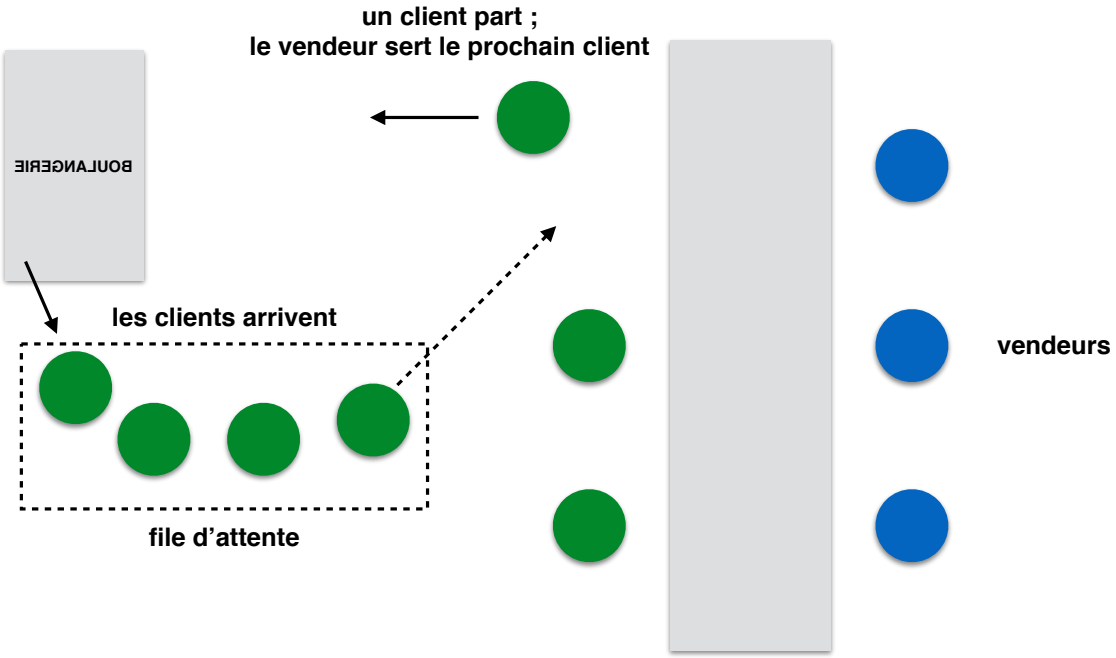
# TP 5 et 6 – Simulation discrète d'une boulangerie

**Objectif:** Le but de ce TP est d'implanter une simulation d'une file d'attente dans une boulangerie. Ce TP va vous occuper pendant deux séances. La première séance, vous disposez de quelques structures de données déjà implantées, une file d'attente et une file de priorité, pour implanter la simulation. La deuxième séance, vous allez réimplanter ces deux structures en utilisant des listes chaînées. Il n'y a qu'un seul rendu à déposer, après la deuxième semaine.

## 1 Introduction – un survol du système




Les clients (verts dans la figure) arrivent avec une certaine fréquence, un par minute, par exemple. S'il y a un vendeur (bleu dans la figure) libre, alors le client est immédiatement servi. Sinon, le client se met à la fin de la file d'attente. Après un certain temps un client a terminé et sort de la simulation (il sort de la boulangerie). Le vendeur qui est libéré inspecte la file d'attente. Si elle est vide, le vendeur reste libre jusqu'à l'arrivé d'un nouveau client. Sinon, le vendeur sert le client en tête de la file d'attente. À la fin de la journée, on affiche quelques statistiques : combien de clients ont été servis ? Quel était le *temps moyen de service* (le temps moyen qu'un client passe à l'intérieur de la boulangerie) ? Les paramètres de la simulation sont

(1) le temps entre l'arrivée des clients, (2) le temps de service d'un client et (3) le nombre de vendeurs.

Le type de simulation qu'on va implanter s'appelle une *simulation à événements discrets*. Cela veut dire que l'horloge du système n'avance pas seconde par seconde en attendant une action. À la place, l'heure saute chaque fois qu'un *événement* se passe. Un événement de notre système peut être soit l'arrivée soit le départ d'un client.

Quand un événement est créé, par exemple : "à l'heure 135 secondes après le début de la simulation, un client va arriver", alors cet événement est placé dans une *file de priorité d'événements* (*event queue* en anglais). L'heure donnée à un événement s'appelle l'heure d'événement (event time) Le cœur de la simulation est la boucle principale : tant que la file de priorité d'événements n'est pas vide, alors on extrait l'événement avec l'heure d'événement la plus proche et on le traite. Par exemple, s'il y a deux événements dans la file de priorité ayant respectivement pour 'heure d'événement 135 secondes et 87 secondes, alors la boucle va d'abord extraire celui avec l'heure d'événement de 87 secondes et le traiter.

Le traitement d'un événement commence en avançant l'heure globale jusqu'à l'heure de l'événement. Le traitement peut créer d'autres événements qui sont insérés dans la file de priorité d'événements. Par exemple, s'il y a un événement qui annonce l'arrivée d'un client à l'heure 135 et si à ce moment il y a un vendeur qui est libre, alors le traitement de l'événement crée un nouvel événement de type : "le vendeur va terminer le service du client à l'heure $135 + X$" où $X$ est le temps de service du client.

La simulation d'une boulangerie comporte deux types d'événements : un événement d'arrivée (d'un client) et un événement de départ (d'un client). Ci-dessous, le traitement de ces deux types d'événements est décrit. Dans les exercices, vous trouvez plus de détails pour l'implantation.

**Traitement d'un événement d'arrivée (du client $C$) :**

1. S'il y a un vendeur $V$ libre, alors
   — le vendeur $V$ commence à servir le client $C$ ;
   — un événement de départ pour le client $C$ à l'heure + $X$ secondes est placé dans la file de priorité d'événements.
2. Sinon,
   — le client $C$ se place dans la file d'attente.
3. Un nouvel événement d'arrivée à l'heure + $Y$ secondes est placé dans la file de priorité d'événements.

**Traitement d'un événement de départ (du client $C$) :**

1. Chercher le vendeur $V$ qui a fini de servir son client $C$.
2. Supprimer le client $C$ du système.
3. Si la file d'attente est vide, alors
   — le vendeur $V$ devient libre.

4. Sinon,
   — le client $C_2$ à la tête de la file d'attente est extrait et le vendeur $V$ commence à le servir ;
   — un événement de départ pour le client $C_2$ à l'heure + $X$ secondes est placé dans la file de priorité d'événements.

# 2 File d'attente à la boulangerie

**▸ Exercice 1 : Votre premier client**
Commencer par récupérer l'archive `bakery.zip` et extraire les fichiers dans votre répertoire de travail. L'archive contient les fichiers suivants :

— `customer.h/.c` : la structure qui modélise un client ;
— `event.h/.c` : la structure qui modélise un événement ;
— `queue.h/.c` : la structure de données qui implante la file d'attente des clients ;
— `prioqueue.h/.c` : la structure de données qui implante la file de priorité des événements ;
— `bakery.c` : le fichier principal qui contient la fonction `main` ;
— `Makefile` : le makefile ;
— `reponse.txt` : ici vous pouvez mettre vos réponses aux questions posées dans l'énoncé.

Familiarisez-vous avec le contenu de ces fichiers. Pendant la première séance vous allez uniquement rédiger le fichier `bakery.c`. Si vous avez des questions, n'hésitez pas à les poser à votre chargé de TP.

Dans le fichier `bakery.c`, la constante `N_VENDORS` indique le nombre de vendeurs dans la simulation. Trois variables globales sont déclarées qui représentent les composantes principales de la simulation :

— `prioqueue *event_queue` – un pointeur sur la file de priorité d'événements ;
— `queue *customer_queue` – un pointeur sur la file d'attente des clients ;
— `customer *vendor[N_VENDORS]` – un tableau qui pour chaque vendeur contient un pointeur sur le client que le vendeur est en train de servir, ou `NULL` si le vendeur est libre.

1. Au début de votre fonction `main`, ajoutez des instructions pour initialiser les trois variables globales décrites ci-dessus avec une file de priorité vide, une file d'attente vide et des pointeurs `NULL` pour chaque vendeur.
2. Ajoutez les instructions nécessaires pour créer un client (avec une heure d'arrivée de 42 secondes, par exemple) et ajoutez-le à la file d'attente. Ensuite, extraire le client à la tête de la file d'attente et afficher son heure d'arrivée. Vérifier que celui est bien 42.

   Quand ca fonctionne, vous pouvez supprimer ces lignes.

**▸ Exercice 2 : La boucle d'événements**
Vous allez maintenant ajouter la boucle d'événements.

1. La boucle d'événement se comporte comme suit :
   — Tant que la file de priorité d'événements n'est pas vide, alors extrayez le prochain événement de la file et traitez-le : appelez la fonction `process_arrival` ou la fonction `process_departure` d'après le type de l'événement. (Pour l'instant, ces deux fonctions ne font rien.)
   — À la fin de la boucle, libérez la mémoire associée à l'événement extrait.

   Écrivez la boucle d'événements dans votre fonction `main` ou dans une fonction séparée qui sera appelée par la fonction `main`.
2. Ajoutez une instruction à la boucle d'événement qui affiche l'heure de l'événement juste avant de le traiter.
3. Dans votre fonction `main`, créez un client et un événement d'arrivée et ajoutez le dernier à la file d'événements avant d'entrer dans la boucle d'événements. Lancez votre programme pour vérifier que la boucle d'événements traite correctement cet événement. Après avoir affiché l'heure de l'événement et appelé la fonction `process_arrival`, votre boucle d'événements devrait terminer (comme la file de priorité d'événements est devenue vide).

### ▸ Exercice 3 : L'arrivée d'un client

Dans cet exercice, vous allez implanter le traitement d'un événement d'arrivée. À l'occurrence d'un tel événement, le programme extrait le client de l'événement et soit l'affecte à un vendeur libre, soit l'insère à la fin de la file d'attente.

1. Ajouter une variable globale `int current_time`, l'heure courant de la simulation, qui représente le nombre de secondes passé depuis le début de la simulation. Au début du programme, initialiser cette variable à 0. Dans la boucle d'événements, lui affecter l'heure de l'événement extrait de la file de priorité.
2. Écrire une fonction `void add_customer(customer *c)`. Cette fonction parcourt les vendeurs en en cherchant un qui est libre. Le premier vendeur libre trouvé va servir le client `c`. Cela se fait en affectant `vendor[vid]=c` où `vid` est le numéro du vendeur. S'il n'y a pas de vendeur libre, alors le client sera ajouté à la file d'attente. Un client est donc toujours dans *exactement un seul endroit du système* : dans la file de priorité en attendant de venir, dans la file d'attente ou en train d'être servi par un vendeur.
3. Réécrire la fonction `process_arrival` pour effectuer le suivant :
   1) Appeler la fonction `add_customer` ;
   2) Soit `time` = l'heure courant de la simulation + 60 ;
   3) Créer un nouveau client avec l'heure d'arrivée = `time` ;
   4) Créer un nouvel événement d'arrivée avec l'heure d'événement = `time` ;
   5) Insérer le nouvel événement dans la file de priorité d'événements.
4. Créez une fonction pour visualiser l'état courant du système et appelez-le à la fin de la boucle d'événement. Exemple d'affichage :

```
 60 | X__ |
120 | XX_ |
180 | XXX |
240 | XXX | X
300 | XXX | XX
360 | XXX | XXX
```

La première colonne montre l'heure courant de la simulation. La deuxième colonne montre les vendeurs, `X` signifie que le vendeur est en train de servir un client, `_` signifie que le vendeur est libre. La troisième colonne montre la file d'attente. Chaque `X` représente un client en attente.

Dans la visualisation ci-dessus, on voit que :
— Le premier client est arrivé à l'heure 60. Il est immédiatement servi par un vendeur.
— Les deuxième et troisième clients arrivent respectivement à l'heure 120 et 180 et sont également immédiatement servis.
— Le quatrième client qui arrive à l'heure 240 est placé dans la file d'attente.

Comme les clients ne sortent pour l'instant jamais du système, la file d'attente va continuer à augmenter indéfiniment (ou jusqu'à ce que la file soit pleine).

5. Définir une constante `CLOSING_TIME` qui représente l'heure de fermeture de la boulangerie. Dans la boucle d'événement, ajouter une condition d'arrêt : si l'heure courant dépasse `CLOSING_TIME`, alors la boucle termine.

### ▸ Exercice 4 : Le départ d'un client

Dans cet exercice, vous allez implanter le traitement d'un événement de départ. À l'occurrence d'un tel événement, le programme supprime le client concerné du système. De plus, le vendeur qui est désormais devenu libre va chercher un autre client dans la file d'attente (s'il y en a un) et commencer à le servir.

1. Écrire la fonction `void remove_customer(customer *c)`. Cette fonction cherche d'abord l'id (`vid`) du vendeur qui s'occupe du client `c` et le libère. La fonction libère également la mémoire associée au client `c`. Si la file d'attente n'est pas vide, alors
   — la fonction extrait le premier client `c2` de la file et associe `vendor[vid]=c2`. Ensuite, elle crée un nouvel événement de départ avec l'heure d'événement = l'heure courant + 150. Cet événement représente l'heure auquel le vendeur terminera de servir le client `c2`. Finalement, la fonction insère l'événement dans la file de priorité d'événements.
2. Écrire la fonction `process_departure` pour qu'elle appelle la fonction `remove_customer` avec le client associé à l'événement.
3. Dans la fonction `add_customer`, après avoir affecté un client à un vendeur, créer un nouvel événement de départ de la même manière qu'en (1).
4. Votre programme devrait maintenant produire un affichage qui ressemble au suivant.

```
 60 | X__ |
120 | XX_ |
180 | XXX |
```

```
210 | _XX |
240 | XXX |
270 | X_X |
300 | XXX |
330 | XX_ |
360 | XXX |
```

Remarquer qu'aux heures 210, 270 et 330, des clients sont *supprimés* de la simulation.

### ▸ Exercice 5 : Arrivée et départ aléatoire

Les clients arrivent pour l'instant avec une ponctualité surhumaine et sont tous servis dans un temps précis de 150 secondes. Vous allez maintenant remplacer cette exactitude par un processus aléatoire.

1. Ajouter la fonction suivante au fichier `bakery.c`. Assurez-vous que les librairies `<math.h>` (pour la fonction `log`) et `<stdlib.h>` (pour la fonction `rand`) sont incluses.

   /ens/IR/IR1/2020-2021/Algo/src/tp05/normal_delay.c

   ```
   double normal_delay(double mean) {
       return -mean*log(1-((double)rand()/RAND_MAX));
   }
   ```

2. L'arrivée des clients sera modélisée par un *processus de Poisson*[1]. Cela veut dire que les clients arrivent indépendamment aux moments aléatoires avec une intensité moyenne fixe, par exemple un client toutes les 60 secondes (en moyenne).
   Pour l'implanter, définir une constante `#define ARRIVAL_RATE (1.0/60)` (un client par minute en moyenne) et remplacer la constante 60 utilisée dans la fonction `process_arrival` par l'appel : `normal_delay(1.0/ARRIVAL_RATE)`. Le temps moyen entre l'arrivée des clients reste à 60 secondes. Vous allez voir ce processus et d'autres dans le cours "Probabilités, statistiques" au troisième semestre.
3. Le temps pour servir un client est pour l'instant exactement 150 secondes. On va le modéliser par une variable aléatoire suivant une *loi normale* de moyenne `MEAN_SERVICE_TIME`[2].
   Définir une constante `#define MEAN_SERVICE_TIME 150` et remplacer 150 par l'appel : `normal_delay(MEAN_SERVICE_TIME)`.
4. Votre programme devrait maintenant produire un affichage qui ressemble au suivant.

   ```
    44 | X__ |
    62 | XX_ |
    75 | XXX |
   118 | XXX | X
   132 | XXX |
   ```

1. https://fr.wikipedia.org/wiki/Processus_de_Poisson
2. https://fr.wikipedia.org/wiki/Loi_normale

```
163 | XXX | X
169 | XXX | XX
176 | XXX | X
243 | XXX |
270 | XXX | X
286 | XXX | XX
```

5. Mettez la constante `CLOSING_TIME` à quelque chose un peu réaliste, modifiez les valeurs des paramètres `N_VENDORS`, `ARRIVAL_RATE` et `MEAN_SERVICE_TIME` et observez les effets.

# 3 Listes chaînées

▸ **Exercice 6 : `queue.c`**

Dans cet exercice, vous allez implanter l'interface dans le fichier `queue.h` en utilisant une liste simplement chaînée. Pendant l'implantation, vous pouvez faire des tests dans une fonction `main` que vous placez dans le fichier `queue.c` et compilez avec la commande `gcc queue.c`

1. Tapez `cp queue.c queue_bak.c` pour faire une copie de la file d'attente au cas où vous en ayez besoin.
2. Télécharger le fichier `queue.c` :

/ens/IR/IR1/2020-2021/Algo/src/tp05/queue.c

```c
#include <stdlib.h>
#include "queue.h"

typedef struct _link {
    customer*       c;
    struct _link*   next;
} link;

struct _queue {
    link*   first;
    link*   last;
    int     size;
};

queue *create_q() {
    queue *q = (queue*)malloc(sizeof(queue));
    q->first = NULL;
    q->last = NULL;
    q->size = 0;
    return q;
```

```
}

void free_q(queue *q) {
    free(q);
}

int size_q(queue *q) {
    return q->size;
}
```

3. Écrire la fonction `void enqueue_q(queue *q, customer *c)` dans le fichier `queue.c`. La fonction alloue la mémoire d'une cellule (type `link`) et y ajoute le client `c`. Ensuite, la fonction insère la nouvelle cellule à la fin de la file d'attente.
4. Écrire la fonction `void display_q(queue *q)` qui parcourt la file d'attente et affiche le contenu sur le terminal. Cette fonction peut être utile afin de déboguer votre implantation. Vous pouvez également ajouter la signature dans le fichier `queue.h` si vous voulez utiliser cette fonction depuis un autre fichier.
5. Écrire la fonction `customer *dequeue_q(queue *q)` dans le fichier `queue.c`. La fonction extrait la cellule en tête de la liste, libère la mémoire associée et renvoie le client qui était stocké dans la cellule.
6. **Indiquer dans le fichier** `reponse.txt` les complexités souhaitées des fonctions `enqueue_q` et `dequeue_q` et comment vous pouvez vérifier (grossièrement) qu'elles sont les bonnes.
7. Vérifier que la simulation du fichier `bakery.c` fonctionne en tapant `make` et `./bakery`

▸ **Exercice 7 : `prioqueue.c`**

Dans cet exercice, vous allez implanter l'interface dans le fichier `prioqueue.h` en utilisant une liste simplement chaînée triée sur les heures des événements.

1. Tapez `cp prioqueue.c prioqueue_bak.c` pour faire une copie de la file d'attente au cas où vous en avez besoin.
2. Vider le fichier `prioqueue.c` et y ajouter les définitions :

```
typedef struct _link {
    event*          e;
    struct _link*   next;
} link;

struct _prioqueue {
    link*   first;
    int     size;
```

```
};
```

3. Écrire les fonctions suivantes :

```
prioqueue *create_pq()
void free_pq(prioqueue *q)
int size_pq(prioqueue *q)
```

4. Écrire la fonction `void insert_pq(prioqueue *q, event *e)` qui alloue la mémoire d'une cellule (type `link`) et y ajoute l'événement `e`. Ensuite, la fonction insère la nouvelle cellule à sa place dans la liste afin de garder la liste triée sur les heures des événements.
5. Écrire la fonction `void display_pq(prioqueue *pq)` qui parcourt la file de priorité et affiche le contenu sur le terminal.
6. Écrire la fonction `event *remove_min_pq(prioqueue *q)` qui extrait la cellule en tête de la liste (qui a la plus petite heure d'événement), libère la mémoire associée et renvoie l'événement qui était stocké dans la cellule.
7. **Indiquer dans le fichier** `reponse.txt` les complexités souhaitées des fonctions `insert_pq` et `remove_min_pq` et comment vous pouvez vérifier (grossièrement) qu'elles sont les bonnes.
8. Vérifier que la simulation du fichier `bakery.c` fonctionne en tapant `make` et `./bakery`

### ▸ Exercice 8 : Statistiques et nettoyage

1. Ajouter un compteur pour le nombre de clients servis et un compteur pour le temps total que les clients ont passé dans le système. Le temps total est égal à la somme pour chaque client de son heure de départ moins son heure d'arrivée. À la fin de la simulation, afficher le nombre de clients servis ainsi que *le temps moyen de service*, le temps total que les clients ont passé dans le système divisé par le nombre de clients servis. Le temps moyen de service est le temps moyen pour un client entre son arrivée et son départ.
2. Essayer des valeurs différentes pour les paramètres `N_VENDORS`, `ARRIVAL_RATE` et `MEAN_SERVICE_TIME` et observer les effets sur le temps moyen de service.
3. Votre programme termine probablement sans libérer toute la mémoire allouée. Les files peuvent contenir des clients et des événements qui ne sont pas utilisés après la fermeture de la boulangerie et les vendeurs peuvent aussi rester avec des clients. Les files elles-mêmes doivent également être libérées à la fin de l'exécution. Réfléchir, pour chaque client et chaque événement, où ils sont créés et où ils doivent être libérés.

### ▸ Exercice 9 : Pour aller plus loin

Trouvez ci-dessous quelques suggestions d'améliorations/variations de votre programme.

1. La boulangerie ferme pour l'instant assez brusquement. Si vous voulez, vous pouvez changer ce comportement. Par exemple, les vendeurs peuvent continuer à servir les clients qui sont à l'intérieur de la boulangerie à l'heure de la fermeture mais pas autoriser de nouveaux clients à se mettre dans la file d'attente.

2. Faire en sorte qu'un client ne reste qu'un certain temps dans la file d'attente avant de s'ennuyer et la quitter. Pour cela vous pouvez introduire un nouveau type d'événement `EVENT_LEAVE_QUEUE` et insérer un tel événement $E$ dans la file de priorité à l'heure de la création de l'événement d'arrivée. Si le client n'est pas servi avant l'heure de $E$, alors il sort immédiatement du système.
3. Votre programme implante un simple système qui s'appelle une file M/M/$c$. Vous pouvez en lire plus sur les liens suivants :

   https://en.wikipedia.org/wiki/M/M/c_queue
   https://fr.wikipedia.org/wiki/Th%C3%A9orie_des_files_d'attente

   Sur le lien https://en.wikipedia.org/wiki/M/M/c_queue#Response_time vous trouvez une formule pour calculer la valeur théorique du temps moyen de service pour un tel système. Implanter cette fonction en `C` et comparer avec votre simulation.