Overview

- Previously: linked lists
  - Easy addition/remove at ends
  - O(1) – constant time – operations:
    - Add to front
    - Add to end
    - Remove from front

- This time: stacks & queues
  - Only permit operations at ends
  - Easily implemented with linked lists
  - Despite simplicity, lots of applications

“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.

To think about:
- Note that both push and pop operate on the top of the stack; if you used a (singly) linked list implementation, which end of the list would hold the top element?
- What if you used a vector instead?
Using Stacks

What does this code do?

```
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)

Applications

- Balancing Game
  - Rules:
    1. To start, make an empty stack.
    2. If you see a, {, or [, push it onto the stack
    3. If you see a }, or ], try to pop the matching delimiter from the stack, but:
       a. If the stack is empty, yell "UNDERFLOW!"
       b. If wrong character is at the top, yell "SYNTAX ERROR!"
    4. 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:

```
function1
function2
function3
data
function4
```

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
```
QUEUES

“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 and taken from the front.

Operations

- Adding an item to a queue: enqueue*

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

*These are the modern names. You’ll find lots of implementations using “push” and “pop” instead, including the STL.

A Simple Queue Class

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

Graphs

In computer science (and math) graphs model relationships between things.

E.g., what cities are connected to each by a highway?

Finding Your Way

If I want to get from Ft. Collins to Golden, how can I find:
  - Some path
  - The shortest path
  - The path going through the fewest other cities
  - ...

Graph Algorithms

- Many powerful algorithms on graphs
  - Answer the above questions and more
  - Study these in CSCI 406 – Algorithms
- Two key algorithms for graph search:
  - Depth-first search – can use recursion OR stacks
  - Breadth-first search – easiest to use queues
Depth-First Search (DFS)
- Push Ft. Collins onto the stack
- While the stack is not empty:
  - Pop a city from the stack
  - If the city is Golden, done!
  - Otherwise, push all adjacent cities onto the stack
(Illustration on board)

Breadth-First Search (BFS)
- Same as DFS, but using a queue
- DFS goes as far as it can go until getting stuck, then backs up to most recent “intersection”
  - Lots of applications, mostly related to other graph algorithms/applications
- BFS goes to all nearest cities first, then the next nearest cities, etc.
  - Great for finding fewest hops
  - With some tweaks, can find shortest path

Up Next
- Today
  - Read Sections 12.1 – 12.3, 14.3
  - Lab 3 due
- Wednesday, January 30
  - Analysis of algorithms & Big O
  - Selection sort
- Friday, February 1
  - Lab 4 – TBD
  - Project 2 – Mazes assigned (graph search!)
  - APT 2 due