Estructures lineals I

Programació 2

Facultat d'Informàtica de Barcelona, UPC

Conrado Martínez

Primavera 2019

- Apunts basats en els d'en Ricard Gavaldà
- Aquestes transparències **no** substitueixen els apunts de l'assignatura, els complementen

Estructures lineals

Una estructura lineal C és un conjunt d'elements d'un cert tipus `T`

$$C = [a_1, a_2, \ldots, a_n]$$

en el que es defineix una relació de successió

- Per tot i, $1 \leq i < n$, a_{i+1} és el successor de a_i. L'últim element a_n no té successor.
- Per tot i, $1 < i \leq n$, a_{i-1} és el predecessor de a_i. El primer element a_1 no té predecessor.

Si $n = 0$ la estructura està buida

Estructures lineals

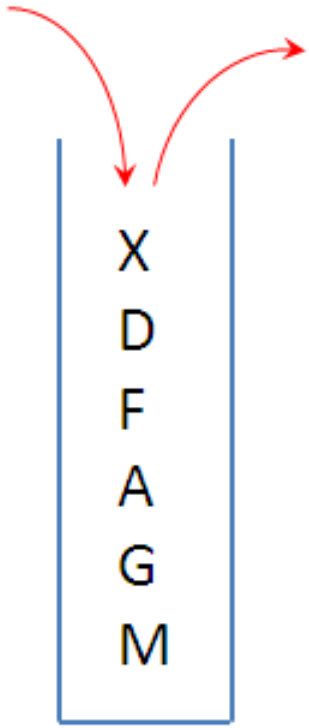
- Piles (*stack*)
- Cues (*queue*)
- Llistes (*list*)

El tipus pila (`stack`)

La classe `stack`

Ofereix tres operacions bàsiques:

- Afegir un nou element al final (*empilar*)
- Treure l'últim element (*desempilar*)
- Examinar l'últim element (*cim*)



Especificació de stack

La classe stack

```
template <class T> class stack {
public:
// Constructores

/* Pre: cert */
/* Post: crea una pila buida  */
stack();

// Destructora
~stack();
```

Especificació de stack

La classe stack

```
// Modificadores

/* Pre: la pila és [a_1,...,a_n], n ≥ 0 */
/* Post: s'afegit l'element x com a últim de la pila, es
         a dir, la pila és ara [a_1,...,a_n,x] */
void push(const T& x);

/* Pre:  la pila és [a_1,...,a_n] i no està buida (n > 0) */
/* Post: s'ha eliminat el darrer element de la pila original,
         es a dir, la pila ara és [a_1,...,a_{n-1}] */
void pop();
```

Especificació de stack

La classe stack

```
// Consultores

/* Pre: la pila és [a_1,...,a_n] i no està buida (n > 0) */
/* Post: Retorna a_n */
T top() const;

/* Pre: cert */
/* Post: Retorna cert si i només si la pila està buida */
bool empty() const;

private:
   ...
};
```

Advertiment

- En molts d'aquests exercicis (i els de cues), tenim problemes perquè intentem accedir a la pila (o la cua) d'una manera que no és la habitual per a piles o cues; per exemple, accedir a tots els elements i no només al que és accessible
- Això genera problemes, per exemple, que la pila és destruïda i cal reconstruir-la, o copiar-la abans.
- Probablement significa que *si necessitem fer això, hauríem d'usar una llista i no una pila o cua*
- Preneu-vos pel valor pedagògic

Alçària d'una pila

L'alçària de la pila $p = [a_1, \ldots, a_n]$ és n, el seu nombre d'elements

```
/* Pre: cert */
/* Post: retorna el nombre d'elements de p */
template <typename T>
int alcaria(const stack<T>& p);
```

Nota. És un exercici. La classe `stack<T>` estàndar té el mètode `size` que retorna el nombre d'elements de la pila sobre la qual s'aplica.

Alçària d'una pila, versió iterativa

```
template <typename T>
int alcaria(const stack<T>& p) {
    int n = 0;
    while (not p.empty()) {
        ++n;
        p.pop();
    }
    return n;
}
```

Veieu algún problema amb aquesta implementació?

Alçària d'una pila

Apliquem un mètode modificador (pop) però la pila no es pot modificar ja que és const &!
Solucions:

- passar-la per valor, es farà una còpia de l'argument en fer la crida
- fer una còpia en un variable local

```
temlate <typename T>
int alcaria(const stack<T>& p) {
    stack<T> aux = p;
    int n = 0;
    while (not aux.empty()) {
        ++n; aux.pop();
    }
    return n;
}
```

Alçària d'una pila

Més solucions:

- passar per referència, la funció calcula la mida de la pila **i** deixa buida la pila!
  - L'usuari és responsable de fer una còpia prèvia, si vol
  - Risc: que se n'obliddi
  - Avantatge: no hi ha cap còpia, si no fa falta

Alçària d'una pila, versió recursiva

```
/* Pre: p = P */
/* Post: retorna n, el nombre d'elements de P i
         p = [] buida */
template <typename T>
int alcaria(stack<T>& p) {
    if (p.empty()) return 0;
    else {
        p.pop();
        return 1 + alcaria(p);
    }
}
```

Si passem p per valor, fariem una una còpia a cada crida $\Rightarrow$ temps i memòria quadràtics en la mida de p!

Suma dels elements d'una pila

Si $P = [a_1, \ldots, a_n]$, suma$(P) = a_1 + \cdots + a_n$

```
/* Pre: p = P */
/* Post: retorna suma(P), i p = [] buida */
int suma(stack<int>& p);
```

Suma dels elements d'una pila

```
int suma(stack<int>& p) {
    if (p.empty()) return 0;
    else {
        int x = p.top();
        p.pop();
        return x + suma(p);
    }
}
```

Suma dels elements d'una pila

Podem aconseguir que la pila p retingui el seu valor original, però encara hem de permetre que s'apliquin mètodes (passem per referència, no per referència constant) modificadors.

```
/* Pre: p = P */
/* Post: retorna suma(P), i p = P */
int suma(stack<int>& p) {
    if (p.empty()) return 0;
    else {
        int x = p.top();
        p.pop();
        int res = x + suma(p);
        p.push(x);
        return res;
    }
}
```

Cerca en una pila

Donada una pila p i un element x, determinar si x apareix en p

```
/* Pre: p = P */
/* Post: retorna cert si i només si x és un element de P */
bool cerca(stack<int>& p, int x);
```

Cerca en una pila

```
bool cerca(stack<int>& p, int x) {
    if (p.empty()) return false;
    else if (p.top() == x) return true;
    else {
        p.pop();
        return cerca(p, x);
    }
}
```

Cerca en una pila

- Si la pila p no conté x, llavors després de la crida $\texttt{cerca}(p, x)$ la pila està buida
- Si la pila $p = [a_1, \ldots, a_{i-1}, x, a_{i+1}, \ldots, a_n]$ amb $a_j \neq x$ per a tot j, $1 \leq j < i$, llavors després de la crida $\texttt{cerca}(p, x)$ tindrem $p = [a_{i+1}, \ldots, a_n]$

Cerca en una pila d'Estudiants

Solució iterativa

```
/* Pre: x > 0, p = P */
/* Post: retorna cert si i només si hi ha algun
         estudiant amb dni x a P */
bool cerca(stack<Estudiant>& p, int x) {
    while (not p.empty()){
      if (p.top().consultar_DNI() == x)
        return true;
      else
        p.pop();
    }
    return false;
}
```

Sumar un nombre a una pila

```
/* Pre: p = [a_1, ..., a_n] */
/* Post: p = [a_1 + k, a_2 + k, ..., a_n + k] */
void suma_k(stack<int>& p, int k);
```

p.ex, $p = [3, 6, 2, 1], k = 2 \rightarrow p = [5, 8, 4, 3]$

Sumar un nombre a una pila: Versió recursiva

```
void suma_k(stack<int>& p, int k) {
    if (not p.empty()) {
        int x = p.top();
        p.pop();
        suma_k(p,k);
        p.push(x+k);
    }
}
```

Sumar un nombre a una pila: Versió iterativa

Necessitarem una pila auxiliar ...

```
void suma_k(stack<int>& p, int k) {
    stack<int> aux;
    while (not p.empty()) {
        aux.push(p.top()+k);
        p.pop();
    }
    p = aux;
}
```

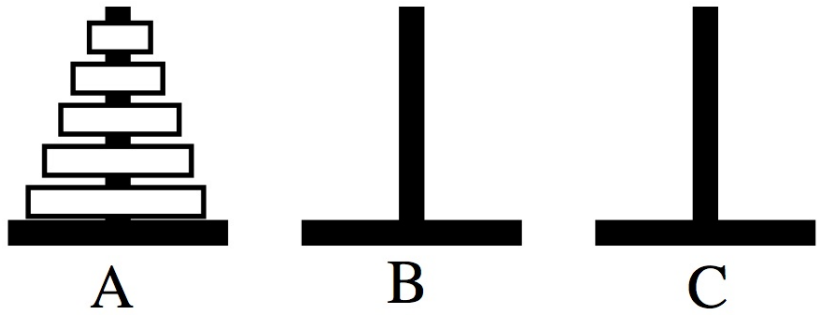
Problema?

Sumar un nombre a una pila: Versió iterativa

Solucions: 1) un altre bucle 2) cridar a una funció "revessar pila" 3) cridar a sumar_k$(p, 0)$

```
void suma_k(stack<int>& p, int k) {
    stack<int> aux;
    while (not p.empty()) {
        aux.push(p.top()+k);
        p.pop();
    }
    while (not aux.empty()) {
      p.push(aux.top());
      aux.pop();
    }
}
```

Aplicacions de les piles: transformació recursiu-iteratiu



Volem dissenyar un algorisme

```
typedef char pole;
void hanoi(int N, pole org, pole aux, pole dst);
```

que imprimeixi els moviments necessaris per a traslladar N discos des del pal `org` fins al pal `dst`, utilitzant el pal `aux` com a pal auxiliar.

Aplicacions de les piles: transformació recursiu-iteratiu

```
void hanoi(int N, pole org, pole aux, pole dst) {
    if (N==1)
      cout << "Mou un disc de " << org
           << " a " << dst << endl;
    else {
      hanoi(N - 1, org, dst, aux);
      cout << "Mou un disc de " << org
           << " a " << dst << endl;
      hanoi(N - 1, aux, org, dst);
    }
}
```

Aplicacions de les piles: transformació recursiu-iteratiu

```
class HanoiTask {
private:
        int n_;
        pole org_, aux_, dst_;
public:
        HanoiTask(int n, pole org, pole aux, pole dst) {
            n_ = n; org_ = org; aux_ = aux; dst_ = dst;
        }
```

Aplicacions de les piles: transformació recursiu-iteratiu

```
class HanoiTask {
        ...
        // consultoras
        int ndiscos() const { return n_; }
        pole origen() const { return org_; }
        pole auxiliar() const { return aux_; }
        pole desti() const { return dst_; }

        // Pre: ndiscos() == 1
        // Post: escriu en el cout el moviment necessari per
        //       a realitzar la tasca (moure un únic disc)
        void escriu_moviment() {
           cout << "Mou disc de " << org_;
           cout << " a " << dst_ << endl;
        }
};
```

Aplicacions de les piles: transformació recursiu-iteratiu

```
void hanoi(int N, pole org, pole aux, pole dst) {
  HanoiTask initial_task(N, org, aux, dst);
  stack<HanoiTask> S;
  S.push(initial_task);
  while (not S.empty()) {
    HanoiTask curr = S.top();
    S.pop();
    if (curr.ndiscos() == 1) {
      curr.escriu_moviment();
    } else {
      // curr.ndiscos() > 1
      ...
    }
  }
}
```

Aplicacions de les piles: transformació recursiu-iteratiu

```
...
} else {
  // curr.ndiscos() > 1

  HanoiTask left_task(curr.ndiscos()-1,
                curr.origen(), curr.desti(), curr.auxiliar());
  HanoiTask middle_task(1, curr.origen(), '*', curr.dest());
  HanoiTask right_task(curr.ndiscos()-1,
                curr.auxiliar(), curr.origen(), curr.desti());
  S.push(right_task);
  S.push(middle_task);
  S.push(left_task);
}
```

Aplicacions de les piles: avaluació d'expressions

En una expressió en notació postfixa s'escriu en primer lloc els operands i a continuació l'operador. Per exemple

```
8 7 + 13 4 - *
```

és l'expressió $(8 + 7) * (13 - 4)$. Un advantatge de la notació postfixa és que mai són necessaris els parèntesis. Direm que les expressions estàn formades per una seqüència de *tokens*. Un token és un valor (8, 7, 13, ...) o un operador (+, -, *, ...). Per simplificar assumirem que tots els operadors són binaris

Aplicacions de les piles: avaluació d'expressions

Escriu una funció que, donat un vector de `Token` que representa una expressió en notació postfixa, avalua l'expressió tornant el seu valor.

```
class Token {
public:
...
};

int avalua(const vector<Token>& expr);
```

Aplicacions de les piles: avaluació d'expressions

```
class Token {
public:
...
bool es_operand() const;
bool es_operador() const;
int valor() const;
char operador() const;
...
};
```

Aplicacions de les piles: avaluació d'expressions

```
int avalua(const vector<Token>& expr) {
  stack<int> S;
  for (int i = 0; i < expr.size(); ++i) {
    if (expr[i].es_operand()) {
      S.push(expr[i].valor());
    } else { // expr[i] és un operador
      int op2 = S.top(); S.pop();
      int op1 = S.top(); S.pop();
      char c = expr[i].operador();
      S.push(opera(c, op1, op2));
    }
  }
  // S.size() == 1
  return S.top();
}
```

Aplicacions de les piles: avaluació d'expressions

```
int opera(char c, int x, int y) {
  if (c == '+') return x+y;
  if (c == '*') return x*y;
  ...
}
```