INF3105 – Files (*Queues*)

Jaël Champagne Gareau

Université du Québec à Montréal (UQAM)

Été 2024

http://cria2.uqam.ca/INF3105/

UQÀM

Sommaire

Les files

- Type de données abstrait simple, similaire à la pile.
- Analogie : file d'attente dans une cafétéria.
- Modèle *FIFO : first-in-first-out* (premier arrivé, premier servi).

Exemples d'applications

- File d'impression.
- Buffers (mémoire tampon) dans les protocoles réseaux.
- Traitement des événements dans un sytème GUI.
- Gestion et ordonnancement de tâches.
- Etc.

Interface **abstraite** d'une file standard

enfiler(*e*)	enqueue(*e*)	Ajoute *e* à la queue de la file.
defiler()	dequeue()	Enlève l'élément à la tête de la file.
tete()	front()	Retourne l'élément à la tête.
taille()	size()	Retourne le nombre d'éléments dans la file.
vide()	empty()	Retourne vrai si la file est vide, sinon faux.

Interface standard C++ d'une file

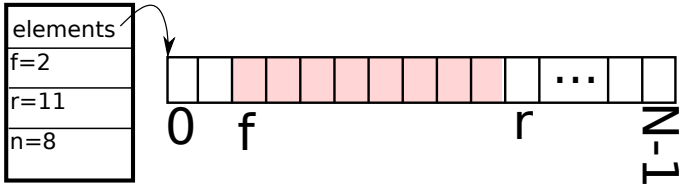
```
1  template <class T> class File {
2    public:
3      File();
4      ~File();
5      int  taille() const; // optionnel
6      bool vide() const;
7      const T& tete() const; // retourne sans enlever l'élément en tête
8      void enfiler(const T& e);
9      // Au choix, l'une des fonctions suivantes :
10     T defiler(); // retourne et enlève l'élément à la tête
11     void defiler(); // enlève l'élément à la tête
12     void defiler(T& e); // copie l'élément à la tête dans sortie et l'enlève
13 };
```

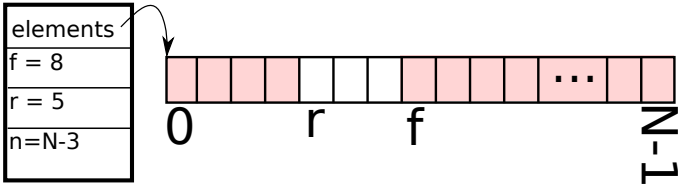
Représentation C++ d'une file

Fichier entête partiel file.h

```
1  template <class T>
2  class FileCirculaire {
3  public:
4      FileCirculaire(int capacite = 100);
5      ~FileCirculaire();
6      int  taille() const;
7      bool vide() const;
8      const T& tete() const;
9      void enfiler(const T& e);
10     T defiler();
11 private:
12     T* elements;
13     int capacite; // capacité de éléments
14     int f; // index sur la tête (front)
15     int r; // index sur la cellule suivant la queue (rear)
16     int n; // nombre d'éléments dans la file
17 };
```

Représentation d'une file circulaire





Constructeur

Version 1

```
1  template <class T>
2  FileCirculaire<T>::FileCirculaire(int _capacite) {
3      capacite = _capacite;
4      elements = new T[capacite];
5      f = r = n = 0;
6  }
```

Version 2

```
1  template <class T>
2  FileCirculaire<T>::FileCirculaire(int _capacite)
3          : elements(new T[initCap]), capacite(_capacite), f(0), r(0), n(0) {}
4
```

Destructeur

```
1  template <class T>
2  FileCirculaire<T>::~FileCirculaire() {
3      delete[] elements;
4      elements = nullptr; // optionnel
5  }
```

Fonctions (1)

```
1  template <class T>
2  int FileCirculaire<T>::taille() const {
3      return n;
4  }
5
6  template <class T>
7  bool FileCirculaire<T>::vide() const {
8      return n == 0;
9  }
10
11 template <class T>
12 const T& FileCirculaire<T>::tete() const {
13     assert(!vide());
14     return elements[f];
15 }
```

Fonctions (2)

```
1  template <class T>void FileCirculaire<T>::enfiler(const T& element) {
2      assert(taille()<capacite);
3      // On peut aussi réallouer le tableau dynamiquement
4      elements[r] = element;
5      r = (r+1) % capacite;
6      n++;
7  }
8
9  template <class T> void FileCirculaire<T>::defiler() {
10     assert(!vide());
11     f = (f+1) % capacite;
12     n−−;
13 }
```

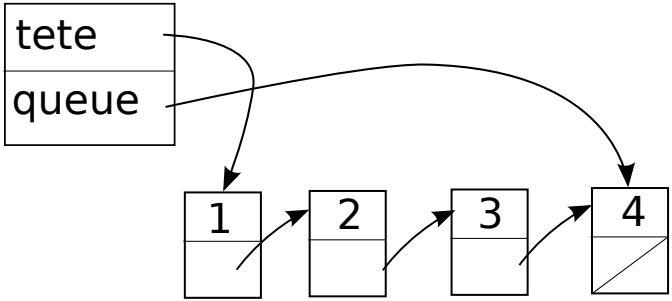
Analyse des opérations

Opération	Complexité
enfiler(e)	O(1)
defiler()	O(1)
tete()	O(1)
taille()	O(1)
vide()	O(1)

Remarques similaire à la pile

- Si on permet la réallocation du tableau, enfiler n'est pas toujours O(1). Voir pile.
- Lorsque le nombre d'éléments est difficile à estimer à l'avance, la perte d'espace (au pire n/2) est difficile à éviter.

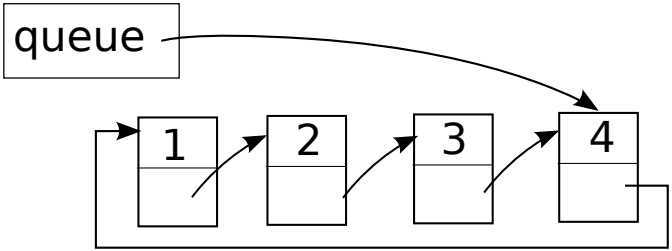
Représentation naïve (intuitive) d'une file avec des cellules



Représentation naïve (intuitive) d'une file en C++

```
1  template <class T>
2  class FileNaive {
3    public:
4      File();
5      ~File();
6      bool vide() const;
7      const T& tete() const;
8      void enfiler(const T&);
9      void defiler();
10   private:
11     struct Cellule {
12       Cellule(const T& c, Cellule* s) : contenu(c), suivante(s) {}
13       T contenu;
14       Cellule* suivante;
15     };
16     Cellule* queue;
17     Cellule* tete
18 };
```

Représentation d'une file avec des cellules



Représentation C++ d'une file

```
1  template <class T>
2  class File {
3    public:
4      File();
5      ~File();
6      bool vide() const;
7      void vider() const;
8      const T& tete() const;
9      void enfiler(const T&);
10     void defiler();
11   private:
12     struct Cellule {
13       Cellule(const T& c, Cellule* s=nullptr) : contenu(c), suivante(s) // ou {contenu=c;}
14       T contenu;
15       Cellule* suivante;
16     };
17     Cellule* queue;
18 };
```

Constructeur et Destructeur

Constructeur

```
1  template <class T>
2  File<T>::File() : queue(nullptr) {}
```

Destructeur - Version 1

```
1  template <class T>
2  File<T>::~File() {
3      while(!vide()) defiler();
4  }
```

Destructeur - Version 2

```
1  template <class T>
2  File<T>::~File() {
3      vider();
4  }
```

Fonctions vide() et tete()

```
1  template <class T>
2  bool File<T>::vide() const {
3      return queue == nullptr;
4  }
5
6  template <class T>
7  const T& File<T>::tete() const {
8      assert(queue != nullptr);
9      return queue->suivante->contenu;
10 }
```

Enfiler

```
1  template <class T>
2  void File<T>::enfiler(const T& element) {
3      if(queue == nullptr) {
4          queue = new Cellule(element);
5          queue–>suivante = queue;
6      } else
7          queue = queue–>suivante = new Cellule(e, queue–>suivante);
8  }
```

Defiler

```
1  template <class T>
2  void File<T>::defiler() {
3      Cellule* c = queue->suivante;
4      if(queue == c)
5          queue = nullptr;
6      else
7          queue->suivante = c->suivante;
8      delete c;
9  }
```

Vider

Version 1

```
1  template <class T>
2  void File<T>::vider() {
3      while(!vide()) defiler();
4  }
```

Version 2

```
1  template <class T>
2  void File<T>::vider() {
3      Cellule* fin = queue;
4      while(queue != nullptr) {
5          Cellule* suivante = queue->suivante;
6          if(suivante == fin) suivante = nullptr;
7          delete queue;
8          queue = suivante;
9      }
10 }
```

Analyse des opérations

Opération	Complexité
enfiler(*e*)	O(1)
defiler()	O(1)
tete()	O(1)
taille()	O(1)
vide()	O(1)

Remarques

- Hypothèse requise : allocation et désallocation de mémoire (opérateurs new et delete) en temps constant, i.e. O(1). Cela dépend de l'allocateur de mémoire (compilateur + système d'exploitation).
- Espace mémoire : on a besoin d'un pointeur par cellule. Négligeable quand les objets sont gros.

Opérateur ==

```
1  template <class T>
2  bool File<T>::operator==(const File<T>& autre) const {
3
4
5
6
7
8
9
10
11
12
13 };
```

Opérateur =

```
1  template <class T>
2  File<T>& File<T>::operator=(const File<T>& autre) {
3
4
5
6
7
8
9
10
11
12     return *this;
13 };
```