INF3105 – Piles (*stacks*)

Éric Beaudry

Université du Québec à Montréal (UQAM)

2017A

Sommaire

Les piles

- Structure de données la plus simple.
- Analogie : piles d'assiettes dans une cafétéria.
- Modèle *LIFO : last-in-first-out* (dernier arrivé, premier servi).

Exemples d'applications

- Programme récursif converti en programme non récursif.
- Évaluation d'opérations arithmétiques.
- Boutons précédent et suivant dans les navigateurs Internet.
- Commandes pushd et popd dans le Shell Bash (Linux et Unix).
- Interpréteurs de langage (pile d'appel de fonction (contexte)).
- Langage *Postscript* dans les imprimantes.
- Etc.

Interface **abstraite** d'une pile

empiler(*e*)	push(*e*)	Place *e* au sommet de la pile.
depiler()	pop()	Enlève l'élément au sommet de la pile.
sommet()	top()	Retourne le sommet de la pile.
taille()	size()	Retourne le nombre d'éléments dans la pile.
vide()	empty()	Retourne vrai si la pile est vide, sinon faux.

Interface **abstraite** en C++ d'une pile

```
template <class T> class Pile {
  public:
    Pile();
    ~Pile();
    int  taille() const; // fonction optionnelle
    bool vide() const;

    const T& sommet() const;
    void empiler(const T& e);
    // Au choix, l'une des fonctions suivantes :
    T depiler(); // retourne l'objet qui etait au sommet de la pile
    void depiler(); // l'objet depile n'est pas retourne
    void depiler(T& e); // depile l'element dans l'objet e en reference
};
```

Représentation C++ d'une pile (version 1)

Fichier entête partiel pile.h

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

    const T& sommet() const;
    void empiler(const T& e);
    T depiler();

  private:
    T* data;
    int capacite;
    int s; // index sur le sommet
};
```

Représentation C++ d'une pile (version 2)

Fichier entête partiel pile.h

```
#include "tableau.h"
template <class T>
class PileTableau{
  public:
    PileTableau(); // optionnel
    ~PileTableau(); // optionnel
    int  taille() const;
    bool vide() const;

    const T& sommet() const;
    void empiler(const T& e);
    T depiler();

  private:
    Tableau<T> elements;
};
```

Constructeur

```
template <class T>
PileTableau<T>::PileTableau()
{
}
```

Destructeur

```
template <class T>
PileTableau<T>::~PileTableau()
{
}
```

Fonctions (1)

```
template <class T>
int PileTableau<T>::taille() const
{
    return elements.taille();
}

template <class T>
bool PileTableau<T>::vide() const
{
    return elements.vide();
}

template <class T>
const T& PileTableau<T>::sommet() const
{
    return elements[elements.taille()-1];
}
```

Fonctions (2)

```
template <class T>
void PileTableau<T>::empiler(const T& element)
{
    elements.ajouter(element);
}

template <class T>
T PileTableau<T>::depiler()
{
    T result = sommet();
    elements.enlever(elements.taille()−1);
    return result;
}
```

Analyse des opérations (sans réallocation)

Opération	**Complexité**
empiler(e)	O(1)
depiler()	O(1)
sommet()	O(1)
taille()	O(1)
vide()	O(1)
vider()	O(1) ou O(n)

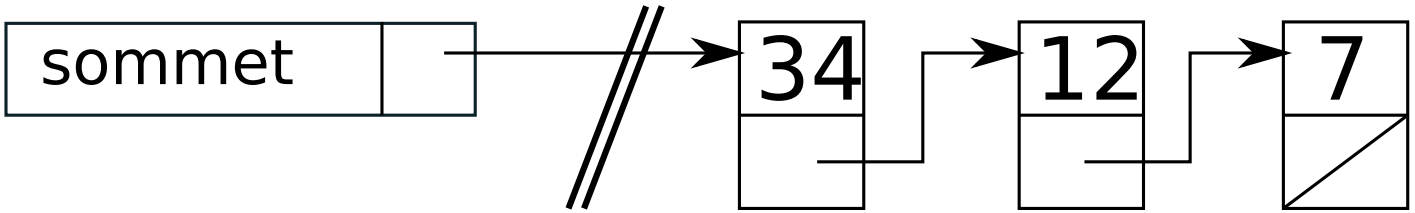
Analyse des opérations (avec réallocation)

Opération	Amortie	Pire cas
empiler(e)	O(1)	**O(n)**
depiler()	O(1)	O(1)
sommet()	O(1)	O(1)
taille()	O(1)	O(1)
vide()	O(1)	O(1)
vider()	O(1)	O(1) ou O(n)

Remarques

- Complexité héritée de `Tableau`.

Représentation d'une pile avec une chaîne de cellules



Représentation C++ d'une pile

Fichier entête

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

  bool vide() const;
  const T& sommet() const;
  void empiler(const T&);
  void depiler();

private:
  class Cellule{
    public:
      Cellule(const T& c, Cellule* s) : contenu(c), suivante(s)
      T contenu;
      Cellule* suivante;
  };
  Cellule* sommet;
};
```

Constructeur

Version 1

```
template <class T>
Pile<T>::Pile() {
  sommet = NULL;
}
```

Version 2

```
template <class T>
Pile<T>::Pile()
 : sommet(NULL) {
}
```

Destructeur

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

Fonctions

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

template <class T>
const T& Pile<T>::sommet() const
{
  assert(sommet!=NULL);
  return sommet->contenu;
}
```

Empiler

```
template <class T>
void Pile<T>::empiler(const T& element)
{
  sommet = new Cellule(element, sommet);
  // optionnel : test l'allocation de memoire
  // assert(sommet != NULL);
}
```

Depiler

```
template <class T>
void Pile<T>::depiler()
{
  assert(sommet!=NULL); // ou : assert(sommet)
  Cellule* suivante = sommet->suivante;
  delete sommet;
  sommet = suivante;
}
```

Depiler (version alternative 1)

```
template <class T>
T Pile<T>::depiler() {
    assert(sommet!=NULL);
    T element = sommet->contenu;
    Cellule* anciensommet = sommet;
    sommet = sommet->suivante;
    delete anciensommet;
    return element;
}
```

Depiler (version alternative 2)

```
template <class T>
void Pile<T>::depiler(T& sortie) {
    assert(sommet!=NULL);
    sortie = sommet->contenu;
    Cellule* anciensommet = sommet;
    sommet = sommet->suivante;
    delete anciensommet;
}
```

Analyse des opérations

Opération	Cas Moyen	Pire cas
empiler(*e*)	O(1)	O(1)
depiler()	O(1)	O(1)
sommet()	O(1)	O(1)
taille()	O(1)	O(1)
vide()	O(1)	O(1)
vider()	O(n)	O(n)

Remarques

- Hypothèse requise : allocation et désallocation de mémoire (opérateurs new et delete) en temps constant, i.e. O(1).
- Complexité spatiale : on a besoin d'un pointeur par cellule. Négligeable quand les objets sont gros.

Rappel

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

  bool vide() const;
  const T& sommet() const;
  void empiler(const T&);
  void depiler();
  // ...
private:
  class Cellule{
    public:
      Cellule(const T& c, Cellule* s) : contenu(c), suivante(s){}
      T contenu;
      Cellule* suivante;
  };
  Cellule* sommet;
};
```

Opérateur ==

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




};
```

Opérateur =

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




  return *this;
};
```