Première partie

Exercices pour les cours-TD

Structures de données (ZZ1, S2)

TD 1 - tables de hachage

1 Table de hachage avec des listes

Considérons une table de hachage utilisée pour stocker des chaînes de caractères. La table de hachage contient $n = 10$ entrées et utilise la fonction $h(s) = \text{length}(s) \bmod 10$. Chaque entrée $i \in [1; n]$ de la table contient une liste, utilisée pour stocker toutes les chaînes s telles que $h(s) = i$.

On représentera par $c \times z$ la chaîne de caractères composée du caractère c écrit z fois. Par exemple, $a \times 3$ représentera la chaîne de caractères aaa.

Question 1 : Représentez la table de hachage contenant les chaînes de caractères $a \times 3$, $b \times 8$ et $c \times 4$.

Question 2 : Représentez la nouvelle table de hachage lorsque vous ajoutez la chaîne de caractères $d \times 13$.

Supposons qu'il y a N chaînes de caractères stockées dans la table de hachage, avec au plus M chaînes par liste.

Question 3 : Montrez que $N \leq n.M$.

Question 4 : Montrez que la recherche d'une chaîne de caractères s dans la table de hachage se fait en $\mathcal{O}(M)$ comparaison de chaînes de caractères.

Question 5 : Si la table de hachage est bien adaptée aux données, on a les deux propriétés suivantes :

— La fonction de hachage h distribue bien les données dans les cases. Dit autrement, $M \approx N/n$.

— Le nombre de cases de la table est du même ordre de grandeur que le nombre de données N. Dit autrement, $n \approx N$.

Dans l'hypothèse où ces propriétés sont vérifiées, quelle est la complexité d'une recherche dans la table de hachage ?

Question 6 : Rappelez la complexité d'une recherche d'une chaîne de caractères dans une liste de N chaînes, et donnez les avantages d'une table de hachage.

Question 7 : Donnez les inconvénients d'une table de hachage.

2 Table de hachage avec adressage ouvert

Il existe une manière de représenter les tables de hachage qui n'utilise pas de liste, mais uniquement un tableau. Quand une collision se produit pour l'ajout d'une chaîne s, on parcourt toutes les cases du tableau dans l'ordre, à partir de $h(s)$, jusqu'à en trouver une qui est libre. La chaîne s est stockée à cet endroit.

Question 8 : Représentez la table de hachage contenant les chaînes de caractères $a \times 3$, $b \times 8$ et $c \times 4$ (ajoutées dans cet ordre). Ajoutez la valeur $d \times 13$ et représentez la nouvelle table de hachage.

Question 9 : Pourquoi ne peut-on pas facilement supprimer une entrée dans une telle table de hachage ?

Pour supprimer des entrées dans une telle table de hachage, on utilise une entrée spéciale, notée *suppr*. La recherche d'une chaîne de caractères se fait en s'arrêtant sur des cases vides (mais pas sur des cases contenant *suppr*). L'ajout d'une chaîne de caractères se fait en s'arrêtant sur des cases vides ou sur des cases contenant *suppr*.

Question 10 : Représentez la nouvelle table de hachage après suppression de la valeur $a \times 3$.

Structures de données (ZZ1, S2)

TD 8 - tris

Question 1 : Dans un tableau non trié, quelle est la complexité dans le pire des cas de la recherche d'un élément ?
Question 2 : Dans un tableau trié, quelle est la complexité dans le pire des cas de la recherche dichotomique d'un élément ?
Question 3 : S'il y a peu de modifications du tableau et beaucoup de recherches, est-ce qu'il est utile de trier le tableau ?

1 Tri par sélection du maximum

Question 4 : Déroulez l'algorithme du tri par sélection du maximum sur le tableau suivant : $(11, 12, 14, 13, 10)$. Vous redessinerez le tableau au moins à chaque fois qu'il est modifié.
Question 5 : Déroulez l'algorithme du tri par sélection du maximum sur le tableau suivant : $(mouse, cat, chicken, dog, horse)$, en considérant l'ordre lexicographique. Vous redessinerez le tableau au moins à chaque fois qu'il est modifié.
Question 6 : Quel est la complexité de l'algorithme quand le tableau initial est déjà trié ?

2 Tri rapide

Question 7 : Considérons le tableau suivant : $(3, 9, 6, 1, 7)$. Triez ce tableau avec l'algorithme du tri rapide, en prenant pour pivot l'élément situé à l'indice médian (indice arrondi à l'inférieur) du tableau à chaque itération. Vous représenterez toutes les étapes du tri.
Question 8 : Considérons le tableau suivant : $(5, 1, 3, 9, 6, 2, 8, 4, 7)$. Triez ce tableau avec l'algorithme du tri rapide, en prenant pour pivot l'élément situé à l'indice médian (indice arrondi à l'inférieur) du tableau à chaque itération. Vous représenterez toutes les étapes du tri.
Question 9 : Quelle est la complexité du tri rapide si tous les éléments sont triés, et qu'on choisit le pivot comme l'élément médian ?

3 Tri fusion

Question 10 : Considérons le tableau suivant : $(7, 9, 3, 2, 6, 4, 1, 5, 8)$. Triez ce tableau avec l'algorithme du tri fusion. Vous représenterez toutes les étapes du tri.
Question 11 : Ecrivez un algorithme permettant de fusionner deux listes l_1 et l_2, toutes les deux triées.
Question 12 : Quelle est la complexité dans le pire des cas de cet algorithme de fusion de deux listes triées ?
Question 13 : Quelle est la complexité dans le pire des cas de l'algorithme de tri fusion ?

Deuxième partie

Fiches de TP

Structures de données (ZZ1, S2)

TP 1 - rappel sur les listes

Dans ce TP, nous allons programmer en langage C quelques fonctions sur les listes, vues dans la matière "structures de données" du premier semestre.

Il est recommandé de tester vos fonctions au fur et à mesure que vous les écrivez. Notez que :

— quelques tests sont proposés dans les parties 1.3 et 2.3,

— des fichiers à compléter sont à votre disposition sur l'ENT.

1 Implémentation d'une pile par liste simplement chaînée

Nous cherchons à implémenter une structure de données de type pile d'entiers (nommée *stack* en anglais). Nous souhaitons que les opérations de base aient une complexité en $\mathcal{O}(1)$, à savoir : `stackCreate`, `stackIsEmpty`, `stackAdd`, `stackTop` et `stackRemove`. Cela peut se programmer avec une liste simplement chaînée, en ajoutant en tête et en retirant en tête.

En plus de ces opérations de base, vous aurez à implémenter d'autres fonctions utiles : `stackSize`, `stackDisplay` et `stackFree`. La complexité de ces opérations sera en $\mathcal{O}(n)$, où n est la taille de la pile.

Vous utiliserez la structure de données suivante :

```
typedef struct stack {
  int value;
  struct stack * next;
} stack;
```

1.1 Opérations de base

Pour toutes les opérations de base qui suivent, vous n'aurez pas de boucle à utiliser.

Question 1 : La fonction `stackCreate` a le prototype suivant : `stack * stackCreate();`. Elle crée une pile vide. Pour cela, il suffit de retourner un élément dont la valeur est `NULL` (définie notamment dans <stdlib.h>). Programmez la fonction `stackCreate`.

Question 2 : La fonction `stackIsEmpty` a le prototype suivant : `int stackIsEmpty(stack * s);`. Elle retourne `1` si la pile `s` est vide, et `0` sinon. Programmez la fonction `stackIsEmpty`.

Question 3 : La fonction `stackAdd` a le prototype suivant : `stack * stackAdd(stack * s, int v);`. Elle retourne une nouvelle pile qui contient un élément de valeur `v`, et dont le suivant pointe sur `s`. Programmez la fonction `stackAdd()`.

Remarque : vous aurez à utiliser la fonction `malloc` qui se trouve dans la bibliothèque <stdlib.h>

Question 4 : La fonction `stackTop` a le prototype suivant : `int stackTop(stack * s);`. Elle retourne la valeur au sommet de la pile `s` (en faisant l'hypothèse que la pile n'est pas vide). Programmez la fonction `stackTop`.

Question 5 : La fonction `stackRemove` a le prototype suivant : `stack * stackRemove(stack * s);`. Elle retourne la pile sans son premier élément (en faisant l'hypothèse que la pile n'est pas vide). Programmez la

fonction `stackRemove`.
Remarque : vous aurez à utiliser la fonction `free` qui se trouve dans la bibliothèque `<stdlib.h>`.

1.2 Opérations utiles

Pour toutes ces opérations, vous aurez une boucle à faire.
Question 6 : La fonction `stackSize` a le prototype suivant : `int stackSize(stack * s);`. Elle retourne le nombre d'éléments de la pile. Programmez la fonction .
Question 7 : La fonction `stackDisplay` a le prototype suivant : `void stackDisplay(stack * s);`. Elle affiche tous les entiers de la pile. Programmez la fonction `stackDisplay`.
Remarque : vous aurez à utiliser la fonction `printf` de la bibliothèque `<stdio.h>`.
Question 8 : La fonction `stackFree` a le prototype suivant : `void stackFree(stack * s);`. Elle supprime tous les éléments de la pile. Programmez la fonction `stackFree`.

1.3 Tests

- Créez une pile vide, et affichez le résultat de la fonction `stackIsEmpty` sur cette pile.
- Créez une pile vide et affichez la.
- Créez une pile vide et effacez tous ses éléments.
- Créez une pile vide, ajoutez 5, puis ajoutez 10, et affichez ses éléments. Vous devriez voir 10 s'afficher en premier, puis 5.
- Créez une pile vide, ajoutez deux éléments, et affichez le résultat de la fonction `stackIsEmpty` sur cette pile.
- Créez une pile vide, ajoutez 10 éléments, et affichez les.

2 Listes doublement chaînées et files (partie bonus)

Nous cherchons à présent à implémenter une structure de données de type file d'entiers (nommée `queue` en anglais). Nous souhaitons que les opérations de base aient une complexité en $\mathcal{O}(1)$, à savoir : `queueCreate`, `queueIsEmpty`, `queueAdd`, `queueGet`, `queueRemove` et `queueDestroy`. Cela peut se programmer avec une liste doublement chaînée, en ajoutant en tête et en retirant en fin.

En plus de ces opérations de base, vous aurez à implémenter d'autres fonctions utiles : `queueSize` et `queueDisplay`. La complexité de `queueSize` sera en $\mathcal{O}(1)$, et la complexité de `queueDisplay` sera en $\mathcal{O}(n)$, où n est la taille de la file.

Vous utiliserez les structures de données suivantes :

```
typedef struct queue_element {
  int value;
  struct queue_element * previous;
  struct queue_element * next;
} queue_element;
typedef struct queue {
  int size;
  struct queue_element * head;
  struct queue_element * tail;
}
```

2.1 Opérations de base

Pour toutes les opérations de base qui suivent, vous n'aurez pas de boucle à utiliser.
Question 9 : La fonction `queueCreate` a le prototype suivant : `queue * queueCreate();`. Elle retourne une file vide. Notez qu'une file vide a une taille de 0, n'a pas de tête, et n'a pas de queue. Programmez la

fonction `queueCreate`.
Remarque : vous aurez à utiliser la fonction `malloc`.

Question 10 : La fonction `queueIsEmpty` a le prototype suivant : `int queueIsEmpty(queue * q);`. Elle retourne `1` si `q` est vide, et `0` sinon. Programmez la fonction `queueIsEmpty`.

Question 11 : La fonction `queueAdd` a le prototype suivant : `void queueAdd(queue * q, int v)`. Elle ajoute la valeur `v` en tête de file. Programmez la fonction `queue`.
Remarque 1 : vous pouvez vous inspirer des figures 3.1, 3.2 et 3.3.
Remarque 2 : pensez bien aux deux cas suivants :

- si la file initiale est vide (elle n'a donc ni tête, ni queue), il faut modifier la tête et la queue de la file,
- si la file initiale n'est pas vide, il faut seulement modifier sa tête.

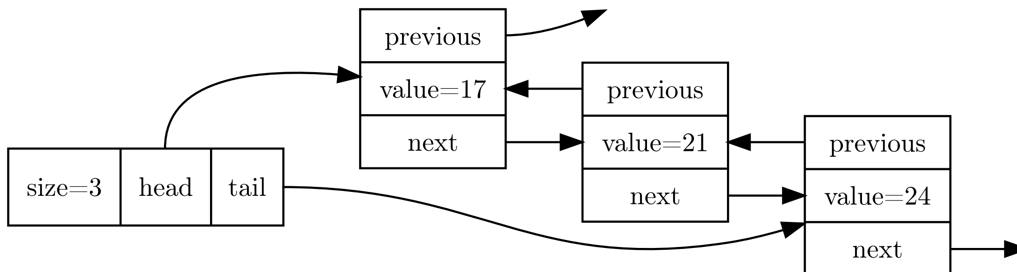

Figure 3.1 – Exemple de pile avant l'ajout.

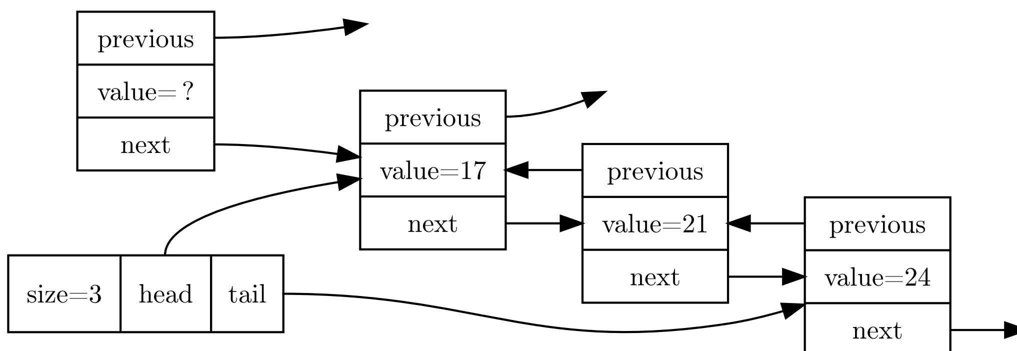

Figure 3.2 – Exemple de pile pendant l'ajout : le nouvel élément est créé et ses pointeurs sont corrects, mais il reste des choses à modifier dans la pile initiale.

Question 12 : La fonction `queueGet` a le prototype suivant : `int queueGet(queue * q);`. Elle retourne la valeur du dernier élément de la file (supposée non vide). Programmez la fonction `queueGet`.

Question 13 : La fonction `queueRemove` a le prototype suivant : `void queueRemove(queue * q);`. Elle supprime le dernier élément de la file (supposée non vide). Programmez la fonction `queueRemove`.
Remarque : vous aurez à utiliser la fonction `free`.

Question 14 : La fonction `queueDestroy` a le prototype suivant : `void queueDestroy(queue * q);`. Elle supprime tous les éléments de la file, puis supprime la file. Programmez la fonction `queueDestroy`.

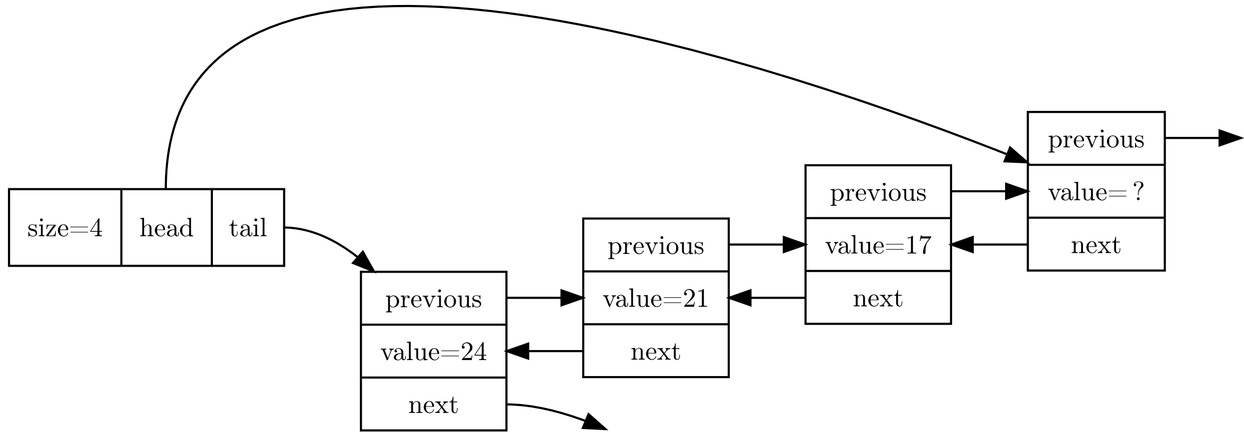

FIGURE 3.3 – Exemple de pile après l'ajout.

2.2 Opérations utiles

Question 15 : La fonction `queueSize` a le prototype suivant : `int queueSize(queue * q)`. Elle retourne le nombre d'éléments dans la file, calculé en $\mathcal{O}(1)$. Programmez la fonction `queue`.
Question 16 : La fonction `queueDisplay` a le prototype suivant : `void queueDisplay(queue * q);`. Elle affiche tous les éléments de la file, dans l'ordre. Programmez la fonction `queueDisplay`.

2.3 Tests

— Créez une file vide, et affichez le résultat de la fonction `queueIsEmpty` sur cette file.
— Créez une file vide et affichez la.
— Créez une file vide et effacez tous ses éléments.
— Créez une file vide, ajoutez 5, puis ajoutez 10, et affichez ses éléments. Vous devriez voir 5 s'afficher en premier, puis 10.
— Créez une file vide, ajoutez trois éléments, effacez-en un, et affichez la file.
— Créez une file vide, ajoutez un élément, effacez-en un, et affichez la file.
— Créez une file vide, ajoutez 100 éléments, effacez-en 100, et affichez la file.

Structures de données (ZZ1, S2)

TP 2 - tables de hachage

Dans ce TP, nous allons implémenter un dictionnaire associant des chaînes de caractères clés à des chaînes de caractères valeurs. Le dictionnaire utilise un tableau de n indices (n valant 10 par défaut, mais pouvant être modifié ensuite), dont chaque case pointe sur une liste de couples $(cle, valeur)$. Notez que la clé cle ne peut apparaître qu'une fois dans une table de hachage.

Vous utiliserez la structure suivante :

```
typedef struct {
  int n;
  list ** tab;
} hashtable;
typedef struct list {
  char * key;
  char * value;
  struct list * next;
} list;
```

Pensez à tester vos fonctions au fur et à mesure. Des tests se trouvent dans la partie 3.

1 Fonctions minimales à implémenter

Question 1 : La fonction `listCreate` a le prototype suivant : `list * listCreate();`. Elle crée une liste vide. Programmez la fonction `listCreate`.
Question 2 : La fonction `hashtableCreate` a le prototype suivant : `hashtable * hashtableCreate();`. Elle crée une table de hachage avec un tableau de $n = 10$ indices. Programmez la fonction `hashtableCreate`.
Question 3 : La fonction `hash` a le prototype suivant : `int hash(char * key, int n);`. Elle transforme la chaîne de caractères `key` en un entier compris entre 0 et $n - 1$. Pour cela, elle calcule la somme de tous les octets de la chaîne de caractères modulo n. Programmez la fonction `hash`.
Question 4 : La fonction `listDisplay` a le prototype suivant : `void listDisplay(list * l);`. Elle affiche la liste `l`. Programmez la fonction `listDisplay`.
Question 5 : La fonction `hashtableDisplay` a le prototype suivant : `void hashtableDisplay(hashtable * h);`. Elle affiche les listes de toutes les cases du tableau de `h`. Programmez la fonction `hashtableDisplay`.
Question 6 : La fonction `listAdd` a le prototype suivant : `list * listAdd(list * l, char * newK, char * newV);`. Elle retourne une liste contenant tous les éléments de l'ancienne liste `l`, ainsi qu'un nouvel élément ayant pour valeur le couple (`newK`, `newV`). Vous pouvez ajouter ce nouvel élément où vous le souhaitez dans la liste. Programmez la fonction `listAdd`.
Remarque : l'élément contiendra les pointeurs `newK` et `newV`, et non pas une copie des chaînes de caractères `newK` et `newV`. C'est plus simple à programmer, mais ce n'est généralement pas recommandé de faire ainsi.
Question 7 : La fonction `listSearch` a le prototype suivant : `char * listSearch(list * l, char * k);`. Elle retourne `NULL` si `key` n'est pas présente dans la liste, et la valeur `value` si le couple (`key`, `value`)

est présent dans la liste. Programmez la fonction `listSearch`.
Remarque : Pour comparer deux chaînes de caractères, vous pourrez utiliser la fonction `strcmp` de la bibliothèque `<string.h>`.
Question 8 : La fonction `hashtableSearch` a le prototype suivant : `char * hashtableSearch(hashtable * h, char * key);`. Elle retourne `NULL` si `key` n'est pas présente dans la table de hachage, et la valeur `value` si le couple (`key`, `value`) est présent dans la table de hachage. Programmez la fonction `hashtableSearch`.
Question 9 : La fonction `hashtableAdd` a le prototype suivant : `void hashtableAdd(hashtable * h, char * key, char * value);`. Elle ajoute le couple (`key`, `value`) dans la table de hachage, dans la liste correspondant à la case `hash(key, h->n)`. Avant d'ajouter un couple, vous vérifiez que la clé n'est pas déjà présente dans la table de hachage. Programmez la fonction `hashtableAdd`.

2 Fonctions bonus

Question 10 : La fonction `hashtableFree` a le prototype suivant : `void hashtableFree(hashtable * h);`. Elle efface toutes les données allouées dans la table de hachage, c'est-à-dire toutes les listes, puis la table de hachage elle-même. Programmez la fonction `hashtableFree`.
Remarque : vous pourrez programmer une fonction `listFree` qui efface toutes les données d'une liste.
Remarque : vous n'avez pas à libérer la mémoire occupées par les chaînes de caractères `key` et `value`. Nous considérons ici que cette mémoire est allouée statiquement (en utilisant des guillemets) et non pas dynamiquement (en utilisant `malloc`).
Question 11 : La fonction `hashtableRehash` a le prototype suivant : `hashtable * hashtableRehash(hashtable * h, int newN);`. Elle crée une nouvelle table de hachage de taille `newN`, y stocke toutes les valeurs de l'ancienne table de hachage `h`, efface `h` puis retourne la nouvelle table de hachage. Programmez la fonction `hashtableRehash`.
Remarque : Vous noterez que la complexité de `hashtableRehash` est $\mathcal{O}(m)$, où m désigne le nombre total d'éléments de la table de hachage.

3 Tests

- Vérifiez que le hash de la chaîne de caractères `"Alice"` avec n=10 donne 478%10 = 8.
- Vérifiez que le hash de la chaîne de caractères `"Bob"` avec n=10 donne 275%10 = 5.
- Vérifiez le hash des chaînes de caractères suivantes, pour n=10 : `"Charles"` donne 6 et `"David"` donne 8.
- Ajoutez (`"Alice"`, `"Dupont"`), (`"Bob"`, `"Dupond"`) et (`"Charles"`, `"Dupon"`) dans la table de hachage. Affichez la et vérifiez qu'il n'y a pas de collision.
- Ajoutez (`"David"`, `"Duppont"`) dans la table de hachage. Affichez la et vérifiez qu'il y a une collision.