Data Structures and Algorithms (資料結構與演算法)

Lecture 3: Stack and Queue

Hsuan-Tien Lin (林軒田)

htlin@csie.ntu.edu.tw

Department of Computer Science
& Information Engineering

National Taiwan University
(國立台灣大學資訊工程系)

Stacks

Stack

- object: a container that holds some elements
- action: [constant-time] push (to the top), pop (from the top)

- last-in-first-out (LIFO): 擠電梯, 洗盤子
- very restricted data structure, but important for computers
—will discuss some cases later

A Simple Application: Parentheses Balancing

- in C, the following characters show up in pairs: (), [], {}, ""

```
good: {xxx(xxxxxx)xxxxx"xxxx"x}
bad:  {xxx(xxxxxx}xxxxx"xxxx"x}
```

- the LISP programming language

```
(append (pow (* (+ 3 5) 2) 4) 3)
```

how can we check parentheses balancing?

Stack Solution to Parentheses Balancing

inner-most parentheses pair $\Longrightarrow$ top-most plate

’(’: 堆盤子上去; ’)’: 拿盤子下來

Parentheses Balancing Algorithm

```
for each c in the input do
    if c is a left character then
        push c to the stack
    else if c is a right character then
        pop d from the stack and check if match
    end if
end for
```

many more sophisticated use in compiler design

System Stack

- recall: function call ⇔ 拿新的草稿紙來算
- old (original) scrap paper: temporarily not used, 可以壓在下面

System Stack: 一疊草稿紙, each paper (stack frame) contains

- return address: where to return to the previous scrap paper
- local variables (including parameters): to be used for calculating within this function
- previous frame pointer: to be used when escaping from this function

some related issues: stack overflow? security attack?

Stacks Implemented on Array (5.1.4)

Reading Assignment

be sure to go ask the TAs or me if you are still confused

Stacks Implemented on Linked List (5.1.5)

Reading Assignment

be sure to go ask the TAs or me if you are still confused

Stack for Expression Evaluation (Supplementary)

$$a/b - c + d * e - a * c$$

- precedence: $\{*, /\}$ first; $\{+, -\}$ later
- steps
  - $f = a/b$
  - $g = f - c$
  - $h = d * e$
  - $i = g + h$
  - $j = a * c$
  - $\ell = i - j$

Postfix Notation

same operand order, but put “operator” **after** needed operands
—can “operate” immediately when seeing operator
—no need to look beyond for precedence

Postfix from Infix (Usual) Notation

- infix:

$$3 \ / \ 4 \ - \ 5 \ + \ 6 \ * \ 7 \ - \ 8 \ * \ 9$$

- parenthesize:

$$3 \ / \ 4 \ - \ 5 \ + \ 6 \ * \ 7 \ - \ 8 \ * \ 9$$

- for every triple in parentheses, switch orders

- remove parentheses

difficult to parenthesize efficiently

Evaluate Postfix Expressions

$$3 4 / 5 - 6 7 * + 8 9 * -$$

- how to evaluate? left-to-right, “operate” when see operator
- 3, 4, / ⇒ 0.75
- 0.75, 5, - ⇒ -4.25
- -4.25, 6, 7, * ⇒ -4.25, 42 (note: -4.25 stored for latter use)
- -4.25, 42, + ⇒ 37.75
- 37.75, 8, 9, * ⇒ 37.75, 72 (note: 37.75 stored for latter use)
- 37.75, 72, - ⇒ ...

stored where?
stack so closest operands will be considered first!

Stack Solution to Postfix Evaluation

Postfix Evaluation

```
for each token in the input do
    if token is a number then
        push token to the stack
    else if token is an operator then
        sequentially pop operands $a_{t-1}, \cdots, a_0$ from the stack
        push token$(a_0, a_1, a_{t-1})$ to the stack
    end if
end for
return the top of stack
```

matches closely with the definition of postfix notation

One-Pass Algorithm for Infix to Postfix

infix ⇒ postfix efficiently?

- at /, not sure of what to do (need later operands) so **store**

$$a/b - c + d * e - a * c$$

- at -, know that a / b can be a b / because - is of lower precedence

$$a/b-c + d * e - a * c$$

- at +, know that ? - c can be ? c - because + is of same precedence but {-, +} is left-associative

$$a/b - c+d * e - a * c$$

- at *, not sure of what to do (need later operands) so **store**

$$a/b - c + d*e - a * c$$

stored where? **stack** so closest operators will be considered first!

Stack Solution to Infix-Postfix Translation

```
for each token in the input do
    if token is a number then
        output token
    else if token is an operator then
        while top of stack is of higher (or same) precedence do
            pop and output top of stack
        end while
        push token to the stack
    end if
end for
```

- here: infix to postfix with operator stack
  —closest operators will be considered first
- recall: postfix evaluation with operand stack
  —closest operands will be considered first
- mixing the two algorithms (say, use two stacks): simple calculator

Some More Hints on Infix-Postfix Translation

```
for each token in the input do
    if token is a number then
        output token
    else if token is an operator then
        while top of stack is of higher (or same) precedence do
            pop and output top of stack
        end while
        push token to the stack
    end if
end for
```

- for left associativity and binary operators
    - right associativity? same precedence needs to wait
    - unary/trinary operator? same
- parentheses? higest priority
    - at '(', cannot pop anything from stack —like seeing '*' while having '+' on the stack
    - at ')', can pop until '(' —like parentheses matching

Queues

Queue

- object: a container that holds some elements
- action: [constant-time] enqueue (to the rear), dequeue (from the front)

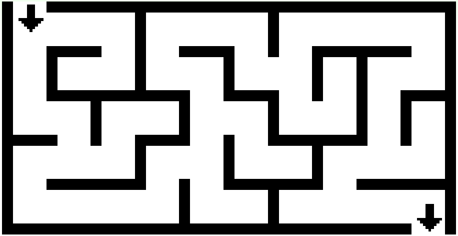
- first-in-first-out (FIFO): 買票, 印表機
- also very restricted data structure, but also important for computers

Queues Implemented on Circular Array (5.2.4)

Reading Assignment

be sure to go ask the TAs or me if you are still confused

The Maze Problem



http://commons.wikimedia.org/wiki/File:Maze01-01.png

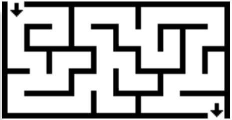
given a (2D) maze, is there a way out?

Recursive Algorithm

$\text{GET-OUT-RECURSIVE}(m, (0, 0))$

Getting Out of Maze Recursively

```
GET-OUT-RECURSIVE(Maze $m$, Postion $(i, j)$)
  mark $(i, j)$ as visited
  for each unmarked $(k, \ell)$ reachable from $(i, j)$ do
    if $(k, \ell)$ is an exit then
      return TRUE
    end if
    if GET-OUT-RECURSIVE$(m, (k, \ell))$ then
      return TRUE
    end if
  end for
  return FALSE
```



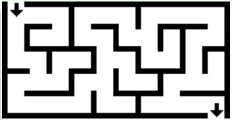
Recursion (Reading Assignment: Section 3.5)

- a function call to itself
- be ware of **terminating conditions**
- can represent programming intentions clearly
- at the expense of **“space”** (why?)

From Recursion to Stack

Getting Out of Maze by Stack

```
GET-OUT-STACK(Maze $m$, Postion $(i, j)$)
    while stack not empty do
        $(i, j) \leftarrow$ pop from stack
        mark $(i, j)$ as visited
        for each unmarked $(k, \ell)$ reachable from $(i, j)$ do
            if $(k, \ell)$ is an exit then
                return TRUE
            end if
            push $(k, \ell)$ to stack [and mark $(k, \ell)$ as todo]
        end for
    end while
    return FALSE
```

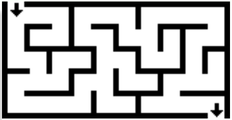


- similar result to recursive version, but conceptually different
  - recursive: one path on the system stack
  - stack: many positions-to-be-explored on the user stack

A General Maze Algorithm

Getting Out of Maze by Container

```
GET-OUT-CONTAINER(Maze $m$, Postion $(i, j)$)
  while container not empty do
    $(i, j) \leftarrow$ remove from container
    mark $(i, j)$ as visited
    for each unmarked $(k, \ell)$ reachable from $(i, j)$ do
      if $(k, \ell)$ is an exit then
        return TRUE
      end if
      insert $(k, \ell)$ to container [and mark $(k, \ell)$ as todo]
    end for
  end while
  return FALSE
```

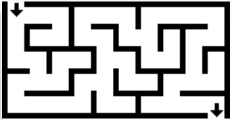


- if “random” remove from **container**: “random walk” to exit

Maze From Stack to Queue

Getting Out of Maze by Queue

```
GET-OUT-QUEUE(Maze $m$, Postion $(i, j)$)
  while queue not empty do
    $(i, j) \leftarrow$ dequeue from queue
    mark $(i, j)$ as visited
    for each unmarked $(k, \ell)$ reachable from $(i, j)$ do
      if $(k, \ell)$ is an exit then
        return TRUE
      end if
      enqueue $(k, \ell)$ to queue [and mark $(k, \ell)$ as todo]
    end for
  end while
  return FALSE
```



- use of stack/queue: store the yet-to-be-explored positions
- stack version : first (lexicographically) way out (explore deeply)
- queue version : shortest way out (explore broadly)

Deques

Deque = Stack + Queue + push_front

- object: a container that holds some elements
- action: [constant-time] push_back (like push and enqueue), pop_back (like pop), pop_front (like dequeue), push_front

- application: job scheduling

Deques Implemented on Doubly-linked List (5.3.2)

Reading Assignment

be sure to go ask the TAs or me if you are still confused

Some Useful Implementations in C++

Standard Template Library (STL)

- container `vector`: dynamically growing dense array
- container `list`: doubly-linked list
- container `deque`: “chunked” linked-list implementation of deque
- container adapter `stack`: turning some container to a stack

```
template <typename T, typename Container = deque<T> >
class stack;
```

- container adapter `queue`: turning some container to a queue

```
template <typename T, typename Container = deque<T> >
class queue;
```

Some Useful Implementations in C++

```cpp
#include <vector>
#include <stack>
#include <queue>
using namespace std;
vector<int> intarray;
stack<char, vector<char> > charstackonvector;
queue<double> doublequeue;
intarray.resize(20); intarray[3] = 5;
charstack.push_back('(');
char c = charstack.pop_back();
doublequeue.push_back(3.14);
double d = doublequeue.pop_front();
```