Chap. 2: Architecture de logiciel

Architecture de logiciel:

- “Décomposition d’un système logiciel en plusieurs modules qui sont simples vus de l’extérieur et qui cachent la complexité à l’intérieur”.
- Le découpage et l’organisation des modules sont d’autant plus importants lorsque plusieurs personnes travaillent conjointement au développement du système logiciel.

Démarche:

- organisation par les structures de données
- modules avec des interfaces simples
- cacher l’implémentation des modules
- Principe: abstraire les structures concrètes vers des structures abstraites.

Plan du chapitre:

(1) Structure de données concrète

(2) Structure de données abstraite

(3) Type de structure abstraite ou type abstrait

(4) Classe

(ref: H.Mössenböck, Object-Oriented Programming in Oberon-2, Springer, 1994)

Modules logiciels: terminologie

- **fournisseur**: module qui **définit** et **exporte** une structure de données
- **client**: module ou unité de programmation qui **utilise** la structure de données

Remarque:

- en toute généralité, plutôt que de modules on parle de **composantes** pour désigner les unités logicielles qui une fois assemblées forment le système.
- Plusieurs normes pour l'**interface** des composantes ont été définies ces dernières années dont les plus utilisées sont CORBA, JavaBeans, COM/COM+, Business Objects.

(1) Structure de données concrète

Dans les langages de programmations plus ancien tel que le Pascal, toutes les structures de données sont visibles dans toutes les parties du programme
-> structures de données concrètes

Structure de données concrète:

- Principe: le client de la structure de données doit connaître la manière dont la structure de donnée a été déclarée dans le module fournisseur pour pouvoir l'utiliser.
- Le client doit donc connaître les détails de l'implémentation de la structure.

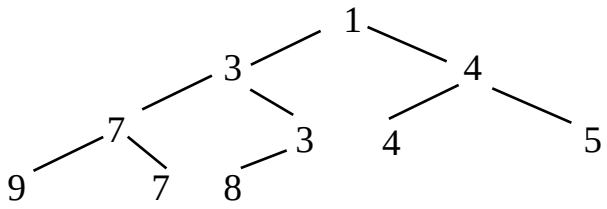
Ex. de structure de données: la queue prioritaire

Définition: “Structure à laquelle les éléments sont ajoutés dans n’importe quel ordre et retirés dans l’ordre de leur priorité.“

Une implémentation efficace: le “tas”, arbre binaire tq la valeur des noeuds parents est ≤ à celles des enfants (Sedgewick R., Algorithms. Addison-Wesley 1988). Il n’y a par contre pas d’ordre particulier entre les enfants.

Simplification: nous considérerons des éléments de type nombre entier qui expriment également leur priorité. Par convention, plus le nombre est petit, plus sa priorité est grande.

Illustration:



- En anglais: heap
- L’arbre est pseudo balancé: ∃ *h* tq toutes les feuilles ont une hauteur de *h* ou *h*-1
- Utilisation de la queue prioritaire en informatique: allocation des ressources, p.e. du processeur aux processus concurrents, queue d’impression (parfois).

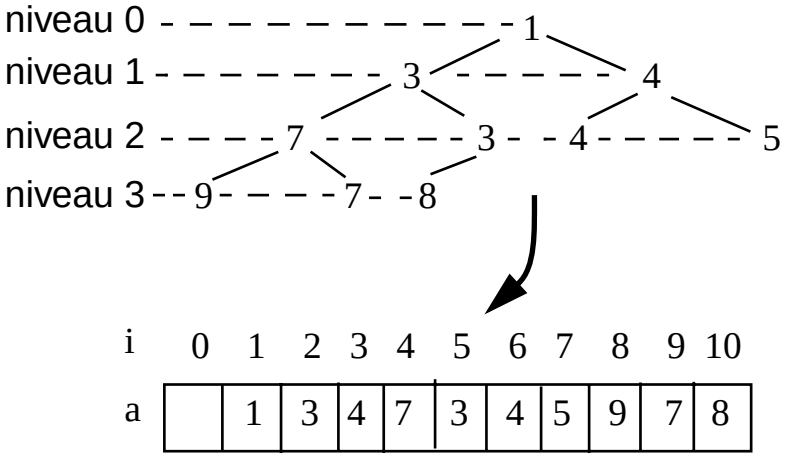
Implémentation concrète

Un tableau d'entiers peut être utilisé comme structure de données concrète.

Transformation arbre -> tableau:

les niveaux de l'arbre sont stockés séquentiellement dans le tableau

Exemple:



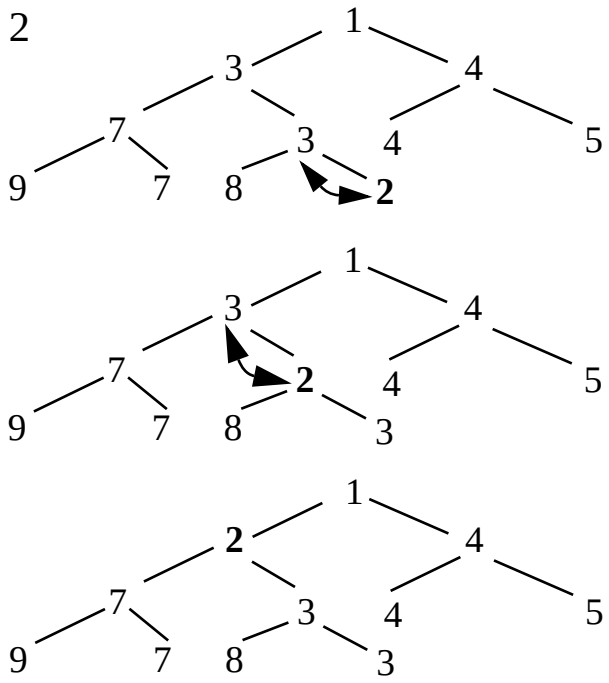
- Avantage: pas de pointeurs à stocker
- Accès aux enfants de a[i]: a[2∗i] et a[2∗i+1]
- Accès au parent de a[i]: a[i DIV 2]
- déclaration de la structure concrète:

```
VAR a: ARRAY 257 OF INTEGER; (* a[0] inutilisé *)
      n: INTEGER; (* nb d'éléments *)
```

Implémentation de l'opération "ajouter"

- Algorithme d'ajout à la queue:
  - le nouvel élément el est ajouté en a[n+1]
  - il est échangé avec le parent tant que l' élément ajouté < que son parent

Ex: ajout de 2



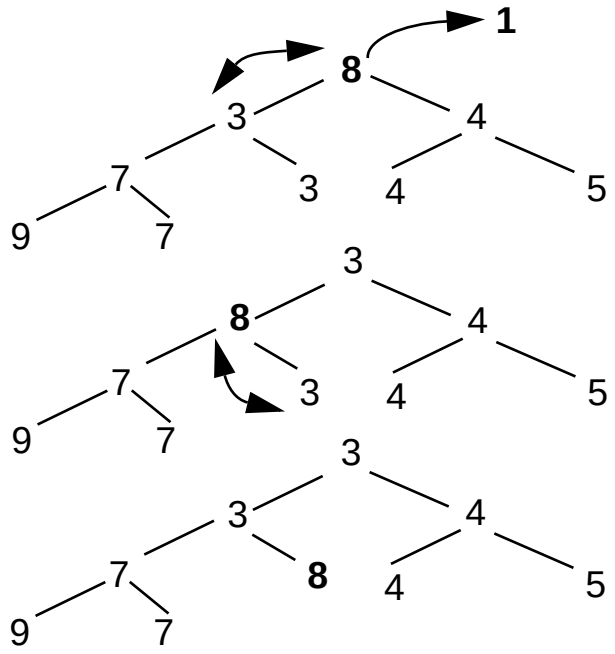
- en Oberon:

```
(* el est virtuellement ajouté en a[n+1] *)
n:=n+1; i:=n;
WHILE el < a[i DIV 2] DO
      a[i]:=a[i DIV 2]; i:=i DIV 2;
END;
a[i]:=el;
```

- Remarque: pour ne pas "sortir" du tableau lors de la boucle while, a[0] est initialisé à MIN(INTEGER) On appelle *sentinelle* un tel élément.

Implémentation de l'opération "retirer"

- Algorithme de "retirer" de la queue:
  - retirer a[1], a[1] est par définition l'élément prioritaire
  - déplacer a[n] en a[1]
  - propager a[1] vers ses enfants, càd l'échanger avec le plus petit de ses enfants tant qu'il est plus grand que ses deux enfants



- en Oberon:

```
el:=a[1]; (* el est l'élément retiré *)
y:=a[n]; (* 8 dans l'ex.*) n:=n-1; i:=1; ok:=FALSE;
WHILE (i <= n DIV 2) & ~ok DO
      j:=i + i; (* j:=2*i *)
      IF (j<n) & (a[j] > a[j+1]) THEN j:=j+1; END;
      IF y>a[j] THEN a[i]:=a[j]; i:=j; ELSE ok:=TRUE; END
END;
a[i]:=y;
```

Désavantages d'une approche "structure de données concrète":

- Les clients doivent connaître la manière dont est implémentée (déclarée) la structure de données et doivent être familier avec les algorithmes d'ajout (Ajouter) et de suppression (Retirer) d'éléments:
  -> le client est noyé par des détails d'implémentation
  -> le même code est présent dans plusieurs modules conduisant à une redondance du code (problèmes de maintenance)
  -> ayant un accès direct aux données, les clients peuvent par inadvertance détruire la consistance des données
- La modification de l'implémentation de la structure de données invalide tous les programmes qui utilise l'ancienne version de l'implémentation.

Remède: structure de données "abstraite".

(2) Structure de données abstraite

Définition: “Structure contenant les données et les opérations que l’on peut effectuer dessus”.

- c’est le même principe que celui des types de données de base (types de données primitifs)
- les opérations ainsi que la manière de les utiliser sont connues des clients
- l’implémentation de la structure de données ET les algorithmes implémentant les opérations sont cachés aux clients
- les données ne sont pas accessibles directement: les clients y accèdent par le biais des opérations définies sur la structure.

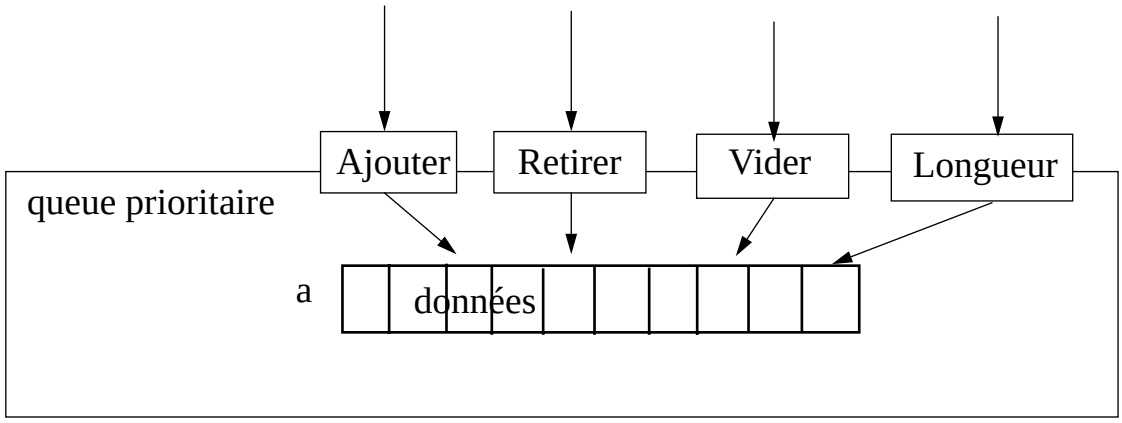
Remarque:

- le principe d’”abstraction” auquel on se réfère n’a pas la même signification que dans le sens commun. Par abstraction on entend le fait d’ignorer les détails pour ne se concentrer que sur un concept plus global, de niveau conceptuel plus élevé (ou plus “abstrait”). Cette technique intellectuelle est utilisée très fréquemment en informatiques et plus généralement dans les sciences. C’est l’arme la plus efficace pour lutter contre la complexité.

Structure de donnée abstraite: illustration

On qualifie la structure de données d'"abstraite" car seul son nom et son interface sont connus des clients, mais pas son implémentation.

Illustration de la structure de données abstraite "queue prioritaire":



Etat

- Une structure de données a un “état” qui est caractérisé par la valeur des données à un instant donné.
- L’état peut-être modifié par l’intermédiaire de certaines des opérations d’accès.

Implémentation d’une structure de données abstraite en Oberon

En Oberon, on fait usage d’un module qui exporte uniquement les procédures (implémentant les opérations). Les données sont cachées aux clients car non exportées.

Retour à notre exemple: La queue est donc implémentée par un module dont l’interface est:

```
DEFINITION QueuePrioritaire;
    PROCEDURE Ajouter (el: INTEGER);
    PROCEDURE Longueur():INTEGER;
    PROCEDURE Retirer (VAR el: INTEGER);
    PROCEDURE Vider;

END QueuePrioritaire.
```

Implémentation

Une implémentation possible du module queue prioritaire est:

```
MODULE QueuePrioritaire;

CONST long = 257; (* nb max d'elem = long -1 *)

VAR n: INTEGER;
    a: ARRAY long OF INTEGER;

PROCEDURE Vider*;
(* Clear *)
BEGIN
    n:=0; a[0]:=MIN(INTEGER);
END Vider;

PROCEDURE Ajouter* (el: INTEGER);
VAR i: INTEGER;
BEGIN
    IF n < long -1 THEN
        n:=n+1;
        i:=n;
        WHILE el< a[i DIV 2] DO
            a[i]:=a[i DIV 2]; i:=i DIV 2;
        END;
        a[i]:=el;
    END;
END Ajouter;
```

Implémentation (suite)

```
PROCEDURE  Retirer*(VAR el: INTEGER);
VAR y, i, j: INTEGER; ok: BOOLEAN;
BEGIN
      IF n > 0 THEN
            el:=a[1]; y:=a[n];
            n:=n-1; i:=1; ok:=FALSE;
            WHILE (i <= n DIV 2) & ~ok DO
                  j:=i + i;
                  IF (j<n) & (a[j] > a[j+1]) THEN j:=j+1; END;
                  IF y>a[j] THEN a[i]:=a[j]; i:=j;
                  ELSE ok:=TRUE; END
            END;
            a[i]:=y;
      END;
END Retirer;

PROCEDURE Longueur*(): INTEGER;
BEGIN
      RETURN n;
END Longueur;

BEGIN
      Vider;
END QueuePrioritaire.
```

Structure abstraite: Quelles opérations ?

Quelles opérations (procédures) inclure dans la structure de données abstraite ?

Critères:

- “suffisance”: un ensemble minimum d’opérations décrivant les caractéristique de la structure.

- “complétude”: toutes les opérations potentiellement utiles pour les clients.

- “opérations primitives”: les opérations dont une implémentation efficace exige la connaissance en détail de l’organisation effecive des données.

Trois types d’opération:

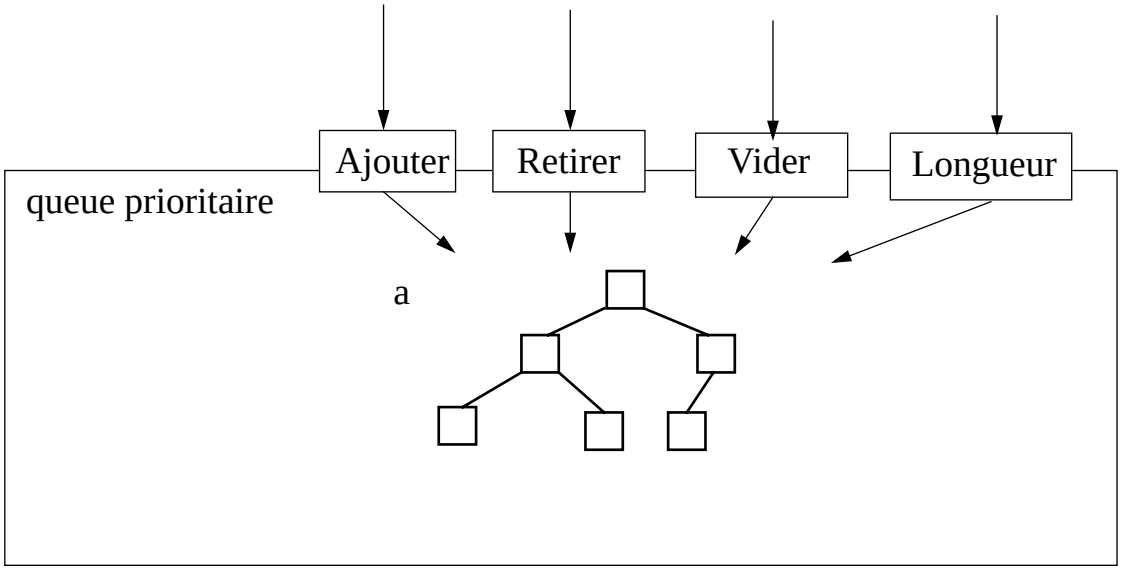
1) Modifieurs: opérations qui modifie l’état de la structure de données.

2) Sélecteurs: opérations qui renseigne sur l’état de la strcuture à un instant donné.

3) Itérateurs: opérations qui peremettent de visiter tous les éléments de la structure.

Changer l’implémentation de la queue prioritaire

Dans l’exemple ci-dessous, au lieu du tableau l’implémentation a recours à un arbre, mais l’interface n’a pas changé



-> les clients ne s’apercoivent pas du changement aussi longtemps que l’interface ne varie pas !

Avantages des structures de données abstraite:

Avantages

- Les clients n'ont pas besoin d'être "familier" avec l'implémentation de la structure. Par exemple, pour la queue prioritaire, il leur suffit de connaître l'interface (l'entête des procédures) et la fonctionalité (ce que fait la procédure).
- L'implémentation de la structure peut-être changée sans que cela n'affecte les clients (à condition de préserver l'interface).
- Les données sont "encapsulées" et cachées dans le module et protégées contre des suppressions involontaires.

Problème résiduel:

- une seule instance de la structure est ainsi définie. Parfois les clients ont besoin de plusieurs instances.

Solution:

- type de structure abstraite (ou plus simplement "type abstrait")

(3) Type de structure abstraite (ou type abstrait)

La déclaration est identique à celle d'une structure abstraite à trois nuances près:

- le **type** de la structure abstraite sera exporté
- la ou les structures de ce type seront déclarées par les clients et non par le module fournisseur
- les opérations (procédures)ont systématiquement un paramètre supplémentaire: la structure de données sur laquelle la procédure doit être appliquée.

Interface du type abstrait queue prioritaire en Oberon

```
DEFINITION QueuePrioritaire;
   TYPE
         QueueP = RECORD END;
   PROCEDURE Ajouter (VAR q: QueueP;el: INTEGER);
   PROCEDURE Longueur(q: QueueP):INTEGER;
   PROCEDURE Retirer (VAR q: QueueP;
                                    VAR el: INTEGER);
   PROCEDURE Vider(VAR q: QueueP);

END QueuePrioritaire.
```

Exemple d'utilisation de la structure de queue

Exemple d'utilisation de la structure queue prioritaire par un module client:

```
MODULE ExempleClient;

IMPORT QueuePrioritaire;

VAR q, r, s : QueuePrioritaire.QueueP; (* 3 queues *)
      el: INTEGER;
...
BEGIN
      ...
      QueuePrioritaire.Vider(q);
      QueuePrioritaire.Vider(r);
      QueuePrioritaire.Vider(s);
      QueuePrioritaire.Ajouter(q, 5); (* l'élément 5 est
                        ajouté à la queue q*)
      ...
      QueuePrioritaire.Retirer(q, el);
      QueuePrioritaire.Ajouter(r, el); (* ajouter à r
            l'élément qui vient d'être retiré de la queue q*)
      ...

END ExempleClient.
```

Implémentation du type abstrait queue prioritaire

```
MODULE QueuePrioritaire;
CONST long = 257; (* nb max d'elem = long -1 *)
TYPE QueueP* = RECORD
                    n: INTEGER;
                    a: ARRAY long OF INTEGER;
               END;

PROCEDURE Vider*(VAR q: QueueP);
(* Clear *)
BEGIN
    q.n:=0; q.a[0]:=MIN(INTEGER);
END Vider;

PROCEDURE Ajouter*(VAR q: QueueP; el: INTEGER);
VAR i: INTEGER;
BEGIN
    IF q.n < long -1 THEN
        q.n:=q.n+1;
        i:=q.n;
        WHILE el < q.a[i DIV 2] DO
            q.a[i]:=q.a[i DIV 2]; i:=i DIV 2;
        END;
        q.a[i]:=el;
    END;
END Ajouter;
```

Implémentation (suite)

```
PROCEDURE  Retirer*(VAR q: QueueP;
                          VAR el: INTEGER);
VAR y, i, j: INTEGER; ok: BOOLEAN;
BEGIN
    IF q.n > 0 THEN
        el:=q.a[1]; y:=q.a[q.n];
        q.n:=q.n - 1; i:=1; ok:=FALSE;
        WHILE (i <= q.n DIV 2) & ~ok DO
            j:=i + i;
            IF (j<q.n) & (q.a[j] > q.a[j+1]) THEN j:=j+1; END;
            IF y>q.a[j] THEN q.a[i]:=q.a[j]; i:=j;
            ELSE ok:=TRUE; END
        END;
        q.a[i]:=y;
    END;
END Retirer;

PROCEDURE Longueur*(q: QueueP;): INTEGER;
BEGIN
     RETURN q.n;
END Longueur;

END QueuePrioritaire.
```

Autres exemples de types abstraits

- le type abstrait “date”

- le type abstrait “poids”

Dans les deux cas, la représentation sur ordinateur (implémentation) de la donnée peut-être réalisée avec un entier (INTEGER)

- Pour les dates: nb de jours écoulé depuis une date de référence, p.ex depuis le 1.1.1900
- Pour les poids: poids exprimé en grammes
- Ces deux types ont le même domaine de valeurs
- Ces deux types ont des opérations différentes

Interface du type abstrait “Date”

```
DEFINITION Date;

    TYPE
        Date = INTEGER;

    PROCEDURE VersDate (VAR date: Date; jour,
                        mois, année: INTEGER);
    PROCEDURE JoursEntre (date1, date2: Date):
                        INTEGER;
    PROCEDURE MoisEntre (date1, date2: Date):
                        INTEGER;
    PROCEDURE PremierJourDuMois(date: Date):
                        String;
    PROCEDURE DateDuJour():Date;
    PROCEDURE VersString (date: Date): String;

    (* etc. *)

END Date.
```

Remarque:

- La procédure VersString convertit une date vers sa représentation textuelle, p.e. 0 -> 1 janvier 1900

Interface du type abstrait “Poids”

```
DEFINITION Poids;

    TYPE
        Poids = INTEGER;

    PROCEDURE VersPoids(VAR poids: Poids;
        p: REAL (* exprimé en kg avec décimales *));
    PROCEDURE Diff(poids1, poids2: Poids): Poids;
    PROCEDURE Somme(poids1,poids2:Poids):Poids;

    (* etc. *)

END Poids.
```

Problème avec cette version des types abstraits:

- Les données et les procédures ne forment pas une unité syntaxique:

  en effet, les procédures sont déclarées en dehors de la structure de données; il n'est pas toujours évident de voir à quel type de données appartient une procédure.

Solution: définir les types abstraits en utilisant les Classes

(4) Les classes

- En Oberon, il est possible de déclarer des procédures spéciales (appelées “méthodes”) qui sont syntaxiquement connectées à un type, plus précisément à un RECORD ou à un POINTER TO RECORD
- Les types RECORD qui “contiennent” des méthodes en plus des champs de données sont appelés des “classes”.
- Les instances (ou valeurs) de ces classes sont applées “objets”.
- Il y a d’autres différences majeures entre les RECORD/procédures standards et les classes:
  - les classes sont extensibles
  - les messages (appel des méthodes) sont liés dynamiquement aux méthodes.

Remarque:

- Nous verrons l’extensibilité et le liage dynamique au chapitre suivant.

Des types abstraits aux types abstraits avec classes

Idée de N. Wirth:

type abstrait:

```
TYPE QueueP* = RECORD
                 n: INTEGER;
                 a: ARRAY long OF INTEGER;
               END;

PROCEDURE Ajouter* ( VAR q: QueueP ; el: INTEGER);

PROCEDURE Longueur* ( q: QueueP ): INTEGER;
```

classe:

```
TYPE QueueP* = POINTER TO RECORD
                 n: INTEGER;
                 a: ARRAY long OF INTEGER;
               END;

PROCEDURE (q: QueueP) Ajouter*( el: INTEGER), NEW;
PROCEDURE (q: QueueP) Longueur*(): INTEGER, NEW;
```

- Le type du paramètre précéde le nom de la méthode indique à quel classe est liée la méthode
- BlackBox ajoute le qualificatif NEW (voir chapitre 3)
- Remarque: bien qu'il soit possible de définir des classes à l'aide des RECORD, il est préférable d'utiliser des pointeurs vers les RECORD. Dans ce cas le passage de paramètre est toujours par valeur.

Interface des classes en Oberon

Interface de la classe queue prioritaire:

```
DEFINITION QueuePrioritaire;
    TYPE
        QueueP = POINTER TO RECORD
              (q: QueueP) Ajouter (el: INTEGER), NEW;
              (q: QueueP) Longueur (): INTEGER, NEW;
              (q: QueueP) Retirer (VAR el: INTEGER), NEW;
              (q: QueueP) Vider, NEW

        END;
END QueuePrioritaire.
```

Remarques:

- les méthodes sont considérées comme des champs constants du RECORD
- elles sont accédées comme les champs d'un enregistrement, par ex. pour VAR q: QueueP, on écrira **q.Ajouter(el)** pour accéder à *Ajouter*
- on dit alors que le message *Ajouter* est envoyé à l'objet q. En effet, ce n'est pas un appel de procédure classique: c'est à l'exécution seulement qu'il sera décidé quelle méthode devra répondre à ce message (liage dynamique).
- l'objet auquel un message est envoyé est appelé le "receveur".
- le receveur est paramètre de chaque méthode; il précède le nom de la méthode.

Utilisation des classes en Oberon

Exemple d'utilisation de la classe queue prioritaire QueueP par un module client:

```
MODULE ExempleClient;

IMPORT QueuePrioritaire;

VAR q, r, s: QueuePrioritaire.QueueP;
    el: INTEGER;
...
BEGIN
    ...
    (*c'est au client d'instancier et d'initialiser les objets*)
    NEW(q); q.Vider();
    NEW(r); r.Vider();
    NEW(s); s.Vider();
    q.Ajouter(5); (* l'élément 5 est ajouté à la queue q *)
    ...
    q.retirer(el); (* l'élément 5 est ajouté à la queue q *)
    r.Ajouter(el); (* ajouter à r
          l'élément qui vient d'être retiré de la queue q*)
    ...
END ExempleClient.
```

Implémentation de la classe QueueP: remarques

- si un objet est référencé par un pointeur, il doit être instancié avant d'être en mesure de recevoir un message, i.e. l'objet doit exister avant d'être capable de traiter des messages -> le pointeur qui le référence ne doit pas être NIL.
- les objets étant déclaré dans le module client, c'est au client d'instancier les objets (avec NEW).
- -> bon principe de programmation: on écrit une méthode d'initialisation pour chaque classe. L'invocation de cette méthode suit immédiatement l'instruction d'instanciation de l'objet.

  Exemple avec la queue prioritaire q:

  NEW(q); q.Vider();

Implémentation de la classe queue prioritaire

```
MODULE QueuePrioritaire;

CONST long = 257; (* nb max d'elem = long -1 *)
TYPE QueueP* = POINTER TO RECORD
                              n: INTEGER;
                              a: ARRAY long OF INTEGER;
                         END;

PROCEDURE (q: QueueP)Vider*, NEW;
(* Clear *)
BEGIN
      q.n:=0; q.a[0]:=MIN(INTEGER);
END Vider;

PROCEDURE (q: QueueP) Ajouter*(el: INTEGER),
NEW;
VAR i: INTEGER;
BEGIN
      IF q.n < long -1 THEN
            q.n:=q.n+1;
            i:=q.n;
            WHILE el < q.a[i DIV 2] DO
                  q.a[i]:=q.a[i DIV 2]; i:=i DIV 2;
            END;
            q.a[i]:=el;
      END;
END Ajouter;
```

Implémentation (suite)

```
PROCEDURE (q: QueueP)Retirer*(VAR el: INTEGER),
NEW;
VAR y, i, j: INTEGER; ok: BOOLEAN;
BEGIN
    IF q.n > 0 THEN
        el:=q.a[1]; y:=q.a[q.n];
        q.n:=q.n - 1; i:=1; ok:=FALSE;
        WHILE (i <= q.n DIV 2) & ~ok DO
            j:=i + i;
            IF (j<q.n) & (q.a[j] > q.a[j+1]) THEN j:=j+1; END;
            IF y>q.a[j] THEN q.a[i]:=q.a[j]; i:=j;
            ELSE ok:=TRUE; END
        END;
        q.a[i]:=y;
    END;
END Retirer;

PROCEDURE (q: QueueP)Longueur*(): INTEGER,
NEW;
BEGIN
      RETURN q.n;
END Longueur;

END QueuePrioritaire.
```

Structures abstraites, types abstraits et types abstraits avec classes: comparaison

Où se trouvent les données et les procédures à l'exécution ?