SAE 1.02

Comparaison d'approches algorithmiques

19 JANVIER 2024

SOFIANE ZEMRANI
OTMANE EL KRETE
RAYAN RIMANE
CORENTIN ROUX

Table des matières

I. Introduction

Nous sommes ravis de présenter ce rapport sur notre contribution au projet Star'But Valley. Confiant en notre capacité à relever les défis présentés par le cahier des charges, nous avons pris en charge le développement du jeu, mettant un accent particulier sur l'ajout d'un système de cuisine, la création d'un système de sauvegarde, et l'amélioration de la gestion des recettes.

Dans ce rapport, nous détaillerons nos choix de développement, de la conception de l'IHM pour le système de cuisine à l'intégration de nouvelles recettes depuis un fichier externe. Une analyse comparative des algorithmes de tri viendra également étayer notre approche.

Nous sommes conscients de l'importance d'une justification claire et pertinente, et c'est avec fierté que nous partageons notre vision et notre travail. Notre équipe est motivée par la rigueur et la créativité, et nous sommes impatients de vous présenter les résultats de notre engagement dans l'évolution de Star'But Valley.

II. Explication des algorithmes

Pour répondre à la problématique de cette SAE nous allons d'abord étudier les structures de donnée utiliser dans notre code.

A. Gestion des recettes

Dans un premier temps, nous souhaitons vous présenter la structure de données utilisée pour la gestion des recettes. À cette fin, nous avons opté pour l'utilisation d'une liste doublement chaînée, offrant une gestion rapide et efficace des données. Cette structure nous paraît être la plus adéquate et optimisée pour nos besoins.

Cette structure de données va nous permettre d'ajouter autant de recettes que voulu et nous pourrons également nous déplacer dans cette structure de données pour aller chercher les recettes précédentes et suivantes.

Pour utiliser une liste doublement chaînée, nous devons introduire la notion de pointeur. En Pascal, les pointeurs sont des variables qui contiennent des adresses mémoire. Ils permettent de manipuler des données en utilisant des références directes à la mémoire, offrant ainsi une flexibilité supplémentaire dans la gestion des données.

Nous allons maintenant passer à l'explication de chaque variable que nous avons ajoutée.

```
PRecetteNode = ^TRecetteNode;
```

Tout d'abord nous allons déclarer une variable de type pointeur.

```
TRecetteNode = record
  Recette: TRecette;
  Next: PRecetteNode;
  Prev: PRecetteNode;
end;
```

Ensuite nous allons créer une variable de type record qui contiendra la variable TRecette, un pointeur sur le suivant et un pointeur sur le précèdent objet dans la liste.

```
TListeRecettes = record
  Premier: PRecetteNode;
  Dernier: PRecetteNode;
end;
```

Nous allons déclarer une variable TListeRecettes qui permet de trouver le premier et le dernier élément de notre chaine grâce à des pointeurs.

1. Initialisation recettes

Nous allons maintenant passer à l'explication de notre initialisation qui permet de crée la liste doublement chainée.

```
procedure InitRecettes();
var
  Fichier: TextFile;
  Ligne: String;
begin
  InitialiserListe(ListeRecettes);
  try
    AssignFile(Fichier, NomFichier);
    Reset(Fichier);

    while not Eof(Fichier) do
    begin
      Readln(Fichier, Ligne);
      AjouterRecette(ListeRecettes, NouvelleRecette(extractLine(Ligne)));
    end;
  finally
    CloseFile(Fichier);
  end;
  //QuickSortList(ListeRecettes);
end;
```

La procédure débute par la déclaration des variables nécessaires, notamment un fichier texte (Fichier) et une chaîne de caractères (Ligne) pour stocker chaque ligne du fichier. La liste de recettes (ListeRecettes) est initialisée à l'aide de la fonction InitialiserListe.

La section principale du code utilise une structure de boucle pour parcourir le fichier texte. À chaque itération, une ligne du fichier est lue à l'aide de Readln et stockée dans la variable Ligne. Les données extraites de cette ligne sont ensuite utilisées pour créer une nouvelle recette, qui est ajoutée à la liste à l'aide de la fonction AjouterRecette.

La procédure prend également en charge la gestion des fichiers en ouvrant le fichier spécifié (AssignFile), en le lisant (Reset), et en le fermant après utilisation (CloseFile). La gestion des erreurs est assurée par la clause try...finally, garantissant la fermeture du fichier même en cas d'erreur.

2. Ajouter une recette

```
procedure AjouterRecette(var Liste: TListeRecettes; NouvelleRecette: PRecetteNode);
begin
  if Liste.Premier = nil then
  begin
    NouvelleRecette^.Prev := nil;
    Liste.Premier := NouvelleRecette;
    Liste.Dernier := NouvelleRecette;
  end
  else
  begin
    NouvelleRecette^.Prev := Liste.Dernier;
    Liste.Dernier^.Next := NouvelleRecette;
    Liste.Dernier := NouvelleRecette;
    Liste.Dernier^.Next := nil;
  end;

  NouvelleRecette^.Next := nil;
end;
```

La procédure AjouterRecette prend en charge l'intégration harmonieuse de nouvelles recettes dans la structure de données chaînée du jeu. Elle s'adapte intelligemment à deux scénarios clés :

Liste Vide :

- Si la liste est vide, la nouvelle recette devient le premier et le dernier élément de la liste.
- Les pointeurs Prev et Next de la nouvelle recette sont définis à nil, indiquant clairement sa position unique dans la liste.

Liste Non Vide :

- Si la liste n'est pas vide, la nouvelle recette est ajoutée à la fin de la liste.
- Les pointeurs Prev et Next sont mis à jour pour refléter correctement les liens entre la nouvelle recette, l'ancien dernier élément et la fin de la liste.

La gestion des recettes repose sur une structure de données réfléchie. Les nœuds (TRecetteNode) renferment une recette (TRecette), tandis que les pointeurs Next et Prev facilitent une organisation chaînée efficace. La liste elle-même (TListeRecettes) est définie par les pointeurs Premier et Dernier.

3. Avoir le nom de l’ingrédient

```
function GetNomIngredient(numRecette: integer; numIngredient: integer): string;
var
  Node: PRecetteNode;
  Count: integer;
begin
  Node := ListeRecettes.Premier;
  Count := 1;

  while (Node <> nil) and (Count < numRecette) do
  begin
    Inc(Count);
    Node := Node^.Next;
  end;

  if (Node <> nil) and (Count = numRecette) then
  begin
    if (Node^.Recette.ingredients[numIngredient].objet <> VIDE) and (Node^.Recette.ingredients[numIngredient].quantite > 0) then
      Result := IntToStr(Node^.Recette.ingredients[numIngredient].quantite) + 'x ' +
        ObjetToString(Node^.Recette.ingredients[numIngredient].objet)
    else
      Result := '';
  end
  else
    Result := '';
end;
```

La fonction commence par initialiser un pointeur de nœud (Node) avec le premier élément de la liste des recettes (ListeRecettes.Premier) et un compteur (Count) à 1.

À l'aide d'une boucle while, la fonction parcourt la liste des recettes jusqu'à ce que la recette spécifiée (numRecette) soit atteinte. À chaque itération, le compteur est incrémenté, et le pointeur de nœud est actualisé en conséquence.

Une fois la recette trouvée, la fonction vérifie sa validité en comparant le compteur (Count) avec le numéro de recette spécifié. Cela garantit que la recette existe dans la liste.

Si la recette est validée, la fonction procède à une vérification de l'ingrédient spécifié (numIngredient). Si cet ingrédient est non vide et possède une quantité supérieure à zéro, une chaîne de résultat est construite. Cette chaîne inclut la quantité de l'ingrédient suivie du nom de l'objet associé. Si l'ingrédient n'est pas valide, la chaîne de résultat est une chaîne vide.

Le résultat final de la fonction est la chaîne construite ou une chaîne vide si la recette ou l'ingrédient spécifié n'est pas trouvé.

4. Avoir le nom de la recette

```
function GetNomRecette(numRecette: integer): string;
var
  Node: PRecetteNode;
  Count: integer;
begin
  Node := ListeRecettes.Premier;
  Count := 1;

  while (Node <> nil) and (Count < numRecette) do
  begin
    Inc(Count);
    Node := Node^.Next;
  end;

  if (Node <> nil) and (Count = numRecette) then
    Result := Node^.Recette.nom
  else
    Result := '';
end;
```

La fonction débute en initialisant un pointeur de nœud (Node) avec le premier élément de la liste des recettes (ListeRecettes.Premier) et un compteur (Count) à 1.

À l'aide d'une boucle while, la fonction parcourt la liste des recettes jusqu'à ce que la recette spécifiée (numRecette) soit atteinte. À chaque itération, le compteur est incrémenté, et le pointeur de nœud est actualisé en conséquence.

Une fois la recette trouvée, la fonction vérifie sa validité en comparant le compteur (Count) avec le numéro de recette spécifié. Cela garantit que la recette existe dans la liste.

Si la recette est validée, la fonction extrait le nom de la recette à partir du nœud actuel (Node). Le nom est ensuite assigné au résultat de la fonction. Si la recette n'est pas valide, le résultat est une chaîne vide.

Le résultat final de la fonction est le nom de la recette ou une chaîne vide si la recette spécifiée n'est pas trouvée.

III. Complexité algorithmique

Nous allons à présent explorer la complexité algorithmique de chaque procédure et fonction que nous avons examinées précédemment. Pour commencer, introduisons la notion de complexité algorithmique. C'est une mesure théorique qui évalue la performance d'un algorithme en fonction de la taille de l'entrée. Son objectif est de quantifier la consommation de ressources (temps, mémoire) nécessaire à un algorithme pour résoudre un problème.

A. Gestion des recettes

1. Initialisation recettes

L'algorithme de la procédure InitRecettes vise à initialiser une liste de recettes en lisant les données à partir d'un fichier externe. Nous allons examiner la complexité de cet algorithme, en détaillant chaque étape majeure de son exécution.

La première opération, InitialiserListe(ListeRecettes), a une complexité constante (O(1)). Elle effectue des opérations sur la structure de données, indépendamment de la taille de l'entrée.

En ce qui concerne la gestion de fichiers (AssignFile, Reset, CloseFile), chaque opération a également une complexité constante (O(1)). Ces opérations gèrent le fichier et ne dépendent pas de la quantité de données.

La boucle While qui parcourt le fichier a une complexité linéaire (O(n)), où n est le nombre total de lignes dans le fichier. Chaque ligne est traitée une fois, dictant la complexité en fonction du nombre d'itérations.

L'opération AjouterRecette a une complexité constante en moyenne (O(1)), dépendant de l'implémentation interne. Ajouter une recette à la liste est généralement une opération constante, en particulier si elle implique une simple insertion dans une liste chaînée.

En combinant ces opérations, la complexité totale de l'algorithme est principalement dictée par la boucle While, soit O(n), où n est le nombre total de lignes dans le fichier.

En conclusion, l'algorithme de InitRecettes présente une complexité linéaire pour la lecture des données depuis un fichier, généralement une performance acceptable. Dans l'ensemble, cet algorithme est optimisé pour une lecture efficace des recettes.

2. Ajouté recette

La procédure `AjouterRecette` a pour objectif d'ajouter une nouvelle recette à une liste chaînée. Examinons la complexité de cet algorithme, en décomposant chaque étape importante.

La première condition vérifie si la liste est vide (`Liste.Premier = nil`). Si tel est le cas, l'ajout de la nouvelle recette se fait en temps constant (O(1)). La nouvelle recette devient le premier et le dernier élément de la liste.

En revanche, si la liste n'est pas vide, plusieurs opérations sont effectuées pour ajouter la recette à la fin de la liste. Ces opérations sont généralement de complexité constante (O(1)), car elles impliquent principalement des manipulations de pointeurs pour ajuster les liens entre les nœuds.

La complexité totale de `AjouterRecette` dépend principalement de l'état de la liste (vide ou non vide). Dans le pire des cas, lorsque la liste n'est pas vide, la complexité est constante (O(1)).

En conclusion, `AjouterRecette` est optimisé pour ajouter efficacement des recettes à une liste chaînée, avec une complexité qui reste constante dans la plupart des cas. Cela en fait une procédure efficiente pour la gestion d'une liste de recettes.

3. Avoir le nom des ingrédients

La fonction `GetNomIngredient` a pour objectif de récupérer le nom d'un ingrédient spécifique dans une recette à partir d'une liste chaînée de recettes. Examinons en détail la complexité de cet algorithme, en décomposant ses différentes étapes

La fonction commence par initialiser un nœud (`Node`) avec le premier élément de la liste. Elle utilise ensuite une boucle While pour parcourir la liste jusqu'à atteindre la recette souhaitée (`numRecette`) ou la fin de la liste. La complexité de cette boucle dépend du nombre de recettes dans la liste, noté n. Par conséquent, la complexité de la boucle est linéaire (O(n)), car elle peut potentiellement parcourir chaque recette dans la liste.

Ensuite, la fonction effectue une vérification conditionnelle pour s'assurer que la recette recherchée a été trouvée. Si oui, elle accède à l'ingrédient spécifique dans la recette, effectue des opérations constantes pour vérifier la validité de l'ingrédient, et construit la chaîne résultante. Dans le pire des cas, la complexité reste constante (O(1)) pour cette partie de la fonction.

Enfin, la fonction retourne le résultat construit ou une chaîne vide si la recette n'est pas trouvée. Cela aussi a une complexité constante (O(1))

En conclusion, la complexité totale de `GetNomIngredient` dépend principalement du nombre de recettes dans la liste (O(n)), avec des opérations constantes pour le reste des étapes. Cette fonction est raisonnablement efficace pour accéder à des informations spécifiques dans une liste chaînée de recettes.

4. Avoir le nom de la recette

La fonction `GetNomRecette` vise à obtenir le nom d'une recette spécifique à partir d'une liste chaînée. Examinons en détail la complexité de cet algorithme, en décomposant ses différentes étapes.

La fonction commence par initialiser un nœud (`Node`) avec le premier élément de la liste. Elle utilise ensuite une boucle While pour parcourir la liste jusqu'à atteindre la recette souhaitée (`numRecette`) ou la fin de la liste. La complexité de cette boucle dépend du nombre de recettes dans la liste, noté n. Par conséquent, la complexité de la boucle est linéaire (O(n)), car elle peut potentiellement parcourir chaque recette dans la liste.

Ensuite, la fonction effectue une vérification conditionnelle pour s'assurer que la recette recherchée a été trouvée. Si oui, elle accède au nom de la recette dans le nœud actuel. Dans le pire des cas, la complexité reste constante (O(1)) pour cette partie de la fonction.

Enfin, la fonction retourne le résultat construit ou une chaîne vide si la recette n'est pas trouvée. Cela aussi a une complexité constante (O(1)).

En conclusion, la complexité totale de `GetNomRecette` dépend principalement du nombre de recettes dans la liste (O(n)), avec des opérations constantes pour le reste des étapes. Cette fonction est raisonnablement efficace pour accéder au nom d'une recette spécifique dans une liste chaînée.

5. Les différents tris

1. Tri Rapide (QuickSort) :

- Complexité Moyenne : O(n log n)
- Pire Cas : O(n^2)
- Avantages : Ce tri est très efficace pour trier des tableaux/listes de moyenne et grande taille rapidement. De plus, il ne requiert pas d'espace mémoire supplémentaire car ne fait qu'échanger la place des cellules, il est donc intéressant à utiliser si l'on se soucis de l'espace mémoire.
- Inconvénients : Ce type de tri n'est pas stable. C'est à dire que sa complexité peut varier selon l'ordre des cases/cellules à trier. Dans le pire des cas, sa complexité est quadratique.

2. Tri Fusion (Merge Sort) :

- Complexité Garantie de O(n log n)
- Avantages : Le Tri Fusion est efficace pour trier rapidement des tableaux/listes de moyenne et grande taille rapidement, tout comme le quick sort. Il est stable, donc sa complexité reste la même peu importe l'ordre des données à trier, même dans le pire des cas.
- Inconvénients : Ce tri pose problème au niveau de l'espace mémoire car il nécessite un espace de stockage supplémentaire pour faire la fusion. On prend donc un risque de 'Stack Over Flow', donc de saturer sa mémoire si on essaye de trier un tableau ou une liste d'une grande taille.

3. Tri par sélection (Insertion Sort) :

- Complexité Garantie : O(n^2)
- Avantages : Le tri par insertion est efficace pour trier de petites quantités de données ou des données presque triées. Ce tri peut être adaptatif, c'est à dire qui peut être plus efficace si la liste est partiellement triée.
- Inconvénients : En dépit de sa complexité quadratique garantie, ce type de tri est très peu efficace pour des ensembles de données importants. Sa complexité peut ne pas être stable si l'implémentation n'est pas soigneusement effectuée.

Pour faire le choix de notre algorithme de tri, nous avons pris en compte toutes ses informations. Nous avons donc opté pour le quick sort pour les raisons qui suivent :

- Exclusion du tri par insertion : Tout d'abord, nous avons exclu l'utilisation du tri par sélection car n'étant pas efficace sur une quantité de données importante, nous avons rapidement déduit que ce ne serait pas un choix optimal pour trier notre liste doublement chainée de 5000 cellules.
- Inconvénient important du tri fusion : Le tri fusion aurait pu être une très bonne option pour trier notre liste si elle était moins importante. Le problème est que l'utilisation du tri fusion sur notre liste nous expose à une erreur de 'Stack Over Flow' (saturation de l'espace mémoire) ce qui est un risque que nous avons préféré ne pas prendre.

Le tri rapide (QuickSort) se distingue par sa complexité moyenne très efficace, O(n log n), ce qui en fait un choix optimal pour trier de grandes quantités de données. Bien que son pire cas puisse être moins performant, les implémentations modernes avec des pivots bien choisis minimisent cette occurrence.

Le tri rapide excelle dans la pratique grâce à son efficacité, surtout lorsque la taille des données est importante. Sa méthode de diviser et conquérir permet une utilisation optimale des ressources, ce qui en fait un choix de prédilection pour de nombreuses applications.

En conclusion, si la performance est une priorité et que l'on travaille avec de grandes quantités de données, le tri rapide est souvent le choix optimal. Son coût moyen est largement compensé par son efficacité, ce qui explique le choix que nous avons fait d'utiliser le tri rapide pour trier nos recettes par ordre alphabétique.

IV. Gestion de la sauvegarde

1. Fonctionnement de la sauvegarde

Nous avons implémenté une sauvegarde automatique qui sauvegarde à chaque fois que le joueur va dans la maison. Nous faisons cette sauvegarde en écrivant dans un fichier texte 'Sauvegarde.txt' le nom du joueur, le nom sa ferme, son nombre de stamina, mais également toutes les informations liées à la date. Voici un exemple des données écrites sur ce fichier.

```
C
C
100
1
PRINTEMPS
1
LUNDI
6
0
```

Les informations de ce fichier sont organisées de façon à ce que la première ligne soit le nom du joueur, le seconde soit le nom de sa ferme, la troisième soit le nombre de stamina, puis nous avons l'année, la saison, le numéro du jour, le jour, l'heure puis les minutes.

Pour récupérer la sauvegarde de notre ancienne partie nous lisons chacune des lignes de notre fichier de façon à récupérer nos informations.

2. Explication plus détaillée du fonctionnement de la sauvegarde

La sauvegarde à lieu dans l'UnitMaisonIHM avec la procedure AutoSave()

La sauvegarde à lieu dans l'UnitMaisonIHM avec la procedure AutoSave()

```
procedure AutoSave();
var
  FichierSauvegarde: TextFile;
begin
  try
    AssignFile(FichierSauvegarde, nomFichier);
    Rewrite(FichierSauvegarde);
    Writeln(FichierSauvegarde, UnitPersonnage.GetNomPersonnage());
    Writeln(FichierSauvegarde, UnitPersonnage.GetNomFerme());
    Writeln(FichierSauvegarde, UnitPersonnage.GetStamina());
    Writeln(FichierSauvegarde, UnitGestionTemps.GetDate().annee);
    Writeln(FichierSauvegarde, UnitGestionTemps.GetDate().saison);
    Writeln(FichierSauvegarde, UnitGestionTemps.GetDate().num);
    Writeln(FichierSauvegarde, UnitGestionTemps.GetDate().jour);
    Writeln(FichierSauvegarde, UnitGestionTemps.GetDate().heures);
    Writeln(FichierSauvegarde, UnitGestionTemps.GetDate().minutes);
  finally
    CloseFile(FichierSauvegarde);
  end;
end;
```

```
end.
```

Cette procédure a pour but d'écrire toutes les informations de notre joueur dans notre fichier sauvegarde.

La procédure est appelée dans la function EcranMaison() de façon à être appelée à chaque fois que le joueur se rend à la maison.

```
function EcranMaison() : TLieu;
var
  choix : integer; //Choix fait par le joueur
begin
  autoSave();
```

Pour récupérer les données sauvegardées, nous avons créé une fonction ChargerPartie() dans l'UnitMenuIHM. Lorsque le joueur se situe dans le menu il peut choisir l'option 'Charger' ce qui va appeler notre fonction ChargerPartie() et récupérer les informations dans le fichier de sauvegarde.