Algorithmique Avancée
Devoir de Programmation : Tries

Bielle Benjamin - 2900825

11 décembre 2014

Sommaire

Chapitre 1

Structures

Arbres de la Briandais

Parmi le code ASCII nous pouvons déterminer plusieurs marqeurs de fin de chain (ou de fin d'un mot). Par exemple, les caratères '*' ou '#' mais ici nous prendrons le caractère '\0' le marqueur de fin de chaine ainsi la programmation de notre projet en sera facilitée.

Pour nous aider dans la construction d'un arbre de la Briandais, nous allons utiliser des primitives de base :

- newBRDtree (clé : Mot, fils : BRDtree, frère : BRDtree).
  → retourne un arbre de la Briandais (BRDtree).

- buildBRDtree (m : Mot).
  → retourne un arbre de la Briandais construit à partir du mot m (BRDtree).

- addBRDtree (m : Mot, Tree : BRDtree).
  → ajoute un mot à un arbre de la Briandais (BRDtree).

- emptyBRDtree ().
  → retourne un arbre de la Briandais vide.

- isEmpty (arbre : BRDtree).
  → test si l'arbre est vide.

- head (m : Mot).
  → retourne la première lettre du mot m.

- tail (m : Mot).
  → idem à la fonction head, retourne le reste des lettres du mot m (tout sauf la premiŕe lettre).

Nous allons construire un arbre de la Briandais à partir du texte **exemple de base** :
"A quel genial professeur de dactylographie sommes nous redevables de la superbe phrase ci dessous, un modele du genre, que toute dactylo connait par coeur puisque elle fait appel a chacune des touches du clavier de la machine a ecrire ?"

Pour avoir une image plus graphique de cet arbre, veuillez voir la figure 1.1 page 4

Tries Hybrides

Comme pour l'arbre de la Briandais, nous allons définir des primitives nous permettant de constuire notre trie hybride :

- Hybrid (clé : Mot, valeur : Mot, inférieur : THybrid, supérieur : THybrid, égale : THybrid).
  → retourne un trie hybride (THybrid).

- emptyTHybrid ().
  → retourne un trie hybride vide.

- isEmpty (trie : THybrid).
  → test si le trie est vide.

- buildTHybrid (m : Mot).
  → retourne le trie hybride du mot m.

- addTHybrid (m : Mot, t : THybrid).
  → ajoute le mot m au trie hybride t.

Idem à la première partie (arbre de la Briandais), nous allons construire un trie hybride à partir du texte **exemple de base** :
Pour avoir une image plus graphique de ce trie, veuillez voir la figure 1.2 page 4

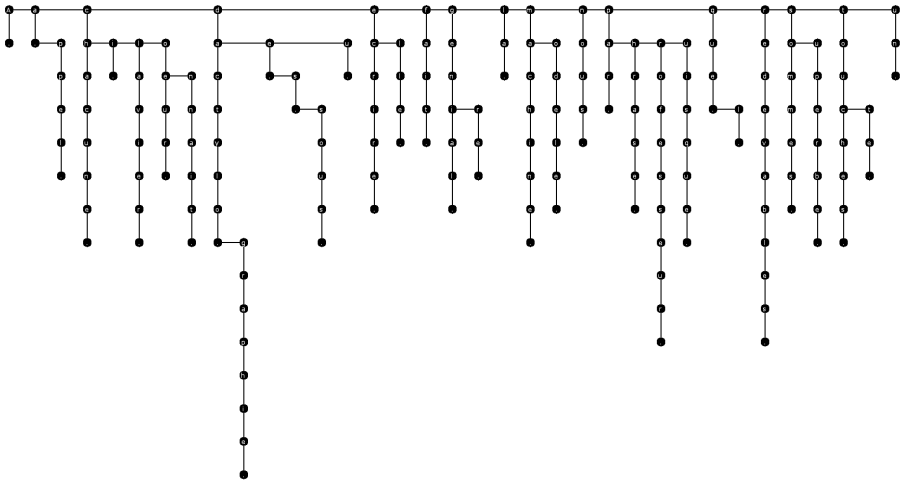

FIGURE 1.1 – Arbre de la Briandais

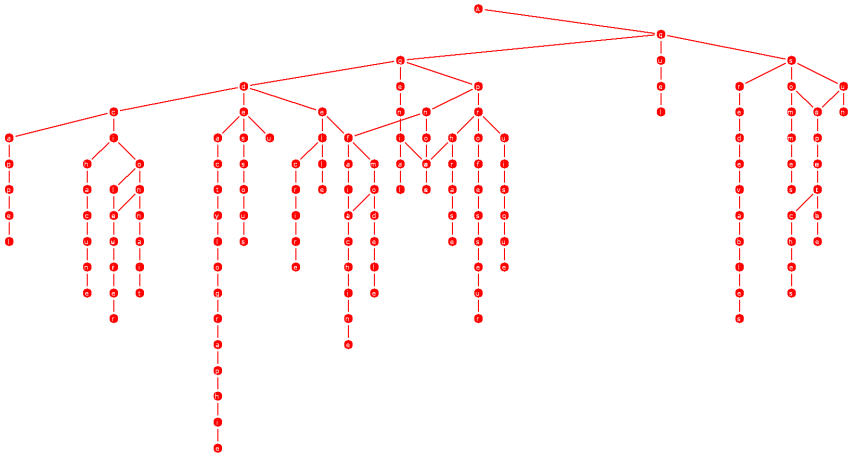

FIGURE 1.2 – Trie Hybride

Chapitre 2

Fonctions avancées pour chacune des structures

Fonctions pour un arbre de la Briandais

fonction de recherche d'un mot dans un dictionnaire : *Recherche(arbre,mot)* $\rightarrow$ *booléen.*

```
int searchBRDtree(char *word, BRDtree *Tree)
{
  if(strcmp(word, "") == 0)
  {
    return isEmpty(T);
  }
  else if(isEmpty(T)) return FALSE;
  if(head(word) < T->key) return FALSE;
  else if(head(word) == T->key) return searchBRDtree(tail(word), T->child);
  else return searchBRDtree(word, T->next);
}
```

fonction qui compte les mots présents dans le dictionnaire : *ComptageMots(arbre)* $\rightarrow$ *entier.*

```
int countWordsBRDtree(BRDtree *Tree)
{
  if(Tree == NULL)
     return 0;
  if(Tree->key == '\0')
     return 1 + countWordsBRDtree(Tree->next);

  return countWordsBRDtree(Tree->child) + countWordsBRDtree(Tree->next);
}
```

fonction qui liste les mots du dictionnaire dans l'ordre alphabétique : *ListeMots(arbre)* $\rightarrow$ *liste[mots].*

```
void addWordList(char *word, wordList **list)
{
   wordList *new = (wordList *) malloc(sizeof(wordList));
   new->word = word;
```

```
   new->next = *list;
   *list = new;
}

void insideListWordsBRDtree(BRDtree *Tree, wordList **list, char *word)
{
   if(Tree == NULL)
      return;
   int n = strlen(word);
   char *newWord = (char *) malloc(sizeof(char) * (n + 1));
   int i;
   for(i = 0; i < n; i++)
       newWord[i] = word[i];
   newWord[i] = Tree->key;
   if(Tree->key == '\0')
       addWordList(newWord, list);
   insideListWordsBRDtree(Tree->child, list, newWord);
   insideListWordsBRDtree(Tree->next, list, word);
}

wordList *listWordsBRDtree(BRDtree *Tree)
{
   wordList *list = (wordList *) malloc(sizeof(wordList));
   list->word = "";
   list->next = NULL;
   insideListWordsBRDtree(Tree, &list, "");

   return list;
}
```

fonction qui compte les pointeurs vers NIL : *ComptageNil(arbre)* $\rightarrow$ *entier*.

```
int countNullBRDtree(BRDtree *Tree)
{
  if(Tree == NULL)
     return 1;
  if(isEmpty(Tree))
     return 2;
  return countNullBRDtree(Tree->child) + countNullBRDtree(Tree->next);
}
```

fonction qui calcule la hauteur de l'arbre : *Hauteur(arbre)* $\rightarrow$ *entier*.

```
int heightBRDtree(BRDtree *Tree)
{
   if(isEmpty(Tree)) return 0;
   return max(1 + heightBRDtree(Tree->child), heightBRDtree(Tree->next));
}
```

fonction qui calcule la profondeur moyenne des feuilles de l'arbre : *ProfondeurMoyenne(arbre)* $\rightarrow$ *entier*.

```
void insideAverage(BRDtree *Tree, int *level, int *count, int currLevel)
{
  if(Tree == NULL) return;
  (*level) += currLevel;
  (*count)++;
  if(Tree->child != NULL)
     insideAverage(Tree->child, level, count, currLevel + 1);
  if(Tree->next != NULL)
     insideAverage(Tree->next, level, count, currLevel);
}

double averageLevelBRDtree(BRDtree *Tree)
{
  int  level = 0, count = 0, currLevel = 1;
  insideAverage(Tree, &level, &count, currLevel);
  if(count == 0) return 0.0;
  return (double) level / count;
}
```

fonction qui prend un mot A en argument et qui indique de combien de mots du dictionnaire le mot A est le préfixe : *Prefixe(arbre,mot)* $\rightarrow$ *entier*.

```
int countPrefixBRDtree(char *prefix, BRDtree *Tree)
{
   if(strcmp(prefix, "") == 0)
      return countWordsBRDtree(Tree);
   if(head(prefix) < Tree->key)
      return 0;
   else if (head(prefix) > Tree->key)
      return countPrefixBRDtree(prefix, Tree->next);
   else
      return countPrefixBRDtree(tail(prefix), Tree->child);
}
```

fonction qui prend un mot en argument et qui le supprime de l'arbre **s'il y figure** : *Suppression(arbre,mot)* $\rightarrow$ *arbre*.

```
BRDtree *delBRDtree(char *word, BRDtree *Tree)
{
   if(Tree == NULL)
      return NULL;
   if(isEmpty(Tree) && strcmp(word, "") == 0)
      return Tree->next;
   if(Tree->key == head(word))
   {
      BRDtree *child = delBRDtree(tail(word), Tree->child);
      if(child == NULL)
          return Tree->next;
      else
          return BRD(Tree->key, child, Tree->next);
   }
```

```
    else if(Tree->key > head(word))
       return Tree;
    else
       return BRD(Tree->key, Tree->child, delBRDtree(word, Tree->next));
  }
```

Fonctions pour un trie hybride

fonction de recherche d'un mot dans un dictionnaire : *Recherche(arbre,mot)* $\rightarrow$ *booléen.*

```
int searchTHybrid(char *word, THybrid *Trie)
{
  #ifdef VERBOSE
  printf("WD [%c]\n",*word);
  #endif
  if (Trie == NULL) return FALSE;
  else if (strcmp(word,"") == 0)
    {
      if (Trie->val == NOTEMPTY) return TRUE;
      else                       return FALSE;
    }
  else
    {
      if (*word < Trie->key)      return searchTHybrid(word, Trie->inf);
      else if (*word > Trie->key) return searchTHybrid(word, Trie->sup);
      else                        return searchTHybrid(++word, Trie->eq);
    }
}
```

fonction qui compte les mots présents dans le dictionnaire : *ComptageMots(arbre)* $\rightarrow$ *entier.*

```
int countWordsTHybrid(THybrid *Trie)
{
  if(Trie == NULL)
    return 0;
  else
    {
      if(Trie->val == EMPTY)
return countWordsTHybrid(Trie->eq)
  + countWordsTHybrid(Trie->inf)
  + countWordsTHybrid(Trie->sup);
      else
return 1 + countWordsTHybrid(Trie->eq)
  + countWordsTHybrid(Trie->inf)
  + countWordsTHybrid(Trie->sup);
    }
}
```

fonction qui liste les mots du dictionnaire dans l'ordre alphabétique : *ListeMots(arbre)* $\rightarrow$ *liste[mots].*

```
void insideListWordsTHybrid(THybrid *Trie, wordList **list, char *word)
{
  if (Trie == NULL) return;
  int size=strlen(word),i;
  char *nw = (char *)malloc(sizeof(char)*(size+1));

  for (i=0; i<size; i++)
    nw[i] = word[i];
  nw[i] = Trie->key;

  if (Trie->val == NOTEMPTY)
    addWordList(nw, list);

  insideListWordsTHybrid(Trie->eq, list, nw);
  insideListWordsTHybrid(Trie->inf, list, nw);
  insideListWordsTHybrid(Trie->sup, list, nw);
}

wordList *listWordsTHybrid(THybrid *Trie)
{
  wordList *list = (wordList *) malloc(sizeof(wordList));
  list->next = NULL;
  insideListWordsTHybrid(Trie, &list, "");
  return list;
}
```

fonction qui compte les pointeurs vers NIL : *ComptageNil(arbre)* $\rightarrow$ *entier*.

```
int countNullTHybrid(THybrid *Trie)
{
  if (Trie == NULL) return 1;
  else
    return  countNullTHybrid(Trie->eq)
      + countNullTHybrid(Trie->inf)
      + countNullTHybrid(Trie->sup);
}
```

fonction qui calcule la hauteur de l'arbre : *Hauteur(arbre)* $\rightarrow$ *entier*.

```
int heightTHybrid(THybrid *Trie)
{
  if (Trie == NULL) return 0;
  else
    return 1 + maxOfThree(heightTHybrid(Trie->eq), heightTHybrid(Trie->inf), heightTHybrid(Trie->sup));
}
```

fonction qui calcule la profondeur moyenne des feuilles de l'arbre : *ProfondeurMoyenne(arbre)* $\rightarrow$ *entier*.

```
void insideAverageTHybrid(THybrid *Trie, int *level, int *count, int currLevel)
{
  if(Trie == NULL) return;
  (*level) += currLevel;
```

```
  (*count)++;
  if(Trie->inf != NULL)
    insideAverageTHybrid(Trie->inf, level, count, currLevel + 1);
  if(Trie->eq != NULL)
    insideAverageTHybrid(Trie->eq, level, count, currLevel + 1);
  if(Trie->sup != NULL)
    insideAverageTHybrid(Trie->sup, level, count, currLevel + 1);
}

double averageLevelTHybrid(THybrid *Trie)
{
  int level=0, count=0, currLevel=0;
  insideAverageTHybrid(Trie, &level, &count, currLevel);
  if (count == 0) return (double) 0;
  return (double) level/count;
}
```

fonction qui prend un mot A en argument et qui indique de combien de mots du dictionnaire le mot A est le préfixe : *Prefixe(arbre,mot)* $\rightarrow$ *entier*.

```
int countPrefixTHybrid(char *prefix, THybrid *Trie)
{
  if (strcmp(prefix,"") == 0)
    {
      return countWordsTHybrid(Trie);
    }
  if (*prefix < Trie->key)
    {
      return countPrefixTHybrid(prefix, Trie->inf);
    }
  else if (*prefix > Trie->key)
    {
      return countPrefixTHybrid(prefix, Trie->sup);
    }
  else
    {
      return countPrefixTHybrid(++prefix, Trie->eq);
    }
}
```

fonction qui prend un mot en argument et qui le supprime de l'arbre **s'il y figure** : *Suppression(arbre,mot)* $\rightarrow$ *arbre*.

```
THybrid *delTHybrid(char *word, THybrid *Trie)
{
  if(Trie == NULL)
    return NULL;
  else if(strcmp(word,"") == 0)
    {
      if(Trie->val == NOTEMPTY)
{
```

```
  if(Trie->eq != NULL)
    {
      Trie->val = EMPTY;
    }
  else if(Trie->inf != NULL)
    {
      Trie->val = EMPTY;
      Trie->eq = Trie->inf;
      Trie->inf = NULL;
    }
  else if(Trie->sup != NULL)
    {
      Trie->val = EMPTY;
      Trie->eq = Trie->sup;
      Trie->sup = NULL;
    }
  else
    {
      free(Trie);
      Trie = NULL;
    }
}
      return Trie;
    }
  else
    {
      if(*word < Trie->key)
{
  Trie->inf = delTHybrid(word, Trie->inf);
}
      else if(*word > Trie->key)
{
  Trie->sup = delTHybrid(word, Trie->sup);
}
      else
{
  Trie->eq = delTHybrid(++word, Trie->eq);
}
      if(Trie->inf == NULL && Trie->sup == NULL && Trie->eq == NULL)
{
  free(Trie);
  Trie = NULL;
}
      else if(Trie->eq == NULL && Trie->inf != NULL)
{
  Trie->eq = Trie->inf;
  Trie->inf = NULL;
}
      else if(Trie->eq == NULL && Trie->sup != NULL)
```

```
{
  Trie->eq = Trie->sup;
  Trie->sup = NULL;
}
      return Trie;
    }
}
```

Chapitre 3

Fonctions complexes

fonction qui prend deux arbres de la Briandais en argument et les fusionnent en un troisième : *mergeBRDtree(arbre1,arbre2)* → *BRDtree.*

```
BRDtree *mergeBRDtree(BRDtree *Tree1, BRDtree *Tree2)
{
  if(Tree1 == NULL) return Tree2;
  if(Tree2 == NULL) return Tree1;

  if(Tree2->key < Tree1->key)
    return newBRDtree(Tree2->key, Tree2->child, mergeBRDtree(Tree1, Tree2->next));
  else if(Tree2->key > Tree1->key)
    return newBRDtree(Tree1->key, Tree1->child, mergeBRDtree(Tree1->next, Tree2));
  else
    return newBRDtree(Tree1->key, mergeBRDtree(Tree1->child, Tree2->child),
                  mergeBRDtree(Tree1->next, Tree2->next));
}
```

fonction permettant de passer d'un trie hybride à un arbre de la Briandais : *THybridToBRDtree (trie)* → *BRDtree.*

```
void tmpToBRDtree (THybrid *trie, BRDtree *tree, char *word)
{
  char *nw;
  int size,i;

  if (trie == NULL){return;}

  size = strlen(word);
  nw = (char *)malloc(sizeof(char)*(size+1));

  for (i=0; i<size; i++)
    nw[i] = word[i];
  nw[i] = trie->key;

  if (trie->val == NOTEMPTY)
    tree = addBRDtree(nw,tree);
```

```
  tmpToBRDtree(trie->eq, tree, nw);
  tmpToBRDtree(trie->inf, tree, word);
  tmpToBRDtree(trie->sup, tree, word);
}

BRDtree *THybridToBRDtree (THybrid *Trie)
{
  BRDtree *brd = emptyBRDtree();
  tmpToBRDtree(Trie, brd, "");
  return brd;
}
```

fonction permettant de passer d'un arbre de la Briandais à un trie hybride : *BRDtreeToTHybrid (arbre)* $\rightarrow$ *THybrid*.

fonction permettant de réquilibrer un trie hybride : *rebalancing (trie)* $\rightarrow$ *THybrid*.

Chapitre 4

Complexités

Arbres de la Briandais

fonction de recherche d'un mot dans un arbre de la Briandais, dans le pire cas on parcours L caractères (L correspond à la longueur du mot) donc notre complexité est en O(L).

fonction de comptage de mots, on parcours tous l'arbre (dans le pire et le meilleur cas) soit L caractères et pour chaque L on parcours sa hauteur soit ln N donc notre complexité est en O(L * ln N).

fonction liste de mots, on parcours simplement tous l'arbre donc comme pour le comptage de mots nous avons une complexité en O(L * ln N).

fonction de comptage des NULL, idem que comptage de mots.

fonction de calcul de la hauteur, on parcours tous l'arbre encore une fois donc notre complexité s'exprime en O(L * log N).

fonction de calcul de la profondeur moyenne de l'arbre, on parcours tous l'arbre donc comme pour les fonctions précédentes on exprime notre complexité en O(L * ln N).

fonction de comptage de prefix, cela est déterminée par la hauteur de l'arbre pour le prefix recherché donc nous avons une complexité en O(L + ln N).

fonction de suppression d'un mot dans l'arbre, cette fonction dépend aussi de la hauteur de l'arbre donc nous avons une complexité en O(log2 N).

Tries Hybrides

fonction de recherche d'un mot dans un trie hybride, dans le pire cas on parcours L caractères (L correspond à la longueur du mot) plus sa haurteur donc notre complexité est en O(L + ln N).

fonction de comptage de mots, on parcours tous le trie soit L caractères et pour chaque L on parcours sa hauteur soit ln N donc notre complexité est en O(L * ln N).

fonction liste de mots, on parcours simplement le trie donc notre complexité est en O(L * ln N).

fonction de comptage des NULL, idem que comptage de mots.

fonction de calcul de la hauteur, on parcours tous le trie donc nous avons une complexité exprimée en O(L * log2 N).

fonction de calcul de la profondeur moyenne du trie, on parcours tous le trie donc notre complexité est en O(L * ln N).

fonction de comptage de prefix, cela est déterminée par la hauteur de l'arbre pour le prefix recherché donc notre complexité est en O(L + ln N).

fonction de suppression d'un mot dans le trie, cette fonction dépend aussi de la hauteur du trie donc notre complexité est en O(log2 N).

La complexité en temps des opérations dans un arbre ternaire de recherche est similaire à celle d'un arbre binaire de recherche.
C'est à dire les opérations d'insertion, de suppression et de recherche ont un temps proportionnelle à la hauteur de l'arbre ternaire de recherche.
L'espace est proportionnelle à la longueur de la chaîne à mémoriser.

Résumé

	search	count words	list words	count Nil	height	average depth	prefix	delete word
Briandais	L	L * ln N	L * ln N	L * ln N	L * ln N	L * ln N	L + ln N	$\log_2 N$
Hybrid Trie	L + ln N	L * ln N	L * ln N	L * ln N	L * ln N	L * ln N	L + ln N	$\log_2 N$

Chapitre 5

Annexe

Structures de données

Ici nous détaillerons les différentes structures de données utilisées dans ce projet.

Arbres de la Briandais

La structure de données *BRDtree* est la structure définissant un arbre de la Briandais.

```
typedef struct BRDtree
{
   char key;
   struct BRDtree *child;
   struct BRDtree *next;
}BRDtree;
```

La structure de données *wordList* est la structure définissant une liste de mots pour un arbre de la Briandais.

```
typedef struct wordList
{
   char *word;
   struct wordList *next;
}wordList;
```

Tries Hybrides

La structure de données *THybrid* est la structure définissant un trie hybride.

```
typedef struct THybrid
{
   char key;
   char val;
   struct THybrid *inf;
   struct THybrid *eq;
   struct THybrid *sup;
}THybrid;
```

Table des figures