**M. BATABE**  **CHAP- 1 DEFINITIONS**

1. **Donnée**

Information numérique et alphanumérique, codée et lisible par la seule machine, en vue de leur enregistrement, traitement, conservation et communication**.**

Exemple**:** On décide de stocker des données définissant des personnes dans une BD. On définira les données de l’entité personne comme ceci :

* Numero : 001255
* Nom : LAWSON
* Prenom : Afi
* Date\_naiss :12/09/1990
* Sexe : F

**001255, LAWSON , Afi ,F, 12/09/1990 constituent les données permettant de caractériser mademoiselle LAWSON .**

**Pour des traitements informatiques concernant plusieurs personnes, il faudra le choix d’une structure pour stocker toutes les infos de ces personnes.**

1. **Structure**

**Une SD est une manière d’organiser et de stocker l’information pour en faciliter l’accès ou d’autres buts.**

La structure est la disposition des parties d’un tout, constituant une sorte d’architecture. La structure, c’est un arrangement ou une disposition.

En informatique, il existe plusieurs types de structure que nous appelons : structure de données et structure de données avancées.

1. **Structure de donnée**

**C’est une interface qui consiste en un ensemble de procédures pour ajouter, effacer, accéder, réorganiser,… les données.**

**Une SD conserve les données et éventuellement les métadonnées.**

**Un type de données abstrait(TDA)= définition des propriétés de la structure et de son interface.**

1. **Structure de données en Informatique**

En informatique, une structure de données est une structure logique destinée à contenir des données afin de les donner une organisation permettant de simplifier leur traitement.

**Une SD implémente concrètement un type abstrait**

1. **Objectifs de l’organisation des données**

Pour prendre un exemple de la vie quotidienne, on peut présenter des numéros de téléphone par département, par nom, profession, par numéro téléphonique comme dans les annuaires téléphoniques, par rue et une combinaison quelconque de ses classements.

En organisant d’une certaine manière des données, on permet un traitement automatique de ces derniers plus efficace et plus rapide.

Le fait d’utiliser une structure de donnée appropriée à un traitement informatique, peut également faire baisser de manière significative la complexité **d’une application informatique et ainsi contribuer à diminuer le taux d’erreurs.**

\*Exemple de structure de données :

Différentes structures de données existent pour des données différentes ou répondant à des contraintes algorithmiques différentes.

* **Structures finies** :
* Constantes
* Variables
* Enregistrements
* **Structures indexées** :
* Tableau sur [1………..n]
* Tableau sur [1………..n] [1………..m]
* **Structures récursives** :
* Listes
* Arbres
* Graphes

1. **Les types de collection**

* **Collection séquentielle**

Elle permet de ranger les objets dans un ordre arbitraire. On parle de collection indexée quand on peut accéder à chaque élément de la collection par un numéro d’ordre ou indexe.

Le choix d’une implémentation particulière dépend d’un certain nombre de compromis, comme l’occupation de la mémoire ou les performances requises pour diverse opération de base :

* Itération
* Indexation
* Suppression
* Ajout d’un élément **(au début, à la fin ou encore dans un emplacement quelconque de la collection**)
* Décompte du nombre d’élément

Il existe deux grands types de collections séquentielles :

* Les listes
* Les tableaux **ou vecteurs**

Un certain nombre de structures de données sont des restrictions de collections séquentielles qui n’autorisent qu’un sous-ensemble des opérations de base :

* Piles
* Files

Les arbres constituent une autre représentation de données.

1. **Collection de données**

Une collection est un regroupement fini de donnée dont le nombre n’est pas fixé à priori. Ainsi les collections que l’on utilise en informatique sont des objets dynamiques. Le nombre de leur élément varie au cours de l’exécution du programme, puisqu’on peut y ajouter et supprimer des éléments en cours de traitement. Plus précisément, les principales Opérations que l’on s’autorise sur les collections sont les suivantes :

* Déterminer le nombre d’éléments de la collection
* Tester l’appartenance d’un élément à la collection
* Ajouter un élément à la collection
* Supprimer un élément de la collection

**Exemple de collection :**

* **Tas de chaussures**

Dans le tas de chaussures, il est très facile d’ajouter une paire de chaussure sur le tas : il suffit de la jeter sans précautions sur les autres chaussures. Par contre pour supprimer une chaussure particulière du tas, ce sera beaucoup plus difficile car il faudra d’abord la retrouver dans cet amas non structuré de chaussures.

A l’inverse du tas de chaussure, les collections informatiques seront structures pour faciliter et optimiser la recherche d’un élément en leur sein. Les éléments peuvent être de différentes types, mais les Operations que l’on effectue sur les collections doivent être indépendant des types de donnée des éléments.

En général, on distingue trois grands types de collection :

* Les séquences
* Les arbres et
* Les graphes

1. **SEQUENCES**

Une séquence est une suite ordonnée d’éléments éventuellement vide, accessible par leur rang dans la séquence.

Dans une séquence, chaque élément a un prédécesseur (sauf le premier élément qui n’a pas de prédécesseur) et un successeur (sauf le dernier élément qui n’a pas de successeur).

Une liste de nom, une main d’un jeu de carte, une pile d’assiette ou fille de spectateur sont des exemples de structure séquentielle de la vie courante.

En langage Python,on utilisera 3 types de séquences :

-Les chaînes de caractères (type str , exemple :`’,`bonjour’, ‘’ça va’’ )

-Les n-uplets (type tuple , exemple : (),(1,2,3),(`a’,2,(1,2,3)))

-Les listes (type list, exemple : [ ],[a,b,c,d,e],[1,`e’,(1,2,[x,y])])

1. **LES ARBRES**

Un arbre est une collection d’élément appelé nœud, organisé de façon hiérarchique à partir d’un nœud particulier appeler la racine de l’arbre.

Dans un arbre, chaque élément a un prédécesseur (sauf la racine de l’arbre qui n’a pas de prédécesseur et peut avoir plusieurs successeur). L’arbre est dit **n-aire** si chaque nœud a au plus **n successeur.** (Voir le chapitre des arbres et leurs caractéristiques)

1. **LES GRAPHES**

Un graphe est une collection d’élément appelé sommet et de relation entre ses sommets. Dans un graphe, chaque élément peut avoir plusieurs prédécesseurs et plusieurs successeurs. Un même élément peut être à la fois prédécesseur et successeur d’un autre sommet (y compris de lui-même). On parle de graphe oriente lorsque les relations entre sommets sont des paires ordonnées de sommets (les relations sont alors appelé **arc du graphe)**. On parle de graphe non oriente si ce sont des paires non orienté (les relations sont alors appelé **arrête du graphe**).

Un graphe facilement représentable par un schéma, ou les sommets sont des points et les arcs des flèches entre deux points (ou les arrête des traits entre deux points).

Un circuit électronique, une toile d’araignée ou l’internet sont des exemples du graphe. Les graphes permettent plus facilement de manipuler les objets et leurs relatons. L’ensemble des techniques et outils mathématiques mis au point en théorie des graphes, permettent de démontrer certaines propriétés des graphes, d’en déduire des méthodes de résolution des algorithmes.

**EXEMPLE : Kpalimé Bassar**

**Lomé Atakpamé Sokodé Kara**

**Alédjo**

**Notsé**

**A C E**

**B D**

1. **CHAINE DE CARACTERES**

Une chaine de caractères est séquence non modifiable de caractère. D’un point de vue syntaxique, une chaine caractère est une suite quelconque de caractère délimitée soit par des apostrophes soit par des Guillaumet. On peut ainsi utiliser des Guillaumet pour délimite des chaines contenant des apostrophes. **‘’**C’est ça ! La vie**’’.**

Exemple**:** Mettre des séquences de mot dans les variables et à l’aide de la syntaxe algorithmique, retrouver les paragraphes énumères

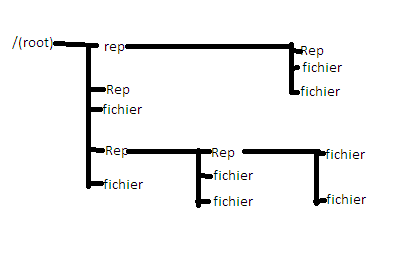
**Lili-bela-bien-chez-notre-oncle-ferdinan-nestor-napolieon-manga-allegrement-six-poulet-sans-claquer-argon-**

**Solution:**

A= ‘’ **Lili** ’’ B= ‘’ **bela** ’’ C= ‘’ **bien** ’’ D= ‘’ **chez**’’ E= ‘’ **notre**’’ F= ‘’ **oncle** ’’ G= ‘’ **ferdinan**’’ H= ‘’ **nestor**’’ I= ‘’ **napoleon** ’’ J= ‘’ **manga**’’ K= ‘’ **allegrement** ’’ L= ‘’ **six**- **poulet**’’ M= ‘’ **sans-claquer**’’ N= ‘’**argon**’’ O= ‘’**-**‘’

**CHAP2-STRUCTURE ARBORESCENTE**

Nous avion appris qu’il n’y a plus de notion de disque sous **UNIX**, il en est de même pour les périphériques ; de façon général, tous est fichier. On ne voit donc qu’au niveau utilisateur, une seule arborescence constitué de répertoires et fichiers décrivant les ressources du système. Le nom d’un fichier sous **Unix** est une suite de caractères. Il n’existe pas de notion de type de fichier et de numéro de version comme sous **MS-DOS**. Le caractère «. » est considère comme étant un caractère du nom du fichier et non pas comme sous **MS-DOS** le séparateurs entre le nom et l’extension. Il est donc possible d’avoir un fichier donc le nom comporte plusieurs points. Un répertoire est un fichier et il en contient aussi. La philosophie des noms de fichier ressemblera donc à celle de l’environnement suivante :



1. **Arbre binaire**

En informatique, un arbre binaire est une structure de donnée qui se présente sous la forme d’une hiérarchie dont chaque élément est appelé nœud. Le nœud de départ étant appelle **racine** de l’arbre. Dans un arbre binaire chaque élément à au plus deux éléments fixe qui sont au niveau inférieur, on parlera de gauche et de nœud droit. Du point de vue de ces éléments fixe, l’élément dont ils sont issu au niveau supérieur est appelé nœud père.

Nous pouvons retenir que pour un arbre binaire, au niveau le plus élevé il Ya un nœud racine, au niveau inférieur il y a au plus deux nœud fils. En continuant à descendre au niveau inférieur on peut en avoir 4, 8,16 … c’est-à-dire la suite des puissances de 2.

Un nœud n’ayant aucun fils est appelé **feuille**. Le nombre de niveau totale autrement dit la distance entre la feuille la plus éloigné et la racine est appelé **hauteur** de l’arbre. Le niveau d’un nœud est appelé **profondeur**.

Exemple :

HA=4 P=Profondeur

P(6)=2 H=Hauteur

P(15)=4

1

2 3

4 5 6 7

8 9 10 11 12 13 14

15

On peut donc conclure que : la profondeur de la feuille la plus éloigné est égale à la Hauteur de l’arbre.

1. **Type d’arbre Binaire**

Il existe trois types d’arbre binaire :

* Un arbre binaire ou binaire unaire est un arbre avec racine dans lequel chaque nœud au plus deux fils.

Exemple :

1

2 3

4 5 6 7

8 9 10 11 12 13 14

15

* Un arbre binaire entier est un arbre où tous les nœuds possèdent zéro ou deux fils.

Exemple :

A

B C

D E F G

H I

* Un arbre binaire parfait est un arbre entier dans lequel toutes les feuilles sont à la même distance de la racine.

Exemple :

1

2 3

4 5 6 7

8 9 10 11 12 13 14 15

1. **Méthodes pour stocker les arbres binaires**

Les arbres binaires peuvent être construits à partir de primitive d’un langage de programmation. Dans un langage avec structure et pointeur, les arbres binaires peuvent être construits avec une structure à trois nœuds qui contiennent quelques données et pointeur vers son fils droit et son fils gauche.

1. **Méthodes d’itérations des arbres binaires**

Souvent il est souhaitable de visité chacun des nœuds dans un arbre et d’y examiner la valeur. Il existe plusieurs ordres dans lesquelles les nœuds peuvent être visité, et chacun à des propriétés utile qui sont exploité par les algorithmes basé sur les arbres binaires. Les différents types l’algorithme que nous allons utiliser sont :

* Parcours préfixe
* Parcours infixe
* Parcours postfixe

Soit une structure Arbre dont la racine est A et une référence gauche et droite à ces deux fils. On peut écrire les fonctions algorithmiques suivantes :

* **Parcours Préfixe**

VisiterPrefixe (Arbre A) {

Visiter(A)

Si Non\_Vide (gauche(A))

VisiterPrefixe (gauche(A))

Si Non\_Vide (droit(A))

VisiterPrefixe (droit(A))

}

* **Parcours Postfixe**

VisiterPostfixe (Arbre A) {

Si Non\_Vide (gauche(A))

VisiterPostfixe (gauche(A))

Si Non\_Vide (droit(A))

VisiterPostfixe (droit(A))

Visiter(A)

}

* **Parcours Infixe**

VisiterInfixe (Arbre A) {

Si Non\_Vide (gauche(A))

VisiterInfixe (gauche(A))

Visiter(A)

Si Non\_Vide (droit(A))

VisiterInfixe (droit(A))

}

Un parcours infixe comme ci-dessus visite chaque entre les nœuds de son sous arbre de gauche et les nœuds de son sous arbre de droite. C’est une manière assez connu de parcourir un arbre binaire de recherche car il donne les valeurs dans l’ordre croissant.

Exemple :

* Préfixe : 1-2-4-5-7-8-3-6-9
* Postfixe : 4-7-8-5-2-9-6-3-1
* Infixe : 4-2-7-5-8-1-3-9-6

Tous ces trois algorithmes récursifs utilisent une pile mémoire proportionnel à la profondeur des arbres. Si nous rajoutons dans chaque nœud une référence à son parent alors nous pouvons utiliser des espaces mémoires uniquement constant et un algorithme itératif. La référence au parent occupe cependant beaucoup d’espace, elle n’est réellement utile que si elle est par ailleurs nécessité ou si la pile mémoire est particulièrement limité.

1. **Les Arbres avec Python.**

Python ne dispose pas en standard de type permettant de représenter les arbres. Nous mettons à disposition le type Arbre qui permet de réaliser les Operations ci-dessous

* Un objet de type arbre n’est pas vide, il contient au moins une racine.
* a.racine est le label de la racine de l’arbre. On peut le changer en écrivant a.racine =xxx
* a=Arbre (v, fils=l) crée l’arbre v qui a pour fils l.
* un arbre est itérable et on peut donc écrire For f in a : print (f.racine). pour afficher les fils de la racine a.
* Len(a) donne le nombre de fils de a
* A [0] donne une référence vers le premier fils de a.

D’autres fonctionnalités sont disponible comme la création d’un arbre à partir du tuple imbriqué : ar = (racine, fils1, fils2, …) où fils1 et fils2 peuvent eu même être un tuple sur le même modèle.

Exemple :

Soit un arbre dont la racine est 5 qui a trois fils 3, 4 et 6. Le nœud 6 a deux fils 7 et 8. On a donc : a=Arbre ((5,3 ?4, (6,7 ,8)))

5

3 4 6

7 8

On peut accéder à un sous arbre par sin numéro d’ordre : soit

a=Arbre ((5,3 ?4, (6,7 ,8)))

Soit b = [2] print(b)

Print(b) ‟ va afficher 6 ”

On peut aussi remplacer un sous arbre particulier par un autre :

Soit a=Arbre ((5,3 ?4, (6,7 ,8)))

a.remplace (1, Arbre ((7, 8, 9)))

Print (a)

a.ajoute (Arbre(10))

Print (a)

On peut modifier un fils ou l’ajouter par son numéro d’ordre (si on ajoute le fils N3 a un nœud qui n’a qu’un fils, la liste des fils est complétée par ses nœuds et aucune exception n’est livrée)

b = Arbre ((10, 11,12))

a=Arbre(5)

a [1]=b,

a [0]=3,

a [1] [2]=13,

a [1] [1]=42.

1. **Les différentes notions sur les arbres**

* **Racine** c’est le nœud source.
* **Branche** liaison deux nœuds.
* **Nœud** c’est un élément d’un arbre.
* **Hauteur d’un arbre** c’est le nombre de branche entre la racine et le nœud le plus éloigné.
* **Hauteur d’un nœud** c’est le nombre de nœud du nœud au nœud le plus éloigné.
* **Nœud frère** c’est un nœud qui se trouve au même niveau.
* **Nœud parent** c’est un nœud qui a au moins un fils.
* **Nœud enfant** c’est un nœud qui a un parent.
* **Nœud ancêtre** c’est un nœud de niveau supérieur à n+2.
* **Degré d’un nœud** c’est le nombre de fils que possède ce nœud.
* **Le degré de l’arbre** c’est le plus grand des degrés de ces nœuds.
* **La taille de l’arbre** c’est le nombre total de ces nœuds.
* **Arbre lexicographique** c’est un arbre qui permet de ranger les mots par ordre lexical.
* **Arbre binaire d’expression arithmétique** c’est un arbre qui permet de retrouver une expression mathématique ou arithmétique. Exemple :

Parcours infixe = (a + b)/(c-d)

/

+ -

a b c d

Parcours infixe = BONJOUR

J

O U

B N O R

Parcours postfixe = BONJOUR

R

O U

N J

B O

1. **Les structures de données arborescentes avancées**

* **Objectif :**

Construire des structures de données avancées adapter à des opérations privilégier tel que :

* recherche, insertion, suppression, prédécesseur, successeur
* minimum, Maximum, extraire-minimum, extrait-maximum

En informatique les données sont stocker sous forme d’enregistrement doté de clé, nous considérons les structures suivantes :

* Structure linéaire (liste, tableau)
* Structure arborescente (B-arbre, arbre binaire, tas binaire)
* Les B-Arbres

Ils forment les arbres équilibrés c.-à-d. que toutes les feuilles sont au même niveau. Le nœud racine d’un B-arbre non vide doit avoir une clé et possède au moins deux fils.

Dans un B-arbre de dégrée minimum t

* Tout nœud autre que la racine doit contenir au moins t-1 clés, et tout nœud au plus 2t-1 clés
* Tout nœud interne autre que la racine possède t fils, et au plus 2t-1 fils
* Un nœud est dit plein s’il contient exactement2t-1 clés
* Si toutes les feuilles sont à la même profondeur l’arbre est équilibré

Exemple :

Le dégrée t=3

1 N0

2 3 N1

4 5 6 16 7 N3

8 9 10 11 12 13 14 15 N4

Au moins n clé=t-1=3-1=2

Au plus n clé=2t-1=2\*3-1=5

Au moins n fils=3

Au plus n fils =2t-1=2\*3-1=5

* Le B-arbre le plus simple est l’arbre dont le dégrée minimum est égale à 2.
* Les arbres AVL

Un arbre binaire est un arbre AVL (Adelson Velskii et Landis) si pour tout sommet la hauteur des sous arbres gauche et droite diffère d’au plus 1

**CH3- STRUCTURE DE DONNEES LINEAIRES**

1. **Introduction**

Dans ce chapitre nous allons décrire les représentations des structures de données de base telles les listes en générale et deux formes restreintes les piles et les files. L’autre but recherché est de voir l’importance de ces structures à travers quelques exemples d’applications.

1. **Les listes**

Les listes sont des structures de données informatique qui permettent au même titre que les tableaux par exemple de garder en mémoire des données en respectant des ordres. On peut ajouter, enlever, ou consulter un élément en début ou en fin de liste, vider une liste pour savoir si elle contient un ou plusieurs éléments.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Tête |  |  | ………………………………… |  |  | Queue |

Ajouter Vider la liste Ajouter

Enlever liste vide Enlever

* **Quelques opérations sur les listes :**

Les différentes opérations sur les listes sont :

1. Tester si la liste est vide ou la mettre à vide
2. Accéder au k-ièmes élément de la liste
3. Insérer un nouvel élément derrière le k-ièmes élément
4. Fusionner deux listes
5. Rechercher un élément d’une valeur particulière
6. Trier la liste

* **Implémentation des listes**

Il existe plusieurs méthodes pour implémenter les listes, les plus courantes sont l’utilisation de tableau et de pointeur

1. **Utilisation de tableau**

Implémenter une liste à l’aide d’un tableau n’est pas très compliquer, les éléments de la liste sont seulement ranger dans le tableau et à leur place respective. Cependant, l’utilisation de tableau possède quelques inconvénients :

* La dimension d’un tableau doit être définie lors des déclarations et ne peut donc pas être modifié dynamiquement lors de l’exécution d’un programme. La solution consiste donc à définir un tableau dont la taille serait suffisante pour accueillir la plus grande liste pouvant être utilisé et d’associer au tableau une variable indiquant le nombre d’éléments contenue dans le tableau.
* Le tableau étant surdimensionné il encombre en générale la mémoire de l’ordinateur.
* Si la taille maximale venait à être augmenter, il faudrait modifier le programme et recompiler.
* Lorsque l’on retire un élément du tableau en particulier en début de liste il est nécessaire de décaler tous les éléments situé après l’élément retiré.

1. **Utilisation de pointeur**

Les pointeurs définissent une adresse dans la mémoire de l’ordinateur. Adresse qui correspond à l’emplacement d’une autre variable. Il est possible à tout moment d’allouer dynamiquement cet espace lors de l’exécution du programme.

1. **Représentation contiguë d’une liste par tableau**

L’avantage de la représentation contiguë par tableau est que l’accès au k-ièmes élément est immédiat. Par contre l’inconvénient est que l’insertion d’un élément est très couteux car il faut décaler d’un cran tous les éléments suivant jusqu’à n+1 affectations. La suppression d’un élément pour les même raison peut entrainer jusqu’à n-1 affectations.

Exemple :

La liste L = < a1, a2, a3, a4, a5 >

On créer un tableau T [5] =

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A1 | A2 | A3 | A4 | A5 |

* **Liste circulaire** : on a une liste L=<A1, A2, A3,……, An-1> dont le suivant du dernier est le premier

A5

A4

A3

A2

A1

* **Listes doublements chainées** : utile quand on veut accéder facilement au prédécesseur de la liste.

A5

A4

A3

A2

A1

1. **La structure Pile**

En informatique, une pile est une structure de donnée fondée sur le principe ‟ dernier arrivé premier sortie ” (**LIFO Last In First Out**) ce qui veut dire les derniers éléments ajouter à la pile seront les dernier à être récupérer.

Le fonctionnement est donc celui d’une pile d’assiette, on ajoute des assiettes sur la pile et on les récupère dans l’ordre inverse en commençant par la dernière ajoutée.

* **Les primitives**

Les primitives communément utilisées pour manipuler les piles sont :

* **Empiler** : ajouter un élément sur la pile (Push)
* **Dépiler** : enlève un élément de la pile et le renvoi (Pop)
* **Vide** : revoit vrai si la pile est vide faux si non
* **Remplissage**: revoit le nombre d’élément dans la pile

Empiler Dépiler

* **Application** : les différentes applications des piles sont
* Dans un navigateur web, une pile sert à mémoriser les pages web visités, l’adresse de chaque nouvelle page visité est empiler et l’utilisateur dépile en cliquant le bouton droit afficher la page précédente
* La fonction annuler la frappe d’un traitement de texte mémorise les modifications apportées au texte dans une pile
* Un algorithme de recherche en profondeur utilise une pile pour mémoriser les nœuds visités
* Les algorithmes récursifs admissent par certaines langages (Lisps, Pascal, C) utilise implicitement une pile d’appel. Dans un langage non récursif (Fortran) on peut donc toujours simuler la récursive en créant les primitives de gestion d’une pile

1. **Les Files**

Une file est une structure de donnée basée sur le principe Premier entré premier sorti (en anglais FIFO First In First Out) ce qui veut dire que les premiers éléments ajouté à la file seront les premier à être récupérer. Le fonctionnement ressemble à une file d’attente où les premières personnes à arrivées sont les premières à sortir de la file

* **Les Primitives**

Voici les primitives communément utilisé pour manipuler les files :

* **ajouter** : ajoute un élément dans la file (en anglais Enqueue)
* **Enlever** : renvoie le prochain élément de la file et le retire de la file (en anglais Dequeue)
* **Vide** : renvoi vrai si la file est vide faux si non
* **Remplissage**: renvoi le nombre d’élément dans la file
* **Les applications**
* En général, on utilise les files pour mémoriser les transactions qui doivent attendre pour être traiter.
* Les serveurs d’impressions, ils doivent traiter les requêtes dans l’ordre d’arriver et les inserts dans une file d’attente.
* Certains moteurs multitâches dans un système d’exploitation qui doivent accorder du temps machine à chaque tâche sans n’en privilégier aucune
* Un algorithme de parcours en largeur utilise une file pour mémoriser les nœuds visités
* On utilise aussi les files pour créer toute sorte de mémoire tampon.

1. **Récapitulation : structure de données avancées et dynamique**
2. **Structure de donnée élémentaire**

Les structure de donné élémentaires les plus connu sont :

* Liste
* Pile
* File
* **Pour quoi faire ?**

Représenter des ensembles dynamique et se munir des opérations de base (recherche, insertion, suppression). Permet également de faire des opérations abstraites caractéristique et d’implémenter des opérations (Programmation).

1. **Structure contiguë et structures chainées**

Les structures les plus connu sont :

* Listes chainé
* Arbres

1. **Tableau de liste contiguë d’éléments**

Chaque élément est localisé efficacement par son indice ou adresse. Le temps d’accès est constant à indice connu. La localité permet d’exploiter les mémoires cache graphique. Soit le tableau T de taille n

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  | T[i] |  |  |  |

1 2 i n

La taille d’un tableau ne peut pas être modifiée au milieu d’un programme. Il faut anticiper la croissance d’un tableau en prévoyant n très grand ce qui entraine une perte d’espace.

1. **Le tableau dynamique**

Un tableau dynamique permet de doubler une taille de n à 2n à la demande. En supposant T soit un tableau de taille 1 au départ, combien de fois doublé la taille de T pour insérer n éléments.

Cout de la procédure égale recopie des éléments dans l’extension. La moitié des n éléments du tableau final a été copié 1 seul fois, Le quart a été copié 2 fois, etc.

1 élément copié

0

0 0

2 éléments copiés

4 éléments copiés

0 0 0 0

1. **Le pointeur**

Le pointeur représente une adresse d’un élément.

Le numéro de votre portable est un pointeur sur vous.

**Notation :** p : pointeur p  : accès élément d’adresse p

1. **Liste chainée**

Une liste chaine est une liste dont l’ordre des éléments est déterminer par le pointeur liant un élément à son successeur.

Recherche dans une liste chainée

Algo RechSeqch :

Entrée : p : liste p

E : elt

Sortie : in booléen

If (p=nul)

Then in🡨false

Else if (e=p .elt)

Then in🡨 true

Else

In🡨 RechSeqch (p .suiv, e)

1. **Avantage des structures chainées**

Concernant les structures chainée il y’a pas de débordent. Insertion et la suppression plus simple que pour un tableau. Pour les éléments de grande taille le déplacement d’élément est plus facile et plus rapide

1. **Liste doublement chinée**

Les listes doublement chainées sont caractérises par une faisabilité supplémentaire pour l’insertion et la suppression.

LC

LDC

1. **Liste trié**

On dispose d’un ordre sur les éléments et la liste est organisée selon cet ordre.

Si la liste est représenter par un tableau alors ordre des éléments égale à ordre des indices.

Si i< j ⬄ T[i] < T[j]

1. **File et Pile :**

Les piles et les files sont des listes non triées dont l’organisation respecte l’ordre d’arrivée des éléments.

Si i<j ⬄ date\_ins(T[i]) <= date\_ins(T[j])

1. **Politique d’insertion et de suppression distinct**

|  |  |  |
| --- | --- | --- |
|  | File | Pile |
| Insertion | Enqueue | En Tête |
| Suppression | entête | En Tète |

1. **File et pile pour parcours d’arbre**

A

B C

D E F G

Parcours File : ABCDEFG

Parcours Pile : ACGFBED

1. **Structure élémentaire et complexité**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Cas non trié** | | | **Cas trié** | | |
|  | Tableau | Liste chainée | Liste 2X chainées | Tableau | Liste chainée | Liste 2X chainées |
| Rechi (E, k) | Ө(1) |  |  | Ө(1) |  |  |
| Reche (E, e) | Ө (n) | Ө (n) | Ө (n) |  | Ө (n) | Ө (n) |
| Insert (E, e) | Ө (n) | Ө (1) | Ө (1) | Ө (n) | Ө (n) | Ө (n) |
| Supp (E, x) | Ө(1) | Ө (n) | Ө (1) | Ө (n) | Ө (n) | Ө (1) |
| Succ (E, x) | Ө (n) | Ө (n) | Ө (n) | Ө (1) | Ө (1) | Ө (1) |
| Pred (E, x) | Ө (n) | Ө (n) | Ө (n) | Ө (1) | Ө (n) | Ө (1) |
| Min (E) | Ө (n) | Ө (n) | Ө (n) | Ө (1) | Ө (1) | Ө (1) |
| Max (E) | Ө (n) | Ө (n) | Ө (n) | Ө (1) | Ө (n) |  |

1. **Les Tables de Hachage**

* **Pour quel besoin ?**

Les tables de hachage permettent de rechercher un élément par sa clé dans un grand ensemble dynamique afin de procéder aux opérations d’insertion et de suppression

* **Application :**

On applique les tables de hachage dans les structures de contrôle système :

* tables des symboles
* tables des transactions
* maintenir et accéder à un dictionnaire, à une base de données
* **Principe de base :**

Pour un tableau le temps d’accès à un élément en fonction de son indice est constant. La complicité au pire des cas donne Rechi(T, k) est dans Ө (1).

Fonction : h : clé 🡪 indice

* Cas simple : table à adressage direct.

E est un sous ensemble de U contenant au plus m éléments connus, h est une fonction

U 🡪 [1………..m] telle que e ≠ e’ => h (e) ≠ h (e’) h(e) détermine e

* Rechercher un élément e : accès au tableau T avec h(e)
* Insérer un élément e : item
* Supprimer un élément e : item

1 2 i m

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |

e

28

24

23

* **Problème de collision** :

Il y’a collision c’est-à-dire e ≠ e’ et h (e) ≠ h (e’)

Chaque entrée T(i) contient un pointeur sur une liste d’éléments e tel que h (e) = i

* **Recherche d’un élément dans une table de hachage** : RechTH (T, e), qui veut dire rechercher e dans la liste T (h(e))
* **Insertion d’un élément dans une table de hachage** : InserTH (T, e) insertion de e dans la liste de T (h(e))
* **Suppression d’un élément dans une table de hachage** : SuppTH (T, e) supprimé e dans la liste de T (h(e)).

**CHAP 4 : STRUCTURE DE DONNEES AVANCEES ET LEURS ALGOEITHMES**

1. **Introduction**

Acquérir les fondements théoriques et pratiques des structures de données et leurs algorithme dans le but de comprendre comment les justifiés, les utilisées et les incorporés dans la résolution d’un problème. L’accent sera mis sur les propriétés fondamentales de ces structures et l’étude de leurs complexités sans toutefois négligé les aspects liés à leurs implantations et à leurs applications.

* **Qu’est-ce que l’algorithmique?**

Un algorithme est une suite finie d’instruction élémentaire constituant un schéma de calcul ou de résolution d’un problème.

L’algorithmique concerne :

1. **Trouver une méthode de résolution exacte ou approprier du problème**

Ex :

Soit trois nombres réels a b c, quels sont les solutions de l’équation

Soit 6 nombres réels a b c d e f, quels sont les solutions de l’équation

1. **Trouver une méthode efficace**: savoir résoudre un problème est une chose le résoudre efficacement en ait une autre.

* **Différence entre algorithme et programme**

Un programme est la réalisation d’un algorithme au moyen d’un langage donné (sur une architecture donnée). Par exemple lors de la programmation on s’occupera parfois explicitement de la gestion de la mémoire (allocation dynamique en C qui est un problème d’implémentation ignoré au niveau algorithmique)

1. **Initiation à l’algorithmique**

* **Rappel**

Un algorithme est en général décomposable en trois parties :

* Un prétraitement qui concerne l’entrée des données au clavier, initialisation des valeurs
* Un traitement qui concerne les calculs et manipulation des données
* Un posttraitement qui permet l’affichage des résultats sur l’écran, écriture dans un fichier

Exemple :

Début

Lire le nbre a

Lire le nbre b

Donner à r la valeur a mod b

Répéter jusqu’à  r = 0

Donner à a la valeur b

Donner à b la valeur r

Donner à r la valeur a mod b

Fin

Affiche b

Fin

* **Les Bases**

Nos algorithmes serons écris en pseudo code c.-à-d. sous une forme standardisée et à l’aide d’un certains nombres de mots clé. Ceci permettra à la fois d’apparaitre les structures communes et de facilité la programmation éventuelle en se rapprochant d’un langage de programmation. Toutefois il faut garder à l’esprit que le choix exact de cette forme et de ces mots clé est en grande parties arbitraire et peut donc légitimement varié. Les instructions souvent utilisé sont :

* Pour obtenir une donnée en entrée au clavier : Lire la valeur de a
* Pour afficher un message ou un résultat à l’écran : Afficher ’le résultat est ’ b
* Pour affecter une valeur à une variable : donner à a la valeur 12
* **Structure alternative**

Le plus souvent un algorithme ne contient pas les instructions de manipulation des données à exécuter les unes après les autres mais aussi des instructions dites de contrôle qui ont un effet sur l’exécution des autres.

Le premier type de telle instruction est celle permettant une exécution conditionnelle

Si condition alors

Instruction à effectuer

Si la condition est réalisée

Fi



Si condition alors

Instruction à effectuer

Si la condition est réalisée

Fi

Sinon

Instruction à effectuer

Si la condition n’est pas réalisée

Fin

Exemple 1 : donner l’algorithme qui a pour but de lire deux nombre au clavier puis de les afficher dans l’ordre croissant puis les rangés dans une structures qui les sort par ordre décroissant.

Exemple 1 : écrire un algorithme qui lit trois nombres au clavier et affiche le maximum des trois. **Correction**

Exemple 1 :

Algo Ordre

Début

Var i, a, b : entier Tab [2] : entier

Lire a

Lire b

Si a < b alors

Afficher a

Afficher b

Tab [1] 🡨 b

Tab [2] 🡨 a

Sinon

Afficher b

Afficher a

Tab [1] 🡨 a

Tab [2] 🡨 b

Fin

Fin

Exemple 2

Algo Maximum

Début

Var a, b, c, max : entier

Si a < b alors

Si b < c alors

Max 🡨 c

Sinon

Max 🡨 b

Fin

Sinon si a < c

Max 🡨 c

Sinon

Max 🡨 a

Fin

Fin

Afficher Max

Fin

* **Structure Répétitive**

Il s’agit de répéter un bloc d’instruction plusieurs

Les deux familles principales consiste à répéter un bloc d’instruction un nombre de fois donnée ou de répéter un bloc d’instruction jusqu’à ce qu’une condition soit vérifiée.

Pour i de 1 à 10 faire

Instruction à effectuer

Fin

Tant que condition Faire

Instruction à effectuer

Fin

Exemple1 :

Faire l’algorithme qui calcule factoriel 15, puis reprendre le même algorithme en utilisant une structure de donnée.

**Exemple**

Début

Donner à r la valeur 1

Pour i de 1 à 15 faire

Donner à r la valeur r\*i

Fin

Afficher r

Fin

1. **Algorithme et instruction sur les listes, Piles, Files, Arbre, Tableau.**

* **Listes**

Une liste tel que déjà vue est une séquence d’objet ordonnée. Les opérations classiques sur les listes sont les suivantes :

* Voire si la liste est vide
* Quel est le premier élément de la liste
* Quel est la liste obtenue si on prive la liste de départ du premier élément

Ces trois opérations sont réalisées en temps constant. Dans les listes chainées, on peut généralement à l’aide d’un pointeur parcourir la liste en passant d’une case à la suivante.

Les liste chainées sont un exemple classique de manipulation de pointeur en C. les liste sont aussi utilisées comme base à d’autres structures de données.

* Liste en Python

Les listes Python ne sont pas identiques aux listes décrites ci-dessus. Quelques cas de commande sur les listes en python :

Len(s) permet de connaitre la longueur de la liste.

S [0] permet de connaitre le premier élément de la liste

S [-1] permet de connaitre le dernier élément de la liste

On peut donc en python accéder à une valeur par son numéro d’ordre et ceci en temps constant.

Exemple : S [10] est la 11ième valeur de la liste

* **Piles**

Les concepts associés aux piles sont LIFO premier arrivé dernier sortie. Les opérations classiques sur les piles sont

* Voire si la pile est vide
* Ajouter un élément à la pile
* Consulter le sommet de la pile
* Retirer l’élément au sommet de la pile

Les instructions sur les piles en python sont :

* Len(p) voir si la pile est vide
* P.append (v) ajoute v au sommet de la pile
* P [-1] est l’élément au sommet de la pile
* P.pop () retire l’élément au sommet de la pile
* **Files**

Les concepts associés aux files sont FIFO premier arrivé premier servie. Les opérations connus sur les files sont

* Voire si la file est vide
* Enfiler un élément
* Défiler un élément

Les instructions sur les files en python sont :

* From collection import deque : pour utiliser le module.
* D=deque : création d’une file vide.
* D= deque ((4, 5, 6)) : création d’une file à partir d’un tuple.
* Len (d) : permet de savoir si la file est vide.
* D.pop () : renvoie l’élément de tête de file.
* D.appendleft(v) : ajoute un élément en fin de file.

Exercice de cour :

Faire un algorithme pour créer une file et une pile et dans chaque structure ranger les 10 premiers chiffres par ordre croissant. Faire un autre algorithme qui après avoir vérifié si la structure est vide ou pas, sortir 4 éléments pour former le nombre 3508.

NB : vous pouvez utiliser les instructions python dans vos algorithmes.

1. **Structure de Donnée en Java**

* La classe String

Le type de donnée string n’est pas un type élémentaire en java c’est une classe. Donc une chaine de type string est un objet qui n’est utilisable qu’à travers les méthodes de la classe string. Pour accéder à la classe string et à toute ses méthode vous devez mettre en avant la déclaration de votre classe l’instruction d’importation de package suivant :

Import java.lang.String ;

Un littéral de chaine est une suite de caractère entre guillemet, exemple ‘’abcdef’’

|  |  |
| --- | --- |
| Déclaration d’une variable String | String strl ; |
| Déclaration d’une variable avec initialisation | String strl=’’abcdef’’ ; |
| On accède à la longueur d’une chaine par la méthode int length () | String strl=’’abcdef’’ ;  Int longueur ;  Longueur=strl.length () ; ici longueur = 6 |

Toutes les autres manipulations sur les objets String nécessite l’emploi des méthodes de la classe String. Nous donnons quelques utilisations classiques sur la classe String.

Le type String possède quelques méthodes classiques d’extensions, de concaténation, de changement de casse, d’extraction, etc.

|  |  |
| --- | --- |
| Concaténation de deux chaines. Un opérateur ou une méthode  Opérateur : + sur les chaines  Ou  Méthode : String .Concat (string s)  Strl3=strl1+strl2 ;  Strl3=strl1.Concat(Strl2) ; | String strl1, strl2, strl3 ;  Strl1=’’Bon’’ ;  Strl2=’’jour’’ ;  Strl3=strl1+strl2 ; |

1. **Les tableaux Arrays, Les matrices**

Dès que l’on travaille avec des éléments homogènes, la première structure de base permettant le regroupement de ces données est le tableau.

Java comme tous les langages algorithmique propose cette structure aux programmeurs. Comme les String pour des raisons d’efficacités dans l’encombrement mémoire les tableaux sont gérés par java comme des objets.

La table d’un tableau doit obligatoirement être définie avant que java accepte que vous l’utilisiez.

* Déclaration d’une variable de tableau à une dimension

Int [] table1 ;

Char [] table2 ;

Float [] table3 ;

* Déclaration d’une variable de tableau avec définition explicite de taille

Int [] table1= new int [5] ;

Char [] table2 = new char [12];

Float [] table3= new Float [8];

Un tableau a une dimension lorsque sa taille a été fixé soit par une définition explicite ou soit par une définition implicite ne peut plus changer de taille c’est donc une structure statique.

FIN.