Compte Rendu SDD calendrier

Cliquot Théo & Marie Bornet

March 8, 2022

Contents

1 Présentation générale

1.1 Objet du TP

On gère un échéancier (ou un agenda) grâce à une liste chaînée à deux niveaux.

Le but de ce TP est de concevoir un emploi du temps consistant en des semaines et chaque semaine pouvant contenir plusieurs actions. On doit pouvoir réaliser des actions basiques avec ce calendrier, notamment afficher, ajouter ou supprimer des actions, charger ou sauvegarder un calendrier depuis un fichier ... Pour cela, on va se servir d'une structure de données déjà vue en cours : la liste chaînée.

1.2 Les structures de données

1.2.1 Action

L'action est caractérisée par un jour de la semaine et une heure, on donne à chaque action un nom permettant de l'identifier facilement en moins de 10 caractères.

Dans le cadre d'un calendrier, ne pouvoir contrôler qu'une seule action ne nous intéresse pas. C'est pour cela qu'on va préférer gérer une liste d'actions. Il va donc nous falloir, en plus des informations

propres à chaque action, un nouveau champ que l'on va nommer "suivant", qui va nous indiquer s'il y a une autre action de prévue après celle en cours.

```
typedef struct action {
  char day;             // Jour de la semaine (de 1 a 7)
  char hour[3];         // Heure (de 00 a 24)
  char name[10];        // nom de l'action
  struct action *suiv; // action suivante
} action_t;
```

En ce qui concerne la liste d'actions, on va utiliser une tête réelle qui pointera vers la tête de notre liste. Ainsi une liste d'actions n'est pas à proprement parler une structure, cependant on va utiliser un define pour la créer.

```
#define listAction_t action_t *
```

Une seule action peut avoir le même jour et la même heure dans la même liste (on peut avoir 2 cours sur le même crenau et le même jour par exemple)

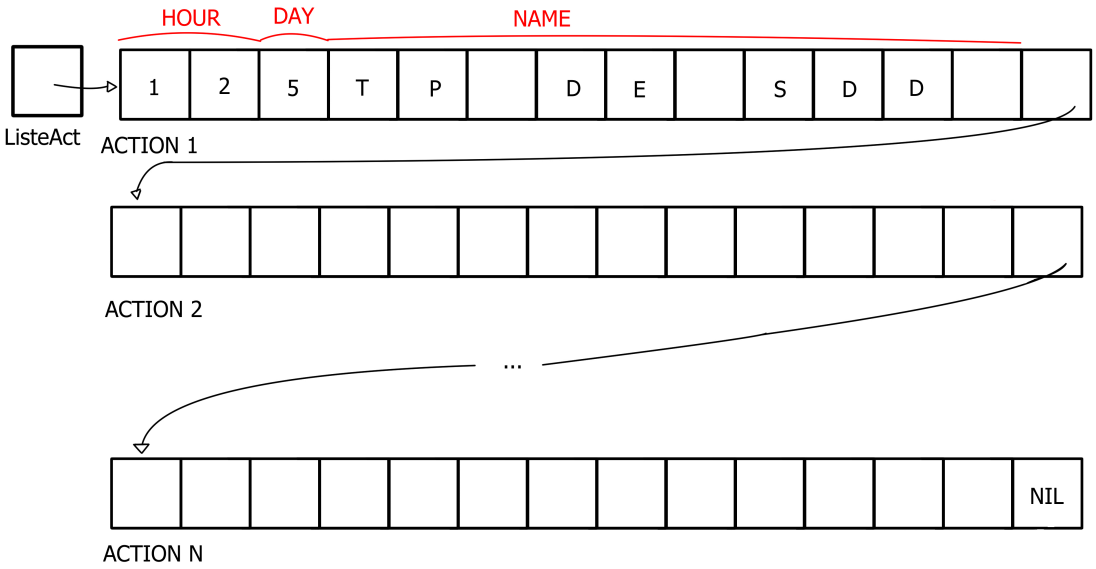

Figure 1: Schéma d'une action

1.2.2 Semaine

La semaine se rapproche très fortement dans sa construction avec la structure action. En effet, on va gérer une semaine de la même façon que l'on gère une action, c'est-à-dire qu'une semaine sera une cellule, ce qui nous permettra de créer une liste de semaines. Une semaine sera aussi unique dans une liste en fonction de son année et de son numéro.

Une semaine est constituée, comme dit précédemment, d'une année, d'un numéro de semaine mais aussi d'un pointeur vers la semaine qui suit celle en cours et d'une liste d'actions, représentant les actions à faire pour cette semaine.

```
  typedef struct week {
  char year[5];          // Annee de la semaine
  char numWeek[3];       // Jour de la semaine (01 a 53)
  struct week *suiv;     // Semaine suivante
  listAction_t listAct; // Actions de la semaine
} week_t;

```

```
8 #define listWeek_t week_t *
```

Un choix a été fait dans action et semaine de séparer le jour de l'heure (pour action) et l'année du numéro de semaine (pour la semaine) pour des raisons de modularité et d'affichage. L'inconvénient étant que cette séparation nous force, dans certaines parties de notre code, à dupliquer les comparaisons, ce qui a pour effet de l'alourdir. Elle nous semblait cependant avantageuse du point de vue de la compréhension globale du code, en plus d'être beaucoup plus simple à modifier si on voulait changer la façon dans nos attributs sont stockés.

En ce qui concerne la façon dont sont stockés les différents attributs, nous avons privilégié les caractères (char) aux entiers (int) pour des raisons d'occupations de la mémoire. En effet, 2 ou 3 caractères prennent moins de place qu'un entier, et ceux-ci suffisent amplement.

1.3 Les fichiers de données

Les seuls fichiers de données nécessaires dans notre cas (entrée comme sortie) sont ceux décrit dans l'énoncé

> fichier texte où chaque ligne donne, sans séparateur :
>
> année, semaine, jour, heure (sans espace), libellé de l'action sur 10 caractères (complété avec des espaces si besoin)
>
> Exemple de ligne : 202215108TPs de SDD

En voici un exemple que l'on a utilisé pour tester notre programme :

```
202215108TP DE SDD
202315108prog_fonc
201015108COUCOU
202215108TP DE SDD
200007504TP CSN
200007510TP CSN2
202315310prog_fonc2
```

1.4 Organisation du code source

Le code source est divisé en 4 fichiers (+ quelques fichiers headers pour connecter le tout). Les commentaires expliquant les fonctions se trouvent dans les headers pour des raisons de lisibilité.

1.4.1 Action.c

Ce fichier contient la déclaration de la structure action (définie plus tôt) ainsi que toutes les fonctions se rapportant à celle-ci, On a notamment les fonctions permettant d'allouer et de libérer une action, de la chercher, insérer ou supprimer d'une liste, de la comparer avec une autre et enfin d'afficher une liste d'actions de façon propre.

1.4.2 semaine.c

Ce fichier contient la déclaration de la structure semaine (définie plus tôt) ainsi que toutes les fonctions se rapportant à celle-ci. On a exactement les mêmes fonctions que celles vues précédemment pour les actions et pour les semaines, ainsi que quelques fonctions supplémentaires provenant du fait que les actions font partie des semaines. On a donc, en plus des fonctions pour chercher, supprimer ou insérer des actions dans une liste de semaines.

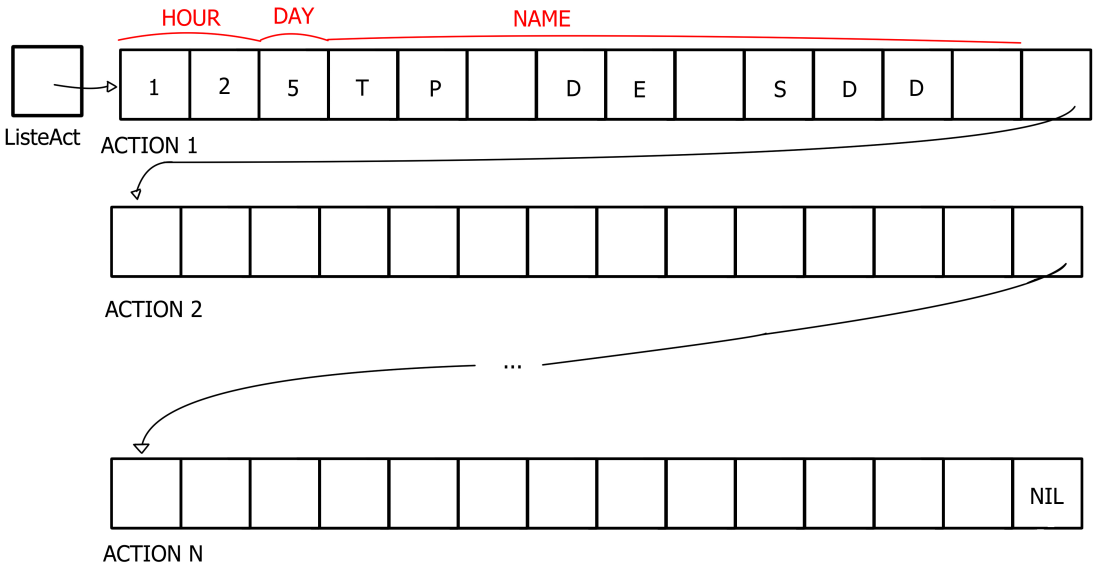

Figure 2: Schéma d'une semaine

1.4.3 main.c

Ce fichier contient Le menu afin de gérer de façon claire et simplifiée la gestion de notre calendrier. C'est notamment dans ce fichier que tout ce qui concerne la lecture ou écriture du calendrier depuis un fichier et faite. Pour le compiler il suffit d'utiliser la commande make afin d'appeler le makefile (cf : makefile).

1.4.4 test.c

Ce fichier contient tous les jeux de tests afin de vérifier que nos différentes fonctions produisent bien le résultat escompté. Pour le compiler il suffit d'appeler le makefile avec comme argument test (cf : makefile)

2 Fonctions

Afin d'avoir une explication générale des fonctions, de leurs paramètres et de leur sortie, un bloc de commentaire est déjà présent dans chaque headers. Cette section s'intéresse plus au corps des fonctions et explique l'algorithme ainsi que les différentes notations.

2.1 Fonctions actions

2.1.1 Sous procédures

1. checkDay

   CheckDay est une fonction présente pour simplifier la vérification et rendre plus lisible les conditions dans les autres fonctions. Elle vérifie seulement si le jour donné en argument **day** est cohérent.

```
1 int checkDay(char day) { return (day > '0' && day < '7'); }
```

2. checkHour

   CheckHour est exactement la même fonction que checkDay mais pour les heures

```
1   int checkHour(char hour[2]) {
2   return strcmp(hour, "00") >= 0 && strcmp(hour, "24") <= 0;
```

```
}
```

3. compareAction CompareAction va comme indiquer dans le bloc de commentaire renvoyer 1 si l'action donné en argument **act** viens avant où est égal chronologiquement avec la date de référence **day** et **hour**.

```
  int compareAction(action_t *act, char day, char hour[2]) {
  return act->day < day || (act->day == day && strcmp(act->hour,
   hour) < 0);
}
```

4. equalAction EqualAction se comporte comme compareAction, mais cette fois-ci elle renvoie vrai seulement si **act** est égal à **day** et **hour**

```
  int equalAction(action_t *a, char day, char hour[2]) {
  return a->day == day && strcmp(a->hour, hour) == 0;
}
```

2.1.2 Procédures

1. initListAction

   initListAction renvoie seulement NULL. En effet, étant donné que notre liste est une tête réelle, si notre liste est vide, on à donc notre tête qui ne pointe sur rien.

```
listAction_t initListAction() { return NULL; }
```

   >

2. newAction

   Le but de newAction est de créer une action à partir des informations nécessaires (**day**, **hour** et **name**) Dans un premier temps on va regarder si les informations fournies sont cohérentes. En effet, si elles ne le sont pas, rien ne sert de continuer. Dans ce cas, on renvoie simplement NULL (et un message pour préciser à l'utilisateur le problème). En cas de réussite, cette fonction va ensuite allouer l'espace mémoire nécessaire pour une action si les informations sont cohérentes. Si on a une erreur lors de l'allocation, on envoie un message d'erreur à l'utilisateur et on arrête le processus (s'il y a une erreur pour une allocation, il y a de grandes chances qu'un plus gros problème soit en train de se produire).

```
  action_t *newAction(char day, char hour[2], char name[10]) {
  action_t *nouv = NULL;

  // Si tout est correct, on alloue l'espace m moire n cessaire
  // Sinon on renvoie NULL;
  if (checkHour(hour) && checkDay(day)) {
    if ((nouv = (action_t *)malloc(sizeof(action_t)))) {
      nouv->day = day;
      strcpy(nouv->hour, hour);
      strncpy(nouv->name, name, 10);
      nouv->suiv = NULL;
    } else {
      printf("ERROR ALLOC DOESN'T WORK");
      exit(-1);
    }
  } else {
```

```
17      printf("INVALID HOUR OR DAY\n");
18    }
19    return nouv;
20  }
```

3. freeAction

   Dans le cas d'une simple action, il suffit juste de free cette dernière (cette fonction est surtout là car c'est une fonction que l'on doit créer pour de nombreux SDD).

```
1  void freeAction(action_t *act) { free(act); }
```

4. freeListAction Cette fonction va libérer une liste d'actions (en O(n), n étant la taille de la liste)

```
1   void freeListAction(listAction_t listAct) {
2   action_t *curr = listAct; // Un pointeur vers notre action
     actuelle
3   action_t *tmp;            // Action temporaire (celle qu'on va
     supprimer)
4
5   // On supprime la t te de liste et on avance jusqu'   arriver
      la fin
6   while (curr) {
7     tmp = curr;
8     curr = curr->suiv;
9     freeAction(tmp);
10  }
11 }
```

5. findAction Cette fonction va chercher dans une liste chaînée rangée la cellule correspondante et ce, en se servant de l'algorithme vu en cours de SDD. Elle retournera un pointeur qui pointe vers un pointeur d'action. Ce dernier contient l'action recherchée si elle existe dans la liste, sinon elle renverra l'action la plus petite supérieur à celle recherché. Dans le cas où les informations fournies sont incohérentes, on ne prend pas la peine de chercher et on renvoie directement NULL.

```
1    action_t **findAction(listAction_t *listAct, char hour[2], char
     day) {
2   action_t **prec =
3       NULL; // Pointeur d'un pointeur contenant l'action
     pr c dente
4   if (checkHour(hour) && checkDay(day)) {
5     prec = listAct;
6     action_t *curr = *listAct; // Un pointeur vers notre action
     actuelle
7
8     // Tant qu'on n'a pas trouv  l'action voulue et qu'on est
     avant
9     // chronologiquement
10    while (curr && compareAction(curr, day, hour)) {
11      prec = &(curr->suiv);
12      curr = curr->suiv;
13    }
14  }
15  return prec;
16 }
```

6. insertAction

   Cette fonction va insérer dans une liste d'actions une action si celle-ci est cohérente et s'il n'existe pas déjà dans la liste une action avec le même jour et la même heure. Cette dernière suit la même logique que l'algorithme vu en SDD et se sert de la fonction findAction décrite précédemment.

```
1  void insertAction(listAction_t *listAct, action_t *nouvAction) {
2
3  // Si notre nouvAction n'est pas correcte, pas besoin de l'
   ajouter
4  if (nouvAction != NULL) {
5
6    // Pointeur de pointeur d'action qui pointe vers l'action
   pr c dant celle
7    // voulue si elle existe sinon voir fonction findAction
8    action_t **prec = findAction(listAct, nouvAction->hour,
   nouvAction->day);
9
10   // Si une action existe d j  avec ce jour et heure,
11   // on ne l'ajoute pas et on le lib re de la m moire
12   if ((*prec) != NULL &&
13   equalAction(*prec, nouvAction->day, nouvAction->hour)) {
14     printf("WE␣ALREADY␣HAVE␣AN␣ACTION␣AT␣THIS␣HOUR␣AND␣DAY␣OF␣THE
   ␣WEEK\n");
15     freeAction(nouvAction);
16
17   }
18   // Sinon on l'ajoute dans notre liste
19   else {
20     nouvAction->suiv = (*prec);
21     *prec = nouvAction;
22   }
23 }
24 }
```

7. supprAction

   Cette fonction va supprimer une action fournie en argument dans une liste d'actions (si elle est dedans, sinon ne fait rien) On vérifie toujours que ce que l'on veut supprimer est cohérent, sinon on à pas besoin de chercher.

```
1  void supprAction(listAction_t *listAct, char hour[2], char day) {
2
3  if (checkHour(hour) && checkDay(day)) {
4    action_t **pprec =
5    findAction(listAct, hour, day); // Comme dans insertAction
6
7    // Si on a bien cette action dans notre liste
8    if (pprec != NULL && *pprec != NULL && equalAction(*pprec, day,
     hour)) {
9
10     action_t *tmp = *pprec; // Action temporaire
11     (*pprec) = (*pprec)->suiv;
12     freeAction(tmp);
13   }
14 }
```

```
15 }
```

8. prettyPrintListAction

   Une fonction afin de visualiser plus joliment le contenu de notre liste. On aurait aussi pu faire un prettyPrintAction et ensuite appeler cette fonction pour toutes les actions de la Liste, cependant cette fonction ne nous aurait pas plus servi que cela.

2.2 Fonctions semaines

La plupart des fonctions propres au semaine suivent les mêmes algorithmes que ceux vu précédemment avec les actions, il y a juste un changement, **day** et **hour** deviennent **numWeek** et **year**. C'est pour cela que l'on ne va pas les décrire entièrement. Seul insertActionInsideWeek et supprActionInsideWeek sont différents de ce que l'on a vu dans action.

2.2.1 Sous procédures

1. checkYear (similaire à checkHour)

```
1   int checkYear(char year[4]) {
2   return strcmp(year, "0000") >= 0 && strcmp(year, "9999") <= 0;
3 }
```

2. checkNumWeek (similaire à checkHour)

```
1   int checkNumWeek(char numWeek[2]) {
2   return strcmp(numWeek, "00") >= 0 && strcmp(numWeek, "52") <= 0;
3 }
```

3. compareWeek (similaire à compareAction)

```
1   int compareWeek(week_t *week, char year[4], char numWeek[2]) {
2   return strcmp(week->year, year) < 0 || (strcmp(week->year, year)
     == 0 &&
3                       (strcmp(week->numWeek, numWeek) < 0));
4 }
```

4. equalWeek (similaire à equalWeek)

```
1   int equalWeek(week_t *week, char year[4], char numWeek[2]) {
2   return strcmp(week->year, year) == 0 && strcmp(week->numWeek,
     numWeek) == 0;
3 }
```

2.2.2 Procedures

1. initListWeek(similaire à initListAction)

```
1 listWeek_t initListWeek() { return NULL; }
```

2. newWeek (similaire à newAction)

```
1   week_t *newWeek(char year[4], char numWeek[2]) {
2   week_t *nouv = NULL; // La nouvelle semaine allou  (null si
     incorrect)
3
```

```
  // Si nos arguments sont coh rents
  if (checkYear(year) && checkNumWeek(numWeek)) {

    // Si l'allocation c'est bien pass
    if ((nouv = (week_t *)malloc(sizeof(week_t)))) {
      strcpy(nouv->year, year);
      strcpy(nouv->numWeek, numWeek);
      nouv->suiv = NULL;
      nouv->listAct = initListAction();
    } else {
      printf("ERROR ALLOC DOESN'T WORK");
      exit(-1);
    }
  } else {
    printf("INVALID YEAR OR WEEK\n");
  }

  return nouv;
}
```

3. freeWeek

   Il faut faire attention car dans freeWeek, on doit bien entendu libérer la place qu'on a utilisé pour la semaine mais avant cela bien penser à supprimer toute la place prise par notre liste d'actions.

```
  void freeWeek(week_t *week) {
  freeListAction(week->listAct); // On lib re en premier la liste
   d'actions
  free(week);                    // Puis la semaine en elle m me
}
```

4. freeListWeek (similaire à freeListAction)

```
  void freeListWeek(listWeek_t week) {
  week_t *curr = week; // Un pointeur vers la semaine actuelle
  week_t *tmp;         // Un pointeur de semaine temporaire
  while (curr) {
    tmp = curr;
    curr = curr->suiv;
    freeWeek(tmp);
  }
}
```

5. findWeek (similaire à findAction)

```
  week_t **findWeek(listWeek_t *listWeek, char year[4], char
   numWeek[2]) {
  week_t **prec = NULL; // Un pointeur de pointeur de semaine
   pointant vers la
            // semaine pr c dent celle recherch e si elle existe
            // (sinon : voir bloc de commentaires dans le header)

  // Si nos argument sont corrects
  if (checkYear(year) && checkNumWeek(numWeek)) {
    week_t *curr = *listWeek; // pointeur vers la semaine actuelle
```

```
    prec = listWeek;

    // Tant qu'on a pas trouv  la bonne semaine ou une plus grande.
    while (curr && compareWeek(curr, year, numWeek)) {
      prec = &(curr->suiv);
      curr = curr->suiv;
    }
  }
  return prec;
}
```

6. insertWeek (similaire à insertAction)

```
week_t **insertWeek(listWeek_t *listWeek, week_t *nouvWeek) {
week_t **prec; // Comme dans findWeek

if (nouvWeek != NULL) {
  prec = findWeek(listWeek, nouvWeek->year, nouvWeek->numWeek);

  // S'il existe d j  une liste dans ce cr neau.
  if ((*prec) != NULL &&
  equalWeek((*prec), nouvWeek->year, nouvWeek->numWeek)) {

    printf("THIS WEEK ALREADY EXIST, NO NEED TO ADD IT\n");

    // On lib re celle en trop.
    freeWeek(nouvWeek);

  }

  // Sinon on l'ajoute
  else {
    nouvWeek->suiv = (*prec);
    *prec = nouvWeek;
  }
}
return prec;
}
```

7. supprWeek (similaire à supprAction)

```
void supprWeek(listWeek_t *listWeek, char year[4], char week[2]) {
week_t **pprec = findWeek(listWeek, year, week); // Comme dans findWeek

// Si la semaine correspond bien    celle voulue
if (pprec != NULL && *pprec != NULL && equalWeek(*pprec, year, week)) {
  // On la supprime
  week_t *tmp = *pprec; // pointeur de semaine temporaire
  (*pprec) = (*pprec)->suiv;
  freeWeek(tmp);
```

```
  }
}
```

8. suprActionInsideWeek

   Cette fonction, ainsi que la suivante, sont davantage propres aux semaines. En effet, si l'on veut modifier seulement une action dans notre liste de semaines (ce qui est le cas pour notre calendrier), il nous faut parcourir/modifier aussi bien les semaines que les actions. C'est dans ces fonctions que toutes les procédures qu'on a vu précédemment prennent leur sens. Grâce à ces dernières, nos deux fonctions principales pour le calendrier deviennent très simples à écrire.

   On réalise un premier parcours de la liste des semaines pour voir s'il existe une semaine avec l'année et le numéro de semaine donné en argument. Si c'est le cas, on parcourt la liste d'actions de cette semaine pour trouver l'action concordant avec nos arguments **day** et **hour**. Si cette action existe, alors il nous suffit de la supprimer. Si l'une de ces deux recherches ne conclue pas, cela signifie qu'il n'existe pas, dans la liste, l'action à supprimer. On peut donc arrêter là. Enfin, on vérifie au début de la procédure si nos argument sont corrects. Si ce n'est pas le cas, on peut éviter de chercher car on sait qu'il n'y aura rien à supprimer.

```
int supprActionInsideWeek(listWeek_t *listWeek, char year[4],
 char week[2],
             char day, char hour[2]) {
// On cherche la semaine de l'action    supprimer
int code = 1;
week_t **precWeek = findWeek(listWeek, year, week);

// Si elle existe
if (*precWeek != NULL && equalWeek(*precWeek, year, week)) {

  // On cherche l'action dans cette semaine
  action_t **curr =
  findAction((&(*findWeek(listWeek, year, week))->listAct), hour,
  day);

  // Si elle existe on la supprime
  if (curr != NULL && equalAction(*curr, day, hour)) {
    supprAction(curr, hour, day);
    if ((*precWeek)->listAct == NULL) {
  supprWeek(precWeek, year, week);
    }
  } else
    code = -1;
} else
  code = -2;

return code;
}
```

9. insertActionInsideWeek

   On va appeler dans un premier temps insereWeek, qui va nous retourner la semaine correspondant aux arguments fournis. Ensuite, on appelle, sur la liste d'actions de cette semaine, la fonction insertAction. Les cas ou une action / semaine existe déjà sur c'est créneaux sont gérés par la fonction insert.

```
1  void insertActionInsideWeek(listWeek_t *listWeek, char year[4],
    char numWeek[2],
2                  char day, char hour[2], char name[10]) {
3  // Si notre semaine et notre action sont coh rentes
4  if (checkNumWeek(numWeek) && checkYear(year) && checkDay(day) &&
5      checkHour(hour)) {
6    // On cherche/ins re si besoin la semaine
7    // Pas besoin de v rifier si week NULL, en effet on l'a d j
     v rifi  avec la
8    // condition au-dessus
9    week_t **week = insertWeek(listWeek, newWeek(year, numWeek));
10   // On ins re l'action
11   insertAction(&(*week)->listAct, newAction(day, hour, name));
12 }
13 }
```

10. prettyPrintListWeek (similaire à prettyPrintListAction)

```
1  void prettyPrintListWeek(listWeek_t listWeek) {
2  week_t *curr = listWeek; // Pointeur sur la semaine courante
3  int i = 0;               // Simple compteur
4
5  printf("
    ============================================================\n");
6  while (curr) {
7    printf("| %d | week %s | year %s |\n", i, curr->numWeek, curr->
    year);
8    if (curr->listAct) {
9      printf("Action :::\n");
10     prettyPrintListAction(curr->listAct);
11     printf("\n\n");
12   }
13   i++;
14   curr = curr->suiv;
15 }
16 printf("
    ============================================================\n\n")
    ;
17 }
```

3 Exécution

3.1 Makefile

Le makefile permet de compiler tous les fichiers nécessaires pour obtenir l'exécutable du menu en une seule commande. Il suffit pour cela d'appeler la commande make dans le répertoire contenant le fichier makefile.

Pour obtenir l'exécutable de l'ensemble du jeu de test, il suffit d'appeler de la même façon la commande make mais avec "test" comme argument.

Toutes les compilations se font avec l'option -g pour pouvoir utiliser valgrind.

```
  CC = gcc
CFLAGS = -Wall -Wextra
```

```
LFLAGS = -g
SOURCES = $(wildcard *.c)
EXEC = prog

all: $(EXEC)

$(EXEC) : menu.o action.o semaine.o
$(CC) $(CFLAGS) -o $@ $^ $(LFLAGS)

%.o: %.c
$(CC) $(CFLAGS) -o $@ -c $< $(LFLAGS)

test: test.o action.o semaine.o
$(CC) $(CFLAGS) -o $@ $^ $(LFLAGS)
clean:
rm -rf *.o
```

3.2 Jeux de tests

Comme indiqué précédemment, tous les jeux de tests se situent dans le fichier test.c. Il suffit ensuite de compiler et exécuter le programme afin de vérifier les différents cas. On peut toujours mettre certains tests en commentaires pour faciliter la lecture d'un cas spécifique.