Data Structures and Algorithms

Discussion 4

Contents

- Stacks and Queues
- Recursion
- Priority Queues
- Function Objects
- Programming Problems

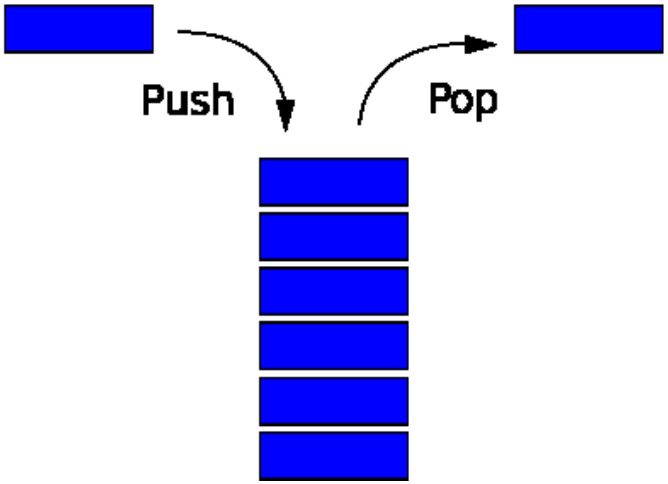
Stacks and Queues

Stack and Queue

- Containers for “inserting” and “removing”
  - Insert();
  - Remove();
  - Top(); or Front();
  - Size()
  - Empty();
- Included in STL
  - #include <stack>
  - #include <queue>

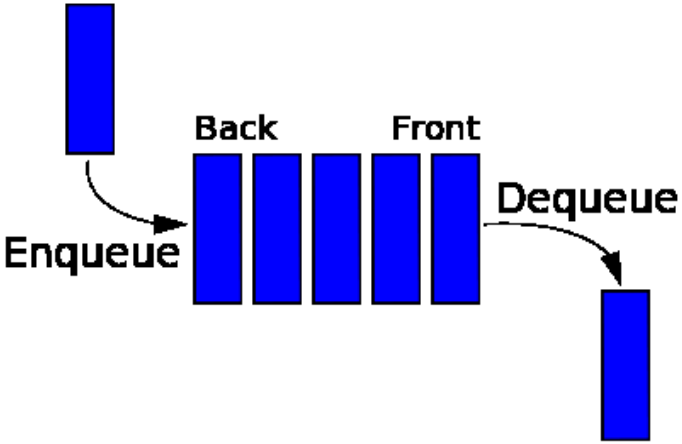
Stack

- Last-in First-out (LIFO)
- Member functions
  - push();
  - pop();
  - top();
  - size()
  - empty();



Queue

- First-in First-out (FIFO)
- Member functions
  - push();
  - pop();
  - front();
  - size()
  - empty();



POP QUIZ!

If you have an unsorted stack, how do you sort it using only one other stack (O(n) extra space)?

POP QUIZ!

If you have an unsorted stack, how do you sort it using only one other stack (O(n) extra space)?

```
//s1 is the original stack, s2 is a stack

while s1 is not empty:
  temp = s1.top(); s1.pop();
  while(s2 is not empty) AND (s2.top() > temp):
    s1.push(s2.top()); s2.pop();
  s2.push(temp)
```

Recursion

Recursion

- **Recursion** = solves a problem by solving smaller instances of the same problem
- All recursive functions can be expressed iteratively (and vice versa)

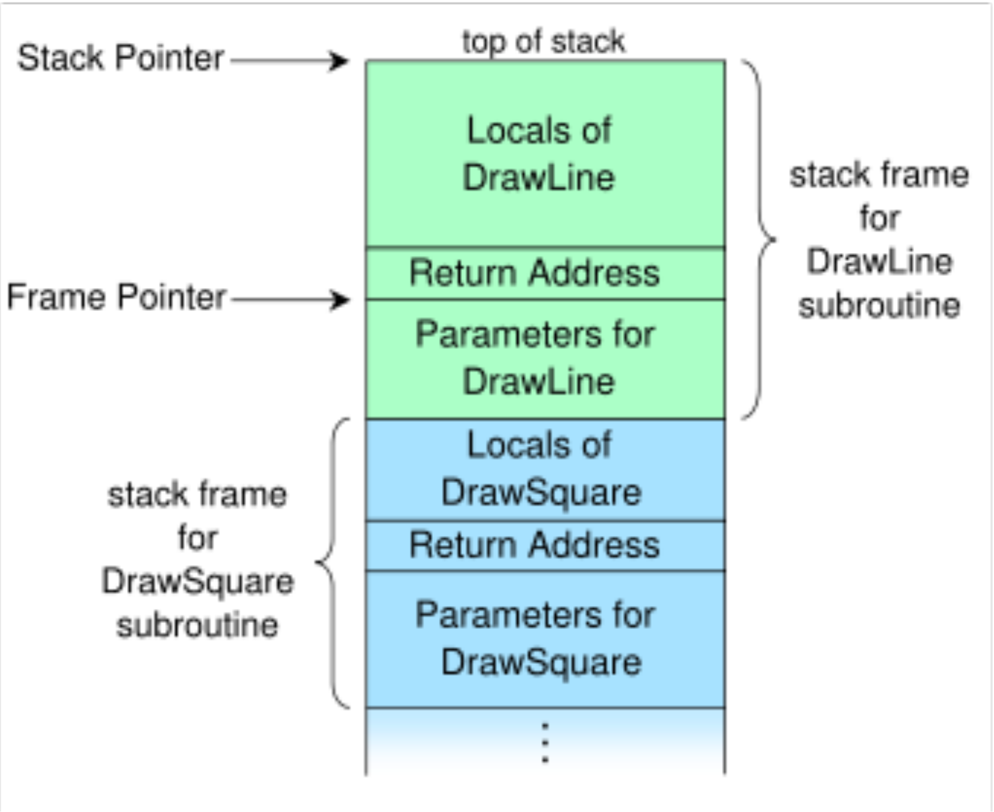
- Recursion advantages:
  - Can be a more intuitive way to solve the problem
  - Some data structures (like trees) are easier to traverse using recursion

Recursion - The Call Stack

- Pseudocode / Python:

```
def drawSquare(args):
    drawLine(args)

def drawLine(args):
    foo = 5
    bar = 3.14
    # do more stuff
```



Recursion

The function call stack is called that because it is **literally** a stack. Functions are logically evaluated in “LIFO” order.

Let's compare a recursive implementation of factorial to its equivalent with an explicit stack rather than the call stack...

Recursion - Factorial Function

```
int fact(int n){
  if (n >= 2){
    // recursive call pushes
    // another stack frame
    // which must store n
    return n * fact(n-1);
  }
  else {
    // base case
    return 1;
  }
}
```

```
int fact(int n){
  stack<int> s;
  // push all recursive calls
  while (n >= 2){
    // push n onto stack
    s.push(n--);
  }

  int f = 1; // base case
  while (!s.empty()){
    f *= s.top();
    s.pop();
  }
  return f;
}
```

Recursion - Factorial Function

Wait a minute…

- That implementation sucks.

---------->

- It takes O(n) space. Do we need that?

```
int fact(int n){
  stack<int> s;
  // push all recursive calls
  while (n >= 2){
    // push n onto stack
    s.push(n--);
  }

  int f = 1; // base case
  while (!s.empty()){
    f *= s.top();
    s.pop();
  }
  return f;
}
```

MichiganEngineering

Recursion - Factorial Function

Wait a minute…

- That implementation sucks.

---------->

- It takes O(n) space. Do we need that?

Better:

```
int fact(int n) {
   int f = 1; //base case
   while(n >= 2) {
      //accumulate n
      f *= n--;
   }
   return f; //done
```

```
int fact(int n){
   stack<int> s;
   // push all recursive calls
   while (n >= 2){
      // push n onto stack
      s.push(n--);
   }

   int f = 1; // base case
   while (!s.empty()){
      f *= s.top();
      s.pop();
   }
   return f;
}
```

MichiganEngineering

Recursion - Factorial Function

Why is this one better?

- Doesn't store `n` for each frame.
- It only needs O(1) space (no stack!)

```
int fact(int n) {
  int f = 1; //base case
  while(n >= 2) {
    //accumulate n
    f *= n--;
  }
  return f; //done
}
```

```
int fact(int n){
  stack<int> s;
  // push all recursive calls
  while (n >= 2){
    // push n onto stack
    s.push(n--);
  }

  int f = 1; // base case
  while (!s.empty()){
    f *= s.top();
    s.pop();
  }
  return f;
}
```

Recursion - Factorial Function

Why is this one better?

- Doesn't store `n` for each frame.
- It only needs O(1) space (no stack!)

```
int fact(int n) {
  int f = 1; //base case
  while(n >= 2) {
    //accumulate n
    f *= n--;
  }
  return f; //done
}
```

Connect to tail recursion…

- Compiler optimizes by turning recursion into a loop
- Can "reuse" the old stack frame.

```
int fact(int n){
  return fact_help(n,1);
}
int fact_help(int n, int f){
  if (n >= 2){
    // "accumulate" n
    return fact_help(n-1,
f*n);
  }
  else { return f; } // done
}
```

Recursion

Recall that some problems are hard (i.e. unnatural) to implement with tail recursion.

What's an example of such a problem?

What can we say about problems that need > O(1) space?

Recursion

Recall that some problems are hard (i.e. unnatural) to implement with tail recursion.

What's an example of such a problem?

- **Tree algorithms.**
- **How to combine results from multiple subtrees?**

What can we say about problems that need > O(1) space?

- **"Reusing" stack frames isn't a great trick anymore.**
- **Number of them we need will depend on input.**

Recursion

Speaking of trees…

- Review
  How do you (deep) copy a tree?
  (Will be helpful in P2!)

- Challenge (Interview question!)
  It turns out you can do tree traversal in O(1) **extra** space.
  How? (Hint: put some wasted space already in the tree to good use!)

Priority Queues

Priority Queue

- An abstract container type that supports two operations:

(1) Insert a new item
(2) Remove the item with the **largest key**

- Can have different implementations (binary heap, pairing heap, etc.)

STL Priority Queue

Implemented with a binary heap (You will implement this in project 2!)

`push`: Inserts an item into the priority queue
`pop`: Removes (without returning) the highest priority item
`top`: Returns (without removing) the highest priority item
`empty`: Returns true if the priority queue is empty
`size`: Returns the number of items in the priority queue

STL Priority Queue

These complexities are specific to the STL priority queue - not inherent to the priority queue itself, which is abstract

push	O(log N)
pop	O(log N)
top	O(1)
empty	O(1)
size	O(1)

STL Priority Queue

```
template <
     class T,
     class Container = vector<T>,
     class Compare = less<typename Container::value_type>
   > class priority_queue
```

Template arguments:

- The type of object stored in the priority queue
- The underlying container used (the default is usually fine)
- A function object (functor) type used to determine the priority between two objects
  - Default: `less<T>`

STL Priority Queue

```
template <
     class T,
     class Container = vector<T>,
     class Compare = less<typename Container::value_type>
   > class priority_queue
```

If you want to override the comparison functor, you must also set the container type used (can still use default vector<TYPE>).

GOOD: `priority_queue<int, vector<int>, greater<int>>`

BAD: `priority_queue<int, greater<int>>`

Function Objects (Functors)

Functors Review

- **Functors**: Objects that can be called as if they were ordinary functions
- Often used with STL containers and algorithms
- Made possible by overloading operator ()

```
class MyClass { … };
MyClass classObject;
classObject(); //looks like a function, but
               //really an object
```

MichiganEngineering

Functor Example

```
class People {
   int age;
public:
   int get_age() {return age;}
};

class PeopleCompare {
public:
   bool operator()(People& p1, People& p2) {
      return p1.get_age() < p2.get_age();
   }
};
```

Functor Example (2)

```
//assuming we already have two People
//objects to work with
PeopleCompare my_functor;
if(my_functor(person1, person2))
   cout << "Person 1 is the youngest\n";
else
   cout << "Person 2 is the youngest\n";
```

MichiganEngineering

Functor + Priority_queue

Priority queue of type People using the functor PeopleCompare:

```
std::priority_queue <People, std::
  vector<People>, PeopleCompare>
  people_queue;
```

Programming Problems

Problem 1

Using two stacks, how can you implement a queue?

What is the worst case complexity for each operation?

What is the amortized complexity for each operation?

Problem 2

Given two strings, how do you check if one is an anagram (permutation) of the other?

Problem 3

You have 20 bottles of pills

Bottle 1 has one pill, Bottle 2 has 2 pills, etc.

19 bottles have 1.0 gram pills

1 bottle has 1.1 gram pills

You are also given a scale that provides an exact measurement (can only use once)

How do you find the bottle that has the 1.1 gram pills?

Problem 3 - Solution

Since 1+2+3+...+19+20 = 210

Answer: (weight – 210) / 0.1

Example: if weight is 211.3, then bottle #13 is heavy bottle

Problem 4

How could you implement a double-ended priority queue (a queue where it is efficient to get **both** the minimum and the maximum values)?