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 $i \in [0; n-1]$ 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 $\mathcal{O}(M)$ comparaisons 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 2 - récursivité

L'objectif de ce TD est d'apprendre à proposer naturellement des algorithmes récursifs, mais aussi de savoir les analyser. Vous écrirez vos algorithmes en pseudocode.

1 Fonctions mathématiques

Le but de cet exercice est de proposer des algorithmes récursifs pour implémenter des fonctions mathématiques célèbres.

Question 1 : Écrire l'algorithme d'Euclide `PGCD` qui prend en entrée deux entiers positifs ou nuls a et b et renvoie leur PGCD.

On rappelle l'algorithme d'Euclide pour calculer le PGCD de deux nombres a et b où $a \geq b$: si $b = 0$ alors $\text{PGCD}(a, 0) = a$ sinon $\text{PGCD}(a, b) = \text{PGCD}(b, a\%b)$.

Question 2 : Ecrire un algorithme `PUISS` qui prend en entrée deux entiers strictement positifs N et p et qui détermine si N peut s'écrire comme une puissance entière de p, c'est-à-dire sous la forme $N = p^c$ avec c un entier positif ou nul. Cet algorithme renverra un booléen.

Question 3 : Ecrire un algorithme `BIN` qui prend en entrée un entier n et renvoie une chaine de caractères décrivant l'écriture binaire de n. L'écriture binaire de 0 sera la chaine vide `""`, l'écriture binaire de 1 sera `"1"`, l'écriture binaire de 13 sera `"1101"`, etc. On notera `"+"` l'opération de concaténation sur les chaines de caractères.

Question 4 : Ecrire un algorithme `BINOM` qui prend en entrée deux entiers positifs n et $p \leq n$. Il renvoie le coefficient binomial $\binom{n}{p} = \frac{n!}{p!(n-p)!}$. L'idée ici est de se servir de la formule de Pascal $\binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1}$.

2 Calcul de suites : questionner la récursivité

La suite de Fibonacci est définie comme ceci :

$$f_n = \begin{cases} 0 \text{ si } n = 0 \\ 1 \text{ si } n = 1 \\ f_{n-1} + f_{n-2} \text{ sinon} \end{cases}$$

Question 5 : En se basant sur la définition ci-dessus, écrire un algorithme récursif `FIBO_NAIF` qui prend en argument un entier positif ou nul n et qui renvoie la valeur de f_n.

`FIBO_NAIF`(2) effectue deux appels récursifs (l'un pour calculer `FIBO_NAIF`(1) et l'autre pour calculer `FIBO_NAIF`(0)), qui eux-mêmes n'utilisent pas d'appels récursifs (leur valeur est calculée directement).

Question 6 : Déterminez combien d'appels récursifs sont effectués lorsqu'on lance `FIBO_NAIF`(3), `FIBO_NAIF`(4) et `FIBO_NAIF`(5). Faites apparaître ces appels récursifs sur un schéma.

Question 7 : Soit A_n le nombre d'appels récursifs effectués lorsqu'on lance `FIBO_NAIF`(n). On a $A_0 = 0$ et $A_1 = 0$. Proposez une formule récursive pour déterminer A_n lorsque $n \geq 2$. Vérifiez les résultats obtenus à la question précédente pour A_2, A_3, A_4 et A_5.

Question 8 : En déduire que $A_n \geq 2^{\frac{n}{2}}$ pour tout $n \geq 2$, en utilisant une preuve par récurrence. Diriez-vous que `FIBO_NAIF` est un algorithme adapté pour calculer les 200 premières valeurs de la suite de Fibonacci ?
Question 9 : On va essayer maintenant de se ramener à un nombre d'appels récursifs *linéaire*, c'est-à-dire tel que $A_n = O(n)$. Ecrire un algorithme récursif `FIBO_MEM` qui prend en argument un entier $n \geq 1$ et un tableau à 2 cases `tab`. La fonction écrit f_{n-1} dans la case d'indice 0, et f_n dans la case d'indice 1, en s'appuyant sur les valeurs du tableau et sur des appels récursifs. Combien `FIBO_MEM`$(n, \texttt{tab})$ effectue-t'elle d'appels récursifs en fonction de n ? Écrire maintenant l'algorithme `FIBO` qui prend en argument un entier n et qui renvoie la valeur de f_n grâce à un appel à `FIBO_MEM`.

3 Premiers pas en recherche opérationnelle (difficile)

Dans cet exercice, nous allons manipuler des listes. On s'autorisera l'utilisation de certaines fonctions, classiques sur les listes, pour rendre nos pseudocodes plus clairs. Etant donné une liste L :

- On notera $L[i]$ l'accès à l'élément i de la liste,
- La fonction *add* prendra en argument une liste et un élément et ajoutera l'élément en fin de liste. Ex. : add$(L, 3)$.
- La fonction *len* renverra la longueur d'une liste. Ex. : len(L).
- La fonction *copy* prendra en argument une liste et renverra une copie. Ex. : copy(L).
- La fonction *sum* prendra en argument une liste et renverra la somme de ses éléments. Ex. : sum(L).
- La fonction *tail* prendra en argument une liste L et renverra une liste L' qui correspond à L sans son premier élément. Ex. : Si $L = [2, 7, 3, 5]$, tail(L) renvoie $[7, 3, 5]$.

Une société de transport par conteneurs cherche à optimiser ses rendements en trouvant à chaque fois la meilleure manière de remplir ses conteneurs. Dans un premier temps, on s'intéresse au problème suivant : on dispose de marchandises, caractérisées par la liste de leur poids respectif, et d'un conteneur qui peut contenir des marchandises jusqu'à un certain poids $P \geq 1$. L'entreprise veut savoir si, en choisissant certaines marchandises, elle peut remplir le conteneur avec le poids maximum P, c'est-à-dire tel qu'il n'y a plus de volume disponible dans le conteneur.

Par exemple, si les poids de nos marchandises sont [2,9,4,6], alors le conteneur de capacité $P = 17$ peut être complètement rempli avec les marchandises 2, 9 et 6. En revanche, il n'est pas possible de remplir complètement le conteneur de capacité $P = 18$.
Question 10 : Proposer un algorithme récursif `PLEIN` qui prend en argument une liste d'entiers `liste` (contenant les poids des marchandises) et une capacité P, et qui renvoie soit une liste contenant un remplissage total (si c'est possible), soit `NULL`. La fonction considérera les marchandises une par une, et essaiera pour chacune de la placer dans le conteneur, puis de ne pas la placer dans le conteneur.
Question 11 : (difficile) On considère les opérations élémentaires suivantes : ajout d'élément dans une liste (utilisation de *add*), calcul de la queue de la liste (utilisation de *tail*) et comparaisons booléennes.

- Donnez un O du nombre d'opérations élémentaires effectuées lorsque `liste` est de longueur n (borne **supérieure**). On vous conseille d'exprimer ce nombre d'opérations C_n en fonction de la valeur de C_{n-1}, puis utiliser les propriétés des suites arithmético-géométrique.
- Identifiez un type d'instances sur lesquelles la complexité sera exponentielle (borne **inférieure**).

La société s'intéresse aussi à un second problème : elle dispose d'une liste de marchandises à transporter, ainsi que de k conteneurs de capacité P. La question est de savoir s'il existe un remplissage des conteneurs permettant de transporter toutes les marchandises de la liste.

Par exemple, pour $k = 2$ et $P = 3$, la liste [2,1,3] peut être rangée dans les conteneurs : le premier conteneur contient 1 et 2, le deuxième contient 3. En revanche, la liste [2,1,4] ne peut être rangée car $4 > P$. Enfin, [2,2,2] ne peut pas être rangée non plus car chaque conteneur ne pourra contenir qu'une seule marchandise. Ci-dessous, un autre exemple dans la Figure 2.1 avec la liste [5,4,1,2,3,2] pour $k = 3$ et $P = 6$.
Question 12 : (Difficile) Proposer un algorithme `RANGE` qui prend en argument une liste d'entiers `liste` (contenant les poids des marchandises), un nombre de conteneurs k et une capacité P, et qui renvoie TRUE si un rangement correct existe, FALSE sinon.

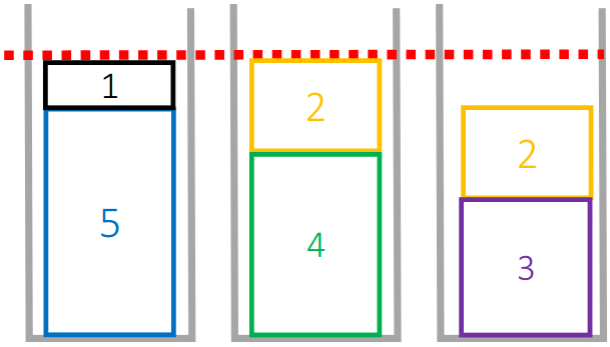


FIGURE 2.1 – La preuve que `RANGE`$([5, 4, 1, 2, 3, 2], 3, 6)$ doit renvoyer TRUE.

Vous procèderez en écrivant d'abord un algorithme intermédiaire récursif `RANGE_REC` qui prend en argument `liste`, un remplissage partiel des boites `rempl` (qui est une liste contenant k listes d'entiers) et une capacité P et qui renvoie TRUE si le remplissage partiel peut être complété en un remplissage correct avec les marchandises de `liste`, FALSE sinon.

Structures de données (ZZ1, S2)

TD 3 - arbres et parcours

L'objectif de ce TD est d'apprendre à analyser la structure d'arbre, et en particulier à connaître les différents types de parcours d'arbres.

1 Parcours en largeur et en profondeur

On va utiliser un pseudocode proche du C afin de proposer des algorithmes sur les arbres **binaires**. Dans cet exercice, nos arbres vont contenir une racine et chaque nœud sera associé à une valeur entière. On va implémenter les arbres de manière récursive : ainsi, chaque arbre pourra en pratique être confondu avec sa racine.

- On définit un arbre/nœud A avec l'instruction newArbre(), qui pourra prendre en argument un entier si l'on souhaite définir directement la valeur de la racine.
- Étant donné un arbre A, on notera $A\rightarrow$gauche le sous-arbre enraciné au fils gauche et $A\rightarrow$droite le sous-arbre enraciné au fils droit. Par exemple, si un nœud A n'a qu'un seul fils et que c'est le fils droit, on aura $A\rightarrow$gauche==`NULL`.
- Enfin, pour accéder à la valeur d'un nœud, on utilisera $A\rightarrow$val.

Question 1 : Dessinez l'arbre A qui est défini par le pseudocode ci-dessous, puis donnez (i) l'ordre dans lequel les valeurs des nœuds sont lues par un parcours en profondeur et (ii) l'ordre dans lequel les valeurs des nœuds sont lues par un parcours en largeur.

A = newArbre(5)
B = newArbre(7)
$A\rightarrow$gauche = B
C = newArbre()
$B\rightarrow$val = 12
$C\rightarrow$val = -2
$A\rightarrow$droite = C
D = newArbre(3)
$B\rightarrow$gauche = D
E = newArbre(-5)
$B\rightarrow$droite = E

Question 2 : Proposez un algorithme **récursif** `Parcours_Prof` qui prend en argument un arbre A et qui affiche les valeurs des nœuds en suivant un parcours en profondeur. Vous utiliserez l'instruction print() pour l'affichage.

Question 3 : On considère les opérations élémentaires suivantes : accès à valeur/fils gauche/fils droit d'un nœud. Etant donné un arbre A contenant n nœuds, déterminez le nombre C_n d'opérations élémentaires effectuées par l'algorithme `Parcours_Prof`. Vous procèderez par récurrence. A-t'on affaire à une complexité logarithmique/linéaire/quadratique/exponentielle ?

Un *arbre binaire complet* (ABC en abrégé) est un arbre binaire dans lequel toutes les hauteurs sont remplies, à l'exception éventuelle de la dernière, dans laquelle les feuilles sont alignées à gauche.

Question 4 : Donnez l'ABC contenant 10 nœuds. Numérotez les nœuds de telle manière à ce que le parcours en profondeur lise les valeurs dans l'ordre suivant : $1, 2, 3, \ldots, 9, 10$.

Question 5 : Pour rappel, la *hauteur* d'un arbre correspond à la longueur du chemin entre la racine et la plus éloignée des feuilles. Donnez la hauteur H_n d'un ABC contenant n nœuds.

Question 6 : (*Difficile*) Proposez un algorithme `Arbre_BC` qui prend en argument un entier positif n et qui renvoie un ABC à n nœuds dont le parcours en profondeur donne $1, 2, 3, \ldots, n-1, n$.

2 Parcours préfixe et postfixe

Dans cet exercice, les arbres représentent des calculs. Chaque nœud de l'arbre a pour valeur soit une lettre minuscule, soit un symbole représentant une opération arithmétique simple.

Question 7 : Regardons la notation suivante : $- \times + a\ b - c\ d + e\ f$. S'agit-il d'une notation préfixe ou infixe ? Dessinez l'arbre binaire admettant cette représentation.

Question 8 : Quelle est la notation postfixe de l'arbre binaire obtenu à la question précédente ?

Question 9 : Si on applique l'algorithme `Parcours_Prof` de la question 2, quelle notation renvoie-t'il ?

Question 10 : Proposez un algorithme **récursif** `Postfixe` qui prend en entrée un arbre et renvoie une chaine de caractères contenant la notation postfixe de l'arbre. On utilisera le même pseudocode que celui de l'exercice précédent.

3 Codage de Prüfer (difficile)

Dans ce problème, une entreprise cherche à déployer un réseau électrique à travers $n \geq 1$ villages numérotés $1, \ldots, n$. Pour chaque réseau possible, l'entreprise voudrait vérifier plusieurs paramètres comme la fiabilité du réseau et sa faisabilité. Un réseau sera ici représenté par un arbre reliant tous les villages, qui sont donc les nœuds de l'arbre. Chaque arête de l'arbre correspond à une ligne reliant deux villages. L'objectif de l'entreprise est de savoir combien de réseaux possibles existent pour n villages.

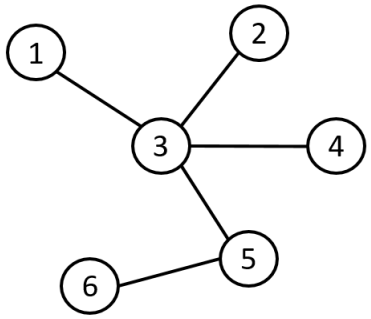
Question 11 : Combien de réseaux sont possibles pour $n = 3$ villages ? Représentez les.

Question 12 : Montrez qu'un arbre à n nœuds possède exactement $n - 1$ arêtes.

On s'intéresse désormais à un algorithme sur les arbres : c'est le *codage de Prüfer*. Etant donné un arbre A, il renvoie une séquence de nombres $L(A)$. Cet algorithme est décrit brièvement ci-dessous, avec un pseudocode rudimentaire.

Function : `Codage`(Arbre A)
T = copie de A
L = [ ] (liste vide)
while il reste au moins 3 sommets dans T **do**
  v = feuille de T avec le numéro minimum
  Ajouter à L le numéro du voisin de v
  Enlever v et son arête adjacente de T
end while
return L

Question 13 : Etant donné l'arbre représenté ci-dessous, donnez son codage de Prüfer. Pour un arbre A à n nœuds, quelle est la taille p_n de la séquence $L(A)$?



Question 14 : Quel arbre A correspond à la séquence $L(A) = (6, 6, 6, 6)$? Quel arbre B correspond à la séquence $L(B) = (1, 2, 3, 4)$?

Pour un nombre n de villages/nœuds, un codage de Prüfer est donc une séquence $(x_1, x_2, \ldots, x_{p_n})$ où $x_i \in \{1, \ldots, n\}$ pour tout $1 \leq i \leq p_n$. Il est connu que la transformation arbre $\rightarrow$ codage de Prüfer est **bijective** : pour chaque arbre/réseau, il existe un unique codage de Prüfer correspondant et vice versa. Lors de l'implémentation informatique, l'entreprise a choisi de représenter ses réseaux par leur codage de Prüfer correspondant, pour avoir une structure de donnée plus compacte à manipuler.

Question 15 : Déterminez le nombre de réseaux possibles pour n villages en vous appuyant sur ce codage (en faisant l'hypothèse que toute séquence de Prüfer correspond à un arbre valide, ce qui est vrai).
Question 16 : (Difficile) Etant donné un codage de Prüfer $(x_1, x_2, \ldots, x_{p_n})$ et un village i, l'entreprise cherche à retrouver directement le nombre $N(i)$ de villages voisins à i dans le réseau, sans avoir à reconstruire tout l'arbre. Déterminez $N(i)$.
Question 17 : (*Difficile*) En vous appuyant sur l'observation faîte à la question 12, proposez un algorithme `Decodage` qui prend en argument un code de Prüfer et construit l'arbre correspondant. Vous pouvez utiliser un pseudocode similaire à celui présenté pour l'algorithme `Codage`.

Structures de données (ZZ1, S2)

TD 4 - arbres AVL

L'objectif de ce TD est de comprendre plus en détail, le fonctionnement, la correction et la complexité des opérations de base sur les ABR et les AVL.

1 Arbre binaire de recherche

On va réutiliser le pseudo-code introduit par le TD 3 sur les arbres binaires au cours de ce TD.

1.1 Suppression dans un ABR

Question 1 : Dessiner des arbres binaires de recherche de hauteur 2, 3, 4, 5 et 6 pour le même ensemble de valeurs $\{1, 4, 5, 10, 16, 17, 21\}$.
Question 2 : Écrire un algorithme de recherche de la valeur maximale dans un ABR non vide.
Question 3 : Quelle est la complexité dans le pire cas de cet algorithme ? On comptera, à une constante près, le nombre d'accès aux informations d'un nœud (valeur et fils) en fonction de la hauteur de l'ABR.
Question 4 : Étendre l'algorithme que vous venez d'écrire pour faire une fonction `ABR-MAX-POP` d'extraction (qui supprime et retourne) de la valeur maximale d'un ABR.
Indice : On supposera ici que l'on peut modifier l'arbre passé en entrée.
Question 5 : En réutilisant la fonction `ABR-MAX-POP`, écrire une procédure de suppression d'une valeur dans un ABR.
Question 6 : Quelle est la complexité dans le pire cas de l'algorithme de suppression dans un ABR ? On comptera à une constante près, le nombre d'accès aux informations d'un nœud (valeur et fils) en fonction de la hauteur de l'ABR.

1.2 Tri à l'aide d'un ABR

Question 7 : Effectuer le parcours infixe sur les ABR vus à la question 1.
Question 8 : En réutilisant les fonctions vues dans le cours sur les arbres et ABR, proposer une procédure qui prend un tableau d'entiers et affiche ces entiers par ordre croissant.
Question 9 : Quelle est la complexité dans le pire cas de cet algorithme ? On comptera à une constante près, le nombre d'accès aux informations d'un nœud (valeur et fils) en fonction de la taille du tableau.
Question 10 : Pour tout $n \in \mathbb{N}$, trouver un tableau T_n pour lequel la complexité dans le pire cas est atteinte.
Question 11 : (Difficile) La fonction `ABR-COMPLET` ci-dessous prend en entrée une hauteur h et retourne l'ABR complet de hauteur h qui contient les valeurs de 1 à $2^{h+1} - 1$. En utilisant cette fonction, définir un tableau T_n pour lequel la complexité du tri en $O(n \log(n))$ pour tout $n \in \mathbb{N}$.

Function : `ABR-COMPLET`(h)
return ABR-COMPLET-AUX$(2^h, h)$

Function : `ABR-COMPLET-AUX`(p, h)

A = new Arbre(p)
if h != 0 **then**
  A→gauche = ABR-COMPLET-AUX($p - 2^{h-1}, h-1$)
  A→droite = ABR-COMPLET-AUX($p + 2^{h-1}, h-1$)
end if
return A

2 AVL

2.1 Complexité

Les AVL présentent une bonne propriété concernant leur hauteur : la hauteur d'un AVL contenant n nœuds est toujours de l'ordre de $\log_2(n)$.
Question 12 : (Difficile) Montrer que tout AVL possédant n nœuds a une hauteur h vérifiant :

$$\log_2(n+1) \leq h+1 \leq 1.5 \log_2(n+2)$$

On utilisera les informations suivantes :

1. La suite de Fibonacci définie par $k \geq 2, F(k) = F(k-1) + F(k-2)$ avec $F(0) = 2$ et $F(1) = 3$ est telle que pour tout $k \in \mathbb{N}$, $F(k) > \frac{1}{\sqrt{5}}(\phi^{k+3} - 1)$ avec $\phi = \frac{1+\sqrt{5}}{2}$.
2. $\phi^2 > \sqrt{5}$
3. $1.5 > \frac{1}{\log_2(\phi)}$

Question 13 : En déduire la complexité dans le pire des cas des algorithmes d'insertion et de suppression d'une valeur dans un AVL. On comptera à une constante près, le nombre d'accès aux informations d'un nœud (valeur et fils) en fonction du nombre de valeur dans l'AVL.

2.2 Insertion et rotation

Question 14 : Construire l'AVL obtenu par l'insertion successive de la suite de valeurs 12, 3, 2, 5, 4, 7, 9, 11, 14 et 10. Faites apparaître toutes les rotations effectuées.
Question 15 : Dans un AVL, après l'insertion d'une valeur et avant qu'une possible rotation ne soit effectuée, montrer que pour chaque nœud n, $|\text{balance}(n)| \leq 2$.
Question 16 : En faisant un schéma, observer que lorsqu'une rotation droite est effectuée après une insertion au niveau d'un nœud b ayant pour fils gauche a :

— la hauteur de l'arbre enraciné en a résultant de la rotation est égale à la hauteur de l'arbre enraciné en b avant l'insertion.
— $|\text{balance}(n)| \leq 1$ pour tout nœud n dans l'AVL après rotation (donc aucune autre rotation n'est nécessaire).

Question 17 : Faire les mêmes observations lorsqu'une rotation gauche-droite est effectuée.

Structures de données (ZZ1, S2)

TD 6 - B-arbres

Nous nous intéressons à l'arbre de la figure 5.1.

1 Recherche dans un B-arbre

Question 1 : Dans ce B-arbre, combien chaque feuille peut-elle stocker de valeurs ? Combien chaque noeud peut avoir de fils ? Quel est l'ordre du B-arbre ?
Question 2 : Détaillez la séquence d'actions pour rechercher si les éléments suivants sont présents dans l'arbre, en effectuant une recherche linéaire sur les clés d'un nœud :

- l'élément 70 (présent),
- l'élément 67 (présent),
- l'élément 60 (absent).

2 Insertions

Question 3 : Insérez successivement les éléments suivants à l'arbre de la figure 5.1.

- Insertion de 43. Indication : cette insertion est simple.
- Insertion de 44 (après avoir inséré 43). Indication : il va falloir couper une feuille de 4 éléments en plusieurs morceaux : une feuille de 1 élément, un élément isolé qui remonte au père, et une feuille de 2 éléments.

3 Suppression

Question 4 : Supprimez la valeur 7 de l'arbre de la figure 5.1.
Question 5 : Supprimez la valeur 50 de l'arbre de la figure 5.1.
Question 6 : Supprimez les valeurs 56, 67 puis 50 de l'arbre de la figure 5.1.

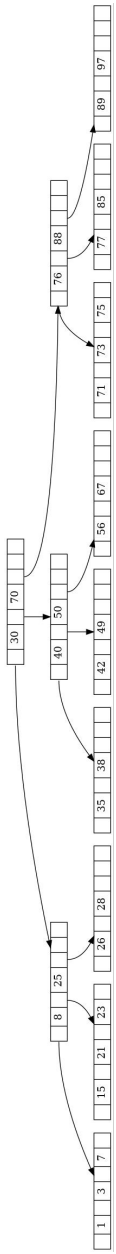


FIGURE 5.1 – Exemple d'un b-arbre (tiré de `http://www.btechsmartclass.com/data_structures/b-trees.html`).

Structures de données (ZZ1, S2)

TD 7 - preuves de programme

1 Preuve du tri par bulles

Considérons l'algorithme du tri par bulles décrit dans l'algorithme 1.

Algorithm 1 Algorithme du tri par bulles.

```
for all i ∈ [1; n − 1] do
  for all j ∈ [1; n − i] do
    if T[j] > T[j + 1] then
      inverser T[j] et T[j + 1]
    end if
  end for
end for
```

Question 1 : Prouvez que l'algorithme termine.
Question 2 : Considérons la propriété suivante : $P(j)$ est vraie si la plus grande valeur de $(T[1], \ldots, T[j+1])$ se trouve en $T[j+1]$. Montrez que cette propriété est un invariant de la boucle interne.
Question 3 : Prouvez que l'algorithme trie bien le tableau. Vous pourrez démontrer que la propriété $Q(i)$ suivante est un invariant de la boucle externe : $Q(i)$ est vraie si les valeurs $(T[n-i+1], \ldots, T[n])$ sont triées par ordre croissant, et aucune valeur de $(T[1], \ldots, T[n-i])$ n'est plus grande que $T[n-i+1]$ (quand cette valeur existe).

2 Preuve de la recherche par dichotomie

Considérons l'algorithme de recherche par dichotomie décrit dans l'algorithme 2.
Question 4 : Prouvez que l'algorithme termine. Vous pourrez montrer qu'on élimine au moins une case à chaque étape.
Question 5 : Prouvez que l'algorithme ne terminerait pas (dans certains cas) si on faisait $j \leftarrow half$ et $i \leftarrow half$. Vous pouvez utiliser un contre-exemple.
Question 6 : Prouvez que l'invariant suivant est vrai : "Si le tableau est trié, alors la valeur recherchée ne peut être qu'entre les indices i (inclus) et j (inclus)".

3 Comparaison de deux chaînes de caractères

Considérons l'algorithme de comparaison de deux chaînes de caractères décrit dans l'algorithme 3.
Question 7 : La première boucle n'a pas de condition sur $s2[i]! = 0$, bien qu'il soit possible à la chaîne $s2$ de s'arrêter avant la chaîne $s1$. Est-ce une erreur ?
Question 8 : Montrez que l'algorithme termine.

Algorithm 2 Algorithme de la recherche dichotomique dans un tableau trié.

```
i ← 1
j ← n
half ← (i + j)/2
while i ≤ j do
  if T[half] = value then
    retourner vrai
  else
    if T[half] < value then
      j ← half − 1
      half ← (i + j)/2
    else
      i ← half + 1
      half ← (i + j)/2
    end if
  end if
end while
retourner faux
```

Algorithm 3 Algorithme de comparaison de deux chaînes de caractères

```
i ← 0
while s1[i] = s2[i] et s1[i]! = 0 do
  i ← i + 1
end while
if s1[i] = s2[i] then
  retourner 0
else
  if s1[i] < s2[i] then
    retourner -1
  else
    retourner 1
  end if
end if
```

Question 9 : Quel est l'invariant de la boucle ?

4 Recherche d'une sous-chaîne dans une chaîne

Considérons l'algorithme de recherche d'une sous-chaîne *substr* dans une chaîne de caractères *str*, décrit dans l'algorithme 4.

Algorithm 4 Algorithme de recherche d'une sous-chaîne *substr* dans une chaîne de caractères *str*

```
for all i ∈ [0; n − length(substr) − 1] do
  if la chaîne str + i démarre par substr then
    retourner vrai
  end if
end for
retourner faux
```

Question 10 : Quel est l'invariant de la boucle ?

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 d'un 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 dessinerez le tableau à chaque tour de boucle.
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 dessinerez le tableau à chaque tour de boucle.
Question 6 : Quelle 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 : Écrivez 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 2.3 et 3.3,

— des fichiers à compléter sont téléchargeables via l'outil GIT.

1 Récupération des fichiers à compléter via l'outil GIT

Dans cette partie, vous allez apprendre la procédure pour télécharger l'ensemble des fichiers à compléter, pour tous les TP, via l'outil de suivi de versions nommé GIT. Notez que le téléchargement d'un ensemble de fichiers n'est qu'une petite fonctionnalité de GIT, mais cela sera la seule que nous utiliserons dans cette matière. Pour information, dans les TP suivants, vous n'aurez plus à utiliser cet outil.

Question 1 : Pour télécharger les fichiers de tous les TP de cette matière, réalisez les opérations suivantes :

— Lancez un terminal.

— Avec la commande `cd`, allez dans le répertoire dans lequel vous voulez sauvegarder le répertoire correspondant à la matière SDD-S2 (il peut s'agir de votre répertoire racine, d'un répertoire `zz1`, d'un répertoire `tp`, etc.).

— Récupérez tous les fichiers à partir du dépôt GIT. Cela se fait en exécutant la commande suivante : `git clone git@gitlab.isima.fr:alguitto/zz1-sdd-s2.git`

— Renommez le répertoire `zz1-sdd-s2` avec le nom que vous souhaitez (si vous voulez que ce répertoire s'appelle simplement `sdd`, utilisez la commande `mv zz1-sdd-s2 sdd`).

— Allez dans ce répertoire avec la commande `cd`.

Remarque : vous pouvez aussi télécharger les fichiers en vous connectant depuis un navigateur sur `https://gitlab.isima.fr/alguitto/zz1-sdd-s2`, en cliquant sur "Code", puis sur "Download source code", puis sur "zip". Il ne vous reste plus qu'à dézipper les fichiers dans le répertoire souhaité.

A présent, vous disposez d'un répertoire qui contient un sous-répertoire par TP. Vous disposez aussi d'un script de correction automatique nommé `correction-automatique.pl` qui exécute un certain nombre de tests, pour chacun des TP, et affiche un pourcentage de réussite par TP. Notez que ces tests automatiques ne sont pas exhaustifs, mais vous donnent néanmoins une indication sur votre réussite aux TP.

Question 2 : Déplacez-vous dans le sous-répertoire `tp1`, correspondant au premier TP.

2 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;
```

2.1 Opérations de base

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

Question 3 : La fonction `stackCreate` a le prototype suivant : `stack * stackCreate();`. Elle crée une pile vide. Pour cela, il suffit de retourner un pointeur dont la valeur est `NULL` (cette valeur est définie notamment dans `<stdlib.h>`), sans faire aucune allocation. Programmez la fonction `stackCreate`.

Question 4 : 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 5 : 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 6 : 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 ; si ce n'est pas le cas, vous pourrez retourner n'importe quelle valeur, par exemple -1). Programmez la fonction `stackTop`.

Question 7 : 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>`.

2.2 Opérations utiles

Pour toutes ces opérations, vous aurez une boucle à faire.

Question 8 : 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 9 : La fonction `stackSize` a le prototype suivant : `int stackSize(stack * s);`. Elle retourne le nombre d'éléments de la pile. Programmez la fonction `stackSize`.

Question 10 : La fonction `stackFree` a le prototype suivant : `void stackFree(stack * s);`. Elle supprime tous les éléments de la pile. Programmez la fonction `stackFree`.

2.3 Exemple de 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 (qui est en haut de la pile), puis 5 (qui est en bas de la pile).
- 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.

3 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` et `queueRemove`. Cela peut se programmer avec une liste doublement chaînée, en ajoutant en tête et en retirant en fin, et en disposant d'une structure avec un pointeur de début et un pointeur de fin.

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

3.1 Opérations de base

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

Question 11 : 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 12 : 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 13 : 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 8.1, 8.2 et 8.3.

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.

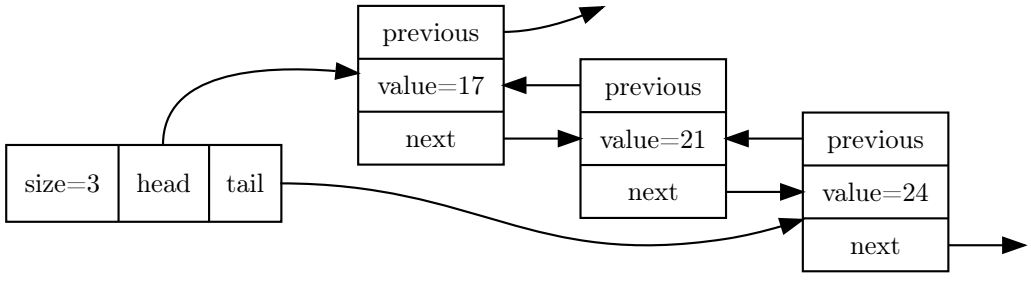


FIGURE 8.1 – Exemple de file avant l'ajout.

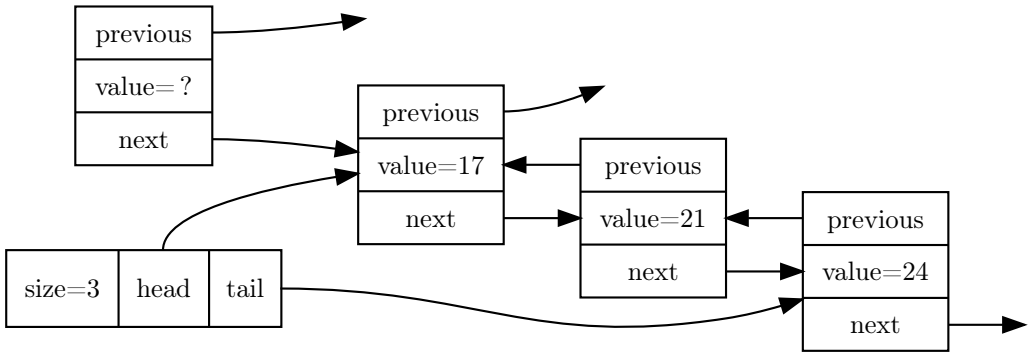


FIGURE 8.2 – Exemple de file pendant l'ajout : le nouvel élément est créé et ses pointeurs sont corrects, mais il reste des choses à modifier dans la file initiale.

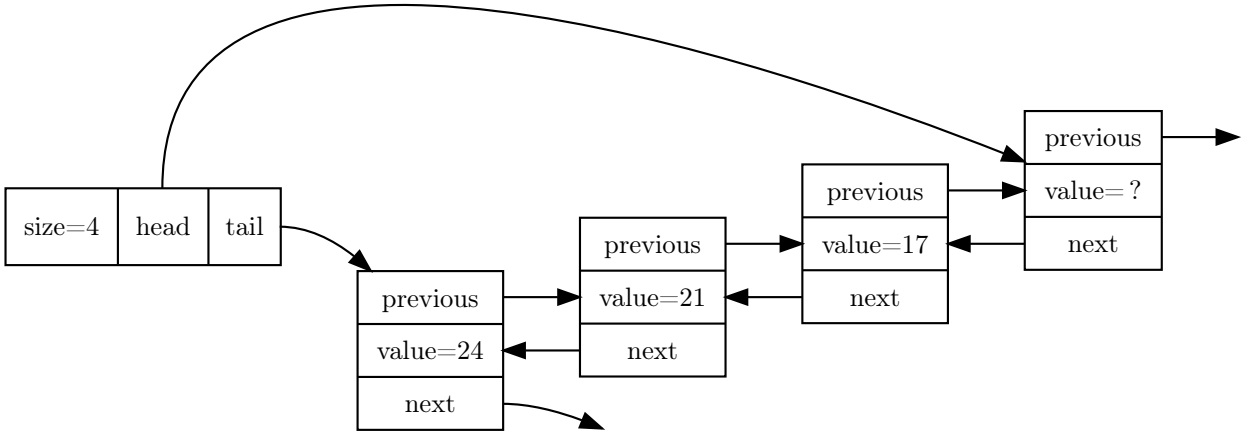


FIGURE 8.3 – Exemple de file après l'ajout.

Question 14 : 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`.
Remarque : vous ne devez pas parcourir la file depuis le début, il faut aller directement à la fin.
Question 15 : 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 16 : 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`.

3.2 Opérations utiles

Question 17 : 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 18 : La fonction `queueDisplay` a le prototype suivant : `void queueDisplay(queue * q);`. Elle affiche tous les éléments de la file, dans l'ordre inverse (donc du dernier élément au premier). Programmez la fonction `queueDisplay`.

3.3 Exemple de 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 chaque 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, et non pas une fois que vous avez tout programmé. Des tests, à titre d'exemple, se trouvent dans la partie 3.

Question 1 : Rendez-vous dans le répertoire `tp2`, que vous avez dû télécharger avec l'outil GIT lors du `tp1`.

1 Fonctions minimales à implémenter

Question 2 : La fonction `listCreate` a le prototype suivant : `list * listCreate();`. Elle crée une liste vide. Programmez la fonction `listCreate`.

Question 3 : La fonction `hashtableCreate` a le prototype suivant : `hashtable * hashtableCreate(int n);`. Elle crée une table de hachage avec un tableau de n indices. Programmez la fonction `hashtableCreate`.

Question 4 : 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 5 : La fonction `listDisplay` a le prototype suivant : `void listDisplay(list * l);`. Elle affiche la liste `l`. Programmez la fonction `listDisplay`.

Question 6 : 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 7 : 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`. Cette solution, plus simple à programmer, n'est généralement pas recommandée.
Question 8 : 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 9 : 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 10 : 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, vérifiez que la clé n'est pas déjà présente dans la table de hachage. Programmez la fonction `hashtableAdd`.

2 Fonctions bonus

Question 11 : 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 12 : 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 (c'est-à-dire deux éléments qui ont le même hash, et qui apparaissent donc dans une même liste).
- Ajoutez (`"David"`, `"Duppont"`) dans la table de hachage. Affichez la et vérifiez qu'il y a une collision.

Structures de données (ZZ1, S2)

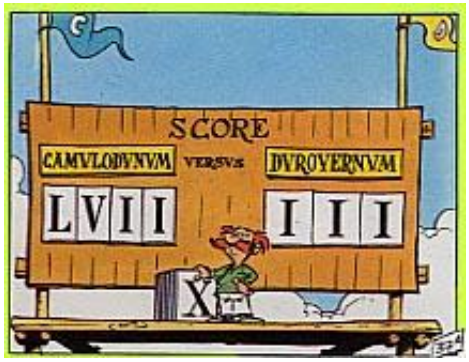
TP 3 - récursivité

L'objectif de ce TP est de d'utiliser la récursivité dans diverses situations. Les fonctions à écrire doivent donc dans la plupart des cas être récursives.

Ainsi, les exercices devront être réalisés **sans utiliser de boucle** (sauf exception).

Toutes les fonctions que vous réaliserez dans ce TP pourront être testées en utilisant le fichier `main.c` fourni. Vous commenterez/décommenterez les appels de fonction dans le main de ce fichier selon ce que vous souhaitez tester.

1 Première fonction récursive



L'objectif de cette partie est d'écrire une fonction récursive permettant d'évaluer un nombre écrit en chiffres romains.
Rappelons que :

- M = 1000
- D = 500
- C = 100
- L = 50
- X = 10
- V = 5
- I = 1

Comme exemple, évaluons le nombre MCMXCIX :

- On est sur le premier M.
  Son successeur est C, C est plus petit que M, donc notre résultat final sera la valeur de M (1000) plus la valeur du reste (CMXCIX).
- La valeur du reste CMXCIX est la suivante :
  C est plus petit que M (une soustraction aura lieu) donc la valeur de CMXCIX est égale à la valeur de MXCIX moins la valeur de C
- La valeur de MXCIX est la suivante :
  M est plus grand que X donc on a 1000+valeur(XCIX).
- La valeur de XCIX est égale à la valeur de CIX moins la valeur de X (car le premier X est plus petit que son successeur).
- La valeur de CIX est égale à 100 + valeur(IX) car C est plus grand que I.
- La valeur de IX est égale à la valeur de X moins la valeur de I, soit 10-1 = 9.

On a donc : CIX = C + 9 = 109
XCIX = CIX - X = 109 - 10 = 99
MXCIX = M + XCIX = 1000 + 99 = 1099
CMXCIX = MXCIX - C = 1099 - 100 = 999
MCMXCIX = 1000 + 999 = 1999

Question 1 : Dans le fichier `romains.c`, vous devez compléter la fonction `nombreRomainToDecimal` qui calcule récursivement et retourne la valeur décimale d'un nombre écrit en chiffres romains. Le nombre en chiffres romains est donné sous forme de chaîne de caractères.

Vous pouvez utiliser la fonction `chiffreRomainToDecimal` qui vous est fournie et qui transforme un caractère (mais pas un nombre) correspondant à un chiffre romain en un entier correspondant à sa valeur décimale.

Les affichages au début et à la fin de la fonction `romains.c` vous permettront d'observer la suite des appels à cette fonction.

Pour tester votre fonction, ne créez pas de `main` dans `romains.c`, mais utilisez le fichier `main.c` (fonction `test_romains`).

Aide :
Calculer un nombre romain, c'est :
- calculer la valeur du premier chiffre (chiffre de gauche),
- calculer le nombre constitué des chiffres suivants
- réaliser une addition ou une soustraction.

Attention de ne pas oublier le cas de base correspondant à un nombre romain de longueur 0.

2 Récursivité et Listes

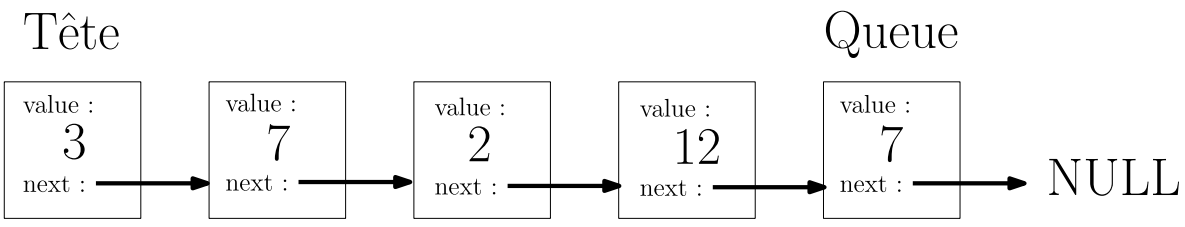
2.1 Module listes

Dans cette partie, nous allons programmer en langage C quelques fonctions sur les listes chaînées en utilisant la récursivité.

Il est recommandé de tester vos fonctions au fur et à mesure que vous les écrivez. Les tests sont à réaliser dans `main.c`.
Notez que pour chaque fonction `listFct` à réaliser dans `listes.c`, une fonction associée `test_listFct` permet de la tester dans `main.c`

Vous utiliserez la structure de données suivante qui est une liste d'entiers :

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



Les fichiers et `listes.c` vous sont fournis avec certaines fonctions de base déjà implémentées (`listCreate` et `listAdd`).

Vous répondrez aux questions suivantes en complétant `listes.c` et vous réaliserez au fur et à mesure les tests dans `main.c`.
La plupart des fonctions de test sont déjà écrites, certaines sont toutefois à compléter.

2.2 Fonctions à écrire

Les fonctions suivantes doivent toutes êtres récursives.
Question 2 : La fonction `listDisplay` a le prototype suivant `void listDisplay(list * )`.
Elle affiche toutes les valeurs de la liste, sur une même ligne, séparées par un espace, dans l'ordre où elles ont été ajoutées (donc en finissant par la tête).

Aide :
- **"Afficher toutes les valeurs"** de la liste, c'est **"afficher toutes les valeurs jusqu'à l'avant dernière"**, puis **"afficher la dernière valeur"** (tête de la liste) .
- Ne pas oublier **le cas de base**, qui survient lorsque le pointeur vers la liste à afficher est `NULL`. La queue de la liste pointant vers NULL, on finira forcément par arriver au cas de base.

Question 3 : La fonction `listInverseDisplay` a le même prototype que `listDisplay`.
Elle affiche la liste selon le même format, mais dans l'ordre inverse (en commençant par la tête).

Question 4 : La fonction `listSearch` a le prototype suivant `int listSearch(list * l, int n);`.
Elle recherche la valeur n dans la liste l et retourne 1 si la valeur n est dans la liste, 0 sinon.

Question 5 : La fonction `listMap` a le prototype suivant `list * listMap(list* , int (*f)(int));`
(le deuxième paramètre est un pointeur de fonction).
Si les valeurs de la listes sont $[n_1, n_2, n_3, \ldots, n_p]$, elle retournera une nouvelle liste dont les valeurs sont $[f(n_1), f(n_2), f(n_3), \ldots, f(n_p)]$.

Pour tester `listMap`, vous réaliserez un test dans `main.c` qui calcule et affiche une liste dont les valeurs sont les carrés de la liste_de_test. Il faudra pour cela avoir préalablement écrit une fonction `carre` puis compléter la fonction `test_listMap`.

Question 6 : La fonction `listFilter` a le prototype suivant `list * listFilter(list* li1 , int (*p)(int))`.
Elle retournera une nouvelle liste `li2` dont les valeurs sont les valeurs de la liste `li1` qui vérifient un prédicat `p` donné, implémenté dans la fonction `p`.
La fonction `p` est une fonction qui retourne 1 si l'entier donné en paramètre respecte le prédicat `p`, et retourne 0 sinon.
Pour tester `listFilter`, vous réaliserez un test dans `main.c` qui calcule et affiche une liste dont les valeurs sont les valeurs de liste_de_test qui sont strictement supérieures à 3. Vous compléterez pour cela la fonction `test_listFilter`. Il faudra préalablement avoir écrit une fonction `sup3`.

Question 7 : La fonction `listFold` a le prototype suivant `int listFold(list*  , int (*op)(int,int),int)`.
La fonction `listFold` sert à appliquer un traitement récursif aux éléments d'une liste.
Si les valeurs de la liste `l1` sont $[n_1, n_2, \ldots, n_p]$, `listfold` retournera l'entier op(n_1, op(n_2,..., op(n_p, base))).

Utiliser ensuite `listFold` dans l'implémentation les fonctions `listSum` , `listProd` et `listLen` dans le fichier `main.c`. Ces fonctions doivent calculer respectivement la somme, le produit des éléments de la liste, et la longueur de la liste (nombre de valeurs). Il faudra pour cela avoir écrit les fonctions `opSum`, `opProd` et `opLen` correspondantes.

Question 8 : La fonction `inverse` a le prototype suivant `list* listInverse(list * l1);`.
Elle retournera une liste `l_2` dont les valeurs sont celles de `l1`, mais dans l'ordre inverse.
Si les valeurs de la liste `l1` sont $[n_1, n_2, \ldots, n_p]$, celles de la liste `l2` seront $[n_p, \ldots, n_2, n_1]$.

Pour programmer la fonction `listInverse`, vous pouvez utiliser une fonction récursive intermédiaire `ajouteInverse` de prototype `list * ajouteInverse(list * l1,list * l2)` que vous aurez préalablement écrite et qui retournera la concaténation de `l1` et de l'inverse de `l2`.
Par exemple, avec les listes :

$$\texttt{l1} : (\text{Tête}) \rightarrow \boxed{7} \rightarrow \boxed{12} \rightarrow \boxed{2} \rightarrow \boxed{7} \rightarrow \boxed{3} \rightarrow \text{NULL}$$

et

$$\texttt{l2} : (\text{Tête}) \rightarrow \boxed{5} \rightarrow \boxed{6} \rightarrow \boxed{2} \rightarrow \text{NULL}$$

alors `ajouteInverse(l1,l2)` retournera la liste

$$(\text{Tête}) \rightarrow \boxed{2} \rightarrow \boxed{6} \rightarrow \boxed{5} \rightarrow \boxed{7} \rightarrow \boxed{12} \rightarrow \boxed{2} \rightarrow \boxed{7} \rightarrow \boxed{3} \rightarrow \text{NULL}$$

Dans `main.c`, la fonction `test_listInverse`permet de la tester.

Question 9 : La fonction `listFree` a le prototype suivant : `void listFree(list *)` .
Elle supprime tous les éléments de la liste.
Programmez la fonction `listFree` récursive.
Vous pourrez tester la bonne libération de la mémoire en utilisant `valgrind`.

3 Changement de base

Dans cette partie, nous utilisons le module `listes` réalisé dans le chapitre précédent pour représenter des nombres dans une base b comprise entre 2 et 16 (par exemple, si $b = 2$ on est en binaire, si $b = 8$ on est en octal, si $b = 10$ on est en décimal, et si $b = 16$ on est en hexadécimal).

La fonction `setBase` permet de définir la base (qui est stockée dans une variable globale `base`).

Modélisation :

— si $b = 2$, le nombre binaire `1101101` sera représenté par la liste chaînée
$$\text{NULL} \leftarrow \boxed{1} \leftarrow \boxed{1} \leftarrow \boxed{0} \leftarrow \boxed{1} \leftarrow \boxed{1} \leftarrow \boxed{0} \leftarrow \boxed{1} \ (\text{Tête}).$$
La tête de la liste est donc le bit de poids faible.

— si $b = 16$, le nombre hexadécimal `1AF5` sera représenté par la liste chaînée
$$\text{NULL} \leftarrow \boxed{1} \leftarrow \boxed{10} \leftarrow \boxed{15} \leftarrow \boxed{5} \ (\text{Tête}).$$
— Le principe est le même pour toutes les autres bases.

L'objectif de cette partie est de fournir un module générique de conversion, en utilisant toujours des fonctions récursives.

Les réponses aux questions sont à fournir dans le fichier `base.h`.

Les tests des fonctions à réaliser sont fournis dans le fichier `main.c`.

Question 10 : Compléter la fonction `printBaseB` qui affichera en base `b` un nombre représenté par une liste `l`.

Par exemple, si la base est 16 et si la liste chainée est NULL $\leftarrow \boxed{1} \leftarrow \boxed{10} \leftarrow \boxed{15} \leftarrow \boxed{5}$ alors la fonction `printBaseB` affichera `1AF5`.

Question 11 : Compléter la fonction `baseToDec` qui calcule la valeur décimale d'un nombre en base b représenté par une liste. Dans cette fonction, il est demandé de ne pas utiliser la fonction puissance.
On utilisera la méthode inspirée du calcul ci-dessous permettant de calculer la valeur décimale de $(1AF5)_{16}$:

$$(1AF5)_{16} = 5 + (16 \times (15 + 16 \times (10 + 16 \times 1))).$$

Vous pourrez au choix écrire une fonction récursive ou utiliser directement l'une des fonctions réalisées dans le module `liste`.

Question 12 : Compléter la fonction `decToBase` qui retourne une liste représentant un nombre en base b correspondant à un nombre décimal donné.

Dans cette fonction récursive, vous utiliserez la méthode des divisions successives que vous utilisez pour les conversions du décimal vers le binaire et qui peut être généralisée en base b.

$$\begin{aligned} 18 &= 2 \times 9 + 0 \\ 9 &= 2 \times 4 + 1 \\ 4 &= 2 \times 2 + 0 \\ 2 &= 2 \times 1 + 0 \\ 1 &= 2 \times 0 + 1 \end{aligned}$$

On s'arrête lorsque le quotient est 0 et on écrit les restes dans l'ordre inverse de l'ordre d'apparition. On obtient $(18)_{10} = (10010)_2$

Bien sûr, il faudra généraliser en remplaçant 2 par la base b.

4 Problème des 8 dames (difficile, bonus)

On dispose d'un échiquier de n par n. Le but est de déterminer la liste des façons de poser n dames sur l'échiquier de sorte qu'aucune dame ne menace une autre dame.

Pour rappel, au jeu d'échecs, une dame menace toutes les pièces de sa ligne, de sa colonne, et des deux diagonales auxquelles elle appartient.

On peut noter que comme deux dames ne peuvent pas se situer sur la même ligne, et comme on doit placer exactement autant de dames qu'il y a de lignes, chaque solution comporte exactement une dame par ligne.

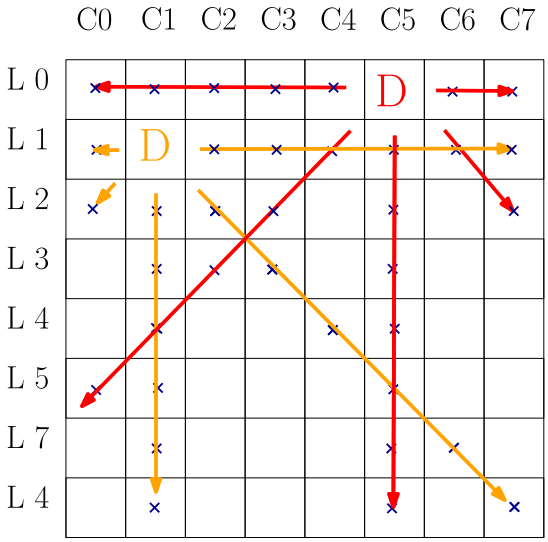
On choisit donc d'encoder un placement des dames par une liste $[C_0, C_1, ..., C_{n-1}]$ où $C_i \in [0..n-1]$ indique l'indice de la colonne sur laquelle est placée la dame de la ligne d'indice i.

Ainsi sur l'échiquier de taille 8×8, représenté ci-dessus (Figure 1), sur lequel on n'a pour l'instant placé que 2 dames, le placement correspondant sera encodé par une liste de 8 entiers commençant ainsi $[5, 1, \ldots]$.

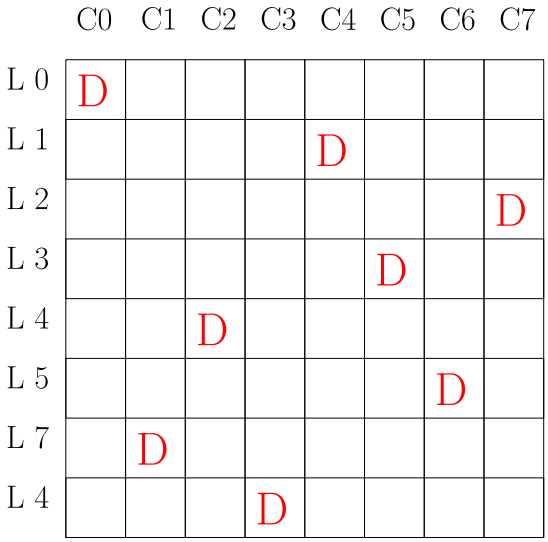
Une solution au problème est un placement dans lequel aucune des dames ne menace une autre dame. Par exemple, le placement $[0, 4, 7, 5, 2, 6, 1, 3]$ est une solution au problème comme le montre l'échiquier de la figure 2.

Il s'agira donc ici de générer la liste des solutions possibles.

Vous travaillerez dans le fichier `dames.c`. Chaque placement sera représenté par une liste d'entiers et l'ensemble des solutions sera une liste chaînée de placements. Vous utiliserez la structure suivante :



- Figure 1 -



- Figure 2 -

```
typedef struct liste_solutions{
int * placement;
struct liste_solutions * next;
} liste_solutions;
```

La fonction `addSolution` permet d'ajouter un placement à la liste des solutions. Cette fonction est fournie.

Si besoin, une fonction `displayGame` vous est également fournie. Elle permet de visualiser l'échiquier de taille $n \times n$ correspondant à un placement donné par une liste d'entiers.

Pour déterminer l'ensemble des solutions, l'idée est d'écrire une fonction récursive qui, connaissant les emplacements des dames sur les premières lignes L_0, L_1 , L_{i-1}, détermine toutes les cases possibles sur lesquelles on peut placer une dame sur la ligne L_i.

Question 13 : Compléter la fonction `casePossible` dont le prototype est `int casePossible(int * placement,int l,int c)`.
Exceptionnellement cette fonction ne doit pas forcément être récursive et peut utiliser une boucle.
À partir d'un placement donné des dames sur les lignes 0, 1, ..., $l-1$ cette fonction retournera un entier valant 1 s'il est possible de placer une dame sur la case de coordonnées $(l;c)$, et retournera 0 sinon.

Par exemple, si le placement est le tableau $[5,2,0,0,0,0,0,0]$, et si $l=2$ et $c=3$, `casePossible` retournera 0 car la case $(2;3)$ est menacée par la dame de la ligne 0 (voir figure 1).
En revanche, pour le même placement mais avec $l=2$ et $c=4$, `casePossible` retournera 1 car la case $(2;4)$ n'est pas menacée.

Aide : On pourra remarquer que deux cases de coordonnées $(l;c)$ et $(l';c')$ sont sur une même diagonale si et seulement si $|l-l'|=|c-c'|$.

Question 14 : Compléter la fonction `addAllSolutions` dont le prototype est `liste_solutions * addAllSolutions(liste_solutions * l_sol, int * placement,int l,int n)`.

Cette fonction récursive doit ajouter à la liste de solutions `l_sol`, toutes les solutions possibles sur un

échiquier de taille n, les dames ayant déjà été placées sur les lignes 0,1,..,$l-1$.

Cette fonction doit donc, pour tous les cas possibles, placer les dames sur les lignes l, $l+1$,...,$n-1$, puis ajouter le placement ainsi obtenu à la liste `l_sol`.

Compléter également la fonction récursive `nbSolutions` qui doit retourner la taille de la liste des solutions, et le programme principal `main` pour qu'il détermine et affiche toutes les solutions dans le cas d'un échiquier de 8×8 cases.

Pour information, il y a 92 solutions dans un échiquier de taille 8×8.

Question 15 : Libérez la mémoire ...

Structures de données (ZZ1, S2)

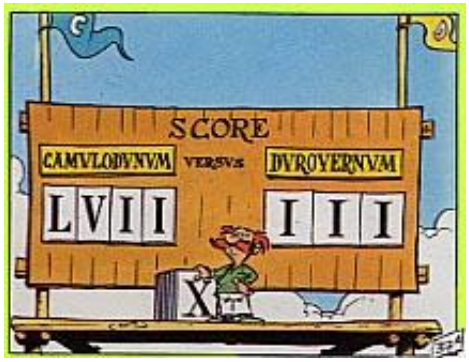
TP 3 - récursivité

L'objectif de ce TP est de d'utiliser la récursivité dans diverses situations. Les fonctions à écrire doivent donc dans la plupart des cas être récursives.

Ainsi, les exercices devront être réalisés **sans utiliser de boucle** (sauf exception).

Toutes les fonctions que vous réaliserez dans ce TP pourront être testées en utilisant le fichier `main.c` fourni. Vous commenterez/décommenterez les appels de fonction dans le main de ce fichier selon ce que vous souhaitez tester.

1 Première fonction récursive



L'objectif de cette partie est d'écrire une fonction récursive permettant d'évaluer un nombre écrit en chiffres romains.
Rappelons que :

- M = 1000
- D = 500
- C = 100
- L = 50
- X = 10
- V = 5
- I = 1

Comme exemple, évaluons le nombre MCMXCIX :

- On est sur le premier M.
  Son successeur est C, C est plus petit que M, donc notre résultat final sera la valeur de M (1000) plus la valeur du reste (CMXCIX).
- La valeur du reste CMXCIX est la suivante :
  C est plus petit que M (une soustraction aura lieu) donc la valeur de CMXCIX est égale à la valeur de MXCIX moins la valeur de C
- La valeur de MXCIX est la suivante :
  M est plus grand que X donc on a 1000+valeur(XCIX).
- La valeur de XCIX est égale à la valeur de CIX moins la valeur de X (car le premier X est plus petit que son successeur).
- La valeur de CIX est égale à 100 + valeur(IX) car C est plus grand que I.
- La valeur de IX est égale à la valeur de X moins la valeur de I, soit 10-1 = 9.

On a donc : CIX = C + 9 = 109
XCIX = CIX - X = 109 - 10 = 99
MXCIX = M + XCIX = 1000 + 99 = 1099
CMXCIX = MXCIX - C = 1099 - 100 = 999
MCMXCIX = 1000 + 999 = 1999

Question 1 : Dans le fichier `romains.c`, vous devez compléter la fonction `nombreRomainToDecimal` qui calcule récursivement et retourne la valeur décimale d'un nombre écrit en chiffres romains. Le nombre en chiffres romains est donné sous forme de chaîne de caractères.

Vous pouvez utiliser la fonction `chiffreRomainToDecimal` qui vous est fournie et qui transforme un caractère (mais pas un nombre) correspondant à un chiffre romain en un entier correspondant à sa valeur décimale.

Les affichages au début et à la fin de la fonction `romains.c` vous permettront d'observer la suite des appels à cette fonction.

Pour tester votre fonction, ne créez pas de `main` dans `romains.c`, mais utilisez le fichier `main.c` (fonction `test_romains`).

Aide :
Calculer un nombre romain, c'est :
- calculer la valeur du premier chiffre (chiffre de gauche),
- calculer le nombre constitué des chiffres suivants
- réaliser une addition ou une soustraction.

Attention de ne pas oublier le cas de base correspondant à un nombre romain de longueur 0.

2 Récursivité et Listes

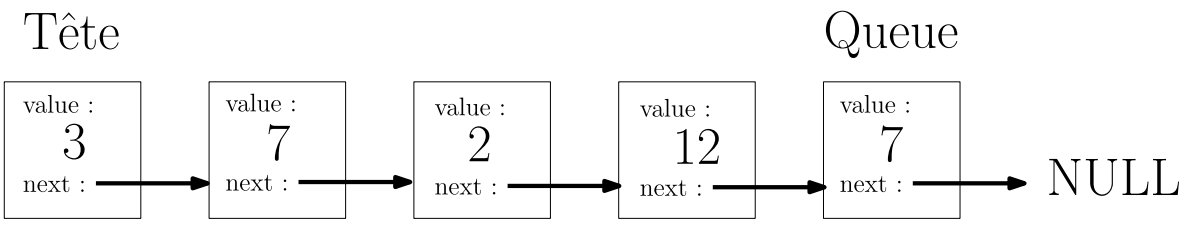
2.1 Module listes

Dans cette partie, nous allons programmer en langage C quelques fonctions sur les listes chaînées en utilisant la récursivité.

Il est recommandé de tester vos fonctions au fur et à mesure que vous les écrivez. Les tests sont à réaliser dans `main.c`.
Notez que pour chaque fonction `listFct` à réaliser dans `listes.c`, une fonction associée `test_listFct` permet de la tester dans `main.c`

Vous utiliserez la structure de données suivante qui est une liste d'entiers :

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



Les fichiers et `listes.c` vous sont fournis avec certaines fonctions de base déjà implémentées (`listCreate` et `listAdd`).

Vous répondrez aux questions suivantes en complétant `listes.c` et vous réaliserez au fur et à mesure les tests dans `main.c`.
La plupart des fonctions de test sont déjà écrites, certaines sont toutefois à compléter.

2.2 Fonctions à écrire

Les fonctions suivantes doivent toutes êtres récursives.
Question 2 : La fonction `listDisplay` a le prototype suivant `void listDisplay(list * )`.
Elle affiche toutes les valeurs de la liste, sur une même ligne, séparées par un espace, dans l'ordre où elles ont été ajoutées (donc en finissant par la tête).

Aide :
- **"Afficher toutes les valeurs"** de la liste, c'est **"afficher toutes les valeurs jusqu'à l'avant dernière"**, puis **"afficher la dernière valeur"** (tête de la liste) .
- Ne pas oublier **le cas de base**, qui survient lorsque le pointeur vers la liste à afficher est `NULL`. La queue de la liste pointant vers NULL, on finira forcément par arriver au cas de base.

Question 3 : La fonction `listInverseDisplay` a le même prototype que `listDisplay`.
Elle affiche la liste selon le même format, mais dans l'ordre inverse (en commençant par la tête).

Question 4 : La fonction `listSearch` a le prototype suivant `int listSearch(list * l, int n);`.
Elle recherche la valeur n dans la liste l et retourne 1 si la valeur n est dans la liste, 0 sinon.

Question 5 : La fonction `listMap` a le prototype suivant `list * listMap(list* , int (*f)(int));`
(le deuxième paramètre est un pointeur de fonction).
Si les valeurs de la listes sont $[n_1, n_2, n_3, \ldots, n_p]$, elle retournera une nouvelle liste dont les valeurs sont $[f(n_1), f(n_2), f(n_3), \ldots, f(n_p)]$.

Pour tester `listMap`, vous réaliserez un test dans `main.c` qui calcule et affiche une liste dont les valeurs sont les carrés de la liste_de_test. Il faudra pour cela avoir préalablement écrit une fonction `carre` puis compléter la fonction `test_listMap`.

Question 6 : La fonction `listFilter` a le prototype suivant `list * listFilter(list* li1 , int (*p)(int))`.
Elle retournera une nouvelle liste `li2` dont les valeurs sont les valeurs de la liste `li1` qui vérifient un prédicat `p` donné, implémenté dans la fonction `p`.
La fonction `p` est une fonction qui retourne 1 si l'entier donné en paramètre respecte le prédicat `p`, et retourne 0 sinon.
Pour tester `listFilter`, vous réaliserez un test dans `main.c` qui calcule et affiche une liste dont les valeurs sont les valeurs de liste_de_test qui sont strictement supérieures à 3. Vous compléterez pour cela la fonction `test_listFilter`. Il faudra préalablement avoir écrit une fonction `sup3`.

Question 7 : La fonction `listFold` a le prototype suivant `int listFold(list*  , int (*op)(int,int),int)`.
La fonction `listFold` sert à appliquer un traitement récursif aux éléments d'une liste.
Si les valeurs de la liste `l1` sont $[n_1, n_2, \ldots, n_p]$, `listfold` retournera l'entier op(n_1, op(n_2,..., op(n_p, base))).

Utiliser ensuite `listFold` dans l'implémentation les fonctions `listSum` , `listProd` et `listLen` dans le fichier `main.c`. Ces fonctions doivent calculer respectivement la somme, le produit des éléments de la liste, et la longueur de la liste (nombre de valeurs). Il faudra pour cela avoir écrit les fonctions `opSum`, `opProd` et `opLen` correspondantes.

Question 8 : La fonction `inverse` a le prototype suivant `list* listInverse(list * l1);`.
Elle retournera une liste `l_2` dont les valeurs sont celles de `l1`, mais dans l'ordre inverse.
Si les valeurs de la liste `l1` sont $[n_1, n_2, \ldots, n_p]$, celles de la liste `l2` seront $[n_p, \ldots, n_2, n_1]$.

Pour programmer la fonction `listInverse`, vous pouvez utiliser une fonction récursive intermédiaire `ajouteInverse` de prototype `list * ajouteInverse(list * l1,list * l2)` que vous aurez préalablement écrite et qui retournera la concaténation de `l1` et de l'inverse de `l2`.
Par exemple, avec les listes :

$$\texttt{l1} : (\text{Tête}) \to \boxed{7} \to \boxed{12} \to \boxed{2} \to \boxed{7} \to \boxed{3} \to \text{NULL}$$

et

$$\texttt{l2} : (\text{Tête}) \to \boxed{5} \to \boxed{6} \to \boxed{2} \to \text{NULL}$$

alors `ajouteInverse(l1,l2)` retournera la liste

$$(\text{Tête}) \to \boxed{2} \to \boxed{6} \to \boxed{5} \to \boxed{7} \to \boxed{12} \to \boxed{2} \to \boxed{7} \to \boxed{3} \to \text{NULL}$$

Dans `main.c`, la fonction `test_listInverse` permet de la tester.

Question 9 : La fonction `listFree` a le prototype suivant : `void listFree(list *)` .
Elle supprime tous les éléments de la liste.
Programmez la fonction `listFree` récursive.
Vous pourrez tester la bonne libération de la mémoire en utilisant `valgrind`.

3 Changement de base

Dans cette partie, nous utilisons le module `listes` réalisé dans le chapitre précédent pour représenter des nombres dans une base b comprise entre 2 et 16 (par exemple, si $b = 2$ on est en binaire, si $b = 8$ on est en octal, si $b = 10$ on est en décimal, et si $b = 16$ on est en hexadécimal).

La fonction `setBase` permet de définir la base (qui est stockée dans une variable globale `base`).

Modélisation :

— si $b = 2$, le nombre binaire `1101101` sera représenté par la liste chaînée
$$\text{NULL} \leftarrow \boxed{1} \leftarrow \boxed{1} \leftarrow \boxed{0} \leftarrow \boxed{1} \leftarrow \boxed{1} \leftarrow \boxed{0} \leftarrow \boxed{1} \ (\text{Tête}).$$
La tête de la liste est donc le bit de poids faible.

— si $b = 16$, le nombre hexadécimal `1AF5` sera représenté par la liste chaînée
$$\text{NULL} \leftarrow \boxed{1} \leftarrow \boxed{10} \leftarrow \boxed{15} \leftarrow \boxed{5} \ (\text{Tête}).$$
— Le principe est le même pour toutes les autres bases.

L'objectif de cette partie est de fournir un module générique de conversion, en utilisant toujours des fonctions récursives.

Les réponses aux questions sont à fournir dans le fichier `base.h`.

Les tests des fonctions à réaliser sont fournis dans le fichier `main.c`.

Question 10 : Compléter la fonction `printBaseB` qui affichera en base `b` un nombre représenté par une liste `l`.

Par exemple, si la base est 16 et si la liste chainée est NULL $\leftarrow \boxed{1} \leftarrow \boxed{10} \leftarrow \boxed{15} \leftarrow \boxed{5}$ alors la fonction `printBaseB` affichera `1AF5`.

Question 11 : Compléter la fonction `baseToDec` qui calcule la valeur décimale d'un nombre en base b représenté par une liste. Dans cette fonction, il est demandé de ne pas utiliser la fonction puissance.
On utilisera la méthode inspirée du calcul ci-dessous permettant de calculer la valeur décimale de $(1AF5)_{16}$:

$$(1AF5)_{16} = 5 + (16 \times (15 + 16 \times (10 + 16 \times 1))).$$

Vous pourrez au choix écrire une fonction récursive ou utiliser directement l'une des fonctions réalisées dans le module `liste`.

Question 12 : Compléter la fonction `decToBase` qui retourne une liste représentant un nombre en base b correspondant à un nombre décimal donné.

Dans cette fonction récursive, vous utiliserez la méthode des divisions successives que vous utilisez pour les conversions du décimal vers le binaire et qui peut être généralisée en base b.

$$\begin{aligned} 18 &= 2 \times 9 + 0 \\ 9 &= 2 \times 4 + 1 \\ 4 &= 2 \times 2 + 0 \\ 2 &= 2 \times 1 + 0 \\ 1 &= 2 \times 0 + 1 \end{aligned}$$

On s'arrête lorsque le quotient est 0 et on écrit les restes dans l'ordre inverse de l'ordre d'apparition. On obtient $(18)_{10} = (10010)_2$

Bien sûr, il faudra généraliser en remplaçant 2 par la base b.

4 Problème des 8 dames (difficile, bonus)

On dispose d'un échiquier de n par n. Le but est de déterminer la liste des façons de poser n dames sur l'échiquier de sorte qu'aucune dame ne menace une autre dame.

Pour rappel, au jeu d'échecs, une dame menace toutes les pièces de sa ligne, de sa colonne, et des deux diagonales auxquelles elle appartient.

On peut noter que comme deux dames ne peuvent pas se situer sur la même ligne, et comme on doit placer exactement autant de dames qu'il y a de lignes, chaque solution comporte exactement une dame par ligne.

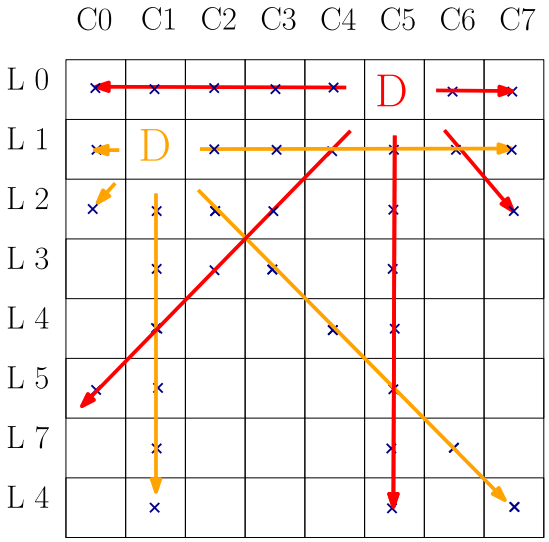
On choisit donc d'encoder un placement des dames par une liste $[C_0, C_1, ..., C_{n-1}]$ où $C_i \in [0..n-1]$ indique l'indice de la colonne sur laquelle est placée la dame de la ligne d'indice i.

Ainsi sur l'échiquier de taille 8×8, représenté ci-dessus (Figure 1), sur lequel on n'a pour l'instant placé que 2 dames, le placement correspondant sera encodé par une liste de 8 entiers commençant ainsi $[5, 1, \ldots]$.

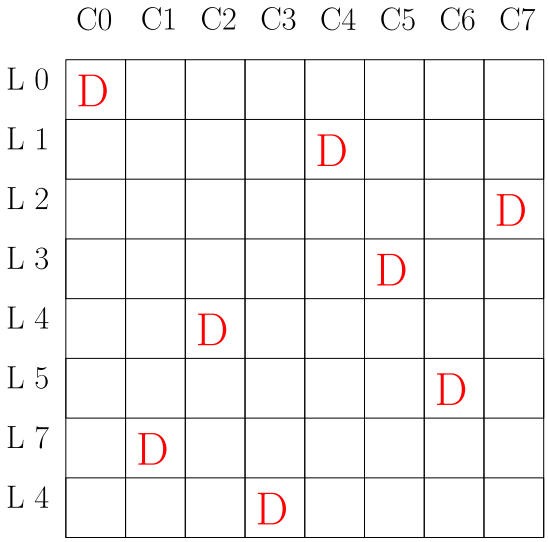
Une solution au problème est un placement dans lequel aucune des dames ne menace une autre dame. Par exemple, le placement $[0, 4, 7, 5, 2, 6, 1, 3]$ est une solution au problème comme le montre l'échiquier de la figure 2.

Il s'agira donc ici de générer la liste des solutions possibles.

Vous travaillerez dans le fichier `dames.c`. Chaque placement sera représenté par une liste d'entiers et l'ensemble des solutions sera une liste chaînée de placements. Vous utiliserez la structure suivante :



- Figure 1 -



- Figure 2 -

```
typedef struct liste_solutions{
    int * placement;
    struct liste_solutions * next;
} liste_solutions;
```

La fonction `addSolution` permet d'ajouter un placement à la liste des solutions. Cette fonction est fournie.

Si besoin, une fonction `displayGame` vous est également fournie. Elle permet de visualiser l'échiquier de taille $n \times n$ correspondant à un placement donné par une liste d'entiers.

Pour déterminer l'ensemble des solutions, l'idée est d'écrire une fonction récursive qui, connaissant les emplacements des dames sur les premières lignes L_0, L_1 , L_{i-1}, détermine toutes les cases possibles sur lesquelles on peut placer une dame sur la ligne L_i.

Question 13 : Compléter la fonction `casePossible` dont le prototype est `int casePossible(int * placement,int l,int c)`.
Exceptionnellement cette fonction ne doit pas forcément être récursive et peut utiliser une boucle.
À partir d'un placement donné des dames sur les lignes 0, 1, ..., $l-1$ cette fonction retournera un entier valant 1 s'il est possible de placer une dame sur la case de coordonnées $(l;c)$, et retournera 0 sinon.

Par exemple, si le placement est le tableau $[5, 2, 0, 0, 0, 0, 0, 0]$, et si $l = 2$ et $c = 3$, `casePossible` retournera 0 car la case $(2; 3)$ est menacée par la dame de la ligne 0 (voir figure 1).
En revanche, pour le même placement mais avec $l = 2$ et $c = 4$, `casePossible` retournera 1 car la case $(2; 4)$ n'est pas menacée.

Aide : On pourra remarquer que deux cases de coordonnées $(l; c)$ et $(l'; c')$ sont sur une même diagonale si et seulement si $|l - l'| = |c - c'|$.

Question 14 : Compléter la fonction `addAllSolutions` dont le prototype est `liste_solutions * addAllSolutions(liste_solutions * l_sol, int * placement,int l,int n)`.

Cette fonction récursive doit ajouter à la liste de solutions `l_sol`, toutes les solutions possibles sur un

échiquier de taille n, les dames ayant déjà été placées sur les lignes 0,1,..,$l-1$.

Cette fonction doit donc, pour tous les cas possibles, placer les dames sur les lignes l, $l+1$,...,$n-1$, puis ajouter le placement ainsi obtenu à la liste `l_sol`.

Compléter également la fonction récursive `nbSolutions` qui doit retourner la taille de la liste des solutions, et le programme principal `main` pour qu'il détermine et affiche toutes les solutions dans le cas d'un échiquier de 8×8 cases.

Pour information, il y a 92 solutions dans un échiquier de taille 8×8.

Question 15 : Libérez la mémoire ...

5 Fractales - triangle de Sierpinski (difficile, bonus)

On souhaite réaliser ici une fonction récursive permettant de visualiser une fractale : le triangle de Sierpinski.

Un algorithme pour obtenir des approximations proches du triangle de Sierpiński peut s'écrire de la manière récurrente suivante :

1. Commencer à partir d'un triangle quelconque du plan. Le triangle canonique de Sierpiński se construit à partir d'un triangle équilatéral ayant une base parallèle à l'axe des abscisses.
2. Tracer les trois segments qui joignent deux à deux les milieux des côtés du triangle, ce qui délimite 4 nouveaux triangles.
3. Enlever le petit triangle central : il y a maintenant trois petits triangles qui se touchent deux à deux par un sommet.
4. Recommencer à la deuxième étape avec chacun des petits triangles obtenus.

Question 16 : On peut remarquer qu'à chaque étape, la fractale est une liste chaînée de triangles. On utilisera donc la représentation suivante :

```
typedef struct point {
    double x;
    double y;
} point;

typedef struct triangle {
    point A;
    point B;
    point C;
} triangle;

typedef struct list_triangles {
    triangle ABC;
    struct list_triangles * next ;
} list_triangles;
```

Dans le fichier `sirpinski.c`, écrire une fonction récursive `void sierpinski_n(list_triangles * l,int n)` qui, à partir d'une liste de triangles `l`, réalise n étapes dans la construction des triangles de Sierpinski (c'est-à-dire réalise n fois le remplacement de chaque triangle de la liste par trois triangles plus petits).

Bien sûr, vous aurez besoin pour cela d'autres fonctions que vous aurez préalablement définies et réalisées.

Question 17 : Visualisation

Pour visualiser la fractale, vous disposez, dans le fichier `sierpinski_SDL.c`, d'un environnement SDL vous permettant de dessiner. Il vous suffira pour l'activer de décommenter l'appel à la fonction `main_SDL` dans le fichier `main.c`.

Dans le fichier `sierpinski_SDL.c`, vous disposez de la fonction `dessineLigne(int x1,int y1,int x2,int y2)` permettant de dessiner une ligne entre deux points de coordonnées $(x1; y1)$ et $(x2, y2)$ dans une fenêtre dont les coordonnées vont de $(0; 0)$ à $(1000; 800)$.

Votre dessin doit être réalisé dans la fonction `dessineFractale()`, dans laquelle vous pouvez faire appel aux fonctions réalisées dans la question précédente (module `sierpinski.h`).

Remarque : pour simplifier, on ne dessine que les contours des triangles, ils ne seront pas remplis en noir.

Pour découper le travail, vous pouvez aussi commencer par réaliser deux fonctions `dessineTriangle` et `dessineListe`, que vous appelerez dans `dessineFractale`.

Question 18 : Observez ce qui se passe quand n devient trop grand (on pourra utiliser valgrind pour comprendre).

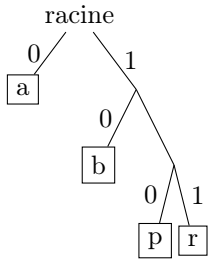
Structures de données (ZZ1, S2)

TP 4 - arbres et parcours

L'objectif de ce TP est d'utiliser les arbres et les parcours d'arbre dans le contexte du codage de Huffman.

Le codage de Huffman est un procédé très utilisé en compression de données. Il permet d'encoder un texte en binaire en utilisant un nombre variables de bits pour chaque caractère, l'idée étant d'utiliser peu de bits quand un caractère est fréquent dans le texte. Cela permet d'obtenir un encodage souvent plus léger que la codage ASCII, qui utilise 8 bits pour chaque caractère (indépendamment de sa fréquence).

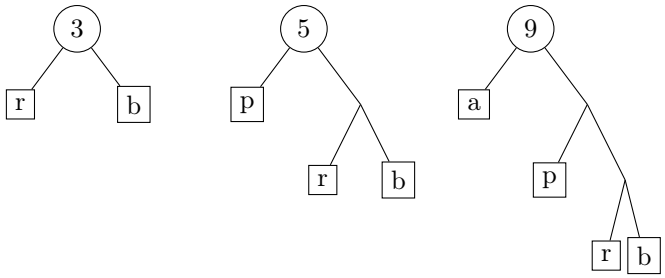
Pour représenter le codage de Huffman d'un ensemble de caractères, on utilise un arbre binaire. Dans un tel arbre, les caractères sont stockés au niveau des feuilles, les noeuds internes ne contiennent pas d'information. Pour obtenir l'encodage d'un caractère, il suffit de suivre le chemin de la racine vers la feuille contenant le caractère : suivre le sous-arbre gauche ajoute un 0, droit un 1. Par exemple dans l'arbre de Huffman ci-dessous, la caractère "a" est encodé par 0, "b" par 10, "p" par 110 et "r" par 111. Ainsi, la chaîne de caractères "barbapapa" est encodée par 10011110011001100 (soit 17 bits au lieu de 9x8=72 bits).



Lorsque l'on connaît le texte que l'on souhaite encoder, il est possible de minimiser la taille de son encodage en optimisant l'arbre. L'algorithme de Huffman permet de construire un tel arbre optimal. L'algorithme se base sur le nombre d'apparitions de chaque caractère dans le texte que l'on souhaite encoder. L'algorithme démarre avec un ensemble d'arbres (chacun contenant uniquement une feuille, donc un caractère) associé avec un poids (le nombre d'apparitions de ce caractère dans le texte). L'algorithme fonctionne ainsi, il retire de l'ensemble les deux arbres de poids minimaux pour former un nouvel arbre en les utilisant comme les sous-arbre gauche et droite. Ce nouvel arbre est alors ajouté à l'ensemble avec un poids égal à la somme des deux sous-arbres. L'algorithme s'arrête lorsqu'il ne reste plus qu'un seul arbre. En reprenant notre exemple, on commence avec l'ensemble de feuilles suivantes avec leur poids :

a : 4	b : 2	p : 2	r : 1

On construit dans les étapes qui suivent les arbres pondérés suivants, dans cet ordre :

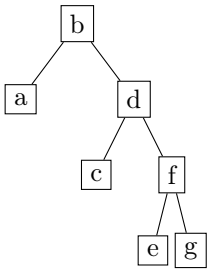


1 Dessiner des arbres

Dans cette première partie, l'objectif est de développer un outil pour afficher les arbres de Huffman que nous allons construire par la suite. Vous travaillerez dans le fichier `arbres.c` et utiliserez la structure suivante pour représenter les arbres :

```
typedef struct chararbre {
  char data;
  struct chararbre *gauche;
  struct chararbre *droite;
} arbre;
```

On souhaite imprimer un arbre de manière verticale, comme s'il avait subi une rotation de 90° sur la droite. Par exemple :




```
        /-g
    /-f
        \-e
/-d
    \-c
b
    \-a
```

Question 1 : Compléter les constructeurs `feuille` et `noeud`.
Question 2 : Compléter la fonction `est_feuille`, qui teste si un arbre est une feuille.
Question 3 : Compléter la fonction `libere_arbre`, qui libère récursivement un arbre de la mémoire. On attend de cette fonction qu'elle mette la variable contenant l'arbre à `NULL`.
Question 4 : Compléter la fonction `infixe_inverse`, qui imprime dans un fichier les caractères stockés sur les noeuds et les feuilles d'un arbre dans l'ordre infixe **inversé**. On souhaite que chaque caractère soit imprimé sur une ligne. Pour imprimer dans un fichier, on utilise la fonction `fprintf`, très similaire à `printf`. Pour imprimer le caractère `c` sur une ligne dans le fichier `f`, il suffit d'utiliser `fprintf(f, "%c\n", c)`. Si vous souhaitez imprimer dans le terminal, il suffit de passer la sortie standard `stdout` en tant que fichier. Tester votre fonction sur l'arbre ci-dessus, et observez que les caractères apparaissent déjà sur la bonne ligne pour l'impression finale.
Question 5 : Compléter la fonction `imprime_blanc(FILE* f, int niveau, int est_droit)` pour commencer chaque ligne de l'impression d'un arbre. Elle imprime dans un fichier les caractères précédant le caractère stocké (des espaces et le dessin de la branche) dans le noeud de la ligne, c'est à dire : des espaces vides suivi de `/-` ou `\-` suivant que le noeud soit un enfant droit ou gauche. Le nombre d'espaces dépend du `niveau`. L'argument `est_droit` est un booléen, qui précise si le noeud est un enfant droit ou non.
Question 6 : En reprenant la structure de la fonction `infixe_inverse`, compléter la fonction auxiliaire récursive `imprime_avec_blancs`, qui imprime un sous-arbre en fonction de son niveau et du fait que sa racine

est un enfant droit ou non. Finalement, compléter la fonction `imprime_arbre`, qui gère le cas particulier de la racine.
Tester la fonction sur l'exemple ci-dessous avec la fonction `test_impression` dans `main.c`, il faut que la réponse soit correcte au caractère près (espaces compris) ! La correction attendue est disponible dans le fichier `impression.txt`.

2 Arbres de Huffman

Dans cette section, les fonctions à implémenter sont dans le fichier `huffman.c` et `huffman.h`. Vous y trouverez les types `arbrepoids`, qui associe un poids à un arbre, et `liste`, qui représente une liste chaînée de telles associations. À l'aide de ces types, on souhaite implémenter l'algorithme de Huffman décrit au début du sujet.
Question 7 : Implémenter la fonction `genere_liste` qui, à partir d'une chaîne de caractères, génère une liste d'associations de feuilles et de poids, où les feuilles contiennent les caractères de la chaîne associés à leur fréquence, c'est-à-dire, l'ensemble initial d'arbres de l'algorithme de Huffman.
Il faut utiliser un tableau des fréquences de taille 256 où les caractères sont associés à leur code ASCII.
Question 8 : Implémenter la fonction `extrait_min`, qui cherche l'association arbre-poids de poids minimal dans une liste, qui la supprime et la retourne.
Question 9 : Implémenter la fonction `huffman` qui réalise l'arbre de Huffman pour une chaîne de caractères donnée. La fonction consistera à générer la liste des feuilles avec les bonnes fréquences, puis itérativement de retirer les deux arbres de poids minimum pour insérer dans la liste un nouvel arbre obtenu en prenant les deux arbres précédents comme sous-arbres gauche et droit. On stockera le caractère "*" à la racine de chaque nouvel arbre généré.

3 Encodage et décodage

Dans cette partie, les fonctions à implémenter sont dans le fichier `codage.c`.
Question 10 : Implémenter la fonction `code_char` qui encode, à la suite de l'encodage passé en argument, un caractère à partir d'un arbre de Huffman. Il s'agit de faire une recherche du caractère et de retourner 0 en cas d'échec.
Puis, implémenter la fonction `code_texte`, qui encode une chaîne de caractères à partir d'un arbre.
Question 11 : Implémenter la fonction `code_texte` qui encode, à la suite de l'encodage passé en argument, un caractère à partir d'un arbre.
Question 12 : Implémenter la fonction `decode_suivant`, qui décode le caractère du début de l'encodage à partir d'un arbre de Huffman ; le caractère est écrit dans un fichier. Cette fonction retourne la suite de l'encodage. Implémenter pour finir `decode` une fonction qui décode un encodage et l'écrit dans un fichier.

Inspirations :
— `https://perso.liris.cnrs.fr/vincent.nivoliers/lifapc/Supports/TP/tp_huffman.pdf`

— `https://www.di.ens.fr/~granboul/enseignement/mmfai/algo2002-2003/tp3/`

Structures de données (ZZ1, S2)

TP 5 - arbres AVL

Dans ce TP, nous allons implémenter en C la structure d'arbre binaire de recherche (ABR), ainsi que la structure d'arbre AVL (AVL).

1 Implémentation ABR

Nous cherchons à implémenter une structure de données de type ABR (nommée *BST* en anglais), dans laquelle chaque nœud possède une clé, telle que chaque nœud du sous-arbre gauche ait une clé inférieure ou égale à celle du nœud considéré, et que chaque nœud du sous-arbre droit possède une clé supérieure strictement à celle-ci.

Vous utiliserez la structure de données suivante comme un nœud dans ABR :

```
struct bst {
    int key;
    struct bst * left;
    struct bst * right;
};
```

1.1 Fonctions à implémenter

Question 1 : La fonction `bstCreateNode` a le prototype suivant : `struct bst * bstCreateNode(int key);`. Elle retourne un nœud avec une clé à `key`. Notez qu'un nœud venant d'être créé n'a pas de fils à gauche, et n'a pas de fils à droite, donc `NULL` pour les deux. Programmez la fonction `bstCreateNode`.
Remarque : vous aurez à utiliser la fonction `malloc`.
Question 2 : La fonction `bstInorderTraversal` a le prototype suivant : `void bstInorderTraversal(struct bst * root);`. Elle affiche les valeurs clés de tous les nœuds de l'ABR par ordre croissant de manière récursive. Par exemple, si les valeurs clés des nœuds dans l'ABR sont `9 7 10 13 11 14`, `bstInorderTraversal` affiche `7 9 10 11 13 14`. Programmez la fonction `bstInorderTraversal`.
Question 3 : La fonction `bstInsert` a le prototype suivant : `struct bst * bstInsert(struct bst * root, int key);`. Elle ajoute un nœud avec la clé `key` dans l'ABR dont la racine est `root`. `bstInsert` cherche d'abord l'endroit où le nouveau nœud doit être ajouté de manière récursive. Ensuite, elle invoque `bstCreateNode` pour créer le nœud et l'attacher à son père avec la valeur de retour de la récursivité. Programmez la fonction `bstInsert`.

Ensuite, nous allons implémenter la fonction `bstDelete` qui supprime un nœud dans un ABR. Pour supprimer un nœud dans un ABR :

- si le nœud à supprimer n'a pas de fils, nous allons le supprimer directement ;
- si le nœud à supprimer a un seul fils, nous allons le remplacer par ce fils ;
- si le nœud à supprimer a deux fils, nous allons le remplacer par :
  * soit le nœud du sous-arbre gauche ayant la plus grande clé ;

* soit le nœud du sous-arbre droit ayant la plus petite clé.

Dans ce TP, nous allons le remplacer par le nœud sous-arbre droit ayant la plus petite clé via la fonction `bstMinValue`.

Question 4 : La fonction `bstMinValue` a le prototype suivant : `struct bst * bstMinValue(struct bst * root);`. Elle cherche le nœud ayant la plus petite clé dans l'ABR dont la racine est `root->right`, puis elle assigne à la variable `root` ce nœud le plus petit, et retourne l'arbre. A noter qu'hormis la valeur de `root`, la fonction `bstMinValue` ne change pas d'autres nœuds au cours de l'exécution. Programmez la fonction `bstInsert`.

Question 5 : La fonction `bstDelete` a le prototype suivant : `struct bst * bstDelete(struct bst * root, int key);`. Elle supprime le nœud ayant la clé `key` dans ABR dont la racine est `root`. Elle doit renvoyer la racine du nouvel arbre obtenu après suppression. Programmez la fonction `bstDelete` en utilisant `bstMinValue`.

Question 6 : La fonction `bstFree` a le prototype suivant : `void bstFree(struct bst * root);`. Elle libère entièrement l'ABR dont la racine est `root`. Programmez la fonction `bstFree`.

Question 7 : La fonction `bstDisplay` a le prototype suivant : `void bstDisplay(struct bst * root);`. Elle affiche l'ABR dont la racine est `root` de la manière suivante :

- `7 [3 [] []] []` représente l'ABR dont la racine est `7`. La racine n'a qu'un seul fils gauche avec la valeur `3`, qui est une feuille.
- `7 [3 [] []] [9 [7 [] []] [10 [] []]]` représente l'ABR dont la racine est `7`. La racine a deux fils. Celui de gauche est `3`, qui est une feuille. Celui de droite est `9`. Ce fils de droite a aussi deux fils, celui de gauche est `7`, celui de droite est `10`.
- ...

Programmez la fonction `bstDisplay`.

1.2 Tests

- Créez un ABR vide et affichez le avec `bstDisplay`.
- Créez un ABR et ajoutez `6 7 5 8 4 9 1 3` dans cet ordre, ensuite affichez l'arbre avec `bstDisplay` et `bstInorderTraversal`.
- Supprimez `7` et affichez le avec `bstDisplay`, puis libérez l'ABR.

2 Implémentation AVL

Nous cherchons à présent à implémenter une structure de données de type AVL. Un arbre AVL est un arbre binaire de recherche (ABR) automatiquement équilibré. Dans un arbre AVL, les hauteurs des deux sous-arbres d'un même nœud diffèrent au plus de un. Pour garantir cette propriété, les opérations d'insertion et de suppression doivent rééquilibrer l'arbre AVL, quand nécessaire.

Vous utiliserez la structure de données :

```
struct avl {
    int key;
    struct avl * left;
    struct avl * right;
};
```

Vous disposez d'une fonction `max` qui retourne le plus grand de deux entiers :

```
int max(int a, int b) {
    if (a>b) {
        return a;
    }
    else {
        return b;
```

```
    }
}
```

2.1 Fonctions à implémenter

Question 8 : La fonction `avlCreateNode` a le prototype suivant : `struct avl * avlCreateNode(int key);`. Elle retourne un nœud avec une clé à `key`. Notez qu'un nœud qui vient d'être créé n'a pas de fils à gauche ou à droite. Programmez la fonction `avlCreateNode`.
Remarque : vous aurez à utiliser la fonction `malloc`.

Ensuite, nous allons chercher à implémenter les fonctions `avlInsert` qui ajoute un nœud dans l'AVL et `avlDelete` qui supprime un nœud dans l'AVL. A noter que dans les deux cas, nous aurons besoin de réaliser des fonctions intermédiaires.

L'insertion et la suppression pour les AVL ressemblent à celles pour les ABR. De plus, dans un arbre AVL, chaque ajout ou suppression nécessite un rééquilibrage de la structure de données afin de garantir les propriétés d'un AVL.

Afin de rééquilibrer l'arbre AVL, le facteur d'équilibre, appelé `balance`, d'un nœud dans un arbre AVL est introduit : la balance d'un nœud est la différence entre la hauteur de son sous-arbre gauche et celle de son sous-arbre droit. Lorsque la valeur absolue de la balance d'un nœud est strictement supérieure à 1, le sous-arbre enraciné dans ce nœud doit être rééquilibré. Deux fonctions, `avlHeight` et `avlGetBalance`, sont nécessaires pour calculer la balance d'un nœud.
Question 9 : La fonction `avlHeight` a le prototype suivant : `int avlHeight(struct avl * root);`. Elle retourne 0 si `root` est `NULL`, sinon, elle retourne la valeur de la hauteur du nœud `root`. Programmez la fonction `avlHeight`.
Question 10 : La fonction `avlGetBalance` a le prototype suivant : `int avlGetBalance(struct avl * root);`. Elle retourne la différence entre la hauteur du sous-arbre gauche et celle du sous-arbre droit du nœud `root`. Programmez la fonction `avlGetBalance`.

Nous allons ensuite implémenter deux fonctions de rotation de base vues en cours, `avlRotateLeft` et `avlRotateRight`.
Question 11 : La fonction `avlRotateLeft` a le prototype suivant : `struct avl * avlRotateLeft(struct avl * x);`. Elle prend un nœud `x`, qui est la racine du sous-arbre à rééquilibrer, et retourne la nouvelle racine après rotation. Programmez la fonction `avlRotateLeft`.
Question 12 : La fonction `avlRotateRight` a le prototype suivant : `struct avl * avlRotateRight(struct avl * y);`. Elle prend un nœud `y`, qui est la racine du sous-arbre à rééquilibrer, et retourne la nouvelle racine après rotation. Programmez la fonction `avlRotateRight`.
Question 13 : Programmez les fonctions `avlRotateLeftRight` et `avlRotateRightLeft`, qui implémentent les doubles rotations nécessaires pour l'insertion.

Nous allons ensuite implémenter la fonction d'ajout `avlInsert` en utilisant les fonctions `avlHeight`, `avlGetBalance`, `avlRotateLeft`, `avlRotateRight`, `avlRotateLeftRight` et `avlRotateRightLeft`.
Question 14 : La fonction `avlInsert` a le prototype suivant : `struct avl * avlInsert(struct avl * root, int key);`. Elle ajoute un nœud avec la clé `key` dans l'arbre AVL dont la racine est `root`. `avlInsert` d'abord cherche l'endroit où le nouveau nœud doit être ajouté (un fils nœud `NULL`) de manière récursive. Ensuite, elle invoque `avlCreateNode` pour créer le nœud et l'attacher sur son père avec la valeur de retour de la récursivité. Après l'insertion, le rééquilibrage sera invoqué. Pour cela, il faut recalculer la nouvelle balance du nœud, puis faire la rotation, si nécessaire. Programmez la fonction `avlInsert`.

Pour supprimer un nœud, outre les fonctions mentionnées ci-dessus, la fonction `avlMinValue` qui est très similaire à `bstMinValue` est également utilisée.
Question 15 : La fonction `avlMinValue` a le prototype suivant : `struct avl * avlMinValue(struct avl * root);`. Elle cherche le nœud ayant la plus petite clé dans l'arbre AVL dont la racine est `root`, puis elle remplace la racine par ce nœud le plus petit et puis le retourne. Programmez la fonction `avlMinValue`.
Question 16 : La fonction `avlDelete` a le prototype suivant : `struct avl * avlDelete(struct avl * root, int key);`. Elle supprime le nœud ayant la clé `key` dans l'arbre AVL dont la racine est `root`, puis

rééquilibre l'arbre. Elle fonctionne de manière récursive. Programmez la fonction `avlDelete` en utilisant `avlMinValue`.
Question 17 : La fonction `avlFree` a le prototype suivant : `void avlFree(struct avl * root);`. Elle libère entièrement l'arbre AVL dont la racine est `root`. Programmez la fonction `avlFree`.
Question 18 : La fonction `avlDisplay` a le prototype suivant : `void avlDisplay(struct avl * root);`. Elle affiche l'arbre dont la racine est `root` de manière similaire à la fonction `bstDisplay`. Programmez la fonction `avlDisplay`.

2.2 Tests

— Créez un arbre AVL vide et affichez le avec `avlDisplay`.
— Créez un AVL et ajoutez `6 7 5 8` dans cet ordre. Ensuite, affichez l'arbre avec `avlDisplay`. Puis, ajoutez `4 9 1 3` et affichez l'arbre avec `avlDisplay`.
— Créez un AVL et ajoutez `6 7 5 8 4 9 1 3` dans cet ordre, puis affichez l'arbre avec `avlDisplay`, puis supprimez `7` et affichez l'arbre avec `avlDisplay`, puis finalement libérez l'arbre AVL

Structures de données (ZZ1, S2)

TP 6 (optionnel) - arbres rouge-noir

Dans ce TP, nous allons implémenter en C la structure d'arbre rouge-noir vue dans la matière. L'arbre rouge-noir est un type particulier d'arbre binaire de recherche (ABR) équilibré.

Dans l'arbre rouge-noir :

— Chaque nœud a une couleur : rouge ou noir.
— Les feuilles (`NULL`) sont considérées noires.
— Un nœud rouge ne peut pas avoir un fils rouge.
— Chaque chemin d'un nœud à ses feuilles traverse le même nombre de nœuds noirs.
— Les opérations de recherche, d'insertion et de suppression sont en $\mathcal{O}(log(n))$.

Vous utiliserez les structures de données suivantes comme un nœud dans l'arbre rouge-noir :

```
enum nodeColor {
    RED,
    BLACK
};

struct rbNode {
    int data, color;
    struct rbNode *link[2];
};
```

Où `link[0]` est pour le fils de gauche et `link[1]` est pour le fils de droite.

1 Fonctions à implémenter

Question 1 : La fonction `rbtCreateNode` a le prototype suivant : `struct rbNode * rbtCreateNode(int data);`. Elle retourne un nœud avec une clé à `data` et `color` à `RED`. Notez qu'un nœud qui vient d'être créé n'a pas de fils à gauche, et n'a pas de fils à droite, donc il faut initialiser les deux pointeurs à `NULL`. Programmez la fonction `rbtCreateNode`.
Remarque : vous aurez à utiliser la fonction `malloc`.

Ensuite, nous allons implémenter la fonction `rbtInsert` qui ajouter un nœud dans l'arbre rouge-noir et `rbtDelete` qui supprime un nœud dans dans l'arbre rouge-noir. A noter que dans les deux cas, les procédures dans l'arbre rouge-noir sont similaires à celles des ABR (cf TP5). De plus, dans un arbre rouge-noir, chaque ajout ou suppression nécessite une vérification/correction de la violation de la règle des noeuds rouges, afin de garantir les propriétés de l'arbre rouge-noir. Durant la vérification/correction, deux rotations de base vues en TP5 (pour les arbres AVL) sont aussi utilisées.

Pour faciliter le suivi des nœuds sur les branches de l'arbre rouge-noir et de leurs couleurs correspondantes, deux tableaux seront utilisés, `struct rbNode *stack[100]` et `int dir[100]`. `stack` est utilisé pour suivre les nœuds sur le chemin allant de la racine au nœud qui a été ajouté/supprimé. `dir` est utilisé pour savoir

si le nœud est un fils gauche (`dir=0`) ou droit (`dir=1`) de son père.. Vous verrez ces deux tableaux dans le fichier que vous devez compléter.

Question 2 : La fonction `rbtInsert` a le prototype suivant : `struct rbNode * rbtInsert(struct rbNode * root, int data);`. Elle ajoute un nœuds avec la clé `data` dans l'arbre rouge-noir dont la racine est `root`. Après l'insertion, la vérification/correction sera invoquée. Programmez la fonction `rbtInsert`.

Question 3 : La fonction `rbtDisplay` a le prototype suivant : `void rbtDisplay(struct rbNode *node);`. Elle affiche l'arbre rouge-noir dont la racine est `node` de la manière suivante :

- `0` signifie `RED` et `1` signifie `BLACK`
- `5(1) [3(1) [] []] [7(1) [6(0) [] []] []]` représente l'arbre rouge-noir dont la racine est `5` en `BLACK`. La racine a deux fils. Celui de gauche est `3` en `BLACK`, qui est une feuille. Celui de droite est `7` en `BLACK`. Le fils de droite a aussi un fils de gauche, `6` en `RED`.

Programmez la fonction `rbtDisplay`.

Question 4 : [Question optionnelle] La fonction `rbtDelete` a le prototype suivant : `struct rbNode * rbtDelete(struct rbNode * root, int data);`. Elle supprime le nœud ayant la clé `data` dans l'arbre rouge-noir dont la racine est `root`. Programmez la fonction `rbtDelete`, en trouvant sur Internet l'algorithme à utiliser pour maintenir les propriétés de l'arbre (cet algorithme n'est pas vu en cours, et n'est pas au programme de l'examen écrit).

2 Tests

- Créez un arbre rouge-noir vide et affichez le avec `rbtDisplay`.
- Créez un arbre rouge-noir vide et ajoutez `6 7 5 8` dans l'ordre. Affichez-le avec `rbtDisplay`. Puis ajoutez les valeurs `4 9`. Affichez-le à nouveau avec `rbtDisplay`.

Structures de données (ZZ1, S2)

TP 7 (optionnel) - B-arbres

Dans ce TP, nous allons implémenter en C la structure de B-arbres vue en cours. Un B-arbre est une structure de données en arbre équilibré. Il stocke les données sous une forme triée et permet une exécution des opérations d'insertion et de suppression en temps toujours logarithmique. Un B-arbre d'ordre m est un arbre m-aire auto-équilibré. Chaque nœud de B-arbre possède $k \leq m-1$ clés et vérifie les propriétés suivantes :

— Chaque nœud a au plus m fils.
— Chaque nœud interne (tous les nœuds sauf la racine et les feuilles) ont au moins $\lceil (m/2) \rceil$ fils.
— Chaque nœud non-feuille a au moins 2 fils.
— Toutes les feuilles sont au même niveau.
— Tout nœud non-feuille avec k fils a $k-1$ clés.

Vous utiliserez la structure de données suivante comme un nœud dans B-arbre :

```
#define MAX 3
#define MIN 2

struct BTreeNode {
    int item[MAX + 1], count;
    struct BTreeNode *linker[MAX + 1];
};
```

Où MAX représente le nombre maximum de clés d'un noeud (on a donc $MAX+1 = m$), `item[]` et `linker[]` sont deux tableaux pour les clés de type `int` et les fils de type `struct BTreeNode *`. `count` sert à indiquer le nombre de clés stockées dans un nœud.

1 Fonctions d'insertion

Pour insérer une clé dans un B-arbre, la fonction `bTreeInsert` sera invoquée. Elle essaiera d'insérer une clé à la position correcte dans le bon nœud du B-arbre. Si cette insertion de clé entraîne qu'un nœud ne satisfait plus les propriétés du B-arbre, le nœud se divisera en deux et la valeur médiane des clés sera insérée dans le nœud parent. Cela peut potentiellement conduire à une nouvelle division du noeud parent.

Pour cela, `bTreeInsert` invoque d'abord la fonction `setValueInNode` dans le fichier `bTree.c`, qui tente de trouver de manière récursive le bon noeud pour l'ajout dans le B-arbre.

Lorsque la sous-fonction récursive de `setValueInNode` retourne 1, si cette insertion entraîne une division du nœud, la fonction principale de `setValueInNode` invoque la fonction `splitNode` pour effectuer l'opération de division et retourne 1. Sinon, la fonction principale invoque la fonction `addValToNode` pour effectuer une insertion normale et retourne 0.

Question 1 : La fonction `addValToNode` a le prototype suivant : `void addValToNode(int item, int pos, struct BTreeNode *node, struct BTreeNode *child);`. Elle ajoute une clé `item` avec le fils correspondant `child` sur la postion `pos` du nœud `node`. Programmez la fonction `addValToNode`.

Question 2 : La fonction `splitNode` a le prototype suivant : `void splitNode(int item, int *pval, int pos, struct BTreeNode *node, struct BTreeNode *child, struct BTreeNode **newNode);`. Elle divise un nœud avec un nombre de clés plus grand que `MAX` en deux nœuds, et renvoie en retour la valeur de la clé intermédiaire et sa position via les paramètres d'entrées. Programmez la fonction `splitNode`.

Lorsque la fonction `setValueInNode` dans `bTreeInsert` est terminée, si son retour est 1, cela signifie qu'après l'insertion, une nouvelle racine doit être créé pour assurer les propriétés du B-arbre. À ce moment, `bTreeCreateNode` sera invoquée afin d'obtenir une nouvelle racine mise à jour.

Question 3 : La fonction `bTreeCreateNode` a le prototype suivant : `struct BTreeNode * bTreeCreateNode(struct BTreeNode *child, int item);`. Elle utilise la clé `item` pour créer un nœud avec l'ancienne racine `root` comme premier fils et `child` comme deuxième fils. Programmez la fonction `bTreeCreateNode`.

Question 4 : La fonction `bTreeInsert` a le prototype suivant : `struct BTreeNode * bTreeInsert(struct BTreeNode * root, int item);`. Elle insère la clé `item` dans le B-arbre dont le racine est root. Programmez la fonction `bTreeInsert` en utilisant `setValueInNode` et bTreeCreateNode.

2 Fonctions d'affichage

Question 5 : La fonction `bTreeInorderTraversal` a le prototype suivant : `void bTreeInorderTraversal(struct BTreeNode *myNode);`. Elle affiche les valeurs clés de tous les nœuds dans B-Tree par ordre croissant de manière récursive. Par exemple, si les valeurs clés des nœuds dans le B-arbre sont `20 35 5 15 30 70`, `bTreeInorderTraversal` affiche `5 15 20 30 35 70`. Programmez la fonction `bTreeInorderTraversal`.

3 Fonctions de suppression (optionnel)

La suppression dans un B-arbre n'est pas au programme de l'examen écrit.

Pour supprimer une clé dans un B-arbre, la fonction `bTreeDelete` sera invoquée. Elle essaiera de supprimer une clé à la position correcte du nœud correct du B-arbre dont la racine est `r`. Si cette suppression de clé entraîne qu'un nœud ne satisfait plus aux propriétés d'un B-Arbre, les nœuds concernés seront ajustés : soit en les déplaçant vers la gauche ou la droite, soit en fusionnant deux nœuds. Lorsque la suppression d'un clé entraîne la suppression d'un nœud entier, il faut conserver ses fils.

Pour cela, `bTreeDelete` invoque d'abord la fonction `delValFromNode` du fichier à compléter, qui tente de trouver de manière récursive la bonne position de suppresion dans le B-arbre dont la racine est `root`.

Lorsqu'un endroit est trouvé pour la suppression, la fonction `removeVal` sera invoquée pour procéder à la suppression si celle-ci n'entraîne pas la violation de la propriété de B-arbre. En revanche, si la propriété B-arbre n'est plus satisfaite, l'ajustement correspondant sera effectué via la fonction `adjustNode`. Dans la fonction `adjustNode`, les sous-fonctions `leftShift`, `rightShift` et `mergeNodes` sont exécutées comme il convient.

Question 6 : La fonction `removeVal` a le prototype suivant : `void removeVal(struct BTreeNode *myNode, int pos);`. Elle supprime un fils sur la postion `pos` du nœud `myNode`. Programmez la fonction `removeVal`.

Question 7 : La fonction `leftShift` a le prototype suivant : `void leftShift(struct BTreeNode *myNode, int pos);`. Elle effectue un déplacement vers la gauche à la position pos du nœud myNode. Programmez la fonction `leftShift`.

Question 8 : La fonction `rightShift` a le prototype suivant : `void rightShift(struct BTreeNode *myNode, int pos);`. Elle effectue un déplacement vers la droite à la position pos du nœud myNode. Programmez la fonction `rightShift`.

Question 9 : La fonction `mergeNodes` a le prototype suivant : `void mergeNodes(struct BTreeNode *myNode, int pos);`. Elle fusionne deux fils aux positions `pos-1` et pos du nœud `myNode`. Programmez la fonction `mergeNodes`.

Question 10 : La fonction `adjustNode` a le prototype suivant : `void adjustNode(struct BTreeNode *myNode, int pos);`. Elle ajuste les nœuds concernés associés à la position `pos` du nœud myNode. Programmez la fonction `adjustNode` en utilisant `leftShift`, `rightShift` et `mergeNodes`.

Question 11 : La fonction `bTreeDelete` a le prototype suivant : `struct BTreeNode * bTreeDelete (struct BTreeNode *myNode, int item);`. Elle supprime la clé `item` dans le B-arbre dont le racine est myNode. Programmez la fonction `bTreeDelete` en utilisant `delValFromNode`.

4 Tests

— Créez un B-arbre et ajoutez `6 7 5 8` dans l'ordre. Affichez-le avec `bTreeInorderTraversal`. Ajoutez ensuite `4 9 15 29` et affichez l'arbre à nouveau avec `bTreeInorderTraversal`.

— Créez un B-arbre et ajoutez `6 7 5 8 4 9 15 29` dans l'ordre, puis affichez-le avec `bTreeInorderTraversal`. Supprimez ensuite `5 9` et affichez l'arbre avec `bTreeInorderTraversal`.

Structures de données (ZZ1, S2)

TP 8 - preuves

Dans ce TP, nous allons vérifier la terminaison d'un programme donné de manière à suivre le variant de la boucle et sa tendance. Notez que chaque boucle du programme doit être surveillée indépendamment.

```
#include <stdio.h>
int main(){
    int i = 12, j;
    while (i >= 0){
        j = 1;
        while (j != i%10){
            j += 1;
        }
        i -= 1;
    }
    return 0;
}
```

A chaque itération de la boucle, nous examinerons le changement du variant et l'enregistrerons dans une liste dédiée.

Vous utiliserez la structure suivante :

```
typedef struct record {
    int variant;
    int limit;
    struct record * next;
} record;
typedef struct list {
    int length;
    record * head;
    record * tail;
} list;
```

Lorsque le nombre d'enregistrements dans la liste atteint 20, une fonction, `veriTer`, qui analyse la tendance du variant de la boucle pour toutes ces 20 itérations est invoquée. Cette fonction vérifie si le variant se rapproche de la condition de fin de la boucle à chaque itération.

Le programme est suivi de la manière suivante :

— Si ce n'est pas le cas, c'est-à-dire que le variant ne s'approche pas de la borne, la boucle ne satisfait donc pas à la propriété de terminaison. Dans ce cas, le contenu de la liste est affiché à l'écran avec un message d'erreur, puis la liste est libérée et le programme se termine.

— Si le variant se rapproche de la borne de boucle, la boucle, au moins pour l'instant, satisfait à la propriété de terminaison. Dans ce cas, le contenu de la liste est affiché, puis vidé. Le programme continue à s'exécuter.
— Lorsque la boucle est sur le point de se terminer, nous affichons le contenu de la liste actuelle et signalons la fin de la boucle, puis nous vidons la liste.

1 Fonctions à implémenter

Question 1 : La fonction `listCreate` a le prototype suivant : `list * listCreate();`. Elle retourne une list vide. Notez qu'une liste vide a une taille de 0, n'a pas de tête, et n'a pas de queue. Programmez la fonction `listCreate`.
Remarque : vous aurez à utiliser la fonction `malloc`.
Question 2 : La fonction `listSize` a le prototype suivant : `int listSize(list * l);`. Elle retourne la longueur actuelle de la liste `l`. Programmez la fonction `listSize`.
Question 3 : La fonction `addRecord` a le prototype suivant : `void addRecord(list * l, int variant, int limit)`. Elle ajoute les valeurs actuelles du variant et de la borne de la boucle en tant qu'enregistrement à la fin de la liste `l`. Programmez la fonction `addRecord`.
Remarque : vous aurez à utiliser la fonction `malloc`.
Question 4 : La fonction `listDisplay` a le prototype suivant : `void listDisplay(list * l);`. Elle affiche tous les éléments de la liste `l`, dans l'ordre. Programmez la fonction `listDisplay`.
Question 5 : La fonction `listFree` a le prototype suivant : `void listFree(list * l);`. Elle supprime tous les éléments de la liste `l`. Programmez la fonction `listFree`.
Remarque : vous aurez à utiliser la fonction `free`.
Question 6 : La fonction `listDestroy` a le prototype suivant : `void listDestroy(list * l);`. Elle supprime tous les éléments de la liste `l`, puis supprime la liste `l`. Programmez la fonction `listDestroy`.
Remarque : vous aurez à utiliser la fonction `free`.
Question 7 : La fonction `veriTer` a le prototype suivant : `int veriTer(list * l);`. Elle retourne `1` si la tendance du variant dans `l` s'approche de la borne de la boucle, et `0` sinon. Programmez la fonction `veriTer`.
Remarque : pour que `veriTer` retourne `1`, la valeur absolue de la différence entre le variant et la borne de la boucle dans chaque enregistrement de cette liste doit être strictement diminuée lorsque l'on parcourt la liste.

2 Tests

— Créez une liste vide et affichez la.
— Créez une liste vide et effacez tous ses éléments.
— Créez une liste vide, ajoutez (variant : 4, limit : 10), (variant : 5, limit : 9), puis ajoutez (variant : 6, limit : 8), et affichez ses éléments.
— Créez une liste vide, ajoutez (variant : 4, limit : 10), (variant : 5, limit : 9), ajoutez (variant : 6, limit : 8), testez avec `veriTer`, et affichez le résultat.
— Complétez le fichier fourni, et vérifiez la terminaison du programme donné.

Structures de données (ZZ1, S2)

TP 9 - Tris

Dans ce TP, nous allons programmer en langage C le *tri rapide* (ou *quicksort* en anglais), qui est un algorithme de tri classique pour les tableaux/listes d'entiers. C'est un algorithme efficace en *complexité moyenne*, car il utilise un nombre moyen de $O(n \log n)$ opérations élémentaires, où n est le nombre d'entiers de la liste considérée. On va trier les entiers dans l'ordre croissant.

Il est recommandé de tester vos fonctions au fur et à mesure que vous les écrivez. Dans le fichier `main.c`, vous trouverez des tests correspondant aux différentes parties du TP. Décommentez les tests de la fonction `main` de ce fichier au fur et à mesure que vous progressez dans le TP. Si vous le souhaitez, vous pouvez ajouter des tests personnels dans ce fichier `main.c` pour débugger, mais ce n'est pas obligatoire.

A travers ce TP seront évalués votre capacité à coder deux versions de l'algorithme, mais aussi à gérer convenablement la mémoire. Il faudra donc éviter les pertes mémoire, et cet aspect sera pris en compte dans l'évaluation. Utilisez `valgrind` !

1 Implémentation du tri rapide sur une liste simplement chaînée

Nous cherchons à implémenter l'algorithme de tri sur une liste chainée (nommée `list`) déjà manipulée dans des TPs précédents. Cette structure vous est rappelée ci-dessous. Une liste chainée est composée de *maillons*, contenant une valeur stockée ainsi qu'un pointeur vers le prochain maillon.

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

Le squelette qui vous est fourni contient trois fichiers : `linkedlist.h`, `linkedlist.c`, et `main.c`. Le fichier à compléter est le fichier `linkedlist.c`. Dans `linkedlist.h`, la structure de liste chainée est déclarée, ainsi que toutes les fonctions à programmer pour ce TP.

Des fonctions classiques sur les listes chainées vous sont fournies. Vous devrez vous servir dans ce TP de certaines d'entre elles :

- `list * listCreate();` Initialise une liste chainée avec un pointeur `NULL`.
- `list * listAdd(list * l, int v);` Ajoute un maillon en tête de liste avec la valeur `v`.
- `int listSize(list * l);` Renvoie la taille de la liste `l`.
- `void listDisplay(list * l);` Affiche la liste `l`.
- `void listFree(list * l);` Libère la mémoire allouée pour la liste `l`.
- `list* listCopy(list* l);` Renvoie une copie de la liste `l`.

1.1 Comparaison des éléments au pivot

Le tri rapide est un algorithme **récursif** fonctionnant sur le principe suivant : on commence par choisir un élément *pivot* dans la liste. Ensuite, on génère deux listes chainées `gauche` et `droite` telles que :

— `gauche` contient non seulement tous les éléments strictement inférieurs au pivot, mais aussi tous les éléments égaux au pivot, sauf un.
— `droite` contient tous les éléments strictement supérieurs au pivot.

Exemple : si `l = [5,7,6,8,5,3]` et que le pivot est 5, alors `gauche` contient 3 et 5 (notez qu'il n'y a qu'une seule des deux occurrences de 5) et `droite` contient 7, 6 et 8.

Observez que les listes `gauche` et `droite` peuvent être vides. De plus, la somme de la taille de ces deux listes doit être égale à la taille de la liste initiale `l` moins un (car on a enlevé de la liste `gauche` une occurrence du pivot).

Enfin, l'algorithme appelle la fonction récursivement sur `gauche` et `droite`, puis reconstruit la liste initiale mais triée, en concaténant la liste `gauche` triée, le pivot, puis la liste `droite` triée.

Question 1 : Ecrire une fonction `listPivot` qui prend en argument une liste chaînée `l` et un entier pivot (supposé présent dans la liste). Elle devra renvoyer les deux listes chaînées `gauche` et `droite` décrites plus haut. La structure à renvoyer est donc de type `list **` (voir squelette).

Attention : on va faire le choix ici de réutiliser les maillons déjà présents dans la liste initiale `l` pour former les listes `gauche` et `droite`. Autrement dit, on ne créera pas de maillons supplémentaires (c'est-à-dire, il est **INTERDIT** d'utiliser ici les fonctions `listAdd` ou `listCopy`). Vous devrez en fait totalement modifier les connexions initiales entre les maillons de la liste `l`. Ainsi la structure de `l` sera totalement altérée après un appel à `listPivot`. Ce choix d'implémentation sert à économiser la mémoire.

Astuce : pensez à libérer la mémoire du premier maillon rencontré dont la valeur est égale au pivot, et qui ne sera donc pas présent dans `gauche` et `droite`.

Question 2 : Écrire une fonction `reassemble` qui prend en argument la liste `gauche` supposée triée, la liste `droite` supposée triée, et un pivot, et qui renvoie la concaténation de ces trois éléments produisant une liste globale parfaitement triée dans l'ordre croissant. De même qu'à la **Question 1**, aucune copie de maillon ne devra être effectuée. On s'autorisera ici uniquement un seul `listAdd`, afin de créer un maillon contenant le pivot. Naturellement, le contenu des pointeurs `gauche` et `droite` pourra être altéré.

Astuce : vous pouvez simplement ajouter un maillon contenant le pivot à la liste `droite` pour commencer, et ensuite relier la fin de la liste `gauche` avec le début de la liste `droite`, pour enfin renvoyer `gauche`.

1.2 Tri rapide

On va implémenter une première version du tri rapide où le pivot sélectionné est le premier élément de la liste initiale `l`.

Question 3 : Écrire une fonction `quickSort_rec` qui est **récursive** et prend en argument une liste chaînée `l`. Elle renvoie la liste triée. On utilisera naturellement les deux fonctions codées précédemment pour économiser la mémoire. A noter que cela altérera la liste initiale `l` qui désormais est irrécupérable. On va régler ce problème dans la question suivante.

Question 4 : Écrire une fonction `quickSort` qui prend en argument une liste `l` et qui trie une copie de `l`.

A ce stade, vous pouvez décommenter les lignes `test_listPivot()` et `test_quickSort()` dans le `main` pour tester les fonctions déjà programmées.

2 Sélection aléatoire du pivot

Sélectionner comme pivot le premier élément de la liste n'est pas très judicieux. En effet, l'algorithme sera d'autant plus efficace si le pivot est proche de la médiane des entiers présents dans la liste initiale. Or, par exemple sur une liste décroissante du type [5,4,3,2,1], le premier élément de la liste est toujours l'élément maximum et ne permet donc pas de scinder efficacement la liste. On retrouve alors la *complexité en pire cas* de l'algorithme, qui est en $O(n^2)$. On va donc tester une nouvelle méthode de sélection du pivot : on va piocher au hasard trois éléments de la liste initiale, puis prendre comme pivot l'élément médian parmi ces trois valeurs.

Question 5 : Écrire une fonction `getRandomElement` qui prend en argument une liste `l`. Elle renverra un élément de la liste tiré au hasard.

Astuce : utilisez `rand() % n` pour tirer au hasard un nombre entre 0 et $n-1$.

Question 6 : Écrire une fonction `getRandomPivot` qui prend en argument une liste chaînée `l` et qui renvoie un pivot sélectionné aléatoirement. La fonction piochera trois éléments de la liste (de manière indépendante, donc le même élément peut être choisi plusieurs fois) et renverra l'élément médian parmi ces trois valeurs.

Question 7 : Écrire une fonction `quickSort_rec_alea` qui prend en argument une liste `l`. Elle effectuera un travail similaire à `quickSort_rec` mais en sélectionnant le pivot de manière aléatoire, en utilisant la fonction `getRandomPivot`.

Question 8 : Écrire une fonction `quickSort_alea` qui effectue un travail similaire à `quickSort`, mais en sélectionnant le pivot aléatoirement.

Lancez le test `test_randomPivot()` : il permet de comparer le tri rapide avec les deux méthodes de sélection du pivot que nous avons programmées. Observez la différence de temps d'exécution sur une liste décroissante de taille 3000 par exemple.