


DATA STRUCTURES AND ALGORITHMS


Lecture 06

Learning outcome 2



ADT STACK

Strana • 3



ADT stack

- ADT stack is a list or a linked list with the constraint:
 - All insertions and removals of elements must be done at only one side of the list, which is then called the top
- Other names are:
 - LIFO (Last In First Out) list
 - FILO (First In Last Out) list

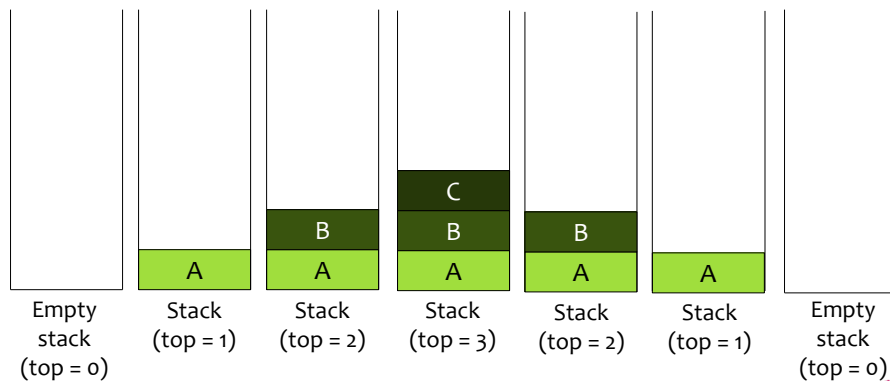


Strana • 4



LIFO/FILO principle

- Each stack has a top - this is the last element added to the stack and the first to be removed from the stack



Strana • 5



Operations

- The ADT stack usually defines the following operations:
 - Pushing (adding) a new element on the top $O(1)$
 - If there is no space on the stack => *overflow*
 - Popping (removing) an element from the top in $O(1)$
 - If the stack is empty => *underflow*
 - Read the value of the element on the top in $O(1)$
 - Retrieve the number of elements on the stack in $O(1)$
 - Check if the stack is empty in $O(1)$

Strana • 6



Basic usage examples

- Program execution
 - Each function call puts a stack frame on the system stack
 - Returning from the function removes the stack frame from the system stack
- Writing data from the file in reverse order
 - We read data from one file and load it on a stack, then empty the stack and write the data to another file
- UNDO functionality
 - Each operation is placed on a stack
 - By issuing the UNDO command, we remove the last element from the stack

Strana • 7



More advanced usage examples

- Calculation of mathematical expressions in reverse Polish notation (RPN)
- Parsing code blocks by the compiler
- Finding your way through a maze (applying backtracking)
- History of pages visited in a web browser
- Check the consistency of tags in HTML and XML

Strana • 8



ADAPTER

Strana • 9



Introduction

- An adapter is a class A that contains a variable whose data type is class B
 - We say that class A adapts class B to our needs
 - Often referred to as a wrapper
- Class A usually provides the user with a more useful / simpler interface than class B
- Class A in its methods actually calls class B methods
- More details about adapters: NRAKO (Advanced application development based on templates), year 4

Strana • 10



Example (1/4)

- Imagine we have a class that knows how to work with a variety of Internet resources:

```
class HttpUtilities {
public:
    string http_get(string url, int port, bool is_secure);
    string http_post(string url, int port, bool is_secure);
    string http_put(string url, int port, bool is_secure);
    string http_delete(string url, int port, bool is_secure);
    void set_credentials(string username, string password);
    void ftp_upload(char* file_data, int port);
    char* ftp_download(string url, int port);
    void make_tweet(string hashtag, string message);
    void make_facebook_post(string message);
    ...
};
```

Strana • 11



Example (2/4)

- Using a class for FTP download is not easy and is prone to errors:

```
HttpUtilities http;
http.set_credentials("sa", "SQL");
char* bytes = http.ftp_download("ftp://bla.com", 21);
string res = convert_bytes_to_text(bytes);
cout << res << endl;
```

Strana • 12



Example (3/4)

- However, all we need is to retrieve text files from FTP
- We create a new class to adapt the previous class to our needs:

```
class FtpTextDownloader {
private:
    HttpUtilities http;
public:
    string download_text(string username, string password,
                        string url, int port) {
        http.set_credentials(username, password);
        char* bytes = http.ftp_download(url, port);
        string res = convert_bytes_to_text(bytes);
        return res;
    }
};
```

Strana • 13



Example (4/4)

- Usage is greatly simplified:

```
FtpTextDownloader x;  
cout << x.download_text("sa", "SQL", "ftp://bla.com", 21);
```

Strana • 14



STACK

Strana • 15



Concrete stack

- In C++, the ADT stack is implemented as a generic class `stack<T>`
- Class `stack<T>` is a container adapter, which means:
 - It contains a second container that must implement at least the following methods:
 - `empty`
 - `size`
 - `back`
 - `push_back`
 - `pop_back`
 - `stack<T>` provides a stack interface, and in the background performs the operations on the contained container
 - `stack<T>` adapts other container to user needs

Strana • 16



Containers contained in `stack<T>`

- In order for a container to be contained in a `stack<T>` class, it must implement at least the required methods
- In standard C++, the following classes may contain containers:
 - `vector<T>`
 - `list<T>`
 - `deque<T>`
 - Our class that implements the required methods

Strana • 17



Problem

- Implement your stack of integers using a vector in the background. Enable the following:

```
int main() {
    MyStack s;
    s.push(11);
    s.push(22);
    s.push(33);

    while (!s.empty()) {
        cout << s.top() << endl;
        s.pop();
    }

    return 0;
}
```

Strana • 18



MyStack.h

```
class MyStack {
private:
    vector<int> v;
public:
    MyStack();
    void push(int val);
    bool empty();
    int& top();
    void pop();
};
```

Strana • 19



MyStack.cpp

```
MyStack::MyStack() {
}

void MyStack::push(int val) {
    v.push_back(val);
}

bool MyStack::empty() {
    return v.empty();
}

int& MyStack::top() {
    return v.back();
}

void MyStack::pop() {
    v.pop_back();
}
```

Strana • 20



Constructing and destroying stack

- There are three basic ways to construct a stack:
 - `stack<int> one;`
 - Creates an empty stack with the contained container of data type `deque<int>`
 - `stack<int, vector<int>> two;`
 - Creates an empty stack with the contained container of data type `vector<int>`
 - `vector<int> v({ 11, 22, 33 });`
`stack<int, vector<int>> three(v);`
 - Creates a stack with the contained container of data type `vector<int>`, containing all elements from vector `v`
- Instead of `vector`, we can use `list<T>`, `deque<T>` or our class that contains the required methods

Strana • 21



Stack basic operations

- `s.push(val)` puts a copy of *val* on top
- `s.emplace(arg1, arg2, ...)` creates a new object on the top
- `s.top()` returns a reference to the element at the top
- `s.pop()` removes and destroys the element at the top
- `s.empty()` returns if the stack is empty
- `s.size()` returns the number of elements on the stack
- The stack has no iterators
 - The only way to retrieve all the elements is to destroy the stack

Strana • 22



ADT QUEUE

Strana • 23



ADT queue

- ADT queue is a list or linked list with restrictions:
 - All element insertions are done at one end of the list, which is then called „the rear”
 - All element extractions are done at the other end of the list, which is then called „the front”
- Other names:
 - LILO (*Last In Last Out*) list
 - FIFO (*First In First Out*) list



Strana • 24



Operations

- The ADT queue usually defines the following operations:
 - Push/enqueue of the new element at the rear in $O(1)$
 - If no more place in the queue => *overflow*
 - Pop/dequeue of the element from the front in $O(1)$
 - If the queue is empty => *underflow*
 - Read the value of the element located directly at the front in $O(1)$
 - Retrieve the number of elements in a queue in $O(1)$
 - Check if the queue is empty in $O(1)$

Strana • 25



Queue applications

- Queues are most commonly used in multi-threaded programming
 - One or more threads (producers) insert elements to one side
 - One or more threads (consumers) remove the elements from the other side
 - Too complex for this course (OOP.NET, Java 2, IIS, ...)
- Single-threaded programming uses queues less often
 - The reason: the same thread is at the entrance and the exit
 - One application would be for the program to save some values and then use them later, with the guarantee that it will take them in the order in which it saved them

Strana • 26



QUEUE

Strana • 27



Concrete queue

- In C++, the ADT queue is implemented as a generic class `queue<T>`
 - In Java it is `Queue<T>`, in C# it is `Queue<T>`, ...
- Class `queue<T>` is a container adapter as well
 - The contained container must implement at least the following methods:
 - `empty`
 - `size`
 - `front`
 - `back`
 - `push_back`
 - `pop_front`

Strana • 28



`queue<T>` contained containers

- In order for a container to be contained in a `queue<T>` class, it must implement at least the required methods
- Following classes may be contained:
 - `list<T>`
 - `deque<T>`
 - Our class that implements the required methods
- Why is `vector<T>` a really bad choice for a contained container?

Strana • 29



Problem

- If we have `queue<T>` that uses `list<T>` as a contained container, describe how the following methods would work:
 - `empty`
 - `size`
 - `front`
 - `back`
 - `push`
 - `pop`

Strana • 30



Constructing and destroying queue

- There are three basic ways to create queue:
 - `queue<int> one;`
 - Creates an empty queue with the `deque<int>` contained container
 - `queue<int, list<int>> two;`
 - Creates an empty queue with the `list<int>` contained container
 - `list<int> l = { 11, 22, 33 };
queue<int, list<int>> three(l);`
 - Creates a queue with a copy of list `l` as the contained container
- `list<T>` can be replaced with `deque<T>` or our class that contains the required methods

Strana • 31



Basic operations with queue

- `q.push(val)` adds a copy of *val* to the rear
- `q.emplace(arg1, arg2, ...)` creates a new object at the rear
- `q.front()` returns a reference to the element at the front
- `q.pop()` removes and destroys the element at the front
- `q.back()` returns a reference to the element at the front
- `q.empty()` returns if the queue is empty
- `q.size()` returns the number of elements in a queue
- Queue has no iterators
 - The only way to retrieve all the elements is to destroy the contents of the queue

Strana 6



Algorithm visualization

- You can find operation visualizations for various structures and algorithms at:
 - www.cs.usfca.edu/~galles/visualization/Algorithms.html

Strana 33



Problem

1. Using the most appropriate container, copy the contents of one file to another file, but in reverse order.

Strana • 34



Solution

```
stack<string> s;

ifstream dat1("Source.cpp");
if (!dat1) {
    cout << "Error opening the file for reading" << endl;
    return 1;
}

string line;
while (getline(dat1, line)) {
    s.push(line);
}

dat1.close();
```

Strana • 35



Solution

```
ofstream dat2("Source_reversed.cpp");
if (!dat2) {
    cout << "Error opening the file for writing" << endl;
    return 1;
}

while (!s.empty()) {
    dat2 << s.top() << endl;
    s.pop();
}

dat2.close();
```

Strana • 36



Zadatak

2. Attached are three files, two invalid and one correct. Let's write a program that will check if a file is balanced in parentheses '{', '[' and '('. The algorithm is as follows:
 - a. Let's go through the file character by character.
 - b. If the character is an opening parenthesis, push it onto the stack.
 - c. If the character is a parentheses, remove the character from the stack and compare whether they are equal. If they are not, the file is not balanced.
 - d. If in the end there is something left on the stack, the file is not balanced.

Strana • 37



Solution

```

stack<char> s;
string line;
int no = 1;
while (getline(dat, line)) {
    for (unsigned i = 0; i < line.length(); i++) {
        if (line[i] == '{' || line[i] == '[' || line[i] == '(') {
            s.push(line[i]);
        }
        else if (line[i] == '}') {
            if (s.empty() || s.top() != '{') {
                cout << "Not balanced, line " << no << ": " << line;
                return 0;
            }
            s.pop();
        }
        ...
    }
}
if (s.empty()) cout << "Well balanced" << endl;
else cout << "Not balanced" << endl;

```



Do it yourself

1. Write a program that can check that a mathematical expression is correct in parentheses. For example, the following expression is correct:

$$(2 * (7 - 5)) / 3$$

While this one isn't:

$$(2 * (7 - 5)) / 3)$$

2. Read cplusplus.com/reference/stack/stack
3. Read cplusplus.com/reference/queue/queue

