5. Listes chaînées

5.1. Introduction

En programmation, pour traiter des données du même type (par exemple, les informations des étudiants), nous avons besoin de tableaux. Les tableaux représentent un concept important dans tout langage de programmation car ils permettent un accès rapide à leurs éléments. Cependant, ils présentent deux inconvénients :

- Les éléments du tableau doivent être contigus en mémoire.
- Il n'est pas possible d'insérer ou de supprimer des éléments dans le tableau sans recréer le tableau à nouveau.

Nous avons donc besoin d'une autre structure de données connue sous le nom de **liste chaînée**.

5.2. Définition

Les listes chaînées sont une structure de données récursive composée de nœuds du même type, reliés les uns aux autres par des pointeurs. Contrairement aux tableaux, ces nœuds peuvent se trouver dans des emplacements non contigus de la mémoire. Les listes chaînées sont constituées d'éléments (enregistrements, nœuds ou cellule), et chaque élément contient un ou plusieurs champs pour stocker des données et un pointeur (lien) vers l'élément suivant de la liste.

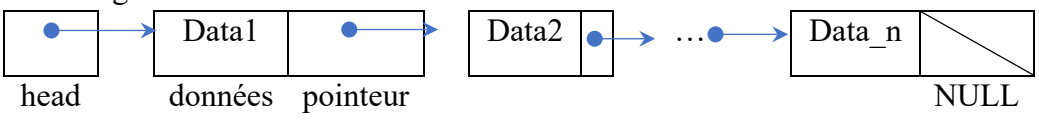
Cette structure permet de modifier sa dimension en insérant ou supprimant des éléments de n'importe quelle position dans la liste. Pour accéder à n'importe quel élément de la liste, il faut partir de son en-tête et parcourir tous les éléments qui le précèdent, ce qui peut prendre plus de temps que l'accès direct aux éléments d'un tableau. On dit donc que c'est une structure de données linéaire en opposition à la structure de tableau qui permet un accès aléatoire.

5.3. La représentation

En langage C, un nœud est représenté à l'aide de structures "struct", tandis qu'un en-tête est représenté par un pointeur.

Pour simplifier l'explication, nous utilisons un champ de données unique de type entier appelé "data" pour tous les enregistrements dans la liste, au lieu d'utiliser des champs de données spécifiques pour chaque type d'enregistrement (comme les informations sur les étudiants, le nom, le prénom, la date, etc.).

La figure suivante illustre la structure des listes chaînées :



5.4. La Déclaration

Déclaration du type des éléments ou des nœuds

C	Algorithme

`typedef struct Node {`
`  int data;`
`  struct Node * next;`
`} Node;`
	`Node structure`
	`  data:entier`
	`  next :^ Node`
	`fin_structure`

Dans la structure des listes chaînées, "data" représente les données stockées dans la liste, telles que le nom et le prénom d'un étudiant, la date d'un événement, etc. Ce champ peut être remplacé par toute autre variable correspondant au type de données que vous souhaitez stocker dans la liste.

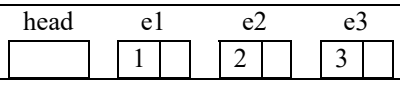
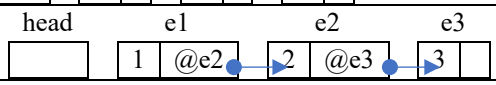
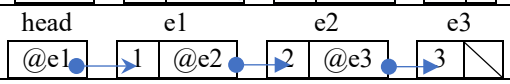
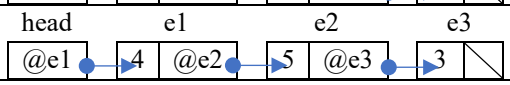
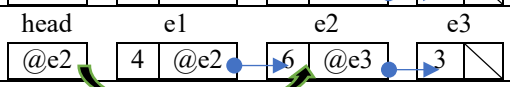
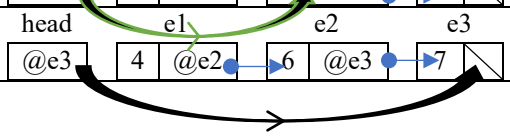
Le champ "next" est un pointeur qui contient l'adresse de l'élément suivant dans la liste, ou NULL s'il ne pointe sur aucun élément. Ce champ est important car il permet de relier les nœuds les uns aux autres pour former la liste chaînée.

Déclaration du type d'en-tête

`typedef Node* List;`	`type List: ^ Node`

Cela signifie que List est identique à Node *.

Exemple

`List head;`	`var head: List`	Une variable simple de type pointeur qui pointe sur le premier élément head
`Node e1, e2,e3 ;`	`e1, e2, e3:Node`	3 variables composées de type `Node`
`e1.data=1;e2.data=2;` `e3.data=3;`	`e1.data←1 e2.data←2` `e3.data←3`	
`e1.next=&e2;` `e2.next=&e3;`	`e1.next←@e2 e2.next←@e3`	
`e3.next= NULL;` `head=&e1;`	`e3.next← NULL` `head←@e1`	
`head->data=4;` `head->next->data=5;`	`head^.data←4` `(head^. next)^.data←5`	
`head= head->next;` `head->data=6;`	`head← head^.next;` `head^.data←6;`	
`head= head->next;` `head->data=7;`	`head← head^.next;` `head^.data←7;`	

L'opération -> en langage C

Étant donné que le pointeur 'head' pointe vers l'élément 'e1', la variable pointée par 'head' et 'e1' sont équivalentes, donc l'expression '(*head).next' peut être utilisée pour accéder au champ 'next' de l'élément pointé par 'head' au lieu de 'e1.next'

En langage C, nous utilisons l'opérateur '->' au lieu de '(*).' pour accéder aux champs de la structure pointée par 'head'. L'expression 'head->next' est donc équivalente à '(*head).next' pour accéder au champ 'next' de l'élément pointé par 'head'

`e1.next ⇔ (*head).next ⇔ head->next`

`e1.data=5; ⇔ (*head).data=5; ⇔ head->data=5;`

Notez que head.data est incorrect car head est un pointeur, pas une structure.

Les pointeurs 'head', 'e1.next', 'e2.next' et 'e3.next' sont tous de même type, ce qui signifie qu'il est possible de faire des affectations entre eux.

Le dernier élément

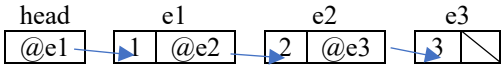
Le dernier élément de la liste n'a pas d'élément suivant, donc NULL est affecté à son pointeur 'next'. Lors du parcours de la liste, NULL est utilisé pour vérifier si le dernier élément a été atteint ou non.

```
e3->next= NULL;
```

```
head= NULL ; C'est une liste vide
```

Le parcours d'une liste chaînée

L'exemple suivant montre comment parcourir les éléments d'une liste chaînée.
Supposons que nous ayons la liste suivante :



Comme "e1.next" pointe vers "e2", on peut faire pointer "head" vers "e2" en effectuant l'opération suivante : "head = e1.next;".

Maintenant que "head" pointe vers "e2", alors "head->next" équivaut à "e2.next".

`head= &e2⇔ head= e1.next⇔ head= head->next`

Donc, pour passer d'un nœud à l'autre, on utilise « head=head->next »

`while (head!= NULL){`
`  //do something`
`  head = head->next;`
`}`
	`TQ (head≠NULL) faire`
	`  //faire qqs chose`
	`  head←head^.next`
	`FTQ`

Pour accéder à tous les éléments de la liste, nous répétons le processus jusqu'à ce que head prenne la valeur de next du dernier nœud, qui est NULL.

Observation: `while (head!= NULL) ⇔ while (head)`

5.5. La Création

Pour créer une liste chaînée, on réserve dynamiquement de la mémoire à partir d'une simple variable de type pointeur.

Supposons que nous ayons une liste vide avec head=NULL; Pour créer un nouvel élément, on utilise la fonction d'allocation dynamique de mémoire malloc().

`List e, head= NULL;`	`var e, head: List` `head← NULL`	head e	Deux listes sont créées (pointeur Node*)

`e = malloc(`
`sizeof(Node));`
`e->data=1;`
`e->next= NULL;`
	`allouer(e,1)`
	`e^.data←1`
	`e^.next←NULL`
		head e @ 1	Un nouvel élément « e » a été créé et ses champs ont été initialisés.

`head = e;`	`head ←e`	head e @ @ 1	Ici, « e et head » ont la même adresse, ce qui signifie qu'ils font référence au même élément.

`e = malloc(`
`sizeof(Node));`
`e->data=2;`
	`allouer(e,1)`
	`e^.data←2`
		1 2	Un nouvel élément « e » a été créé et peut être ajouté en tête de la liste en le reliant au premier élément de la liste grâce à son champ next.
			e->next=head; head=e;
			ou à la fin
			head->next=e

Remarque en C++

`e = malloc( sizeof(Node));⇔ e = new Node;`

6. Opérations sur les listes chaînées

Nous allons maintenant créer un ensemble de sous-programmes pour gérer les listes, tels que l'ajout ou la suppression d'un élément, l'affichage de tous les éléments de la liste, la recherche dans la liste, etc. Il est recommandé de regrouper toutes ces fonctions dans une bibliothèque dédiée aux listes.

Observation

Il existe plusieurs façons de créer des fonctions pour ajouter ou supprimer un élément de la liste :

- En utilisant des fonctions qui prennent une liste en paramètre et renvoient une liste. Dans ce cas, la liste peut être passée par valeur.
- En utilisant des procédures et un élément auxiliaire (sentinelle) pour éviter le passage par adresse. Dans ce cas, la liste peut être passée par valeur.
- En utilisant des procédures sans auxiliaire. Dans ce cas, la liste doit être passée par adresse.
- En utilisant des fonctions qui prennent une liste en paramètre et retournent une valeur booléenne (bool) pour nous dire si l'opération a réussi (true) ou non (false). Dans ce cas, la liste doit être passée par adresse. Nous utiliserons cette dernière méthode.

6.1. L'Affichage d'une liste

```
void display_list(List head) {
  while (head != NULL) {
    printf("%d->", head->data);
    head = head->next;
  }
  printf("fin\n");
}
```

```
procedure display_list(List head)
debut
  TQ (head ≠ NULL) faire
    ecrire(head->data,"->")
    head ← head^.next
  FTQ
  printf("fin")
fin
```

```
void display_list(List head) {
  if (head){
      printf("%d->", head->data);
      display_list(head->next);
  } else printf("fin\n");
}
```

Nous itérons à travers chaque élément de la liste et affichons les données associées. Nous notons ici que la liste a été passée par valeur, et donc la tête de la liste d'origine ne sera pas modifié si nous changeons la valeur de head, nous l'utilisons donc pour parcourir la liste sans danger.

6.2. Taille de la liste

Parcourez la liste et ajoutez 1 jusqu'à ce que nous arrivions à NULL

```
int size_list(List head){
  int n=0;
  while (head != NULL) {
    head= head->next;
    n++;
  }
  return n;
}
```

```
int size_list(List
head){
if (!head)  return 0;
return
1+ size_list(head-
>next);
}
```

```
fonction size_list(List head) :
entier
  var n:entier
debut
  n←0
  TQ (head ≠ NULL) faire
    n←n+1
    head ← head^.next
  FTQ
  size_list←n
fin
```

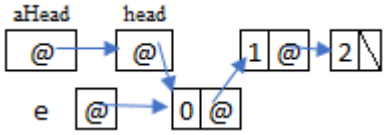
6.3. Ajouter un élément à la liste

Le processus d'ajout d'un élément à une liste chaînée se déroule en 3 étapes :

1. Créer et initialiser un nœud
2. Déterminer l'emplacement du nœud.
3. Ajoutez le nœud à la liste en réaffectant les pointeurs.

Ajouter un élément au début de la liste (en tête)

1. L'ajout d'un élément au début de la liste nécessite de changer la tête de la liste, il est donc important de passer la liste par adresse pour que cette modification soit visible dans l'appelant.
2. Créer un nouvel élément, et s'il échoue, renvoyer false à l'appelant
3. Initialiser l'élément
4. Remplacer « next » de « e » pour pointer vers le premier élément de la liste
5. Modifier la tête de la liste pour pointer vers le nouvel élément, « aHead et e » sont deux variables locales qui sont supprimées immédiatement après l'exécution de la procédure



`int add_head(List* aHead, int d){`	`fonction add_head(aHead:^List, d:entier) :bool`
	`  var e:List`
	`debut`
		1

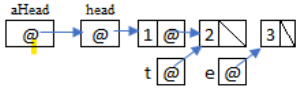
`  List e = malloc(sizeof(Node));`
`  if (e == NULL) {`
`  return 0;`
`  }`
	`  allouer(e,1)`
	`  si (e = NULL) alors`
	`  add_head ←faux`
	`  sinon`
		2

`  e-> data = d;`	`  e^.data←d`	3
`  e-> next = *aHead;`	`  e^.next←aHead^`	4

`  *aHead=e;`
`  return 1;`
`}`
	`  aHead^←e`
	`  add_head ←vrai`
	`  fsi`
	`fin`
		5

Ajouter un élément à la fin

1. Il est possible que nous ajoutions un élément dans l'en-tête, nous devons donc la passer par adresse. Créer un nouvel élément
2. Initialiser l'élément et affectez NULL au « next » car ce sera le dernier élément
3. Si la liste est vide, on va l'insérer dans la tête
4. Si la liste contient au moins un élément, on cherche le dernier élément
5. Insérer l'élément en dernier



`int append_end(List*aHead, int d){`
`  List t;`
`  List e = malloc(sizeof(Node));`
`  if (e == NULL) {`
`  return 0;`
`  }`
	`fonction append_end(aHead:^List, d:entier): bool`
	`  var e,t:List`
	`debut`
	`  allouer(e,1)`
	`  si (e = NULL) alors`
		1

	append_end←faux **else**	
e->data = d; e->next = NULL;	e^.data←d e^.next← NULL	2
if (*aHead == NULL) *aHead = e;	**si** (aHead^=NULL) **alors** aHead^←e	3

else {
t= *aHead;
while (t-> next != NULL)
t= t-> next;
	sinon
	t←aHead^
	TQ (t^.next≠NULL) **faire**
	t←t^.next
	FTQ
		4

t-> next=e;
}
return 1;
}
	t^.next←e
	fsi
	append_end←vrai
	fsi
	fin
		5

6.4. Supprimer un élément de la liste

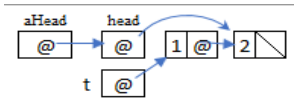
Le processus de suppression d'un nœud d'une liste se déroule en 4 étapes :

1. Déterminez le nœud précédent du nœud à supprimer.
2. Garder l'adresse du nœud à supprimer dans une variable
3. Connectez le nœud précédent au nœud suivant du nœud à supprimer.
4. Videz la mémoire réservée par le nœud à supprimer.

Il y a donc 3 cas, soit la liste est vide, contient un seul élément, ou contient plus d'un élément.

Supprimer l'élément du début (la tête)

1. Il est possible que nous supprimions un élément de l'en-tête, nous devons donc passer la liste par adresse
2. Si la liste est vide, il n'y a pas d'élément à supprimer, donc on renvoie false.
3. Stocke l'adresse du premier élément à supprimer dans t
4. Connecter la tête avec le deuxième élément
5. Vider la mémoire réservée par le premier élément

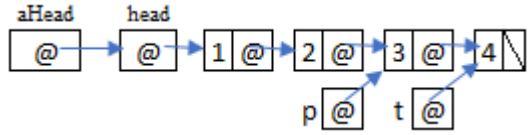


int delete_head(List*aHead) { List t;	**function** delete_head(aHead:^List): bool **var** t:List **debut**	1
if (*aHead== NULL) **return** 0;	**si** (aHead^ ==NULL) **alors** delete_head← faux **sinon**	2
t = *aHead;	t←aHead^	3
*aHead =t-> next;	aHead← t^.next	4

free(t);
return 1;
}
	desallouer(t)
	delete_head←vrai
	fsi
	fin	

Supprimer un élément de la fin

1. Il est possible que nous supprimions un élément de la tête, nous devons donc passer la liste par adresse. t contient le dernier élément et p est l'avant-dernier élément
2. Si la liste est vide, il n'y a pas d'élément à supprimer, donc on retourne false
3. Si la liste ne contient qu'un seul élément, supprimez-le directement de la tête
4. Si la liste contient plus d'un élément, on cherche le dernier élément t et l'avant-dernier p



5. On affecte NULL à « next » de l'avant-dernier p, car il est devenu le dernier et on supprime le dernier t de la mémoire.

`int delete_end (List*aHead) {` `  List t, p;`	**function** delete_end(aHead:^List): bool **var** t, p:List **debut**	1
if (*aHead== NULL) **return** 0;	**si** (aHead^ ==NULL) **alors** delete_end ←faux **sinon**	2

if ((*aHead)->next ==NULL) {
free(*aHead);
*aHead = NULL; }
	si (aHead^.next =NULL) **alors**
	desallouer(*aHead)
	*aHead←NULL
		3

else {
t = *aHead;
while (t->next != NULL) {
p=t ;
t= t->next;
}
	sinon
	t←aHead^
	TQ (t^.next ≠ NULL) **faire**
	p←t
	t←t^.next
	FTQ
		4

p->next=NULL;
free(t);
}
return 1;
}
	p^.next←NULL
	desallouer(t)
	fsi
	delete_end ←vrai
	fsi
	fin
		5

6.5. Supprimer la liste

1. Nous supprimons de l'en-tête jusqu'à ce que la liste devienne vide
2. Ou en utilisant la fonction delete_head jusqu'à ce qu'elle renvoie false

void delete_list(List*aHead) {
List t;
while(*aHead!= NULL) {
t = *aHead;
*aHead =t-> next;
free(t);
}
}
	procedure delete_ list(aHead:^List)
	var t:List
	debut
	TQ (aHead^ ≠NULL) **faire**
	t←aHead^
	aHead^←t^.next
	desallouer(t)
	FTQ
	fin
		1

```
void delete_list(List*aHead) {
  while (delete_head(aHead));
}
```

```
procedure delete_ list(aHead:^List)
debut
  TQ (aHead^ ≠NULL) faire
    delete_head(aHead)
  FTQ
fin
```

2

6.6. Programme principal (utilisation)

```
int main() {
  List head =NULL;
  add_head(&head, 3);
  add_head(&head, 2);
  append_end(&head, 4);
  add_head(&head, 1);
  append_end(&head, 5);
  printf("size=%d\n", size_list(head));
  display_list(head);
  delete_head(&head);
  delete_end(&head);
  printf("size=%d\n", size_list(head));
  display_list(head);
  delete_list(&head);
  printf("size=%d\n", size_list(head));
  display_list(head);
  return 0;
}
```

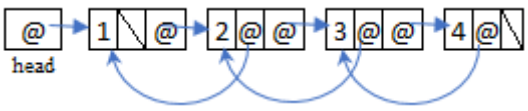
```
debut
  add_head(@head, 3)
  add_head(@head, 2)
  append_end(@head, 4)
  add_head(@head, 1)
  append_end(@head, 5)
  ecrire("size=", size_list(head))
  display_list(head)
  delete_head(@head)
  delete_end(@head)
  ecrire("size=", size_list(head))
  display_list(head)
  delete_list(@head)
  ecrire("size=", size_list(head))
  display_list(head)
fin
```

- Le programme affichera
  size=5
  1->2->3->4->5->fin
- Puis il affichera
  size=3
  2->3->4->fin
- A la fin il affichera
  size=0
  fin

7. Liste doublement chaînée (Double linked list)

En plus des données et du pointeur qui pointe vers l'élément suivant, une liste doublement liée contient un autre pointeur, généralement appelé "prev", qui pointe vers l'élément précédent. Ce pointeur facilite le parcours de la liste dans les deux sens et simplifie ainsi le processus de suppression ou d'insertion d'un élément avant celui sélectionné.

La figure suivante montre la structure d'une liste doublement chaînée



7.1. Déclaration

```
typedef struct Node {
```

```
Node structure
```

int data; **struct** Node* next, * prev; } Node;	data:entier next, prev :^Node **fin_structure**	

« data » représente les données stockées dans la liste. "next" est un pointeur qui contient l'adresse de l'élément suivant, tandis que "prev" est un pointeur qui contient l'adresse de l'élément précédent.
Les opérations d'ajout et de suppression sont les suivantes

7.2. Ajouter un élément au début (tête)

e-> next = *aHead;	e^.next←aHead^	1
e-> prev = NULL; **if**(*aHead!= NULL) (*aHead)->prev = e;	e^.prev←NULL **si**(*aHead ≠ NULL)**alors** aHead^.prev←e **fsi**	2
*aHead=e;	aHead^←e	3

1. Changer « next » de « e » pour qu'il pointe vers le premier élément
2. Il pointe vers NULL (premier élément)
   Le « prev » du premier élément, s'il existe, pointe vers le nouvel élément.
3. Modifier la tête de la liste pour pointer vers le nouvel élément

7.3. Ajouter un élément à la fin

e-> next = NULL;	e^.next← NULL	1

if (*aHead == NULL) {
e-> prev = NULL;
*aHead = e;
}
	si (aHead^=NULL) **alors**
	e^.prev←NULL;
	aHead^←e
		2

else {
t= *aHead;
while (t-> next != NULL)
t= t-> next;
	sinon
	t←aHead^
	TQ (t^.next≠NULL) **faire**
	t←t^.next
	FTQ
		3

e-> prev = t;
t-> next=e;
}
	e^.prev←t;
	t^.next←e
	fsi
		4

1. NULL car ce sera le dernier élément
2. Dans le cas où la liste est vide, elle est ajoutée dans l'en-tête, tandis que prev indique NULL
3. Si la liste contient au moins un élément, le dernier élément est recherché
4. Le prev du nouvel élément fait référence au dernier élément de la liste. Insérer l'élément en dernier

7.4. Supprimer l'élément du début (la tête)

1. Stocke l'adresse du premier élément à supprimer
2. Se lier avec le deuxième élément . Si la liste n'est pas vide, le « prev » du premier élément doit être NULL
3. Vider la mémoire réservée par le premier élément

t = *aHead;	t←aHead^	1
*aHead =t-> next;	aHead← t^.next	2

`if(*aHead!= NULL)` `  (*aHead)->prev = NULL;`	**si**(*aHead ≠ NULL)**alors** aHead^.prev← NULL **fsi**	
`free(t);`	`desallouer(t)`	3

7.5. Supprimer un élément du dernier

1. Dans le cas où la liste contient plus d'un élément, le dernier élément t est recherché, et il n'est pas nécessaire de sauvegarder l'avant-dernier car il est accessible.
2. On récupère l'avant-dernier au moyen de « prev » du dernier t.
   Nous attribuons NULL à « next » de l'avant-dernier car il est devenu le dernier.
   Nous supprimons le dernier t de la mémoire

`t = *aHead->next;`
while `(t->next != NULL)`
`  t= t->next;`
	t←aHead^
	TQ (t^.next ≠ NULL) **faire**
	  t←t^.next
	FTQ
		1

`p=t->prev;`
`p ->next=NULL;`
`free(t);`
	p←t^.prev
	p^.next←NULL
	desallouer(t)
		2

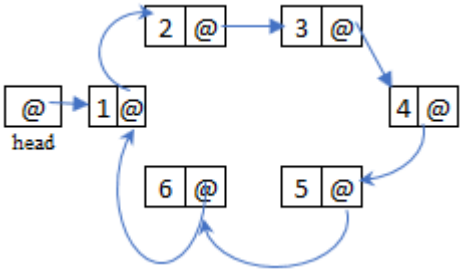
7.6. Remarque :

Le « prev » du premier élément peut être utilisé pour faire référence au dernier élément de la liste, ce qui accélère le processus d'accès au dernier élément pour l'ajout ou la suppression.

8. Listes chainées spéciales

En plus des listes simples et doubles liées, il existe des listes circulaires simples et doubles liées

La liste circulaire est une liste chaînée normale, sauf que le dernier élément ne porte pas NULL, mais fait référence au premier élément de la liste, comme c'est le cas dans la double liste circulaire, où le « next » du dernier élément fait référence au premier élément et le « prev » du premier élément au dernier élément.

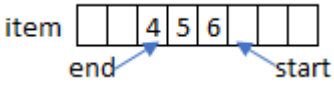


8.1. Les files (Queue)

La **file** est une structure de données abstraite qui permet de stocker un ensemble d'enregistrements du même type. Elle offre deux opérations essentielles : l'ajout d'un nouvel élément, également connu sous le nom d'insertion (en anglais : enQueue, en français : enfiler), et la suppression d'un élément, connue sous le nom de suppression (en anglais : deQueue, en français : défiler). Cette structure respecte la propriété FIFO (First In First Out), ce qui signifie que le premier élément ajouté est le premier élément à être supprimé. En d'autres termes, l'ordre de sortie est identique à l'ordre d'entrée.

Exemple : liste d'événements, file d'attente, liste des fichiers envoyés à l'imprimante...

Une file peut être implémentée à l'aide d'un tableau en utilisant deux indices pour suivre la position de la tête (ou "start" en anglais) et de la queue (ou "end" en anglais). Lorsqu'un élément est ajouté à la



file, il est placé à la position de la queue et l'indice de la queue est incrémenté. De même, lorsqu'un élément est supprimé de la file, il est retiré de la position de la tête et l'indice de la fin est également incrémenté. Si la fin du tableau est atteinte, on peut revenir au début du tableau pour continuer à ajouter des éléments si des emplacements sont disponibles, ou bien on peut allouer un nouveau tableau de taille supérieure et y copier les éléments existants.

8.1.1. Utilisation du tableau

1. Déclaration : une structure est créée qui contient une table d'éléments allouée dynamiquement en mémoire, un emplacement de début « start » pour l'ajout, un emplacement « end » pour la suppression et « capacity» qui contient le nombre d'éléments qu'on peut ajouter dans la table.
2. init() : La table est créée et la valeur -1 est affectée au start et à end pour indiquer que la file est vide. Si le processus de création échoue, la fonction renvoie false.
3. isEmpty() : La file est vide si « start » et « end » vaux -1.
4. isFull() : La file est pleine si la valeur de « start+1 » est la même que la valeur de « end », et nous utilisons **mod** «**%** » si nous atteignons la fin de la table pour la ramener au début.
5. enQueue() : s'assure que la file n'est pas pleine, puis ajoute 1 à start et ajoute x à la table.
6. deQueue() : renvoie le premier élément du tableau vers lequel pointe end, et ajoute 1 à end. Dans le cas où la file est vide, elle informe l'utilisateur.

Remarque :

Normalement, lorsqu'une fonction rencontre une situation d'erreur ou de comportement inattendu, elle n'est pas censée retourner une valeur qui pourrait être erronée ou mal interprétée par la fonction appelante. Au lieu de cela, elle doit lever une exception (erreur) qui sera interceptée et gérée dans la fonction appelante ou dans une autre fonction de la pile d'appels. En signalant explicitement l'erreur, l'exception permet d'identifier la source du problème et de faciliter la recherche et la résolution de celui-ci.

En C++ on peut écrire

```
if(isEmpty(Q)) throw -1;
```

typedef struct Queue{
  int *item;
  int start, end, capacity;
} Queue;
	Queue **structure**
	  item:^entier
	  start, end, capacity: entier
	fin_structure
		1

bool init(Queue *Q, **int** capacity) {
  Q->start = -1;
  Q->end = -1;
  Q->capacity= capacity;
  Q->item=
(**int***)malloc(**sizeof**(**int**)*capacity);
  return Q-> item != NULL;
}
	fonction
	init(Q:^Queue,capacity:entier):bool
	debut
	  Q^.start ←-1
	  Q^.end ←-1
	  Q^.capacity = capacity;
	  allouer(Q^.item, capacity)
	  init← Q^.item ≠ NULL
	fin
		2

bool isEmpty(Queue Q){
  return Q.start ==-1 **&&** Q.end==-1;
}
	fonction isEmpty(Q: Queue):bool
	debut
	  isEmpty← Q.start =-1 **et** Q.end=-1
	fin
		3

bool isFull(Queue Q){
 return
  (Q.start+1)% Q.capacity == Q.end;
}
	fonction isFull(Q: Queue):bool
	debut
	 isFull←
	  (Q.start+1) **mod** Q. capacity = Q.end
	fin
		4

void enQueue (Queue Q,int x){
 if(isFull(Q)){
  printf("error: Queue is full");
  return;
 }
 if(isEmpty(Q))
  Q.start=Q.end=0;
 else
  Q.start= (Q.start +1)% Q.capacity;
 Q.item[Q.start]=x;
}
	procedure enQueue(Q:Queue, x:entier)
	debut
	 si(isFull(Q))**alors**
	  ecrire("error: Queue is full")
	 sinon
	  si isEmpty(Q) alors
	  Q.start←0 Q.end←0
	  sinon
	  Q.start←(Q.start+1) **mod** Q.capacity
	  fsi
	  Q.item[Q.start] ←x
	 fsi
	fin
		5

int deQueue(Queue Q){
 if(isEmpty(Q)){
  printf("error: Queue is empty");
  return;
 }
 int x= Q.item[Q.end];
 if (Q.start == Q.end)
  Q.start = Q.end= -1;
 else
  Q.end= (Q.end+1)% Q.capacity;
 return x;
}
	fonction deQueue(Q: Queue) : entier
	debut
	  si(isEmpty(Q))**alors**
	  ecrire("error: Queue is empty")
	  deQueue ←-1
	  sinon
	  deQueue←Q.item[Q.end]
	  si Q.start = Q.end **alors**
	  Q.start←-1 Q.end←-1
	  sinon
	  Q.end←(Q.end+1) **mod** Q.capacity
	  fsi
	  fsi
	fin
		6

8.1.2. Utilisation de listes chainées :

Une implémentation simple de file à l'aide de tableaux peut entraîner des problèmes de performance si la file est de taille importante ou si elle est utilisée de manière intensive, car chaque insertion ou suppression peut nécessiter de déplacer tous les éléments restants dans le tableau pour maintenir la propriété FIFO. Pour éviter ces problèmes, il est préférable d'utiliser des structures de données plus performantes telles que des listes chaînées.

Pour simuler une file à l'aide de listes, il est nécessaire d'ajouter et de supprimer des éléments à deux extrémités différentes de la liste. Par exemple, on peut ajouter de nouveaux éléments à la fin de la liste et supprimer les éléments au début de la liste. Cette approche peut également être inversée, en ajoutant des éléments au début de la liste et en supprimant les éléments à la fin. Dans les deux cas, la structure de liste permet des insertions et des suppressions rapides et efficaces, sans nécessiter de déplacements coûteux d'éléments comme avec l'implémentation à l'aide de tableaux.

1. Déclaration : Nous créons une structure qui contient deux champs, le premier fait référence au premier élément de la liste et le second au dernier élément de la liste.
2. initQ(): en affectant null aux first et last.
3. isEmpty(): Une pile vide est une liste vide.
4. enQueue():est la même que la fonction « append_end », et pour éviter de parcourir tous les éléments de la liste pour arriver au dernier élément, on enregistre toujours l'adresse du dernier élément dans Q.last. Dans le cas où la liste est vide, on ajoute le nouvel élément au start et last, mais s'il n'est pas vide, on colle le nouvel élément avec le dernier élément, puis on change last pour qu'il pointe vers le nouvel élément.
5. deQueue():est identique à la fonction « delete_head » sauf que la fonction deQueue renvoie l'élément qui a été supprimé. Donc, avant de supprimer l'élément t, elle sauvegardons t-> data dans x, puis le supprime et renvoie la valeur de x.

typedef struct Queue{
  struct Node***** first, *****last;
  int size;
} Queue;
	Queue **structure**
	  first, last:^Node
	  size :entier
	fin_structure
		1

Queue *initQ(){
 Queue *Q=**new** Queue;
 Q->first =NULL;
 Q->last =NULL;
 Q->size =0;
 return Q
}
	procedure initQ (Q: ^Queue)
	debut
	  Q^.first← NULL
	  Q^.last← NULL
	fin
		2

bool isEmpty(Queue *Q){
  return Q->size==0;
}
	fonction isEmpty(Q: Queue):bool
	debut
	  isEmpty← Q.first= NULL
	fin
		3

void enQueue(Queue *Q, **int** x) {
  Node *e = malloc(**sizeof**(Node));
  e->data = x;
  e->next = NULL;
  if(isEmpty(Q))
  Q->first =e;
  else
  Q->last->next =e;
  Q->last =e;
  Q->size++;
}
	procedure enQueue(var Q: Queue, x:entier)
	  var e:^Node
	debut
	  allouer(e,1)
	  e^.data← x
	  e^.next← NULL
	  si (isEmpty(Q)) **alors**
	  Q.first←e
	  sinon
	  Q.last^.next←e
	  fsi
	  Q.last←e
	fin
		4

```
int deQueue(Queue *Q) {
  Node* t; int x;
  if(isEmpty(Q)){
    printf("error: Queue is empty");
    exit(1);
  }
  t = Q->first;
  x = t ->data;
  Q->first = t-> next;
  free(t);
  return x;
}
```

```
fonction deQueue(var Q: Queue): entier
  var t:^Node
début
  si (isEmpty(Q)) alors
    printf("error: Queue is empty");
    deQueue←-1
  sinon
    t← Q.first
    deQueue ← t^.data
    Q.first← t^.next
    desallouer(t)
  fsi
fin
```

5

8.2. Les piles (Stack):

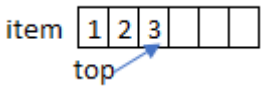
Il s'agit d'une structure de données abstraite constituée d'un ensemble d'enregistrements de même type, dans laquelle seules deux opérations peuvent être effectuées : l'ajout d'un nouvel élément, ce processus est appelé push (ou empiler en français), et la suppression d'un élément du groupe, ce processus est connu sous le nom de pop (ou dépiler en français), et ces opérations ont lieu à une seule extrémité du groupe appelée le sommet (ou top en anglais). Cette structure de données a la caractéristique de LIFO (Last In First Out), c'est-à-dire que le dernier élément ajouté est le premier à être supprimé, et l'ordre de sortie est donc l'opposé de l'ordre d'entrée.

Exemple :

Les piles sont couramment utilisées dans des situations où les données doivent être traitées dans un ordre inversé à celui dans lequel elles ont été reçues, telles que

- La gestion de la mémoire dans les systèmes informatiques
- Le navigateur Web enregistre la liste des pages visitées dans une pile.
- La liste des opérations dans Word, par exemple, est stockée dans une pile et est utilisée pour annuler les modifications.

Les piles peuvent être implémentées à l'aide de tableaux ou de listes chaînées, mais les listes chaînées sont souvent préférées car elles offrent des performances plus prévisibles lors de l'ajout ou de la suppression d'éléments.



8.2.1. Utilisation du tableau

1. Déclaration : Une structure est créée qui contient une table d'éléments item allouée dynamiquement, et top l'endroit d'ajout ou de suppression et capacity qui représente la taille.
2. init(): La table est créée et la valeur 0 est affectée à top pour indiquer que la pile est vide.
3. isEmpty():La pile est vide si la valeur de top est 0.
4. isFull(): Si le tableau est plein, top est égal à capacity.
5. Pop(): La fonction Pop permet de décrémenter 'top' et de renvoyer le dernier élément qu'il pointe.
6. Push (): La fonction Push permet d'ajouter l'élément x à la table et d'incrémenter de 1 le pointeur 'top'. Vous devez vous assurer que la pile (table) n'est pas pleine.

```
typedef struct Stack{
  int *item;
  int top, capacity;
} Stack;
```

```
Stack structure
  item:^entier
  Top, size:entier
fin_structure
```

1

bool init(Stack *s, int capacity) {
  s->top = 0;
  s-> capacity = capacity;
  s->
>item=(int*)malloc(sizeof(int)*size);
  return s-> item != NULL;
}
	fonction init(s:^ Stack, capacity:entier):bool
	debut
	  s^.top←0
	  s^. capacity ← capacity
	  allouer(s^.item, capacity)
	  init← s^.item ≠ NULL
	fin
		2

bool isEmpty(Stack s){
  return s.top==0;
}
	fonction isEmpty(s:Stack):bool
	debut
	  init←s.top=0
	fin
		3

bool isFull(Stack s){
  return s.top==s.capacity;
}
	fonction isFull(s: Stack):bool
	debut
	  isFull←s.top= s.capacity
	fin
		4

int Pop(Stack s){
  int x;
  if(isEmpty(s)){
  printf("error: Stack is empty");
  exit(1);
  }
  s.top--;
  x=s.item[s.top];
  return x;
}
	fonction Pop(s: Stack) : entier
	debut
	  si(isEmpty(s))**alors**
	  écrire("error: Stack is empty");
	  Pop←-1
	  sinon
	  s.top← s.top -1
	  Pop←s.item[s.top]
	  fsi
	fin
		5

void Push(Stack s,int x){
  if(isFull(s)){
  printf("error: Stack is full");
  exit(1);
  }
  s.item[s.top]=x;
  s.top++;
}
	procedure Push(s: Stack, x: entier)
	debut
	  si (isFull(s)) **alors**
	  écrire("error: Stack is full");
	  sinon
	  s.item[s.top] ←x
	  s.top← s.top +1
	  fsi
	fin
		6

8.2.2. En Utilisant des listes chainées :

Pour simuler une pile à l'aide de listes, l'ajout et la suppression doivent se faire du même côté (au début ou à la fin).

1. isEmpty():Une pile vide est une liste vide.
2. Pop(): La fonction pop est la même que la fonction delete_head sauf que la fonction pop renvoie l'élément qui a été supprimé. Donc, avant de supprimer le premier élément t, nous sauvegardons t-> data dans x, puis le supprimons et renvoyons la valeur de x.
3. Push ():La fonction push est la même que la fonction add_head

bool isEmpty(List head){
  return head==NULL;
}
	fonction isEmpty(head: List):bool
	debut
	  isEmpty ← head= NULL
	fin
		1

`int Pop(List*aHead) {`
`  List t; int x;`
`  if(*aHead==NULL){`
`  printf("error: Stack is empty");`
`  exit(1);`
`  }`
`  t = *aHead;`
`  *aHead =t-> next;`
`  x=t->data;`
`  free(t);`
`  return x;`
`}`
	`fonction Pop(aHead:^List): entier`
	`  var t:List`
	`début`
	`  si(isEmpty(aHead^))alors`
	`  écrire("error: Stack is empty");`
	`  Pop←-1`
	`  sinon`
	`  t←aHead^`
	`  aHead← t^.next`
	`  Pop← t^.data`
	`  desallouer(t)`
	`  fsi`
	`fin`
		2

`void Push(List* aHead, int x) {`
`  List e = malloc(sizeof(Node));`
`  e-> data = x;`
`  e-> next = *aHead;`
`  *aHead=e;`
`}`
	`procedure Push(aHead:^List, x:entier)`
	`  var e:List`
	`debut`
	`  allouer(e,1)`
	`  e^.data← x`
	`  e^.next←aHead^`
	`  aHead^←e`
	`fin`
		3