Plan

1. Introduction

2. Pile et file

3. Dictionnaire
    - Principe
    - Implémentation par tableau
    - Implémentation par liste liée
    - Implémentation par table de hachage
    - Illustration

Plan

1. Introduction

2. Pile et file

3. Dictionnaire
    - Principe
    - Implémentation par tableau
    - Implémentation par liste liée
    - Implémentation par table de hachage
    - Illustration

Dictionnaire : principe

Un dictionnaire est une collection de paires (clé, valeur) où

- clé est une valeur permettant d'identifier de manière unique un élément du dictionnaire. Par exemple : un entier, une chaîne de caractères, etc.
- valeur est une valeur qu'on souhaite associer à cet élément.

Interface :

- `Insert(d, key, value)` : insère la paire (`key`,`value`) dans le dictionnaire `d`. Si la clé s'y trouve déjà, sa valeur est mise à jour.
- `Search(d, key)` : cherche la clé `key` dans le dictionnaire `d`. Si la valeur est trouvée, la valeur associée est renvoyée, sinon `NULL`.
- `Remove(d, key)` : supprime la clé `key` (et sa valeur) du dictionnaire.
- + création d'un dictionnaire vide et libération de la mémoire.
- + parcours de toutes les clés

Dictionnaire : principe

Un dictionnaire s'appelle aussi un tableau associatif ou une table de symboles.

Un dictionnaire peut être vu comme un généralisation d'un tableau dont les indices sont remplacés par n'importe quel type de clé.

`Insert(d, "Pierre",4)` $\Leftrightarrow$ `d["coucou"] = 4`

Les clés sont souvent supposées pouvoir être ordonnées mais ce n'est pas toujours nécessaire.

Contraintes de performance implicites :

- Les opérations d'insertion et de recherche doivent être les plus efficaces possibles.
- L'espace mémoire consommé doit être minimal et idéalement s'adapter au nombre de clés.

Applications

Les applications sont très nombreuses :

- Liste de contacts : clé = nom, valeur = numéro de téléphone, adresse
- Dictionnaire : clé = mot, valeur = définition
- Recherche internet : clé = mot clé, valeur = liste de pages web
- domain name service (DNS) : clé = nom de domaine (`www.uliege.be`), valeur = adresse IP (`139.165.X.Y`)
- Compilateur : clé = nom de variable, valeur = valeur et type de la variable.
- Système de fichier : clé = nom de fichier, valeur = localisation du fichier sur le disque
- ...

Ensemble : un dictionnaire simplifié

Un ensemble (set) est une collection de clés uniques.

Équivalent à un dictionnaire, sans valeurs associées aux clés.

Interface :

- `Insert(s, key)` : ajoute la clé `key` dans l'ensemble `s` si elle ne s'y trouve pas déjà.
- `Contains(s, key)` : renvoie 1 si la clé `key` est contenue dans l'ensemble `s`, 0 sinon.
- `Remove(s, key)` : supprime la clé `key` de l'ensemble `s`.
- $+$ création et libération de la structure.
- $+$ parcours des clés de l'ensemble.

Les techniques d'implémentation d'un dictionnaire peuvent s'adapter trivialement à l'ensemble.

Exemple d'interface en c

En supposant des clés de type `int` et des valeurs de type `void *`.

dict.h

```c
#ifndef _DICT_H
#define _DICT_H

typedef struct Dict_t Dict;

Stack *dictCreate();
void   dictFree(Stack *);
void   dictInsert(Dict *, int, void *)
void  *dictSearch(Dict *, int);
int    dictContains(Dict *, int);
void  *dictRemove(Dict *, int);

#endif
```

set.h

```c
#ifndef _SET_H
#define _SET_H

typedef struct Set_t Set;

Stack *setCreate();
void   setFree(Stack *);
void   setInsert(Set *, int);
int    setContains(Set *, int);
void  *setRemove(Set *, int);

#endif
```

Remarque : on ne peut pas remplacer le type de la clé par un type `void *`, sauf à rajouter des pointeurs de fonctions en argument aux fonctions `dicoCreate` et `setCreate`. Par exemple pour passer une fonction de comparaison des clés.

Application 1 : filtrage d'adresses emails

(voir Partie 5, slide 234)

```
...
#include "set.h"

int main() {
  char buffer[10000];

  // construction de l'ensemble
  Set *s = setCreate(100);
  while (fgets(buffer, 1000, stdin)) {
    int lenstr = strlen(buffer) - 1;
    buffer[lenstr]='\0';
    setInsert(s, strdup(newstring));
  }

  // filtrage
  if (setContains(s, "p.geurts@uliege.be")) {
    ...
  }
  setFree(s);
}
```

Utilisation : `./filter < adresses.txt`

Application 2 : suppression des doublons dans un fichier

```
...
#include "set.h"

int main() {
  char buffer[1000];
  Set *s = setCreate(100);
  while (fgets(buffer, 1000, stdin)) {
    int lenstr = strlen(buffer)-1;
    buffer[lenstr]='\0';

    if (!setContains(s, buffer)) {
      printf("%s\n", buffer);
      setInsert(s, strdup(newstring));
    }
  }
  setFree(s);
}
```

Utilisation : `./removeduplicates < list.txt > listunique.txt`

Application 3 : compter la fréquence des mots

```
...
#include "dict.h"

int main() {
  char buffer[1000];
  Dict *d = dictCreate(25000);

  while (fgets(buffer, 1000, stdin)) {
    char *string = buffer;
    char *token;
    while ((token = strsep(&string," \t\n,;.:?\"\r*-_()[]'")) != NULL) {
      if (strlen(token)>0) {
        int c = dictSearch(d, token);
        if (c == -1) {
          dictInsert(d, strdup(token), 1);
        } else {
          dictInsert(d, token, c+1);
        }
      }
    }
  }
  dictPrint(d); // affiche le contenu du dictionnaire
  dictFree(d);
}
```

Utilisation : `./countwords < le_rouge_et_le_noir.txt > countings.csv`

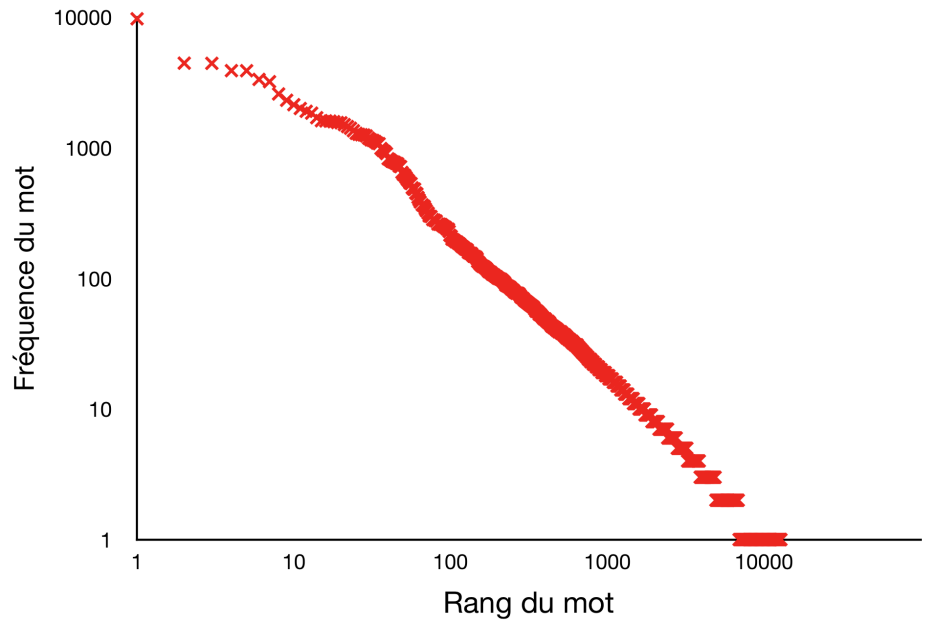
Application 3 : compter la fréquence des mots

Application en linguistique, compression de données, etc.

La loi de Zipf[5] dit que la fréquence du $n$ème mot le plus fréquent dans un texte est $f(n) = \frac{K}{n}$ où K est une constante.

Exemple : "le rouge et le noir" de Stendhal[6] (180000 mots dont 13000 uniques)

Mot	Rang	Freq.
de	1	9738
il	2	4454
la	3	4439
le	4	3902
à	5	3898
l	6	3344
et	7	3212
un	8	2587
que	9	2315
d	10	2140



5. https://fr.wikipedia.org/wiki/Loi_de_Zipf
6. Téléchargé de https://www.gutenberg.org

Plan

1. Introduction

2. Pile et file

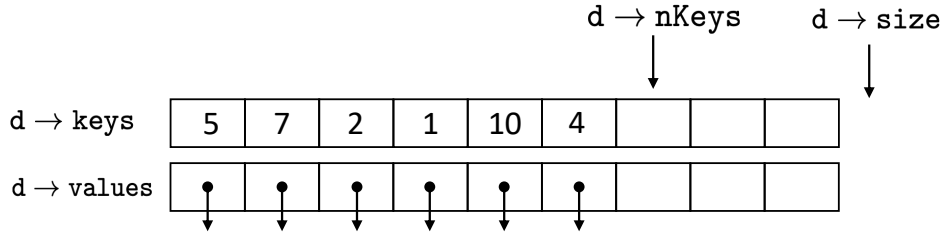
3. Dictionnaire
    - Principe
    - Implémentation par tableau
    - Implémentation par liste liée
    - Implémentation par table de hachage
    - Illustration

Implémentation par tableau

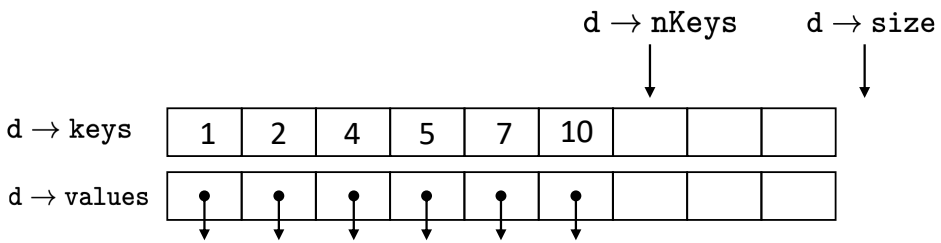
```
struct Dict_t {
  int *keys;           // clés à valeurs entières
  void **values;       // valeurs de type (void *)
  unsigned int size;   // taille du tableau
  unsigned int nKeys; // nombre de clés
};
```

Deux options :

- Tableau de clés non trié : recherche linéaire et insertion à la fin



- Tableau de clés trié : recherche dichotomique et insertion en décalant vers la droite.



(Implémentation laissée comme exercice)

Complexité

En temps : *(en fonction du nombre de clés n dans le dictionnaire)*

- Tableau non trié :
  - Recherche : $O(n)$
    - On parcourt le tableau jusqu'à trouver la clé recherchée ($O(n)$).
  - Insertion : $O(n)$
    - On cherche la clé dans le tableau ($O(n)$) : si présente, on écrase sa valeur ($O(1)$), sinon on la place à la fin du tableau ($O(1)$).
- Tableau trié :
  - Recherche : $O(\log n)$
    - On utilise la recherche dichotomique.
  - Insertion : $O(n)$
    - On insére la clé en fin de tableau et on la fait remonter vers la gauche jusqu'à sa position dans l'ordre (cf. tri par insertion)

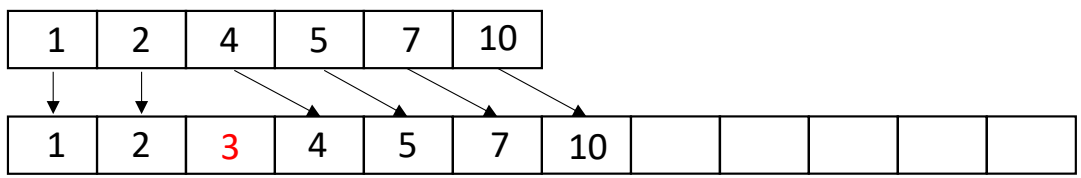
En espace : $O(1)$, car la taille du tableau ne dépend pas du nombre de paires (clé, valeur) stockées.

Tableau extensible

Problème de l'approche précédente : la taille du tableau est fixée et n'évolue pas en fonction des données (complexité en espace $O(1)$).

Solution :

- Initialisez les tableaux `keys` et `values` à une certaine taille.
- Quand les tableaux deviennent trop petits :
  - On alloue des nouveaux tableaux, deux fois plus grands.
  - On recopie les anciennes clés et valeurs dans ces nouveaux tableaux.
  - On libére les anciens tableaux.



Complexités :

- en temps : $O(n)$ (inchangée)
- en espace : $O(n)$ (au pire, on gaspille $O(n)$).

Plan

1. Introduction

2. Pile et file

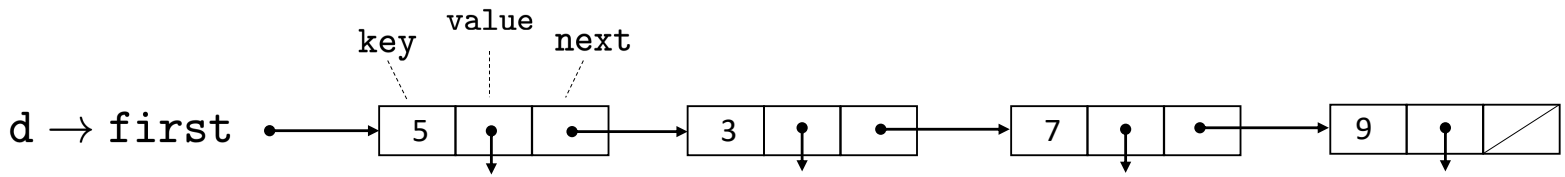
3. Dictionnaire
    - Principe
    - Implémentation par tableau
    - Implémentation par liste liée
    - Implémentation par table de hachage
    - Illustration

Implémentation par liste liée

```
typedef struct Node_t {
  int key;                // clés à valeurs entières
  void *value;            // valeurs de type (void *)
  struct Node_t *next;
} Node;

struct Dict_t {
  Node *first;
  unsigned int nKeys;  // nombre de clés (optionnel)
};
```

Recherche en parcourant la liste, insertion en début de liste.



(Implémentation laissée comme exercice)

Complexités

En temps : *(en fonction du nombre de clés n dans le dictionnaire)*

- Recherche : $O(n)$
  - On parcourt la liste depuis le début et on s'arrête quand on trouve la clé recherchée (ou quand la fin de liste est atteinte).
- Insertion : $O(n)$
  - On recherche la clé dans la liste ($O(n)$). Si présente, on écrase sa valeur ($O(1)$), sinon on l'ajoute en début de liste ($O(1)$).

En espace : $O(n)$

- L'espace mémoire nécessaire croît linéairement avec le nombre de paires stockées

Plan

1. Introduction

2. Pile et file

3. Dictionnaire
    - Principe
    - Implémentation par tableau
    - Implémentation par liste liée
    - Implémentation par table de hachage
    - Illustration

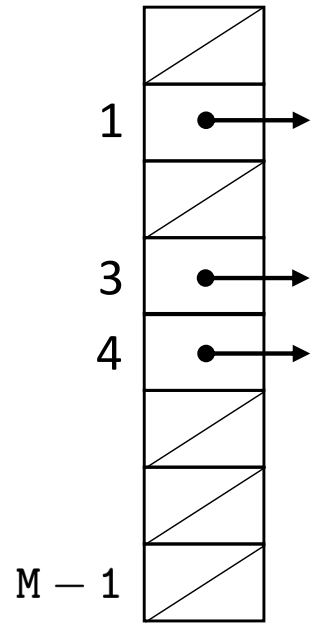
Implémentation par tableau : deuxième version

Si on suppose que les clés prennent des valeurs entre 0 et $M-1$ avec M petit, on peut obtenir une implémentation beaucoup plus efficace.

```
struct Dict_t {
  void *values[M];
}

void dictInsert(Dict d, int key, void *value) {
  d -> values[key] = value;
}

void *dictSearch(Dict d, int key) {
  return values[key];
}
```



Complexités :

- en temps : $O(1)$ pour la recherche et l'insertion.
- en espace : $O(1)$, car l'espace mémoire ne dépend pas du nombre de clés.

Table de hachage : fonction de hachage

Problème de l'approche précédente : les clés doivent être entières et prendre des valeurs pas trop grandes.

Idée 1 : Soit U l'ensemble des valeurs possibles de clés, même non entières. On définit une fonction de hachage h :

$$h : U \to \{0, \ldots, M-1\}$$

envoyant chaque clé $k \in U$ vers une position $h(k)$ dans la table.

Insertion et recherche modifiées (toujours $O(1)$ si h est $O(1)$) :

```
void dictInsert(Dict d, int key, void *value) {
  d -> values[h(key)] = value;
}

void *dictSearch(Dict d, int key) {
  return values[h(key)];
}
```

Table de hachage : gestion des collisions

Problème : Sans connaître les clés, il est difficile de garantir que $h(k_1) \neq h(k_2)$ lorsque $k_1 \neq k_2$. Et c'est impossible dès que $|U| > M \Rightarrow$ le code précédent est incorrect.

Idée 2 : Chaque case i de la table contient une liste liée retenant toutes les clés k insérées telles que $h(k) = i$.

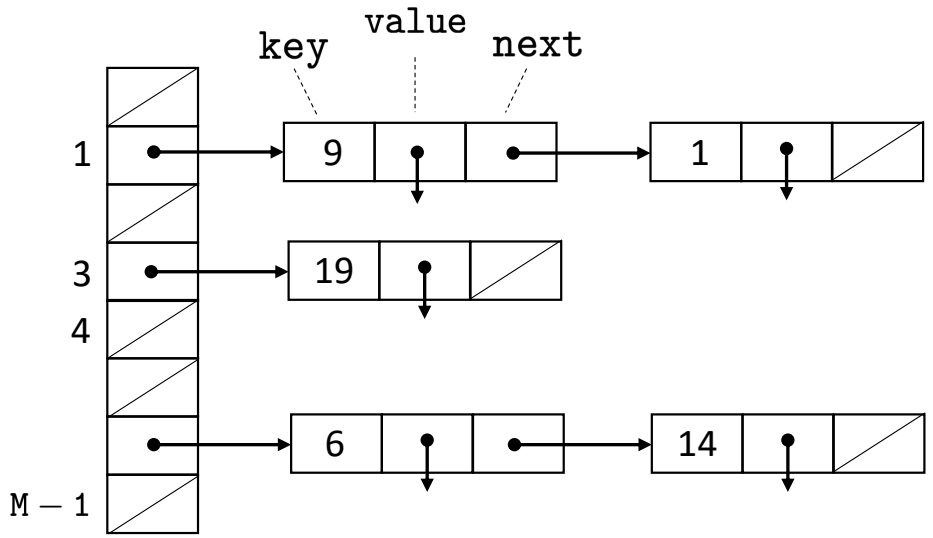


Table de hachage : implémentation (structure et création)

```c
typedef struct Node_t {
  int key;
  void *value;
  struct Node_t *next;
} Node;

struct Dict_t {
  Node **array;
  unsigned int arraySize; // taille du tableau
  unsigned int nKeys;     // nombre de clés (optionnel)
};

Dict *dictCreate(int m) {
  Dict *d = malloc(sizeof(Dict));
  if (d == NULL) exit(-1);
  d -> array = calloc(m, sizeof(Node));
  if (d -> array == NULL) exit(-1);
  d -> arraySize = m;
  d -> nKeys = 0;
  return d;
}
```

Remarque : Contrairement à `malloc`, `calloc` initialise la mémoire à 0 (ou NULL) en plus de faire l'allocation. Nécessaire ici !

Table de hachage : implémentation (recherche)

```
void *dictSearch(Dict d, int key, void *value) {
  Node *p = d -> array[h(d, key)];
  while (p != NULL && p -> key != key)
    p = p -> next;

  if (p != NULL)
    return p -> value;
  else
    return NULL;
}
```

Remarque : La fonction de hachage `h` prend la table en argument (voir plus loin).

Table de hachage : implémentation (insertion)

```
void *dictInsert(Dict d, int key, void *value) {
  Node *p = d -> array[h(d, key)];
  while (p != NULL && p -> key != key)
    p = p -> next;
  if (p != NULL)
    p -> value = value;
  else {
    Node *newNode = malloc(sizeof(Node));
    newNode -> key = key;
    newNode -> value = value;
    newNode -> next = d -> array[h(d, key)];
    d -> array[h(d, key)] = newNode;
    d -> nKeys++;
  }
}
```

Complexité : éléments d'analyse

La complexité dépend de la taille M de la table et de la fonction de hachage (supposée $O(1)$).

Dans le pire cas :

- Toutes les clés sont envoyées dans la même case.
- $\Rightarrow$ Insertion et recherche en $O(n)$.

Dans le meilleur cas :

- Les clés sont réparties uniformément dans toutes les cases de la table.
- $\Rightarrow$ Insertion et recherche en $O(n/M)$.
- $\Rightarrow$ Proche de $O(1)$ si M est $O(n)$.

Complexité en espace : $O(M + n)$ (pour la table et pour les éléments de liste liée).

Fonction de hachage

Le choix de la fonction de hachage détermine l'efficacité des opérations :

- Elle doit être facile à calculer (c'est-à-dire $O(1)$).
- Elle doit répartir les clés aussi uniformément que possible dans la table (très difficile à assurer).

Lorsque les clés sont à valeurs entières, une approche simple est la méthode de division :

$$h(k) = k \mod M.$$

En C :

```
static unsigned int h(Dict *d, int key) {
  return key % d->arraySize;
}
```

Fonction de hachage : chaînes de caractères

Lorsque la clé n'est pas à valeur entière, il faut d'abord passer par une fonction d'encodage.

Exemples pour les chaînes de caractères :

- On additionne les caractères de la chaîne (naïf) :

```c
static unsigned int h(Dict *d, char *key) {
  unsigned int hash = 0;
  while (*key != '\0') {
    hash += *key;
    key++;
  }
  return hash % d->arraySize;
}
```

- Une meilleure approche (`djb2`) :

```c
static unsigned int h(Dict *d, char *key) {
  unsigned long hash = 5381;
  while (*key != '\0') {
    hash = hash * 33 + *key;
    key++;
  }
  return hash % d->arraySize;
}
```

Implémentation générique

On peut définir une table de hachage plus générique en stockant dans la structure une fonction de comparaison et une fonction d'encodage de clés (passées en arguments à `dictCreate`) :

```
typedef struct Node_t {
  void *key;
  void *value;
  struct Node_t *next;
} Node;

struct Dict_t {
  Node **array;
  unsigned int arraySize;
  unsigned int nKeys;

  int (*compareFunction)(const void *key1, const void *key2);
  unsigned long (*encodeKey)(const void *key);
};

static unsigned int h(Dict *d, void *key) {
  return d -> encodeKey(key) % d->arraySize;
}
```

Plan

1. Introduction

2. Pile et file

3. Dictionnaire
    - Principe
    - Implémentation par tableau
    - Implémentation par liste liée
    - Implémentation par table de hachage
    - Illustration

Application au filtrage d'adresses email

Quelle est la complexité du code du slide 320 en fonction de l'implémentation de l'ensemble ?

Création et remplissage de l'ensemble :

- $O(n^2)$ dans le pire cas pour toutes les implémentations
- $O(n \log n)$ avec l'implémentation tableau si on trie le tableau seulement quand toutes les adresses ont été lues (possible seulement si toutes les adresses sont connues à l'avance).
- $O(n)$ avec la table de hachage si fonction de hachage uniforme et table suffisamment grande.

Recherche d'une adresse :

- $O(n)$ dans le pire cas pour la liste liée et la table de hachage.
- $O(\log n)$ pour le tableau trié.
- $O(1)$ avec la table de hachage si fonction de hachage uniforme et table suffisamment grande.

(Complexité de la suppression de doublons et du comptage de mots ?)

Application au filtrage d'adresses email

Tests empiriques :

- Génération de données : n chaînes de caractères (`a-z`) aléatoires de longueur 10.
- Requêtes : $10n$ chaînes prises au hasard dans la liste (recherches positives) ou en dehors (recherches négatives).

Tableau trié (par fusion) et recherche dicho. versus table de hachage ($M = 2n$, `djb2`) :

Création de la structure :

n	Tri (s)	Table (s)
12500	0,003	0,002
25000	0,005	0,002
50000	0,012	0,007
100000	0,026	0,013
200000	0,057	0,036
400000	0,128	0,085

Recherches positives

n	Rech. dico. (s)	Table (s)
500000	3,09	1,31
1000000	7,95	2,89

Recherches négatives

n	Rech. dico. (s)	Table (s)
500000	3,76	1,23
1000000	9,21	2,72

La table de hachage permet de traiter trois fois plus d'adresses à la seconde.

Conclusion

L'implémentation par table de hachage est en pratique plus efficace que les implémentations par liste liée et par tableau, malgré une complexité dans le pire cas identique, voire moins bonne.

Les performances de la recherche dans un tableau trié sont plus stables mais l'insertion est beaucoup plus couteuse, à cause du décalage.

D'autres structures existent qui permettent de combiner la facilité d'insertion de la liste liée avec la rapidité et la stabilité de la recherche dichotomique. Par exemple les arbres binaires de recherche (INFO0902).

Il existe aussi des structures spécificiques pour les chaînes de caractères. Par exemple les tries.