1. Types de données abstraits

Un **type de données** est une collection d'objets qui ont des propriétés identiques indépendamment de toute représentation.

Un type de données abstrait est un type de données muni d'une signature (c'est à dire de fonctions de manipulation typées) et accompagné de sa sémantique (qui permet de décrire précisément son fonctionnement).

1.1.Les piles.

La pile est un type de données abstrait particulier dans lequel toute insertion ou suppression d'un élément se fait à une extrémité appelée *dessus* ou *sommet de la pile* . L'appellation anglo-saxonne consacrée pour les piles est "*LIFO list*" ou liste "*dernier entré premier sorti*". Ceci s'explique intuitivement en comparant une pile à une pile d'assiettes sur une étagère : si on se refuse à manipuler plus d'un élément à la fois, les seules actions possibles sont l'ajout (empilement) ou le retrait (dépilement) d'un élément.

Une application importante des piles est la mise en œuvre de la récursivité dans les applications. Tous les langages comme Caml ou Pascal qui permettent l'emploi de la récursivité utilisent une pile pour garder une trace de l'état de chaque variable de chaque procédure rendue active à un moment donné. Ces états portent le nom de contexte. Quand une procédure P est appelée, son contexte est placé au sommet de la pile. Si P appelle procédure P1, le contexte de P1 sera placé au sommet de la pile. A la fin de l'exécution de P, son contexte devra se trouver au sommet de la pile puisque ont ne peut revenir à P avant que toutes les autres procédures ne lui aient "rendu la main".

1.1.1.Implémentation en Java.

Il existe, dans le package java.util une classe Stack, descendante de la classe Vector qui implémente une pile. En voici la description :

Constructeurs	
Stack() Crée une pile vide	
Methodes	
`boolean`	**empty**() Teste si la pile est vide.
Object	**peek**() Renvoie l'objet au sommet de la pile, sans le dépiler.
Object	**pop**()

	Renvoie l'objet au sommet de la pile. Supprime cet objet de la pile.
Object	**push**(Object item) Place l'objet item au sommet de la pile.
int	**search**(Object o) Renvoie la position de l'objet o dans la pile..

Cependant pour bien comprendre le fonctionnement d'une pile, nous allons en construire une en Java. La classe Pile disposera des opérations suivantes :

opération	Commentaire
est_vide	renvoie vrai si la pile est vide
empile	place un élément dans la pile
depile	supprime un élément de la pile

1.1.1.1.A l'aide d'un tableau.

La méthode la plus simple consiste à utiliser un tableau pour stocker les éléments de la pile. Dans la classe ci-dessous, on utilise un tableau d'objets :

```
public class Pile {


  private Object[] elements; // Tableau d'objets
  private int count;         // Nombre d'objets dans la pile


  // Constructeur par défaut
  Pile() {
    elements = new Object[100];  // taille par défaut
    count = 0;
  }

  Pile(int taille) {
    elements = new Object[taille];
    count = 0;
  }

  public boolean est_vide() {
    return (count == 0);
  }

  public boolean est_pleine() {
    return (count >= elements.length);
  }


  public void empile(Object o) {
    if ( ! est_pleine() ) {
      elements[count] = o;
      count ++;
    }
  }

  public Object depile() {
    if ( ! est_vide() ) {
      count --;
      return elements[count];
    } else return null;
  }

}
```

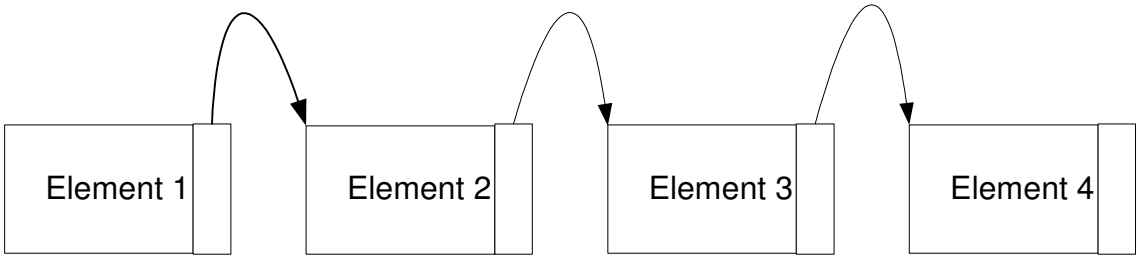
Voici un exemple d'utilisation de la classe Pile :

```
public class Essai {

 public static void main(String[] arg) {
   Pile p = new Pile(10);
   p.empile("la naranja ya esta seca");
   p.empile("amarillo esta el limon");
   p.empile("la sandia esta llorando");
   p.empile("esta riendo el melon");
   System.out.println(p.depile());
 }
}
```

1.1.1.2.A l'aide d'une liste chaînée.

Une solution plus élégante consiste à utiliser une liste chaînée. Une liste chainée est une structure de données dans laquelle chaque élément « pointe » sur l'élément suivant de la liste. Dans la figure ci-dessous, Element1 contient une référence vers Element2 qui contient une référence vers Elements3 qui contient une référence vers Element4 qui ne contient qu'une référence nulle.




```
class Element {

  private Object valeur;
  private Element suivant;

  Element() {
    valeur = null;
    suivant = null;
  }

  Element(Object o) {
    valeur = o;
    suivant = null;
  }

  public void setSuivant(Element e) {
    suivant = e;
  }

  public Element getSuivant() {
    return suivant;
  }

  public Object getValeur() {
    return valeur;
  }
}
```

```
public class Pile {


  private Element racine;


  // Constructeur
  Pile() {
    racine = null;
  }

  public boolean est_vide() {
    return (racine == null);
  }

  public void empile(Object o) {
    Element e = new Element(o);
    e.setSuivant(racine);
    racine = e;
  }

  public Object depile() {
    Element e = racine;
    racine = e.getSuivant();
    return e.getValeur();
  }

}
```

Et voici un exemple d'utilisation de la classe Pile :

```
public class Essai1 {

  public static void main(String[] arg) {
    Pile p = new Pile();
    p.empile("una casita muy redondida");
    p.empile("con dos ventanitas");
    p.empile("y una puertita");
    p.empile("y el timbre : Driiiiiin");
    System.out.println(p.depile());
  }
}
```

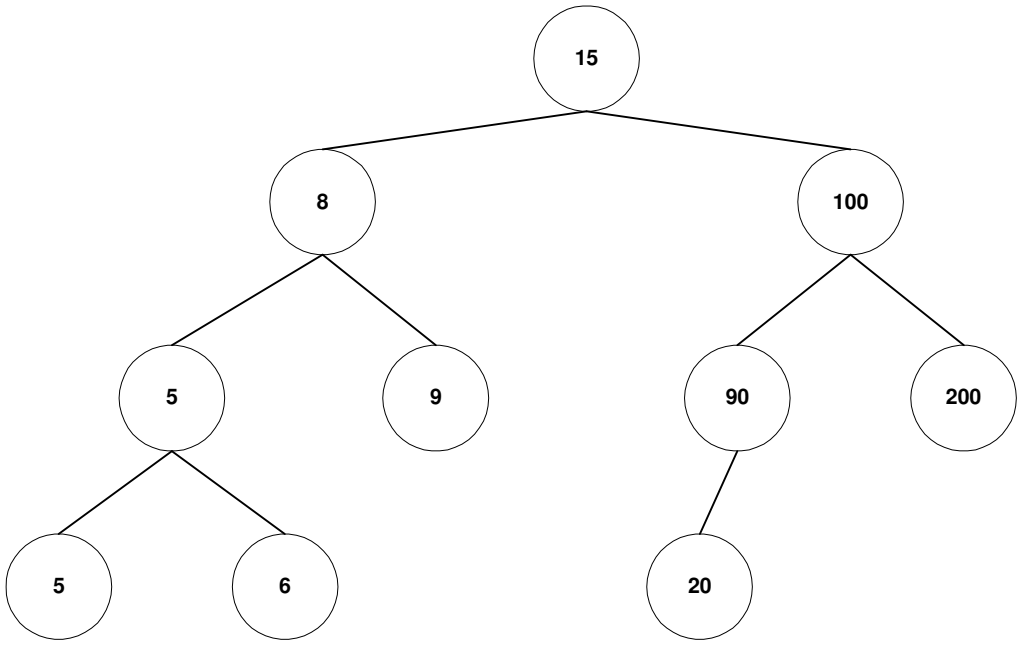
1.2.Les arbres.

Les listes et piles sont des structures dynamiques unidimensionnelles. Les arbres sont leur généralisation multidimensionnelle (une liste est un arbre dont chaque élément a un et un seul fils). On utilise un vocabulaire inspiré des arbres généalogiques. Chaque composante d'un arbre contient une valeur, et des liens vers ses *fil*s. On peut distinguer un *noeud*, qui est une composante ayant des fils, et une *feuille*, qui n'en possède pas (mais est fils d'un noeud) mais souvent on préfère utiliser un type noeud y compris pour les feuilles (tous les fils sont mis à NULL), surtout si l'arbre est évolutif (une feuille peut devenir un noeud, et inversement). Une

branche rassemble un lien vers un fils mais aussi ce fils et toute sa descendance. Le noeud *racine* est le seul qui n'est le fils d'aucun noeud (tous les autres noeuds sont donc sa *descendance*). Comme pour le premier d'une liste, l'adresse de la racine est nécessaire et suffisante pour accéder à l'intégralité d'un arbre. Un arbre N-aire ne contient que des noeuds à N fils (et des feuilles). Une liste est donc un arbre unaire. De nombreux algorithmes ont été développés pour les arbres binaires (nous verrons que tout arbre peut être représenté par un arbre binaire). Comme les listes, les arbres sont complètement dynamiques, mais les liens sont également gourmands en mémoire. L'accès aux données est encore séquentiel, mais on verra qu'il reste néanmoins rapide.

1.2.1.Implémentation en Java.

On va s'intéresser à un type d'arbre particulier. Un arbre binaire (donc chaque nœud possède au maximum deux fils). De plus, l'arbre est construit de telle manière qu'un élément quelconque de l'arbre est toujour supérieur ou égal à son fils gauche et inférieur à son fils droit.



1.2.1.1.A l'aide d'un tableau.

La méthode traditionnelle consiste à utiliser un tableau pour stocker les éléments d'un tel arbre.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
15	8	100	5	9	90	200	5	6			20								

On considère alors que les deux fils d'un élément placé au rang n du tableau se trouvent aux rang 2*n et 2*n + 1 du tableau. Par exemple, les deux fils de l'élément 100, placé au rang 3 du tableau se trouvent aux rangs 2*3 (6) et 2*3+1 (7).

Voici une écriture possible :

```
public class Arbre {


  private String[] elements;


  Arbre() {
    elements = new String[100];
  }

  Arbre(int taille ) {
    elements = new String[taille];
  }


  private void ajoute(String o, int indice ) {
    if (elements[indice]==null) elements[indice]=o ; else
    if (elements[indice].compareTo(o)<=0) ajoute(o,indice*2+1); else
    ajoute(o,indice*2);
  }
  public void add(String o) {

    ajoute(o,1);

  }

  private void imprime(int indice) {
    if (elements[indice]!=null) {
      imprime(indice*2);
      System.out.println(elements[indice]);
      imprime(indice*2+1);
    }
  }
  public void print() {
    imprime(1);
  }
}
```

```
public class Test {

  public static void main(String[] arg) {
    Arbre p = new Arbre();
    p.add("lunes");
    p.add("martes");
    p.add("miercoles");
    p.add("juvenes");
    p.add("viernes");
    p.add("sabato");
    p.add("domingo");
    p.print();
  }
}
```

1.2.1.2.A l'aide d'une liste chaînée.

Une autre solution plus élégante consiste à utiliser, comme pour la pile, une liste chaînée.

```
class Element {

  String valeur;
  Element gauche,droite;

  Element() {
    valeur=null;
    gauche=null;
    droite=null;
  }

  Element(String o) {
    valeur=o;
    gauche=null;
    droite=null;
  }

  public String getValeur() { return valeur;}
  public void setValeur(String o) { valeur = o;}

  public Element getGauche() {return gauche;}
  public void setGauche(Element e){ gauche = e;}

  public Element getDroite() {return droite;}
  public void setDroite(Element e){ droite = e;}


}

public class Arbre {


  private Element racine;


  Arbre() {
    racine = null;
  }


  private void ajoute(Element ou, Element quoi ) {
    if (ou.getValeur().compareTo(quoi.getValeur())>0) {
     if (ou.getGauche()==null) ou.setGauche(quoi); else
ajoute(ou.getGauche(),quoi);
    } else {
     if (ou.getDroite()==null) ou.setDroite(quoi); else
ajoute(ou.getDroite(),quoi);
    }
  }

  public void add(String o) {
    Element e = new Element(o);
```

```
    if (racine == null) racine = e ; else ajoute(racine,e);
  }

  private void imprime(Element quoi) {
    if (quoi!=null) {
      imprime(quoi.getGauche());
      System.out.println(quoi.getValeur());
      imprime(quoi.getDroite());
    }
  }
  public void print() {
    imprime(racine);
  }
}
```

```
public class Test {

  public static void main(String[] arg) {
    Arbre p = new Arbre();
    p.add("lunes");
    p.add("martes");
    p.add("miercoles");
    p.add("juvenes");
    p.add("viernes");
    p.add("sabato");
    p.add("domingo");
    p.print();
  }
}
```