جامعة أبو بكر بلقايد- تلمسان

Faculté de Technologie

Département de Génie Biomédical

Informatique biomédical

Support de travaux pratiques

Structures des données avancées

Pr KADRI Benamar

3 eme année Licence informatique biomédicale hospitalière

Table des matières

Introduction générale

Les listes chainées

1. Introduction …… 5
2. Structure des maillons d'une liste chainée …… 6
3. Déclaration d'une liste chainée…… 6
4. Fonctionnement de la liste chainée …… 6
5. Type de liste chainée …… 7
6. Opérations sur les listes …… 8
7. Enoncé de TP…… 8
8. Solution de TP…… 9

Les files d'attente

1. La structure d'une file …… 14
2. Les opérations sur les files…… 15
3. Enoncé de TP…… 15
4. Solution de TP…… 16

Les piles

1. La structure d'une pile…… 20
2. Les opérations sur les piles…… 20
3. Enoncé de TP…… 21
4. Solution de TP…… 21

Les arbres

1. Définition …… 25
2. Caractéristiques d'un arbre …… 25
3. Arbre binaire…… 26
4. Parcours d'un arbre binaire…… 26
5. Un arbre binaire de recherche…… 27
6. Enoncé du TP…… 28
7. Solution De TP …… 29

Référence …… 32

Liste des figures

Figure 1 : Liste chainée 5
Figure 2 : Fonctionnement des listes chainées……………………………………………. 6
Figure 3 : Liste doublement chainée ……………………………………………………… 7
Figure 4 : Liste doublement chainée circulaire …………………………………………… 8
Figure 5 : Ajout des éléments dans une chainée ………………………………………….. 8
Figure 6 : Suppression des éléments d'une chainée ……………………………………… 9
Figure 7 : File d'attente …………………………………………………………………….. 14
Figure 8 : Politique de gestion d'une file d'attente ……………………………………… 15
Figure 9 : Les piles ………………………………………………………………………. 20
Figure 11 : Les arbres ……………………………………………………………………….. 25
Figure 12 : Les arbres binaires…………………………………………………………….. 26
Figure 13 : Les arbres binaires de recherche …………………………………………… 27
Figure 14 : Suppression des éléments d'un arbre ………………………………………. 28

Introduction générale

Ce support de TP est destiné au troisième année licence informatique biomédicale, il est composé de quatre travaux pratiques permettant aux étudiants de maitriser, en fin de semestre, un concept important du module d'informatique conventionnel : « les structure des données ».

Le document est composé de quatre travaux pratiques : les listes chainées, les files, les piles et les arbres de recherche avec rappel théorique précédant chaque travail pratique.

Le support de TP est organisé comme suit :

TP1 : Il permet aux étudiants d'apprendre des notions générales sur les listes chainées, comme la déclaration, le parcours, la recherche, l'affichage des éléments de la liste … L'exemple utilisé dans ce TP est inspiré du domaine du génie biomédical, il s'agit d'une liste chainée des patients d'un hôpital et l'exécution des programmes sera en utilisant un menu textuel.

TP2 : C'est une implémentation du concept de la file d'attente en utilisant les listes chainées. Le concept des files d'attente est un concept très courant en informatique pour gérer la file d'attente des processus, la liste des taches à imprimer… Ici encore, pour rester dans le domaine biomédical, on va utiliser les files d'attente pour gérer des patients.

TP3 : C'est une copie du TP2 avec changeant de la politique d'accès aux éléments de la liste chainée : LIFO au lieu de FIFO.

TP4 : IL est consacré aux notions d'arbres binaires et permet aux étudiants de bien appréhender les opérations sur les arbres binaires ainsi que les fonctions récursives.

Rappels sur les listes chainées

1. Introduction

Généralement les ensembles de même type sont groupés dans des structures contiguës « tableaux ou matrice ». L'inconvénient de ces structures est la pré-allocation, en effet, on doit connaitre à l'avance la taille des données pour réserver l'espace mémoire nécessaire dans la RAM.

Par exemple pour sauvegarder la liste des étudiants dans une application de gestion de scolarité en utilisant les tableaux, on doit déclarer à l'avance un tableau de structure/classe étudiant avec une taille fixe. Le problème est qu'on ne connaît pas le nombre d'étudiants qu'on va utiliser dans notre application. On opte donc pour une valeur maximale, ce qui entraine forcément à un gaspillage de l'espace mémoire puisque le maximum n'est atteint qu'une ou deux fois au cours de l'utilisation de l'application.

Pour remédier à ce gaspillage de l'espace mémoire, on utilise généralement une liste chainée, dans laquelle les éléments de la liste « étudiant dans notre application », ne sont pas sauvegardés dans un tableau mais ils sont créés à la demande, en utilisant une allocation dynamique « pointeur ». A chaque création, un élément est ajouté dans cette liste et est chainé aux autre grâce à un pointeur.

Une liste de type chainée/dynamique est composée d'un ensemble d'éléments « maillons » liés les uns aux autres grâce à des pointeurs. On doit connaitre le premier maillon pour atteindre les autres. Le concept est celui que nous connaissons dans les vraies chaines de personnes dans les banques, les entrées de stades …On voit le premier de la chaine, et lui peut voir celui qui esr derrière lui et ainsi de suite. Dans la chaine, un maillon connait le suivant ou bien le précédent ou même les deux « précédant et suivant ».

Les éléments (maillons, noeuds ou liens) contenant des données sont créés à la demande et contrairement à un tableau, ces éléments peuvent être éparpillées en mémoire et reliées entre eux par des liens logiques (des pointeurs), c'est-à-dire un ou plusieurs champs dans chaque structure contenant l'adresse d'une ou plusieurs structures de même type.

L'accès aux éléments de la liste est séquentiel c-à-d. on commence toujours par un élément initial dit tête de la liste qui constitue un repère pour accéder aux restes des éléments. Il est similaire au nom de tableau dans les listes de taille fixe (Figure 1).

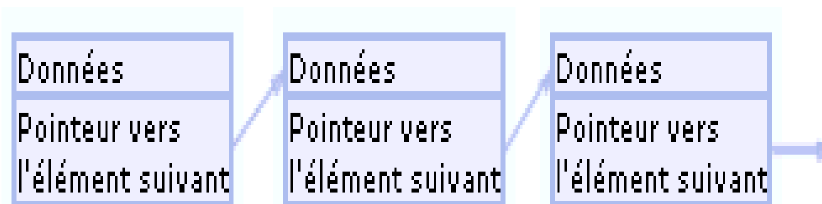

Figure 1 : liste chainée

A l'inverse des tableaux, une liste dynamique peut contenir théoriquement n'importe quel nombre d'éléments du fait que la liste est étendue suite aux besoins du programme.

Une liste chainée permet d'économiser l'espace mémoire car les maillons de la liste sont créés

suivant les besoins du programme et non pas lors de la compilation.

2. Structure des maillons d'une liste chainée

Les maillons d'une liste sont des structures habituelles « vues dans le chapitre précédent » ou bien des objets instances de classe dans le cas de la programmation orientée objet, et qui contiennent un ou plusieurs champs de données « nom, prénom…etc » ainsi que des champs supplémentaires pour enregistrer le lien « pointeur » vers l'élément suivant ou précédent dans la liste.

Un pointeur c'est un champ qui contient l'adresse du prochain/précédent élément de la liste. Grâce à ce pointeur on peut se déplacer d'un maillon à un autre jusqu'à l'arrivé au maillon cherché.

3. Déclaration d'une liste chainée

La déclaration d'une liste chainée est composée de la structure qui compose les maillons de la liste ainsi que le premier élément de cette liste « tète de liste » qui constitue le repère de la liste.

Structure d'un maillon

```
class element {
            int var;
            element  suivant;
};
Element tete ;
```

4. Fonctionnement de la liste chainée

En réalité le fonctionnement de la liste chainée est dû aux pointeurs qui assurent la récursivité de la liste et le lien entre les nœuds de la liste, mais cela n'est pas suffisant. En effet, il est nécessaire de conserver une « trace » du premier élément similaire aux chaine habituelles afin de pouvoir accéder aux autres éléments. C'est pourquoi, un pointeur vers le premier élément de la liste est indispensable.

Ce pointeur est appelé pointeur de tête, il est fixé à **null** au début de la création de la liste tête ensuite, il prendra la valeur du premier élément de la liste et gardera cette valeur jusqu'à la suppression de la liste.

D'autre part, étant donné que le dernier enregistrement ne pointe vers rien, il est nécessaire de donner à son pointeur la valeur NULL.

Le programmeur aura le choix d'ajouter les éléments de la liste au début de cette liste ce qui veut dire que le nœud tête changera la valeur après chaque opération d'ajout, ou à la fin de la liste.

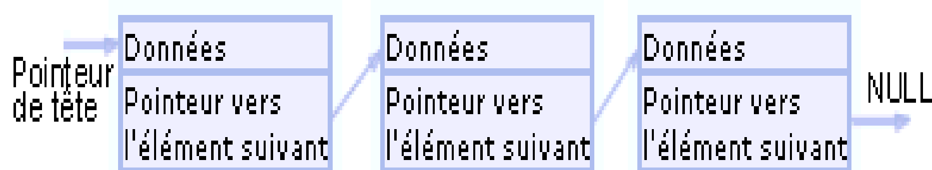

Figure 2 : Fonctionnement des listes chainées

5. Type de liste chainée

Suivant le nombre de champs pointeurs utilisé dans la structure ainsi que la méthode d'utilisation de ce pointeur plusieurs variantes de listes existent.

5.1 Liste simplement chainée

Cette liste ne contient qu'un seul pointeur vers le maillon suivant, le dernier maillon contient la valeur null.

Dans cette liste on doit toujours sauvegarder le premier maillon « tête de liste », qui est utilisé comme repère pour accéder à la liste.

Un nouvel élément peut être ajouté en tête de liste « pile » ou bien à la fin de la liste « file ».

La suppression d'un élément consiste à remplacer la valeur du pointeur du maillon précédent par celle du maillon suivant, ce qui va causer l'ignorance de ce maillon ce qui entraine sa suppression par le système.

Déclaration

```
Class element {
  Public :
        Int var
        Element  suivant;
}
Elment  tete ;
```

5.2 Liste doublement chainée

Cette liste est similaire à la liste précédente, cependant elle contient deux pointeurs l'un de ces pointeurs pointe vers le maillon suivant et l'autre vers le maillon précédent, ce qui va permettre un parcours de la liste en double sens. Ce type de liste est très pratique si les éléments dans le maillon sont ordonnés, pour diriger le parcours de la liste vers la direction la plus performante en matière de temps de recherche. A titre d'exemple, si une liste contient des éléments entiers ordonnés d'une manière croissante/décroissante la direction du parcours de la liste se fera suivant la valeur de l'élément recherché ainsi que la position du maillon courant « avant en arrière »

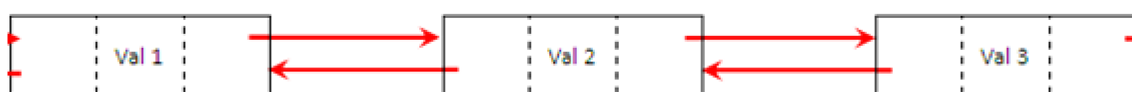

Figure 3 : Liste doublement chainée

Déclaration

```
Class element {
  Public :
       Int var
       Element  suivant;
       Element  precedent ;
}
Elment  tete ;
```

5.3 Les listes circulaires

Une liste circulaire est une liste simplement ou doublement chainée, mais qui ne contient pas de dernier élément « null ». En effet, le dernier élément contient un pointeur vers le premier élément. Dans cette liste on ne garde pas la tête de liste comme repère de liste car chaque maillon de la liste est considéré comme tête de liste.

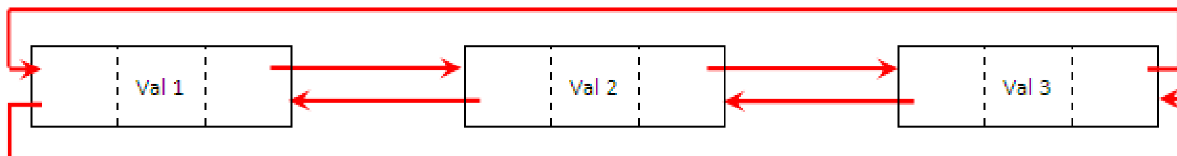

Figure 4 : Liste doublement chainée circulaire

6. Opérations sur les listes

6.1 Ajouter un élément

Avant d'ajouter un élément il faut décider la position dans laquelle on veut l'ajouter, en tête de la liste « pile », fin de la liste « file », ou suivant un ordre bien définit dans le cas d'une liste ordonnée.

L'ajout d'un élément commence par la création de cet élément « créer un espace mémoire » ensuite ajouter cet élément dans la liste. Cet ajout se fait en lui donnant l'adresse du premier élément dans le cas d'une pile, ou en remplaçant le pointeur du dernier élément par l'adresse de l'élément crée, ou en cherchant l'emplacement adéquat dans le cas d'une liste ordonnée. Dans ce dernier cas, on effectue le changement de la valeur des pointeurs du prédécesseur et le nouvel élément.

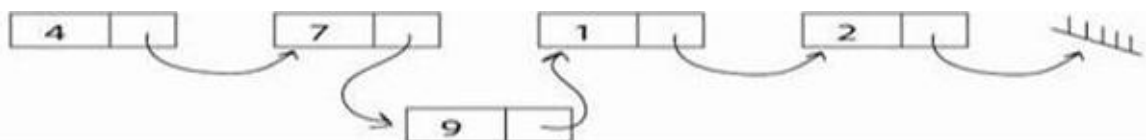

Figure 5 : Ajout des éléments dans une chainée

6.2 Vider la liste

Vider une liste chainée est très facile, et consiste à affecter à la tête de liste la valeur null. Ceci entraine la perte de toutes les adresses des structures dans cette liste. Ensuite, le ramasse miette se charge de la libération de l'espace mémoire de ces éléments.

Une autre manière de vider la liste est de la parcourir jusqu'au dernier élément et supprimer avec **delete** toutes les structures.

6.3 Supprimer un élément

La suppression d'un élément de la liste consiste premièrement à trouver cet élément ensuite changer le pointeur de son élément prédécesseur en lui donnant l'adresse de son successeur. Cette action va laisser le maillon supprimé non pointé ce qui va entrainer sa suppression par le ramasse miette.

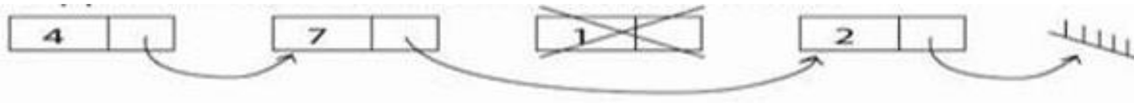

Figure 6 : Suppression des éléments d'une liste chainée

6.4 Affichage des éléments

L'affichage des éléments de la liste chainée consiste à parcourir la liste depuis le premier élément jusqu'au dernier élément et afficher les valeurs de chaque maillon.

6.5 Chercher un élément

La recherche d'un élément dans la liste consiste à parcourir la liste depuis le premier élément jusqu'au maillon qui contient la valeur recherchée ou bien arriver à la fin de la liste sans succès.

7. Enoncé du TP

On désire implémenter un logiciel qui gère une liste chainées qui contient comme maillon des patient « id,nom, prenom, age, adresse, date_naiss » :

- Donner la déclaration de la classe patient qui compose cette liste.
- Donner la déclaration de la liste.
- Donner la fonction qui permet de tester si la liste est vide.
- Donner la fonction qui permet d'ajouter un élément dans la liste.
- Donner la fonction qui permet d'afficher les éléments de la liste.
- Donner la fonction qui permet de calculer le nombre des éléments dans la liste.

- Donner la fonction qui permet de chercher et afficher les éléments dans la liste « par nom ».
- Donner la fonction qui permet de vider la liste.

- Les fonctions précédentes sont lancées en utilisant un menu « en utilisant la boucle switch » avec des numéros pour chaque commande permettant de choisir la fonction à exécuter en utilisant ce numéro.
- L'application est composée de deux classes seulement
    - La classe principale qui contient la méthode main ()
    - La classe patient qui contient la déclaration des maillons de la file.

8. Solution de TP

Déclaration de la classe

```
public class patient {
    String nom ;
    int num;
    public static int cmpt=0;
    patient next ;

    public String getNom() {
        return nom;
    }

    public int getNum() {
        return num;
    }

    public element getNext() {
        return next;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    public void setNum(int num) {
        this.num = num;
    }

    public void setNext(element next) {
        this.next = next;
    }

    public patient(String nom) {
        this.nom = nom;
```

```
        this.num = cmpt++;
    }
    public String toString() {
        return " patient {" + "nom=" + nom + ", num=" + num +'}';
    }
}
```

Déclaration de la liste

```
public static patient head = null ;
```

Tester si la liste est vide

```
public static boolean est_vide (){
        return head == null;
}
```

Ajouter un élément dans la liste

```
public static void ajouter(patient m ){
    if (is_empty()){
        head=m;
        m.next=null ;
        return ;}
    patient e = head ;
    while (e.next != null){
        e=e.next;}
    e.next=m;
    m.next=null;
  return ;
 }
```

Afficher les éléments de la liste

```
public static void afficher (){
    patient h=head;
      if (est_vide ()){
        System.out.println("la liste est vide");
        Return;
    }

    while(h!=null){
        System.out.println(h);
        h=h.next;
    }
}
```

Compter du nombre des éléments dans la liste

```
public static void compter(){
    int compt=0;
    patient c=head;
    while (c!=null){
          compt++;
          c=c.next;
```

```
    }
    System.out.println(compt);
}
```

La recherche par nom

```
public static void chercher (){
    System.out.println(" donner  nom ");
    String d=sc.next();
    patient h=head;
    while(h!=null){
        if (h.getNom().equals(d)){
        System.out.println(h);
        return ;
        }
         h=h.next;
    }
}
```

Vider la liste

```
public static void vider(){
    head=null;
}
```

Programme principale

```
public static void main(String[] args){
 int choix =0 ;
        do {
        System.out.println("pour tester la question 1 tapez 1");
        System.out.println("pour tester la question 2 tapez 2");
        System.out.println("pour tester la question 3 tapez 3");
        System.out.println("pour tester la question 4 tapez 4");
        System.out.println("pour tester la question 5 tapez 5");
        System.out.println("pour tester la question 6 tapez 6");
        choix= sc.nextInt();
        switch (choix) {
          case 1 : est_vide ();
                   break;
          case 2 : {
                System.out.println("entre le nom");
                String p=sc.next();
               patient m = new patient (p);
                ajouter(m);
                    };
                   break;
          case 3 : afficher();
                    break;
          case 4 : compter();
                     break;
          case 5 : chercher();
```

```
                    break;
          case 6 : vider();
                    break;
          }
     }while (choix!=0);
}
```

Rappel sur les files d'attentes

1. La structure file

Une file est une liste chainée ou l'insertion et la suppression des nouveaux nœuds sont gérées avec une politique **FIFO** (First In First out), c.-à-d., le premier inséré, est le premier servi. Une file est une implémentation informatique d'une file d'attente où le premier arrivé est le premier servi.

Figure 7 : File d'attente

Une file utilise deux opérations principales « enfiler et défiler » :

- Enfiler un objet o : veut dire ajouter cet objet dans la fin de la liste
- Défiler un objet o : veut dire retirer cet objet de la liste pour être servi.

Un exemple informatique de l'utilisation des files d'attentes en système d'exploitation est la mise en attente des demandes d'entrée/sortie comme la demande d'accès au disque dur, réseau, clavier…etc dans lesquelles les processus demandant une ressource sont stockés dans une liste gérée par une politique FIFO.

Un autre exemple plus connu de l'utilisation des files sur ordinateur est celui de l'impression des documents reçus par une imprimante qui imprime le premier document arrivé et termine par le dernier.

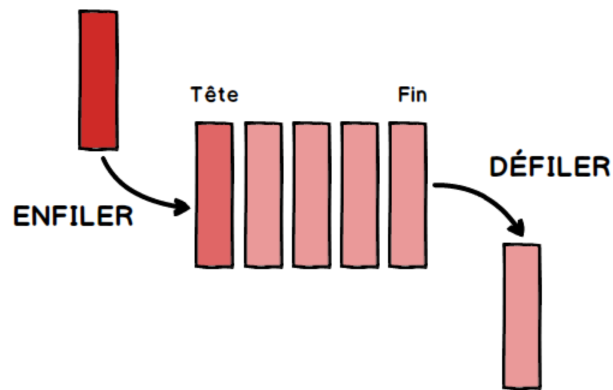

Figure 8 : Politique de gestion d'une file d'attente

2. Les opérations sur les files

Les opérations permises sur les files sont :

- Tester si la file est vide
- Enfiler un nouvel élément (ajouter un élément à la fin de la file)
- Défiler : supprimer le dernier élément de la file
- Consulter la valeur de l'élément à la fin de la file sans le défiler
- Connaître le nombre d'éléments dans la file.

3. Enoncé du TP

On désire implémenter un logiciel qui gère une file d'attente des patients dans une salle d'attente d'un médecin :

- Donner la déclaration de la classe patient qui compose la file
- Donner la déclaration de la file d'attente.
- Donner la fonction qui permet de tester si la file est vide.
- Donner la fonction qui permet d'ajouter un élément dans la file d'attente.
- Donner la fonction qui permet d'afficher les éléments de la file d'attente.
- Donner la fonction qui permet de consulter la valeur du sommet de la file.
- Donner la fonction qui permet de supprimer un élément de la file d'attente.
- Donner la fonction qui retourne le nombre des éléments dans la file.
- Donner la fonction qui permet de vider la file.

- Les fonctions précédentes sont lancées en utilisant un menu « en utilisant la boucle switch » avec des numéros pour chaque commande permettant de choisir la fonction à exécuter en utilisant ce numéro.
- L'application est composée de deux classes seulement
    - La classe principale qui contient la méthode main ()
    - La classe patient qui contient la déclaration des maillons de la file.

4. Solution du TP

Déclaration de la classe

```
public class patient {
    String nom ;
    int num;
    public static int cmpt=0;
    patient next ;

    public String getNom() {
        return nom;
    }

    public int getNum() {
        return num;
    }

    public element getNext() {
        return next;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    public void setNum(int num) {
        this.num = num;
    }

    public void setNext(element next) {
        this.next = next;
    }

    public patient(String nom) {
        this.nom = nom;
        this.num = cmpt++;
    }
    public String toString() {
        return " patient {" + "nom=" + nom + ", num=" + num +'}';
    }
}
```

Déclaration de la liste

```
public  static patient head = null ;
```

Tester si la liste est vide

```
public static boolean est_vide (){
        return head == null;
}
```

Ajouter un élément dans la liste

```
public static void ajouter(patient m ){
    if (est_vide ()){
        head=m;
        m.next=null ;
        return ;
      }
    patient e=head;
    while (e.next != null) e=e.next;
     e.next=m;
     m.next=null;
   return ;
 }
```

Afficher les éléments de la liste

```
public static void afficher (){
    patient h=head;
       if (est_vide ()){
        System.out.println("la liste est vide");
        Return;
    }

    while(h!=null){
        System.out.println(h);
        h=h.next;
    }
}
```

Consulter le premier sommet de la file

```
    public static void consulter(){
       if (est_vide ()){
        System.out.println("la liste est vide");
      }
       else{
           patient h=head ;
           while(h.next!=null){
                h=h.next;
             }
         System.out.println(h);
      }
}
```

Supprimer un élément de la file d'attente

```
public static void supprimer(){
      if (is_empty()){
       System.out.println("la liste est vide");
    }
          else{ head = head.next;}
}
```

Compter du nombre des éléments dans la liste

```
public static void compter(){
    int compt=0;
    patient c=head;
    while (c!=null){
          compt++;
          c=c.next;
    }
     System.out.println(compt);
}
```

Vider la liste

```
public static void vider(){
    head=null;
}
```

Programme principale

```
public static void main(String[] args){
 int choix =0 ;
       do {
       System.out.println("pour tester la question 1 tapez 1");
       System.out.println("pour tester la question 2 tapez 2");
       System.out.println("pour tester la question 3 tapez 3");
       System.out.println("pour tester la question 4 tapez 4");
       System.out.println("pour tester la question 5 tapez 5");
       System.out.println("pour tester la question 6 tapez 6");
       System.out.println("pour tester la question 7 tapez 7");
       System.out.println("pour tester la question 8 tapez 8");
       choix= sc.nextInt();
       switch (choix) {
         case 1 : est_vide ();
                  break;
         case 2 : {
               System.out.println("entre le nom");
               String p=sc.next();
              patient m = new patient (p);
               ajouter(m);
```

```
                };
               break;
      case 3 : afficher();
                break;
      case 4 : consulter();
                break;
     case 5 : supprimer();
                break;

      case 6 : compter();
                 break;
      case 7 : chercher();
                   break;
      case 8 : vider();
                    break;
         }
    }while (choix!=0);
}
```

Rappel sur les piles

1. La structure de pile

Les piles sont similaires aux files d'attente, dans le fait qu'elles ont deux opérations principales « insérer et supprimer » un élément.

Généralement c'est une liste chainée gérée par une politique LIFO (Last In First Out : dernier entré, premier sorti), où un élément est inséré en tête de liste, et en cas de suppression on supprime cet élément « le dernier inséré ».

Une pile utilise deux opérations principales « empiler et dépiler » :

- Empiler un objet o : veut dire ajouter cet objet dans tête de la liste
- Dépiler un objet o : veut dire retirer cet objet de la liste.

En informatique une pile sert essentiellement à stocker des données qui ne peuvent pas être traitées immédiatement, ce qui veut dire que les taches à traiter sont toujours en tête de liste. Un exemple des piles en système d'exploitation est les appels et les retours de fonctions dans lesquelles le système d'exploitation enregistre « empile » l'état de la fonction appelante « variables locales et pointeurs ainsi que les valeurs des registres du processeur » dans une pile et exécute la fonction appelée et à la fin de l'exécution de cette fonction, il récupère l'état de la fonction appelante « variables locale » depuis la pile et continue l'exécution de la fonction appelante.

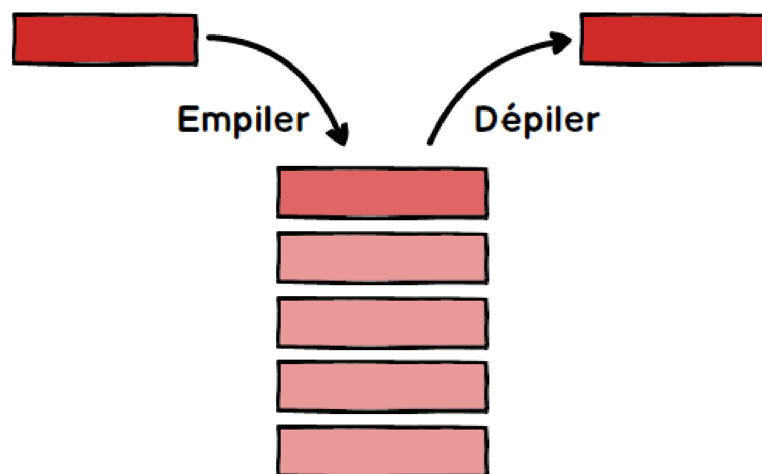

Figure 9 : Les piles

2. Opérations sur les piles

Les opérations sur les piles sont similaires aux opérations effectuées sur les files d'attente, la seule déférence, est la politique d'insertion et de suppression qui devient LIFO au lieu de FIFO. Ces opérations sont comme suit :

- Tester si la pile est vide
- Empiler un nouvel élément (ajouter un élément au sommet de la pile)
- Dépiler le dernier élément
- Consulter la valeur de l'élément au sommet de la pile sans le dépiler
- Connaître le nombre d'éléments dans la pile.

3. Enoncé du TP

On désire implémenter un logiciel qui gère une pile qui contient des médicaments dans une pharmacie :

- Donner la déclaration de la classe médicament qui compose la pile
- Donner la déclaration de la pile.
- Donner la fonction qui permet de tester si la pile est vide.
- Donner la fonction qui permet d'ajouter un élément dans la pile.
- Donner la fonction qui permet d'afficher les éléments de la pile.
- Implémenter la fonction qui permet de supprimer un élément de la pile.
- Donner la fonction qui permet de consulter la valeur du sommet de la pile.
- Donner la fonction qui retourne le nombre des éléments dans la pile.
- Donner la fonction qui permet de vider la pile.

- Les fonctions précédentes sont lancées en utilisant un menu « en utilisant la boucle switch » avec des numéros pour chaque commande permettant de choisir la fonction à exécuter en utilisant ce numéro.
- L'application est composée de deux classes seulement
    - La classe principale qui contient la méthode main ()
    - La classe médicament qui contient la déclaration des maillons de la pile.

4. Solution du TP

Déclaration de la classe

```
public class patient {
    String nom ;
    int num;
    public static int cmpt=0;
    patient next ;

    public String getNom() {
        return nom;
```

```
    }

    public int getNum() {
        return num;
    }

    public element getNext() {
        return next;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    public void setNum(int num) {
        this.num = num;
    }

    public void setNext(element next) {
        this.next = next;
    }

    public patient(String nom) {
        this.nom = nom;
        this.num = cmpt++;
    }
    public String toString() {
        return " patient {" + "nom=" + nom + ", num=" + num +'}';
    }
}
```

Déclaration de la liste

```
public  static  patient  head  =  null  ;
```

Tester si la liste est vide

```
public static boolean est_vide (){
        return head == null;
}
```

Ajouter un élément dans la liste

```
public static void ajouter(patient m ){
    if (est_vide ()){
        head=m;
        m.next=null ;
     }
    else {
            m.next=head;
            head=m;
            }
}
```

Afficher les éléments de la liste

```
public static void afficher (){
    patient h=head;
      if (est_vide ()){
       System.out.println("la liste est vide");
       Return;
    }
   Patient h=head;
    while(h!=null){
        System.out.println(h);
        h=h.next;
    }
}
```

Consulter le premier sommet de la file

```
    public static void consulter(){
      if (est_vide ()){
       System.out.println("la liste est vide");
      }
      else{ System.out.println(head);}
}
```

Supprimer un élément de la file d'attente

```
public static void supprimer(){
      if (is_empty()){
       System.out.println("la liste est vide");
    }
          else{ head = head.next;}
}
```

Compter du nombre des éléments dans la liste

```
public static void compter(){
    int compt=0;
    patient c=head;
    while (c!=null){
          compt++;
          c=c.next;
    }
     System.out.println(compt);
}
```

Vider la liste

```
public static void vider(){
```

```
   head=null;
}
```

Programme principale

```
public static void main(String[] args){
 int choix =0 ;
       do {
       System.out.println("pour tester la question 1 tapez 1");
       System.out.println("pour tester la question 2 tapez 2");
       System.out.println("pour tester la question 3 tapez 3");
       System.out.println("pour tester la question 4 tapez 4");
       System.out.println("pour tester la question 5 tapez 5");
       System.out.println("pour tester la question 6 tapez 6");
       System.out.println("pour tester la question 7 tapez 7");
       System.out.println("pour tester la question 8 tapez 8");
       choix= sc.nextInt();
       switch (choix) {
         case 1 : est_vide ();
                  break;
         case 2 : {
               System.out.println("entre le nom");
               String p=sc.next();
              patient m = new patient (p);
               ajouter(m);
                   };
                  break;
         case 3 : afficher();
                   break;
         case 4 : consulter();
                   break;
        case 5 : supprimer();
                   break;

         case 6 : compter();
                    break;
         case 7 : chercher();
                      break;
         case 8 : vider();
                       break;
            }
       }while (choix!=0);
   }
```

Les arbres

1. Définition

Les arbres sont des structures de données très utilisées en informatique dans plusieurs domaines à l'instar de l'organisation des systèmes de fichiers « composé de répertoires dont chacun est composé d'autres répertoires … ».
Un arbre est composé d'un ensemble de nœuds reliés entre eux par des branches « arrêtes », selon une organisation hiérarchique et l'ensemble est relié à une racine.
Leur usage est multiple car ils captent l'idée de hiérarchie, une faculté cognitive naturelle des êtres humains, permettant de simplifier la compréhension des problèmes par la division en sous problèmes plus petits.

Les nœuds dans un arbre contiennent des données représentant un ensemble de valeurs organisées hiérarchiquement. Les nœuds sont connectés entre eux par des arêtes « pointeurs » qui représentent la relation père/fils.

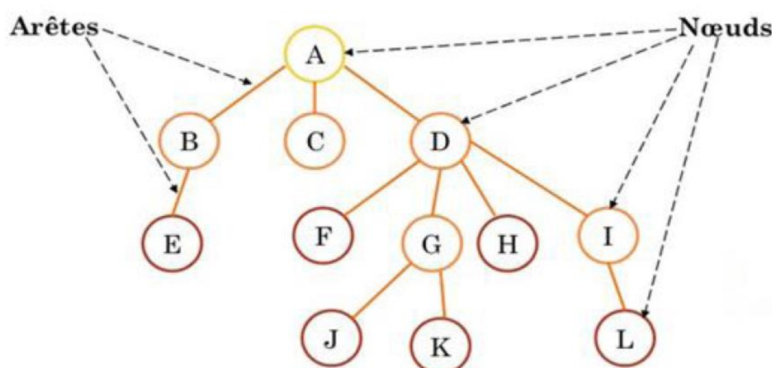

Figure 10 : Les arbres

2. Caractéristiques d'un arbre

- **Racine** : c'est le nœud qui n'a pas de prédécesseur (le père de tous les fils dans l'arbre).
- **Nœud interne** : tout nœud ayant au moins un fils et au moins un père.
- **Feuille « nœud externe » :** c'est un nœud qui n'a pas de fils.
- **Branche :** c'est une suite de nœuds connectées de la racine jusqu'à une feuille
- **Descendants d'un Nœud** : ce sont tous les nœuds qui se trouvent sur la branche allant de ce nœud jusqu'aux feuilles.
- **Ascendants d'un nœud :** ce sont tous les nœuds se trouvant sur la branche allant de ce nœud jusqu'à la racine.
- **Taille de l'arbre :** c'est le nombre de nœuds qu'il possède.
    - Taille de l'arbre de l'exemple = 12
    - Taille d'un arbre vide = 0
- **Degré d'un nœud :** c'est le nombre de ses fils, le degré de B est 1, le degré de D est 4.
- **Degré d'un arbre :** est le degré maximum de ses nœuds, le degré de notre exemple est 4.
- **Le niveau d'un nœud :** est la distance qui sépare ce nœud de la racine.
    - Niveau de la racine = 0

- Le niveau de chaque nœud est égal au niveau de son père plus 1

- Arbre m-aire: est un arbre d'ordre m et le degré maximum d'un nœud est égal à m.
- Arbre binaire : est un arbre où le degré maximum d'un nœud est égal à 2.

3. Arbre binaire

- Un arbre binaire est un arbre où chaque nœud a deux fils.
- Le premier fils d'un nœud n est appelé Fils gauche (FG) et le deuxième fils est appelé Fils droit (FD).

Un arbre binaire est dit strictement binaire si chaque nœud interne a exactement
2 fils différents de NIL.

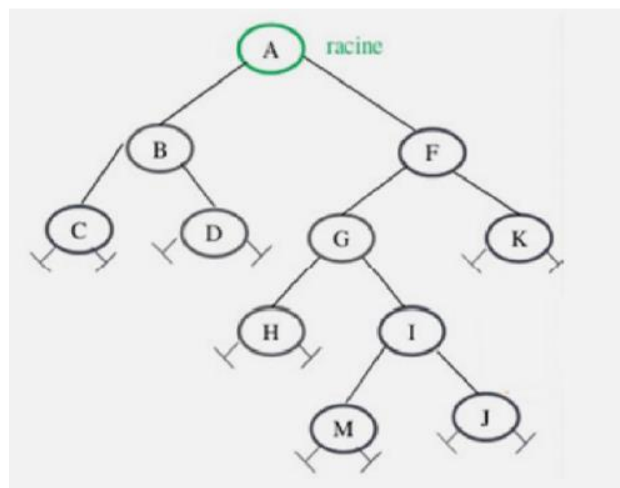

Figure 11 : Les arbres binaires

4. Parcours d'un arbre binaire

Le parcours d'un arbre binaire consiste à passer par tous ses nœuds pour effectuer des recherches ou bien pour insérer un nouvel élément dans un emplacement bien définit « suivant un ordre »
On distingue deux types de parcours :

- Des parcours en largeur explorent l'arbre niveau par niveau.
- Des parcours en profondeur explorent l'arbre branche par branche où on descend le plus profondément possible dans l'arbre puis une fois qu'une feuille est atteinte, on remonte pour explorer les autres branches.

Le parcours en profondeur peut se faire en:

- Le Préordre (Préfixe): où on affiche la racine avant ses fils (R, FG, FD)
- L'Inordre (Infixe): où on affiche FG puis racine puis FD (FG, R, FD).
- Le Postordre (Postfixe): où on affiche les fils avant la racine (FG, FD, R).

5. Arbre binaire de recherche

Un arbre binaire de recherche est un arbre binaire dans lequel chaque nœud possède une clé. Le fils droit d'un nœud donné est strictement inférieur à la clé de son nœud père et le fils gauche a

une clé strictement supérieure à la clé de son nœud père, les clés d'une valeur existante sont interdites.

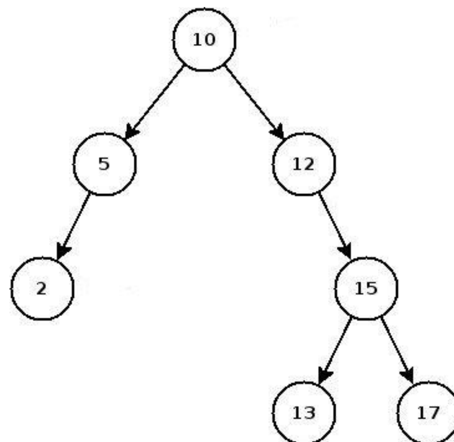

Figure 12 : Les arbres binaires de recherche

- **La recherche dans Un arbre binaire de recherche**

La recherche dans un arbre binaire d'un nœud ayant une clé particulière est un procédé récursif :

1) On commence par examiner la racine, si c'est la clé recherchée l'algorithme est terminé et on affiche la position de la clé recherchée.
2) Si la valeur de la clé recherchée est strictement inférieure à la valeur du nœud en question, on se déplace vers le fils gauche et on répète l'étape 1.
3) Si la valeur de la clé recherchée est strictement supérieure à la valeur du nœud en question, la recherche continue dans le fils droit et on répète l'étape 1.
4) Si on atteint une feuille dont la clé n'est pas celle recherchée, on conclut que la clé n'existe pas dans l'arbre.

- **Insertion d'un élément dans un ABR**

L'insertion d'un nœud commence par une recherche si la clé recherchée existe dans l'arbre ou non. Si la valeur existe on ne fait rien, sinon on continu la recherche jusqu'à arriver à une feuille, on ajoute un nouveau nœud. Ce dernier sera un fils droit de cette feuille si la clé est inférieure à la clé de la feuille ou un fils gauche si la clé est supérieure à la clé de la feuille.

- **Suppression d'un élément dans un ABR**

On commence par rechercher la clé du nœud à supprimer dans l'arbre.

Plusieurs cas sont possibles :

- Suppression d'une feuille : Il suffit d'enlever le nœud en question puisqu'il n'a pas de fils.

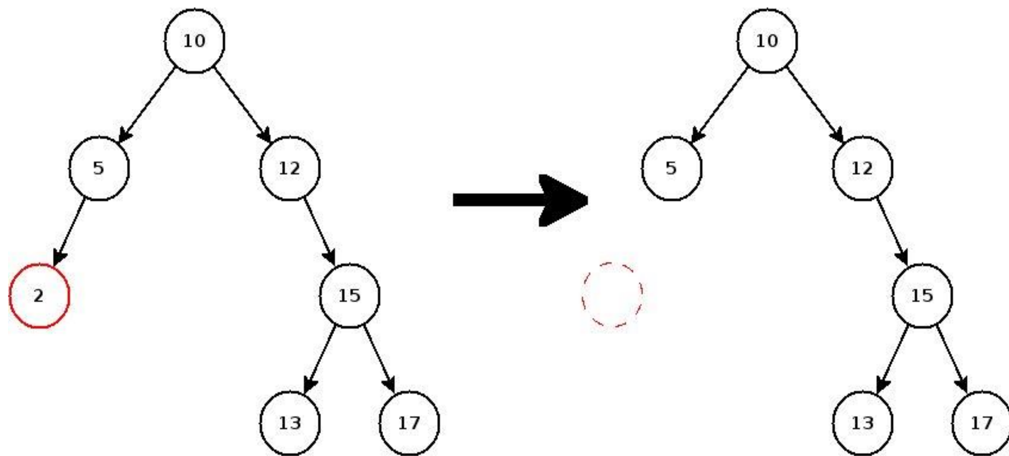

Figure 13 : Suppression des éléments d'un arbre binaire de recherche

- Suppression d'un nœud avec un enfant : on remplace le nœud en question par son fils.
- Suppression d'un nœud avec deux enfants : dans ce cas on ne peut pas simplement supprimer ou remplacer le nœud par son enfant.
  - On doit trouver la plus petite clé du sous-arbre droit et remplacer la valeur du nœud à supprimer par la valeur trouvée.
  - On remplace la clé du nœud à supprimer par la plus grande clé du sous arbre gauche

6. Enoncé du TP

On voudrait manipuler un arbre binaire de recherche qui contient des nombres entiers, donner l'implémentation des opérations suivantes :

- Tester si un arbre est vide
- Tester si un nœud est une feuille
- Parcourir les éléments d'un arbre
- Suppression d'un arbre
- Taille d'un arbre
- Nombre de feuilles d'un arbre
- Recherche d'un élément dans un arbre
- Parcourir les éléments d'un arbre
- La hauteur d'un arbre
- Le nombre des nœuds interne d'un arbre
- Le nombre des feuilles d'un arbre
- Tester si un arbre est complet
- Le parcours d'un arbre

- Les fonctions précédentes sont manipulées en utilisant un menu « en utilisant la fonction switch » avec des numéros pour chaque fonction permettant de choisir la fonction à exécuter en utilisant ce numéro.
- L'application est composée de deux classes seulement
  - La classe principale qui contient la méthode main()
  - La classe nœud qui contient la déclaration des maillons de la chaine.

7. Solution du TP

- **Déclaration d'un arbre**

La structure d'un arbre binaire de recherche va contenir :

- La clé « key »
- Pointeur vers l'élément gauche
- Pointeur vers l'élément droit

```
Public class nœud {
                Int key ;
                nœud gch ;
                nœud drt ;
   };
```

- **Tester si un arbre est vide**

```
bool EstVide(noeud elmt)
{
  If(elmt== null) return true;
   Else return false;
}
```

- **Création d'un arbre vide et tester si un arbre est vide**

```
Nœud arbre=NULL ;
```

- **Recherche d'un élément dans un arbre**

```
void chercher(noeud elmt, int key){
 while(elmt!= null) {
             if(key == elmt.clef) {
                                              System.out.println("cet element existe") ;
                                              return ;
                                   }
             if(key > elmt.clef )   elmt = elmt.drt;
             else    elmt = elmt.gch;
  }
  if (elmt== null)  System.out.println("cet element n' existe pas") ;

}
```

- **Tester si un nœud est une feuille**

```
bool est_feuille(noeud elmt){
if(elmt.drt== null && elmt.gch== null) return true;
else return false;
}
```

- **Nombre de feuilles d'un arbre**

```
int nombre_de_feuille(noeud elmt){
if(elmt==null) return 0;
if(est_feuille(elmt)) return 1 ;
else return nombre_de_feuille(elmt.drt)+ nombre_de_feuille(elmt.gch) ;
}
```

- **Parcourir un arbre**

- ✓ **Post fixe**

```
void parcourir(noeud elmt){
if(elmt.gch!= null) afficher(elmt.gch);
if(elmt.drt!= null) afficher(elmt.drt);
System.out.println("la cle est "+elmt.cle);
}
```

- ✓ **Préfixe**

```
 void parcourir (noeud elmt){
System.out.println("la cle est "+elmt.cle);
if(elmt.gch!= null) afficher(elmt.gch);
if(elmt.drt!= null) afficher(elmt.drt);
}
```

- ✓ **Infixe**

```
void parcourir (noeud elmt){
if(elmt.gch!= null) afficher(elmt.gch);
System.out.println("la cle est "+elmt.key);
if(elmt.drt!= null) afficher(elmt.drt);
}
```

- **Suppression d'un arbre**

```
void supprimer (noeud elmt){
if(elmt.gch!= null) supprimer (elmt.gch);
if(elmt.drt!= null) supprimer (elmt.drt);
delete elmt ;
}
```

- **Niveau d'un nœud**

```
void niveau(noeud  elmt, int key)
{
int Niveau=1 ;
 while(elmt) {
            if(key == elmt.clef) {
                                        System.out.println ("le niveau est" + Niveau) ;
                                    };
            if(key > elmt.clef ) {
                                            elmt = elmt.drt;
                                            Niveau++ ;
                                        }
            else {
                            elmt = elmt.gch;
                            Niveau++ ;
              }
  }
}
```

- **Taille d'un arbre « Nombre des nœuds d'un arbre »**

```
void taille(noeud *elmt){
if(elmt.gch!= null) taille (elmt.gch);
```

```
if(elmt.drt!= null) taille (elmt.drt);
taille++ ;
}
```

Récursive

```
Int taille(noeud elmt){
{
  if( elmt== null) return 0;
  else return 1 + taille (elmt.drt)+ taille (elmt.gch);
}
```

- **La hauteur d'un arbre**

```
int hauteur (noeud  elmt)
{
  if ( elmt==null ) return 0;
  else return 1 + max(hauteur (elmt.gch) , hauteur (elmt.drt);
}
//*******************************************
int max(int a,int b)
{
  If(a>b) return a;
   Else return b;
}
```

Références

- Zegour, D. E. (2026). *Structures de données : Piles, files et arbres binaires*. École Supérieure d'Informatique (ESI).
- Lafore, R. (2024). *Data Structures and Algorithms in Java* (2e éd.). Sams Publishing.
- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2024). *Structures de données et algorithmes en Java* (6e éd.). Wiley (adaptation française disponible).
- Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4e éd.). Addison-Wesley Professional.
- Slimani, T. (2019). *Programmation et structures de données avancées en langage C*. Éditions L'Harmattan.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2024). *Algorithmique* (4e éd.). Dunod.