INF3105 – Files (queue / deque)

Éric Beaudry

Université du Québec à Montréal (UQAM)

2017A

Sommaire

Les files

- Structure de données 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.
- 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

```
template <class T> class File {
  public:
    File();
    ~File();
    int  taille() const; // optionnel
    bool vide() const;
    void vider() const;
    const T& tete() const; // retourne sans enlever l'element en tete
    void enfiler(const T& e);
    // Au choix, l'une des fonctions suivantes :
    T defiler(); // retourne et enleve l'element a la tete
    void defiler(); // enleve l'element a la tete
    void defiler(T& e); // copie l'element a la tete dans sortie et l'enleve
};
```

Représentation C++ d'une file

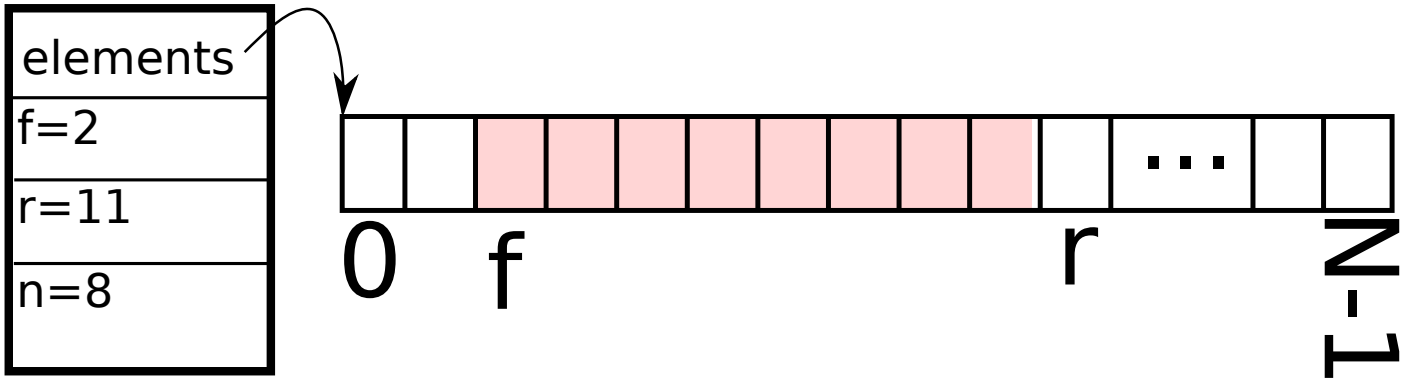
Fichier entête partiel file.h

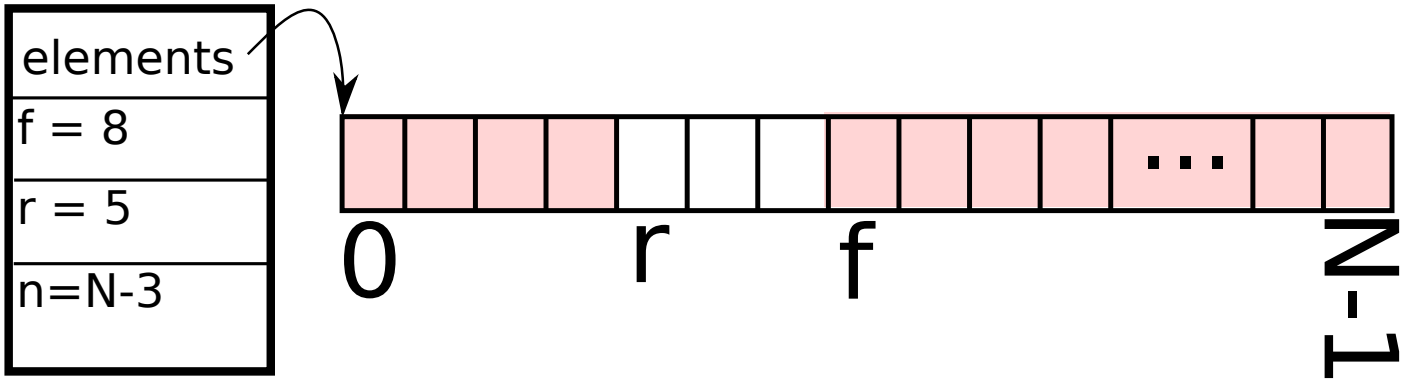
```
template <class T>
class FileCirculaire{
public:
 FileCirculaire(int capacite=100};
 ~FileCirculaire();
  int  taille() const;
  bool vide() const;

  const T& tete() const;
  void enfiler(const T& e);
  T defiler();

private:
  T* elements;
  int capacite; // capacite de elements
  int f; // index sur la tete (front)
  int r; // index sur la cellule suivant la queue (rear)
  int n; // nombre d'elements dans la file
};
```

Représentation d'une file ciculaire





Constructeur

Version 1

```
template <class T>
FileCirculaire<T>::FileCirculaire(int _capacity) {
  capacite = _capacity;
  elements = new T[capacite];
  f = r = 0;
  n = 0;
}
```

Version 2

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

Destructeur

```
template <class T>
FileCirculaire<T>::~FileCirculaire()
{
  delete[] elements;
  //elements = NULL; // optionnel
}
```

Fonctions (1)

```
template <class T>
int FileCirculaire<T>::taille() const
{
  return n;
}

template <class T>
bool FileCirculaire<T>::vide() const
{
  return n == 0;
}

template <class T>
const T& FileCirculaire<T>::tete() const
{
  assert(!vide());
  return elements[f];
}
```

Fonctions (2)

```
template <class T>void FileCirculaire<T>::enfiler(const T& element)
{
  assert(taille()<capacite);
  // On peut aussi reallouer le tableau dynamiquement
  elements[r] = element;
  r = (r+1) % capacite;
  n++;
}
template <class T> void FileCirculaire<T>::defiler()
{
  assert(!vide());
  f = (f+1) % capacite;
  n--;
}
```

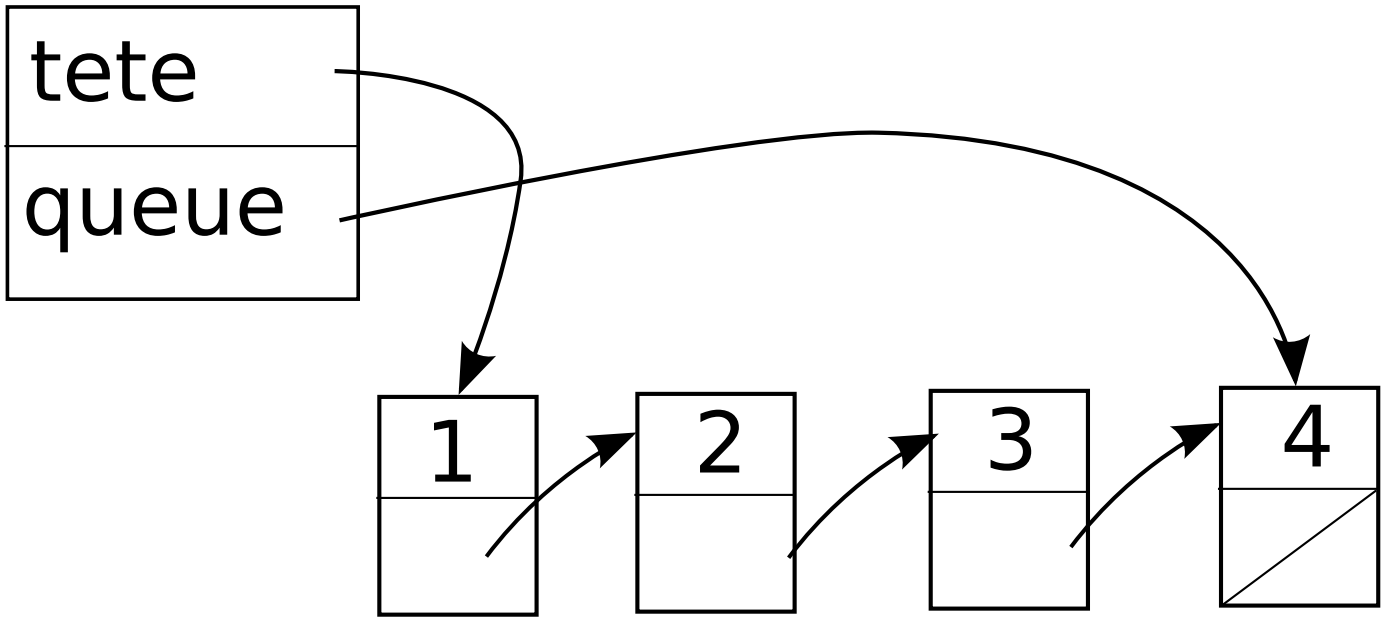
Analyse des opérations

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, voir 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 liste avec des cellules

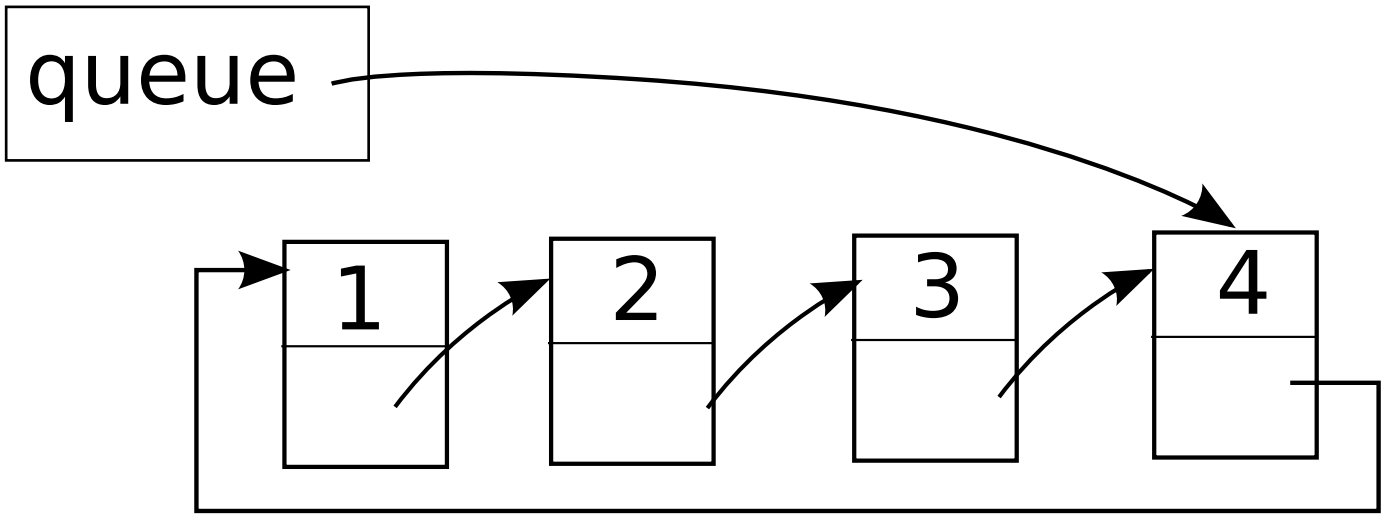


Représentation naïve (intuitive) d'une file en C++

```
template <class T> class FileNaive{
public:
  File();
  ~File();

  bool vide() const;
  const T& tete() const;
  void enfiler(const T&);
  void defiler();
private:
  class Cellule{
    public:
      Cellule(const T& c, Cellule* s) : contenu(c), suivante(s)
      T contenu;
      Cellule* suivante;
  };
  Cellule* queue;
  Cellule* tete
};
```

Représentation d'une file liste avec des cellules



Représentation C++ d'une file

```
template <class T> class File{
public:
  File();
  ~File();
  bool vide() const;
  void vider() const;
  const T& tete() const;
  void enfiler(const T&);
  void defiler();
private:
  class Cellule{
    public:
      Cellule(const T& c, Cellule* s=NULL) : contenu(d), suivante(s) // ou {contenu=c;}
      T contenu;
      Cellule* suivante;
  };
  Cellule* queue;
};
```

Constructeur

Version 1

```
template <class T>
File<T>::File()
{
  queue = NULL;
}
```

Version 2

```
template <class T>
File<T>::File()
 : queue(NULL)
{
}
```

Destructeur

Version 1

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

Version 2

```
template <class T>
File<T>::~File()
{
    vider();
}
```

Fonctions vide() et tete()

```
template <class T>
bool File<T>::vide() const
{
  return queue==NULL;
}

template <class T>
const T& File<T>::tete() const
{
  assert(queue!=NULL);
  return queue->suivante->contenu;
}
```

Enfiler

```
template <class T>
void File<T>::enfiler(const T& element)
{
  if(queue==NULL){
    queue = new Cellule(element);
    queue->suivante = queue;
  }else
    queue = queue->suivante = new Cellule(e, queue->suivante);
}
```

Defiler

```
template <class T>
void File<T>::defiler()
{
  Cellule* c = queue->suivante;
  //T e = c->contenu;
  if(queue==c)
    queue = NULL;
  else
    queue-> suivante = c->suivante;
  delete c;
  // return e;
}
```

Vider

Version 1

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

Version 2

```
template <class T>
void File<T>::vider() {
    Cellule* fin = queue;
    while(queue!=NULL){
        Cellule* suivante = queue->suivante;
        if(suivante==fin) suivante = NULL;
        delete queue;
        queue = suivante;
    }
}
```

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 ==

```
template <class T>
bool File<T>::operator==(const File<T>& autre) const
{




};
```

Opérateur =

```
template <class T>
File<T>& File<T>::operator=(const File<T>& autre)
{




  return *this;
};
```