Implémentation de quelques structures de données usuelles en Basic Casio

Louloux

Introduction

J'ai pensé plusieurs fois à un tutoriel présentant certaines structures de données pratiques, car nombreux sur ce site ont appris l'informatique sur Internet et n'ont pas connaissance de celles-ci, ou ne savent simplement pas comment les implémenter. Le but est que ce tutoriel soit compréhensible par tous, ceux maîtrisant déjà ces structures de données peuvent bien sûr sauter directement à la case 'implémentation'.

Pour partir sur de bonnes bases, je vais d'abord définir ce qu'est une *structure de données* : c'est une structure logique destinée à contenir et organiser des données. Selon la structure, l'ajout, la modification ou l'accès aux données se font différemment. Les List et Mat de la calculatrices sont des implémentations natives de structures de données que sont les tableaux et les matrices. L'utilisateur d'une structure de données n'a généralement pas besoin de savoir comment elle est implémentée, mais il dispose de différentes opérations sur celle-ci et de la connaissance de leurs temps d'exécution. Mais, le Basic n'étant pas un langage fonctionnel, on ne peut gérer ses structures de données que manuellement. Si vous voulez en savoir plus, je ne peux que vous conseiller les pages Wikipédia associées à ces notions, car elles sont assez complètes et accessibles, et comportent des petits schémas qui vous aideront à visualiser, si les concepts vous paraissent trop abstraits.

Bon, vous êtes prêts ? On démarre !

Table des matières

1 Les piles : First In, Last Out **3**
1.1 Présentation 3
1.1.1 Définition 3
1.1.2 Exemple d'algorithme faisant intervenir une pile 3
1.2 Implémentation avec une List 4

2 Les files : First In, First Out **6**
2.1 Présentation 6
2.1.1 Définition 6
2.1.2 Exemple d'algorithme faisant intervenir une file 6
2.2 Implémentation avec une List 7

3 Les graphes **9**
3.1 Présentation 9
3.1.1 Exemple introductif 9
3.1.2 Généralités sur les graphes 10
3.1.3 Notion de matrice d'adjacence 10
3.1.4 Problème proposé 10
3.2 Implémentation par matrice d'adjacence 11

Chapitre 1

Les piles : First In, Last Out

1.1 Présentation

1.1.1 Définition

Les piles sont omniprésentes en informatique. Lorsque vous faites un retour arrière dans votre navigateur, ou annulez votre dernière action dans un logiciel par exemple, vous faites intervenir ce genre de structures. En plus d'être utiles dans la création d'une interface utilisateur, elles trouvent leur utilité dans de nombreux algorithmes. Nous traiterons dans ce chapitre l'exemple simple de la vérification du parenthésage d'une chaîne de caractères.

Une pile, comme son nom l'indique, permet d'empiler des éléments, c'est-à-dire de les poser au sommet de la pile, et de les y récupérer. On ne peut pas récupérer un élément en-dessous sans avoir d'abord récupéré ceux au-dessus. On considère 4 opérations possibles sur une pile :

1. Créer une pile vide
2. Tester si la pile est vide
3. Empiler un élément
4. Dépiler l'élément au sommet de la pile

1.1.2 Exemple d'algorithme faisant intervenir une pile

Un exemple simple d'utilisation d'une pile est la vérification du bon parenthésage d'une chaîne de caractères. Par bon parenthésage on entend qu'une parenthèse ou un crochet est toujours fermé plus loin, et qu'on ferme ceux-ci dans le bon ordre.

Voici quelques exemples :

$(a * b)[(c + d)(e - f)]$ est bien parenthésée.
$(a * b)[(c + d)(e - f)$ est mal parenthésée car un crochet n'est pas fermé.
$(a * b)[(c + d)(e - f])$ est mal parenthésée à cause de l'ordre de fermeture.
$a * b)[(c + d)(e - f)]$ est mal parenthésée car la première parenthèse fermante ne ferme rien.

Cet algorithme peut être implémenté de manière efficace à l'aide de piles, en empilant les parenthèses/crochets ouvrant(e)s rencontré(e)s et en les dépilant lorsqu'on rencontre la parenthèse ou le crochet fermant(e) associé(e). Si une parenthèse ferme un crochet ou inversement, ou si il n'y a rien à fermer, on affiche un message signalant le mauvais parenthésage. À la fin, on vérifie que la pile est vide. Cet algorithme vérifiant une expression s s'écrit ainsi :

```
Créer la pile p
Pour tout caractère c de s:
    Si c = '(' ou c = '[':
        Empiler c dans p
    Sinon si c = ')' ou c = ']':
        Si p est vide:
            Afficher "Parenthésage incorrect !"
            Stopper le programme
        Sinon:
            Dépiler d de p
            Si (d = '(' et c = ']') ou (d = '[' et c = ')'):
                Afficher "Parenthésage incorrect !"
                Stopper le programme
Si p est vide:
    Afficher "Parenthésage correct."
Sinon:
    Afficher "Parenthésage incorrect !"
```

1.2 Implémentation avec une List

On en arrive à la partie intéressante : l'implémentation de cette structure de données en Basic. Eh bien, c'est la plus facile à implémenter de celles que je vous présente dans ce tuto, et c'est pour cela d'ailleurs que je vous la présente en premier. Nous allons simplement utiliser une `List`, et garder en mémoire une variable indiquant le nombre d'éléments que contient la pile. Notre pile pourra contenir 999 éléments sur les modèles récents (OS $\geq$ 2.0), 255 sur les vieux modèles. Ainsi, si la `List 1` est choisie, et `N` la variable contenant sa taille, les opérations sur la pile s'écrivent :

Créer une pile vide	$999 \to Dim\ List\ 1$ $0 \to N$
Tester si la pile est vide	$If\ N = 0$ ...
Empiler un élément E	$Isz\ N$ $E \to List\ 1[N]$
Dépiler un élément	$List\ 1[N] \to E$ $N - 1 \to N$

Maintenant, nous avons tous les outils en main pour implémenter l'algorithme que j'ai utilisé comme exemple. Comme notre List doit contenir des entiers, nous pouvons numéroter '(', ')', '[' et ']' de 1 à 4. Si vous le souhaitez, essayer d'implémenter l'algorithme vous-même, sinon vous trouverez mon implémentation en page suivante.

`StrMid` permet d'obtenir une sous-chaîne d'une `Str` de taille et d'indice de début spécifiés, et `StrComp` compare 2 `Str`, en renvoyant 0 si elle sont égales.

Nous écrirons la flèche simple `->` et la double-flèche `=>`

```
"Expression "?->Str 1
999->Dim List 1
0->N
For 1->I To StrLen(Str 1)
    StrMid(Str 1,I,1)->Str 2
    0->D
    StrCmp(Str 2,"(")=0=>1->D
    StrCmp(Str 2,")")=0=>2->D
    StrCmp(Str 2,"[")=0=>3->D
    StrCmp(Str 2,"]")=0=>4->D
    If D=2 Or D=4
    Then If N=0
         Then "Mal Parenthésée !"
              Stop
         IfEnd
         List 1[N]->E
         N-1->N
         If D!=E+1
         Then "Mal Parenthésée !"
              Stop
         IfEnd
    Else If D=1 Or D=3
         Then Isz N
              D->List 1[N]
         IfEnd
    IfEnd
Next
If N=0
Then "Bien parenthésée."
Else "Mal parenthésée !"
IfEnd
```

Voilà, on en a terminé avec les piles. Passons maintenant à leurs copines les files, un tantinet plus compliquées à implémenter en Basic, je ne vous le cache pas.

Chapitre 2

Les files : First In, First Out

2.1 Présentation

2.1.1 Définition

La file est une structure de donnée très présente en informatique, ainsi que sa variante la file de priorités. Ici encore, une file correspond à ce qu'elle désigne dans la vraie vie : on entre dans une file par un côté, on sort par l'autre. On dit "first in, first out", en opposition au "first in, last out" des piles.

De même que pour les piles, on considère 4 opérations sur les files :

1. Créer une file vide
2. Tester si la file est vide
3. Enfiler un élément en queue de file
4. Défiler l'élément en tête de la file

2.1.2 Exemple d'algorithme faisant intervenir une file

Les utilisations les plus courantes des files sont loin d'être accessibles car interviennent en théorie des graphes, donc je vous propose de présenter une première utilisation plus accessible ici, et dans ma troisième partie sur les graphes je ferai également intervenir une file. Je ne vous cache pas que l'algo présenté est un peu plus compliqué que le précédent, et le suivant le sera plus encore, mais l'essentiel est que vous compreniez bien l'implémentation des files que je vous propose, et ayiez un aperçu de leur utilisation.

L'algorithme que j'ai donc choisi est celui du pot de peinture, qui colorie en noir la partie de l'écran qui contient le pixel de coordonnées spécifiées. On considérera que notre file ne dépassera jamais une taille de 999.

Le principe de l'algorithme est qu'on va créer une file ne contenant initialement que le pixel de départ, et puis effectuer une boucle dans laquelle, à chaque étape, on traite le pixel en tête de file en coloriant ses voisins non coloriés, et en les ajoutant eux-mêmes à la file. La file sera vide quand tous les pixels de la zone auront été coloriés.

On peut représenter les pixels dans la file par des nombres complexes. Si vous n'avez pas encore vu cela en cours, sachez que ce sont des nombres de la forme $z = a + ib$, contenant ainsi 2 nombres a et b, qui sont respectivement la partie réelle et la partie imaginaire de z. En Basic ils

sont utiles car permettent de représenter un couple (a, b) comme un nombre $z = a + ib$, et de le stocker dans des `List`. On accède à a en faisant `ReP z`, à b en faisant `ImP z`, et naturellement $a + ib\ +\ c + id\ =\ a + b + i(c + d)$.
Si z représente les coordonnées d'un pixel, ses voisins sont $z - 1$, $z + 1$, $z - i$ et $z + i$.

En partant du pixel (xi, yi), l'algorithme s'écrit :

```
Créer la file f
Enfiler (xi, yi) dans f
Tant que f est non vide:
    Défiler (x, y) de f
    Pour (x2, y2) parmi (x-1, y), (x+1, y), (x, y-1), (x, y+1):
        Si (x2, y2) n'est pas hors de l'écran et est éteint:
            Allumer (x2, y2)
            Enfiler (x2, y2) dans f
```

2.2 Implémentation avec une List

Voyons tout d'abord l'implémentation des files. On doit pouvoir ajouter des éléments à la fin, et récupérer ceux au début. La difficulté réside dans le fait de pouvoir ajouter des éléments indéfiniment à la fin sans être bloqué par la taille de la List, du moment qu'on en enlève assez au début. Il faut donc réutiiliser l'espace se situant avant les données, libéré en récupérant les données en tête de file. On va donc imaginer que la `List` servant de support à la file est cyclique : la case 1 est voisine de la case 999. Ainsi, on maintient deux compteurs correspondant à la case de début et la case de fin de notre file, et lorsqu'un compteur atteint 1000, on remet sa valeur à 1. Le danger est que si la file contient plus de 999 éléments, on écrira des éléments les uns sur les autres, donc il faut prévoir de ne pas arriver à cette situation.

Ainsi, si la `List 1` est choisie, et `D` et `F` les variables contenant les indices de début et de fin de la file, les opérations sur la file s'écrivent :

Créer une file vide	$999 \to Dim\ List\ 1$ $1 \to D$ $1 \to F$
Tester si la file est vide	$If\ D = F$ ...
Enfiler un élément E	$E \to List\ 1[F]$ $Isz\ F$ $F = 1000\ \ =>\ \ 1 \to F$
Défiler un élément	$List\ 1[D] \to E$ $Isz\ D$ $D = 1000\ \ =>\ \ 1 \to D$

Prêts pour l'implémentation de l'algo en Basic ? Encore une fois, les courageux peuvent le tenter eux-mêmes, les autres trouveront une implémentation en page suivante.

J'ai déjà fourni quelques informations sur mon implémentation lors de la présentation du problème, je vous encourage donc à bien relire celles-ci si vous ne comprenez pas mon code.

(`!=` représente le signe "différent de", `->` représente $\rightarrow$, et `=>` représente la double-flèche)

```
ClrText
"Point de départ :"
"X "?->X
"Y "?->Y
999->Dim List 1
X+iY->List 1[1]
PxlOn Y,X
1->D
2->F
{1,-1,i,-i}->List 2
While D != F
    List 1[D]->Z
    Isz D
    D=1000=>1->D
    For 1->I To 4
        List 2[I]->C
        ReP (Z+C)->A
        ImP (Z+C)->B
        If A>0 And B>0 And A<128 And B<64
        Then If Not PxlTest B,A
             Then PxlOn B,A
                  A+iB->List 1[F]
                  Isz F
                  F=1000=>1->F
             IfEnd
        IfEnd
    Next
WhileEnd
```

Chapitre 3

Les graphes

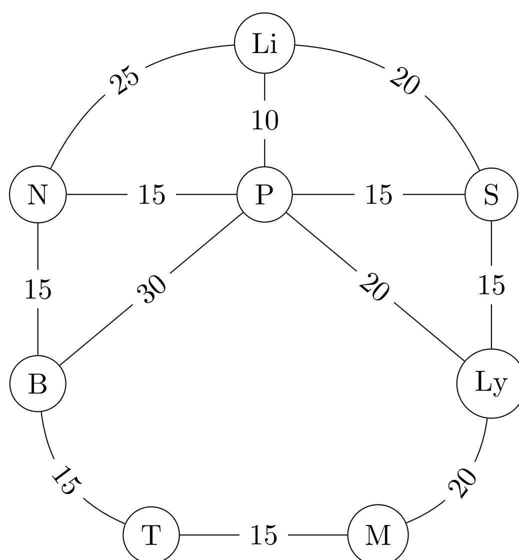
3.1 Présentation

La théorie des graphes est une immense part de l'informatique, qui permet de résoudre énormément de problèmes de manière très performante. Calcul d'un itinéraire dans un GPS, répartition des ressources dans une entreprise, organisation d'un réseau, recherche de données, intelligence artificielle : autant de domaines dans lesquels interviennent les graphes.

Si un niveau mathématique élevé est nécessaire pour explorer pleinement cette théorie, les bases sont en revanche accessibles sans connaissances précises, c'est pourquoi je vous propose une petite introduction aux graphes dans ce tutoriel. La définition mathématique d'un graphe ne vous parlera sûrement pas, donc je vais attaquer avec un exemple, et formaliser cela ensuite.

3.1.1 Exemple introductif

Dans un futur proche, un nouveau moyen de transport ultrapide est mis en place pour relier les grandes villes françaises, l'*hyperfly*. On peut représenter les lignes de ce transport, avec leur durée en minutes, de la manière suivante :



On a en fait dessiné un graphe, où les villes sont appelées les *noeuds* ou *sommets* et les lignes les *arcs* ou *arêtes*. Les lignes comportent une valeur, donc on dit qu'elles sont *pondérées*. Elles n'ont pas de sens, on dit que le graphe est *non-orienté*.

3.1.2 Généralités sur les graphes

Pour ceux que ça intéresse, voilà quelques définitions plus formelles de la notion approchée précédemment, ainsi que des exemples.

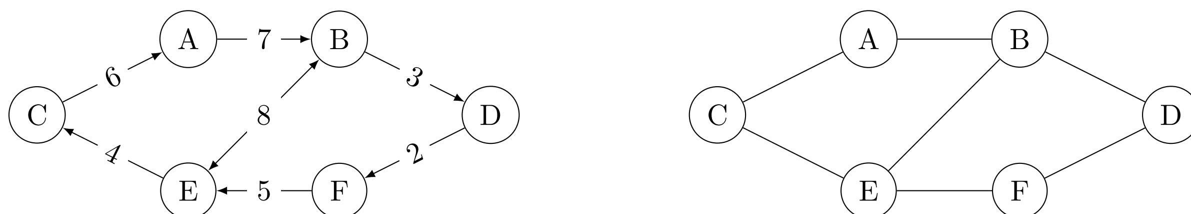
Un graphe non-orienté pondéré est un couple `(S, A)` où :

- `S` est un ensemble de sommets.
- `A` est un ensemble de triplets (s_1, s_2, p), où $(s_1, s_2) \in S^2$ et où p est le poids de l'arête entre s_1 et s_2.

Pour un graphe orienté, il suffit de considérer qu'une arête (s_1, s_2, p) va de s_1 à s_2 mais pas de s_2 à s_1, et d'écrire les arêtes doubles dans les deux sens.

Pour un graphe non-pondéré, il suffit de construire `A` comme un ensemble de couples (s_1, s_2).

Voici par exemple deux graphes proches, le premier orienté et pondéré, le second non-orienté et non-pondéré :



3.1.3 Notion de matrice d'adjacence

On peut construire la matrice d'adjacence d'un graphe de la manière suivante : on numérote les n sommets de 1 à n. Pour chaque arête (i, j, p), la case à la ligne i et à la colonne$j$ de la matrice contient p, et si le graphe n'est pas orienté, la case à la ligne j et à la colonne i de la matrice également. Les autres cases valent 0.

Pour le premier graphe de l'exemple précédent, on a ainsi la matrice suivante :

	A	B	C	D	E	F
A	0	7	0	0	0	0
B	0	0	0	3	8	0
C	6	0	0	0	0	0
D	0	0	0	0	0	2
E	0	8	4	0	0	0
F	0	0	0	0	5	0

3.1.4 Problème proposé

Un algorithme fréquemment rencontré est la recherche d'une plus court chemin entre deux sommets dans un graphe. Je vous propose donc d'implémenter l'algorithme de recherche du plus court chemin pour le problème de l'*hyperfly*, pour trouver le chemin le plus rapide entre deux des villes disposant de ce moyen de transport. Je ne rentrerai pas dans les détails sur l'algorithme car tel n'est pas mon but, mais je vous présenterai l'implémentation de celui-ci en utilisant une file et un graphe représenté par matrice d'adjacence.

(Note : on appelle *successeur* d'un sommet tout sommet étant l'extrêmité d'une des arêtes partant de celui-ci.)

L'algorithme choisi, appelé *Breadth First Search* (parcours en largeur), utilise une file pour parcourir en largeur les sommets de graphe depuis le sommet de départ, et met à jour leur distance à celui-ci, et le prédécesseur de chaque sommet dans le chemin depuis le sommet de départ. Pour aller de s_1 à s_2, avec M la matrice d'adjacence, l'algo s'écrit :

```
Créer une file f
Créer un tableau d initialisé à 0 (pour les distances)
Créer un tableau p initialisé à 0 (pour les prédécesseurs)
Enfiler s1 dans f
Tant que f est non vide:
    Défiler s de f
    Pour sp parmi les successeurs de s:
        Si p[sp] = 0 ou d[s] + M[s,sp] < d[sp]:
            d[s] + M[s,sp] -> d[sp]
            s -> p[sp]
            Enfiler sp dans f
Remonter les successeurs depuis s2, afficher le chemin
```

3.2 Implémentation par matrice d'adjacence

On représentera le graphe en Basic par une `Mat`, les tableaux par des `List`.

Pour le problème de l'*hyperfly*, on a la matrice suivante :

	Li	*N*	*P*	*S*	*B*	*Ly*	*T*	*M*
Li	0	25	10	20	0	0	0	0
N	25	0	15	0	15	0	0	0
P	10	15	0	15	30	20	0	0
S	20	0	15	0	0	15	0	0
B	0	15	30	0	0	0	15	0
Ly	0	0	20	15	0	0	0	20
T	0	0	0	0	15	0	0	15
M	0	0	0	0	0	20	15	0

En Basic, on identifiera donc les villes aux entiers de 1 à 8 comme suit :

Lille	1
Nantes	2
Paris	3
Strasbourg	4
Bordeaux	5
Lyon	6
Toulouse	7
Marseille	8

Voici enfin mon implémentation :

```
8->N
[[0, 25, 10, 20, 0, 0, 0, 0]
 [25, 0, 15, 0, 15, 0, 0, 0]
 [10, 15, 0, 15, 30, 20, 0, 0]
 [20, 0, 15, 0, 0, 15, 0, 0]
 [0, 15, 30, 0, 0, 0, 15, 0]
 [0, 0, 20, 15, 0, 0, 0, 20]
 [0, 0, 0, 0, 15, 0, 0, 15]
 [0, 0, 0, 0, 0, 20, 15, 0]]->Mat A
ClrText
"Départ "?->A
"Arrivée "?->B
N->Dim List 2
N->Dim List 3
999->Dim List 1
A->List 1[1]
1->D
2->F
A->List 3[A]
While D!=F
    List 1[D]->S
    Isz D
    D=1000=>1->D
    For 1->I To N
        If Mat A[S,I] And (Not List 3[I] Or List 2[S]+Mat A[S,I]<List 2[I])
        Then  List 2[S]+Mat A[S,I]->List 2[I]
             S->List 3[I]
             I->List 1[F]
             F=1000=>1->F
        IfEnd
    Next
WhileEnd
N->Dim List 4
B->S
1->I
While S!=A
    S->List 4[I]
    List 3[S]->S
    Isz I
WhileEnd
A->List 4[I]
I->Dim List 5
For I->K To 1 Step -1
    List 4[K]->List 5[I+1-K]
Next
List 5
```

Pour exécuter l'algorithme sur un autre graphe, changez simplement la matrice entrée au début, et la valeur du paramètre N qui représente le nombre de noeuds du graphe.

Conclusion

Voilà, j'ai fini cette petite présentation de 3 structures de données usuelles très utiles en informatique. Si vous n'avez pas tout compris, ce n'est pas grave, essayer de vous approprier ces structures de données en faisant de petits programmes simples, et pensez-y lorsque vous vous trouverez face à des algos qui peuvent se servir de telles structures.

N'hésitez pas à me poser des questions dans le topic dédié sur Planète Casio (pas par MP svp), je ferai mon possible pour vous répondre.

Si vous avez besoin du fichier source .tex de ce tutoriel, demandez-le moi par MP.

Copyleft

Tutoriel réalisé par Louis Sugy, sous licence *CC BY-NC-SA* : attribution, pas d'utilisation commerciale, et partage à l'identique.

Pages Wikipédia reliées

Si vous souhaitez approfondir le sujet, voilà les liens de quelques pages Wikipédia assez complètes à propos des notions abordées :

— Structure de données

— Piles
— Files

— Théorie des graphes
— Parcours en largeur