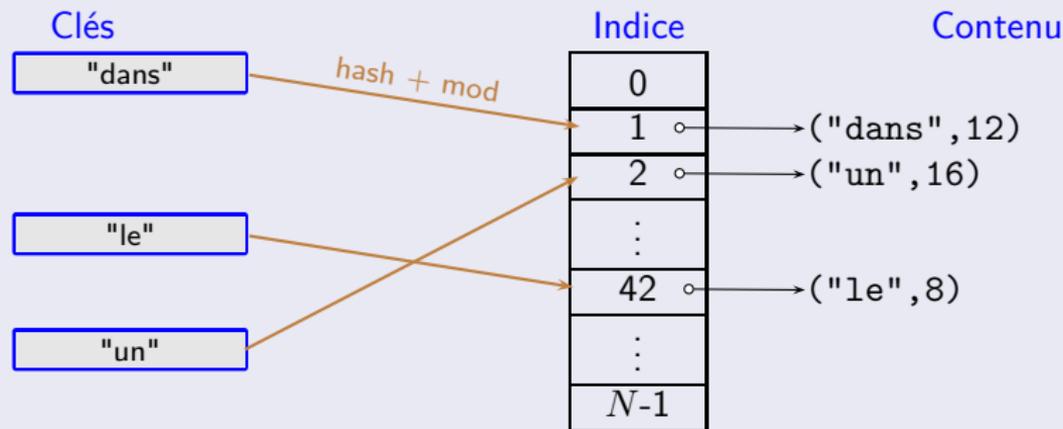


C10 Tableaux associatifs, hachage

3. Techniques d'implémentation

Visualisation

{ "dans":12 , "le" : 8, "un" : 16, "bol" : 10}



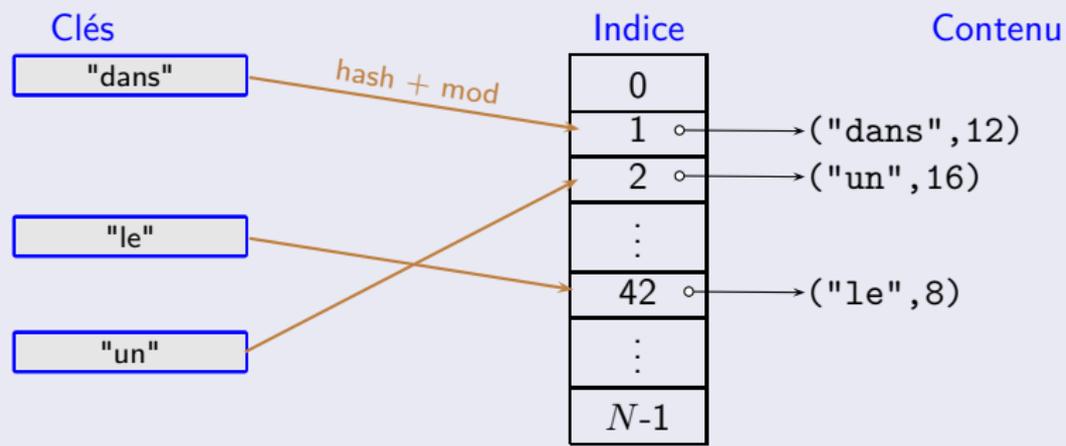
Certaines cases du tableau peuvent restées vides! Le *taux de charge* de la table est défini comme le rapport entre le nombres de cases occupés et N .

C10 Tableaux associatifs, hachage

3. Techniques d'implémentation

Collision

{ "dans":12 , "le" : 8, "un" : 16, "bol" : 10}

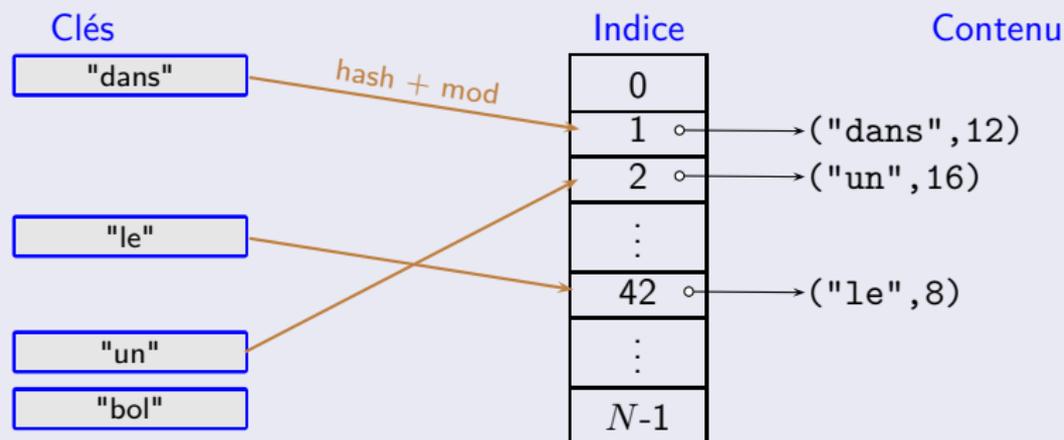


C10 Tableaux associatifs, hachage

3. Techniques d'implémentation

Collision

{ "dans":12 , "le" : 8, "un" : 16, "bol" : 10}

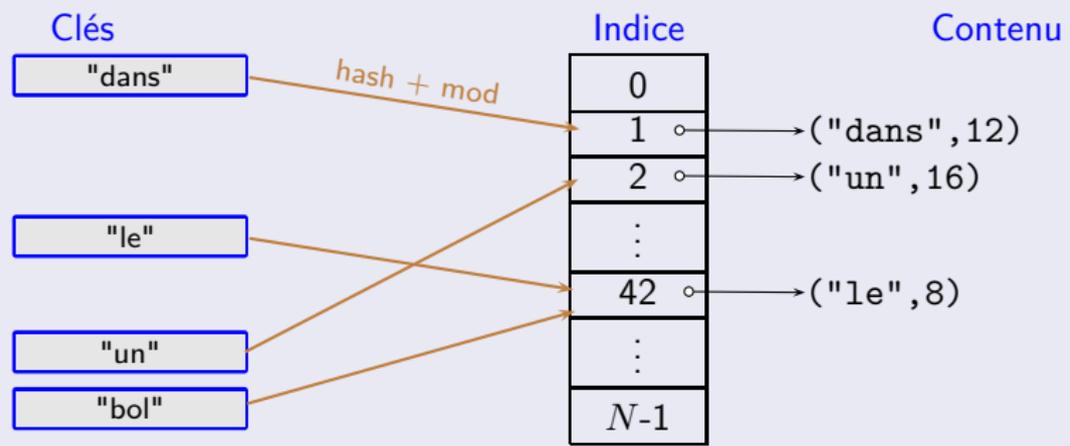


C10 Tableaux associatifs, hachage

3. Techniques d'implémentation

Collision

{ "dans":12 , "le" : 8, "un" : 16, "bol" : 10}

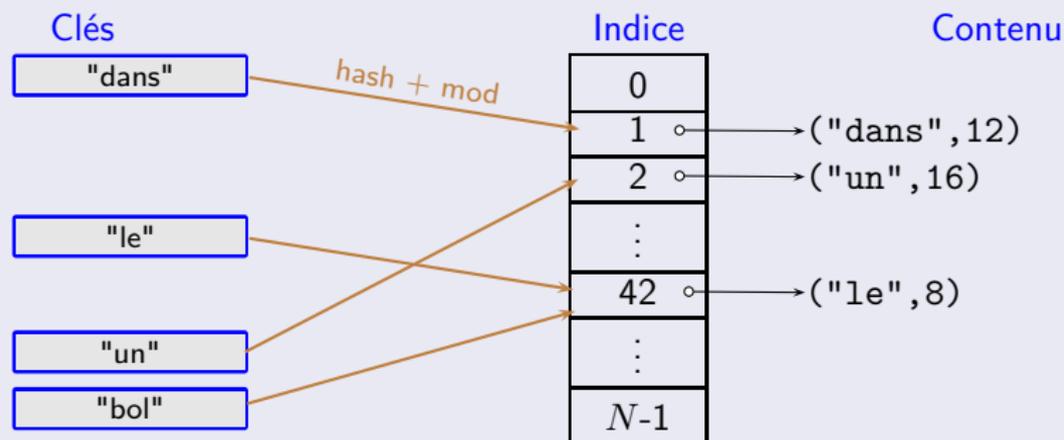


C10 Tableaux associatifs, hachage

3. Techniques d'implémentation

Collision

{ "dans":12 , "le" : 8, "un" : 16, "bol" : 10}



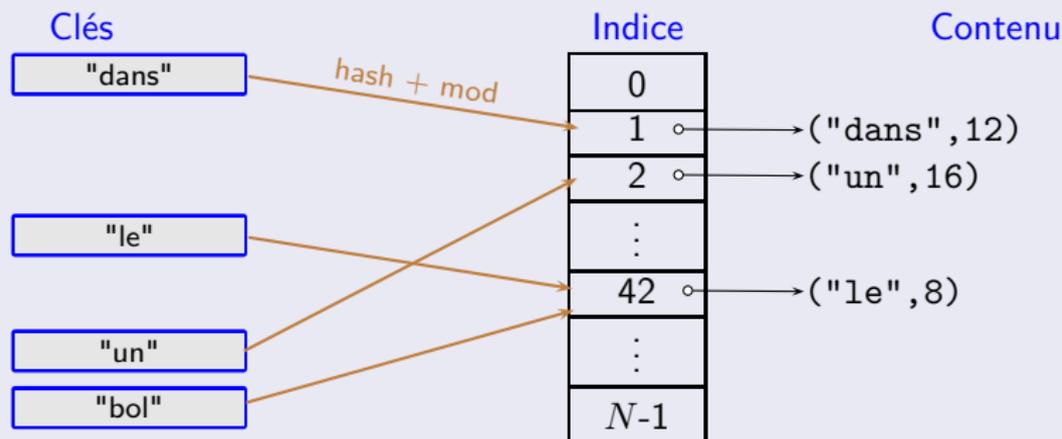
- Deux clés différentes ("bol" et "le"), doivent être rangées au même indice dans un tableau, c'est une **collision**.

C10 Tableaux associatifs, hachage

3. Techniques d'implémentation

Collision

{ "dans":12 , "le" : 8, "un" : 16, "bol" : 10}



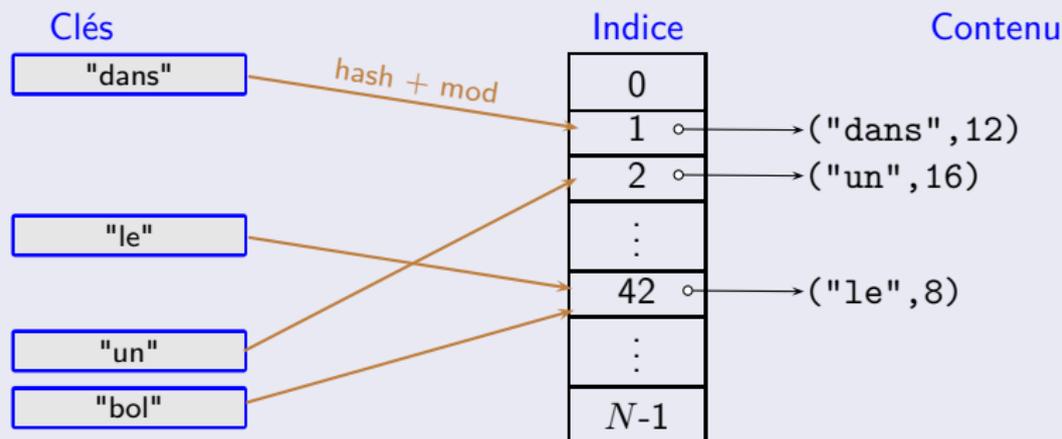
- Deux clés différentes ("bol" et "le"), doivent être rangées au même indice dans un tableau, c'est une **collision**.
- La résolution par chaînage consiste à stocker des listes chaînées, dans le tableau. On dit alors que chaque case du tableau est un **seau** (ou **bucket** en anglais) .

C10 Tableaux associatifs, hachage

3. Techniques d'implémentation

Collision

{ "dans":12 , "le" : 8, "un" : 16, "bol" : 10}



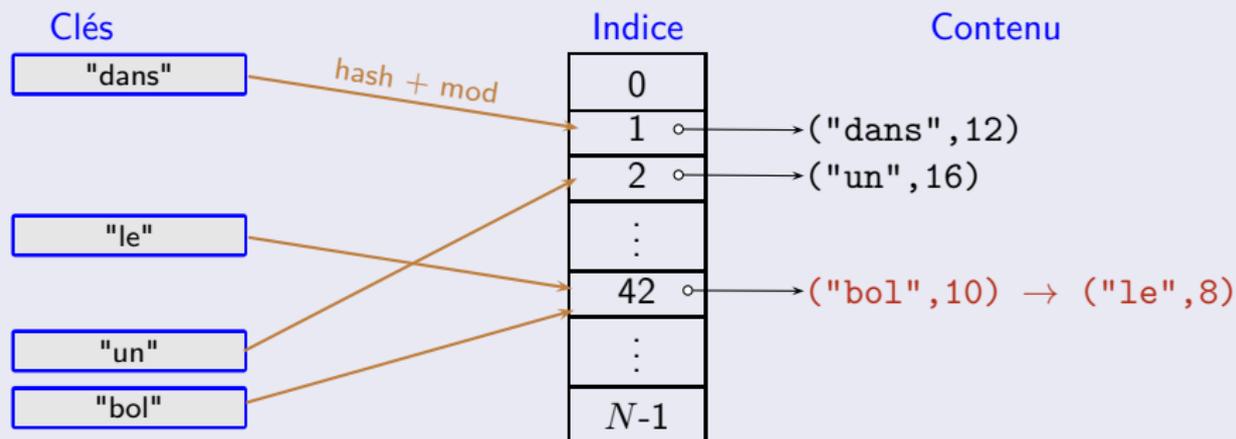
- Deux clés différentes ("bol" et "le"), doivent être rangées au même indice dans un tableau, c'est une **collision**.
- La résolution par chaînage consiste à stocker des listes chaînées, dans le tableau. On dit alors que chaque case du tableau est un **seau** (ou **bucket** en anglais) .

C10 Tableaux associatifs, hachage

3. Techniques d'implémentation

Collision

{ "dans":12 , "le" : 8, "un" : 16, "bol" : 10}



- Deux clés différentes ("bol" et "le"), doivent être rangées au même indice dans un tableau, c'est une **collision**.
- La résolution par chaînage consiste à stocker des listes chaînées, dans le tableau. On dit alors que chaque case du tableau est un **seau** (ou **bucket** en anglais) .

Complexité des opérations

- En faisant l'hypothèse que le calcul de la fonction de hachage s'effectue en temps constant, le coût des opérations (appartenance, ajout, suppression, ...) est proportionnel à la longueur des seaux.

Complexité des opérations

- En faisant l'hypothèse que le calcul de la fonction de hachage s'effectue en temps constant, le coût des opérations (appartenance, ajout, suppression, ...) est proportionnel à la longueur des seaux.
- Cette longueur dépend de la fonction de hachage et de la longueur de la table de hachage N , et est difficile à majorer en pratique.

Complexité des opérations

- En faisant l'hypothèse que le calcul de la fonction de hachage s'effectue en temps constant, le coût des opérations (appartenance, ajout, suppression, ...) est proportionnel à la longueur des seaux.
- Cette longueur dépend de la fonction de hachage et de la longueur de la table de hachage N , et est difficile à majorer en pratique.
- Une évaluation expérimentale sera vue en TP.

Complexité des opérations

- En faisant l'hypothèse que le calcul de la fonction de hachage s'effectue en temps constant, le coût des opérations (appartenance, ajout, suppression, ...) est proportionnel à la longueur des seaux.
- Cette longueur dépend de la fonction de hachage et de la longueur de la table de hachage N , et est difficile à majorer en pratique.
- Une évaluation expérimentale sera vue en TP.
- Si la taille de la table de hachage est de l'ordre du nombre de clés, et que la fonction de hachage permet de répartir uniformément les clés alors en moyenne, la complexité des opérations (appartenance, ajout, suppression, ...) est en $O(1)$.

Complexité des opérations

- En faisant l'hypothèse que le calcul de la fonction de hachage s'effectue en temps constant, le coût des opérations (appartenance, ajout, suppression, ...) est proportionnel à la longueur des seaux.
- Cette longueur dépend de la fonction de hachage et de la longueur de la table de hachage N , et est difficile à majorer en pratique.
- Une évaluation expérimentale sera vue en TP.
- Si la taille de la table de hachage est de l'ordre du nombre de clés, et que la fonction de hachage permet de répartir uniformément les clés alors en moyenne, la complexité des opérations (appartenance, ajout, suppression, ...) est en $O(1)$.
- On peut éventuellement utiliser un tableau dynamique de façon à augmenter la taille de la table de hachage lorsque son taux de charge devient trop important.

Les listes chaînées de clés/valeurs

- On commence par créer des listes chaînées de clés/valeurs. Les clés sont des chaînes de caractères, et les valeurs des entiers positifs. Pour simplifier, on suppose qu'un mot a au maximum 26 lettres :

Les listes chaînées de clés/valeurs

- On commence par créer des listes chaînées de clés/valeurs. Les clés sont des chaînes de caractères, et les valeurs des entiers positifs. Pour simplifier, on suppose qu'un mot a au maximum 26 lettres :

```
1  struct node
2  {
3      char word[26];
4      int occ;
5      struct node *next;
6  };
7  typedef struct node node;
8  typedef node* list;
```

Les listes chaînées de clés/valeurs

- On commence par créer des listes chaînées de clés/valeurs. Les clés sont des chaînes de caractères, et les valeurs des entiers positifs. Pour simplifier, on suppose qu'un mot a au maximum 26 lettres :

```
1  struct node
2  {
3      char word[26];
4      int occ;
5      struct node *next;
6  };
7  typedef struct node node;
8  typedef node* list;
```

- Les prototypes des fonctions nécessaires :

Les listes chaînées de clés/valeurs

- On commence par créer des listes chaînées de clés/valeurs. Les clés sont des chaînes de caractères, et les valeurs des entiers positifs. Pour simplifier, on suppose qu'un mot a au maximum 26 lettres :

```
1 struct node
2 {
3     char word[26];
4     int occ;
5     struct node *next;
6 };
7 typedef struct node node;
8 typedef node* list;
```

- Les prototypes des fonctions nécessaires :
 - Test si une clé est présente : `bool is_in(list l, char w[26])`

Les listes chaînées de clés/valeurs

- On commence par créer des listes chaînées de clés/valeurs. Les clés sont des chaînes de caractères, et les valeurs des entiers positifs. Pour simplifier, on suppose qu'un mot a au maximum 26 lettres :

```
1  struct node
2  {
3      char word[26];
4      int occ;
5      struct node *next;
6  };
7  typedef struct node node;
8  typedef node* list;
```

- Les prototypes des fonctions nécessaires :
 - Test si une clé est présente : `bool is_in(list l, char w[26])`
 - Ajout d'une nouvelle clé : `void insert(list *l, char w[26])`

Les listes chaînées de clés/valeurs

- On commence par créer des listes chaînées de clés/valeurs. Les clés sont des chaînes de caractères, et les valeurs des entiers positifs. Pour simplifier, on suppose qu'un mot a au maximum 26 lettres :

```
1  struct node
2  {
3      char word[26];
4      int occ;
5      struct node *next;
6  };
7  typedef struct node node;
8  typedef node* list;
```

- Les prototypes des fonctions nécessaires :
 - Test si une clé est présente : `bool is_in(list l, char w[26])`
 - Ajout d'une nouvelle clé : `void insert(list *l, char w[26])`
 - Récupérer la valeur associée à une clé : `int value(list l, char w[26])`

Les listes chaînées de clés/valeurs

- On commence par créer des listes chaînées de clés/valeurs. Les clés sont des chaînes de caractères, et les valeurs des entiers positifs. Pour simplifier, on suppose qu'un mot a au maximum 26 lettres :

```
1 struct node
2 {
3     char word[26];
4     int occ;
5     struct node *next;
6 };
7 typedef struct node node;
8 typedef node* list;
```

- Les prototypes des fonctions nécessaires :
 - Test si une clé est présente : `bool is_in(list l, char w[26])`
 - Ajout d'une nouvelle clé : `void insert(list *l, char w[26])`
 - Récupérer la valeur associée à une clé : `int value(list l, char w[26])`
 - Modification d'une valeur : `void update(list *l, char w[26], int v)`

La table de hachage

On définit alors la table de hachage comme un tableau de `SIZE` alvéoles contenant chacune une liste chaînée (la constante `SIZE` pouvant être définie en début de programme) les prototypes des fonctions à écrire sont :

- `bool is_in_hashtable(list ht[SIZE], char w[26])`

La table de hachage

On définit alors la table de hachage comme un tableau de `SIZE` alvéoles contenant chacune une liste chaînée (la constante `SIZE` pouvant être définie en début de programme) les prototypes des fonctions à écrire sont :

- `bool is_in_hashtable(list ht[SIZE], char w[26])`
- `void insert_in_hashtable(list ht[SIZE], char w[26])`

La table de hachage

On définit alors la table de hachage comme un tableau de `SIZE` alvéoles contenant chacune une liste chaînée (la constante `SIZE` pouvant être définie en début de programme) les prototypes des fonctions à écrire sont :

- `bool is_in_hashtable(list ht[SIZE], char w[26])`
- `void insert_in_hashtable(list ht[SIZE], char w[26])`
- `void update_hashtable(list ht[SIZE], char w[26], int n)`

La table de hachage

On définit alors la table de hachage comme un tableau de `SIZE` alvéoles contenant chacune une liste chaînée (la constante `SIZE` pouvant être définie en début de programme) les prototypes des fonctions à écrire sont :

- `bool is_in_hashtable(list ht[SIZE], char w[26])`
- `void insert_in_hashtable(list ht[SIZE], char w[26])`
- `void update_hashtable(list ht[SIZE], char w[26], int n)`
- `int get_val_hashtable(list ht[SIZE], char w[26])`

La table de hachage

On définit alors la table de hachage comme un tableau de `SIZE` alvéoles contenant chacune une liste chaînée (la constante `SIZE` pouvant être définie en début de programme) les prototypes des fonctions à écrire sont :

- `bool is_in_hashtable(list ht[SIZE], char w[26])`
- `void insert_in_hashtable(list ht[SIZE], char w[26])`
- `void update_hashtable(list ht[SIZE], char w[26], int n)`
- `int get_val_hashtable(list ht[SIZE], char w[26])`

Cette implémentation sera vue en détail en TP.

C10 Tableaux associatifs, hachage

5. Implémentation en OCaml

Implémentation avec le type `array`

On peut créer un type représentant un tableau de liste. Les éléments des listes étant des couples (c,v) où c est de type `string` et v de type `int` :

```
1 type hashtable = (string * int) list array
```

C10 Tableaux associatifs, hachage

5. Implémentation en OCaml

Implémentation avec le type `array`

On peut créer un type représentant un tableau de liste. Les éléments des listes étant des couples (c,v) où c est de type `string` et v de type `int` :

```
1 type hashtable = (string * int) list array
```

Sur les listes, les opérations nécessaires sont les mêmes que celles vu sur l'implémentation en C :

C10 Tableaux associatifs, hachage

5. Implémentation en OCaml

Implémentation avec le type `array`

On peut créer un type représentant un tableau de liste. Les éléments des listes étant des couples (c,v) où c est de type `string` et v de type `int` :

```
1 type hashtable = (string * int) list array
```

Sur les listes, les opérations nécessaires sont les mêmes que celles vu sur l'implémentation en C :

- Le test d'appartenance :

```
1 let rec is_in cle l =  
2   match l with  
3   | [] -> false  
4   |(s,v)::t -> s=cle || (is_in cle t);;
```

C10 Tableaux associatifs, hachage

5. Implémentation en OCaml

Implémentation avec le type `array`

On peut créer un type représentant un tableau de liste. Les éléments des listes étant des couples (c,v) où c est de type `string` et v de type `int` :

```
1 type hashtable = (string * int) list array
```

Sur les listes, les opérations nécessaires sont les mêmes que celles vu sur l'implémentation en C :

- Le test d'appartenance :

```
1 let rec is_in cle l =  
2   match l with  
3   | [] -> false  
4   |(s,v)::t -> s=cle || (is_in cle t);;
```

- La mise à jour de la valeur associée à une clé (on utilise `failwith` en cas d'absence de la clé).

C10 Tableaux associatifs, hachage

5. Implémentation en OCaml

Implémentation avec le type `array`

On peut créer un type représentant un tableau de liste. Les éléments des listes étant des couples (c,v) où c est de type `string` et v de type `int` :

```
type hashtable = (string * int) list array
```

Sur les listes, les opérations nécessaires sont les mêmes que celles vu sur l'implémentation en C :

- Le test d'appartenance :

```
1 let rec is_in cle l =  
2   match l with  
3   | [] -> false  
4   |(s,v)::t -> s=cle || (is_in cle t);;
```

- La mise à jour de la valeur associée à une clé (on utilise `failwith` en cas d'absence de la clé).
- La récupération de la valeur associée à une clé.

C10 Tableaux associatifs, hachage

5. Implémentation en OCaml

Implémentation avec le type `array`

- On se fixe une taille pour la table de hachage se définit comme un tableau de liste de couples `string*int` :

```
1 let ht = Array.make size []
2 let reader = open_in "mots.txt"
```

C10 Tableaux associatifs, hachage

5. Implémentation en OCaml

Implémentation avec le type `array`

- On se fixe une taille pour la table de hachage se définit comme un tableau de liste de couples `string*int` :

```
1 let ht = Array.make size []  
2 let reader = open_in "mots.txt"
```

- La fonction de hachage `string -> unit` transforme une chaîne de caractère en un entier compris entre 0 (inclus) et `size` (exclus)

C10 Tableaux associatifs, hachage

5. Implémentation en OCaml

Implémentation avec le type `array`

- On se fixe une taille pour la table de hachage se définit comme un tableau de liste de couples `string*int` :

```
1 let ht = Array.make size []  
2 let reader = open_in "mots.txt"
```

- La fonction de hachage `string -> unit` transforme une chaîne de caractère en un entier compris entre 0 (inclus) et `size` (exclus)
- On doit alors écrire les fonctions pour :

C10 Tableaux associatifs, hachage

5. Implémentation en OCaml

Implémentation avec le type `array`

- On se fixe une taille pour la table de hachage se définit comme un tableau de liste de couples `string*int` :

```
1 let ht = Array.make size []  
2 let reader = open_in "mots.txt"
```

- La fonction de hachage `string -> unit` transforme une chaîne de caractère en un entier compris entre 0 (inclus) et `size` (exclus)
- On doit alors écrire les fonctions pour :
 - test si la clé `c` est présente (`-> bool`) :
`let is_in_ht (ht:hashtable) c`

Implémentation avec le type `array`

- On se fixe une taille pour la table de hachage se définit comme un tableau de liste de couples `string*int` :

```
1 let ht = Array.make size []
2 let reader = open_in "mots.txt"
```

- La fonction de hachage `string -> unit` transforme une chaîne de caractère en un entier compris entre 0 (inclus) et `size` (exclus)
- On doit alors écrire les fonctions pour :
 - test si la clé `c` est présente (`-> bool`) :
`let is_in_ht (ht:hashtable) c`
 - ajout de `(c,v)` dans la table (`-> unit`) :
`let add_ht (ht:hashtable) c v`

C10 Tableaux associatifs, hachage

5. Implémentation en OCaml

Implémentation avec le type `array`

- On se fixe une taille pour la table de hachage se définit comme un tableau de liste de couples `string*int` :

```
1 let ht = Array.make size []  
2 let reader = open_in "mots.txt"
```

- La fonction de hachage `string -> unit` transforme une chaîne de caractère en un entier compris entre 0 (inclus) et `size` (exclus)
- On doit alors écrire les fonctions pour :
 - test si la clé `c` est présente (`-> bool`) :
`let is_in_ht (ht:hashtable) c`
 - ajout de (c,v) dans la table (`-> unit`) :
`let add_ht (ht:hashtable) c v`
 - renvoie la valeur associée à `c` (`-> int`) :
`let get_value_ht (ht:hashtable) c`

C10 Tableaux associatifs, hachage

5. Implémentation en OCaml

Implémentation avec le type `array`

- On se fixe une taille pour la table de hachage se définit comme un tableau de liste de couples `string*int` :

```
1 let ht = Array.make size []  
2 let reader = open_in "mots.txt"
```

- La fonction de hachage `string -> unit` transforme une chaîne de caractère en un entier compris entre 0 (inclus) et `size` (exclus)
- On doit alors écrire les fonctions pour :
 - test si la clé `c` est présente (`-> bool`) :
`let is_in_ht (ht:hashtable) c`
 - ajout de (c,v) dans la table (`-> unit`) :
`let add_ht (ht:hashtable) c v`
 - renvoie la valeur associée à `c` (`-> int`) :
`let get_value_ht (ht:hashtable) c`
 - met à jour la valeur associée à `c` (`-> unit`) :
`let update_ht (ht:hashtable) c v`

C10 Tableaux associatifs, hachage

5. Implémentation en OCaml

Module Hashtbl

La bibliothèque standard d'OCaml propose une implémentation des tables de hachage via le module `Hashtbl`.

- On inclus ce module avec :

```
1 open Hashtbl
```

Module Hashtbl

La bibliothèque standard d'OCaml propose une implémentation des tables de hachage via le module `Hashtbl`.

- On inclus ce module avec :

```
1 open Hashtbl
```

- La fonction de hachage (`Hashtbl.hash`) est prédéfinie par OCaml et s'applique à une valeur de n'importe quel type.

Module Hashtbl

La bibliothèque standard d'OCaml propose une implémentation des tables de hachage via le module `Hashtbl`.

- On inclus ce module avec :

```
1 open Hashtbl
```

- La fonction de hachage (`Hashtbl.hash`) est prédéfinie par OCaml et s'applique à une valeur de n'importe quel type.
- On doit donner une taille initiale lors de la création de la table de hachage :

```
1 let my_ht = Hashtbl.create 10000
```

A noter que lorsque le taux de remplissage de cette table devient trop important, elle est redimensionnée (la taille double), c'est donc un tableau dynamique.

Fonctions du module `Hashtbl`

Parmi les fonctions disponibles, on retrouve celles vues dans l'implémentation en langage C :

Fonctions du module Hashtbl

Parmi les fonctions disponibles, on retrouve celles vues dans l'implémentation en langage C :

- `Hashtbl.mem` pour le test d'appartenance.

Fonctions du module Hashtbl

Parmi les fonctions disponibles, on retrouve celles vues dans l'implémentation en langage C :

- `Hashtbl.mem` pour le test d'appartenance.
- `Hashtbl.add` pour ajouter une couple (clé, valeur).

Fonctions du module Hashtbl

Parmi les fonctions disponibles, on retrouve celles vues dans l'implémentation en langage C :

- `Hashtbl.mem` pour le test d'appartenance.
- `Hashtbl.add` pour ajouter une couple (clé, valeur).
- `Hashtbl.find` pour trouver la valeur associée à une clé.

Fonctions du module Hashtbl

Parmi les fonctions disponibles, on retrouve celles vues dans l'implémentation en langage C :

- `Hashtbl.mem` pour le test d'appartenance.
- `Hashtbl.add` pour ajouter une couple (clé, valeur).
- `Hashtbl.find` pour trouver la valeur associée à une clé.
- `Hashtbl.replace` pour modifier la valeur associée à une clé.

Calcul des coefficients du binôme

- 1 Rappeler la définition des coefficients binomiaux à l'aide de factoriel ainsi que la relation de récurrence liant les coefficients binomiaux

Calcul des coefficients du binôme

- 1 Rappeler la définition des coefficients binomiaux à l'aide de factoriel ainsi que la relation de récurrence liant les coefficients binomiaux
- 2 Ecrire une fonction en OCaml utilisant la définition à base de factoriels permettant de calculer $\binom{n}{k}$ (avec n et k deux entiers naturels $n \geq k$). Donner sa complexité.

Calcul des coefficients du binôme

- 1 Rappeler la définition des coefficients binomiaux à l'aide de factoriel ainsi que la relation de récurrence liant les coefficients binomiaux
- 2 Ecrire une fonction en OCaml utilisant la définition à base de factoriels permettant de calculer $\binom{n}{k}$ (avec n et k deux entiers naturels $n \geq k$). Donner sa complexité.
- 3 Proposer une version récursive utilisant la relation de récurrence vue à la question 1.

Calcul des coefficients du binôme

1. Rappeler la définition des coefficients binomiaux à l'aide de factoriel ainsi que la relation de récurrence liant les coefficients binomiaux
2. Ecrire une fonction en OCaml utilisant la définition à base de factoriels permettant de calculer $\binom{n}{k}$ (avec n et k deux entiers naturels $n \geq k$). Donner sa complexité.
3. Proposer une version récursive utilisant la relation de récurrence vue à la question 1.
4. On note $N(n, k)$ le nombre d'appels nécessaires pour calculer $\binom{n}{k}$. Donner $N(n, 0)$, $N(n, n)$ et une relation de récurrence liant $N(n, k)$, $N(n - 1, k)$ et $N(n - 1, k - 1)$.

Calcul des coefficients du binôme

- 1 Rappeler la définition des coefficients binomiaux à l'aide de factoriel ainsi que la relation de récurrence liant les coefficients binomiaux
- 2 Ecrire une fonction en OCaml utilisant la définition à base de factoriels permettant de calculer $\binom{n}{k}$ (avec n et k deux entiers naturels $n \geq k$). Donner sa complexité.
- 3 Proposer une version récursive utilisant la relation de récurrence vue à la question 1.
- 4 On note $N(n, k)$ le nombre d'appels nécessaires pour calculer $\binom{n}{k}$. Donner $N(n, 0)$, $N(n, n)$ et une relation de récurrence liant $N(n, k)$, $N(n - 1, k)$ et $N(n - 1, k - 1)$.
- 5 Prouver par récurrence que $N(n, k) = 2^{\binom{n}{k}} - 1$. En déduire la complexité de la fonction récursive écrite à la question 3.

Mémoïsation avec une table de hachage

- La complexité exponentielle de la version récursive est lié à l'apparition de nombreux appels récursifs identiques.

Mémoïsation avec une table de hachage

- La complexité exponentielle de la version récursive est lié à l'apparition de nombreux appels récursifs identiques.
- L'idée est donc de mémoriser les résultats des appels récursifs déjà effectués afin d'en avoir le résultat sans les relancer. Cette technique de programmation s'appelle la **mémoïsation**

Mémoïsation avec une table de hachage

- La complexité exponentielle de la version récursive est lié à l'apparition de nombreux appels récursifs identiques.
- L'idée est donc de mémoriser les résultats des appels récursifs déjà effectués afin d'en avoir le résultat sans les relancer. Cette technique de programmation s'appelle la **mémoïsation**
- Quelle structure de données vous paraît adaptée ?

Mémoïsation avec une table de hachage

- La complexité exponentielle de la version récursive est lié à l'apparition de nombreux appels récursifs identiques.
- L'idée est donc de mémoriser les résultats des appels récursifs déjà effectués afin d'en avoir le résultat sans les relancer. Cette technique de programmation s'appelle la **mémoïsation**
- Quelle structure de données vous paraît adaptée ?
- Proposer une implémentation en OCaml

Mémoïsation avec une table de hachage

- La complexité exponentielle de la version récursive est lié à l'apparition de nombreux appels récursifs identiques.
- L'idée est donc de mémoriser les résultats des appels récursifs déjà effectués afin d'en avoir le résultat sans les relancer. Cette technique de programmation s'appelle la **mémoïsation**
- Quelle structure de données vous paraît adaptée ?
- Proposer une implémentation en OCaml
- Donner la complexité de cette nouvelle implémentation.

Mémoïsation

- La **mémoïsation** consiste à stocker dans une structure de données les valeurs renvoyées par une fonction afin de ne pas les recalculer lors des appels identiques suivant.

Mémoïsation

- La **mémoïsation** consiste à stocker dans une structure de données les valeurs renvoyées par une fonction afin de ne pas les recalculer lors des appels identiques suivant.
- Les tableaux associatifs sont alors des structures de données adaptées, les clés sont les paramètres d'appels de la fonction et les valeurs le résultat de l'appel.

Mémoïsation

- La **mémoïsation** consiste à stocker dans une structure de données les valeurs renvoyées par une fonction afin de ne pas les recalculer lors des appels identiques suivant.
- Les tableaux associatifs sont alors des structures de données adaptées, les clés sont les paramètres d'appels de la fonction et les valeurs le résultat de l'appel.
- On peut alors tester si la valeur a déjà été calculée (présence de la clé) en $O(1)$ et suivant le cas de figure calculer la valeur associée puis la stocker ($O(1)$) ou alors la récupérer ($O(1)$).