CSCI 262
Data Structures

4 – Stacks and Queues

STACKS

“Last in, first out”

Stacks are a LIFO (Last in, first out) structure. Think of pancakes:

This pancake was put on top last.

Which one would you eat first?
Which would you eat second?

Three Operations

- **Top**: Look at the top item on the stack.
- **Push**: Add an item to the top of the stack.
- **Pop**: Remove the top item from the stack.

A Simple Stack Class

```cpp
class stack {
public:
    char top();
    void push(char c);
    void pop();
    size_t size();
    bool is_empty();

private:
    // private stuff
};
```

These operations are sometimes combined, e.g., `pop()` may return the top value on the stack as well as removing it from the stack.

Using Stacks

What does this code do?

```cpp
stack letters;
string text = "Data structures";
for (int j = 0; j < text.length(); j++) {
    letters.push(text[j]);
}
while (!letters.is_empty()) {
    cout << letters.top();
    letters.pop();
}
```
Applications

- Syntax analysis
  - Are parentheses, brackets, etc. balanced?
  - Nested structures (e.g., functions & variable scopes)
- Traversing/searching branching structures
  - Trees
  - Mazes
- Programming languages/processors
  - Forth, Postscript
  - Stack machines (e.g., Java virtual machine)

Balancing Game

Rules:
- To start, make an empty stack.
- If you see a \(, \}, or \], push it onto the stack
- If you see a \), \}, or \], try to pop the \textit{matching} delimiter from the stack, but:
  - If the stack is empty, yell “UNDERFLOW!”
  - If wrong character is at the top, yell “SYNTAX ERROR!”
- When the game ends, if your stack is empty, yell “I WIN!” else yell “SYNTAX ERROR!”

Balancing Game Inputs

- (easy)
- \([x];
- \{um}\-
- \{(a)\{(b)\}{(c)}
- \((x + y)*(m[a]){z})
- \((x + y)*{(m[a])}{z})

“The Stack”

When we talk about “the stack”, we usually mean a very specific stack; the memory stack of a running program:

```
#include <stack>
template <class ValueType>
class stack

Operations:
push(ValueType v)  // push value onto top of stack
pop()  // pop (remove) top value
top()  // return top value
size()  // return number of elements
empty()  // true if no elements
```

STL Stack

```
#include <stack>
template <class ValueType> class stack

Operations:
push(ValueType v)  // push value onto top of stack
pop()  // pop (remove) top value
top()  // return top value
size()  // return number of elements
empty()  // true if no elements
```
“First in, first out”

Queues are a FIFO (first in, first out) structure. Think of a line of people waiting their turn:

If people are polite, the first in line is done first.

Queue vs. Stack

Stack: All interactions are with the top of the stack.

Queue: items are added to the back (or rear) and taken from the front.

Operations

- Adding an item to a queue: enqueue*

- Removing an item from a queue: de-queue*

*A Simple Queue Class

```cpp
class queue {
public:
    char front();
    void enqueue(char c);
    void dequeue();
    size_t size();
    bool is_empty();
private:
    // private stuff
};
```

Using Queues

What does this code do?

```cpp
queue letters;
string text = "Data structures";
for (int j = 0; j < text.length(); j++) {
    letters.enqueue(text[j]);
}

while (!letters.is_empty()) {
    cout << letters.front();
    letters.dequeue();
}
```

Uses for Queues

Anywhere you need to keep things in order, particularly by time of arrival:

- Buffering character input
- Print jobs
- Process scheduling
- I/O request scheduling
- Web page request servicing
- Event handling (GUI, simulations, etc.)
# STL Queue

```cpp
#include <queue>

template <class ValueType>
class queue

Operations:
    push(ValueType v) // enqueue (add value to back)
    pop() // dequeue (remove front value)
    front() // return front value
    back() // return back value
    size() // return number of elements
    empty() // true if no elements
```

## Up Next

- **Friday, September 8**
  - Lab 3 – TBD
  - Nothing due (yay!)
- **Monday, September 11**
  - Sets and Maps
  - Read Sections 12.6 – 12.8