# Rapport de Projet de Méthodologie de la Programmation

## Objectif du rapport

Ce rapport présentera le projet réalisé pour le module “Méthodologie de la Programmation” de l’ENSEEIHT Toulouse.  
Il abordera les différentes parties de la conception de ce projet. Il y sera présenté les choix techniques, les algorithmes utilisés, la structure du programme et les difficultés rencontrée.

## Introduction

Le projet proposé consiste à créer un système de gestion d’arbre généalogique d’un individue. L’arbre généalogique permet à un individu de répertorier l’ensemble de ses ancêtres. On peut représenter l’ensemble des ancêtres par un arbre où chaque ou chaque individu est représenté par un nœud qui peut avoir deux parents. > Exemple : > - Bernard > - pere : Jackie > - pere : Dominique > - mere : Jane > - mere : Dominique

On voudra pouvoir designer les individus en relation des uns par rapport aux autres avec la notion de génération d’ancêtre et de descendant > Exemple : > - Un individue est sont propre ancetre/decendant de generation 0 > - Le pere d’un individue est sont ancetre de generation 1 > - La grand mere d’un individue est son ancetre de generation 2 > - Le fils d’un individue est sont dedendant de generation 1

## Plan

Les modules étant déjà une sous-division structurante du projet du programme nous aborderont chaque aspect de conception pour chaque module.

Le rapport suivra le plan suivant :

[TOC]

## Architecture du programme :

Afin d’assurer une bonne lisibilité et une bonne modularité, le code du projet a été décomposé en plusieurs moules ada par rapport à leurs fonctions.

Dans cette partie les modules constituant ce projet seront présenté avec leur but. Nous nous intéresserons qu’aux modules appartenant au projet et nous omettrons les modules standard de ada.

Les modules développé sont : - **p\_arbre\_binaire** : Ce module générique contient l’entièreté des fonctions et des types nécessaire pour faire fonctionner un arbre binaire non trié. On y trouve le type génériqueT\_AB et les sous programmes permettant de le manipuler. *Paramètre de généricité* : - T : type des donnés - “=” : comparaison entre deux T - put(x :T) : procédure d’affichage de T - **p\_linked\_list** : Ce module générique contient les sous-programmes et les types afin de faire une liste chainée. Pour éviter de répéter du travail. On utilise un module développé dans le cadre des TPs *Paramètre de généricité* : - T : type des donnés - afficher(x :T) : procédure d’affichage de T - **p\_personne** : Ce module contient le type t\_personne qui représente un individu. Il y a aussi les différents sous-programmes permettant de le manipuler (init/getter/setter/comparaison/affichage). - **p\_arbre\_genealogique** : Ce module instancie p\_arbre\_binaire avec t\_persone et ses sous programme associé. Dans ce module on retrouve aussi toutes les procédures de manipulation d’un arbre généalogique qui ne n’ont pas de sens dans un arbre binaire non trié quelconque. Dans ce module on renomme T\_AB en T\_AG. Dans ce module on instancie aussi p\_linked\_list avec t\_personne car certaines fonctions ont besoin de renvoyer une liste d’individus. - **p\_front** : dans ce module on retrouve toutes les procédures et fonction d’entré/sortie nécessaire pour le programme principal(main). - **main** et le programme principal il contient la logique du menu et fait les appels à p\_arbre\_généalogique et à p\_arbre\_binaire en fonction des entré de l’utilisateur.

## Choix de conception

## Types

Dans cette partie nous nous intéresserons aux types contenue dans les différents modules.

### p\_arbre\_binaire :

Dans ce module on veut mettre en place un arbre binaire non trié. Un arbre binaire non trié structurellement n’est pas diffèrent de tout autre arbre binaire. Il est constitué de cellules contenant chacune deux références à d’autre cellule et un élément de donné

Pour ce faire il nous faut trois types : - **T\_Element**(privé) : Un type en paramètre de généricité du module. - **T\_AB**(visible privé) : Un pointeur vers une cellule de l’arbre(T\_AB\_Cell). Représente un arbre binaire. - **T\_AB\_Cell**(privé) : Un record qui représente une cellule d’arbre :

- gauche : T\_AB  
- droite : T\_AB  
- valeur : T\_Element

### p\_personne

Nous allons vouloir représenter des individus. Pour ce faire on utilisera un type record T\_Personne :

- id : integer  
- nom : unbounded\_string  
- prenom : unbounded\_string

Le type T\_personne n’est encore utilisé que pour contenir le nom et le prénom de l’individue mais le type pourrait être très facilement étendu en modifiant la déclaration du type.

### P\_linked\_list

Certains sous programmes nécessiterons une liste pour fonctionner. Il faudra donc un type T\_linked\_list qui représente une liste

Dans ce module on a 2 types :  
- T\_linked\_list\_cell : Un record qui constitue la liste

- value : T  
- next : T\_linked\_list

* T\_linked\_list : un pointeur vers T\_linked\_list\_cell.

### p\_arbre\_genealogique

Nous allons vouloir faire un arbre généalogique. On peut se servir de l’arbre binaire de p\_arbre\_binaire pour faire notre arbre généalogique. On utilisera le type T\_AG qui est une implémentation de T\_AB avec T\_personne renommé.

Les sous programmes de ce module nécessite de faire des listes de personnes nous implémenterons donc le module p\_linked\_list avec T\_personne pour obtenir une t\_linked\_list qu’on renommera en L\_personne.

## Fonctions Et Procédures

Dans cette partie nous nous intéresserons aux procédures et fonctions importantes de chaque module.

Nous omettrons certaines fonctions et procédures qui ne sont pas complexe. La spécification complète des packages avec la spécification des sous-programmes seront disponible en annexe.

### Fonctions et procédures de P\_arbre\_binaire :

#### Calculer la taille (calcul\_taille)

Pour calculer la taille d’un arbre binaire se calcule avec un algorithme de récursion. En effet on peut décomposer le problème en sommer la taille des deux sous arbre + 1.

##### Spécification

-- nom : calcul\_taille  
-- sémantique : renvoie nombre de nœuds de l’arbre   
-- paramètres :  
 -- arbre : in T\_AB -- Arbre dont on veut le nombre de nœuds  
-- retour : integer --nombre nœud de l’arbre  
-- pré-condition :   
-- post-condition :   
-- Tests de la procédure :  
function calcul\_taille(arbre : in T\_AB) return integer;

##### Rafinage

R0 : calculer le nombre de nœuds de l’arbre   
R1 : comment “calculer le nombre de nœuds de l’arbre” --arbre : in T\_AB  
ajouter le noeud courant + sosu arbre gauche + sous\_arbre droit  
R2 : comment “ajouter le noeud courant + sosu arbre gauche + sous\_arbre droit”  
 if(arbre = null) then  
 return 0;  
 else  
 return 1+ calcul\_taille(arbre.all.gauche) + calcul\_taille(arbre.all.droite)  
 end if;

#### Rechercher un nœud (rechercher)

Rechercher fais partie des fonctions qui change entre un arbre binaire trié et non trié. Dans un arbre non trié nous n’avons aucune information pour trouver l’élément. Il nous faut donc parcourir récursivement l’arbre pour trouver l’element.

##### Specification

-- nom : rechercher  
-- sémantique : rechercher un noeud dans un arbre  
-- paramètres : racine : in T\_AB --l’arbre où on fait une recherche  
 -- valeur : in T\_element -- le noeud à rechercher  
 -- arbre : in T\_AB -- Arbre dont on veut le nombre de nœuds  
-- retour : T\_AB -- le noeud  
-- pré-condition : l’arbre est initialisé  
-- post-condition : retourne le noeud ou null si le noeud existe pas  
-- Tests de la procédure :  
function rechercher(racine : in T\_AB ; valeur : in T\_element) return T\_AB

##### Raffinage

R0 : ”rechercher un noeud dans un arbre”  
R1 : comment “rechercher un noeud dans un arbre” -- arbre : in T\_AB ; valeur : in   
 if(arbre =null or else arbre.all.element = valeur) then  
 return arbre;  
 end if;  
 result := rechercher(arbre.all.gauche);  
 if(result = null)  
 result := rechercher (arbre.all.droite);  
 end if;  
 return result;

#### afficher arbre (afficher)

On cherche à reproduire la trace ci-dessou. Pour ce faire on va utiliser un algorithme récursif. Mais cependant on voit qu’afficher un sous arbre n’est pas la même chose qu’afficher un arbre en effet la tabulation varie en fonction de la profondeur il nous faudra donc passer la profondeur dans les paramètres. On fera aussi une récursion avec l’ordre racine->sous arbre gauche->sous arebre droit. ##### Trace

[1] Minet Bernard  
 Pere : [2] Minet Jack  
 Pere : [4] Minet Pierre  
 Mere : [5] Brosse Anne  
 Mere : [3] Legrand Jean

plus généralement :

<element>  
 <label-gauche> : <element>  
 <label-gauche> : <element>  
 <label-droit> : <element>  
 <label-droit> : <element>

##### Spécification

-- nom : afficher  
-- sémantique : affiche un arbre   
-- paramètres :  
 -- arbre : in T\_AB -- Arbre qu’on veut afficher  
 -- profondeur : in integer -- Profondeur de cette arbre pour affichage (déterminé le décalage de l’arbre)  
 -- etiquette\_gauche : in string(0..10) -- Étiquette pour la partie gauche   
 -- etiquette\_droite : in string(0..10) -- Étiquette pour la partie droite  
-- pré-condition :   
-- post-condition :   
-- Tests de la procédure :  
procedure afficher(arbre : in T\_AB; profondeur : in integer; ethiquette\_gauche, etiquette\_droite : in string(0..10))

##### Rafinage

R0 : “affiche un arbre”  
R1 : Comment “affiche un arbre” -- arbre : in T\_AB; profondeur : in integer;   
 -- etiquette\_gauche : in string(0..10);   
 -- etiquette\_droite : in string(0..10)   
 if(arbre = null) then raise arbre\_exception; end if;  
 afficher racine;   
 if(arbre.all.droite /= null )then   
 afficher sous arbre droit   
 end if;  
 if(arbre.all.gauche /= null )then   
 afficher sous arbre gauche   
 end if;  
R2 : Comment “afficher racine” -- arbre : in T\_AB;   
 afficher\_element(arbre.all.element);  
 new\_line;  
R2 : Comment afficher sous arbres droite -- arbre : in T\_AB; profondeur : in integer;   
 -- etiquette\_gauche : in string(0..10);   
 -- etiquette\_droite : in string(0..10)   
 for i in 1..profondeur loop  
 put(“ ”);  
 end loop;  
 put(etiquette\_droite);  
 put(“ : ”);  
 afficher(arbre.all.droite, profondeur+1, etiquette\_droite, etiquette\_gauche)  
R2 : Comment afficher sous arbres gauche -- arbre : in T\_AB; profondeur : in integer;   
 -- etiquette\_gauche : in string(0..10);   
 -- etiquette\_droite : in string(0..10)   
 for i in 1..profondeur loop  
 put(“ ”);  
 end loop;  
 put(etiquette\_gauche);  
 put(“ : ”);  
 afficher(arbre.all.gauche, profondeur+1, etiquette\_droite, etiquette\_gauche)

#### Supprimer élément(supprimer)

La suppression d’un élément est très similaire avec la recherche. On fait juste attention à rechercher avec une profondeur d’avance afin de pouvoir set le bon pointeur de sous arbre à null.

##### Spécification

-- nom : supprimer  
-- sémantique : suprime un noeud de l’arbre   
-- paramètres :  
 -- arbre : in T\_AB -- Arbre dont on veut suprimer un nœud  
 -- element : in T\_element -- Element à supprimer dans l’arbre  
-- pré-condition :   
-- post-condition :   
-- Tests de la procédure :  
procedure supprimer(arbre: in out T\_AB; element : in T\_Element)

##### Raffinage

R0 : suprimer un noeud de l’arbre   
R1 : comment “supprimer un noeud de l’arbre” -- arbre : in T\_AB; element : in T\_element  
 if(suprimer\_rec(arbre, valeur) = false) then  
 raise noeud\_absent  
 end if;  
R2 : comment “supprimer récursivement le noeud et ses antécédents" aka "suprimer\_rec"  
 -- arbre : in T\_AB; element : in T\_element  
 -- return : boolean  
 if(arbre=null) then  
 return false;  
 end if;  
 if(arbre.all.gauche /= null and then arbre.all.gauche.all.element = valeur) then  
 arbre.all.gauche:=null;  
 return true;  
 end if;  
 if(arbre.all.droit /= null and then arbre.all.droit.all.element = valeur) then  
 arbre.all.droit:=null;  
 return true;  
 end if;  
 res := supprimer\_rec(arbre.all.droit,valeur)  
 if(not res) then   
 res := supprimer\_rec(arbre.all.gauche, valeur);  
 end if;  
 return res;

### Fonctions et procédures de p\_arbre genealogique :

### Fonctions et procédures de main :