Introduction to Object-Oriented Programming

COMP2011: Data Structures — Stack & Queue

Dr. Cindy Li
Dr. Brian Mak
Dr. Dit-Yan Yeung

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

Data Structures

- Computer science is the study of how to process information (data) efficiently using computers.
- A data structure helps store, organize, and manipulate data in a particular way so that they can be processed efficiently by computers.
- Different applications require different data structures.
- Examples: array, linked list, (binary) tree, stack, queue, etc.
- An abstract data type (ADT) is the mathematical model of a data structure that is independently of the its implementation. It may be used to analyze the efficiency of algorithms.

Stack and Queue

Stack and queue let you insert and remove items at the ends only, not in the middle.

Part I

Stack




Stack: How it Works

Consider a pile of cookies.

- more cookies: new cookies are added on top, one at a time.
- fewer cookies: cookies are consumed one at a time, starting at the top.

As an ADT, insertions and removals of items on a stack are based on the *last-in first-out (LIFO)* policy.

It supports:

- Data: an ordered list of data/items.
- Operations (major ones):

  top : get the value of the top item

  push : add a new item to the top

  pop : remove an item from the top

Stack of int Data — stack.h

```
#include <iostream>                                    /* File: int-stack.h */
#include <cstdlib>
using namespace std;
const int BUFFER_SIZE = 5;

class int_stack
{
  private:
    int data[BUFFER_SIZE];                          // Use an array to store data
    int top_index;                                  // Starts from 0; -1 when empty

  public:
    // CONSTRUCTOR member functions
    int_stack(void);                                // Default constructor
    // ACCESSOR member functions: const => won't modify data members
    bool empty(void) const;                         // Check if the stack is empty
    bool full(void) const;                          // Check if the stack is full
    int size(void) const;                           // Give the number of data currently stored
    int top(void) const;                            // Retrieve the value of the top item
    // MUTATOR member functions
    void push(int);                                 // Add a new item to the top of the stack
    void pop(void);                                 // Remove the top item from the stack
};
```

Stack of int Data — Constructors, Assessors

```
#include "int-stack.h"                                  /* File: int-stack1.cpp */

            /***** Default CONSTRUCTOR member function *****/
int_stack::int_stack(void) { top_index = -1; }   // Create an empty stack

            /***** ACCESSOR member functions *****/
// Check if the int_stack is empty
bool int_stack::empty(void) const { return (top_index == -1); }

// Check if the int_stack is full
bool int_stack::full(void) const { return (top_index == BUFFER_SIZE-1); }

// Give the number of data currently stored
int int_stack::size(void) const { return top_index+1; }

// Retrieve the value of the top item
int int_stack::top(void) const
{
    if ( !empty( ) )
        return data[top_index];

    cerr << "Warning:  Stack is empty; can't retrieve any data!" << endl;
    exit(-1);
}
```

Stack of int Data — Mutators

```
#include "int-stack.h"                                  /* File: int-stack2.cpp */

        /***** MUTATOR member functions *****/
void int_stack::push(int x)              // Add a new item to the top of the stack
{
    if ( !full( ) )
        data[++top_index] = x;
    else
    {
        cerr << "Error:  Stack is full; can't add (" << x << ")!" << endl;
        exit(-1);
    }
}

void int_stack::pop(void)                    // Remove the top item from the stack
{
    if ( !empty( ) )
        --top_index;
    else
    {
        cerr << "Error:  Stack is empty; can't remove any data!" << endl;
        exit(-1);
    }
}
```

Stack of int Data — Test Program

```
#include "int-stack.h"                                  /* File: int-stack-test.cpp */

void print_stack_info(const int_stack& s)
{
    cout << "No.  of data currently on the stack = " << s.size( ) << "\t";
    if ( !s.empty( ) ) cout << "Top item = " << s.top( );
    cout << endl << "Empty:   " << boolalpha << s.empty( );
    cout << "\t\t" << "Full:   " << boolalpha << s.full( ) << endl;
}

int main(void)
{
    int_stack a; print_stack_info(a);
    a.push(4); print_stack_info(a);
    a.push(15); print_stack_info(a);
    a.push(26); print_stack_info(a);
    a.push(37); print_stack_info(a);
    a.pop( ); print_stack_info(a);
    a.push(48); print_stack_info(a);
    a.push(59); print_stack_info(a);

    return 0;
}  /* compile: g++ -L. -o int-stack-test int-stack-test.cpp -lintstack */
```

Example: Decimal to Binary Conversion — Illustration

- e.g., $26_{(10)} = 11010_{(2)}$
- Algorithm to convert $N_{(10)} = M_{(2)}$:

Step 1 : divide N by 2 successively

Step 2 : each time push the remainder onto a stack

Step 3 : print the answer by popping the stack successively

2	26		
2	13	...	0
2	6	...	1
2	3	...	0
2	1	...	1
	0	...	1

Example: Decimal to Binary Conversion

```
#include "int-stack/int-stack.h"                    /* File: decimal2binary.cpp */

int main(void) // Convert +ve decimal number to binary number using an stack
{
    int_stack a;
    int x, number;

    while (cin >> number)
    {                                                // Conversion: decimal to binary
        for (x = number; x > 0; x /= 2)
            a.push(x % 2);

        // Print a binary that is stored on a stack
        cout << number << "(base 10) = ";
        while ( !a.empty( ) )
        {
            cout << a.top( );
            a.pop( );
        }
        cout << "(base 2)" << endl;
    }

    return 0;
}  // Compile: g++ -o decimal2binary -Lint-stack decimal2binary.cpp -lintstack
```

Example: Balanced Parentheses — Illustration

- e.g., [()][()()]() is balanced but [(]) is not.
- Algorithm to check balanced parentheses:

Step 1 : Scan the given character expression from left to right.

Step 2 : If a left paranthesis is read, push it onto a stack.

Step 3 : If a right paranthesis is read, check if its matching left parenthesis is on the top of the stack.

Step 4 : If Step 3 is true, pop the stack and continue.

Step 5 : If Step 3 is false, return false and stop.

Step 6 : If the end of the expression is reached, check if the stack is empty.

Step 7 : If Step 6 is true, return true otherwise false.

Example: Balanced Parentheses

```
#include "char-stack/char-stack.h"                    /* File: balanced-paren.cpp */
const char L_PAREN = '('; const char R_PAREN = ')';
const char L_BRACE = '{'; const char R_BRACE = '}';
const char L_BRACKET = '['; const char R_BRACKET = ']';

bool check_paren(char_stack& a, char c);
bool balanced_paren(const char* expr)
{
    char_stack a;
    for (const char* s = expr; *s != '\0'; ++s)                          // Step 1
        switch (*s)
        {
            case L_PAREN: case L_BRACE: case L_BRACKET:                  // Step 2
                a.push(*s); break;

            case R_PAREN:                                                // Step 3-5
                if (!check_paren(a, L_PAREN)) return false; break;
            case R_BRACE:
                if (!check_paren(a, L_BRACE)) return false; break;
            case R_BRACKET:
                if (!check_paren(a, L_BRACKET)) return false; break;
        }

    return a.empty( );                                                   // Step 6-7
}
```

Example: Balanced Parentheses ..

```
bool check_paren(char_stack& a, char c)
{
    if (a.empty( )) return false;
    if (a.top( ) != c) return false;

    a.pop( );
    return true;
}

int main(void)                    // To check if a string has balanced parantheses
{
    char expr[1024];
    cout << "Input an expression containing parentheses:  ";
    cin >> expr;

    cout << boolalpha << balanced_paren(expr) << endl;
    return 0;
}
/* Compile:
   g++ -Lchar-stack -o balanced-paren balanced-paren.cpp -lcharstack */
```

Part II

Queue

Queue: How it Works

Consider the case when people line up for tickets.

- more people: new customers join the back of a queue, one at a time.
- fewer people: the customer at the front buys a ticket and leaves the queue.

As an ADT, insertions and removals of items on a queue are based on a *first-in first-out (FIFO)* policy.

It supports:

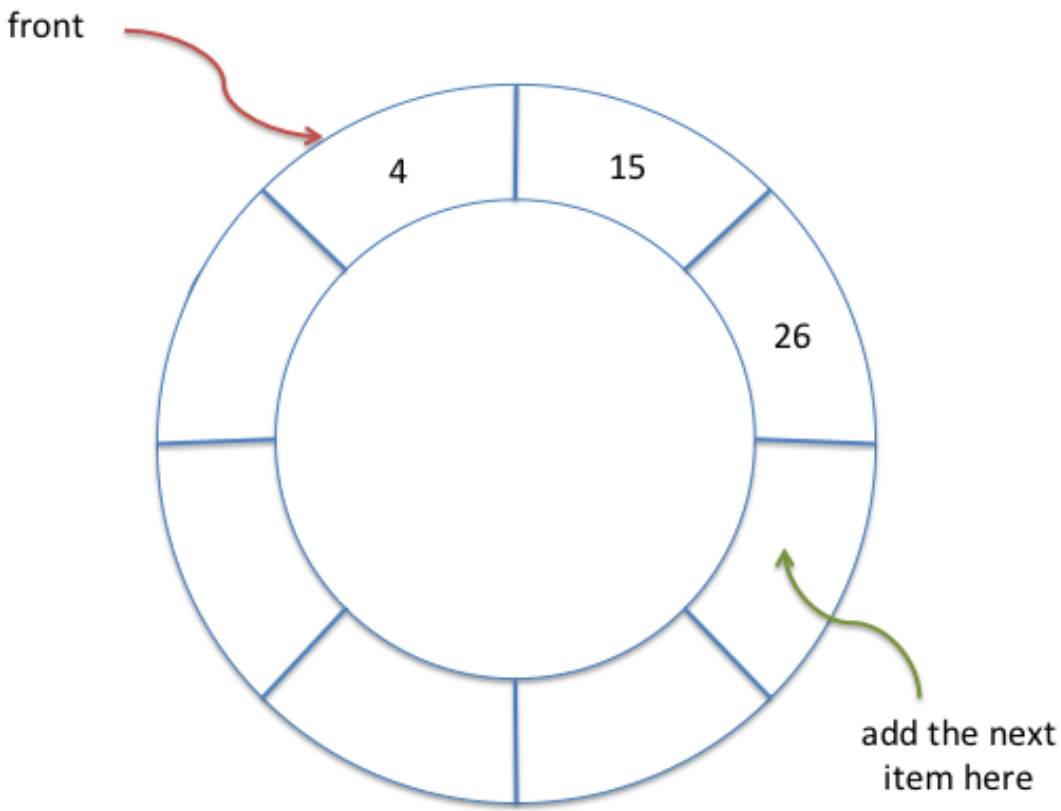
- Data: an ordered list of data/items.
- Operations (major ones):

front : get the value of the front item

enqueue : add a new item to the back

dequeue : remove an item from the front

Circular Queue of int Data — Illustration



Circular Queue of int Data — queue.h

```
#include <iostream>                                    /* File: int-queue.h */
#include <cstdlib>
using namespace std;
const int BUFFER_SIZE = 5;

class int_queue                                        // Circular queue
{
  private:
    int data[BUFFER_SIZE];                             // Use an array to store data
    int num_items;                                     // Number of items on the queue
    int first;                                         // Index of the first item; start from 0

  public:
    // CONSTRUCTOR member functions
    int_queue(void);                                   // Default constructor
    // ACCESSOR member functions: const => won't modify data members
    bool empty(void) const;                            // Check if the queue is empty
    bool full(void) const;                             // Check if the queue is full
    int size(void) const;                              // Give the number of data currently stored
    int front(void) const;                             // Retrieve the value of the front item
    // MUTATOR member functions
    void enqueue(int);                                 // Add a new item to the back of the queue
    void dequeue(void);                                // Remove the front item from the queue
};
```

Circular Queue of int Data — Constructors, Assessors

```
#include "int-queue.h"                                  /* File: int-queue1.cpp */

                /***** Default CONSTRUCTOR member function *****/
// Create an empty queue
int_queue::int_queue(void) { first = 0; num_items = 0; }

                /***** ACCESSOR member functions *****/
// Check if the int_queue is empty
bool int_queue::empty(void) const { return (num_items == 0); }

// Check if the int_queue is full
bool int_queue::full(void) const { return (num_items == BUFFER_SIZE); }

// Give the number of data currently stored
int int_queue::size(void) const { return num_items; }

// Retrieve the value of the front item
int int_queue::front(void) const
{
    if ( !empty( ) ) return data[first];
    cerr << "Warning:  Queue is empty; can't retrieve any data!" << endl;
    exit(-1);
}
```

Circular Queue of int Data — Mutators

```
#include "int-queue.h"                                  /* File: int-queue2.cpp */

void int_queue::enqueue(int x)      // Add a new item to the back of the queue
{
    if ( !full( ) )
    {
        data[(first+num_items) % BUFFER_SIZE] = x;
        ++num_items;
    } else {
        cerr << "Error:  Queue is full; can't add (" << x << ")!" << endl;
        exit(-1);
    }
}

void int_queue::dequeue(void)          // Remove the front item from the queue
{
    if ( !empty( ) )
    {
        first = (first+1) % BUFFER_SIZE;
        --num_items;
    } else {
        cerr << "Error:  Queue is empty; can't remove any data!" << endl;
        exit(-1);
    }
}
```

Circular Queue of int Data — Test Program

```
#include "int-queue.h"                                  /* File: int-queue-test.cpp */

void print_queue_info(const int_queue& a) {
     cout << "No.  of data currently on the queue = " << a.size( ) << "\t";
     if (!a.empty( )) cout << "Front item = " << a.front( );
     cout << endl << "Empty:   " << boolalpha << a.empty( );
     cout << "\t\t" << "Full:   " << boolalpha << a.full( ) << endl;
}

int main(void) {
     int_queue a; print_queue_info(a);
     a.enqueue(4); print_queue_info(a);
     a.enqueue(15); print_queue_info(a);
     a.enqueue(26); print_queue_info(a);
     a.enqueue(37); print_queue_info(a);
     a.dequeue( ); print_queue_info(a);
     a.enqueue(48); print_queue_info(a);
     a.enqueue(59); print_queue_info(a);
     a.dequeue( ); print_queue_info(a);
     a.dequeue( ); print_queue_info(a);
     a.dequeue( ); print_queue_info(a);
     a.dequeue( ); print_queue_info(a);
     a.dequeue( ); print_queue_info(a);
     return 0;
}   /* compile: g++ -L. -o int-queue-test int-queue-test.cpp -lintqueue */
```